Global EDF Schedulability Analysis for Synchronous Parallel Tasks on Multicore Platforms

Hoon Sung Chwa*, Jinkyu Lee†, Kieu-My Phan*, Arvind Easwaran‡, Insik Shin*
*Dept. of Computer Science, KAIST, South Korea
†Dept. of Electrical Engineering and Computer Science, The University of Michigan, U.S.A.
‡School of Computer Engineering, NTU, Singapore
insik.shin@cs.kaist.ac.kr

***Abstract*—The trend towards multi-core/many-core architectures is well underway. It is therefore becoming very important to develop software in ways that take advantage of such parallel architectures. This particularly entails a shift in programming paradigms towards fine-grained, thread-parallel computing. Many parallel programming models have been introduced targeting such intra-task thread-level parallelism. However, most successful results on traditional multi-core real-time scheduling are focused on sequential programming models. For example, thread-level parallelism is not properly captured into the concept of *interference*, which is key to many schedulability analysis techniques. Thereby, most interference-based analysis techniques are not directly applicable to parallel programming models. Motivated by this, we extend the notion of interference to capture thread-level parallelism more accurately. We then leverage the proposed notion of parallelism-aware interference to derive efficient EDF schedulability tests that are directly applicable to synchronous parallel task models on multi-core platforms. Our evaluation results indicate that the proposed analysis significantly advances the state-of-the-art in EDF schedulability analysis for synchronous parallel tasks.**

I. Introduction

An irreversible shift towards multi-core processors is underway. Recent development trends suggest that the chip industry is moving to multi-/many-core architectures to better manage trade-offs between performance, power efficiency, and reliability in deep submicron technology. For example, Intel designed a research chip with 80 cores [1], and Tilera introduced a range of processors with up to 100 cores [2]. Given the increasing emphasis on multi-/many-core chip design, software parallelism is likely to be one of the greatest constraints on computer performance. This inherently entails a shift in programming paradigms towards fine-grained thread-parallel computing, rather than relatively coarse-grained application-level parallelism.

A popular technique to achieve fine-grained, thread-level parallelism operates on the principle of *divide-and-conquer*. It breaks down a larger task into many smaller subtasks, runs those subtasks in parallel, and merges the results once each subtask completes computation. The synchronous parallel programming model embodies this kind of parallel decomposition. A synchronous parallel task consists of a sequence of parallel region, called *segments*, and each segment includes one or more threads. Two important aspects are associated with the synchronous parallel task model: *segment-level synchronization and thread-level parallelism*. All the threads belonging to the same segment are released at the same time and are not restricted to run simultaneously with at most m degree of parallelism, where m is the number of available cores. Segments are subject to synchronization rules between two consecutive segments. All the threads belonging to one segment must complete their own execution in order to move onward to the next segment. A good example is the fork-join programming model, which has been increasingly employed in many programming environments, such as Java [3], OpenMP [4], and Cilk [5].

A shift from uni-core to multi-core processors allows *inter-task parallelism*, where several applications (tasks) can execute simultaneously on multi-core processors. In addition, a shift from single-thread to multi-thread tasks allows *intra-task parallelism*, where even a single task can have multiple threads running simultaneously to take full advantage of multi-core processing. Despite the growing importance of intra-task parallelism, most real-time multiprocessor scheduling studies are focused on inter-task parallelism of sequential tasks, and relatively much less attention has been paid to understanding intra-task parallelism towards the schedulability analysis of synchronous parallel tasks. For example, a large number of studies extensively investigated the schedulability analysis of sequential tasks under EDF (earliest-deadline-first) [6] and fixed-priority global scheduling [6], producing many influential results. Such results include the concept of problem window and interference [7], [8], interference bounding techniques [9], [10], response time computation methods [8], [11], and optimal priority assignment [12]. Many other scheduling algorithms have been proposed for sequential models in order to take advantage of multiprocessors more effectively, including optimal algorithms such as pfair [13], DP-Fair [14], and RUN [15]. In addition, some approaches [16], [17] have been also proposed for scheduling tasks with pipeline precedence constraints in distributed real-time systems. However, the insights behind those successful results are not directly applicable to synchronous parallel tasks, due to the unique characteristics of thread-level parallelism.

For example, the notion of interference has been well defined in the sequential task case and serves as the basis for many schedulability analysis methods [7]–[9]. However, the current notion of interference does not capture thread-level parallelism because it assumes that each task has only a single thread to run at any time instant. Hence, those analysis methods are not directly applicable or easily extensible to synchronous parallel tasks.

Recently, a few analysis methods have been proposed for schedulability of synchronous parallel tasks [18]–[22].

Those methods can broadly fall into two categories: *direct* and *indirect*. Indirect analysis methods [19]–[21] share a principle of task decomposition. Each single synchronous parallel task is transformed into multiple independent sequential subtasks such that each individual subtask is assigned its own intermediate deadline. Schedulability analysis is then performed over intermediate deadlines after task decomposition at the expense of potentially incurring some non-trivial decomposition overheads. On the other hand, direct analysis methods [18], [22] perform analysis without task decomposition. A recent study [18] considers only a certain type of intra-task thread-level parallelism (i.e., *gang scheduling*), which allows a fixed degree of parallelism – all threads run or none does. A more recent study [22] considers a more general parallel task model, directed acyclic graph (DAG) model, for the single DAG task case, yet leaving unresolved the multiple synchronous parallel task case.

Motivated by this, the goal of this paper is direct schedulability analysis for synchronous parallel tasks. The rationale for this goal is that indirect EDF schedulability analysis through task decomposition involves the additional overhead of intermediate deadline assignment and a substantial amount of pessimism (discussed in Section VI).

Contribution. The main contribution of this paper is to introduce, to the authors' best knowledge, the first EDF schedulability condition that is directly applicable to *synchronous (malleable) parallel task* systems in which parallel tasks are allowed to execute with any arbitrary degree of thread-level parallelism up to the number of available processors. To this end, this paper extends the concept of interference capturing thread-level parallelism more accurately with novel notions of *critical interference* and *p-depth critical interference*. This paper provides evaluation results, showing that the proposed schedulability analysis significantly outperforms the state-of-the-art approaches available for synchronous parallel tasks.

II. Related Work

There has been an increasing attention to parallel task models in the context of real-time scheduling [18]–[28]. The work in [23], [24] considers soft real-time scheduling focusing on bounding tardiness upon deadline miss, while hard real-time systems aim at ensuring all deadlines are met. In this paper, we consider hard real-time scheduling.

In general, a parallel task is said to be (1) *moldable* if the task executes on exactly a certain number of processors, which is determined before execution and remains unchanged, or (2) *malleable* if the task can execute on any variable number of processors, which can be dynamically changing at runtime.

A recent study [18] considers *gang scheduling* [29] of moldable parallel tasks, in which a group of threads (e.g., all threads of the same task) run with a predetermined degree of parallelism or none do. This work introduces a new notion of interference (*interference rectangle*) to characterize inter-task interference according to such behavior of all-or-none thread execution. It then derives a schedulability analysis for global EDF gang scheduling. Since such a new interference notion does not capture the situation where any arbitrary number of threads execute in parallel, however, it is difficult to apply the notion to the malleable parallel task case.

More recent studies [19]–[22] consider malleable parallel task models, in which threads can execute any arbitrary degree of parallelism up to the number of available processors. One of the widely used malleable parallel task models is the *fork-join* model [19]. A fork-join task consists of a sequence of segments such that every odd-numbered segment contains a single (master) thread and every even-numbered segment consists of multiple (worker) threads. The master thread that runs sequentially forks off a number of worker threads which execute blocks of code in parallel. After the execution of the parallelized code, the worker threads join back into the master thread, which continues onward to another parallel region or the end of the program. Relaxing the requirement of such parallel/non-parallel alternation of the fork-join model, a more general synchronous parallel task model [20], [21] is considered such that each segment can have any number of threads. In this paper, we consider this task model.

A few studies [19]–[21] share a common principle of task decomposition for schedulability analysis. They decompose a single synchronous parallel task into multiple independent sequential sub-tasks through intermediate deadline assignment. This approach is safe — satisfying the intermediate deadlines of all sub-tasks leads to meeting the deadlines of their aggregate synchronous parallel tasks. They then employ existing schedulability analysis for those sequential sub-tasks. More specifically, Lakshmanan, *et al.* [19] propose a partitioned preemptive fixed-priority scheduling algorithm with a provable performance for fork-join tasks, under the assumption that all parallel segments have the same number of parallel threads. Saifullah, *et al.* [20] decompose a parallel task into a set of sequential sub-tasks such that the density of each segment is upper bounded by some value, and this bound is used to derive a resource augmentation bound. Nelissen, *et al.* [21] decompose a parallel task such that the maximum density among all segments in a parallel task is minimized. However, such an indirect analysis via task decomposition can be pessimistic, because task decomposition can incur non-trivial overheads (discussed in Section VI). Furthermore, it requires modifications to existing operating systems to support task decomposition. Thereby, this paper seeks to derive direct schedulability analysis for synchronous parallel tasks.

Another recent study [22] considers the sporadic DAG model that is a general form of the synchronous parallel task model. In the DAG model, each vertex corresponds to a single thread and each edge represents a precedence constraint. A thread can execute only after all of its predecessors have been executed. This work presents a new notion of demand and load for the sporadic DAG task model and applies it to derive an efficient EDF schedulability test for the single sporadic DAG task with a focus on the arbitrary deadline case. However, this work does not present schedulability analysis for a case in which multiple DAG tasks share processors, leaving it as future work. On the other hand, our work considers scheduling of multiple synchronous parallel tasks and presents EDF schedulability tests for the case.

III. System Model

We consider a multi-core platform, where sporadic, synchronous parallel tasks run over m identical processors under global EDF scheduling. A synchronous parallel task consists

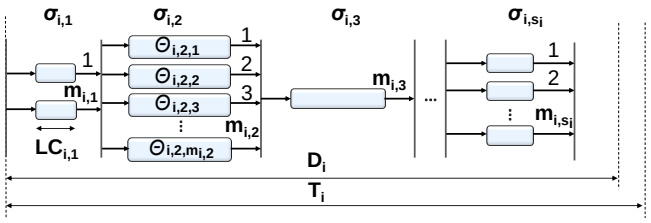

Fig. 1. A synchronous parallel task τ_i

of a series of segments, where each segment contains one or more synchronous threads. A synchronous parallel task has two major properties in terms of execution mechanism: *segment-level synchronization* and *thread-level parallel execution*. All threads within a single segment are released simultaneously, are able to execute in parallel, and must finish their execution prior to proceeding to the next segment. Thereby, no segments within a single task can overlap each other.

A set of tasks is denoted by τ. For a sporadic, synchronous parallel task τ_i, T_i is the minimum separation, D_i is the relative deadline, s_i is the number of segments, and $\sigma_{i,j}$ represents the j-th segment (see Figure 1). Each segment $\sigma_{i,j}$ has its own internal threads $\theta_{i,j,k}$, and $m_{i,j}$ represents the number of threads in the segment. The maximum parallelism of a task τ_i (denoted with m_i) is then defined as the maximum value of $m_{i,j}$ among all segments, i.e., $m_i = \max_j m_{i,j}$. In this paper, there is no restriction on $m_{i,j}$; it can be larger than m, the number of processors.

For an individual thread $\theta_{i,j,k}$, $C_{i,j,k}$ is its worst-case execution time requirement (WCET). A segment $\sigma_{i,j}$ is associated with two WCET parameters: $C_{i,j}$ and $LC_{i,j}$. $C_{i,j}$ is defined as the maximum amount of time to complete the execution of all the threads within the segment $\sigma_{i,j}$ on a single core. On the other hand, $LC_{i,j}$ is defined as the minimum amount of time needed to execute all threads in $\sigma_{i,j}$ assuming that it can use as many processors as possible for its execution. That is,

$$C_{i,j} = \sum_{k=1}^{m_{i,j}} C_{i,j,k} \quad \text{and} \quad LC_{i,j} = \max_{1 \le k \le m_{i,j}} \{C_{i,j,k}\}. \tag{1}$$

Similarly, C_i and LC_i represent the maximum and minimum execution times of a task τ_i, respectively. We then define them as

$$C_i = \sum_{j=1}^{s_i} C_{i,j} \quad \text{and} \quad LC_i = \sum_{j=1}^{s_i} LC_{i,j}. \tag{2}$$

A sporadic, synchronous parallel task τ_i invokes a series of jobs, each separated from its predecessor by a minimum of T_i time units. We consider a constrained deadline D_i such that $D_i \le T_i$. It should be $LC_i \le D_i$ but not necessarily $C_i \le D_i$. Let U_i denote the utilization of τ_i and be defined as $U_i = C_i/T_i$.

We denote the k-th job of a task τ_i with J_i^k. We will omit the superscript in the notation for simplicity when no ambiguity arises. For a job J_i^k, r_i^k and d_i^k are its release time and deadline. The *execution window* of a job J_i^k is then defined as interval $[r_i^k, d_i^k)$. In this paper, every single job J_i is assigned its own priority under EDF scheduling. Thereby, different jobs have different priorities, but all threads within a single job have the same priority, breaking ties arbitrarily.

In this paper, we assume quantum-based time and without loss of generality, let one time unit denote the quantum length. All task parameters are assumed to be specified as multiples of this quantum length.

IV. Interference-based Schedulability Analysis for Synchronous Parallel Tasks

In this section, we derive schedulability analysis of global EDF scheduling for sporadic synchronous parallel task systems with constrained deadlines. In the real-time scheduling literature, the notion of interference has been employed in many schedulability analysis methods [8], [9], [30]–[32], using the following definitions:

- Interference $I_k(a, b)$: the sum of all intervals in which τ_k is ready for execution but cannot execute due to other higher-priority tasks in $[a, b)$.
- Interference $I_{i,k}(a, b)$: the sum of all intervals in which τ_i is executing and τ_k is ready to execute but not executing in $[a, b)$.

With the above definitions, the relation between $I_k(a, b)$ and $I_{i,k}(a, b)$ serves as an important basis for deriving schedulability analysis. In the single-thread task case, it is intuitive to construct such a relation on m processors as follows [8]:

$$I_k(a, b) = \frac{1}{m} \sum_{\forall \tau_i \in \tau} I_{i,k}(a, b). \tag{3}$$

However, it is not straightforward to build such a relation in the multi-thread task case, as illustrated in the following example.

Example 4.1: As an example, suppose that two threads of higher-priority τ_i and one thread of lower-priority τ_k are ready for execution on two processors at time t. Then, the two threads of τ_i will run on two processors in $[t, t+1)$, delaying the execution of τ_k. According to the above definitions, τ_i imposes interference on τ_k in $[t, t+1)$, yielding $I_k(t, t+1) = 1$ and $I_{i,k}(t, t+1) = 1$. However, Eq. (3) no longer supports such definitions.

The above example suggests a need for extending the concept of interference for the parallel task model, and this raises three problems: (1) how to calculate the interference on τ_k when only some (but not all) threads of τ_k are interfered, (2) how to calculate the interference of τ_i on τ_k when only some (but not all) threads of τ_i interfere with τ_k, and (3) how to calculate the interference of threads of task τ_k on other threads of the same task τ_k.

To address problem (1), we introduce a new concept called *critical interference*. A thread is said to be a *critical* thread if it finishes last among all the threads belonging to the same segment. With the notion of critical threads, we can now extend the traditional definition of interference towards the synchronous parallel task model as follows:

- Critical interference $\mathcal{I}_k(a, b)$: the sum of all intervals in which ***a critical thread of*** τ_k is ready for execution

but cannot execute due to other higher-priority ***threads*** in $[a, b)$.

- Critical interference $\mathcal{I}_{i,k}(a, b)$: the sum of all intervals in which ***at least one thread of*** τ_i is executing and ***the critical thread of*** τ_k is ready to execute but not executing in $[a, b)$.

Note that when all tasks have a single thread, then the single thread is equal to the critical thread and our definition is the same as the traditional definition of interference.

To address problem (2), we introduce a new concept called *p-depth critical interference*. The p-depth critical interference of a task τ_i on τ_k characterizes not only the length of the delay τ_i causes to τ_k but also the number of threads of τ_i that cause the delay.

To address problem (3), we incorporate the notion of intra-task interference into both the critical interference and the p-depth critical interference such that they include interference on a critical thread by other non-critical threads of the same task.

A. Interference-based Schedulability Analysis

We first seek to identify a necessary condition for any synchronous parallel task to miss a deadline on m processors. In synchronous parallel tasks, all the threads belonging to the same segment $\sigma_{k,u}$ are released at the same time, but they can complete execution at different time instants depending on their execution behavior. A thread of a segment $\sigma_{k,u}$ is said to be a *critical* thread (denoted as θ_{k,u,v^*}) if it finishes last among all the threads of the segment. A segment is then considered as *complete* as soon as its critical thread completes execution. We define interference on a critical thread θ_{k,u,v^*} over interval $[a, b)$ (denoted as $\mathcal{I}_{k,u,v^*}(a, b)$) as the cumulative length of all intervals in which the critical thread is ready to execute but not executing due to the execution of higher-priority threads belonging to other tasks as well as belonging to the same task. To avoid any confusion, it is worth noting that $\mathcal{I}_{k,u,v^*}(a, b)$ includes intra-task interference that a critical thread θ_{k,u,v^*} receives from other threads $\theta_{k,u,x}$ of the same task τ_k. According to our definition, $\mathcal{I}_k(a, b)$ is a total interference imposed collectively on all the critical threads of τ_k, i.e.,

$$\mathcal{I}_k(a, b) = \sum_{\forall \sigma_{k,u} \in \tau_k} \mathcal{I}_{k,u,v^*}(a, b). \tag{4}$$

We let J_k^* denote the job instance of a synchronous parallel task τ_k that receives the maximum critical interference among all the jobs of τ_k. For J_k^*, r_k^* and d_k^* are its release time and deadline. Suppose J_k^* missed a deadline. Then, one can see that at least one critical thread of J_k^* must not execute for C_{k,u,v^*} time units, and all critical threads do not execute for LC_k time units in total. On the other hand, when J_k^* receives the amount of interference smaller than or equal to $D_k - LC_k$, every job of τ_k has sufficient time to complete the execution of all critical threads prior to a deadline under any work-conserving scheduling. This observation yields the following lemma.

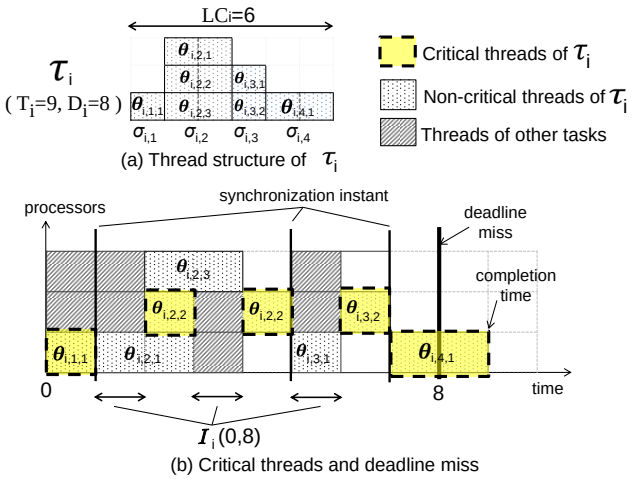

Fig. 2. An example of synchronous parallel task τ_i on 3 processors. (a) Task τ_i has 4 segments with $T_i = 9$, $D_i = 8$, and $LC_i = 6$. (b) Task τ_i misses a deadline at 8. Here, critical threads are $\theta_{i,1,1}$, $\theta_{i,2,2}$, $\theta_{i,3,2}$, and $\theta_{i,4,1}$, because they finish last among all threads in the same segment. Two critical threads, $\theta_{i,2,2}$ and $\theta_{i,3,2}$, were blocked for 3 time units (i.e., [1,2), [3,4), and [5,6)), yielding $\mathcal{I}_i(0, 8) > D_i - LC_i$.

Lemma 1: A set of synchronous parallel tasks (denoted by τ) is schedulable on m processors if

$$\forall \tau_k \in \tau, \quad \mathcal{I}_k(r_k^*, d_k^*) \leq D_k - LC_k. \tag{5}$$

Figure 2 shows an example that we will consider throughout this paper. Figure 2(a) shows the thread structure of task τ_i and its parameters. In Figure 2(b), a job of τ_i is released at 0 with a deadline of 8. The figure shows that the execution of two critical threads, $\theta_{i,2,2}$ and $\theta_{i,3,2}$, were delayed for 3 time units collectively, resulting in $\mathcal{I}_i(0, 8) > D_i - LC_i$. This makes it infeasible for the last thread $\theta_{i,4,1}$ to fully execute for $C_{i,4,1}$ time units before the deadline of 8. This leads to the deadline miss of task τ_i.

As shown in Example 4.1, it is not as straightforward as Eq. (3) to build the relation between $\mathcal{I}_k(a, b)$ and $\mathcal{I}_{i,k}(a, b)$. This is mainly because $\mathcal{I}_{i,k}(a, b)$ does not capture how many threads of τ_i interfere with the critical threads of τ_k. We thereby introduce a new concept of p-depth critical interference that characterizes the number of interfering threads, and this new notion will bridge $\mathcal{I}_k(a, b)$ and $\mathcal{I}_{i,k}(a, b)$ effectively for synchronous parallel tasks. Let us define the *p-depth critical interference* $\mathcal{I}_{i,k}(p, a, b)$ of task τ_i on task τ_k during interval $[a, b)$ as the cumulative length of all intervals in which (1) a critical thread of τ_k is ready to execute but does not execute and (2) exactly p number of threads of τ_i are executing (see Figure 3). It is worth noting that when it comes to the intra-task interference case, $\mathcal{I}_{k,k}(p, a, b)$ corresponds to a case where a critical thread of τ_k is not executing while exactly p number of other non-critical threads of τ_k are executing. The p-depth critical interference enables to represent the behavior of parallel execution in more detail, allowing to figure out exactly how many threads of a task τ_i are executing simultaneously when τ_i delays the execution of another task τ_k. A total critical interference $\mathcal{I}_{i,k}(a, b)$ can be decomposed into individual p-depth critical interferences as follows:

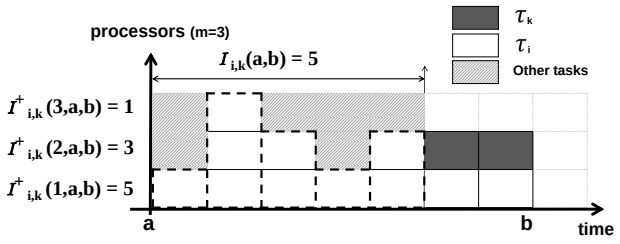

Fig. 3. The notion of p-depth critical interference and at least p-depth critical interference. Suppose that task τ_k has a lower priority than task τ_i. A job of τ_k is released at time instant a, but it cannot execute in $[a, a+5)$ due to the execution of other higher priority tasks. In this example, task τ_i executes a single thread in intervals $[a, a+1)$ and $[a+3, a+4)$, which corresponds to 1-depth critical interference on τ_k. This yields $\mathcal{I}_{i,k}(1, a, b) = 2$. Task τ_i executes two threads in intervals $[a+2, a+3)$ and $[a+4, a+5)$, leading to $\mathcal{I}_{i,k}(2, a, b) = 2$. And $\mathcal{I}_{i,k}(3, a, b) = 1$.

$$\mathcal{I}_{i,k}(a,b) = \sum_{p=1}^{m} \mathcal{I}_{i,k}(p,a,b). \tag{6}$$

The p-depth critical interference also makes it easy to constitute a total interference $\mathcal{I}_k(a,b)$ out of individual interferences of each task on task τ_k on m processors as follows.

Lemma 2: For any work-conserving algorithm, the total critical interference $\mathcal{I}_k(a,b)$ imposed on task τ_k in interval $[a,b)$ is equal to the total amount of contribution of individual threads to the interference on θ_{k,u,v^*} divided by the number of processors, i.e.,

$$\mathcal{I}_k(a,b) = \frac{1}{m} \sum_{\forall \tau_i \in \tau} \sum_{p=1}^{m} \mathcal{I}_{i,k}(p,a,b) \times p. \tag{7}$$

Proof: Since the scheduling algorithm is work-conserving, in the time instants in each of which a critical thread of a task is ready but not executing, each processor must be occupied by all the other threads of another task and including itself. The total amount of the contribution to the critical interference on τ_k is $\sum_{\forall \tau_i \in \tau} \sum_{p=1}^{m} \mathcal{I}_{i,k}(p,a,b) \times p$. If it is divided by the number of processors, we can get the length of cumulative intervals in which a critical thread of τ_k is ready to execute but cannot in an interval $[a,b)$. ■

For notational convenience, we define *at least p-depth critical interference* $\mathcal{I}^+_{i,k}(p,a,b)$ as the sum of intervals in which *at least* p number of threads of τ_i execute simultaneously delaying the execution of the critical thread of τ_k in $[a,b)$. By definition, we have

$$\mathcal{I}^+_{i,k}(p,a,b) = \sum_{q=p}^{m} \mathcal{I}_{i,k}(q,a,b). \tag{8}$$

In the example shown in Figure 3, at least 2-depth critical interference of τ_i on τ_k is calculated as the sum of all intervals in which two or more threads of τ_i are executing imposing interference on τ_k. Then, $\mathcal{I}^+_{i,k}(2,a,b) = \mathcal{I}_{i,k}(2,a,b) + \mathcal{I}_{i,k}(3,a,b) = 3$. We note that $\mathcal{I}_{i,k}(a,b)$ is equal to $\mathcal{I}^+_{i,k}(1,a,b)$ by definition.

Building upon the notion of at least p-depth critical interference, the following theorem derives a schedulability condition. Note that it is possible to upper bound the value of $\mathcal{I}^+_{i,k}(p, r^*_k, d^*_k)$ by $D_k - LC_k$ for all $1 \le p \le m$ in the schedulability analysis of τ_k. This serves as a basis for workload bound techniques, which will be shown in the next section.

Theorem 1: A task set τ is schedulable under any work-conserving algorithm on m identical processors if for each task $\tau_k \in \tau$,

$$\sum_{\forall \tau_i \in \tau} \sum_{p=1}^{m} \min(\mathcal{I}^+_{i,k}(p, r^*_k, d^*_k), D_k - LC_k) < m(D_k - LC_k). \tag{9}$$

Proof: The proof is by contraposition. We wish to prove that if a task set τ is not schedulable under any work-conserving algorithm, $\exists \tau_k \in \tau, \sum_{\forall \tau_i \in \tau} \sum_{p=1}^{m} \min(\mathcal{I}^+_{i,k}(p, r^*_k, d^*_k), D_k - LC_k) \ge m(D_k - LC_k)$. Suppose task τ_k misses a deadline. Then, by Lemma 1,

$$\mathcal{I}_k(r^*_k, d^*_k) > D_k - LC_k. \tag{10}$$

We note that, by Lemma 2 and Eq. (8),

$$\begin{aligned} \mathcal{I}_k(r^*_k, d^*_k) &= \frac{1}{m} \sum_{\forall \tau_i \in \tau} \sum_{p=1}^{m} \mathcal{I}_{i,k}(p, r^*_k, d^*_k) \times p \\ &= \frac{1}{m} \sum_{\forall \tau_i \in \tau} \sum_{p=1}^{m} \mathcal{I}^+_{i,k}(p, r^*_k, d^*_k). \end{aligned} \tag{11}$$

Let α denote $\{(i,p) \mid \forall \tau_i \in \tau, 1 \le p \le m,\ \mathcal{I}^+_{i,k}(p, r^*_k, d^*_k) \ge D_k - LC_k\}$, and $|\alpha|$ denotes the number of elements in α.

If $|\alpha| \le m$,

$$\begin{aligned} &\sum_{\forall \tau_i \in \tau} \sum_{p=1}^{m} \min(\mathcal{I}^+_{i,k}(p, r^*_k, d^*_k), D_k - LC_k) \\ &= |\alpha| \cdot (D_k - LC_k) + \sum_{\forall (i,p) \notin \alpha} \mathcal{I}^+_{i,k}(p, r^*_k, d^*_k) \\ &= |\alpha| \cdot (D_k - LC_k) + m \cdot \mathcal{I}_k(r^*_k, d^*_k) - \sum_{\forall (i,p) \in \alpha} \mathcal{I}^+_{i,k}(p, r^*_k, d^*_k) \\ &\qquad (\because \text{ by } Eq.(11)) \\ &\ge |\alpha| \cdot (D_k - LC_k) + m \cdot \mathcal{I}_k(r^*_k, d^*_k) - |\alpha| \cdot \mathcal{I}_k(r^*_k, d^*_k) \\ &\qquad (\because \text{ by } \mathcal{I}^+_{i,k}(p, r^*_k, d^*_k) \le \mathcal{I}_k(r^*_k, d^*_k)) \\ &= |\alpha| \cdot (D_k - LC_k) + (m - |\alpha|) \cdot \mathcal{I}_k(r^*_k, d^*_k) \\ &> |\alpha| \cdot (D_k - LC_k) + (m - |\alpha|) \cdot (D_k - LC_k) \quad (\because \text{ by } Eq.(10)) \\ &= m \cdot (D_k - LC_k). \end{aligned}$$

Otherwise, $|\alpha| > m$,

$$\begin{aligned} &\sum_{\forall \tau_i \in \tau} \sum_{p=1}^{m} \min(\mathcal{I}^+_{i,k}(p, r^*_k, d^*_k), D_k - LC_k) \\ &\ge |\alpha| \cdot (D_k - LC_k) > m \cdot (D_k - LC_k). \end{aligned}$$

■

V. Workload-based Schedulability Test

Note that Theorem 1 includes the interference terms of $\mathcal{I}^+_{i,k}(p, r^*_k, d^*_k)$, but it is generally difficult to calculate those values precisely. Most existing approaches [8], [10], [31]–[33] instead seek to derive upper bounds on interference based on workload: the workload $W_i(a,b)$ of τ_i is the sum of all intervals in which τ_i is executing in interval $[a,b)$. This section derives a schedulability test using a workload-based interference bound. To this end, we seek to derive a bound on the whole sum of such individual bounded interferences, i.e., $\sum_{p=1}^{m} \min(\mathcal{I}^+_{i,k}(p, r^*_k, d^*_k), D_k - LC_k)$.

Let us first restrict our discussion to a case where the number of threads in each segment $\sigma_{i,j}$ is smaller than or equal to the number of processors ($m_i \leq m$), and we will relax this restriction later.

A. The number of threads in any segment is not larger than the number of processors ($m \geq m_i$) for all i.

Along with the notion of p-depth critical interference, we introduce the notion of *p-depth workload*. The p-depth workload $W_{i,k}(p,a,b)$ of task τ_i is the sum of all intervals in which exactly p number of threads of τ_i are executing in a way that those p number of threads are all of higher priority than that of a critical thread of τ_k in interval $[a,b)$. Similarly to $\mathcal{I}^+_{i,k}(p,a,b)$, we define the notion of *at least p-depth workload* $W^+_{i,k}(p,a,b)$ as follows:

$$W^+_{i,k}(p,a,b) = \sum_{q=p}^{m} W_{i,k}(q,a,b). \tag{12}$$

By definition, we have that $\mathcal{I}_{i,k}(p,a,b) \leq W_{i,k}(p,a,b)$ and that $\mathcal{I}^+_{i,k}(p,a,b) \leq W^+_{i,k}(p,a,b)$. Thus, the following inequality hold for any J_k of τ_k:

$$\begin{aligned} &\sum_{p=1}^{m} \min(\mathcal{I}^+_{i,k}(p, r_k, d_k), D_k - LC_k) \\ \leq &\sum_{p=1}^{m} \min(W^+_{i,k}(p, r_k, d_k), D_k - LC_k) \stackrel{\text{def.}}{=} \widehat{W^+_{i,k}}. \end{aligned} \tag{13}$$

According to Inequality (13), an upper-bound on the interference term can be obtained by finding the maximum possible value of $\widehat{W^+_{i,k}}$ in any scheduling window of a job of τ_k. For the traditional single-thread task model, it requires us to identify the worst-case release pattern of τ_i. For this multi-thread task model, in addition to the *worst-case release pattern*, we also need to identify the *worst-case execution pattern in a segment*, which characterizes the execution of parallel threads in the same segment. Note that the equation of $\widehat{W^+_{i,k}}$ contains the *min* operation (see Inequality (13)), which does not allow $W^+_{i,k}(p, r_k, d_k)$ to contribute to $\widehat{W^+_{i,k}}$ any greater than $D_k - LC_k$. The worst-case release and execution patterns thus should maximize $\widehat{W^+_{i,k}}$ in the presence of the $\min$ operation.

We next consider two cases to discuss worst-case release and execution patterns for maximizing $\widehat{W^+_{i,k}}$: inter-task ($i \neq k$) and intra-task cases ($i = k$).

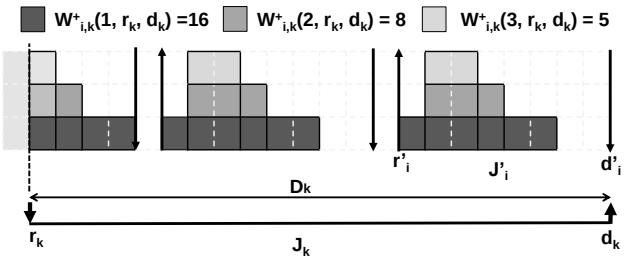

Fig. 4. The worst-case release pattern in which $\widehat{W^+_{i,k}}$ is maximized.

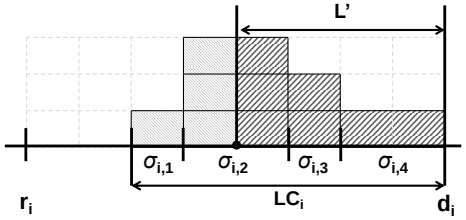

Fig. 5. The worst-case situation in which the carry-in is maximized

1) Inter-Task Workload: To simplify the presentation, we use the following terms. A job is said to be a *carry-in* job of an interval $[a,b)$ if it is released before a but has a deadline within $[a,b)$, or a *body* job if its release time and deadline are both within $[a,b)$.

Worst-case release pattern. Under a given execution pattern, we can now determine the worst-case release pattern. Figure 4 shows the worst-case release pattern, in which task τ_i has the maximum amount of $\widehat{W^+_{i,k}}$ that interferes with job J_k over interval $[r_k, d_k)$ under EDF scheduling. As shown in the figure, all the jobs of τ_i are released periodically, and its last body job (J'_i) of the interval $[r_k, d_k)$ has a deadline equal to that of J_k (i.e., $d'_i = d_k$). For the carry-in job, we consider the worst-case situation in which all threads of the carry-in job are executed as late as possible (see Figure 5). With this release pattern we can include the largest number of segments of τ_i having higher priority than J_k in the interval $[r_k, d_k)$, thus maximizing the value of $\widehat{W^+_{i,k}}$. We recapitulate the above result in the following lemma:

Lemma 3: Under a given execution pattern, the release pattern of task τ_i that maximizes $\widehat{W^+_{i,k}}$ is: (1) jobs are minimally separated, (2) a deadline of a job of τ_i aligns with the deadline of the job of τ_k, and (2) all threads of the carry-in jobs are executed as late as possible (right before the deadline).

Worst-case execution pattern. If at all time instants, task τ_i executes all of its available threads (that are released and do not finish), then τ_i is said to *execute with the maximum degree of parallelism*. Note that since the number of processors is larger than or equal to the number of threads in a segment, there are enough processors to execute all released threads of τ_i simultaneously.

Lemma 4: For a given release pattern, when task τ_i executes with the maximum degree of parallelism, $\widehat{W^+_{i,k}}$ is maximized.

Proof: Note that by the assumption $m \geq m_i$ for all task

τ_i, it is possible for τ_i to execute with the maximum degree of parallelism. The proof is constructed by induction on the maximum degree of parallelism m_i of task τ_i:

Base case: m_i=1, the hypothesis is trivially true.

Now suppose the hypothesis is true for all tasks τ_j with the maximum degree of parallelism $m_j = x$. We will prove that the hypothesis is true for all tasks τ_i with the maximum degree of parallelism $m_i = x+1$. First we construct a task τ_i' by deleting one thread from every segment of τ_i. Then τ_i' has $m_i' = x$. Thus the hypothesis is true for τ_i'.

Now we add one thread to every segment of τ_i'. The extra workload caused by the additional threads can contribute to some $W^+_{i,k}(p, r_k, d_k)$. In the presence of the min operation, the contribution of this extra workload to $\widehat{W^+_{i,k}}$ can be maximized when it is added to as many smallest $W^+_{i,k}(p, r_k, d_k)$s as possible (i). By the definition of $W^+_{i,k}(p, r_k, d_k)$, $W^+_{i,k}(p, r_k, d_k) \leq W^+_{i,k}(q, r_k, d_k)$ when $p > q$ (ii). From (ii), (i) can be satisfied when all $x+1$ threads are executed with the maximum degree of parallelism. Thus the hypothesis is satisfied for $m_i = x+1$. The lemma is proven. ■

Note that in the worst-case execution pattern defined above, all segment $\sigma_{i,j}$ execute only for $LC_{i,j}$.

Calculating worst-case workload. By using the worst-case execution pattern and the worst-case release pattern, we can now determine the interval of length D_k which maximizes $\widehat{W^+_{i,k}}$. Define $W^*_{i,k}(p, D_k)$ as the value of $W^+_{i,k}(p, r_k, d_k)$ in such an interval. Figure 5 shows how to compute the maximum amount of carry-in workload. Let us denote by L' the length of a carry-in interval. In the figure, when $L' = 4$, the last three segments of $\sigma_{i,2}, \sigma_{i,3}$, and $\sigma_{i,4}$ are considered as carry-in workloads, since their execution can fully/partially fit into the carry-in interval. If $\sigma_{i,3}$ executes only for a duration smaller than $C_{i,3}$, then in order to guarantee the schedulability for task τ_i in the worst case, $\sigma_{i,2}$ still needs to finish before $d_i - (LC_{i,3} + LC_{i,4})$, leaving enough room for $\sigma_{i,3}$ to execute for $C_{i,3}$. Thus we can consider all threads to execute for their WCET to calculate the carry-in jobs.

Let us consider a bound $BD^+_{i,k}(p, D_k)$ on the *at least p-depth body-job workload* in any interval of length D_k and another bound $CI^+_{i,k}(p, L')$ on the *at least p-depth carry-in workload* in any carry-in interval of length L'. The maximum number of body jobs of τ_i over an interval of length D_k is $\left\lfloor \frac{D_k}{T_i} \right\rfloor$. Then, $BD^+_{i,k}(p, D_k)$ and $CI^+_{i,k}(p, L')$ are calculated as follows:

$$BD^+_{i,k}(p, D_k) = \left\lfloor \frac{D_k}{T_i} \right\rfloor \cdot \sum_{\forall j : m_{i,j} \geq p} LC_{i,j}, \tag{14}$$

$$CI^+_{i,k}(p, L') \stackrel{\text{def.}}{=} \begin{cases} 0, & \text{if } L' \leq 0, \\ \sum_{j=h:m_{i,j}\geq p}^{s_i} LC_{i,j}, & \text{else if } 0 < L' \leq LC_i \text{ and} \\ & \quad m_{i,h-1} < p, \\ \sum_{j=h:m_{i,j}\geq p}^{s_i} LC_{i,j} + & \text{else if } 0 < L' \leq LC_i \text{ and} \\ \quad (L' - \sum_{j=h}^{s_i} LC_{i,j}), & \quad m_{i,h-1} \geq p, \\ \sum_{\forall j:m_{i,j}\geq p} LC_{i,j}, & \text{otherwise,} \end{cases}$$

where h indicates the earliest segment that is fully included in the carry-in interval. Then, the $(h-1)$-th segment can execute partially within the carry-in interval. In the example in Figure 5, h indicates the 3rd segment, and the 2nd segment (the $(h-1)$-th segment) partially contributes to $CI^+_{i,k}(p, L')$, for each p, by 1. Then, the at least p-depth workload $W^+_{i,k}(p, r_k, d_k)$ that will contribute to the worst case, for $i \neq k$, is expressed as follows:

$$W^*_{i,k}(p, D_k) = BD^+_{i,k}(p, D_k) + CI^+_{i,k}(p, D_k \% T_i). \tag{15}$$

We can then compute bounds on the amount of inter-task interference as follows:

$$\widehat{W^+_{i,k}} \leq \sum_{p=1}^{m_i} \min(W^*_{i,k}(p, D_k), D_k - LC_k). \tag{16}$$

2) Intra-Task Workload: The critical threads of task τ_k can get interference from the other threads belonging to the same task. For the intra-task interference, the worst-case release pattern is already determined as the execution window of a job of τ_k. Then, threads within a single job can interfere with each other if they belong to the same segment, but not otherwise. For each segment, all threads except the critical thread can interfere on the critical thread, and it is clear that the interference of a single thread $\theta_{k,u,v}$ on the critical thread $\theta_{k,u,v*}$ is upper bounded by $C_{k,u,v}$. With the same reasoning for the inter-task interference case, $\widehat{W^+_{k,k}}$ is maximized under the worst-execution pattern where the maximum possible number of threads of task τ_k execute in parallel as much as possible. Then, $W^+_{k,k}(p, r_k, d_k)$ that will contribute to the worst case is calculated as follows:

$$W^*_{k,k}(p, D_k) \stackrel{\text{def.}}{=} \sum_{\forall j : m_{k,j} \geq p+1} LC_{k,j}. \tag{17}$$

We can compute bounds on the amount of intra-task interference as

$$\widehat{W^+_{k,k}} \leq \sum_{p=1}^{m_k} \min(W^*_{k,k}(p, D_k), D_k - LC_k). \tag{18}$$

3) Total Workload:

Lemma 5: When $m \geq m_i$ for all task τ_i, a task set τ is schedulable under global EDF scheduling on m identical processors if for each task $\tau_k \in \tau$,

$$\begin{aligned} \sum_{i \neq k} \sum_{p=1}^{m_i} \min(W^*_{i,k}(p, D_k), D_k - LC_k) \\ + \sum_{p=1}^{m_k} \min(W^*_{k,k}(p, D_k), D_k - LC_k) \\ \leq m(D_k - LC_k). \end{aligned} \tag{19}$$

Proof: From Lemmas 3 and 4, and Inequality (13), the left-hand side is an upper bound of all the inter-task interferences of all tasks τ_i on task τ_k, where $i \neq k$, and the intra-task interference of task τ_k on itself. Then, by Theorem 1, if Eq. (19) is satisfied for all tasks in a task set τ, the task set is schedulable under global EDF scheduling on m identical processors. ■

B. The number of processors is smaller than the number of threads in some segments ($m < m_i$) for some i

We now remove the restriction of $m_{i,j} \leq m$ for each segment $\sigma_{i,j}$ such that m_i can be larger than m.

Lemma 6: When $m < m_i$ for some tasks τ_i, Inequality (19) still holds.

Proof: Let $M = \max_i\{m_i\}$. Suppose we have M processors. Then, from Lemmas 3 and 4, the following holds for any job J_k of τ_k:

$$\begin{aligned}&\sum_{p=1}^{M} \min(W^+_{i,k}(p, r_k, d_k), D_k - LC_k)\\ \leq\ &\sum_{i\neq k}\sum_{p=1}^{m_i} \min(W^*_{i,k}(p, D_k), D_k - LC_k)\\ &+\sum_{p=1}^{m_k} \min(W^*_{k,k}(p, D_k), D_k - LC_k). \end{aligned} \tag{20}$$

When $m < M$, the value of $W^+_{i,k}(p, r_k, d_k)$ may change because the execution pattern changes. However, we will prove that:

$$\begin{aligned}&\sum_{p=1}^{m} \min(W^+_{i,k}(p, r_k, d_k), D_k - LC_k)\\ \leq&\sum_{p=1}^{M} \min(W^+_{i,k}(p, r_k, d_k), D_k - LC_k). \end{aligned} \tag{21}$$

When we have M processors, the workload in the p-th processor can be considered as the at least p-depth workload $W^+_{i,k}(p, a, b)$. Since we decrease the number of processors from M to $m < M$, we have to re-distribute the workload in $M-m$ processors to the remaining m processors. Choose $M - m$ smallest value of $W^+_{i,k}(p, a, b)$. Let $W_{(M-m)}$ be the largest value among them and $\sum W_{(M-m)}$ be the sum of them. There are two cases.

Case 1. ($W_{(M-m)} < D_k - LC_k$). Then $\min(W^+_{i,k}(p, a, b), D_k - LC_k) = W^+_{i,k}(p, a, b)$ for all the $(M - m)$ chosen values. Hence the amount subtracted from the left-hand side of Inequality (20) is exactly $\sum W_{(M-m)}$. The amount then added to the remaining m processors can be at most $\sum W_{(M-m)}$. Hence Inequality (21) holds.

Case 2. ($W_{(M-m)} \geq D_k - LC_k$). Then $\min(W^+_{i,k}(p, a, b), D_k - LC_k) = D_k - LC_k$ for all the m non-chosen values. Hence when we add the $M-m$ chosen workloads to the remaining m workloads, all will be dropped because the original workload already exceed $D_k - LC_k$. Thus some amount may be subtracted from the left-hand side of the Inequality (20) and none is added. Hence Inequality (21) holds.

From Inequality (20) and (21), Lemma 6 holds. ■

C. Schedulability Test

From Lemmas 5 and 6, we have the following theorem.

Theorem 2: A task set τ is schedulable under global EDF scheduling on m identical processors if for each task $\tau_k \in \tau$,

$$\begin{aligned}&\sum_{i\neq k}\sum_{p=1}^{m_i} \min(W^*_{i,k}(p, D_k), D_k - LC_k)\\ &\quad+\sum_{p=1}^{m_k} \min(W^*_{k,k}(p, D_k), D_k - LC_k)\\ &\quad\leq m(D_k - LC_k). \end{aligned} \tag{22}$$

Complexity. We denote the number of tasks in a task set by n. Note that it requires $O(n)$ to calculate Eq. (22) for a given τ_k. Therefore, the schedulability test in Theorem 1 requires $O(n^2)$.

It is worth noting that $W^*_{i,k}(p, D_k)$ in Eq. (15) is a generalization of a workload-based interference bound for the single-thread task case [30]. They are equivalent when task τ_i has a single segment with a single thread.

VI. Evaluation

The goal of this paper is to develop global EDF schedulability analysis that is directly applicable to a set of synchronous parallel tasks, and this section presents simulation results for the evaluation of our proposed analysis.

Simulation Environment. In order to understand how the proposed analysis behaves to synchronous parallel tasks, we employ a simulation parameter in task set generation. This parameter controls the ratio of the number of synchronous parallel tasks to the number of entire tasks in each task set from 0% to 100%. We generate task sets by adapting the technique proposed for the sequential task model in [33]. For both sequential and parallel tasks τ_i, their task parameters are determined as follows: period and deadlines ($T_i = D_i$)[1] are uniformly chosen in $[100, 1000]$. For the parallel task case, the number of segments (s_i) and the number of threads within each segment $\sigma_{i,j}$ ($m_{i,j}$) are uniformly distributed in $[1, 5]$ and $[1, 3m/2]$, respectively, where m is the number of processors. All threads within the same segment share the same WCET, and the WCET is uniformly chosen in $[1, T_i/s_i]$. For the sequential task case, $C_i = LC_i$ are uniformly chosen in $[1, T_i]$. We generate 40,000 task sets for $m = 4$ and $m = 8$ with the parallel task ratio from 0% to 100%.

According to the parameters determined as described above, we first generate a set of m tasks and then keep creating an additional new task set by adding a new task into the old set until the system utilization (i.e., $U_{sys} = \sum_{\tau_i\in\tau} U_i$) becomes greater than m. Table I characterizes the task sets generated. In the table, the *parallelism index* of a task τ_i (denoted by PF_i) is defined as the ratio of its WCET upper-bound C_i to its WCET lower-bound LC_i (i.e., $\mathsf{PF}_i = C_i/LC_i$), and it indicates how much a given task can be allowed to exploit intra-task parallelism. It is intuitive that PF_i grows as the ratio of parallel tasks becomes larger.

For the generated task sets, we perform simulations for our schedulability test in Theorem 2 (denoted by OUR). We

[1] In this section, we only show the results of implicit deadline tasks due to space limitation, but the behaviors of constrained deadline tasks are similar to those of implicit ones.

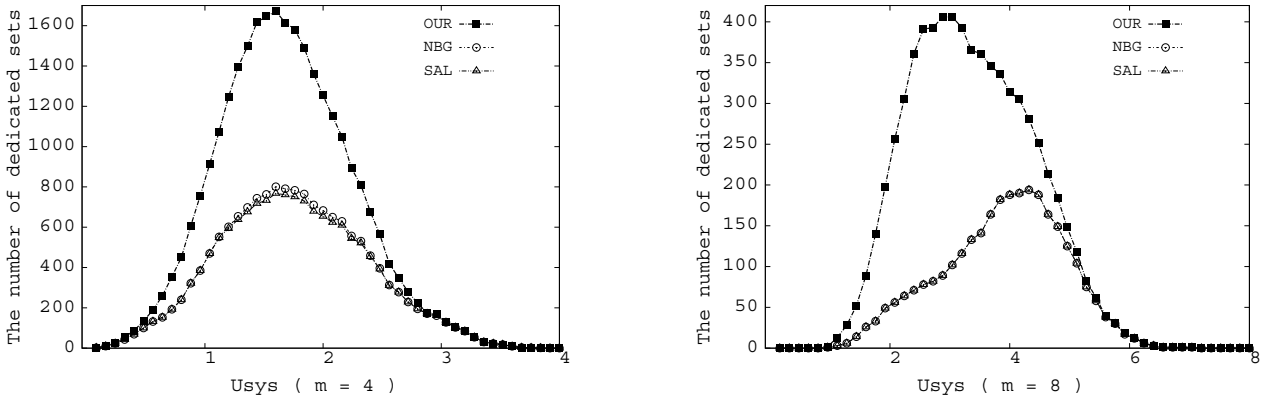

Fig. 6. Schedulability under different system utilizations

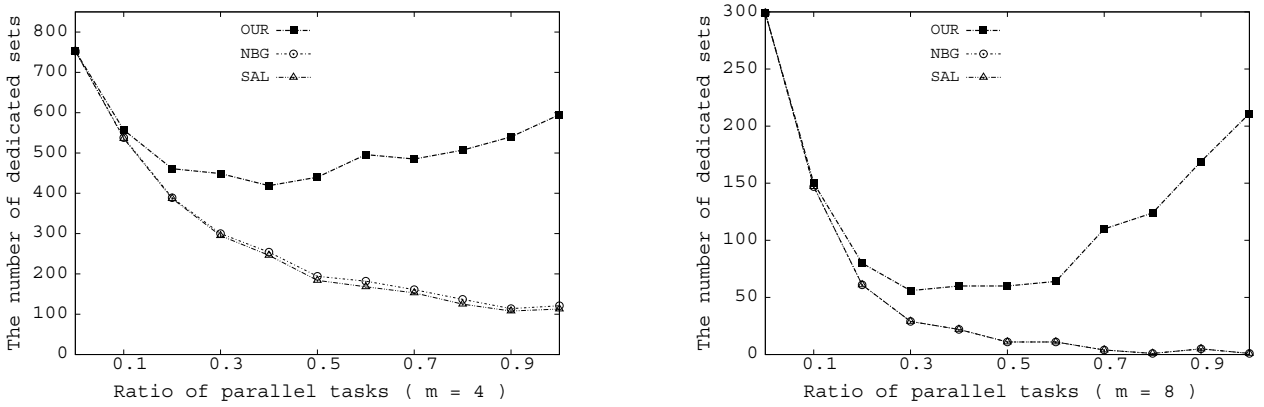

Fig. 7. Schedulability under different parallel task ratios

TABLE I
CHARACTERISTICS OF GENERATED TASK SETS FOR SIMULATION

Avg.	m	Parallel task ratio				
		0.1	0.3	0.5	0.7	0.9
Number of Tasks	4	4.4	4.7	4.9	5.2	5.6
	8	8.4	8.6	9.2	9.7	10.5
U_{sys}	4	2.9	2.8	2.7	2.7	2.7
	8	6.0	5.7	5.6	5.5	5.5
PF_i	4	1.3	1.9	2.4	2.8	3.2
	8	1.7	2.9	4.0	5.0	5.9

compare our tests with two other related approaches [20], [21] (referred to as SAL and NBG, respectively).[2]

In those approaches, a single parallel task is decomposed into multiple sequential sub-tasks such that each sub-task corresponds to a thread and is assigned its own offset and deadline. Then, sub-tasks belonging to different segments within the same task are separated by their offsets and deadlines. This way, sub-tasks are subject to experiencing interference from sub-tasks belonging to other parallel tasks and the sub-tasks belonging to the same segment. While those approaches are not designed for deadline-based analysis, we employ an existing deadline-based schedulability test [8], upon which our proposed schedulability tests are built, for the decomposed sequential sub-tasks.[3] We note that the schedulability tests used for SAL and NBG have the same time complexity of $O(n^2)$ as our proposed schedulability test, but SAL and NBG require additional computations for assigning the deadlines of sub-tasks.

Simulation Results. In Figure 6, we plot the number of task sets deemed schedulable by each schedulability test, with different system utilizations for $m = 4$ and $m = 8$. The figure shows that OUR significantly outperforms other schedulability tests. In both cases of $m = 4$ and $m = 8$, OUR finds 81% and 134% more schedulable task sets which are deemed schedulable by neither NBG nor SAL, respectively. We can interpret such a consistent gap as the benefit of using direct schedulability analysis for synchronous parallel tasks, compared to indirect approaches based on task decomposition.

Figure 7 plots the same simulation results presented in Figure 6 from a different angle, showing them over different parallel task ratios from 0% to 100%. Note that when the ratio is 0% (i.e., there are only sequential tasks in task sets), all the

[2]Other related analysis techniques are not included in our evaluation, because those in [18], [19] are applicable to more restrictive parallel task models and the one in [22] is applicable only to the single DAG task case, while the multiple parallel task case is of our interest.

[3]Those decomposition-based approaches can work with other schedulability analysis, such as response time analysis (RTA) [9]. As demonstrated in the literature [8], [9], it is possible to extend the concept and technique behind deadline-based analysis toward response time analysis. Thereby, our proposed notion of parallelism-aware interference can be extended toward response time analysis and it will be possible to compare our approach with others according to response time analysis. However, this is beyond the scope of this paper due to the limit of space.

analyses (OUR, SAL and NBG) yield the same results. This is because task decomposition is no longer necessarily applied to sequential tasks and our schedulability test is reduced to the existing one [8] for the sequential task case.

One interesting observation is that task decomposition-based approaches find less task sets deemed schedulable as the parallel task ratio increases, while our approach finds a larger number of such tasks on average. This can be interpreted that the overheads of task decomposition are accumulated with a growing number of parallel tasks. On the other hand, OUR is relatively much insensitive to the parallel task ratio, implying that it is effectively dealing with the thread-level parallelism and segment-level synchronization of the synchronous parallel task model.

More technically, Table I shows that when the parallel task ratio grows, the parallelism index increases and thereby the WCET lower-bound LC_k of a task τ_k generally decreases. This gives task τ_k more room to accommodate larger interference from other tasks, leading to better schedulability. However, it can be interpreted that such a potential for schedulability improvement is not well exploited in NBG and SAL since they do not consider intra-task parallelism directly. In those approaches, the entire execution window of a synchronous parallel task is divided into smaller intervals of its sub-tasks through intermediate deadlines, and this seems to severely limit the flexibility in executing subtasks; they now need to execute only within their own artificial execution windows while corresponding threads have flexibility in running even outside such artificial windows in our approach. Such a difference leads to a significant gap between schedulability for synchronous parallel tasks.

VII. Conclusion

The motivation for our work was the desire to understand the thread-level parallelism and segment-level synchronization of synchronous parallel tasks in the context of hard real-time multi-core scheduling. In this paper, we extended the notion of interference formalizing it at a finer-grained thread level and building a connection to the notion at a task level. We then generalized interference-based analysis methods according to the new proposed notion of interference, introducing the first global EDF schedulability conditions that are directly applicable to a set of synchronous (malleable) parallel tasks. Our evaluation results showed that it significantly improves the state-of-the-art analysis techniques available for synchronous parallel tasks.

This paper incorporated thread-level parallelism directly into schedulability analysis focusing on the EDF algorithm. However, we believe the schedulability of synchronous parallel tasks can be advanced much more significantly if thread-level parallelism is directly reflected into scheduling algorithms as well. Hence, a direction of our future work includes developing new real-time scheduling algorithms that support intra-task parallelism and synchronization directly.

Acknowledgement

This work was supported in part by BRL (2009- 0086964), BSRP (2010-0006650, 2012-R1A1A1014930), NCRC (2012-0000980), NIPA (H0503-12-1041), KEIT (2011-10041313), DGIST CPS Global Center, and KIAT (M002300089) funded by the Korea Government (MEST/MKE).

References

[1] "Intel's teraflops research chip." [Online]. Available: http://techresearch.intel.com/ProjectDetails.aspx?Id=151.
[2] "Comming soon tile-gx100 the first 100 cores processors in the world," http://internalcomputer.com/coming-soon-tile-gx100-the-first-100-cores-processor-in-the-world.computer, Feb, 2011.
[3] D. Lea, "A java fork/join framework," in *Proceedings of the ACM Java Grande Conference*, 2000.
[4] *OpenMP*, http://openmp.org .
[5] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," *ACM SIGPLAN Notices*, vol. 33, no. 5, pp. 212–223, 1998.
[6] C. Liu and J. Layland, "Scheduling algorithms for multi-programming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.
[7] T. P. Baker, "Multiprocessor EDF and deadline monotonic schedulability analysis," in *RTSS*, 2003.
[8] M. Bertogna, M. Cirinei, and G. Lipari, "Improved schedulability analysis of EDF on multiprocessor platforms," in *ECRTS*, 2005.
[9] M. Bertogna and M. Cirinei, "Response-time analysis for globally scheduled symmetric multiprocessor platforms," in *RTSS*, 2007.
[10] S. Baruah, "Techniques for multiprocessor global schedulability analysis," in *RTSS*, 2007.
[11] N. Guan, M. Stigge, W. Yi, and G. Yu, "New response time bounds of fixed priority multiprocessor scheduling," in *RTSS*, 2009.
[12] R. Davis and A. Burns, "Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems," in *RTSS*, 2009.
[13] S. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: a notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, 1996.
[14] G. Levin, F. Shelby, S. Caitlin, P. Ian, and B. Scott, "DP-FAIR: a simple model for understanding optimal multiprocessor scheduling," in *ECRTS*, 2010.
[15] P. Regnier, G. Lima, E. Massa, G. Levin, and S. A. Brandt, "Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor," in *RTSS*, 2011.
[16] P. Jayachandran and T. Abdelzaher, "Transforming distributed acyclic systems into equivalent uniprocessors under preemptive and non-preemptive scheduling," in *ECRTS*, 2008.
[17] N. Serreli, G. Lipari, and E. Bini, "The demand bound function interface of distributed sporadic pipelines of tasks scheduled by edf," in *ECRTS*, 2010.
[18] S. Kato and Y. Ishikawa, "Gang EDF scheduling of parallel task systems," in *RTSS*, 2009.
[19] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *RTSS*, 2010.
[20] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *RTSS*, 2011.
[21] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic, "Techniques optimizing the number of processors to schedule multi-threaded tasks," in *ECRTS*, 2012.
[22] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *RTSS*, 2012.
[23] C. Liu and J. H. Anderson, "Supporting soft real-time dag-based systems on multiprocessors with no utilization loss," in *RTSS*, 2010.
[24] ——, "Supporting soft real-time parallel applications on multicore processors," in *RTCSA*, 2012.
[25] J. H. Anderson and J. M. Calandrino, "Parallel real-time task scheduling on multicore platforms," in *RTSS*, 2006.
[26] G. Manimaran, C. S. R. Murthy, and K. Ramamritham, "A new approach for scheduling of parallel tasks in real-time multiprocessor systems," *Real-Time Systems*, vol. 15, pp. 39–60, 1998.
[27] O.-H. Kwon and K.-Y. Chwa, "Scheduling parallel task with individual deadlines," *Theoretical Computer Science*, vol. 215(1), pp. 209–223, 1999.
[28] M. Holenderski, R. J. Brill, and J. J. Lukkien, "Parallel-task scheduling on multiple resources," in *ECRTS*, 2012.
[29] J. K. Ousterhout, "Scheduling techniques for concurrent systems," in *ICDCS*, 1982.
[30] M. Bertogna, M. Cirinei, and G. Lipari, "Schedulability analysis of global scheduling algorithms on multiprocessor platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, pp. 553–566, 2009.
[31] J. Lee, A. Easwaran, and I. Shin, "LLF Schedulability Analysis on Multiprocessor Platforms," in *RTSS*, 2010.
[32] H. S. Chwa, H. Back, S. Chen, J. Lee, A. Easwaran, I. Shin, and I. Lee, "Extending task-level to job-level fixed priority assignment and schedulability analysis using pseudo-deadlines," in *RTSS*, 2012.
[33] T. Baker, "An analysis of EDF schedulability on a multiprocessor," *IEEE Transactions on Parallel Distributed Systems*, vol. 16, no. 8, pp. 760–768, 2005.