Limited-Preemption Scheduling on Multiprocessors

Bipasa Chattopadhyay  
bipasa@cs.unc.edu

Sanjoy Baruah  
baruah@cs.unc.edu

Department of Computer Science  
The University of North Carolina at Chapel Hill

ABSTRACT

Limited-preemption scheduling of sporadic task systems upon a multiprocessor platform is considered, when inter-processor migration is permitted. A schedulability test is derived by extending a prior schedulability test for fully-preemptive scheduling of sporadic task systems to the above-mentioned platform. We describe how the derived schedulability test can be used as a schedulability test for multiprocessor, multi-GPU systems with busy-waiting. Experimental evaluations are used to show the efficacy of the derived schedulability test in the context of the multiprocessor, multi-GPU system model.

1. INTRODUCTION

Several issues arise when implementing real-time scheduling algorithms. An important design choice is whether preemptions are enabled or disabled during task execution. This is not a trivial choice and many issues have to be considered.

In fully-preemptive scheduling (or simply preemptive scheduling) preemptions are enabled and a higher priority job can preempt a lower priority job at any time. The lower priority job can resume execution after all other higher priority jobs have completed execution. In non-preemptive scheduling, preemptions are disabled and a higher priority job may have to wait for a lower priority job to finish executing, before it can start executing. The latter delays the execution of a higher priority job and reduces schedulability. This is one of the main disadvantages of non-preemptive scheduling.

Run-time overheads in preemptive scheduling are higher when compared to non-preemptive scheduling. Each time a job gets preempted and resumes execution, run-time overheads for managing scheduling queues and reloading cache lines are incurred. This makes the worst-case execution cost of a task less predictable. This in turn makes it harder to estimate the number of preemptions a job may incur during its execution, resulting in inflated worst-case execution costs for tasks under preemptive scheduling. Further, when preemptions are enabled, a preemption may be forbidden if a job is executing in a critical section. Thus, non-trivial locking protocols for arbitrating access to shared resources are needed to augment preemptive scheduling. This increases the complexity of implementing preemptive scheduling algorithms and the associated run-time overheads. In contrast, arbitrating access to shared resources is trivial in non-preemptive scheduling on uniprocessors and requires simple synchronization techniques on multiprocessors. (Preemptions are disabled on a per-processor basis).

An alternative to fully-preemptive scheduling and non-preemptive scheduling is a restricted model of preemptive scheduling referred to as limited-preemptive scheduling. In limited-preemptive scheduling each job can execute preemptively on a processor until it needs to execute non-preemptively, possibly to access a shared resource. One of the objectives of this type of scheduling is to allow non-preemptive access to shared resources while still preserving the schedulability of the system.

Some examples of shared resources are shared memory and network bandwidth; more recently work has been done on incorporating Graphical Processing Units (GPUs) as a shared resource in real-time systems. GPUs are used widely for their ability to speed up graphical computations. General purpose computing on GPUs has allowed GPUs to be used in applications outside of graphics. GPUs can be incorporated in real-time systems as shared processing units and a task can use a GPU or CPU at different times during its execution.

This Research. In this paper we consider limited-preemption scheduling under the Global Earliest Deadline First (GEDF) scheduling algorithm for multiprocessors. We use the sporadic task model [5, 20] with an extension to incorporate non-preemptive executions of tasks. Our main contribution is a schedulability test for this platform. In addition, we show how to apply this schedulability test to a multiprocessor, multi-GPU system. In such systems the execution of a task on a GPU is non-preemptive. A task can execute preemptively on the processor and then request access to a GPU. After a request is made, one option is for the task to busy-wait non-preemptively on the processor until its non-preemptive execution on the GPU is complete. This can be thought of as a limited-preemption scheduling problem.

Organization. The sporadic task model we use is described in Section 2. We discuss related work in Section 3. The schedulability test and some of its properties are pre-
sented in Section 4. The multiprocessor multi-GPU system model is described in Section 5. Prior analysis approaches for such systems are also described in Section 5. A schedulability test for the multiprocessor multi-GPU system model is presented in Section 6. Our experimental evaluations are presented in Section 7. We summarize the paper in Section 8.

2. SYSTEM MODEL

A sporadic task system \( \tau = \tau_1, \ldots, \tau_n \) consists of \( n \) sporadic tasks \( \tau_i, 1 \leq i \leq n \). In the traditional model \([5, 20]\) each sporadic task \( \tau_i = (C_i, D_i, T_i) \) is characterized by a worst-case execution time \( C_i \), a relative deadline \( D_i \), and a minimum inter-arrival separation period parameter \( T_i \). Such a sporadic task generates a potentially infinite sequence of jobs, with successive job arrivals separated by at least \( T_i \) time units. Each job has a worst-case execution requirement equal to \( C_i \), that is fully-preemptive, and has an absolute deadline that occurs \( D_i \) time units after its arrival time. The utilization \( U_i \) of task \( \tau_i \) is \( \frac{C_i}{D_i} \).

In the model used in this paper each sporadic task \( \tau_i = (C_i, L_i, D_i, T_i) \) has an additional parameter \( L_i \) which represents the total length for which a job of a task may need to execute non-preemptively. The total execution requirement of such a task is \( C_i + L_i \), where \( C_i \) is fully-preemptive and \( L_i \) is non-preemptive. We assume that \( L_i \) may be non-contiguous. For each task \( \tau_i \), we let the length of its non-preemptive execution time be represented as an ordered set \( \{L_{i1}, L_{i2}, \ldots, L_{ik}\} \), where \( L_{ij}, j \in \{1, 2, \ldots, k\} \), represents the maximum length of the \( j^{th} \) longest contiguous non-preemptive execution of task \( \tau_i \), and \( \max(L_{ij}) = L_{ik} \). \( L_i \) is the sum of all non-preemptive execution lengths: \( L_i = \sum_{j=1}^{k} L_{ij} \). Such a sporadic task can be fully represented as, \( \tau_i = (C_i, \{L_{i1}, L_{i2}, \ldots, L_{ik}\}, D_i, T_i) \). Further, we assume that the preemptive and the non-preemptive execution of a task can be interleaved in any manner, i.e. we assume that we do not know the start and end points of the non-preemptive execution, \( L_{ij} \), of a task.

The \( D_i \) and \( T_i \) parameters denote the same task properties as in the traditional model. The utilization of a task \( U_i = \frac{C_i + L_i}{D_i} \). We henceforth refer to such tasks as limited-preemption sporadic tasks and \( n \) such tasks make up a limited-preemption sporadic task system \( \tau \). We denote \( U(\tau) = \sum_{i=1}^{n} U_i \) as the total utilization.

A traditional sporadic task system is said to be a constrained-deadline sporadic task system if for each task \( \tau_i \in \tau \), \( D_i \leq T_i \), and an implicit-deadline sporadic task system if \( D_i = T_i \). This definition is applicable to limited-preemption sporadic task systems as well. In this paper, we restrict our attention to constrained-deadline and implicit-deadline limited-preemption sporadic task systems.

The computing platform consists of a multiprocessor with \( m \) identical unit-capacity processors. Inter-processor migration is permitted. The scheduling algorithm is GEDF (Global Earliest Deadline First).

We derive a schedulability test for constrained-deadline and implicit-deadline limited-preemption sporadic task systems for the given computing platform. In the derivation of the schedulability test we use the concept of Demand Bound Function of a limited-preemption sporadic task \( \tau_i \).

**Definition 1.** For any interval of length \( t \), the demand bound function \( DBF(\tau, t) \) of a task \( \tau_i \) bounds the maximum cumulative execution requirement by jobs of \( \tau_i \) that both arrive in, and have deadlines within, any interval of length \( t \).

It has been shown in \([5]\), that \( DBF(\tau, t) \) of a sporadic task \( \tau_i \) is as follows:

\[
DBF(\tau_i, t) = \max(0, (\frac{t - D_i}{T_i}) + 1)C_i).
\]

By extension, \( DBF(\tau, t) \) of a limited-preemption sporadic task \( \tau_i \) is as follows:

\[
DBF(\tau_i, t) = \max(0, (\frac{t - D_i}{T_i}) + 1)(C_i + L_i)).
\]

3. RELATED WORK

A recent survey \([11]\) discusses and compares existing approaches for limited-preemption scheduling. The following approaches have been proposed in the literature: Preemption thresholds scheduling, Deferred preemptions scheduling and Fixed Preemption Points. The approach that we adopt in this work is deferred preemptions scheduling and is described below. Please refer \([11, \text{Section 2}]\) for a description of the other approaches.

Deferred preemptions scheduling was first introduced by Baruah \([2]\) under Earliest Deadline First (EDF) scheduling. In this approach, each task \( \tau_i \) can execute non-preemptively for a total length of say, \( q_i \).

It has been explained in \([11]\) that there are two ways in which non-preemptive regions can be implemented, floating, and activation-triggered.

A floating non-preemptive region can be defined by the programmer by inserting specific primitives in the task code that disable and enable preemption. However, the start and end time of this region is not specified. Thus, from an analysis perspective the non-preemptive region can be thought as “floating” in the code with a duration not exceeding some constant \( q_i \).

An activation-triggered non-preemptive region can be triggered by the arrival of a higher priority job say at time \( t \) and programmed by a timer to last exactly \( q_i \) time units, unless the currently executing job finishes earlier, after which preemption is enabled. Any further arrivals do not postpone the time \( t + q_i \) at which preemptions are enabled. Once a preemption takes place at or after time \( t + q_i \), a new higher-priority job can trigger another non-preemptive region.

Analysis in \([2]\) assumes floating non-preemptive regions and computes the longest non-preemptive execution \( q_i \) for each task \( \tau_i \) without compromising the feasibility of the system. Analysis in \([6]\) assumes the activation-triggered model and computes a function \( Q(t) \) that takes as input the time to the deadline of the executing job, and provides the amount of time for which such a job could execute non-preemptively when a new high-priority job arrives without compromising the feasibility of the system.

The analysis in \([2, 6, 21]\) was derived for uniprocessor EDF. We are unaware of any analysis under GEDF for multiprocessors for the system model described in Section 2. In this work we present schedulability analysis for multiprocessor GEDF scheduling for the limited-preemption sporadic task model described in Section 2. Our model assumes floating non-preemptive regions and in our analysis we determine whether, given the length of the longest non-preemptive region \( L_i \) for each task \( \tau_i \), the system is schedu-
4. SCHEDULABILITY TEST

The schedulability test described here extends the schedulability test described in [3] for fully-preemptive sporadic task systems to limited-preemption sporadic task systems. Note, that if \( L_i = 0 \) for all tasks \( \tau_i = (C_i, L_i, D_i, T_i) \), then the task system is a fully-preemptive sporadic task system. In the following discussion a task (task system) is assumed to be a limited-preemption sporadic task (task system) unless mentioned otherwise.

The general framework of how we derive the schedulability test is the same as described in [3]. We consider each task \( \tau_k \) separately; when considering a specific \( \tau_k \), we identify sufficient conditions for ensuring that \( \tau_k \) cannot miss any deadlines. To ensure that no deadlines are missed by any task in \( \tau \), these conditions are checked for each of the \( n \) tasks, \( \tau_1, \tau_2, \ldots, \tau_n \).

Consider any legal sequence of job requests of task system \( \tau \), for which GEDF misses a deadline. Suppose that a job of task \( \tau_k \) is the first to miss a deadline, and that this deadline miss occurs at time-instant \( t_d \). Let \( t_a \) denote this job’s arrival time: \( t_a = t_d - D_k \).

**Definition 2.** Let \( t_0 \) denote the latest time-instant at or before \( t_a \), at which at least one processor has finished executing all jobs that arrive before \( t_0 \) and have absolute deadlines at most \( t_d \).

Let \( t = t_d - t_0 \) and \( A_k = t_a - t_0 \). (Consequently, \( t \) is also \( A_k + D_k \).)

Note, that the definition of \( t_0 \) is the same as that in [3]. However, in [3] only jobs with absolute deadlines at most \( t_d \) are considered in the analysis. (This is valid in the analysis for fully-preemptive systems since jobs with absolute deadlines greater than \( t_d \) do not contribute to the deadline miss at time \( t_d \).) In limited-preemptive systems a job of a task \( \tau_k \) having absolute deadline greater than \( t_d \) can contribute to the deadline miss at time \( t_d \). In the following Lemmas it will become clear that a job of a task \( \tau_k \) with absolute deadline greater than \( t_d \) does not start executing in the interval \([t_0, t_a]\) but it can start executing in the interval \([t_a, t_d]\) and cause a deadline miss at time \( t_d \).

**Lemma 1.** A job of task \( \tau_k \) with absolute deadline greater than \( t_d \) does not start executing in the interval \([t_0, t_a]\).

**Proof:** Let us assume that a job of task \( \tau_k \) with absolute deadline greater than \( t_d \) starts executing in the interval \([t_0, t_a]\). Let this job of task \( \tau_k \) start executing at time-instant \( t_0 + \delta, 0 \leq \delta < (t_a - t_0) \). As per GEDF this implies that at least on one processor all jobs with absolute deadline at most \( t_d \) finished executing by time-instant \( t_0 + \delta \) and no job with absolute deadline at most \( t_d \) arrived at \( t_0 + \delta \). This makes \( t_0 + \delta + \epsilon, \epsilon \approx 0 \), the latest time-instant \( t_a \) at which at least one processor has finished executing all jobs that arrived before \( t_0 + \delta + \epsilon \) and with absolute deadline at most \( t_d \). Since \( t_0 + \delta + \epsilon > t_0 \) and by Definition 2 of time-instant \( t_0 \), we have a contradiction to our assumption. The lemma follows. □

**Lemma 2.** A job of task \( \tau_k \) with absolute deadline greater than \( t_d \) can start executing in the interval \([t_a, t_d]\).

**Figure 1:** The schedule generated by GEDF on two processors. CPU1 and CPU2, for jobs of tasks \( \tau_1, \tau_2, \tau_3 \) and \( \tau_k \) is shown. Note, that jobs \( J_k \) and \( J_1 \) are released at time \( t_a \) and job \( J_1 \) is released immediately after time \( t_a \). Jobs \( J_2 \) and \( J_3 \) have an absolute deadline greater than \( t_d \) and cause job \( J_k \) to experience non-preemptive blocking which leads to job \( J_k \) missing its deadline at time \( t_d \).

**Proof:** The scenario shown in Figure 1 can be observed under GEDF when non-preemptive execution is permitted. This scenario was first shown in [8]. Since we consider implicit-deadline and constrained-deadline tasks, only one job of task \( \tau_k \) with absolute deadline greater than \( t_d \) can execute non-preemptively, for at most \( L_i \) time units, and contribute to the deadline miss of task \( \tau_k \) at time \( t_d \). Note, that the preemptive execution of jobs with absolute deadline greater than \( t_d \) does not contribute to the deadline miss at time \( t_d \).

We now identify conditions necessary for a deadline miss to occur; i.e., for \( \tau_k \)'s job to execute for strictly less than \( C_k + L_k \) time units over \([t_a, t_d]\). In order for \( \tau_k \)'s job to execute for strictly less than \( C_k + L_k \) time units over \([t_a, t_d]\), it is necessary that all \( m \) processors execute jobs other than \( \tau_k \)'s for strictly more than \( D_k - (C_k + L_k) \) time units over \([t_a, t_d]\). Let us denote by \( \Gamma_k \) a collection of intervals, not necessarily contiguous, of cumulative length exactly \( D_k - (C_k + L_k) \) over \([t_a, t_d]\), during which all \( m \) processors are executing jobs other than \( \tau_k \)'s job in this GEDF schedule.

For each \( i, 1 \leq i \leq n \), let \( I(\tau_i) \) denote the contribution of \( \tau_i \) to the work done in this GEDF schedule during \([t_0, t_a) \cup \Gamma_k \). In order for a deadline miss to occur, it is necessary that the total amount of work that executes over \([t_0, t_a) \cup \Gamma_k \) satisfy the following condition:

\[
\sum_{\tau_i \in \tau} I(\tau_i) > m \times (A_k + D_k - (C_k + L_k)). \tag{3}
\]

This follows from the observation that all \( m \) processors are, by definition, completely busy executing this work over the \( A_k \) time units in the interval \([t_0, t_a) \), as well as the intervals in \( \Gamma_k \) of total length \( D_k - (C_k + L_k) \). Note, that the total length of the intervals in \([t_0, t_a) \cup \Gamma_k \) is equal to

\[
(A_k + D_k - (C_k + L_k)).
\]

Let us say that \( \tau_k \) has a carry-in job in this GEDF schedule if there is a job of \( \tau_k \) that arrives before \( t_0 \) and has not completed execution by \( t_0 \). In the following discussion, we compute upper bounds on \( I(\tau_i) \) if \( \tau_i \) has no carry-in job.
(this is denoted as $I_1(\tau_i)$), or if it does (denoted as $I_2(\tau_i)$).

We separately compute $B(\tau_i)$ which is the maximum non-preemptive blocking due to task $\tau_i$.

**Computing $I_1(\tau_i)$**. Let us consider the situation when all jobs of $\tau_i$ arrive in the interval $[t_0, t_d)$ and as a result task $\tau_i$ has no carry-in work. $I_1(\tau_i)$ is the total work contributed by all jobs of $\tau_i$ that arrive in the interval $[t_0, t_d)$ and have absolute deadlines at most $t_d$. (Later, we compute $B(\tau_i)$ to determine the maximum non-preemptive blocking due to a job of task $\tau_i$ with absolute deadline greater than $t_d$.)

Let us first consider a task $\tau_i$ such that $i \neq k$. In this case, it follows from Definition 1 of the demand bound function that the total work is at most $DBF(\tau_i, A_k + D_k)$; furthermore this total contribution cannot exceed the total length of the intervals in $[t_0, t_d) \cup \Gamma_k$. Hence, the contribution of $\tau_i$ to the total work that must be done by GEDF over $[t_0, t_d) \cup \Gamma_k$ is at most

$$\min(DBF(\tau_i, A_k + D_k), A_k + D_k - (C_k + L_k)). \quad (4)$$

Now, consider the case $i = k$. In this case, the job of $\tau_k$ arriving at time-instant $t_0$ does not contribute to the work that must be done by GEDF over $[t_0, t_d) \cup \Gamma_k$; hence, its execution requirement must be subtracted. Also, this contribution cannot exceed the length of the interval $[t_0, t_d)$ i.e., $A_k$.

Putting these pieces together, we get the following bound on the contribution of $\tau_i$ to the total work that must be done by GEDF over $[t_0, t_d) \cup \Gamma_k$:

$$I_1(\tau_i) \overset{\text{def}}{=} \begin{cases} \min(DBF(\tau_i, A_k + D_k), A_k + D_k - (C_k + L_k)), & \text{if } i \neq k \\ \min(DBF(\tau_i, A_k + D_k) - (C_k + L_k), A_k), & \text{if } i = k. \end{cases} \quad (5)$$

**Computing $I_2(\tau_i)$**. Let us now consider the situation when $\tau_i$ arrives before $t_0$, and hence potentially carries in some work in the interval $[t_0, t_d)$.

It was shown in [7] that the total work of a sporadic task $\tau_i$ can be upper-bounded by considering the scenario in which some job of $\tau_i$ has a deadline at $t_d$, and all jobs of $\tau_i$ execute at the very end of their scheduling windows.

Let the demand bound function $DBF'(\tau_i, t)$ denote the maximum amount of work that can be contributed by $\tau_i$ over a contiguous interval of length $t$. The definition of $DBF'(\tau_i, t)$ in [3] for sporadic tasks can be extended to limited-preemption sporadic tasks as follows:

$$DBF'(\tau_i, t) = \left[ \frac{t}{T_i} \right] \times (C_i + L_i) + \min(C_i + L_i, t \mod T_i). \quad (6)$$

In computing $\tau_i$’s contribution to the total amount of work that must execute over $[t_0, t_d) \cup \Gamma_k$, let us first consider $i \neq k$. In this case, it follows from the definition of demand bound function $DBF'$ that the upper bound on the amount of work contributed by task $\tau_i$ is $DBF'(\tau_i, A_k + D_k)$; furthermore, this contribution cannot exceed the total length of the intervals in $[t_0, t_d) \cup \Gamma_k$. Hence, the contribution of $\tau_i$ to the total work that must be done by GEDF over $[t_0, t_d) \cup \Gamma_k$ is at most:

$$\min(DBF'(\tau_i, A_k + D_k), A_k + D_k - (C_k + L_k)). \quad (7)$$

Now, consider the case $i = k$. In this case, we know that a job of task $\tau_k$ has a deadline at $t_d$ and that this job of $\tau_k$ arrives at time-instant $t_0$ and does not contribute to the work that must be done by GEDF over $[t_0, t_d) \cup \Gamma_k$; hence, its execution requirement must be subtracted. Also, this contribution cannot exceed the length of the interval $[t_0, t_d)$ i.e., $A_k$.

From the discussion above we get the following bound on the contribution of $\tau_i$ to the total work that must be done by GEDF over $[t_0, t_d) \cup \Gamma_k$:

$$I_2(\tau_i) \overset{\text{def}}{=} \begin{cases} \min(DBF'(\tau_i, A_k + D_k), A_k + D_k - (C_k + L_k)), & \text{if } i \neq k \\ \min(DBF'(\tau_i, A_k + D_k) - (C_k + L_k), A_k), & \text{if } i = k. \end{cases} \quad (8)$$

**Computing $B(\tau_i)$**. The maximum non-preemptive blocking due to task $\tau_i$ over $[t_0, t_d) \cup \Gamma_k$ is caused by a job of task $\tau_i$ with absolute deadline greater than $t_d$. Since we consider constrained-deadline and implicit-deadline tasks there can be only one such job of task $\tau_i$ in the intervals in $[t_0, t_d) \cup \Gamma_k$. Note, that if this job arrives before $t_0$ then it cannot be considered a carry-in job and $I_2(\tau_i)$ upper bounds its contribution. Therefore, in computing $B(\tau_i)$ we only need to account for the maximum non-preemptive blocking due to a job of task $\tau_i$ that arrives in the interval $[t_0, t_d)$.

We know from Lemmas 1 and 2 that a job of task $\tau_i$ that arrives in the interval $[t_0, t_d)$ with an absolute deadline greater than $t_d$ can start executing only in the interval $[t_0, t_d)$, of length $D_k$. Therefore, the maximum non-preemptive blocking due to task $\tau_i$ is as follows:

$$B(\tau_i) \overset{\text{def}}{=} \begin{cases} \min(L_i, D_k), & \text{if } i \neq k \\ 0, & \text{if } i = k. \end{cases} \quad (9)$$

**Putting the pieces together**. Let us now compute the total amount of carry-in work over the intervals in $[t_0, t_d) \cup \Gamma_k$. By Definition 2 of $t_0$, at most $m$ tasks have not completed execution at time-instant $t_0$. Consequently, at most $m$ tasks can contribute an amount $I_2(\tau_j)$ and the remaining $(n - m)$ tasks must contribute $I_1(\tau_i)$. However, as per Definition 2, on at least one processor all tasks with absolute deadline at most $t_d$ have completed execution before $t_0$. Thus, on at least one processor the carry-in work is contributed by a job of a task $\tau_j$ with absolute deadline greater than $t_d$. Further, it can be shown that $D_j > t$ and that the maximum amount of carry-in work that task $\tau_j$ can contribute is the length of its non-preemptive execution $L_j$. Hence, the total amount of carry-in work can be written as:

$$\sum_{i = 1}^{m} I_2(\tau_i) + \max\{L_j\}_{D_j > t}, \quad (10)$$

where $\sum_{i = 1}^{m} I_2(\tau_i)$ returns the sum of the $(m - 1)$ largest values of $I_2(\tau_i), 1 \leq i \leq n$.

Note, that Equation 10 upper bounds the maximum carry-in work and the maximum non-preemptive blocking due to jobs that arrive before $t_0$. We now compute the maximum non-preemptive blocking due to jobs that arrive in the interval $[t_0, t_d)$. From
Equation 9 we know that the total non-preemptive blocking caused by such jobs can be expressed as $\sum_{\tau_i \in \tau} B(\tau_i)$. However, this can be pessimistic for the following reason.

We know that a job of task $\tau_k$ arrives at time $t_a$ and has a deadline at $t_d$, therefore on at least one processor jobs with absolute deadline greater than $t_d$ will not start executing in the interval $[t_a, t_d]$ until the job of task $\tau_k$ has met its deadline. However, as per our assumption the job of task $\tau_k$ misses its deadline. Therefore, the total non-preemptive blocking is at most $(m-1) \times D_k$.

Let $B_n$ be the maximum non-preemptive blocking due to jobs of all tasks that arrive in the interval $[t_a, t_d]$. $B_n$ is as follows:

$$B_n = \min \left( \sum_{\tau_i \in \tau} B(\tau_i), (m-1) \times D_k \right)$$

Let us denote by $I_{Diff}(\tau_i)$ the difference between $I_2(\tau_i)$ and $I_1(\tau_i)$:

$$I_{Diff}(\tau_i) = I_2(\tau_i) - I_1(\tau_i).$$

Condition (3) may be re-written as follows:

$$\sum_{\tau_i \in \tau} I_1(\tau_i) + \sum_{(m-1)\max} I_{Diff}(\tau_i) + \max \{L_j\} D_{j,>t} + B_n$$

$$> m \times (A_k + D_k - (C_k + L_k)).$$

Observe that all the terms in Condition (13) above are completely defined for a given task system, once a value is chosen for $A_k$. Hence for a deadline miss of $\tau_k$ to occur, there must exist some $A_k$ such that Condition (13) is satisfied. Conversely, in order for all deadlines of $\tau_k$ to be met it is sufficient that Condition (13) be violated for all values of $A_k$. Theorem 1 follows immediately:

**Theorem 1.** Task system $\tau$ is GEDF-schedulable upon $m$ unit-capacity processors if for all tasks $\tau_i \in \tau$ and all $A_k \geq 0$,  

$$\sum_{\tau_i \in \tau} I_1(\tau_i) + \sum_{(m-1)\max} I_{Diff}(\tau_i) + \max \{L_j\} D_{j,>t} + B_n$$

$$\leq m \times (A_k + D_k - (C_k + L_k))$$

where $I_1(\tau_i)$, $I_{Diff}(\tau_i)$, and $B_n$ are as defined in Equations 5, 12, and 11 respectively.

4.1 Properties

**Run-time Complexity.** For a given $\tau_k$ and $A_k$, it is easy to see that Condition (14) can be evaluated in time linear in $n$, the number of tasks in the task system:

- Compute $I_1(\tau_i)$, $I_2(\tau_i)$, $B(\tau_i)$ and $I_{Diff}(\tau_i)$ for each task $\tau_i$ - total time is $O(n)$.
- Use linear-time selection [9] on $\{I_{Diff}(\tau_1), I_{Diff}(\tau_2), \ldots, I_{Diff}(\tau_n)\}$ to determine the $(m-1)$ tasks that contribute to the second sum on the RHS.
- Compute $\max \{L_j\} D_{j,>t}$ and $B_n$ - total time is $O(n)$.

We now determine the values of $A_k$. First, we derive the range for the values of $A_k$ and then determine the individual values of $A_k$ for which Condition (14) must be verified.

**Theorem 2.** If Condition (14) is to be violated for any $A_k$, then it is violated for some $A_k$ satisfying the condition below:

$$A_k \leq \frac{S_{\Sigma} - D_k(m - U(\tau)) + \sum_{i}(T_i - D_i)U_i + L_i + m(C_k + L_k)}{m - U(\tau)}$$

where $S_{\Sigma}$ denotes the sum of the $m$ largest $(C_i + L_i)$.

**Proof:** It can be seen that $I_1(\tau_i) \leq DBF(\tau_i, A_k + D_k)$, $I_2(\tau_i) \leq DBF(\tau_i, A_k + D_k) + (C_i + L_i)$, and $B_n \leq \sum_{\tau_i \in \tau} L_i$.

From this, it can be shown that the LHS of Condition (14) is $\leq S_{\Sigma} + \sum_{\tau_i \in \tau} DBF(\tau_i, A_k + D_k) + L_i$.

For this to exceed the RHS of Condition (14), it is necessary that:

$$S_{\Sigma} + \sum_{\tau_i \in \tau} (DBF(\tau_i, A_k + D_k) + L_i) > m(A_k + D_k - (C_k + L_k))$$

$$\Rightarrow S_{\Sigma} + (A_k + D_k)U(\tau) + \sum_{i}(T_i - D_i)U_i + L_i > m(A_k + D_k - (C_k + L_k))$$

(bounding DBF using the technique in [5])

$$\Rightarrow S_{\Sigma} + D_kU(\tau) + \sum_{i}(T_i - D_i)U_i + L_i + m(D_k - (C_k + L_k)) > A_k(m - U(\tau))$$

$$\Rightarrow A_k \leq \frac{S_{\Sigma} - D_k(m - U(\tau)) + \sum_{i}(T_i - D_i)U_i + L_i + m(C_k + L_k)}{m - U(\tau)}$$

The theorem follows. □

Further, we only need to consider the non-negative values of $A_k$. It can also be shown that Condition (14) need only be tested at those values of $A_k$ at which $DBF(\tau_i, A_k + D_k)$ changes for some $\tau_i$. To be specific, it is shown in [10, p. 82] that it is sufficient to test only those values of $A_k$ that satisfy:

$$A_k = D_i - D_k + j \times T_i$$

for some $\tau_i \in \tau$ and some $j \in \{0, 1, 2, \ldots\}$. The bound on the maximum $A_k$ grows exponentially as $m - U(\tau)$ approaches 0. However, for values of $U(\tau)$ bounded by a constant strictly less than the number of processors $m$ the following property holds:

**Property 1.** The condition in Theorem 1 can be tested in time pseudo-polynomial in the task parameters, for all task systems $\tau$ for which $U(\tau)$ is bounded by a constant strictly less than the number of processors $m$.

**Sufficient/Necessary.** Theorem 1 is an extension to the schedulability test in [3]. The latter was derived for fully-preemptive sporadic task systems. The schedulability test in [3] has been shown to be a generalization of the uniprocessor schedulability test in [5]. It is sufficient and necessary when $m = 1$ and sufficient but not necessary when $m > 1$.

A limited-preemption sporadic task system $\tau$ is a fully-preemptive sporadic task system if $L_i = 0$ for all tasks $\tau_i \in \tau$. It can be shown that, if $L_i = 0$ for all tasks $\tau_i \in \tau$ then the schedulability test in Theorem 1 reduces to the schedulability test in [3]. This leads to the following property.
Property 2. For fully-preemptive sporadic task systems, the schedulability test in Theorem 1 is sufficient and necessary when \( m = 1 \) and sufficient but not necessary when \( m > 1 \).

We now show that the above property continues to hold for limited-preemption sporadic task systems, such that \( L_i > 0 \) for some task \( \tau_i \in \tau \).

Lemma 3. For a task \( \tau_k \) and for \( m = 1 \) processor, Condition (17) determines whether the exact processor demand over an interval \( t \) of length \( A_k + D_k \) is at most the length of the interval.

\[
(\sum \tau_i \in \tau I_i(\tau_i) + (C_k + L_k)) + \max \{L_i\}_{D_i > t} \leq A_k + D_k \tag{17}
\]

Proof: As per Equation (5) the first term in the LHS of Condition (17) is equal to the processor demand in an interval \( t \) by jobs arriving in this interval and having deadlines within this interval. By Definition 2 of \( T_0 \) no task \( \tau_i \) with \( D_i < t \) can be active at time-instant \( t_0 \) and one or many tasks with \( D_i > t \) can be active at time-instant \( t_0 \). Under EDF and \( m = 1 \) only one such task can execute non-preemptively in the interval \( t \). The second term in the LHS of Condition (17) accounts for the maximum amount of non-preemptive blocking possible due to limited-preemptivity. Thus, the LHS of Condition (17) gives the exact processor demand over some interval \( t \).

Observe that Condition (14) reduces to Condition (17) for \( m = 1 \) by adding \((C_k + L_k)\) to both LHS and RHS. Thus, by Lemma 3, Condition (14) determines whether the exact processor demand over an interval \( t \) is at most the length of the interval for \( m = 1 \). For \( m > 1 \), Condition (14) upper bounds the amount of work carried in on \( m - 1 \) processors (Refer Equation 8). This leads to the following property.

Property 3. For limited-preemption sporadic task systems, such that \( L_i > 0 \) for some task \( \tau_i \in \tau \), the schedulability test in Theorem 1 is sufficient and necessary when \( m = 1 \) and sufficient but not necessary when \( m > 1 \).

Sustainability. The concept of sustainability was introduced and explained in [4]. A schedulability test for the task model described in Section 2 is sustainable if any task system deemed schedulable by the schedulability test remains schedulable when the parameters of one or more individual job[s] are changed in any, some, or all of the following ways: (i) decreased preemptive execution time, \( C_i \), (ii) decreased non-preemptive execution time, \( L_i \), (iii) larger inter-arrival times, \( T_i \), (iv) larger relative deadlines, \( D_i \).

Sustainability can also be defined with respect to the scheduling policy. A scheduling policy for the task model described in Section 2 is sustainable if a scheduling policy meets all deadlines when scheduling any collection of jobs generated by a task system and continues to meet the deadlines when the parameters of one or more individual job[s] are changed in any, some, or all of the following ways: (i) decreased preemptive execution time, \( C_i \), (ii) decreased non-preemptive execution time, \( L_i \), (iii) larger inter-arrival times, \( T_i \), (iv) larger relative deadlines, \( D_i \).

By definition, if a scheduling policy \( A \) is sustainable then any sufficient schedulability test for \( A \) is sustainable. If a scheduling policy is not sustainable then every schedulability test for \( A \) is not necessarily sustainable. It is shown in [1, Observation 1,2] that GEDF scheduling of fully-preemptive sporadic task systems is sustainable with respect to decreased execution requirements and larger inter-arrival times. To the best of our knowledge, and as stated in [1], it is yet unknown whether GEDF scheduling of fully-preemptive sporadic task systems is sustainable with respect to larger relative deadlines.

Also, to the best of our knowledge, it is unknown whether GEDF scheduling of limited-preemption sporadic task systems is sustainable with respect to decreased execution times (preemptive and non-preemptive), larger inter-arrival times, and larger relative deadlines. The schedulability test in Theorem 1, however, is sustainable with respect to these parameters. Due to space constraints we present the proof sketch to show sustainability with respect to decreased non-preemptive execution times and larger relative deadlines.

Similar arguments can be made for showing sustainability with respect to decreased preemptive execution times and larger inter-arrival times.

Lemma 4. The schedulability test in Theorem 1 is sustainable with respect to decreased non-preemptive execution times.

Proof Sketch: In our analysis, for each interval of length \( t \) where a deadline can be missed, we consider the affect of the maximum time by which the execution of a job of each task can be delayed due to non-preemptivity. If the length of the non-preemptive execution time of a job decreases then the maximum time by which the execution of a job of each task can be delayed due to non-preemptivity does not increase. Thus, if our schedulability test deems a task system to be schedulable then it will continue to be schedulable if the length of the non-preemptive execution time of a job decreases.

Lemma 5. The schedulability test in Theorem 1 is sustainable with respect to larger relative deadlines.

Proof Sketch: Let us consider a task system \( \tau \) that was schedulable as per our schedulability test. Now assume that the relative deadline of a job of a task \( \tau_i \) was increased and a deadline miss occurs.

In our schedulability test we determine the maximum cumulative execution requirement for all tasks within an interval of length \( t \). For each task we assume that it has a deadline at the end of the interval \( t \) and ensure that all tasks meet their deadline. All possible values of \( t \) where a deadline miss can occur are considered. Note, that increasing the relative deadline of a job of a task \( \tau_i \) (while still ensuring \( D_i \leq T_i \)) does not change the maximum cumulative execution requirement of task \( \tau_i \) over any interval of length \( t \). Thus, if a job of \( \tau_i \) misses a deadline as per our assumption then our schedulability test would have failed. Thus, by contradiction the Lemma follows.

The schedulability test in Theorem 1 is sustainable under several different parameter relaxations. Note, that this is also true under any combination of different parameter relaxations. In our demand based analysis we essentially
ensure that a task system is schedulable over any interval of length \( t \) if the total execution demand is at most the maximum cumulative execution demand. Thus, we obtain the following property:

Property 4. The schedulability test in Theorem 1 is sustainable with respect to (i) decreased preemptive execution time, \( C_i \), (ii) decreased non-preemptive execution time, \( L_i \), (iii) larger inter-arrival times, \( T_i \), (iv) larger relative deadlines, \( D_i \).

5. MULTI-GPU SYSTEM MODEL

We have described a schedulability test in Theorem 1 and derived and discussed some of its properties. In Section 6 we show how Theorem 1 can be used as a schedulability test for a multiprocessor multi-GPU system. First, we describe the multi-GPU system model.

A job of a task running on a processor can initiate execution on a GPU. Several aspects of GPU program execution are described in [15]. A GPU has an execution engine (EE) and one or two DMA copy engines (CEs). A copy engine transmits data between system memory and GPU memory, and an execution engine performs some computation on a given data. A possible sequence of events when a job executes on a GPU is described in [15] and is as follows. First, the copy engine copies data from the system memory to the GPU memory, followed by the computation on the execution engine. Finally, the copy engine copies the results from the GPU memory back to the system memory. Further, GPU operations on the various engines are non-preemptive.

Let us assume that each task \( \tau_i \) makes \( k \) requests to the GPU represented as an ordered set, \( \{G_{i1}, G_{i2}, \ldots, G_{ik}\} \), where \( G_{ij} > 0, j \in \{1, 2, \ldots, k\} \), represents the \( j \)th longest GPU execution. Each request \( G_{ij} \) can either be a request to a copy engine, execution engine, or a combination of requests to the copy engine and execution engine. Let \( G_i \) be the sum of the execution length of all the GPU requests a task makes: \( G_i = \sum_{j=1}^{k} G_{ij} \). If a task does not make any GPU requests, \( G_i = 0 \).

Once a job running on a processor initiates execution on a GPU it can either self-suspend or it can busy-wait on the processor until the GPU execution is complete. When a job self-suspends, the processor is available for other jobs to execute. This is preferable because the job wastes processor cycles when busy-waiting. However, busy-waiting benefits from lower overheads (compared to the cost of suspending and resuming tasks). Therefore, busy-waiting is preferable only if for all tasks the non-preemptive critical section on the GPU is short. Empirical results obtained in [10, Chapter 7] show that busy-waiting implemented as spin-based locks is useful if a task uses a resource for at most a few microseconds.

In the case of busy-waiting, a job can busy-wait preemptively, i.e., it busy-waits until a job with a higher priority preempts it on the CPU while it continues to execute non-preemptively on the GPU. This is different from self-suspension only because a lower or equal priority job can start executing on the CPU after a job self-suspends.

An alternative to preemptive busy-waiting is non-preemptive busy-waiting. In this case a job busy-waits non-preemptively on a processor until it completes its GPU execution. This is illustrated with the help of Figure 2. In Figure 2, job \( J_1 \) arrives at time \( a_1 \) and has a deadline at time \( d_1 \), and job \( J_2 \) arrives at time \( a_2 \) and has a deadline at time \( d_2 \). Job \( J_1 \) starts executing on the GPU at time \( a_1 \) while busy-waiting non-preemptively on the CPU. Job \( J_2 \) with a shorter deadline, thus higher priority, has to wait for job \( J_1 \) to complete its GPU execution before it can preempt it and start executing on the CPU. Non-preemptive busy-waiting has the least run-time overheads when compared to preemptive busy-waiting and self-suspensions.

Our analysis focuses on the multi-GPU system model with non-preemptive busy-waiting. Tasks under this system model can be modeled as limited-preemption sporadic tasks. For each limited-preemption sporadic task \( \tau_i \in \tau \), \( L_{ij} \) is set to the length of the \( j \)th longest non-preemptive critical section on a GPU.

In our system we use a simple synchronization approach (a version of the locking protocol described in [8] for non-nested, short resource requests) to control access to the GPUs. For each GPU we assume there is a spin-lock controlling access to it and for each spin-lock there is a corresponding FIFO-ordered wait queue. If a job that requests for a GPU can acquire any spin-lock it can access the GPU protected by that spin-lock, otherwise it is assigned to a wait queue. A shortest queue mechanism is used to determine which wait queue a job is assigned to. Once a job is assigned to a wait queue, it waits on this queue until it can acquire the corresponding spin-lock. For a task \( \tau_i \) the length of its non-preemptive execution \( L_{ij} \) is equal to the sum of the execution requirement on a GPU \( G_{ij} \), and the amount of time it must wait to access a GPU. The latter, is computed in Section 6.

We assume that all GPUs are identical, i.e., the execution requirement \( G_{ij} \) of a task is the same irrespective of the GPU on which it executes. The number of GPUs is denoted by \( g \). The number of identical, unit-capacity processors continues to be denoted by \( m \).

5.1 Prior GPU Analysis

Several GPU management frameworks have been designed and implemented including TimeGraph [17], RGEM [16], Gdev [18]. Analysis of the RGEM framework includes blocking analysis that is incorporated into classical fixed-priority scheduling response-time analysis for multiprocessors. Elliott et al. designed and implemented GPUSync [15]. In [14] Elliott et al. describe and present the blocking analysis of a \( k \)-exclusion locking protocol for globally-scheduled job-level static-priority systems for self-suspending sporadic tasks. The term \( k \)-exclusion means that there are \( k \) copies of some resource, for example GPUs. Recent analysis in [19] provides response-time analysis for self-suspending sporadic tasks under rate monotonic scheduling for multiprocessors. Blocking analysis is done using a linear programming technique. In [12] a schedulability test for self-suspending tasks under GEDF scheduling is described. This work was not aimed at a GPU platform but can be extended to GPUs.
In our analysis we assume GEDF scheduling. We present a schedulability test for the non-preemptive busy-waiting multi-GPU system model using the results of the schedulability analysis obtained in Theorem 1.

6. MULTI-GPU SCHEDULABILITY TEST

Given the execution length of each GPU request $G_{ij}$ of a task $\tau_i$ we first need to determine the length of the non-preemptive execution $L_{ij}$. For this we have to determine the amount of time a task $\tau_i$ may have to wait to access one of the $g$ GPUs. This can be computed by upper bounding the number of GPU requests at any time. The following Lemmas follow directly from the results obtained in [8].

**Lemma 6.** There can be at most $m$ GPU requests at any time, one per processor.

**Proof:** A job of a task can request for a GPU only when it is executing on some processor. If a job executing on a processor requests for a GPU then it busy-waits non-preemptively until the request is serviced by the GPU. Thus, no other job of any task can execute on this processor and as a result no other GPU request can be made from this processor. Since there are $m$ processors there can be at most $m$ GPU requests at any time. □

**Lemma 7.** The length of any wait queue is at most \[ \left\lceil \frac{m}{g} \right\rceil - 1. \]

**Proof:** From Lemma 6 there can be at most $m$ GPU requests at any time. If a shortest queue mechanism is used to distribute these requests across the $g$ wait queues, one for each GPU, then the number of requests on each wait queue is \[ \left\lceil \frac{m}{g} \right\rceil. \] Of these requests, one is satisfied by the GPU. Therefore, the length of a wait queue is at most \[ \left\lceil \frac{m}{g} \right\rceil - 1. \] Note, that when $g = m$ the length of any wait queue is 0. □

It has been shown in [22] that FIFO-ordered spin-locks offer strong progress guarantees and is an effective mechanism for non-preemptive busy-waiting.

Let $W_{ij}$ be the amount of time a job of a task $\tau_i$ has to wait upon making the $j^{th}$ request to the GPU.

**Theorem 3.** The schedulability test in Theorem 1 can be applied to the system model under consideration if for each task $\tau_i$ and $j^{th}$ GPU request $G_{ij}$, $L_{ij} = G_{ij} + W_{ij}$, where:

\[
W_{ij} = \frac{r}{\left\lceil \frac{m}{g} \right\rceil - 1} G_{rs},
\]

\[r \in \{1, \ldots, i-1, i+1, \ldots, n\}, s \in \{1, \ldots, k\}.\] (18)

**Proof:** For a task $\tau_i$ the length of its non-preemptive execution $L_{ij}$ is equal to the sum of the execution time of its $j^{th}$ GPU request, $G_{ij}$, and the amount of time it must wait on a wait queue, $W_{ij}$. From Lemma 7, the length of any wait queue is at most \[ \left\lceil \frac{m}{g} \right\rceil - 1. \] Since we consider implicit-deadline and constrained-deadline task systems, two jobs of the same task cannot be on any of the $g$ wait queues at the same time. Thus, Equation (18) upper bounds the term $W_{ij}$. □

Note, that if $G_i = 0$, then a task $\tau_i$ does not make any GPU requests. Therefore, it does not have to wait to access the GPU and $L_i = 0$. Also, when $g = m$, $L_{ij} = G_{ij}$ in the multi-GPU system model with non-preemptive busy-waiting.

With non-preemptive busy-waiting we do not get any analytical benefits when $g > m$. However, we can get analytical benefits when $g > m$ for the self-suspending and preemptive busy-waiting system models discussed in Section 5. Also, in our system model we assume that the GPU can execute only one task at a time. However, the GPU copy engine (CE) and GPU execution engine (EE) can in fact non-preemptively execute jobs of two different tasks at the same time. See Figure 3. This parallelism can be exploited under non-preemptive busy waiting when $g < m$, and in the self-suspending and preemptive busy-waiting system model. We leave this as future work.

![Figure 3: Scheduling scenario for m = 1 and g = 1 under preemptive busy-waiting. Jobs J1 and J2 execute in parallel on the GPU CE and GPU EE effectively reducing the total time spent executing on the GPU.](image)

7. EXPERIMENTAL EVALUATION

We perform experiments to determine the effectiveness of our schedulability test in the context of the multiprocessor, multi-GPU system model. We randomly generated task sets and determined the percentage of task sets that were schedulable by our schedulability test. Each task set was generated as follows. The UUnifast-Discard algorithm described in [13] was used to generate $n$ task utilizations, $\{u_1 \ldots u_n\}$, of some total utilization, $u(\tau)$. The period $T_i$, for each task was generated according to a log-uniform distribution in the range 10ms to 1000ms. All task periods were set to integer values by rounding down from any non-integer value. The total execution requirement of a task without using a GPU was set to $u_i \times T_i$. A portion $g_i$, was chosen from a uniform distribution in the range $[0, u_i \times T_i]$ to denote the execution of a task on the GPU. We assumed that a task exploits the parallelism provided by a GPU and executes in lesser time on a GPU when compared to a CPU. Speed up $SP$, is the ratio of the execution time on the CPU and the execution time on the GPU. Thus, $G_i = g_i/SP$. For simplicity we assume that a job of a task makes one request to the GPU. The remaining execution time, $(u_i \times T_i) - g_i$, was set to $C_i$. $L_i$ was computed from $G_i$. If $G_i = 0$ then $L_i = 0$. Else, $L_i$ was computed using Equation (18) for $k = 1$. Note, that $L_i = G_i$ when $m = g$. Task deadline $D_i$, was set equal to $T_i$ for implicit-deadline task systems, and was chosen from a uniform distribution in the range $[u_i \times T_i, T_i]$ for constrained-deadline task systems. Utilization $u_i$, generated above is referred to as the effective utilization of a task and $u(\tau)$ as the total effective utilization of task system $\tau$. The actual utilization
of a task is $U_i = \frac{C_i}{T_i}$ and the actual total utilization is $U(\tau) = \sum_{i=1}^{n} U_i(\tau)$.

The value of $SP$ for different tasks in a task set depends on the amount of parallelism of each task on the GPU. For simplicity, in our experiments we assume that all tasks have the same speed up. A speed up strictly greater than 1 is needed to justify the use of a GPU. However, higher values of $SP$ are better.

For the experimental results reported in Figure 4, implicit-deadline task sets were generated for $m$ processors with total effective utilization in the range $[m \times 0.05, m \times 2]$ and in increments of $m \times 0.1$. For each of the total effective utilization values, 1000 sets of effective utilization values were generated such that each set had $n = m \times 10$ values. Different values of $m$, $n$, and $SP$ were used in these experiments. In Figure 4, the results are shown for $m = 4$, $n = 40$, and $SP = 30$. From the generated utilization values and $SP = 30$ the following task sets were generated:

- LPE - limited-preemptive task set with $g = m$.
- LPL - limited-preemptive task set with $g = m/2$.
- FP - fully-preemptive task set that does not use GPUs for any part of its computation. The parameters for each task $T_i$ were as follows; $C_i = u_i \times T_i$, $G_i = 0$, $L_i = 0$. Thus, the actual utilization of a task was equal to its effective utilization.

We refer to task sets generated from the same effective utilization values as corresponding task sets. Thus, LPE, LPL, and FP are corresponding task sets.

To analyze the affect of the speed up, $SP$, implicit-deadline task sets were generated for $m$ processors and $g$ GPUs with $m = g$. The total effective utilization of the task sets generated was in the range $[m \times 0.05, m \times 2]$ in increments of $m \times 0.1$. For each total effective utilization, 1000 task sets were generated with $n = m \times 10$ tasks and $SP = 30$ and then for each of the 1000 task sets, corresponding task sets i.e., task sets with the same effective utilization values, were generated with $SP = 20$. The results for $m = 4$ and $n = 40$ are shown in Figure 5(a). With smaller values of $SP$ the length of the non-preemptive execution $L_i$ increases. This reduces schedulability. For comparison, we also show the schedulability of the corresponding fully-preemptive task set. We observe that for smaller values of $SP$ the schedulability of FP can be better than that of LPE.

---

**Figure 4:** $m = 4$, $n = 40$, $SP = 30$

Our schedulability test was applied to each of the above generated task sets and the results obtained are shown in Figure 4. We observe that task set LPE has better schedulability than FP for higher values of total effective utilization and LPE has significantly better schedulability than LPL. Recall from our discussion in Section 1 that worst-case execution times for tasks under fully-preemptive scheduling are often larger. Thus, in practice the schedulability of FP will be lower than what is shown in the graph for a given task set. From this experiment we can conclude that a small difference in the ratio of the number of processors to the number of GPUs makes a significant difference in schedulability. This is due to an increase in the length of the non-preemptive execution $L_i$.

**Figure 5**

The above experiment was repeated to analyze the affect of the number of tasks in a task set. In this case, for each total effective utilization mentioned above, 1000 task sets with $n = m \times 5$ tasks were generated and then 1000 task sets with $n = m \times 10$ tasks were generated. In both cases $SP$ was the same ($SP = 30$). The results for $m = 4$ are shown in Figure 5(b). With smaller number of tasks in a task set the effective utilization of each task increases and as a result the length of its preemptive $C_i$ and non-preemptive $L_i$ execution increases. Therefore, schedulability decreases. We observe that for smaller values of total effective utilization
LPE with \( n = 40 \) tasks has better schedulability than LPE with \( n = 20 \) tasks. However, for larger values of total effective utilization the schedulability of LPE for larger number of tasks is dominated by the number of tasks that contribute to the term \( B_n \) where as for smaller number of tasks, even though the length of \( L_i \) is greater for each task, the number of tasks that contribute to the term \( B_n \) is fewer. Thus, we observe that for larger values of total effective utilization the schedulability of LPE for both \( n = 40 \) and \( n = 20 \) tasks is comparable. For comparison, we also show the schedulability of the corresponding fully-preemptive task sets denoted as \( FP \) with \( n = 40 \) and \( n = 20 \) tasks.

8. CONCLUSION

Limited-preemption scheduling is an alternative to the extreme options of fully-preemptive scheduling and non-preemptive scheduling. While preemptions are better from a schedulability perspective, run-time overheads incurred by arbitrary preemptions can be large.

In this paper, we have obtained a pseudo-polynomial time schedulability test for limited-preemption scheduling under GEDF. We have shown how to apply this schedulability test to a multiprocessor multi-GPU system with non-preemptive busy-waiting. We have also indicated how further analysis may provide better analytical results for the multi-GPU system model under consideration. A comparison of analytical results under different multi-GPU system models is merited.

Acknowledgement

Work supported by NSF grants CNS 1016954, CNS 1115284, CNS 1218693, and CNS 1239135. We would like to thank Glenn Elliott, and James Anderson for insightful discussions about GPUs and locking protocols. We would also like to thank the anonymous reviewers for their useful feedback.

9. REFERENCES


