Scheduling Parallel Real-Time Tasks on Multi-core Processors

Karthik Lakshmanan, Shinpei Kato, Ragunathan (Raj) Rajkumar
Department of Electrical and Computer Engineering
Carnegie Mellon University, Pittsburgh, USA
klakshma, shinpei, raj@ece.cmu.edu

Abstract

Massively multi-core processors are rapidly gaining market share with major chip vendors offering an ever-increasing number of cores per processor. From a programming perspective, the sequential programming model does not scale very well for such multi-core systems. Parallel programming models such as OpenMP *present promising solutions for more effectively using multiple processor cores. In this paper, we study the problem of scheduling periodic real-time tasks on multiprocessors under the* fork-join *structure used in* OpenMP. *We illustrate the theoretical best-case and worst-case periodic* fork-join *task sets from a processor utilization perspective. Based on our observations of these task sets, we provide a partitioned preemptive fixed-priority scheduling algorithm for periodic* fork-join *tasks. The proposed multiprocessor scheduling algorithm is shown to have a resource augmentation bound of* 3.42*, which implies that any task set that is feasible on* m *unit speed processors can be scheduled by the proposed algorithm on* m *processors that are* 3.42 *times faster.*

1 Introduction

Major chip manufacturers have recently ramped up the development of massively multi-core processors for a variety of reasons including power consumption, memory speed mismatch, and instruction-level parallelism limits. For example, AMD has introduced a 12-core Opteron [1] processor targeting the datacenter server market, while Intel has developed a 48-core single-chip computer for cloud computing [2]. Projecting these trends into the future, chip manufacturers predict hundreds of processor cores per chip in the near future. These developments, however, demand a dramatic change in conventional programming paradigms and software models.

Sequential programming models proved to be quite useful when processor manufacturers pushed for faster and faster processor clock speeds. As the semiconductor vendors shift the scaling trends towards more and more processor cores, the benefits of sequential programming start to diminish in comparison to the inability to take advantage of the available parallelism. Parallel programming models such as *OpenMP* [3] are promising candidates for taking advantage of future massive multi-core processors[1]. These models have the capability to parallelize specific segments of tasks, thereby leading to shorter response times when possible.

Most of the results in classical multiprocessor real-time scheduling theory [4, 5, 6, 7, 8, 9] are focused on the sequential programming model, where the problem is to schedule many sequential real-time tasks on multiple processor cores. Parallel programming models introduce a new dimension to this problem, where jobs may be split into parallel execution segments at specific points. Recent results [10, 11] have considered different task models for parallel programming. In this work, we focus on the *fork-join* programming model employed by the *OpenMP* system. To the best of our knowledge, prior work has not seriously studied the fork-join task model in the context of real-time systems. *OpenMP* is a mature system for parallel programming, and is expected to play a pivotal role in shaping future programming paradigms for multi-core processors.

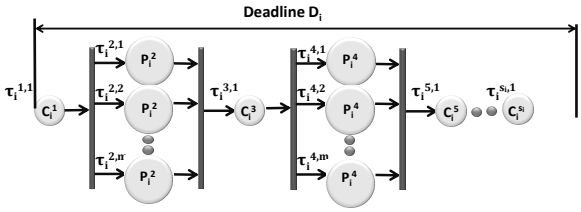


Figure 1. Fork-Join Task Model.

Fork-Join is a popular parallel programming paradigm employed in systems such as Java [12] and OpenMP. In this work, we study *basic* fork-join tasks as shown in Figure 1. Each basic fork-join task begins as a single master thread that executes sequentially until it encounters the first *fork* construct, where it splits into multiple parallel threads which execute the parallelizable part of the computation. After the parallel execution region, a *join* construct is used to synchronize and terminate the parallel threads, and resume the master execution thread. This structure of *fork* and *join* can be repeated multiple times within a job execution. A *general* fork-join task is one where the parallel execution regions themselves can be fork-join structures. A

[1] In the context of this work, we will use the terms processor and processor core interchangeably.

1052-8725/10 $26.00 © 2010 IEEE
DOI 10.1109/RTSS.2010.42

study of such nested fork-join structures is beyond the scope of this work, and presents a direction in which our results can be extended. In the context of this work, we also assume that the parallel execution regions themselves can be preempted individually on their respective processors. This assumption may not hold good in certain systems, where approaches such as gang scheduling [11] may be required. Henceforth, when we use the term *fork-join*, it denotes the *basic* fork-join task model as described above.

Many real-time systems, such as radar tracking, autonomous driving, and video surveillance, exhibit a data-parallel nature that lends itself easily to the *fork-join* model. As the problem sizes scale and processor speeds saturate, the only way to meet task deadlines in such systems would be to parallelize the computation. In this work, we therefore study the problem of scheduling periodic real-time tasks with implicit-deadlines that employ the *fork-join* parallelization construct. We primarily focus on preemptive fixed-priority scheduling algorithms due to the readily available support for fixed-priority scheduling in commercial operating systems such as Linux and Windows, and in industry standards such as POSIX and Real-Time Specification for Java. Dynamic priority scheduling for *fork-join* tasks is beyond the current scope of this work. We also restrict our attention to tasks with a pre-specified static number of threads, as dictated by the `OMP_NUM_THREADS` environment variable in OpenMP. Although the number of threads in a parallel region can be dynamically adjusted in specific implementations of OpenMP, we do not consider such a task model in this work. Analyzing such dynamic *fork-join* task structures forms an important aspect of future work.

Contributions: The main contributions of this paper are: (i) the best-case and worst-case basic fork-join task sets from a feasibility perspective, (ii) the task *stretch* transform to reduce the scheduling penalty of basic fork-join structures, and (iii) a task partitioning algorithm for deadline-monotonic scheduling with a resource augmentation bound of 3.42.

Organization: We introduce our task model in Section 2. For such task sets, we show the theoretical best-case and worst-case task characteristics in Section 3. Based on our observations of these key task structures, we develop a task transform in Section 4. We then describe a partitioned preemptive fixed-priority scheduling approach for scheduling *fork-join* task sets in Section 5 and provide key resource augmentation results for the same. Finally, we provide concluding remarks in Section 6.

2 Task Model

Each task in the system starts out as a sequential thread, and then alternates between parallel and sequential segments. We represent each task as
$\tau_i : ((C_i^1, P_i^2, C_i^3, P_i^4, ..., P_i^{s_i-1}, C_i^{s_i}), m_i, T_i)$, where:

- s_i is the number of computation segments (both parallel and non-parallel) in task τ_i. Note that s_i is by definition an *odd* integer since we define fork-join tasks to both start and finish with a non-parallel execution segment.
- m_i is the number of parallel OpenMP threads spawned by task τ_i in its parallel regions. $m_i > 1$ for parallelized tasks and $m_i = 1$ for sequential tasks. We also assume that $m_i \leq m$, where m is the total number of available processing cores. Note that we assume that each parallel region of a task has an equal number of parallel threads.
- C_i^s is the worst-case execution time of sequential segment s in task τ_i on an unit-speed processor, where $1 \leq s \leq s_i$ and s is *odd*. For ease of presentation, we will refer to this sequential segment as $\tau_i^{s,1}$.
- P_i^s is the worst-case execution time of each thread spawned in the parallel segment s of task τ_i on an unit-speed processor, where $1 < s < s_i$ and s is *even*. For ease of presentation, we will refer to these m_i parallel threads as $\tau_i^{s,1}$ through τ_i^{s,m_i}. We also assume that these parallel threads are independent of each other except for the requirement that *all* parallel threads of stage s need to complete before the execution of stage $(s+1)$ can begin.
- T_i is the period of task τ_i (also equal to its relative deadline $D_i = T_i$).

Using this task model, we also define the *minimum execution length* of a task τ_i as:

$$\eta_i = \sum_{s=0}^{\frac{s_i-1}{2}} (C_i^{2s+1}) + \sum_{s=1}^{\frac{s_i-1}{2}} (P_i^{2s})$$

Observe that η_i is the response time of task τ_i when it is assigned m_i processor cores exclusively so that there is no interference from any other tasks.

Correspondingly, the *maximum execution length* of a task τ_i can be defined as:

$$C_i = \sum_{s=0}^{\frac{s_i-1}{2}} (C_i^{2s+1}) + m_i \sum_{s=1}^{\frac{s_i-1}{2}} (P_i^{2s})$$

where, C_i is the response time of task τ_i when it is assigned a single processor core exclusively.

For notational convenience, we also define

$$P_i = \sum_{s=1}^{\frac{s_i-1}{2}} (P_i^{2s})$$

as the minimum parallel segment execution time.

The ratio of the *maximum execution length* to *minimum execution length* is defined as the *parallelism speedup factor* Υ_i possible for task τ_i under maximal parallelism

$$\Upsilon_i = \frac{C_i}{\eta_i}$$

Based on these definitions, the following properties hold:

- For a task τ_i that cannot be parallelized, $\Upsilon_i = 1$.
- For a task that can be *fully* parallelized on m_i cores i.e. $s_i = 3$, $C_i^1 = C_i^3 = 0$, and $P_i^2 = \frac{C_i}{m_i}$, $\Upsilon_i = m_i$.

Proposition 1. *For a given processor speed, if the minimum execution length η_i of any task τ_i is greater than its period (implicit deadline) T_i, then τ_i is not schedulable on any number of processors with the same speed.*

Proof. The proof trivially follows from the definition of *minimum execution length*. η_i is the response time of τ_i when it is assigned m_i processor cores *exclusively*. Adding additional processor cores does not result in any additional speed up for τ_i since it never spawns more than m_i parallel execution threads in any of its parallel segments. Therefore, if jobs of τ_i cannot meet their implicit deadline on m_i processors, then they cannot meet their deadlines with any number of additional processor cores. □

A task set τ in this task model is represented as $\tau: \{\tau_1, \tau_2, ..., \tau_n\}$, where each task τ_i follows the *fork-join* structure. Without loss of generality, we assume that these tasks are sorted in non-decreasing order of task periods. The implicit-deadline nature of these tasks also means that these tasks are also sorted in non-decreasing order of relative deadlines.

Using the *fork-join* parallel real-time task model, we next characterize the best-case and worst-case task set structures for multiprocessor systems.

3 Worst-Case and Best-Case Task sets in the Fork-Join Model

Consider a processor with m processor cores. From a feasibility analysis perspective, we are interested in (i) the task set with *maximum* processor utilization that is *feasible* on the given processor with m cores, and (ii) the task set with *minimum* processor utilization that is *infeasible* on the given processor with m cores. In this section, we develop both task sets, and identify the characteristics for developing scheduling algorithms for fork-join task sets.

3.1 Best-Case Fork-Join Task Structure

The theoretical best-case structure for fork-join tasks to be scheduled on m processor cores happens when each task τ_i in the task set τ has the structure of $\tau_i : ((0, \frac{C_i}{m}, 0), m, T_i)$. The utilization of each task τ_i is $\frac{C_i}{T_i}$. We will now proceed to show that if any task set τ comprises solely of tasks with type $\tau_i : ((0, \frac{C_i}{m}, 0), m, T_i)$, then τ is feasible as long as the total utilization of τ does not exceed m.

Proposition 2. *When each task τ_i in a fork-join task set τ has the structure $\tau_i : ((0, \frac{C_i}{m}, 0), m, T_i)$ with an implicit deadline of T_i, then the task set τ is feasible on m processor cores as long as the cumulative utilization does not exceed m.*

Proof. The proof follows by showing that there exists a scheduling algorithm which guarantees that jobs of each task τ_i in the given fork-join task set τ will meet their deadlines.

Under the task structure $\tau_i : ((0, C_i/m, 0), m, T_i)$, each fork-join task is composed of exactly m parallel threads that execute for C_i/m time units each. Consider a Global EDF schedule of τ, where at any time instant t, all m processor cores will execute parallel threads from the same task. Given that each task is composed of exactly m parallel threads of equal execution requirements that are released simultaneously, the schedules on all the m processor cores are identical. Therefore, for the given task set to be feasible under Global EDF, it is sufficient that the threads scheduled on each individual processor core schedulable under EDF. From the schedulability condition proposed in [13], it follows that τ is schedulable as long as each individual processor core's utilization does not exceed 1 or the total utilization of τ does not exceed m. □

Corollary 3. *When each task τ_i in a fork-join task set τ has the structure $\tau_i : ((0, \frac{C_i}{m}, 0), m, T_i)$ with an implicit deadline of T_i, then the task set τ is schedulable under Global RMS on m processor cores as long as the cumulative utilization does not exceed $m \ln 2$.*

Proof. Using the argument from Proposition 2 that each task in τ is composed of exactly m parallel threads of equal execution requirements that are released simultaneously, the schedules on all the m processor cores are identical. It is therefore sufficient that the threads on each individual processor core are schedulable under uniprocessor RMS. From the schedulability condition in [13], it follows that τ is schedulable as long as each individual processor core's utilization does not exceed $\ln 2$ or the total utilization of τ does not exceed $m \ln 2$. □

It should be noted here that Proposition 2 is closely related to the result proved in [14], which shows that EDF with all jobs parallelized on all processors is optimal. Proposition 2 derives this result in the context of the fork-join task model.

We now show a worst-case task set that has the minimum total utilization among all infeasible fork-join task sets.

3.2 Theoretical Worst-Case Fork-Join Taskset

The theoretical worst case for fork-join task sets from a schedulability perspective is shown in Figure 2. This scenario comprises an infeasible fork-join task set with the least possible cumulative utilization among all infeasible fork-join task sets. In this scenario, there are two tasks τ_i

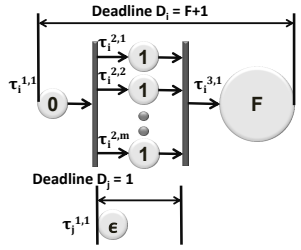


Figure 2. Worst-Case Fork-Join Task Set.

and τ_j. Task τ_i has the structure $\tau_i : ((0, 1, F), m, 1 + F)$,

and τ_j has the structure $\tau_j : ((\epsilon), 1, 1)$. For $F >> m$ and arbitrarily small $\epsilon > 0$, the utilization U of this task set is:

$$U = \frac{m+F}{1+F} + \frac{\epsilon}{1} = 1 + \epsilon + \frac{m-1}{1+F} \approx 1 \qquad (1)$$

The utilization U is slightly greater than 1.

Proposition 4. *The fork-join task set comprising of two tasks $\tau_i : ((0, 1, F), m, 1 + F)$ and $\tau_j : ((\epsilon), 1, 1)$ is unschedulable on a system with no greater than m processing cores.*

Proof. The infeasibility follows from the critical instant when both τ_i and τ_j are released together at time 0. At this instant, τ_i will have m parallel threads with unit execution requirements that are ready to be scheduled, while τ_j will have 1 thread with ϵ execution requirement that is ready. There is at most a total of m available processor cores, therefore over the time interval $[0, 1]$ no more than m time units of execution can be completed causing either τ_j to miss its implicit deadline of 1 or one of the parallel threads of τ_i to face a preemption of ϵ. If any parallel thread of τ_i faces a preemption of ϵ from τ_j, then it would cause τ_i to miss its deadline since the *minimum execution length* of τ_i is equal to its period (implicit deadline) of $(1 + F)$ and an additional preemption of ϵ would cause it to miss its deadline by ϵ. Therefore, the task set comprising of tasks τ_i and τ_j is unschedulable. □

Lemma 5. *For $F >> m$ and arbitrarily small $\epsilon > 0$, the fork-join task set comprising of two tasks $\tau_i : ((0, 1, F), m, 1 + F)$ and $\tau_j : ((\epsilon), 1, 1)$ is an infeasible task set with the least possible utilization among all infeasible task sets.*

Proof. The proof is obtained by contradiction. For large $F >> m$ and small $\epsilon > 0$, the utilization of the task set with tasks $\tau_i : ((0, 1, F), m, 1 + F)$ and $\tau_j : ((\epsilon), 1, 1)$ is arbitrarily close to 1 as shown in Equation (1). From Proposition 4, it also follows that this task set with τ_i and τ_j is unschedulable on m processor cores. Suppose there exists another task set τ that has a lower utilization $U_\tau \leq 1$, and τ is infeasible on m processors. Consider a transform of task set τ to τ', where each task τ_i in τ is considered as a conventional sequential task $\tau'_i : ((C_i), 1, T_i)$ in task set τ'. Taskset τ' is schedulable under the Earliest-Deadline First (EDF) scheduling algorithm [13] on a uniprocessor. Therefore, the original task set τ is feasible by scheduling the fork-join tasks as sequential tasks on one processor core using EDF. This results in a contradiction that the fork-join task set τ is infeasible. □

Analyzing the worst-case fork-join task set shown in Figure 2, note that the minimum execution length $C_i = (1+F)$ of τ_i is equal to its period of $T_i = (1 + F)$. This translates to allowing no slack for any execution segment of τ_i. The parallel execution segments of τ_i therefore have an execution requirement of 1, and an inherited deadline of 1 since it would cause τ_i to miss its deadline otherwise. Therefore, these parallel execution segments can be considered as *subtasks* with an execution requirement of 1, period of $(1+F)$, and a constrained deadline of 1. Therefore, no additional task with a period (and deadline) less than or equal to 1 is schedulable on the m cores since the execution of τ_i has zero available slack. In this scenario, a single processor has a total utilization of 1, while the remaining $(m-1)$ processors have a utilization of $\frac{1}{1+F}$ each. For arbitrarily large F, this task set demonstrates that the m processor utilization bound of fork-join task sets reaches only the single processor utilization bound of 1 for EDF.

The following are our key observations from the theoretical best-case and worst-case task sets:

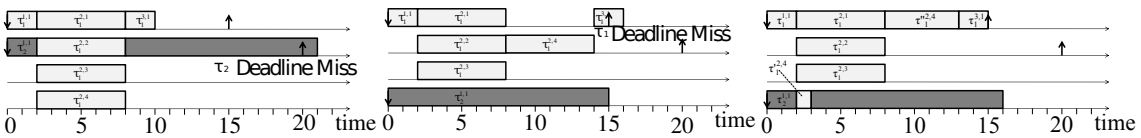
(1) Uniformly parallelized tasks with a *parallelism speedup factor* Υ of m provide the most benefit from a schedulable utilization perspective (from Proposition 2). Transforming the given *fork-join* tasks to resemble the best-case task structure might prove useful to improve schedulable utilization.

(2) There exist task sets with a total utilization slightly greater than and arbitrarily close to 1 that are unschedulable on a system with m processor cores (from Lemma 5). Therefore, conventional *utilization bounds* such as those employed in [13] may not be useful in characterizing the performance of scheduling algorithms for *fork-join* task sets. Resource augmentation bounds such as those presented in [16] seem to be promising candidates for the performance analysis of scheduling algorithms in the context of *implicit-deadline fork-join* task sets. Based on these observations, we now define a task transform that reduces the problem of scheduling *fork-join* task sets on multiprocessors to the problem of scheduling *constrained-deadline* task sets on multiprocessors. We then develop key resource augmentation bounds for analyzing the schedulability of such task sets.

4 Task Transformation

Fork-join task sets on multiprocessor systems can have schedulable utilization bounds slightly greater than and arbitrarily close to uniprocessor schedulable utilization bounds. From the perspective of schedulability, it is therefore desirable to avoid such task structures as much as possible. In this section, we propose a fork-join task *stretch* transform that avoids fork-join structures when possible. Based on this task transform, we will develop a partitioned preemptive fixed-priority scheduling algorithm for fork-join task sets.

We first illustrate the task *stretch* transform with an example fork-join task set having two tasks $\tau_1 : ((2, 6, 2), 4, 15)$ and $\tau_2 : ((15), 1, 20)$ to be scheduled on 4 processors. Under global scheduling (see Figure 3(a)), τ_2 misses its deadline since threads of τ_1 have a higher priority (under both EDF and DM). Under partitioned scheduling based on First-Fit Decreasing (FFD), the deadline miss for τ_2 may be avoided by allocating it exclusively to a processor. However, such an allocation causes τ_1 to miss its deadline instead, as shown in Figure 3(b). Now consider an alter-



(a) Unschedulable under traditional global Scheduling.
(b) Unschedulable under partitioned Scheduling (with FFD).
(c) Schedulable under *stretch* Transform.

Figure 3. Scheduling a task set $\tau_1 : ((2, 6, 2), 4, 15)$ and $\tau_2 : ((15), 1, 20)$ on 4 cores.

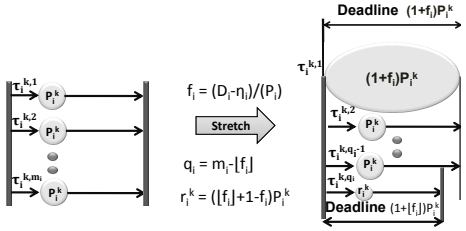


Figure 4. Task *Stretch* Transformation.

native allocation, where the master thread $(\tau_1^{1,1}, \tau_1^{2,1}, \tau_1^{3,1})$ of task τ_1 is allocated to processor 1, requiring 10 time units of execution every 15 time units. Therefore, τ_1 will have an available slack of 5 time units on processor 1. The parallel segments $\tau_1^{2,2}$ and $\tau_1^{2,3}$ are allocated to processors 2 and 3 respectively. In order to consume the 5 time units of available slack on processor 1, we also assign 1 time unit of parallel segment $\tau_1^{2,4}$ to processor 4, while the remaining 5 time units are sequentially executed after $\tau_1^{2,1}$ on processor 1. This *stretching* of task τ_1 ensures that task τ_2 can be accommodated on processor 4. Thus, the two tasks can be scheduled without missing any deadlines (see Figure 3(c)).

We now formally develop the task *stretch* transform for scheduling basic fork-join task sets. Let us consider a fork-join task τ_i : $((C_i^1, P_i^2, C_i^3, P_i^4, ..., P_i^{s_i-1}, C_i^{s_i}), m_i, T_i)$. Task τ_i can be alternatively represented as the sequence of subtasks $\tau_i : (\tau_i^{1,1}, (\tau_i^{2,1}, ..., \tau_i^{2,m_i}), \tau_i^{3,1}, (\tau_i^{4,1}, ..., \tau_i^{4,m_i}), ..., \tau_i^{s_i,1})$. The *master string* of a non-stretched task τ_i is defined as the *thread sequence* $(\tau_i^{1,1} \rightarrow \tau_i^{2,1} \rightarrow ..., \tau_i^{s,1} \rightarrow ... \rightarrow \tau_i^{s_i,1})$ comprising of exactly one thread $\tau_i^{s,1}$ for each computation segment s (either parallel or sequential). The goal of our task *stretch* transform illustrated in Figure 5 is to increase the *execution length* of the *master string* of each task τ_i to equal the task period T_i. We define the operation $\oplus$ of *coalescing* a thread with the *master string* as inserting $\tau_i^{k,j}$ into the *master string* while respecting thread precedence constraints and offsets.

For applying the *stretch* transform to a task τ_i with period T_i and maximum execution time C_i, we need to consider the following two cases for the value of C_i for task τ_i:

• $C_i \leq T_i$. Under the task *stretch* transform, τ_i can be treated as a *non-parallel* task with execution requirement C_i, task period T_i, and an implicit deadline D_i of T_i. From an implementation perspective, this simply requires all the subtasks of τ_i to be statically assigned to the same processor core. The actual program need not be modified for the *stretch* transform, as it can be easily accomplished at the OS scheduler level when assuming unique priorities or by assigning `OMP_NUM_THREADS` to 1. We can therefore ignore such tasks in this section.

• $C_i > T_i$. In this case, the *stretch* transform cannot avoid the fork-join structures completely. However, it can maximize the slack available to the parallelized segments by assigning the *master string* to its own processor core and eliminating any interference to the *master string*. From an implementation perspective, these tasks can also be transformed at the OS scheduler level without requiring modifications to the original task as we will show later.

The *stretch* transform is given in Figure 4. Consider the original task τ_i scheduled on m processor cores with an implicit deadline of $D_i = T_i$. The positive slack L_i available to task τ_i when it is scheduled exclusively without interference from other tasks is given by:

$$L_i = (D_i - \eta_i) \tag{2}$$

For the example task τ_1 given in Figure 3(c), $D_1 = 15$ and $\eta_1 = 10$ yielding a slack of $L_1 = 5$.

For notational convenience, we define

$$f_i = \frac{L_i}{\sum_{s=1}^{\frac{s_i-1}{2}} P_i^{2s}} = \frac{L_i}{P_i} \tag{3}$$

In the context of example task τ_1 in Figure 3(c), $L_1 = 5$ and $P_1 = 6$ resulting in $f_1 = \frac{5}{6}$.

As $C_i > T_i$, the *master string* can always be stretched to have an execution length of $D_i = T_i$. Specifically, stretching can be performed by having two or more parallel segments (or parts of them) being executed in sequence to take up the available slack. The remainder of any partial execution segment executed in this sequence will be executed in another core. The slack can therefore be distributed proportionally among the set of parallel segments to create constrained deadlines for scheduling purposes. Each parallel segment $2s$, $\forall 1 \leq s \leq (s_i - 1)/2$, is assigned a deadline d_i^{2s} as follows:

$$d_i^{2s} = P_i^{2s}(f_i + 1) \; \forall 1 \leq s \leq \tfrac{s_i-1}{2}$$

In the case of τ_1 in Figure 3(c), d_1^2 would be equal to $6(\frac{5}{6} + 1) = 11$.

In order to reduce the utilization loss arising from task partitioning, the *master string* of task τ_i is assigned its own

Algorithm: *StretchTask*
input: $\tau_i : (\tau_i^{1,1}, (\tau_i^{2,1}, ..., \tau_i^{2,m_i}), ..., \tau_i^{s_i,1})$
output:$(\tau_i^{master}, \{\tau_i^{cd}\})$
$\tau_i^{master} \leftarrow ()$
$\{\tau_i^{cd}\} \leftarrow \{\}$
if $C_i \leq T_i$ **then**
  ▷ *Make the task sequential*
  for $s \leftarrow 1$ to $\frac{s_i-1}{2}$ **do**
    $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{2s-1,1} : (C_i^{2s-1})$
    for $k \leftarrow 1$ to m_i **do**
      $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{2s,k} : (P_i^{2s})$
    end for
  end for
  $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{s_i,1} : (C_i^{s_i})$
else
  ▷ *Stretch the task to its deadline*
  $f_i \leftarrow \frac{D_i - \eta_i}{\sum_{s=1}^{\frac{s_i-1}{2}} P_i^{2s}}$
  $q_i \leftarrow (m_i - \lfloor f_i \rfloor)$
  for $s \leftarrow 1$ to $\frac{s_i-1}{2}$ **do**
    $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{2s-1,1} : (C_i^{2s-1})$
    for $k \leftarrow 1$ to m_i **do**
      if $k = 1$ or $k > q_i$ **then**
        ▷ *Part of the master string*
        $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{2s,k} : (P_i^{2s})$
      else
        if $k < q_i$ **then**
          ▷ *Create a new parallel thread*
          $\{\tau_i^{cd}\} \leftarrow \{\tau_i^{cd}\} \cup \tau_i^{2s,k} : (P_i^{2s}, (1+f_i)P_i^k)$
        else
          ▷ *Split among a parallel thread and the master string*
          $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i'^{2s,k} : ((f_i - \lfloor f_i \rfloor)P_i^k)$
          ▷ *Create a new parallel thread*
          $\{\tau_i^{cd}\} \leftarrow \{\tau_i^{cd}\} \cup \tau_i''^{2s,k} : ((\lfloor f_i \rfloor + 1 - f_i)P_i^k, (1 + \lfloor f_i \rfloor)P_i^k)$
        end if
      end if
    end for
  end for
  $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{s_i,1} : (C_i^{s_i})$
end if
return $(\tau_i^{master}, \{\tau_i^{cd}\})$

Figure 5. The task *Stretch* Transformation. Each newly created constrained deadline parallel thread is represented as $\tau : (C, D)$ with C as the worst-case execution time and D representing the deadline. When appending to an existing thread, we use $\tau : (C)$ to represent the execution time of the subtask appended.

processor. The slack available in each of the parallel segments for the *master string* is given by:

$$L_i^{2s} = (d_i^{2s} - P_i^{2s})$$

For example task τ_1, $L_1^2 = 5$ is the slack.

As the master string is assigned its own processor core, we can allocate this slack L_i^{2s} to threads from the same parallel segment as much as possible and coalescing them with the master string $\tau_i^{2s,1}$. After this allocation, each parallel segment will comprise of:

- $\tau_i^{2s,1}$ with a computation requirement of d_i^{2s} and constrained deadline of d_i^{2s},
- $(m_i - \lfloor f_i \rfloor - 2)$ parallel threads with a computation requirement of P_k^{2s} and a constrained deadline of d_i^{2s}
- one remaining thread with a computation requirement of $r_k^{2s} = (\lfloor f_i \rfloor + 1 - f_i)P_i^{2s}$ and a constrained deadline of $(1 + \lfloor fi \rfloor)P_i^{2s}$. The remaining computation will be executed on the processor allocated to the master string.

The total number of parallel threads in each parallel segment is given by:

$$q_i = (m_i - \lfloor f_i \rfloor)$$

where, all q_i threads have the same relative deadline, $(q_i - 2)$ threads have equal execution requirements, one thread (part of the master string) can possibly have a larger execution requirement than the others, and one thread can possibly have a smaller execution requirement than the others.

In the context of τ_1 in Figure 3(c) $q_i = 4$. τ_1 has equal execution requirements on processors 2 and 3, while there is more execution requirement from τ_1 on processor 1, and less execution requirement from τ_1 on processor 4.

The *master string* is schedulable by itself exclusively on a processor since it has an execution requirement of exactly $D_i = T_i$ due to the *stretch* operation described above.

4.1 Advantages of Task Stretch

The key advantages of task stretch are as follows: **(a)** The master string has an execution requirement exactly equal to the task period (and implicit deadline). This leads to an efficient task allocation under partitioned multiprocessor scheduling algorithms by avoiding any fragmentation of the available processor utilization. **(b)** The parallel threads can be statically assigned a release offset, avoiding any release jitter that may arise otherwise if the master string were to be co-scheduled with other higher-priority threads. The static offset ϕ_i^{2s} for threads $\tau_i^{2s,r}$ $\forall 1 \leq s \leq (s_i - 1)/2$ and $\forall 1 \leq r \leq q_i$ in parallel segment $2s$ is given by:

$$\phi_i^{2s} = \sum_{k=0}^{k=(s-1)} C_i^{2k+1} + \sum_{k=1}^{k=(s-1)} d_i^{2k}$$

(c) Each parallel thread $\tau_i^{2s,r}$ $\forall 1 \leq s \leq (s_i - 1)/2$ can be considered to be a uniprocessor constrained-deadline task

with a release offset of ϕ_i^{2s}, deadline of d_i^{2s}, and a period of T_i. This enables us to leverage known results for scheduling constrained deadline tasks on multiprocessors. We will use this approach for the scheduling algorithm that we propose for *fork-join* task sets. **(d)** The density of a subtask is defined as the ratio of its computation requirement to its relative deadline. All parallel threads other than those belonging to the master string itself, have a maximum density δ_i^{max} of:

$$\delta_i^{max} = \max_{s=1}^{\frac{s_i-1}{2}}\{\frac{P_i^{2s}}{(1+f_i)P_i^{2s}}, \frac{(\lfloor f_i \rfloor + 1 - f_i)P_i^{2s}}{(1+\lfloor f_i \rfloor)P_i^{2s}}\}$$

$$= \max_{s=1}^{\frac{s_i-1}{2}}\{\frac{P_i^{2s}}{(1+f_i)P_i^{2s}}, \frac{P_i^{2s} - (f_i - \lfloor f_i \rfloor)P_i^{2s}}{(1+f_i)P_i^{2s} - (f_i - \lfloor f_i \rfloor)P_i^{2s}}\}$$

$$\forall f_i \geq 0 \Longrightarrow \delta_i^{max} = \frac{1}{1+f_i} \quad (4)$$

This property is useful in developing resource augmentation bounds for partitioned preemptive fixed-priority scheduling for fork-join task sets. It is important to observe here that the task stretch transform is a means for achieving guaranteed schedulable utilization within a resource augmentation bound, and does not always result in the best possible schedulable utilization, hence we would need an in-depth analysis for our model.

4.2 Implementation Considerations

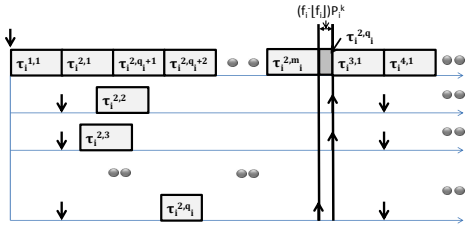


Figure 6. Implementation Considerations.

As mentioned earlier, the task *stretch* transform can be accomplished by the OS scheduler, and does not require any changes to existing code. The master string of each task τ_i with $C_i > T_i$ must be assigned its own processor core. This can be easily accomplished in many standard operating systems, for example, using the `sched_setaffinity` call which tells the scheduler to run a task on a particular core or processor. The parallel threads τ_i^{2s,q_i+1} through τ_i^{2s,m_i} need to be coalesced with the master string, and this can be realized by assigning these threads to the same core as the master string. The processor assignment for parallel threads $\tau_i^{2s,2}$ through τ_i^{2s,q_i} $\forall 1 \leq s \leq (s_i-1)/2$ can be determined by a task partitioning algorithm such as the one we describe in the next section. These threads can also be *pinned* to their cores using the `sched_setaffinity` call. The only parallel threads that need special treatment are τ_i^{2s,q_i}, since all other parallel threads fully execute on the same core. The thread τ_i^{2s,q_i} needs to have a *budget* of r_i^{2s} on its assigned processor, upon exhausting which it needs to be migrated to the core assigned for the master string (see Figure 6). This can be readily implemented in systems like Linux/RK [17] [18] [19] with support for semi-partitioning.

1: **Algorithm:** *Partitioned-FJ-DMS*
2: *input:* $\tau : \{\tau_1, \tau_2, ..., \tau n\}$
3: *output:*Processor Core Assignment
4: $\tau^{fdd} \leftarrow \{\}$
5: **for** $i \leftarrow 1$ to n **do**
6: $(\tau_i^{master}, \{\tau_i^{cd}\}) \leftarrow$ *StretchTask*(τ_i)
7: **if** $\{\tau^{cd}\} = \{\}$ **then**
8: ▷ *Task has* $C_i \leq T_i$
9: ▷ τ_i *is not fully stretched*
10: $\tau^{fdd} \leftarrow \tau^{fdd} \cup \tau_i^{master}$
11: **else**
12: Assign a processor exclusively for τ_i^{master}
13: $\tau^{fdd} \leftarrow \tau^{fdd} \cup \{\tau_i^{cd}\}$
14: **end if**
15: **end for**
16: Assign processors for τ^{fdd} using *FBB-FFD* ([21])
17: **return**

Figure 7. A Partitioned DMS Algorithm for Fork-Join task sets.

5 Fixed-Priority Partitioned Fork-Join Scheduling

Multiprocessor scheduling algorithms are traditionally classified as (i) partitioned approaches, where tasks are not allowed to migrate across processor cores, and (ii) global scheduling, where tasks are allowed to migrate across processor cores. In this work, we focus on *subtask*-level partitioned scheduling, where each *subtask* $\tau_i^{s,k}$ obtained from the task *stretch* transform of task τ_i is statically assigned to a processor core. We also restrict our attention to preemptive fixed-priority scheduling, specifically *deadline-monotonic* scheduling (DMS) [20].

As mentioned in the earlier section, our approach is to *stretch* the tasks whenever possible to avoid fork-join (FJ) structures that could potentially lead to unschedulability at low utilization levels. The constrained deadline subtasks generated by the *stretch* transform (if any) can be scheduled using a standard partitioned constrained-deadline task scheduling algorithm such as FBB-FFD [21]. Our partitioned DMS scheduling algorithm for fork-join task sets is provided in Figure 7. As can be seen, the master strings are allocated to their own individual cores, while the remaining constrained-deadline tasks are scheduled using *FBB-FFD*.

The FBB-FFD (Fisher Baruah Baker - First-Fit Decreasing) algorithm uses a variant of the First-Fit Decreasing bin-packing heuristic, wherein tasks are considered for allocation in the decreasing order of deadline-monotonic priorities, and each task is allocated to the first available core that satisfies a sufficient schedulability condition proposed in [21].

We will now provide key resource augmentation results for our fixed-priority scheduling algorithm in the context of FJ task sets leveraging the analysis of FBB-FFD from [21].

5.1 Analysis

The resource augmentation bound for the m-processor partitioned deadline-monotonic scheduling algorithm provided in Figure 7 is a *processor speedup factor* of 3.42. This implies that if a task set τ is feasible on m identical unit speed processors, then the *Partitioned-FJ-DMS* algorithm is guaranteed to successfully partition and schedule this task set on a platform comprising of m processors that are each 3.42 times as fast as the original. In order to derive this result, we use the notion of a *Demand Bound Function* (DBF) [22] for each task τ_i, which represents the largest cumulative execution requirement of all jobs that can be generated by τ_i to have both their arrival times and their deadlines within a contiguous interval of length t. For a task τ_i with a total computation requirement of C_i, period of T_i, and a deadline of D_i ($\leq T_i$) given by:

$$DBF(\tau_i, t) = \max\left(0, (\lfloor \frac{t - D_i}{T_i} \rfloor + 1)C_i\right) \quad (5)$$

The density of constrained-deadline task τ_i is $\delta_i = \frac{C_i}{D_i}$. The cumulative demand bound function δ_{sum} is defined as:

$$\delta_{sum} = \max_{t>0}\left(\frac{\sum_{i=1}^{n} DBF(\tau_i, t)}{t}\right) \quad (6)$$

The total utilization u_{sum} is given by $u_{sum} = \sum_{i=1}^{n} \frac{C_i}{T_i}$.

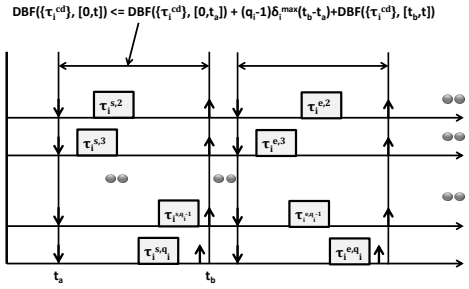
Lemma 6. *For any given implicit-deadline task τ_i with the maximum execution length less than or equal to the corresponding task period, the* stretch *transform does not affect the Demand Bound Function, i.e. if the stretched task is represented as $\tau_i^{stretched}$*

$$\forall \tau_i \; s.t. \; C_i \leq T_i, \forall t, DBF(\tau_i, t) = DBF(\tau_i^{stretched}, t)$$

Proof. The demand bound function for τ_i is given by Equation (5). Using the implicit-deadline nature of τ_i:

$$DBF(\tau_i, t) = \max(0, (\lfloor \frac{t}{T_i} \rfloor)C_i)$$

We consider τ_i with maximum execution length less than or equal to the corresponding task period, and hence τ_i will get fully *stretched*. Observe that the total execution requirement, period, and deadline of the stretched task $\tau_i^{stretched}$ are the same as that of the original task τ_i. Therefore, the demand bound function remains unchanged i.e. $DBF(\tau_i^{stretched}, t) = DBF(\tau_i, t)$. □



Proof. **Figure 8. Demand bound function of $\tau_i^{stretched}$.**

Corollary 7. *For any given implicit-deadline task τ_i with a maximum execution length less than or equal to the corresponding task period, the resulting* stretched *task $\tau_i^{stretched}$ from applying the* stretch *transform has a Demand Bound Function $DBF(\tau_i^{stretched}, t)$ that is bounded from above by $\frac{C_i t}{T_i - \eta_i}$, when $0 \leq \eta_i \leq T_i$, i.e.*

$$\forall t, DBF(\tau_i^{stretched}, t) \leq \frac{C_i t}{T_i - \eta_i}$$

Proof. The proof follows from Lemma 6. The Demand Bound Function of $\tau_i^{stretched}$ is given by:

$$\begin{aligned} DBF(\tau_i^{stretched}, t) &= DBF(\tau_i, t) = max(0, \lfloor \tfrac{t}{T_i} \rfloor C_i) \\ &\leq \tfrac{C_i t}{T_i} \\ &\leq \tfrac{C_i t}{T_i - \eta_i} \text{ (since } 0 \leq \eta_i \leq T_i) \end{aligned}$$

□

Lemma 8. *For any given implicit-deadline task τ_i with a maximum execution length greater than the corresponding task period, the resulting* stretched *task $\tau_i^{stretched}$ from applying the* stretch *transform has a Demand Bound Function $DBF(\tau_i^{stretched}, t)$ bounded from above by $\frac{C_i t}{T_i - \eta_i}$, i.e.*

$$\forall t, DBF(\tau_i^{stretched}, t) \leq \frac{C_i t}{T_i - \eta_i}$$

After the *stretch* transform (from Figure 5) is applied on a task τ_i with maximum execution length greater than the corresponding task period, the task $\tau_i^{stretched}$ can be represented as two components: (i) master string τ_i^{master}, and (ii) set of constrained-deadline sub tasks $\{\tau_i^{cd}\}$.

The demand bound function of $\tau_i^{stretched}$ over an interval t can be represented as:

$$DBF(\tau_i^{stretched}, t) \leq DBF(\tau_i^{master}, t) + DBF(\{\tau_i^{cd}\}, t) \quad (7)$$

The *stretch* transform ensures that the master string τ_i^{master} has an execution requirement of T_i, which equals the original task period T_i, therefore,

$$DBF(\tau_i^{stretched}, t) \leq t \quad (8)$$

The constrained-deadline subtasks in set $\{\tau_i^{cd}\}$ are offset (as shown in Figure 8) to ensure that subtasks belonging

to different parallel execution segments of τ_i are never simultaneously active. If the release offset of subtasks corresponding to the execution segment s of τ_i is represented as ϕ_i^s, and its deadline is D_i^s, then

$$\phi_i^s + D_i^s \leq \phi_i^s$$

From the DBF perspective, this property of subtasks in $\{\tau_i^{cd}\}$ ensures that the demand from $\{\tau_i^{cd}\}$ over any interval of length t does not exceed $\delta_i^{max}(q_i - 1)t$ (see Figure 8).

$$DBF(\{\tau_i^{cd}\}, t) \leq \delta_i^{max}(q_i - 1)t \leq \frac{1}{1 + f_i}(q_i - 1)t$$

(using Equation (4)) using Equations (3) & (2) with $D_i = T_i$ (using $P_i \geq 0$)

$$DBF(\{\tau_i^{cd}\}, t) \leq \frac{(q_i - 1)P_i}{P_i + T_i - \eta_i}t \leq \frac{(q_i - 1)P_i}{T_i - \eta_i}t \quad (9)$$

Using Equations (8) and (9) in (7), we get

$$\begin{aligned} DBF(\tau_i^{stretched}, t) &\leq t + \frac{(q_i - 1)P_i}{T_i - \eta_i}t \\ &\leq \frac{T_i - \eta_i + (q_i - 1)P_i}{T_i - \eta_i}t \\ &\leq \frac{f_i P_i + (m_i - \lfloor f_i \rfloor - 1)P_i}{T_i - \eta_i}t \\ &\leq \frac{m_i P_i}{T_i - \eta_i}t \\ &\leq \frac{C_i}{T_i - \eta_i}t \end{aligned}$$

□

For proving the resource augmentation bound for *Partitioned-FJ-DMS*, we will use the following result from Theorem 2 of [21],

Theorem ([21]): *Any constrained sporadic task system τ is successfully scheduled by FBB-FFD on m unit-capacity processors for*

$$m \geq \frac{\delta_{sum} + u_{sum} - \delta_{max}}{1 - \delta_{max}} \quad (10)$$

Using this, we can now provide the resource augmentation bound for Partitioned-FJ-DMS.

Theorem 9. *If any fork-join task set τ is feasible on m identical unit speed processors, then Partitioned-FJ-DMS is guaranteed to successfully allocate this task set on m identical processors that are each 3.42 times as fast as the original.*

Proof. The fork-join task set τ is feasible on m identical unit speed processors, which implies

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq m \quad (11)$$

Otherwise, τ is not feasible since over any time interval of length t, only mt units of computation cycles are available on an unit speed processor, while τ demands an utilization greater than m.

Consider the *minimum execution length* η_i for any task τ_i. It must be the case that

$$\forall 1 \leq i \leq n \;\; \eta_i \leq T_i \quad (12)$$

Otherwise, τ_i would be unschedulable on unit-speed processors from Proposition 1.

On a processor that is ν times faster, the *minimum execution length* η_i^ν is given by

$$\forall 1 \leq i \leq n \;\; \eta_i^\nu = \frac{\eta_i}{\nu} \leq \frac{T_i}{\nu} \quad (13)$$

There are two possible cases for each task τ_i in *Partitioned-FJ-DMS*,

Case 1. Task τ_i can be stretched into an implicit-deadline task on ν speed processors i.e. on a ν speed processor, the maximum execution length ($C_i^\nu = \frac{C_i}{\nu}$) of τ_i is less than or equal to its period T_i. In this scenario, $\tau_i^{stretched}$ is treated as a standard implicit-deadline task. Therefore, using Corollary 7, we get: $DBF(\tau_i^{stretched}, t) \leq \frac{C_i^\nu t}{(T_i - \eta_i^\nu)}$

Case 2. Task τ_i cannot be stretched into an implicit-deadline task on ν speed processors i.e. on a ν speed processor the maximum execution length ($C_i^\nu = \frac{C_i}{\nu}$) of τ_i is greater than the task period T_i. In this scenario, using Lemma 8, we see that $DBF(\tau_i^{stretched}, t) \leq \frac{C_i^\nu t}{(T_i - \eta_i^\nu)}$

Using the above cases for all the tasks in the task set,

$$\delta_{sum}^v = \max_{t>0}\left(\frac{\sum_{i=1}^{n} DBF(\tau_i^{stretched}, t)}{t}\right) \leq \sum_{i=1}^{n} \frac{C_i^\nu}{(T_i - \eta_i^\nu)} \quad (14)$$

From Equation (13),

$$\forall 1 \leq i \leq n \;\; \eta_i^\nu \leq \tfrac{T_i}{\nu} \Longrightarrow (T_i - \eta_i^\nu) \geq T_i(1 - \tfrac{1}{\nu})$$

Using this in Equation (14),

$$\begin{aligned} \delta_{sum}^\nu \leq \sum_{i=1}^{n} \frac{C_i^\nu}{T_i(1 - \frac{1}{\nu})} &\Longrightarrow \delta_{sum}^\nu \leq \tfrac{1}{\nu - 1} \sum_{i=1}^{n} \frac{C_i}{T_i} \\ &\Longrightarrow \delta_{sum}^\nu \leq \tfrac{1}{\nu - 1} u_{sum} \end{aligned}$$

Also, on ν speed processors,
the total utilization $u_{sum}^\nu = \frac{u_{sum}}{\nu}$ and $\delta_{max}^\nu = \frac{\delta_{max}}{\nu}$.

Using Equation (10), the task set is schedulable if

$$m \geq \tfrac{\delta_{sum}^\nu + u_{sum}^\nu - \delta_{max}^\nu}{1 - \delta_{max}^\nu}$$

Therefore, the task set is schedulable if

$$\Longrightarrow m \geq \frac{\frac{u_{sum}}{\nu - 1} + \frac{u_{sum}}{\nu} - \frac{\delta_{max}}{\nu}}{1 - \frac{\delta_{max}}{\nu}}$$

From Equation (11), the task set is schedulable if

$$m \geq \frac{\frac{m}{\nu - 1} + \frac{m}{\nu} - \frac{\delta_{max}}{\nu}}{1 - \frac{\delta_{max}}{\nu}}$$

This is an increasing function of δ_{max} for $m \geq \frac{\nu}{2}$

The density of any parallel thread with constrained deadlines is bounded from above in Equation (4) as

$$\forall f_i \geq 0 \Longrightarrow \delta_i^{max} = \frac{1}{1+f_i} \text{ and } f_i = \frac{T_i - \eta_i}{P_i}$$

$\Longrightarrow \frac{1}{1+f_i} = \frac{P_i}{T_i - \eta_i + P_i} \leq 1$ (since $\eta_i \leq T_i$ from Inequality (12)).

Therefore, when $m \geq \frac{\nu}{2}$, the schedulability is ensured if

$$m \geq \frac{\frac{m}{\nu-1} + \frac{m}{\nu} - \frac{1}{\nu}}{1 - \frac{1}{\nu}}$$

$$m(1 - \frac{1}{\nu}) \geq \frac{m}{\nu - 1} + \frac{m}{\nu} - \frac{1}{\nu}$$
$$\nu - \frac{1}{\nu - 1} \geq 3 - \frac{1}{m} \Longleftarrow \nu \geq (2 + \sqrt{2})$$

This holds good for all $m \geq \frac{\nu}{2}$, using $\nu \geq (2 + \sqrt{2}) \approx 3.42$ for all $m \geq 2$ processors.

For the uniprocessor case, the feasibility of task set τ implies that all the fork-join tasks can be stretched into implicit-deadline task sets. In this scenario, the task set is schedulable on the processor with speed $\nu \geq 3.42$ under RMS [13], since $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \leq (3.42) ln 2$

Hence, any feasible fork-join task set τ on m unit-speed processors is guaranteed to be scheduled by Partitioned-FJ-DMS on m processors, each with speed 3.42. □

6 Concluding Remarks

Sequential programming paradigms are ineffective in harnessing the processing capability of evolving massive multi-core systems. Established parallel programming paradigms such as *OpenMP* are promising candidates for extracting better performance from multi-core processors. Parallelism in *OpenMP* is achieved through basic *fork-join* task structures. In this paper, we have introduced the problem of scheduling *implicit-deadline periodic fork-join* task sets on multiprocessor systems. We illustrated that the worst-case schedulable utilization for *fork-join* task sets on multiprocessors can be slightly greater than and arbitrarily close to 100% (single processor) even when a large number of cores is available. Based on our observations from the worst-case task set characteristics, we define a task *stretch* transform that can be performed by the OS scheduler to avoid fork-join structures as much as possible. We also showed that the stretched task sets can be scheduled using partitioned preemptive fixed-priority multiprocessor scheduling algorithms, and provided the associated resource augmentation bound of 3.42. The task stretch transform is easily implementable on standard operating systems. There are many avenues for possible future work including synchronization protocols and energy management for *fork-join* task sets. Preempting parallel threads during execution may result in better system utilization but it also leads to overheads such as cache pollution and synchronization delays. Quantifying these overheads and a detailed comparison with different parallel scheduling models, such as gang scheduling and co-scheduling, is also part of our future work in this direction.

References

[1] "AMD sets the new standard for price, performance, and power for the datacenter," *AMD Press Release*, March 2010.

[2] "Single-chip cloud computer," *Intel Research*, Dec 2009.

[3] *OpenMP, http://openmp.org*.

[4] S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *OPERATIONS RESEARCH*, vol. 26, 1978.

[5] K. Ramamritham, J. Stankovic, and P. Shiah, "Efficient scheduling algorithms for real-time multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 184–194, 1990.

[6] A. Khemka and R. K. Shyamasundar, "An optimal multiprocessor real-time scheduling algorithm," *J. Parallel Distrib. Comput.*, vol. 43, no. 1, pp. 37–45, 1997.

[7] M. Dertouzos and A. Mok, "Multiprocessor online scheduling of hard-real-time tasks," *IEEE Transactions on Software Engineering*, vol. 15, pp. 1497–1506, 1989.

[8] J. Y.-T. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance Evaluation 2*, p. 237250, Dec 1982.

[9] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah, "A categorization of real-time multiprocessor scheduling problems and algorithms," in *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004.

[10] S. Collette, L. Cucu, and J. Goossens, "Integrating job parallelism in real-time scheduling theory," *Information Processing Letters*, vol. 106, no. 5, pp. 180 – 187, 2008.

[11] S. Kato and Y. Ishikawa, "Gang edf scheduling of parallel task systems," in *RTSS '09: Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 459–468.

[12] D. Lea, "A java fork/join framework," *In Proceedings of the ACM 2000 Java Grande Conference*, p. 3643, June 2000.

[13] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[14] C.-C. Han and K.-J. Lin, "Scheduling parallelizable jobs on multiprocessors," dec. 1989, pp. 59 –67.

[15] J. P. Lehoczky, "Fixed priority scheduling of periodic task sets with arbitrary deadlines," in *RTSS*, 1990, pp. 201–213.

[16] S. Funk, J. Goossens, and S. Baruah, "On-line scheduling on uniform multiprocessors," in *In Proceedings of the IEEE Real-Time Systems Symposium (December 2001), IEEE Computer*. Society Press, 2001, pp. 183–192.

[17] S. Oikawa and R. Rajkumar, "Portable RK: A portable resource kernel for guaranteed and enforced timing behavior," *RTAS, IEEE*, vol. 0, p. 111, 1999.

[18] K. Lakshmanan, R. Rajkumar, and J. Lehoczky, "Partitioned fixed-priority preemptive scheduling for multi-core processors," in *ECRTS '09: Proceedings of the 2009 21st Euromicro Conference on Real-Time Systems*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 239–248.

[19] S. Kato, R. Rajkumar, and Y. Ishikawa, "A loadable real-time scheduler suite for multicore platforms," *http://www.contrib.andrew.cmu.edu/ shinpei/papers/techrep09.pdf*, 2010.

[20] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings, "real-time scheduling: the deadline-monotonic approach," in *in Proc. IEEE Workshop on Real-Time Operating Systems and Software*, 1991, pp. 133–137.

[21] N. Fisher, S. Baruah, and T. P. Baker, "The partitioned scheduling of sporadic tasks according to static-priorities," in *ECRTS*. IEEE Computer Society, 2006, pp. 118–127.

[22] S. K. Baruah, A. K. Mok, and L. E. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *In Proceedings of the 11th Real-Time Systems Symposium*. IEEE Computer Society Press, 1990, pp. 182–190.