Spring 5-15-2014

Global EDF Scheduling for Parallel Real-Time Tasks

Jing Li
Washington University in St. Louis

Washington University in St. Louis

School of Engineering and Applied Science

Department of Computer Science and Engineering

Global EDF Scheduling for Parallel Real-Time Tasks

by

Jing Li

A thesis presented to the Graduate School of Arts and Sciences
of Washington University in partial fulfillment of the
requirements for the degree of

Master of Science

May 2014
Saint Louis, Missouri

Contents

List of Tables

List of Figures

Acknowledgments

I would like to first thank both my academic advisors, Dr. Chenyang Lu and Dr. Kunal Agrawal, who has been guiding my study and teaching me how to do great research from big pictures (like finding important and hard problems to tackle) to every detail (such as presentation skills, writing skills and proving techniques). I am grateful to Dr. Christopher Gill, who is extremely supportive in all of my researches.

I am also thankful for the many collaborators who made this research possible: Abusayeed Saifullah, who started the research on parallel real-time scheduling and led me into this research area; David Ferry, who started the implementation of the first parallel real-time platform and also who I have been closely working with and learn from; Kevin Kieselbach, who helped me to build the platform interface and framework; Zheng Luo, who helped me to implement the prototype system and to run extensive measurements to test the system.

Finally, this research was supported in part by NSF grants CCF-1136073 (CPS), CCF-1439478 and CCF-1337218.

Jing Li

Washington University in Saint Louis
May 2014

ABSTRACT OF THE THESIS

Global EDF Scheduling for Parallel Real-Time Tasks

by

Jing Li

Master of Science in Computer Science

Washington University in St. Louis, May 2014

Research Advisor: Professor Chenyang Lu, Professor Kunal Agrawal

As multicore processors become ever more prevalent, it is important for real-time programs to take advantage of intra-task parallelism in order to support computation-intensive applications with tight deadlines. In this thesis, we consider the Global Earliest Deadline First (GEDF) scheduling policy for task sets consisting of parallel tasks. Each task can be represented by a directed acyclic graph (DAG) where nodes represent computational work and edges represent dependences between nodes.

In this model, we prove that GEDF provides a capacity augmentation bound of $4 - \frac{2}{m}$ and a resource augmentation bound of $2 - \frac{1}{m}$. The capacity augmentation bound acts as a linear-time schedulability test since it guarantees that any task set with total utilization of at most $m/(4 - \frac{2}{m})$ where each task's critical-path length is at most $1/(4 - \frac{2}{m})$ of its deadline is schedulable on m cores under GEDF. In addition, we present a pseudo-polynomial time fixed-point schedulability test for GEDF; this test uses a carry-in work calculation based on the proof for the capacity bound.

Finally, we present and evaluate a prototype platform — called PGEDF — for scheduling parallel tasks using GEDF. PGEDF is built by combining the GNU OpenMP runtime system and the LITMUS^{RT} operating system. This platform allows programmers to write parallel OpenMP tasks and specify real-time parameters such as deadlines for tasks.

We perform two kinds of experiments to evaluate the performance of GEDF for parallel tasks. (1) We run numerical simulations for DAG tasks. (2) We execute randomly generated tasks using PGEDF. Both sets of experiments indicate that GEDF performs surprisingly well and outperforms an existing scheduling techniques that involves task decomposition.

Chapter 1

Introduction

During the last decade, the increase in performance processor chips has come primarily from increasing numbers of cores. This has led to extensive work on real-time scheduling techniques that can exploit multicore and multiprocessor systems. Most prior work has concentrated on ***inter-task*** parallelism, where each task runs sequentially (and therefore can only run on a single core) and multiple cores are exploited by increasing the number of tasks. This type of scheduling is called ***multiprocessor scheduling***. When a model is limited to inter-task parallelism, each individual task's total execution requirement must be smaller than its deadline since individual tasks cannot run any faster than on a single-core machine. In order to enable tasks with higher execution demands and tighter deadlines, such as those used in autonomous vehicles, video surveillance, computer vision, radar tracking and real-time hybrid testing [54], we must enable parallelism within tasks.

In this paper, we are interested in parallel scheduling, where in addition to inter-task parallelism, task sets contain ***intra-task*** parallelism, which allows threads from one task to run in parallel on more than a single core. While there has been some recent work in this area, many of these approaches are based on task decomposition [43, 64, 63], which first decomposes each parallel task into a set of sequential subtasks with assigned intermediate release times and deadlines, and then schedules these sequential subtasks using a known multiprocessor scheduling algorithm. In this work, we are interested in analyzing the performance of ***global EDF*** (***GEDF***) schedulers *without* any decomposition.

We consider a general task model, where each task is represented as a ***directed acyclic graph (DAG)*** and where each node represents a sequence of instructions (thread) and each edge represents a dependency between nodes. A node is ***ready*** to be executed when all its predecessors have been executed. GEDF works as follows: for ready nodes at each time step, the scheduler first tries to schedule as many jobs with the earliest deadline as it can; then it schedules jobs with the next earliest deadline, and so on, until either all cores are busy or no more nodes are ready.

Compared with other schedulers, GEDF has benefits, such as automatic load balancing. Efficient and scalable implementations of GEDF for sequential tasks are available for Linux [47] and LITMUS$^{\mathrm{RT}}$ [16], which can be used to implement GEDF for parallel tasks if decomposition is not required. Prior theory analyzing GEDF for parallel tasks is either restricted to a single recurring task [10] or considers response time analysis for soft-real time tasks [51]. In this paper, we consider task sets with n tasks and analyze their schedulability under a GEDF scheduler in terms of augmentation bounds.

We distinguish between two types of augmentation bounds, both of which are called "resource augmentation" in the previous literature. By standard definition, a scheduler $\mathcal{S}$ provides a ***resource augmentation bound*** of b if the following condition holds: if an ideal scheduler can schedule a task set on m unit-speed cores, then $\mathcal{S}$ can schedule that task set on m cores of speed b. Note that the ***ideal scheduler (optimal schedule)*** is only a hypothetical scheduler, meaning that if a feasible schedule ever exists for a task set then this ideal scheduler can guarantee to schedule it. Unfortunately, Fisher et al. [32] proved that optimal online multiprocessor scheduling of sporadic task systems is impossible. Since there may be no way to tell whether the ideal scheduler can schedule a given task set on unit-speed cores, a resource augmentation bound may not provide a schedulability test.

Therefore, we distinguish resource augmentation from a ***capacity augmentation bound*** that can serve as an easy schedulability test. If on unit-speed cores, a task set has total utilization of at most m and the critical-path length of each task is smaller than its deadline, then scheduler $\mathcal{S}$ with capacity augmentation bound b can schedule this task set on m cores of speed b. Note that the ideal scheduler cannot schedule any task set that does not meet these utilization and critical-path length bounds on unit-speed cores; therefore, a capacity augmentation bound of b implies a resource augmentation bound of b. Capacity augmentation bounds have the advantage that they directly lead to schedulability tests, since one can easily check the bounds on utilization and critical-path length for any task set.

The contributions presented in this paper are as follows:

1. For a system with m identical cores, we prove a capacity augmentation bound of $4 - \frac{2}{m}$ (which approaches 4 as m approaches infinity) for sporadic task sets with *implicit deadlines* — the relative deadline of each task is equal to its period. Another way to understand this bound is: if a task set has total utilization at most $m/(4 - \frac{2}{m})$ and the critical-path length of each task is at most $1/(4 - \frac{2}{m})$ of its deadline, then it can be scheduled using GEDF on unit-speed cores.

2. While the capacity augmentation bound functions as a linear-time schedulability test, we further provide a fixed-point schedulability test that may admit more task sets but takes pseudo-polynomial time to compute.

3. For a system with m identical cores, we prove a resource augmentation bound of $2 - \frac{1}{m}$ (which approaches 2 as m approaches infinity) for sporadic task sets with *arbitrary deadlines*.

4. We also show that GEDF's capacity bound for parallel task sets (even with implicit deadlines) is lower bounded by $2 - \frac{1}{m}$. In particular, we show example task sets with utilization m where the critical-path length of each task is no more than its deadline, while GEDF misses a deadline on m cores with speed less than $\frac{3+\sqrt{5}}{2} \approx 2.618$.

5. We conduct simulation experiments to show that the capacity augmentation bound is safe for task sets with different DAG structures (as mentioned above, checking the resource augmentation bound is difficult since we cannot compute the optimal schedule). Simulations show that GEDF performs surprisingly well. All simulated random task sets meet their deadlines with 50% utilization (core speed of 2). We also compare GEDF with a scheduling technique that decomposes parallel tasks and then schedules decomposed subtasks using GEDF [63]. For all of the DAG task sets considered in our experiments, the GEDF scheduler without decomposition has better performance.

6. To demonstrate the feasibility of parallel GEDF scheduling in real systems, we implement a prototype platform named ***PGEDF***. PGEDF supports standard OpenMP programs with parallel for-loops. Therefore, it supports a subset of DAGs — namely ***synchronous*** tasks where the program consists of a sequence of segments which can be parallel or sequential and parallel segments are represented using parallel for-loops. While not as general as DAGs, these synchronous tasks constitute a large subset of interesting parallel programs. PGEDF integrates the GNU OpenMP runtime system [60]

and LITMUS$^{\mathrm{RT}}$ patched Linux kernel [16], where the former executes each task with parallel threads and the latter is responsible for scheduling threads of all tasks under GEDF scheduling.

7. We evaluate the performance of PGEDF with randomly generated synthetic task sets. With those task sets, all deadlines are met when total utilization is less than 30% (core speed of 3.3) in PGEDF. We compare PGEDF with an existing parallel real-time platform, RT-OpenMP [30], which was also designed for synchronous tasks but under decomposed fixed priority scheduling. We find that for most task sets, PGEDF performs better.

In the rest of the paper, Chapter 2 reviews related work and Chapter 3 describes the DAG task model with intra-task parallelism. Proof for a capacity augmentation bound of $4 - \frac{2}{m}$ and a fixed point schedulability test based on capacity augmentation bound are presented in Chapters 4 and 5 respectively. We prove a resource augmentation bound of $2 - \frac{1}{m}$ in Chapter 6. In Chapter 7, we present an example to show the lower bound on capacity bound for GEDF. Chapter 8 shows the simulation results. Then we describe the implementation of our PGEDF platform in Chapter 9 and evaluate it in Chapter 10. Finally, Chapter 11 gives concluding remarks.

Chapter 2

Related Work

Most prior work on ***hard real-time scheduling*** atop multiprocessors has concentrated on sequential tasks [22]. In this context, many sufficient schedulability tests for GEDF and other global fixed priority scheduling algorithms have been proposed [3, 66, 34, 12, 8, 7, 44, 11, 13]. In particular, for implicit deadline hard-real time tasks, the best known utilization bound is $\approx 50\%$ using partitioned fixed priority scheduling [4] or partitioned EDF [9, 52]; this trivially implies a capacity bound of 2. [9] proved that global EDF has a capacity augmentation bound of $2 - 1/m$ for sequential tasks on multiprocessors.

Earlier work considering intra-task parallelism makes strong assumptions on task models [45, 20, 55]. For more realistic parallel tasks, e.g. ***synchronous tasks***, Kato et al.[38] proposed a gang scheduling approach. The synchronous model, a special case of the more general DAG model, represents tasks with a sequence of multi-threaded segments with synchronization points between them (such as those generated by parallel for-loops).

Most other approaches for scheduling synchronous tasks involve decomposing parallel tasks into independent sequential subtasks, which are then scheduled using known multiprocessor scheduling techniques, such as deadline monotonic [31] or GEDF [8]. For a restricted set of synchronous tasks, Lakshmanan et al. [43] prove a capacity augmentation bound of 3.42 using deadline monotonic scheduling for decomposed tasks. For more general synchronous tasks, Saifullah et al. [64] proved a capacity augmentation bound of 4 for GEDF and 5 for deadline monotonic scheduling. The decomposition strategy was improved in [58] for using less cores. For the same general synchronous model, the best known augmentation bound is 3.73 [39] also using decomposition. The decomposition approach in [64] was recently extended to general DAGs [63] to achieve a capacity augmentation bound of 4 under GEDF on decomposed tasks (note that in that work GEDF is used to schedule sequential decomposed tasks, not parallel tasks directly). This is the best augmentation bound known for task sets with multiple DAGs. For scheduling synchronous tasks withough decomposition, [19] and [6] presented schedulability tests for GEDF and partitioned fixed priority scheduling respectively.

More recently, there has been some work on scheduling general DAGs without decomposition. Nogueira et al. [59] explored the use of work-stealing for real-time scheduling. The paper is mostly experimental and focused on soft real-time performance. The bounds for hard real-time scheduling only guarantee that tasks meet deadlines if their utilization is smaller than 1. Liu and Anderson [51] analyzed the response time of GEDF without decomposition for soft real-time tasks. A resource augmentation bound of $2 - \frac{1}{m}$ for GEDF was proved for a staged DAG model [5]. Baruah et al. [10] proved that when the task set is a *single DAG task* with arbitrary deadlines, GEDF provides a resource augmentation bound of 2. For multiple DAGs, Bonifaci et al. [14] also show the same resource augmentation bound $2 - \frac{1}{m}$, but do not consider capacity augmentation. They also proved that global deadline monotonic scheduling has a resource augmentation bound of $3 - \frac{1}{m}$.

Various platforms support sequential real-time tasks on parallel machines [16, 47]. Our platform prototype, PGEDF, is based on LITMUS$^{\text{RT}}$ [16]. As for parallel tasks, we are aware of two systems [39, 30] that support parallel real-time tasks based on different decomposition strategies. Kim et al. [39] used a reservation-based OS to implement a system that can run parallel real-time programs for an autonomous vehicle application, demonstrating that parallelism can enhance performance for complex tasks. Ferry et al. [30] developed a parallel real-time scheduling service on standard Linux. However, since both systems adopted task decomposition approaches, they require users to provide exact task structures and subtask execution time details in order to decompose tasks correctly. The system presented [30] also requires modifications to the compiler and runtime system to decompose, dispatch and execute parallel applications. The platform prototype presented here does not require decomposition or such detailed information.

Scheduling parallel tasks without deadlines has been addressed by parallel-computing researchers [62, 26, 23, 1]. Soft real-time scheduling has been studied for various optimization criteria, such as cache misses [17, 2], makespan [67] and total work done by tasks that meet deadlines [42].

Chapter 3

Task Model and Definitions

This chapter presents a model for DAG tasks. We consider a system with m identical unit-speed cores. The task set τ consists of n tasks $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$. Each task τ_i is represented by a directed acyclic graph (DAG), and has a period P_i and deadline D_i. We represent the j-th subtask of the ith task as node W_i^j. A directed edge from node W_i^j to W_i^k means that W_i^k can only be executed after W_i^j has finished executing. A node is ***ready*** to be executed as soon as all of its predecessors have been executed. Each node has its own worst-case execution time C_i^j. Multiple source nodes and sink nodes are allowed in the DAG, and the DAG is not required to be fully connected. Figure 3.1 shows an example of a task consisting of 5 subtasks in the DAG structure.

For each task τ_i in task set τ, let $C_i = \sum_j C_i^j$ be the total worst-case execution time on a single core, also called the ***work*** of the task. Let L_i be the critical-path length (i.e. the worst-case execution time of the task on an infinite number of cores). In Figure 3.1, the critical-path (i.e. the longest path) starts from node W_1^1, goes through W_1^3 and ends at node W_1^4, so the critical-path length of DAG W_1 is $1 + 3 + 2 = 6$. The work and the critical-path length of any job generated by task τ_i are the same as those of task τ_i.

We also define the notion of ***remaining work*** and ***remaining critical-path length*** of a partially executed job. The remaining work is the total work minus the work that has already been done. The remaining critical-path length is the length of the longest path in the unexecuted portion of the DAG (including partially executed nodes). For example, in Figure 3.1, if W_1^1 and W_1^2 are completely executed, and W_1^3 is partially executed such that 1 unit (out of 3) of work has been done for it, then the remaining critical-path length is $2 + 2 = 4$.

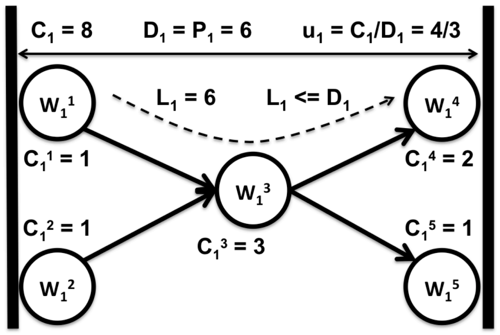


Figure 3.1: Example task with work $C_i = 8$ and critical-path length $L_i = 6$.

Nodes do not have individual release offsets and deadlines when scheduled by the GEDF scheduler; they share the same absolute deadline of their jobs. Therefore, to analyze the GEDF scheduler, we do not require any knowledge of the DAG structure beyond the total worst-case execution time C_i, deadline D_i, period P_i and critical-path length L_i. We also define the utilization of a task τ_i as $u_i = \frac{C_i}{P_i}$.

On unit speed cores, a task set is not schedulable (by any scheduler) unless the following conditions hold:

- The critical-path length of each task is less than its deadline.

$$L_i \leq D_i \tag{3.1}$$

- The total utilization is smaller than the number of cores.

$$\sum_i u_i \leq m \tag{3.2}$$

In addition, we denote $J_{k,a}$ as the a-th job instance of task k in system execution. For example, the i-th node of $J_{k,a}$ is represented as $W^i_{k,a}$. We denote $r_{k,a}$ and $d_{k,a}$ as the absolute release time and absolute deadline of job $J_{k,a}$ respectively. Relative deadline D_k is equal to $d_{k,a} - r_{k,a}$. Since in this paper we address sporadic tasks, the absolute release time has the following properties:

$$\begin{aligned} r_{k,a+1} &\geq d_{k,a} \\ r_{k,a+1} - r_{k,a} &\geq d_{k,a} - r_{k,a} = D_k \end{aligned}$$

Chapter 4

Capacity Augmentation Bound of $4 - \frac{2}{m}$ for GEDF

In this chapter, we propose a capacity augmentation bound of $4 - \frac{2}{m}$ for *implicit deadline tasks*, which yields an easy schedulability test. In particular, we show that GEDF can successfully schedule a task set, if the task set satisfies two conditions: (1) its total utilization is at most $m/(4 - \frac{2}{m})$ and (2) the critical-path length of each task is at most $1/(4 - \frac{2}{m})$ of its period (and deadline). Note that this is equivalent to saying that if a task set meets conditions from Inequalities 3.1 and 3.2 on processors of unit speed, then it can be scheduled on m cores of speed $4 - \frac{2}{m}$ (which approaches 4 as m approaches infinity).

The gist of the proof is the following: at a job's release time, we can bound the remaining work from other tasks under GEDF with speedup $4 - \frac{2}{m}$. Bounded remaining work leads to bounded interference from other tasks, and hence GEDF can successfully schedule all of them.

4.1 Notation

We first define a notion of interference. Consider a job $J_{k,a}$, which is the a-th instance of task τ_k. Under GEDF scheduling, only jobs that have absolute deadlines earlier than the absolute deadline of $J_{k,a}$ can interfere with $J_{k,a}$. We say that a job is ***unfinished*** if the job has been released but has not completed yet. Due to implicit deadlines ($D_i = P_i$), at most one job of each task can be unfinished at any time.

There are two sources of interference for job $J_{k,a}$. (1) ***Carry-in work*** is the work from jobs that were released before $J_{k,a}$, did not finish before $J_{k,a}$ was released, and have deadlines before the deadline of $J_{k,a}$. Let $R_i^{k,a}$ be the carry-in work due to task τ_i and let $R^{k,a} = \sum_i R_i^{k,a}$ be the total carry-in from the entire task set onto the job $J_{k,a}$. (2) Other than carry-in work, the jobs that were released after (or at the same time as) $J_{k,a}$ was released can also interfere with it if their deadlines are either before or at the same time as $J_{k,a}$. Let $n_i^{k,a}$ be the number of jobs of task τ_i, which are released after the release time of $J_{k,a}$ but have deadlines no later than the deadline of $J_{k,a}$ (that is, the number of jobs from task τ_i that entirely fall in between the release time and deadline of $J_{k,a}$, i.e. the time interval $[r_{k,a}, d_{k,a}]$.) For example, in the right hand side of Figure 4.1, one entire job $J_{1,3}$ falls within time interval $[r_{3,1}, d_{3,1}]$ of job $J_{3,1}$, so $n_1^{3,1} = 1$. By definition (and $D_i = P_i$), every task i has the property that

$$n_i^{k,a} D_i \leq D_k \tag{4.1}$$

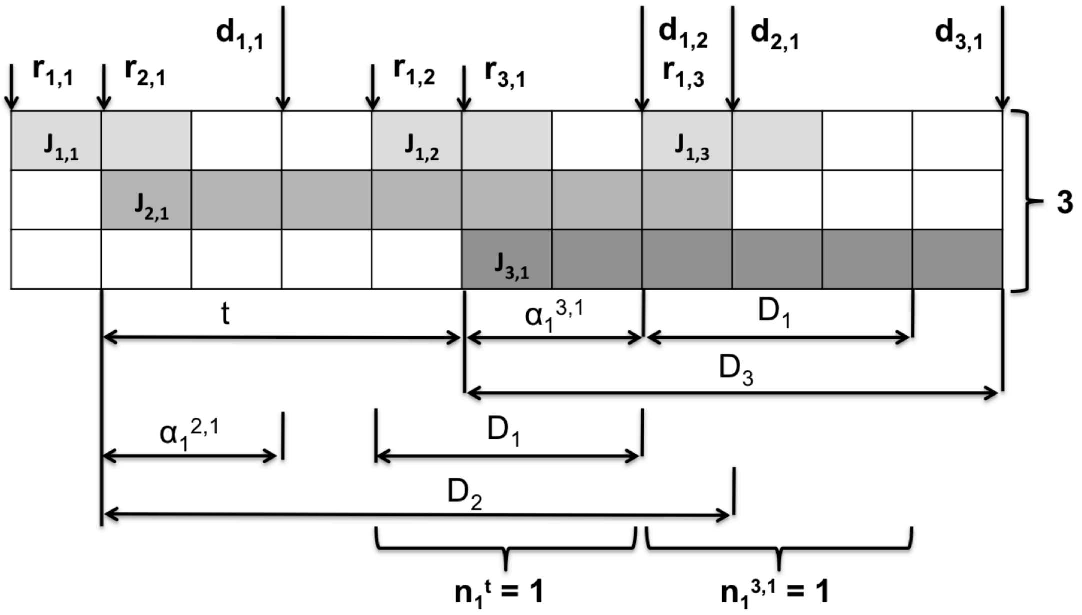


Figure 4.1: Example task set execution trace

Therefore, the total amount of work $A^{k,a}$, that can interfere with $J_{k,a}$ (including $J_{k,a}$'s work) and (to prevent any deadline misses) must be finished before the deadline of $J_{k,a}$ is the sum of the carry-in work and the work that was released at or after $J_{k,a}$'s release.

$$A^{k,a} = R^{k,a} + \sum_i u_i n_i^{k,a} D_i. \tag{4.2}$$

Note that the work of the job $J_{k,a}$ itself is also included in this formula. That is, in this formulation, each job interferes with itself.

4.2 Proof for Capacity Augmentation Bound

Consider a GEDF schedule with m cores each of speed b. Each time step can be divided into b sub-steps such that each core can do one unit of work in each sub-step. We say a sub-step is ***complete*** if all cores are working during that sub-step, and otherwise we say it is ***incomplete***.

First, a couple of straight-forward lemmas.

Lemma 1 *On every incomplete sub-step, the remaining critical-path length of each unfinished job reduces by 1.*

Lemma 2 *In any t contiguous time steps (bt sub-steps) with unfinished jobs, if there are t^* incomplete sub-steps, then the total work done during this time, F_t is at least*

$$F^t \geq bmt - (m-1)t^*.$$

Proof. The total number of complete sub-steps during t steps is $bt - t^*$, and the total work during these complete steps is $m(bt - t^*)$. On an incomplete sub-step, at least one unit of work is done. Therefore, the total work done in incomplete sub-steps is at least t^*. Adding the two gives us the bound. □

We now prove a sufficient condition for the schedulability of a job.

Lemma 3 *If interference $A^{k,a}$ on a job $J_{k,a}$ is bounded by*

$$A^{k,a} \leq bmD_k - (m-1)D_k,$$

then job $J_{k,a}$ can meet its deadline on m identical cores with speed of b.

Proof. Note that there are D_k time steps (therefore bD_k sub-steps) between the release time and deadline of this job. There are two cases:

Case 1: The total number of incomplete sub-steps between the release time and deadline of $J_{k,a}$ is more than D_k, and therefore, also more than L_k. In this case, $J_{k,a}$'s critical-path length reduces on all of these sub-steps. After at most L_k incomplete steps, the critical-path is 0 and the job has finished executing. Therefore, it can not miss the deadline.

Case 2: The total number of incomplete sub-steps between the release and deadline of $J_{k,a}$ is smaller than D_k. Therefore, the total amount of work done during this time is more than $bmD_k - (m-1)D_k$ by the condition in Lemma 2. Since the total interference (including $J_{k,a}$'s work) is at most this quantity, the job cannot miss its deadline. □

We now define additional notation in order to prove that if the carry-in work for a job is bounded, then GEDF guarantees a capacity augmentation bound of b. Let $\alpha_i^{k,a}$ be the number of time steps between the absolute release time of $J_{k,a}$ and the absolute deadline of the carry-in job of task i. Hence, for $J_{k,a}$ and its carry-in job $J_{j,b}$ of task j

$$\alpha_j^{k,a} = d_{j,b} - r_{k,a} \tag{4.3}$$

To make the notation clearer, we give an example that is also illustrated in Figure 4.1. There are 3 sporadic tasks with implicit deadlines: the (execution time, deadline, period) for tasks τ_1, τ_2 and τ_3 are (2, 3, 3), (7, 7, 7) and (6, 6, 6) respectively. For simplicity, assume they are sequential tasks. Since tasks are sporadic, $r_{1,2} > d_{1,1}$. $\alpha_1^{3,1}$ is the number of time steps between the release time of job $J_{3,1}$ and the deadline of the carry-in job $J_{1,2}$ from task 1. In this example, $\alpha_1^{3,1} = 2$. Similarly, $\alpha_2^{3,1} = 3$. Also, $n_1^{3,1} = 1$.

For either periodic or sporadic tasks, task i has the property

$$\alpha_i^{k,a} + n_i^{k,a} D_i \leq D_k \tag{4.4}$$

Since $\alpha_i^{k,a}$ is the remaining length of the carry-in job and $n_i^{k,a}$ is the number of jobs of task τ_i entirely falling in the period (relative deadline) of job $J_{k,a}$, then as in Figure 4.1, $\alpha_1^{3,1} + n_1^{3,1} D_1 = 2 + 1 * 3 = 5 < 6 = D_3$.

Lemma 4 *If the cores' speed is $b \geq 4 - \frac{2}{m}$ and the total carry-in work $R^{k,a}$ from every task τ_i satisfies the condition*

$$R^{k,a} \leq \sum_i u_i \alpha_i^{k,a} + m \cdot \max_i (\alpha_i^{k,a}),$$

then job $J_{k,a}$ always meets its deadline under global EDF.

Proof. The total amount of interfering work (including $J_{k,a}$'s work) is $A^{k,a} = R^{k,a} + \sum_i u_i n_i^{k,a} D_i$. Hence, according to the condition in Lemma 4, the total amount of work is:

$$\begin{aligned} A^{k,a} &= R^{k,a} + \sum_i u_i n_i^{k,a} D_i \\ &\leq \sum_i u_i \alpha_i^{k,a} + m \max_i (\alpha_i^{k,a}) + \sum_i u_i n_i^{k,a} D_i \\ &\leq \sum_i u_i (\alpha_i^{k,a} + n_i^{k,a} D_i) + m \max_i (\alpha_i^{k,a}) \end{aligned}$$

Using eq.(4.4) to substitute D_k into the formula, then

$$A^{k,a} \leq \sum_i u_i D_k + m D_k$$

Since the total task set utilization does not exceed the number of cores m, by eq.(3.2), we replace $\sum_i u_i$ with m. And since $b \geq 4 - \frac{2}{m}$ and $m \geq 1$, we get

$$\begin{aligned} A^{k,a} &\leq 2mD_k \leq (3m-1)D_k \\ &\leq (4 - \frac{2}{m})mD_k - (m-1)D_k \\ &\leq bmD_k - (m-1)D_k \end{aligned}$$

Finally, according to Lemma 3, since the interference satisfies the bound, job $J_{k,a}$ can meet its deadline. $\square$

We now complete the proof by showing that the carry-in work is bounded as required by Lemma 4 for every job.

Lemma 5 *If the core's speed $b \geq 4 - \frac{2}{m}$, then, for either periodic or sporadic task sets with implicit deadlines, the total carry-in work $R^{k,a}$ for every job $J_{k,a}$ in the task set is bounded by*

$$R^{k,a} \leq \sum_i u_i \alpha_i^{k,a} + m \max_i (\alpha_i^{k,a})$$

Proof. We prove this theorem by induction from absolute time 0 to the release time of job $J_{k,a}$.

Base Case: For the very first job of all the tasks released in the system (denoted $J_{l,1}$), no carry-in jobs are released before this job. Therefore, the condition trivially holds and the job can meet its deadline by Lemma 4.

$$R^{l,1} = 0 \leq \sum_i u_i \alpha_i^{l,1} + m \max_i (\alpha_i^{l,1})$$

Inductive Step: Assume that for every job with an earlier release time than $J_{k,a}$, the condition holds. Therefore, according to Lemma 4, every earlier released job meets its deadline. Now we prove that the condition also holds for job $J_{k,a}$.

For job $J_{k,a}$, if there is no carry-in work from jobs released earlier than $J_{k,a}$, so that $R^{k,a} = 0$, the property trivially holds. Otherwise, there is at least one unfinished job (a job with carry-in work) at the release time of $J_{k,a}$.

We now define $J_{j,b}$ as the job with the earliest release time among all the unfinished jobs at the time that $J_{k,a}$ was released. For example, at release time $r_{3,1}$ of $J_{3,1}$ in Figure 4.1, both $J_{1,2}$ and $J_{2,1}$ are unfinished, but $J_{2,1}$ has the earliest release time. By the inductive assumption, the carry-in work $R^{j,b}$ at the release time of job $J_{j,b}$ is bounded by

$$R^{j,b} \leq \sum_i u_i \alpha_i^{j,b} + m \max_i (\alpha_i^{j,b}) \tag{4.5}$$

Let t be the number of time steps between the release time $r_{j,b}$ of $J_{j,b}$ and the release time $r_{k,a}$ of $J_{k,a}$.

$$t = r_{k,a} - r_{j,b}$$

Note that $J_{j,b}$ has not finished at time $r_{k,a}$, but by assumption it can meet its deadline. Therefore its absolute deadline $d_{j,b}$ is later than the release time $r_{k,a}$. So, by eq.(4.3)

$$t + \alpha_j^{k,a} = r_{k,a} - r_{j,b} + \alpha_j^{k,a} = d_{j,b} - r_{j,b} = D_j \tag{4.6}$$

In Figure 4.1, $t + \alpha_1^{3,1} = r_{3,1} - r_{2,1} + \alpha_1^{3,1} = d_{2,1} - r_{2,1} = D_2$.

For each τ_i, let n_i^t be the number of jobs that are released after the release time $r_{j,b}$ of $J_{j,b}$ but before the release time $r_{k,a}$ of $J_{k,a}$. The last such job may have a deadline after the release time of $r_{k,a}$, but its release time is before $r_{k,a}$. In other words, n_i^t is the number of jobs of task τ_i, which fall entirely into the time interval $[r_{j,b}, r_{k,a} + D_i]$. By definition of $\alpha_i^{k,a}$, to job $J_{k,a}$, the deadline of the unfinished job of task τ_i is $r_{k,a} + \alpha_i^{k,a}$. Therefore, for every τ_i,

$$\alpha_i^{j,b} + n_i^t D_i \leq r_{k,a} + \alpha_i^{k,a} - r_{j,b} = t + \alpha_i^{k,a} \tag{4.7}$$

As in the example in Figure 4.1, one entire job of task τ_1 falls within $[r_{2,1}, r_{3,1} + D_1]$, making $n_1^t = 1$ and $d_{1,2} = r_{3,1} + \alpha_1^{3,1}$. Also, since $d_{1,1} \leq r_{1,2}$, $\alpha_1^{2,1} + n_1^t D_1 = \alpha_1^{2,1} + D_1 \leq d_{1,2} - r_{2,1} = r_{3,1} + \alpha_1^{3,1} - r_{2,1} = t + \alpha_1^{3,1} \leq t + D_1$.

Comparing between t and $\alpha_j^{k,a}$, when $t \le \frac{1}{2}D_j$, by eq.(4.6), $\alpha_j^{k,a} = D_j - t \ge \frac{1}{2}D_j \ge t$. There are two cases:

Case 1: $t \le \frac{1}{2}D_j$ and hence $\alpha_j^{k,a} \ge t$:

Since by definition $J_{j,b}$ is the earliest carry-in job, other carry-in jobs to $J_{k,a}$ are released after the release time of $J_{j,b}$ and therefore are not carry-in jobs to $J_{j,b}$. In other words, the carry-in jobs to $J_{j,b}$ must have been finished before the release time of $J_{k,a}$, which means that the carry-in work $R^{j,b}$ is not part of the carry-in work $R^{k,a}$. So the carry-in work $R^{k,a}$ is the sum of those released later than $J_{j,b}$

$$R^{k,a} = \sum_i u_i n_i^t D_i \le \sum_i u_i(t + \alpha_i^{k,a}) \qquad \text{(from eq.(4.7))}$$

By assumption of case 1, $t \le \alpha_j^{k,a} \le \max_i \left(\alpha_i^{k,a}\right)$. Hence, replacing $\sum_i u_i$ with m using eq.(3.2), we can prove that

$$R^{k,a} \le \sum_i u_i \alpha_i^{k,a} + m \max_i \left(\alpha_i^{k,a}\right)$$

Case 2: $t > \frac{1}{2}D_j$:

Since $J_{j,b}$ has not finished executing at the release time of $J_{k,a}$, the total number of incomplete sub-steps during the t time steps $(r_{j,b}, r_{k,a}]$ is less than L_j. Therefore, the total work done during this time is at least F^t where

$$\begin{aligned} F^t &= bmt - (m-1)L_j & \text{(from Lemma 2)} \\ &\ge bmt - (m-1)D_j & \text{(from eq.(3.1))} \end{aligned}$$

The total amount of work from jobs that are released in time interval $(r_{j,b}, r_{k,a}]$ (i.e, entire jobs that fall in between the release time of job $J_{j,b}$ and the release time of job $J_{k,a}$ plus its deadline) is $\sum_i u_i n_i^t D_i$, by the definition of n_i^t. The carry-in work $R^{k,a}$ at the release time of job $J_{k,a}$ is the sum of the carry-in work $R^{j,b}$ and newly released work $\sum_i u_i n_i^t D_i$ minus the finished work during time interval t, which is

$$\begin{aligned} R^{k,a} &= R^{j,b} + \sum_i u_i n_i^t D_i - F^t \\ &\leq R^{j,b} + \sum_i u_i n_i^t D_i - (bmt - (m-1)D_j) \end{aligned} \tag{4.8}$$

By the assumption in eq.(4.5), we can replace $R^{j,b}$ and get

$$\begin{aligned} R^{k,a} &\leq \sum_i u_i \alpha_i^{j,b} + m \max_i \left(\alpha_i^{j,b}\right) \\ &\quad + \sum u_i n_i^t D_i - bmt + (m-1)D_j \\ &\leq \sum_i u_i \left(\alpha_i^{j,b} + n_i^t D_i\right) + m \max_i \left(\alpha_i^{j,b}\right) \\ &\quad - bmt + (m-1)D_j \end{aligned}$$

According to eq.(4.7), we can replace $\alpha_i^{j,b} + n_i^t D_i$ with $t + \alpha_i^{k,a}$, reorganize the formula, and get

$$\begin{aligned} R^{k,a} &\leq \sum_i u_i \left(t + \alpha_i^{k,a}\right) + m \max_i \left(\alpha_i^{j,b}\right) \\ &\quad - bmt + (m-1)D_j \\ &\leq \left(\sum_i u_i \left(t + \alpha_i^{k,a}\right) - mt\right) \\ &\quad + m \max_i \left(\alpha_i^{j,b}\right) + (m-1)D_j - (b-1)mt \end{aligned}$$

Using eq.(3.2) to replace m with $\sum_i u_i$ in the first item, using eq.(4.4) to get $\max_i(\alpha_i^{j,b}) \leq D_j$ and to replace $\max_i(\alpha_i^{j,b})$ with D_j in the second item, and since $t > \frac{1}{2}D_j$,

$$\begin{aligned}
& R^{k,a} \\
\leq\ & \sum_i u_i \alpha_i^{k,a} + mD_j + (m-1)D_j - (b-2)mt - mt \\
\leq\ & \sum_i u_i \alpha_i^{k,a} + mD_j - mt + 2(m-1)t - (b-2)mt \\
\leq\ & \sum_i u_i \alpha_i^{k,a} + m(D_j - t) + 0 \qquad (\text{since } b \geq 4 - \tfrac{2}{m}) \\
\leq\ & \sum_i u_i \alpha_i^{k,a} + m\alpha_j^{k,a} \qquad (\text{from eq.(4.6)})
\end{aligned}$$

Finally, since $\alpha_j^{k,a} \leq \max_i\left(\alpha_i^{k,a}\right)$, we can prove that

$$R^{k,a} \leq \sum_i u_i \alpha_i^{k,a} + m \max_i \left(\alpha_i^{k,a}\right)$$

Hence, by induction, if the core speed $b \geq 4 - \frac{2}{m}$, for every $J_{k,a}$ in task set

$$R^{k,a} \leq \sum_i u_i \alpha_i^{k,a} + m \max_i \left(\alpha_i^{k,a}\right)$$

□

From Lemmas 4 and 5, we can easily derive the following capacity augmentation bound theorem.

Theorem 1 *If, on unit speed cores, the utilization of a sporadic task set is at most m, and the critical-path length of each job is at most its deadline, then the task set can meet all their implicit deadlines on m cores of speed $4 - \frac{2}{m}$.*

Theorem 1 proves the speedup factor of GEDF and it also can be restated as follows:

Corollary 1 *Given that a sporadic task set τ with implicit deadlines satisfies the following conditions: (1) total utilization is at most $1/(4 - \frac{2}{m})$ of the total system capacity m and (2) the critical path L_i of every task $\tau_i \in \tau$ is at most $D_i/(4 - \frac{2}{m})$, then GEDF can schedule this task set τ on m cores.*

Chapter 5

Fixed Point Schedulability Test

In Chapter 4, we described a capacity augmentation bound for the GEDF scheduler, which acts as a simple linear time schedulability test. In this chapter, we describe a tighter fixed point schedulability test for parallel task sets under a GEDF scheduler. We start with a schedulability test similar to one for sequential tasks. Then, we improve the calculation of the carry-in work — this improvement is based on some of the equations used in the proof for our capacity augmentation bound. Finally, we further improve the interference calculation by considering the calculated finish time and altogether derive the fixed point schedulability test.

5.1 Basic Schedulability Test

Given a task set, we denote $\widehat{R_i^k}$ as an upper bound on the carry-in work from task τ_i to a job of task τ_k, and $\widehat{R^k} = \sum_i \widehat{R_i^k}$ as an upper bound on the total carry-in work from the entire task set to a job of task τ_k. We also denote $\widehat{A_i^k}$ and $\widehat{A^k}$ as the corresponding upper bounds on individual and total interference to task τ_k. In addition, $\widehat{n_i^k}$ is an upper bound on the number of task τ_i's interfering jobs, which are not part of the carry-in jobs, but interfere with task τ_k. Finally, we use $\widehat{f_k}$ to denote an upper bound on the relative completion time of task τ_k. If $\widehat{f_k} \leq D_k$, then task τ_k is schedulable, and otherwise it is not.

Then from equation 4.2, we can derive

$$\widehat{A_i^k} \leq \widehat{R_i^k} + u_i \widehat{n_i^k} D_i = \widehat{R_i^k} + \widehat{n_i^k} C_i \tag{5.1}$$

$$\widehat{A^k} = \sum_i \widehat{A_i^k} \le \sum_i \left(\widehat{R_i^k} + \widehat{n_i^k} C_i\right) = \widehat{R^k} + \sum_i \left(\widehat{n_i^k} C_i\right) \tag{5.2}$$

From Lemma 2, we can easily derive that on a unit-speed system with m cores, the maximum completion time of task τ_k is

$$\widehat{f_k} \le \frac{1}{m}\left(\widehat{A^k} + (m-1)L_k\right) \tag{5.3}$$

This is simply because the maximum number of incomplete steps before the completion of task τ_k is its critical-path length L_k and the maximum total available work (having deadlines no later than the completion time) is the maximum total interference $\widehat{A^k}$. Note that the execution time of task τ_k is incorporated in the calculation of total interference, which we will show below.

Consider a job $J_{k,a}$ of task τ_k, which finishes at its absolute deadline $d_{k,a}$. Note that, in order to achieve the maximum interference in order to calculate the upper bound on $\widehat{A_i^k}$, the last job of task τ_i which interferes with $J_{k,a}$ should have the same absolute deadline as $J_{k,a}$, that is, $d_{k,a}$. Hence, in the worst case, the upper bound on the number of interfering jobs that begin after $J_{k,a}$ is released (that is, they are not carry-in jobs) is

$$\widehat{n_i^k} = \left\lfloor \frac{D_k}{D_i} \right\rfloor \tag{5.4}$$

Note that the execution time of task τ_k itself is considered as part of its interference as well, i.e. $\widehat{n_k^k} = 1$.

Obviously there could at most be one carry-in job of task τ_i to the job $J_{k,a}$ of task τ_k. Moreover, if in the worst-case of $\widehat{A_i^k}$, this job has already finished before the release time of $J_{k,a}$, then $\widehat{R_i^k} = 0$. By the definition of carry-in jobs and Equation (5.4) for $\widehat{n_i^k}$, we can see that the length between the deadline of carry-in job and the release time of job $J_{k,a}$ is $D_k - \widehat{n_i^k} D_i$. If the carry-in job has not finished when job $J_{k,a}$ is released, then $D_k - \widehat{n_i^k} D_i$ has to be longer than $D_k - \widehat{f_i}$, where $\widehat{f_i}$ is the upper bound of task τ_i's completion time.

We denote X_i^k below as the upper bound for the maximum carry-in work

$$X_i^k = \begin{cases} C_i & \left(D_k - \widehat{n_i^k} D_i > D_i - \widehat{f_i}\right) \\ 0 & \left(D_k - \widehat{n_i^k} D_i \le D_i - \widehat{f_i}\right) \end{cases} = \left\lceil \frac{D_k - \widehat{n_i^k} D_i}{D_i - \widehat{f_i}} - 1 \right\rceil C_i$$

Then obviously, the upper bound of total carry-in work to task τ_k is

$$\widehat{R^k} = \sum_i \widehat{R_i^k} \le \sum_i X_i^k \tag{5.5}$$

Combining the above calculations together, we can derive the basic fixed point calculation of the maximum completion time of task τ_k:

$$\begin{aligned} \widehat{f_k} &\le \frac{1}{m}\left(\widehat{R^k} + \sum_i \left(\widehat{n_i^k} C_i\right) + (m-1)L_k\right) && (5.6) \\ &\le \frac{1}{m}\left(\sum_i \left(\left(\left\lceil \frac{D_k - \widehat{n_i^k} D_i}{D_i - \widehat{f_i}} - 1\right\rceil + \left\lfloor \frac{D_k}{D_i} \right\rfloor\right) C_i\right) + (m-1)L_k\right) && (5.7) \end{aligned}$$

The fixed point schedulability test works as follows: in the beginning, we set the completion time $\widehat{f_k}$ of each task to be the same as its relative deadline D_k; then we iteratively use Equation (5.7) to calculate a new value of completion time $\widehat{f_k}'$ for all τ_k; we only update $\widehat{f_k}$ if the calculated new value is less than D_k; finally, the calculation will stop if there is no more update for all $\widehat{f_k}$. In the end, we use Equation (5.7) again to calculate the final upper bound of completion time $\widehat{f_k}''$: if for all tasks $\widehat{f_k}'' \le D_k$, then the task set is deemed schedulable; otherwise, not.

Obviously, before the last step of calculating $\widehat{f_k}''$, in each iteration, $\widehat{f_k}$ will not be larger than D_k. After the first iteration, each $\widehat{f_k}$ will either stays at D_k or decrease (because $\widehat{f_k}'$ is less than D_k). More importantly, $\widehat{f_k}$ will decrease or stay the same when at least one $\widehat{f_i}$ of another task τ_i decreases. In conclusion, $\widehat{f_k}$ will not increase in each iteration. Therefore, the fixed point calculation will converge.

Note that there is a subtlety about this calculation. Because of the assumption $\widehat{f_i} \leq D_i$ of Equation (5.4), Equation (5.7) is only correct when the finish time of each task in the task set is no more than its relative deadline. This is the reason why in the fixed point calculation, we do not update $\widehat{f_k}$ if the calculated new value $\widehat{f_k}'$ is larger than D_k. After the last step (calculating $\widehat{f_k}''$) of the fixed point calculation, if the task set is schedulable, i.e. the assumption is satisfied, we actually did correctly calculate an upper bound on the interference and therefore an upper bound on the completion time. Therefore, if this test says that a task set is schedulable, it is indeed schedulable. If the test says that the task set is unschedulable, then the test may be underestimating the interference. In this case, however, this inaccuracy it does not matter, since even the underestimation makes the task set unschedulable, so even the correct estimation will also deem the task set unschedulable.

5.2 Improving the Carry-In Work Calculation

In the basic test, we calculate the carry-in work using Equation (5.5). However, this upper bound calculation X_i^k may be pessimistic, if task τ_k has a very short period, while task τ_i has a very long period. This is because if the carry-in job of τ_i to τ_k has not finished before τ_k is released, then the entire C_i will be counted as interference. However, GEDF, as a greedy algorithm, might have already executed most of the computation of the carry-in job. Inspired by the proof of the capacity augmentation bound for GEDF, we propose another upper bound for $\widehat{R^k}$.

Note that in the proof of Lemma 5, there are the two cases. The calculation of $X^k = \sum_i X_i^k$ in the basic test is similar to Case 1, but without knowing the first carry-in job. Therefore, from Case 2, we can also obtain another upper bound Y^k for $\widehat{R^k}$ without knowing the first carry-in job. After getting the two upper bounds of $\widehat{R^k}$, we can simply take the minimum of X^k and Y^k and achieve a schedulability test.

For $\widehat{R^k}$, if there is no unfinished carry-in job, then $\widehat{R^k} = 0$ for job $J_{k,a}$. Otherwise, say $J_{j,b}$ is the carry-in job with the earliest release time among all the unfinished jobs at the release time of $J_{k,a}$. From Inequality (4.8), on m unit-speed cores,

$$R^{k,a} \leq R^{j,b} + \sum_i n_i^t C_i + (m-1)L_j - mt$$

where t is the interval between the release time $r_{j,b}$ of $J_{j,b}$ and the release time $r_{k,a}$ of $J_{k,a}$ and n_i^t is the number of jobs of task τ_i that are released during this time.

In the worst case for A^k (where every last interfering job of each τ_i has the same deadline as $J_{k,a}$'s deadline), from Equation (5.4), we can calculate t:

$$\begin{aligned} t &= D_j + \widehat{n_j^k} D_j - D_k \\ n_i^t &\leq \left\lceil \frac{t}{D_i} \right\rceil = \left\lceil \frac{D_j + \widehat{n_j^k} D_j - D_k}{D_i} \right\rceil \end{aligned}$$

Therefore, if task τ_j is indeed the task having the first carry-in job, then the maximum of the carry-in work $\widehat{R^k}$ of task τ_k can be bounded by Y_j^k where

$$\begin{aligned} Y_j^k \leq\ & Y^j + \sum_i \left(\left\lceil \frac{D_j + \widehat{n_j^k} D_j - D_k}{D_i} \right\rceil C_i \right) \\ & + (m-1)L_j - m(D_j + \widehat{n_j^k} D_j - D_k) \end{aligned} \tag{5.8}$$

Note that the bound Y_j^k is an upper bound on $\widehat{R^k}$ only if task τ_j is indeed the task whose job $J_{j,b}$ is the unfinished carry-in job with the earliest release time. However, we do not know which task is actually task τ_j — in fact, it can be different for each job $J_{k,a}$ of task τ_k. Therefore, we take the maximum of Y_j^k for all the tasks τ_j in the task set. Therefore, without knowing task τ_j, we can bound the maximum total carry-in work $\widehat{R^k}$ by overestimating Y^k:

$$\widehat{R^k} \leq Y^k \leq \max_j Y_j^k \tag{5.9}$$

Both Y^k from Inequality (5.9) and $\sum_i X_i^k$ from Inequality (5.5) can be used to bound the carry-in work $\widehat{R^k}$. Hence, we can improve the basic test by using

$$\widehat{R^k} \leq \min\left(X^k, Y^k\right) \leq \min\left(\sum_i X_i^k, \max_j Y_j^k\right) \tag{5.10}$$

for the calculation of completion time in Formula (5.6).

5.3 Improving the Calculation for Completion Time

Finally, note that in Formula (5.6), we calculate the maximum number of interfering but not carry-in jobs using Equation (5.4), in which we assume that the completion time of task τ_k is exactly D_k. However, if task τ_k actually finishes earlier than its deadline, it may suffer from less interference. Such a calculation is no different than for a sequential task set on a single core, so we can similarly derive the improved calculation of $\widehat{n_i^k}$ using

$$\widehat{n_i^k} = \min\left(\left\lfloor \frac{D_k}{D_i} \right\rfloor, \left\lfloor \frac{D_k - f_k}{D_i} + 1 \right\rfloor\right) \tag{5.11}$$

We can then use this new calculation for $\widehat{n_i^k}$ in our calculation of interference, leading to a potentially tighter interference calculation.

Chapter 6

Resource Augmentation Bound of $2 - \frac{1}{m}$ for GEDF

In this chapter, we prove the resource augmentation bound of $2 - \frac{1}{m}$ for GEDF scheduling of arbitrary deadline tasks.

For sake of discussion, we convert the DAG representing a task into an equivalent DAG where each sub-node does $\frac{1}{m}$ unit of work. An example of this transformation of Task τ_1 in Figure 3.1 is shown in job W_1 in Figure 6.1 (see the upper job). A node with work w is split into a chain of mw sub-nodes with work $\frac{1}{m}$. For example, since in Figure 6.1 $m = 2$, node W_1^1 with worst-case execution time of 1 is split into 2 sub-nodes $W_1^{1,1}$ and $W_1^{1,2}$ each with length $\frac{1}{2}$. The original incoming edges come into the first node of the chain, while the outgoing edges leave the last node of the chain. This transformation does not change any other characteristic of the DAG, and the scheduling does not depend on this step — the transformation is done only for clarity of the proof.

6.1 Proof for Resource Augmentation Bound

First, some definitions. Since the GEDF scheduler runs on cores of speed $2 - \frac{1}{m}$, each step under GEDF can be divided into $(2m-1)$ sub-steps of length $\frac{1}{m}$. In each sub-step, each core can do $\frac{1}{m}$ units of work (i.e. execute one sub-node). In a GEDF scheduler, on an incomplete step, all ready nodes are executed (Observation 1). As in Chapter 4, we say that a sub-step is ***complete*** if all cores are busy, and ***incomplete*** otherwise. For each sub-step t, we define

$\mathcal{F}_{\mathcal{I}}(t)$ as the set of sub-nodes that have *finished* executing under an ideal scheduler after sub-step t, $\mathcal{R}_{\mathcal{I}}(t)$ as the set of sub-nodes that are *ready* (all their predecessors have been executed) to be executed by the ideal scheduler before sub-step t, and $\mathcal{D}_{\mathcal{I}}(t)$ as the set of sub-nodes completed by the ideal scheduler in sub-step t. Note that $\mathcal{D}_{\mathcal{I}}(t) = \mathcal{R}_{\mathcal{I}}(t) \cap \mathcal{F}_{\mathcal{I}}(t)$. We similarly define $\mathcal{F}_{\mathcal{G}}(t)$, $\mathcal{R}_{\mathcal{G}}(t)$, and $\mathcal{D}_{\mathcal{G}}(t)$ for GEDF scheduler.

Observation 1 *The GEDF scheduler completes all the ready nodes in an incomplete sub-step. That is,*

$$\mathcal{D}_{\mathcal{G}}(t) = \mathcal{R}_{\mathcal{G}}(t), \textit{ if } t \textit{ is incomplete sub-step,} \tag{6.1}$$

Note for the ideal scheduler, each original step consists of m sub-steps, while for GEDF with speed $2 - \frac{1}{m}$ each step consists of $2m - 1$ sub-steps. For example, in Figure 6.1 for step t_1, there are two sub-steps $t_{1(1)}$ and $t_{1(2)}$ under ideal scheduler, while under GEDF there is an additional $t_{1(3)}$ (since $2m - 1 = 3$).

Theorem 2 *If an ideal scheduler can schedule a task set τ (periodic or sporadic tasks with arbitrary deadlines) on a unit-speed system with m identical cores, then global EDF can schedule τ on m cores of speed $2 - \frac{1}{m}$.*

Proof. In a GEDF scheduler, on an incomplete sub-step, all ready sub-nodes are executed (Observation 1). Therefore, after an incomplete sub-step, GEDF must have finished all the released sub-nodes and hence must have done at least as much work as the ideal scheduler. Thus, for brevity of our proof, we leave out any time interval when all cores under GEDF are idling, since at this time GEDF has finished all available work and at this time the Theorem is obviously true. We define time 0 as the first instant when not all cores are idling under GEDF and time t as any time such that for every sub-step during time interval $[0, t]$ at least one core under GEDF is working. Therefore for every incomplete sub-step GEDF will finish at least 1 sub-node (i.e. $\frac{1}{m}$ unit of work). We also define sub-step 0 as the last sub-step before time 0 and hence by definition,

$$\mathcal{F}_{\mathcal{G}}(0) \supseteq \mathcal{F}_{\mathcal{I}}(0) \text{ and } |\mathcal{F}_{\mathcal{G}}(0)| \geq |\mathcal{F}_{\mathcal{I}}(0)| \tag{6.2}$$

For each time $t \geq 0$, we now prove the following: If the ideal unit-speed system can successfully schedule all tasks with deadlines in the time interval $[0, t]$, then on speed $2 - \frac{1}{m}$ cores, so can GEDF. Note again that during the interval $[0, t]$ an ideal scheduler and GEDF have tm and $2tm - t$ sub-steps respectively.

Case 1: In $[0, t]$, GEDF has at most tm incomplete sub-steps.

Since there are at least $(2tm - t) - tm = tm - t$ complete steps, the system can complete $|\mathcal{F}_{\mathcal{G}}(t)| - |\mathcal{F}_{\mathcal{G}}(0)| \geq m(tm - t) + (tm) = tm^2$ work, since each complete sub-step can finish executing m sub-nodes and each incomplete sub-step can finish executing at least 1 sub-node. We define $I(t)$ as the set of all sub-nodes from jobs with absolute deadlines no later than t. Since the ideal scheduler can schedule this task set, we know that $|I(t)| - |\mathcal{F}_{\mathcal{I}}(0)| \leq mt * m = tm^2$, since the ideal scheduler can only finish at most m sub-nodes in each sub-step and during $[0, t]$ there are mt sub-steps for the ideal scheduler. Hence, we have $|\mathcal{F}_{\mathcal{G}}(t)| - |\mathcal{F}_{\mathcal{G}}(0)| \geq |I(t)| - |\mathcal{F}_{\mathcal{I}}(0)|$. By eq.(6.2), we get $|\mathcal{F}_{\mathcal{G}}(t)| \geq |I(t)|$. Note that jobs in $I(t)$ have earlier deadlines than the other jobs, so under GEDF, no other jobs can interfere with them. The GEDF scheduler will never execute other sub-nodes unless there are no ready sub-nodes from $I(t)$. Since $|\mathcal{F}_{\mathcal{G}}(t)| \geq |I(t)|$, i.e. GEDF has finished at least as many sub-nodes as the number in $I(t)$, this implies that GEDF must have finished all sub-nodes in $I(t)$. Therefore, GEDF can meet all deadlines since it has finished all work that needed to be done by time t.

Case 2: In $[0, t]$, GEDF has more than tm incomplete sub-steps.

For each integer s we define $f(s)$ as the first time instant such that the number of incomplete sub-steps in interval $[0, f(s)]$ is exactly s. Note that the sub-step $f(s)$ is always incomplete, since otherwise it wouldn't be the first such instant. We show, via induction, that $\mathcal{F}_{\mathcal{I}}(s) \subseteq \mathcal{F}_{\mathcal{G}}(f(s))$. In other words, after $f(s)$ sub-steps, GEDF has completed all the nodes that the ideal scheduler has completed after s sub-steps.

Base Case: For $s = 0$, $f(s) = 0$. By eq.(6.2), the claim is vacuously true.

Inductive Step: Suppose that for $s - 1$ the claim $\mathcal{F}_{\mathcal{I}}(s-1) \subseteq \mathcal{F}_{\mathcal{G}}(f(s-1))$ is true. Now, we prove that $\mathcal{F}_{\mathcal{I}}(s) \subseteq \mathcal{F}_{\mathcal{G}}(f(s))$.

In $(s-1, s]$, the ideal system has exactly 1 sub-step. So,

$$\mathcal{F}_{\mathcal{I}}(s) = \mathcal{F}_{\mathcal{I}}(s-1) \cup \mathcal{D}_{\mathcal{I}}(s) \subseteq \mathcal{F}_{\mathcal{I}}(s-1) \cup \mathcal{R}_{\mathcal{I}}(s) \tag{6.3}$$

Since $\mathcal{F}_{\mathcal{I}}(s-1) \subseteq \mathcal{F}_{\mathcal{G}}(f(s-1))$, all the sub-nodes that are ready before sub-step s for the ideal scheduler, will either have already been executed or are also ready for the GEDF scheduler one sub-step after sub-step $f(s-1)$; that is,

$$\mathcal{F}_{\mathcal{I}}(s-1) \cup \mathcal{R}_{\mathcal{I}}(s) \subseteq \mathcal{F}_{\mathcal{G}}(f(s-1)) \cup \mathcal{R}_{\mathcal{G}}(f(s-1)+1) \tag{6.4}$$

For GEDF, from sub-step $f(s-1)+1$ to $f(s)$, all the ready sub-nodes with earliest deadlines will be executed and then new sub-nodes will be released into the ready set. Hence,

$$\begin{aligned} & \mathcal{F}_{\mathcal{G}}(f(s-1)) \cup \mathcal{R}_{\mathcal{G}}(f(s-1)+1) \\ \subseteq\ & \mathcal{F}_{\mathcal{G}}(f(s-1)+1) \cup \mathcal{R}_{\mathcal{G}}(f(s-1)+2) \\ \subseteq\ & \ldots \subseteq \mathcal{F}_{\mathcal{G}}(f(s)-1) \cup \mathcal{R}_{\mathcal{G}}(f(s)) \end{aligned} \tag{6.5}$$

Since sub-step $f(s)$ for GEDF is always incomplete,

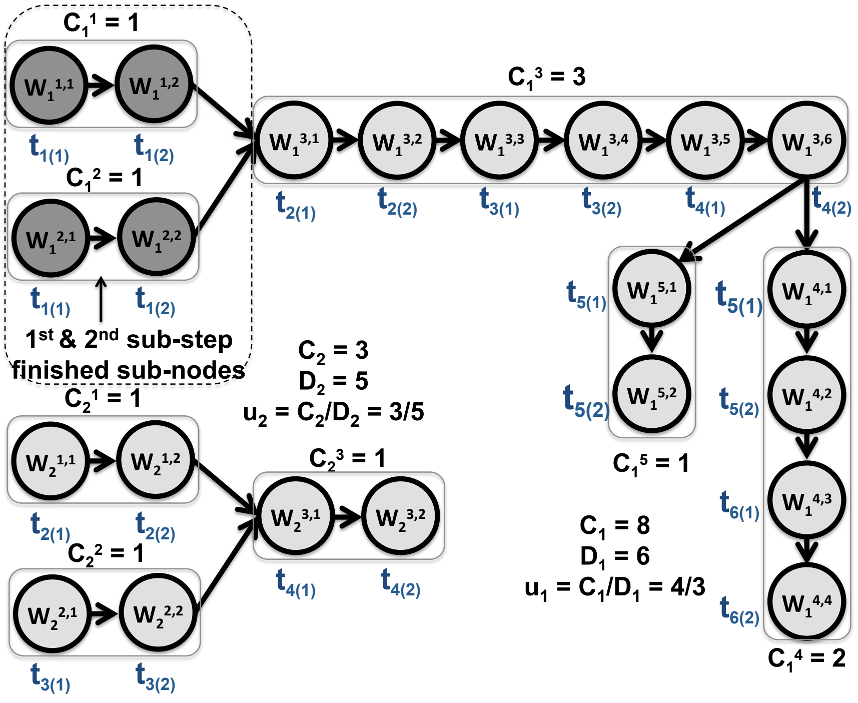
$$\begin{aligned} & \mathcal{F}_{\mathcal{G}}(f(s)) \\ =\ & \mathcal{F}_{\mathcal{G}}(f(s)-1) \cup \mathcal{D}_{\mathcal{G}}(f(s)) \\ =\ & \mathcal{F}_{\mathcal{G}}(f(s)-1) \cup \mathcal{R}_{\mathcal{G}}(f(s)) \qquad \text{(from eq.(6.1))} \\ \supseteq\ & \mathcal{F}_{\mathcal{G}}(f(s-1)) \cup \mathcal{R}_{\mathcal{G}}(f(s-1)+1) \qquad \text{(from eq.(6.5))} \\ \supseteq\ & \mathcal{F}_{\mathcal{I}}(s-1) \cup \mathcal{R}_{\mathcal{I}}(s) \qquad \text{(from eq.(6.4))} \\ \supseteq\ & \mathcal{F}_{\mathcal{I}}(s) \qquad \text{(from eq.(6.3))} \end{aligned}$$

By time t, there are mt sub-steps for the ideal scheduler, so GEDF must have finished all the nodes executed by the ideal scheduler at sub-step $f(mt)$. Since there are exactly mt incomplete sub-steps in $[0, f(mt)]$ and since the number of incomplete sub-steps by time t is at least mt, the time $f(mt)$ is no later than time t. Since the ideal system does not miss any deadline by time t, GEDF also meets all deadlines. $\square$

6.2 An Example Providing an Intuition for the Proof

We provide an example in Figure 6.1 to illustrate the proof of Case 2 and compare the execution trace of an ideal scheduler (this scheduler is only considered "ideal" in the sense that it makes all the deadlines) and GEDF. In addition to task 1 from Figure 3.1, Task τ_2 consists of two nodes connected to another node, all with execution time of 1 (each split into 2 sub-nodes in the figure). All tasks are released by time t_0. The system has 2 cores, so GEDF has a resource augmentation bound of 1.5. Figure 6.1(a) is the execution for the ideal scheduler on unit-speed cores, while Figure 6.1(b) shows the execution under GEDF on speed 2 cores. One step is divided into 2 and 3 sub-steps, representing the speedup of 1 and 1.5 for the ideal scheduler and GEDF respectively.

Since the critical-path length of Task τ_1 is equal to its deadline, intuitively it should be executed immediately even though it has the latest deadline. That is exactly what the ideal scheduler does. However, GEDF (which does not take critical-path length into consideration) will prioritize Task τ_2 first. If GEDF is only on a unit-speed system,Task τ_1 will miss deadline. However, when GEDF gets speed-1.5 cores, all jobs are finished in time. To illustrate Case 2 of the above theorem, consider $s = 2$. Since $t_{2(3)}$ is the second incomplete sub-step under GEDF, $f(s) = 2(3)$. All the nodes finished by the ideal scheduler after second sub-step (shown above in dark grey) have also been finished under GEDF by step $t_{2(3)}$ (shown below in dark grey).


(a) Scheduled under unit-speed ideal scheduler.

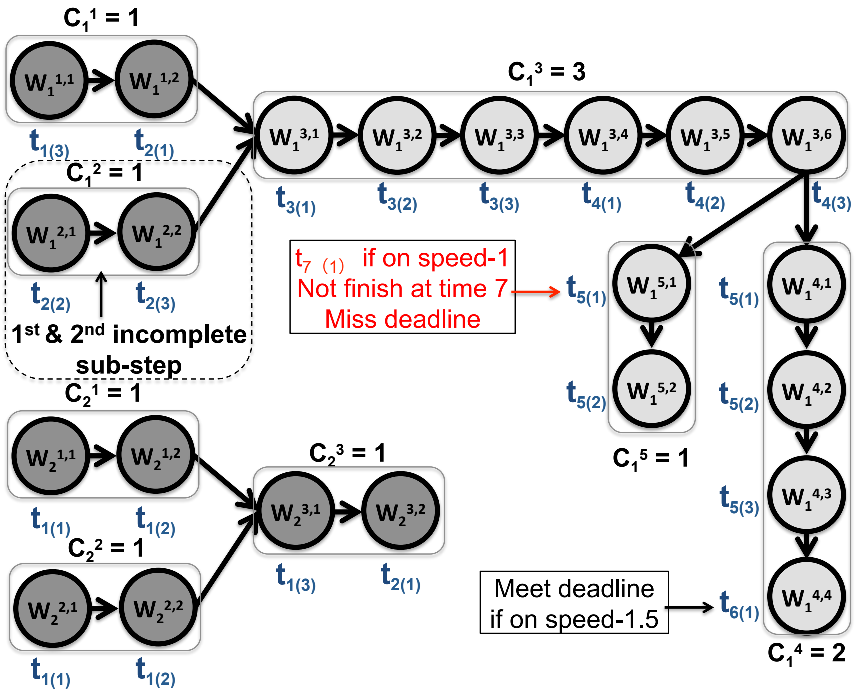

(b) Scheduled under 2-speed GEDF scheduler.

Figure 6.1: Examples of task set execution on 2 cores.

Chapter 7

Lower Bound on Capacity Augmentation Bound of GEDF

While the above proof guarantees a bound, since the ideal scheduler is not known, given a task set, we cannot tell if it is feasible on speed-1 cores. Therefore, we cannot tell if it is schedulable by GEDF on cores with speed $2 - \frac{1}{m}$.

One standard way to prove resource augmentation bounds is to use lower bounds on the ideal scheduler, such as Inequalities 3.1 and 3.2. As previously stated, we call the resource augmentation bound proven using these lower bounds a ***capacity augmentation bound*** in order to distinguish it from the augmentation bound described above. To prove a capacity augmentation bound of b under GEDF, one must prove that if Inequalities 3.1 and 3.2 hold for a task set on m unit-speed cores, then GEDF can schedule that task set on m cores of speed b. Hence, the capacity augmentation bound is also an easy schedulability test.

First, we demonstrate a counter-example to show proving a capacity augmentation bound of 2 for GEDF is impossible.

In particular, in Figure 7.1 we show a task set that satisfies inequalities 3.1 and 3.2, but cannot be scheduled on m cores of speed 2 by GEDF. In this example, $m = 6$ as shown in Figure 7.2. The task set has two tasks. All values are measured on a unit-speed system, shown in Figure 7.1. Task τ_1 has 13 nodes with total execution time of 440 and period of 88, so its utilization is 5. Task τ_2 is a single node, with execution time and implicit deadline both 60 and hence utilization of 1. Note the total utilization (6) is exactly equal

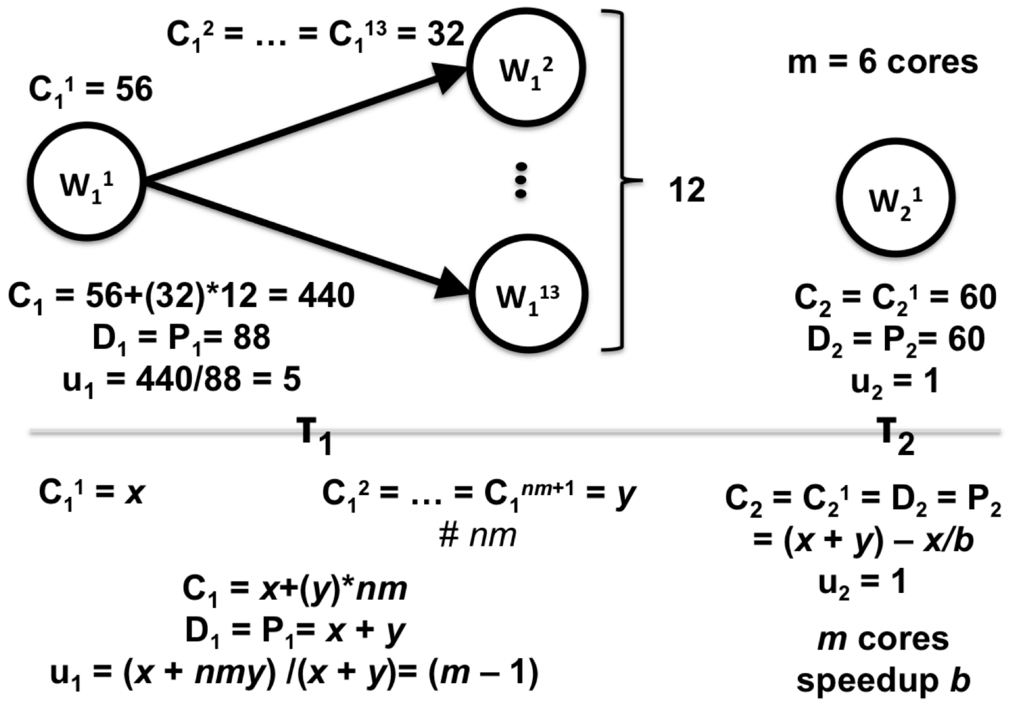


Figure 7.1: Structure of the task set that demonstrates GEDF does not provide a capacity augmentation bound less than $(3 + \sqrt{5})/2$

to m, satisfying inequality 3.2. The critical-path length of each task is equal to its deadline, satisfying inequality 3.1.

The execution trace of the task set on a 2-speed 6-core core under GEDF is shown in Figure 7.2. The first task is released at time 0 and is immediately executed by GEDF. Since the system under GEDF is at speed 2, $W_1^{1,1}$ finishes executing at time 28. GEDF then executes 6 out of the 12 parallel nodes from Task τ_1. At time 29, task τ_2 is released. However, its deadline is $r_2 + D_2 = 29 + 60 = 89$, which is later than deadline 88 of task τ_1. Nodes from task τ_1 are not preempted by task τ_2 and continue to execute until all of them finish their work at time 60. Task τ_1 successfully meets its deadline. The GEDF scheduler finally gets to execute task τ_2 and finishes it at time 90, so task τ_2 just fails to meet its deadline of 89. Note that this is not a counter-example for the resource augmentation bound shown in Theorem 2, since no scheduler can schedule this task set on unit-speed system either.

Second, we demonstrate that one can construct task sets that require capacity augmentation of at least $\frac{3+\sqrt{5}}{2}$ to be schedulable by GEDF. We generate task sets with two tasks whose

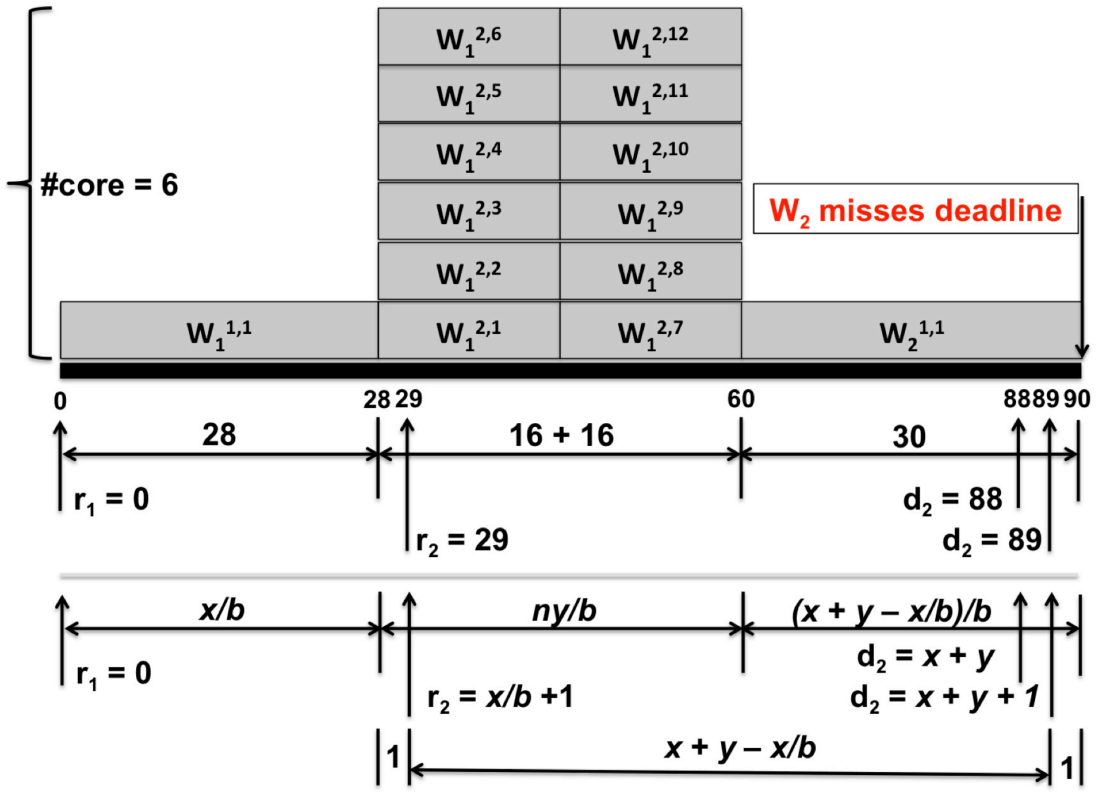


Figure 7.2: Execution of the task set under GEDF at speed 2

structure depends on m, speedup factor b and a parallelism factor n, and show that for large enough m and n, the capacity augmentation required is at least $b \geq \frac{3+\sqrt{5}}{2}$. As in the lower part of Figure 7.1, task τ_1 is structured as a single node with work x followed by nm nodes with work y. Its critical-path length is $x + y$ and so is its deadline. The utilization of task τ_1 is set to be $m - 1$, hence

$$m - 1 = \frac{x + nmy}{x + y} \tag{7.1}$$

Task τ_2 is structured as a single node with work and deadline equal to $x + y - \frac{x}{b}$ (hence utilization 1). Therefore, the total task utilization is m and Inequalities 3.1 and 3.2 are met. As the lower part of Figure 7.2 shows, Task τ_2 is released at time $\frac{x}{b} + 1$.

We want to generate a counter example, so we want task τ_2 to barely miss the deadline by 1 sub-step. In order for this to occur, we must have

$$(x + y - \frac{x}{b}) + 2 = \frac{ny}{b} + \frac{1}{b}(x + y - \frac{x}{b}). \tag{7.2}$$

Reorganizing and combining eq.(7.1) and eq.(7.2), we get

$$(m-2)b^2 = ((3bn - b - n - b^2 n + 1)m + (b^2 - 2bn - 1))y \tag{7.3}$$

In the above equation, for large enough m and n, we have $(3bn - b - n - b^2 n + 1) > 0$, or

$$1 < b < \frac{3}{2} - \frac{1}{2n} + \frac{1}{2}\sqrt{5 - \frac{2}{n} + \frac{1}{n^2}} \tag{7.4}$$

So, there exists a counter-example for any speedup b which satisfies the above conditions. Therefore, the capacity augmentation required by GEDF is at least $\frac{3+\sqrt{5}}{2}$. The example above with speedup of 2 comes from such a construction. Another example with speedup 2.5 can be obtained when $x = 36050$, $y = 5900$, $m = 120$ and $n = 7$.

Chapter 8

Simulation Evaluation

In this chapter, we present results of our simulation results of the performance of GEDF and the robustness of our capacity augmentation bound.[1] We randomly generate task sets that fully load machines, and then simulate their execution on machines of increasing speed. The capacity augmentation bound for GEDF predicts that all task sets should be schedulable by the time the core speed is increased to $4 - \frac{2}{m}$. In our simulations, all task sets became schedulable before the speed reached 2.

We also compared GEDF with the another method that provides capacity bounds for scheduling multiple DAGs (with a DAG's utilization potentially more than 1) on multicores [63]. In this method, which we call ***DECOMP***, tasks are decomposed into sequential subtasks and then scheduled using GEDF.[2] We find that GEDF without decomposition performs better than DECOMP for most task sets.

8.1 Task Sets and Experimental Setup

We generate two types of DAG tasks for evaluation. For each method, we first fix the number of nodes n in the DAG and then add edges.

(1) Erdos-Renyi method $G(n,p)$ **[21]:** For a DAG with n nodes, there are $n^2/2$ possible valid edges. We go through each valid edge and add it with probability p, where p is a

[1]Note that, due to the lack of a schedulability test, it is difficult to experimentally test the resource augmentation bound of $2 - 1/m$ or through simulation.

[2]For DECOMP, end-to-end deadline (instead of decomposed subtask's deadline) miss ratios were reported.

parameter (i.e. DAGs with e valid edges will have ep edges in average). Note that this method does not necessarily generate a connected DAG. Although the bound also does not require the DAG of a task to be fully connected, connecting more of its nodes can make it harder to schedule. Hence, we modified the algorithm slightly in the last step, to add the fewest edges needed to make the DAG connected.

(2) Special synchronous task $L(n, m)$: As shown in Figure 7.1, synchronous tasks like it, in which highly parallel segments follow sequential segments, makes scheduling difficult for GEDF since they can cause deadline misses for other tasks. Therefore, we generate task sets with alternating sequential and highly parallel segments. Tasks in $L(n, m)$ (m is the number of processors) are generated in the following way. While the total number of nodes in the DAG is smaller than n, we add another sequential segment by adding a node, then generate the next parallel layer randomly. For each parallel layer, we uniformly generate a number t between 1 and $\lfloor \frac{n}{m} \rfloor$, and set the number of nodes in the segment to be $t * m$.

Given a task structure generated by either of the above methods, worst-case execution times for individual nodes in the DAG are picked randomly between $[50, 500]$. The critical-path length L_i for each task is then calculated. After that, we assign a period (equal to its deadline) to each task. Note that a valid deadline is at least the critical-path length. Two types of periods were assigned to tasks.

(1) Harmonic Period: All tasks have periods that are integral powers of 2. We first compute the smallest value a such that 2^a is larger than a task's critical-path length L_i. We then randomly assign the period either 2^a, 2^{a+1} or 2^{a+2} to generate tasks with varying utilization. All tasks are then released at the same time and simulated for the hyper-period of the tasks.

(2) Arbitrary Period: An arbitrary period is assigned in the form $(L_i + \frac{C_i}{0.5m}) * (1 + 0.25 * gamma(2, 1))$, where $gamma(2, 1)$ is the Gamma distribution with $k = 2$ and $\theta = 1$. The formula is designed such that, for small m, tasks tend to have smaller utilization. This allows us to have a reasonable number of tasks in a task set for any value of m.

Several parameters were varied to test the system: $G(n, p)$ vs $L(n, m)$ DAGs, different p for $G(n, p)$, harmonic vs arbitrary Periods, numbers of Core m (4, 8, 16, 32, 64). Task sets are created by adding tasks to them until the total utilization reaches 99% of m. Each task set is simulated for 20 times the longest period in a task set. For each setting, we generated 1000 task sets. We first simulated the task sets for each setting on cores of speed 1, and increased the speed in steps of 0.2. For each setting, we measured the ***failure ratio*** — the number of task sets where *any* task missed its deadline over the number of total simulated task sets. We stopped increasing the speed for a task set when no deadline was missed.

8.2 Simulation Results

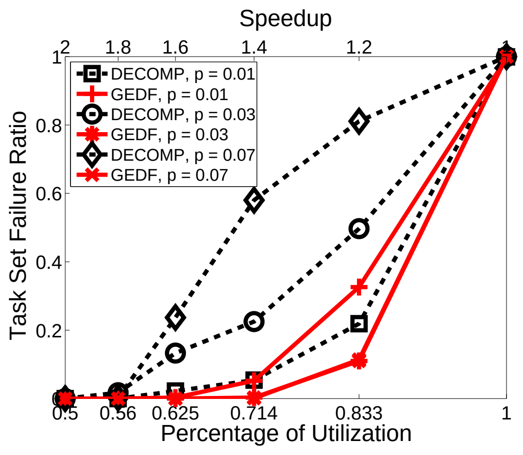
We first present the results for task sets generated by the Erdos-Renyi method for various setting of p and different numbers of processors to see the effect of these parameters on the performance of GEDF.

8.2.1 Erdos-Renyi Method

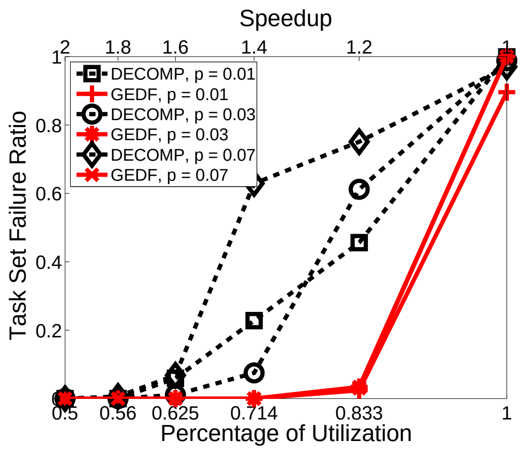
For this method, we generate two types of task sets: (1) **Fixed p task sets**: In this setting, all task sets have the same p. We varied the values of p over $\{0.01, 0.02, 0.03, 0.05, 0.07, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8 \text{ and } 0.9\}$. (2) **Random p task sets**: We also generated task sets where each task has a different, randomly picked, value of p.

Figures 8.1(a), 8.1(b) and 8.1(c) show the failure rate for fixed-p task sets as we varied p and kept m constant at 64. GEDF without decomposition outperforms DECOMP for all settings of p. It appears that GEDF has the hardest time when $p \leq 0.1$, where tasks are more sequential. But even then, all tasks are schedulable with speed 1.8. At $p > 0.1$, GEDF never requires speed more than 1.4, while DECOMP often requires a speed of 2 to schedule all task sets. We can also see that different task sets with different p values affect GEDF less than DECOMP. Trends are similar for other values of m.

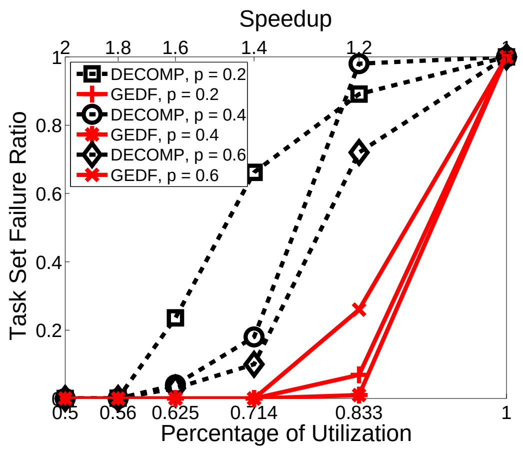
Figures 8.1(d), 8.1(e) and 8.1(f) show the failure rate for fixed-p task sets as we varied p and kept m constant at 16. GEDF without decomposition still outperforms DECOMP for almost all cases. Comparing the results between 64-core and 16-core task sets with same p, we can see that DECOMP improves greatly, while GEDF only improves slightly. This is mostly



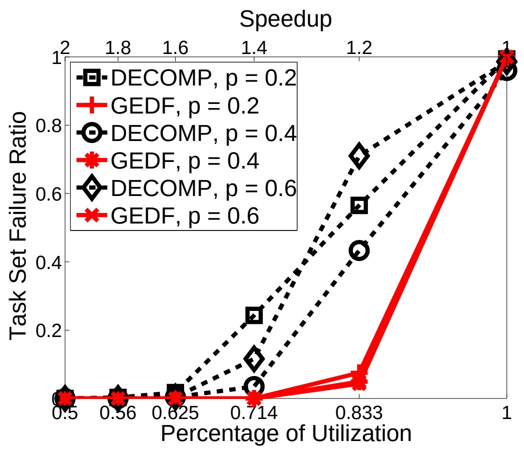
(a) For $p = (0.01, 0.03, 0.07)$ and $m = 64$.



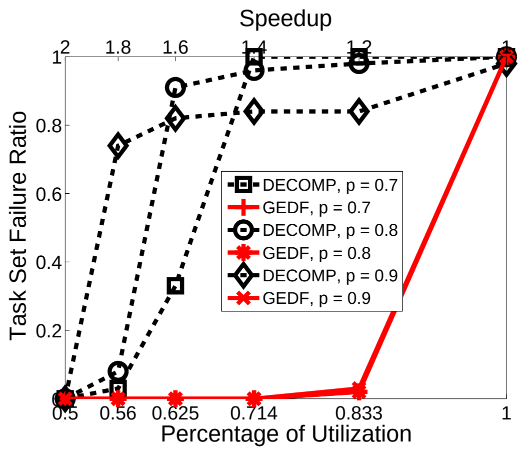
(d) For $p = (0.01, 0.03, 0.07)$ and $m = 16$.



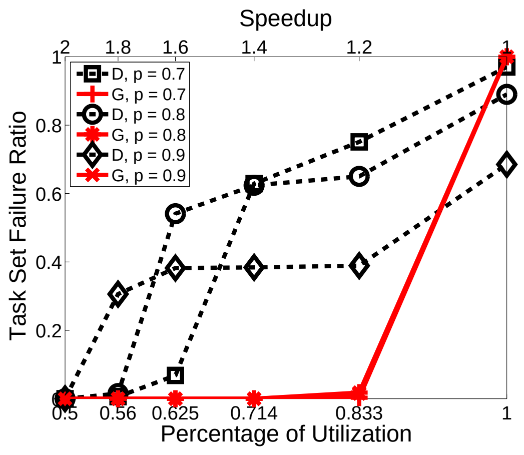
(b) For $p = (0.2, 0.4, 0.6)$ and $m = 64$.



(e) For $p = (0.2, 0.4, 0.6)$ and $m = 16$.



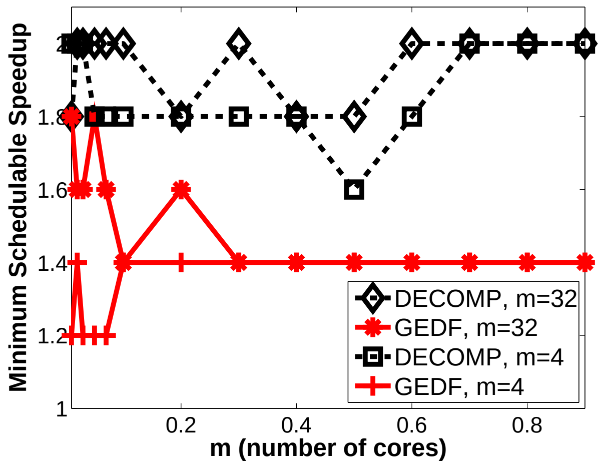
(c) For $p = (0.7, 0.8, 0.9)$ and $m = 64$.



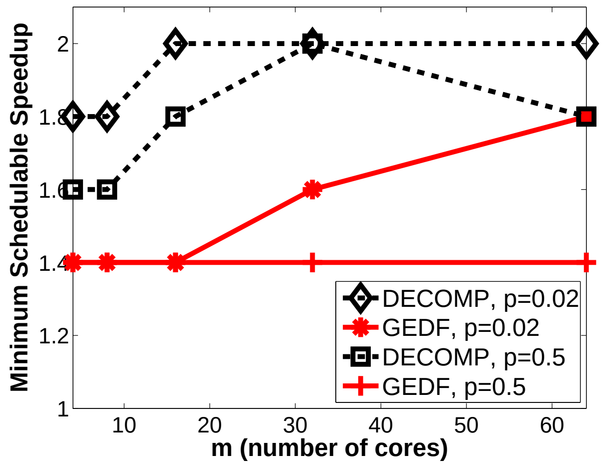
(f) For $p = (0.7, 0.8, 0.9)$ and $m = 16$.

Figure 8.1: Failure ratio of GEDF (solid line) vs. DECOMP (dashed line) for $G(n,p)$ tasks with different task set utilization percentages (speedups). The left three figures show the results for 64-core, and right three for 16-core. From top down, figures show results with small, medium and large values of p respectively.

because for GEDF, most task sets are schedulable at the speedup of 1.4. This required speedup might have approached to the limit, so there is no more space for improvement.



(a) Minimum schedulable speedup as p changes for different $m = (32, 8)$.

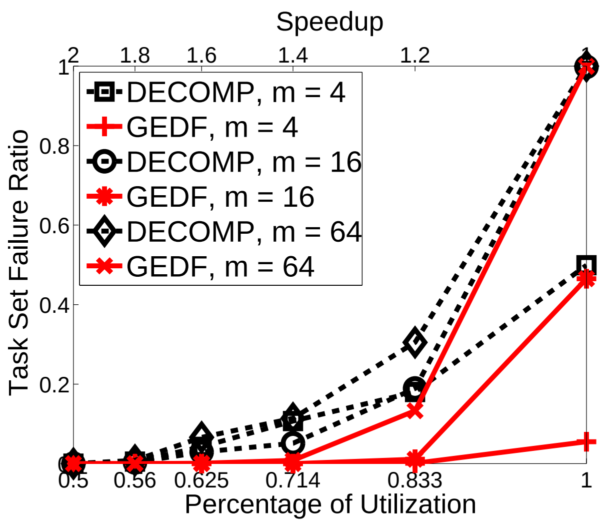


(b) Minimum schedulable speedup as m changes for different $p = (0.02, 0.5)$.

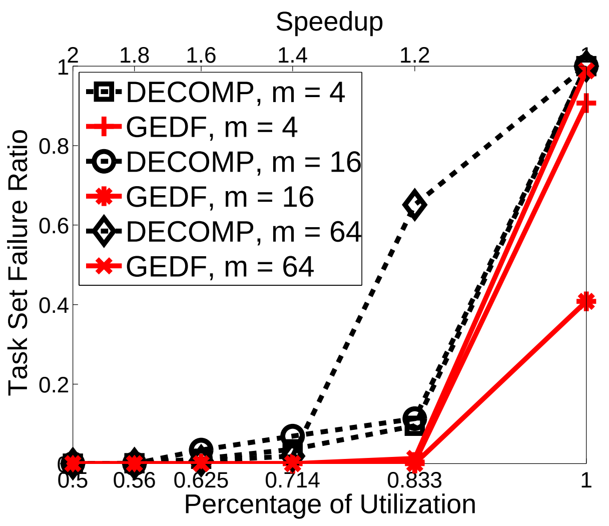
Figure 8.2: The left figure shows the effect of varying p on the speedup required to make all task sets schedulable.The right figure shows the effect of varying m on the speedup required to make all task sets schedulable. (harmonic period)

In Figure 8.1, we show detailed results for 64 and 16-core simulation results. The results for 32, 8 and 4-core have a similar trend: GEDF performs better than DECOMP; generally both schedulers perform better with lower cores. Figure 8.2 shows the minimum speedup at which all task sets are schedulable. In particular, in Figure 8.2(a) we can see that with fewer cores, both schedulers generally require the same or less speedup to schedule all 1000 task sets. The trend with different p is less obvious. It seems task sets with more near-sequential tasks (low p) are harder to schedule in general. However, highly connected DAGs (high p) are hard only for DECOMP to schedule, but not for GEDF. This is may because those DAGs make DECOMP harder to generate good decomposition results. For $p = 0.02$ and $p = 0.5$, in Figure 8.2(b) we vary m. Results for other p and m are similar. This figure also indicates that GEDF without decomposition generally needs less speedup to schedule the same task sets. Again, increasing m increases the speedup required in most cases.

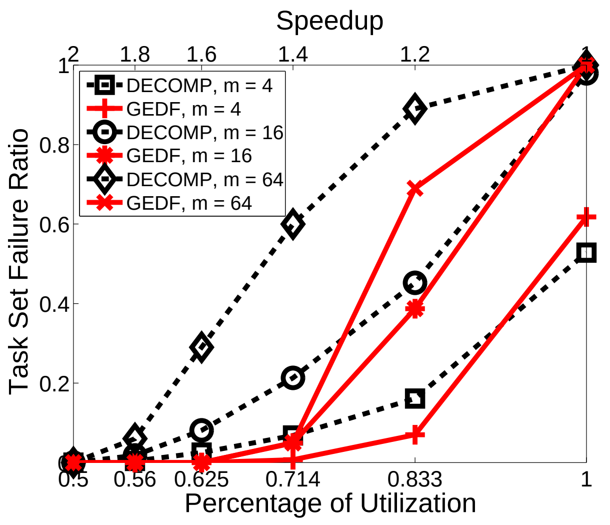
We now see the effect of m. In order to keep the figures from getting too cluttered, from now on, we only show results with $m = 4$, 16 and 64. The trends for $m = 8$ and 32 are similar and their curves usually lie in between 4 and 64. Figure 8.3(a) shows the failure ratio of the



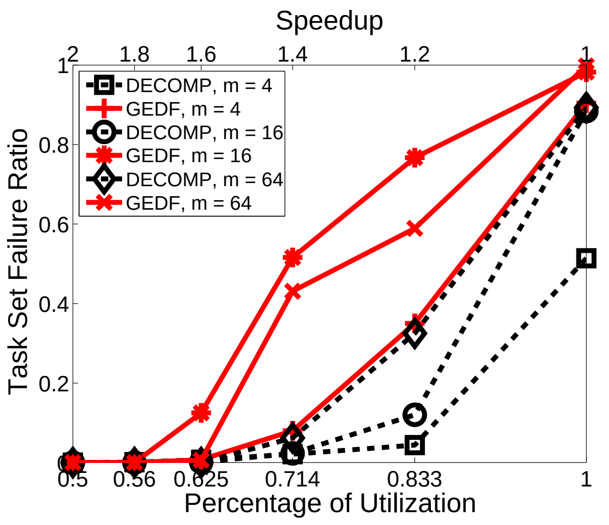
(a) Comparison as m changes ($G(n, p)$ tasks, $p = 0.02$, harmonic period).



(b) Comparison as m changes ($G(n, p)$ tasks, $p = 0.02$, arbitrary period).



(c) Comparison as m changes($G(n, p)$ tasks, random p, harmonic period).



(d) Comparison as m changes ($L(n, m)$ tasks, harmonic period).

Figure 8.3: Performance of GEDF (solid line) vs. DECOMP (dashed line) for different values of m. GEDF is always better than DECOMP. In general, increasing the number of processors generally increases failure rates.

fixed-p task sets as we kept p constant at 0.02 and varied m. Again, GEDF outperforms DECOMP for all settings, even though small p is harder for GEDF. When $m = 4$, GEDF can schedule all task sets at speed 1.4. The increase of m does not influence DECOMP much, while it becomes slightly harder for GEDF to schedule a few (out of 1000) task sets.

A similar trend holds in the other cases in Figure 8.3 which show the results for different parameter settings (arbitrary instead of harmonic period, random p instead of fixed p, etc).

Figure 8.3 also allows us to see other effects. For instance, we can compare the failure rates of harmonic vs. arbitrary periods by comparing Figures 8.3(b) and 8.3(a). The figures suggest that, in general, the harmonic and arbitrary period task sets behave similarly. It does appear that tasks with arbitrary periods are slightly easier to schedule, especially for GEDF. This is at least partially explained by the observation that, with harmonic periods, many tasks have the same deadline, making it difficult for GEDF to distinguish between them. These trends also hold for other parameter settings, and therefore we omit those figures to reduce redundancy.

We also compare the effect of fixed vs. random p by comparing Figure 8.3(c) to Figure 8.3(a). The former shows the failure ratio for GEDF and DECOMP for task sets where p is not fixed, but is randomly generated for each task, as we vary m. Again, GEDF outperforms DECOMP. Note, however, that GEDF appears to have a harder time for random p than in the fixed p experiment.

8.2.2 Synchronous Method

Figure 8.3(d) shows the comparison between GEDF and DECOMP with varying m for specially constructed synchronous task sets. In this case, the failure ratio for GEDF is higher than for task sets generated with the Erdos-Renyi Method. We can also see that sometimes DECOMP outperforms GEDF in terms of failure ratio and required speedup. This indicates that synchronous tasks with highly parallel segments are indeed more difficult for GEDF to schedule. However, even in this case, we never require a speedup of more than 2. Even though Figure 7.1 demonstrates that there exist task sets that require speedup of more than 2, such pathological task sets never appeared in our randomly generated sample.

In conclusion, simulation results indicate that GEDF performs better than predicted by the capacity augmentation bound. For most task sets, GEDF is also better than DECOMP.

Chapter 9

Parallel GEDF Platform

To demonstrate the feasibility of parallel GEDF scheduling, we implemented a simple prototype platform called ***PGEDF*** by combining GNU OpenMP runtime system and the LITMUS^{RT} system. PGEDF is a straightforward implementation based on these off-the-shelf systems and simply sets appropriate parameters for both OpenMP and LITMUS^{RT} without modifying either. It is also easy to use this platform; the user can write tasks as programs with standard OpenMP directives and compile them using the g++ compiler. In addition, the user provides a task-set configuration file that specifies the tasks in the task-set and their deadlines. The platform uses the GEDF plug-in of LITMUS^{RT} to execute the tasks. To validate the theory we present, PGEDF is configured for CPU intensive workloads and cache or I/O effects are beyond the scope this paper. We first describe the relevant aspects of OpenMP and LITMUS^{RT} and then describe the specific settings that allow us to run parallel real-time tasks on this platforms.

9.1 Background

We briefly introduce the GNU OpenMP runtime system and the LITMUS^{RT} patched Linux operating system, with an emphasis on the particular features that our PGEDF relies on in order to realize parallel GEDF scheduling.

9.1.1 OpenMP Overview

OpenMP is a specification for parallel programs that defines an open standard for parallel programming in C, C++, and Fortran [60]. It consists of a set of compiler directives, library routines and environment variables, which can influence the runtime behavior. Our PGEDF implementation uses a GNU implementation of OpenMP runtime system (***GOMP***), which is part of the GCC (GNU Compiler Collection).

In OpenMP, *logical parallelism* in a program is specified through compiler pragma statements. For example, a regular for-loop can be transformed into a parallel for-loop by simply adding `#pragma omp parallel for` above the regular `for` statement. This gives the system permission to execute the iterations of the loop independently in parallel with each other. If the compiler does not support OpenMP, the pragma will be ignored, and the for-loop will be executed sequentially. On the other hand, if OpenMP is supported, then the runtime system can choose to execute these iterations in parallel. OpenMP also supports other parallel constructs; however, for our prototype of PGEDF, we only support parallel synchronous tasks. These tasks are described as a series of segments which can be parallel or sequential. A parallel segment is described as a parallel for-loop while a sequential segment consists of arbitrary sequential code. Therefore, we will restrict our attention to parallel for-loops.

We now briefly describe the OpenMP (specifically GOMP) runtime strategy for such programs. Under GOMP, each OpenMP program starts with a single thread, called the ***master thread***. During execution, when the runtime system encounters the first parallel section of the program, the master thread will create a ***thread pool*** and assign that ***team*** of threads to the parallel region. The threads created by the master thread in the thread pool are called ***worker threads***. The number of worker threads can be set by the user.

The master thread executes the sequential segments. In parallel segments (parallel for-loops), each iteration is considered a unit of work and ***maps*** (distributes) the work to the team of threads according to the chosen policies, as specified by arguments passed to the `omp_set_schedule()` function call. In OpenMP, there are three different kind of policies: ***dynamic***, ***static*** and ***guided*** policies. In the ***static 1*** policy, all iterations are divided among the team of threads at the start of the loop, and iterations are distributed to threads one by one: each thread in the team will get one iteration at a time in a round robin manner.

Once a thread finishes all its assigned work from a particular parallel segment, it ***waits*** for all other threads in the team to finish before moving on to the next segment of the task. The waiting policy can be set by via the environment variable `OMP_WAIT_POLICY`. Using passive synchronization, waiting threads are blocked and put into the Linux sleep queue, where they do not consume CPU cycle while waiting. On the other hand, active synchronization would let waiting threads spin without yielding the processors, which would consume CPU cycles while waiting.

One important property of the GOMP, upon which our implementation relies, is that the team of threads for each program is ***reusable***. After the execution of a parallel region, the threads in the team are not destroyed. Instead, all threads except the master thread wait for the next parallel segment, again according to the policy set by `OMP_WAIT_POLICY`. The master thread continues the sequential segment. When it encounters the next parallel segment GOMP runtime system detects that it already has a team of threads available to it, and simply reuses them for executing this segment, as before.

9.1.2 LITMUS$^{\text{RT}}$ Overview

LITMUS$^{\text{RT}}$ (Linux Testbed for Multiprocessor Scheduling in Real-Time Systems) is an algorithm-oriented real-time extension of Linux [16]. It focuses on multiprocessor real-time scheduling and synchronization. Many real-time schedulers, including global, clustered, partitioned and semi-partitioned schedulers are implemented as plug-ins for Linux. Users can use these schedulers for real-time tasks, and standard Linux scheduling for non-real-time tasks.

In LITMUS$^{\text{RT}}$, the GEDF implementation is meant for sequential tasks. A typical LITMUS$^{\text{RT}}$ real-time program contains one or more ***rt_tasks*** (real-time tasks), which are released periodically. In fact, each `rt_task` can be regarded as a ***rt_thread***, which is a standard Linux thread with real-time parameters. Under the GEDF scheduler, a `rt_task` can be suspended and migrated to a different CPU core according to the GEDF scheduling strategy. The platform consists of three main data structures to hold these tasks: a release queue, a one-to-one processor mapping, and a shared ready queue. The release queue is implemented as a priority queue with a clock tick handler, and is used to hold waiting-to-be-released jobs. The one-to-one processor mapping has the thread that corresponds to each job that is currently executing on each processor. The ready queue (shared by all processors) is implemented as a priority queue by using binomial heaps for fast queue-merge operations triggered by jobs with coinciding release times.

In order to run a sequential task as a real-time task under GEDF, LITMUS^{RT} provides an interface to configure a thread as an `rt_tasks`. The following steps must be taken to properly configure these [18]:

1. First, function `init_rt_thread()` is called to initialize the user-space real-time interface for the thread.
2. Then, the real-time parameters of the thread are set by calling `set_rt_task_param(getid(),&rt_task_param)`: the `getid()` function will return the actual thread ID in the system; the real-time parameters, including period, relative deadline, execution time and budget policy, are stored in the `rt_task_param` structure; these parameters will then be stored in the TCB (thread control block) using the unique thread ID and they will be validated by the kernel.
3. Finally, `task_mode(LITMUS_RT_TASK)` is called to start running the thread as a real-time task.

The periodic execution of jobs of `rt_tasks` is achieved by calling LITMUS^{RT} system calls as well. In particular, after each period, `sleep_next_period()` must be called to ask LITMUS^{RT} to move the thread from the run queue to the release queue. The thread sleeps in the release queue and the GEDF scheduler within the LITMUS^{RT} will automatically move it to the ready queue at its next absolute release time. The thread will eventually be automatically woken up and executed according to GEDF priority based on its absolute deadline.

9.2 PGEDF Platform Implementation

Now we describe how our PGEDF platform integrates the GOMP runtime with GEDF scheduling in LITMUS^{RT} to execute parallel real-time tasks. The PGEDF platform provides two key features: parallel task execution and real-time GEDF scheduling. The GOMP runtime system is used to perform parallel execution of each task, while real-time execution and GEDF scheduling is realized by the LITMUS^{RT} GEDF plug-in.

9.2.1 Programming Interface

Currently, PGEDF only supports synchronous task sets with implicit deadlines — tasks which consist of a sequence of segments and each segment is either a parallel segment (specified using a parallel-for loop) or a sequential segment (specified as regular code).

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

Figure 9.1: Task Program Format

The task structure is shown in Figure 9.1. Tasks are C or C++ programs that include a header file (`task.h`) and conform to a simple structure: instead of a `main` function, a programmer specifies a `run` function, which is executed when a job of the task is invoked. Tasks can also specify optional `initialize` and `finalize` functions, each of which (if not null) will be called once, before the first and after the last call to the run function, respectively. These optional functions let tasks set up and clean up resources as needed.

Additionally, a configuration file must be specified for the task set, specifying runtime parameters (such as program name and arguments) and real-time parameters (such as worst-case execution time, critical-path length, and period) for each task in the task set. This separate specification makes it flexible and easy to maintain; e.g., we do not have to recompile tasks in order to change timing constraints. The configuration file format is shown in Figure 9.2.

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

Figure 9.2: Format of the Configuration File

9.2.2 PGEDF Operation

Unlike sequential tasks where there is only one thread per `rt_task`, for parallel tasks there is a team of threads generated by OpenMP. Since all the threads in the team belong to the same task, we must set all their deadlines (periods) to be the same. In addition, we must make sure that all the threads of all the tasks are properly executed by the GEDF plug-in in LITMUS. We now describe how to set the parameters of both OpenMP and LITMUS to properly enforce this policy.

We first describe the specific parameter settings we have to use to ensure correct execution: (1) We specify the `static 1` policy within OpenMP to ensure that each thread gets approximately the same amount of work. (2) We also set the OpenMP thread synchronization policy to be passive. As discussed in Chapter 9.1.1, PGEDF cannot allow spinning waiting of threads. By using blocking synchronization, once a worker thread finishes its job, it will go to sleep immediately and yield the processor to threads from other tasks. Then the GEDF scheduler will assign the available core to the thread in the front of the prioritized ready queue. Thus, the idle time of one task can be utilized by other tasks, which is consistent with GEDF scheduling theory. (3) For each task, we set the number of threads to be equal to the number of available cores, m, using the GOMP function call *omp_set_num_threads(m)*. This means that if there are n tasks in the system, we will have a total of mn threads in the system. (4) In LITMUS^{RT}, the `budget_policy` is set equal to `NO_ENFORCEMENT` and the execution time of a thread is set to be the same as the relative deadline, as we do not need bandwidth control.

In addition to this parameter settings, PGEDF also does additional work to ensure that all the task parameters are set correctly. In particular, the actual code that is executed by PGEDF for each task is shown in Figure 9.3. In this code, before we run the task's actual run function for the first time, PGEDF performs some additional initialization in the form of a parallel for-loop. In addition, after each periodic execution of the task's run function, PGEDF executes an additional for-loop.

```
#pragma omp parallel for schedule(static, 1)
for (unsigned i = 0; i < num_cores; i++)
        rt_thread(period, deadline);

for (unsigned j = 0; j < num_periods; j++)
{
        task.run(task_argc, task_argv);

        #pragma omp parallel for schedule(static, 1)
        for (unsigned i = 0; i < num_cores; i++)
                sleep_next_period();
}
```

Figure 9.3: Main Structure of Each Real-Time Task in PGEDF

Let us first look at the initial for-loop. This parallel for-loop is meant to set the proper real-time parameters for this task to be correctly scheduled by GEDF plug-in in LITMUS^{RT}. We must set the real-time parameters for the entire team of OpenMP threads of this task. However, OpenMP threads are designed to be invisible to programmers, so we have no direct access to them. We get around this problem by using this initial for-loop, which has exactly m iterations — recall that each task has exactly m threads in its thread pool. Note that before this parallel for-loop, we set the scheduling policy to be ***static 1*** policy, which is a round robin static mapping between iterations and threads. Therefore, due to the `static 1` policy, each iteration is mapped to exactly 1 thread in the thread pool. Therefore, even though we cannot directly access OpenMP threads, we can still set real-time parameters for them inside the initial parallel for-loop by calling `rt_thread(period, deadline)` within this loop. This function is defined within the PGEDF platform to perform configuration for LITMUS^{RT}. In particular, the configuration steps described in the itemized list in the previous section are performed by this function. Since the thread team is reused for all parallel regions of the same program, we only need to set the real-time parameters for it once during task initialization; we need not set it at each job invocation.

After initialization, each task is periodically executed by `task.run(task_argc, task_argv)`, inside which there could be multiple parallel for-loops executed by the same team of threads. Periodic execution is achieved by the parallel for-loop after the `task.run` function; after each job invocation, this loop ensures that `sleep_next_period()` is called by each thread in the thread pool. Note again that since the number of iterations in this parallel for-loop is m, each thread will get exactly one iteration ensuring that each thread calls this function. This last for-loop is similar to the initialization for-loop, but tells the system that all the threads in the team of this task have finished their work and that the system should only wake them up when next period begins.

We can now briefly argue that these settings guarantee the correct GEDF execution. After we appropriately set the real-time parameters, all the relative deadlines will be automatically converted to absolute deadlines when scheduled by the LITMUS^{RT}. Since each thread in the same team of a particular task has the same deadline, all threads of this task have the same priority. Also, threads of a task with an earlier deadline have higher priority than the threads of the task with later deadlines — this is guaranteed by LITMUS^{RT} GEDF plug-in. Since the number of threads allocated to each program is equal to the number of cores, as required by GEDF, each job can utilize the entire machine when it is the highest priority task and has enough parallelism. If it does not have enough parallelism, then some of its threads sleep and yield the machine to the job with the next highest priority. Therefore, the GEDF scheduler within the LITMUS^{RT} enforces the correct priorities using the ready queue.

Chapter 10

Experimental Evaluation of PGEDF

We now describe our empirical evaluation of PGEDF using randomly generated tasks in OpenMP. Our experiments indicate that the parallel GEDF scheduling algorithm provides good real-time performance and that PGEDF outperforms the only other openly available parallel real-time platform, RT-OpenMP [30], in most cases.

10.1 Experimental Machines

Our experimental hardware is a 16-core machine with two Intel Xeon E5-2687W processors. We use the LITMUS^{RT} patched Linux kernel 3.10.5 and the GOMP runtime system from GCC version 4.6.3. The first core of the machine is always reserved in LITMUS^{RT} for releasing jobs periodically when running experiments. In order to test both single-socket and multi-socket performance, we ran two configurations — one with 7 experimental cores (with 1 reserved for releasing jobs and the other 8 disabled) and one with 14 experimental cores (with 1 reserved for releasing jobs and 1 disabled). For experiments with m available cores for task sets ($m = 7$ or 14 in our experiments) and one reserved core for releasing tasks, we set the number of cores for the system through the Linux kernel boot time parameter `maxcpus=`$m + 1$. After rebooting the system, only $m + 1$ total cores are available and the rest of the cores are disabled entirely.

10.2 Task Set Generation

We ran our experiments on synchronous tasks written in OpenMP as shown in Figure 9.1. Each task consists of a sequence of segments, where each segment is a parallel for-loop. The segments are of varying lengths and numbers of iterations. We ran 6 categories of task sets (shown in Table 10.1), with *T7:LP:LS:Har* using 7 cores and the rest using 14 cores. Here, we describe how we randomly generate task sets for our empirical evaluation. For each task, we first randomly selected its period (and deadline) D in a range between 4ms to 128ms. For task sets with ***harmonic*** deadlines, periods were always chosen to be one of {4ms, 8ms, 16ms, 32ms, 64ms, 128ms}, while for ***arbitrary*** deadlines, periods can be any value between 4ms and 128ms.

The task sets vary along two other dimensions: (1) Tasks may have ***low-parallelism*** or ***high-parallelism***. We control the parallelism by controlling the average number of iterations in each parallel for-loop. For low-parallelism task sets, the number of iterations in each parallel for-loop is chosen from a log-normal distribution with mean 8. For high-parallelism task sets, the number of iterations is chosen from a log-normal distribution with mean 12. In Table 10.1, the high parallelism task sets have HP in their label while low-parallelism tasks have LP in their label. Note that high-parallelism task sets have fewer tasks per task set on average since each individual task typically has higher utilization. (2) Tasks may have ***low-slack*** (LS) or ***high-slack*** (HS). We control the slack of a task by controlling the ratio between its critical path length and deadline. For low-slack task, their critical path length can be as large as their period. For high-slack tasks, their critical path length is at most half their deadline. In general, low-slack tasks are more difficult to schedule.

Name	Total #Cores	Deadline	L/D	Avg. #iterations	Avg. #Tasks per TaskSet
T14:LP:LS:Har	14	Harmonic	100%	8	5.03
T14:HP:LS:Har	14	Harmonic	100%	12	3.38
T14:LP:HS:Har	14	Harmonic	50%	8	8.58
T14:HP:HS:Har	14	Harmonic	50%	12	5.22
T7:LP:LS:Har	7	Harmonic	100%	8	3.66
T7:HP:HS:Har	7	Harmonic	50%	12	3.47
T14:HP:LS:Arb	14	Arbitrary	100%	12	3.33

Table 10.1: Task Set Characteristics

For all types of jobs, the execution time of each iteration was chosen from a log-normal distribution with a mean of 700 micro-seconds. Segments were added to the task until adding another segment would make its critical-path length longer than the desired maximum ratio (1/2 for high-slack tasks and 1 for low-slack tasks). Each task set starts empty and tasks were successively added until the total utilization ratio was between 98% and 100% of m — the number of cores in the machine. For example, for 14-core experiments, total utilization was between 13.72 and 14.

As with the numerical simulation experiments described in Chapter 8, we wished to understand the effects of speedup. We achieved the desired speedup by scaling down the execution time of each iteration of each segment of each task in each task set. For each experiment, we first generated 100 task sets with total utilization between $0.98m$ and m, and then scaled down the execution time by the desired speedup $1/b$. For example, for a speedup of 2, a iteration with execution time of 700 micro-seconds will be scaled down to 350 micro-seconds, and the total utilization of the task set will be about 7 for a 14-core experiment. In this manner, without scaling the actual core speed, we can achieve the desired speedup compared to the original task set. We evaluate the following speedup values {5, 3.3, 2.5, 2, 1.8, 1.6, 1.4, 1.2}, which correspond to total utilizations {20%, 30%, 40%, 50%, 56%, 62.5%, 71.4%, 83.3%} of m.

10.3 Baseline Platform

We compared the performance of PGEDF with the only other open source platform, RT-OpenMP from [30] — labeled ***RT-OpenMP*** — that can schedule parallel synchronous task sets on multicore system. RT-OpenMP is based on a task decomposition scheduling strategy similar to the DECOMP algorithm in Chapter 8: parallel tasks are decomposed into sequential subtasks with intermediate release times and deadlines. These sequential tasks are scheduled using a partitioned deadline monotonic scheduling strategy [31]. This decomposition based scheduler was shown to guarantee a capacity augmentation of 5 [64]. In theory, any valid bin-packing strategy provides this augmentation bound. The original paper [30] compared a worst-fit and best-fit bin-packing strategy for partitioning and found that worst-fit always performed better. Therefore, we only compare PGEDF (solid line in figures) vs. RT-OpenMP (dashed line in figures) with worst-fit bin-packing.

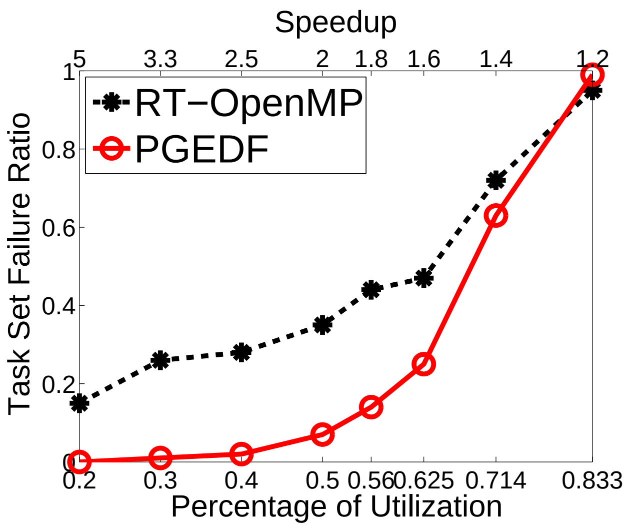
10.4 Experiment Results

For all experiments, each task set was run for 1000 hyper-periods for harmonic deadlines and 1000 times the largest period for arbitrary deadlines. In our experiments, we say that a task set ***failed*** if any task missed any deadline over the entire run of the experiment. In all figures, we plot the ***failure rate*** — the ratio of the failed task sets to the total number of task sets. The x-axis is the task set's utilization as a percentage of m. For example, 50% utilization in a 14-core experiment has a total utilization of 7. This setting is also equivalent to running the experiment on a machine of speed-2 — this speedup factor is shown on the top of the figures as the x-axis.

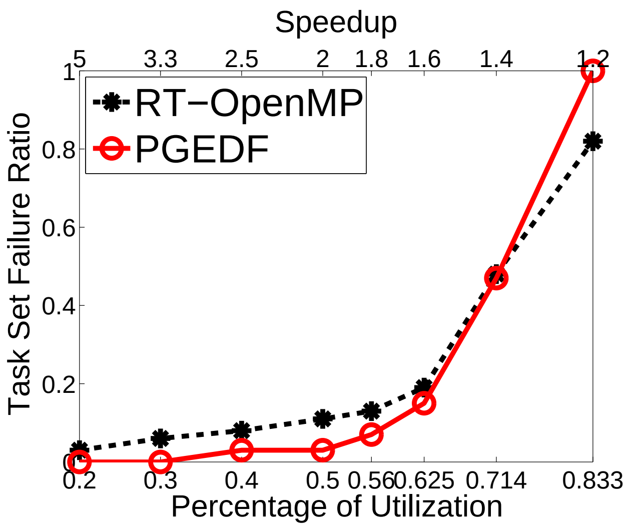
Figure 10.1(a) shows the failure ratio for task sets with low-parallelism, low-slack and harmonic periods on 14 cores. PGEDF outperforms RT-OpenMP for almost all utilizations. For instance at speed 3.3, PGEDF cannot schedule 1 task set, while RT-OpenMP fails on 26. At speed 5, GEDF can schedule all task sets, but 15 task sets miss deadlines under RT-OpenMP.

First, we look at the effect of slack. Recall that low-slack task sets can have tasks with long critical-path lengths (as long as the deadline) while high-slack jobs have smaller critical-path lengths (at most half the deadline). Let us first compare Figures 10.2(b) and 10.1(b) which show the failure ratios of high and low-slack task sets at the high-parallelism setting. For both systems, the high-slack tasks are easier to schedule, as expected. We see similar results in Figures 10.2(a) and 10.1(a) when comparing high and low-slack tasks with low-parallelism. However, for both settings, RT-OpenMP appears to be more sensitive to slack than PGEDF. This is due to the fact that RT-OpenMP performs a careful decomposition of tasks based on the available slack — therefore, in a low-slack setting, it is harder for it to find a good decomposition.

We now look at the influence of the degree of parallelism in task sets. First, we look at the task sets with high-slack. Figures 10.2(a) and 10.2(b) show the results for high and

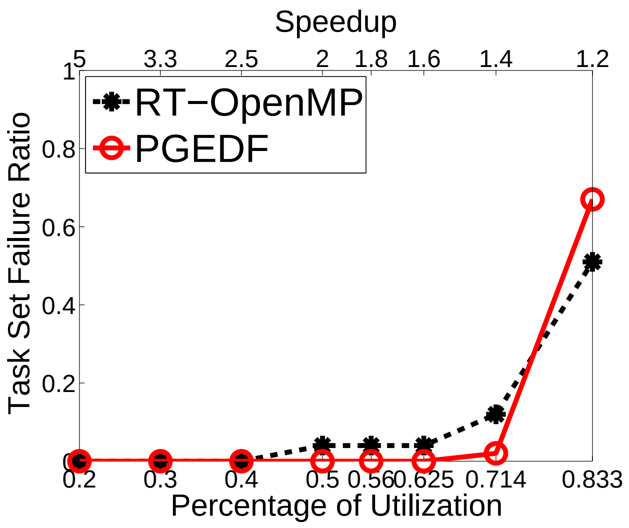


(a) T14:LP:LS:Har (with low-parallelism).

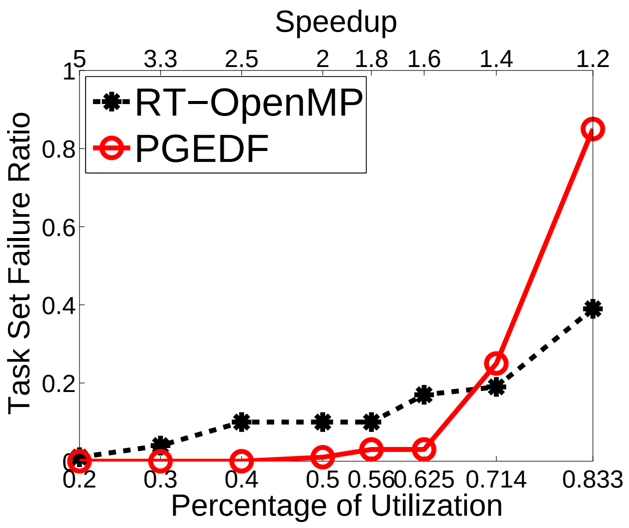


(b) T14:HP:LS:Har (with high-parallelism).

Figure 10.1: Failure ratio of PGEDF vs. RT-OpenMP with different percentages of utilization (speedup) for 14-core task sets with low-slack and harmonic periods.



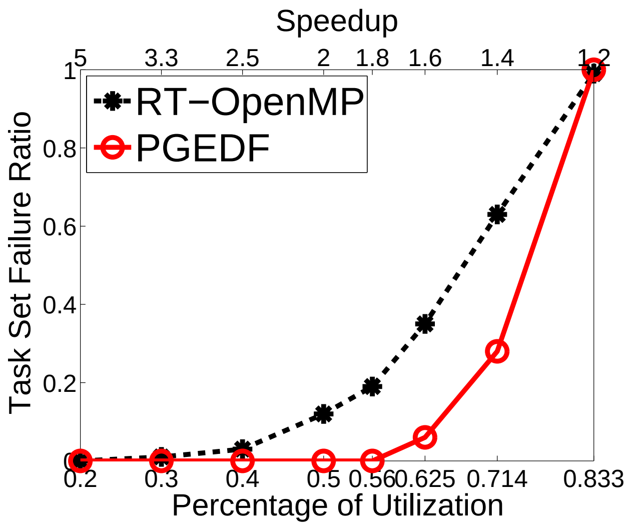
(a) T14:LP:HS:Har (with low-parallelism).



(b) T14:HP:HS:Har (with high-parallelism).

Figure 10.2: Failure ratio of PGEDF vs. RT-OpenMP with different percentages of utilization (speedup) for 14-core task sets with high-slack and harmonic periods.

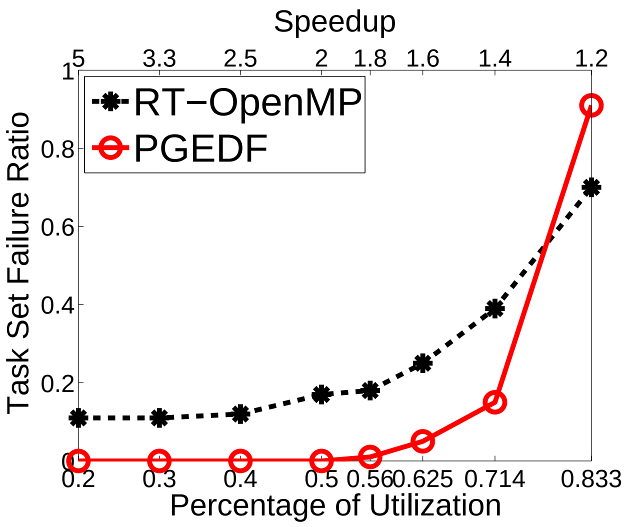
low-parallelism task sets for high-slack setting. Note that higher-parallelism task sets have a higher failure ratio than the low-parallelism task sets for both platforms, but the difference is not significant. Now we take a look at the low-slack case — Figures 10.1(a) and 10.1(b) show the results for high and low-parallelism task sets. Now the results are reversed — both platforms perform better on high-parallelism task sets than on low-parallelism task sets. We believe that these results are due to the fact that low-parallelism task sets have a larger number of total tasks per task set (shown in Table 10.1) — which leads to higher overhead due to a larger number of total threads. For low-slack tasks, the slack between deadline and critical-path length is relatively small, so they are more sensitive to overhead — therefore, when there are a large number of threads (low-parallelism task sets), they perform worse. Also note that this effect is much more pronounced on RT-OpenMP than on PGEDF, indicating that PGEDF may be less effected by overheads and more scalable.



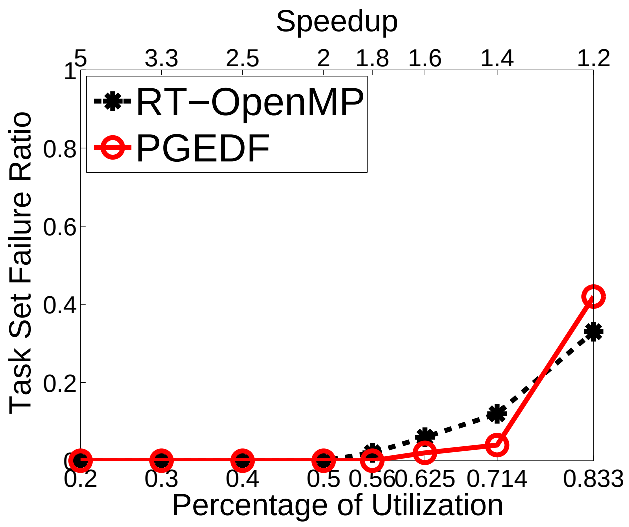
(a) T14:HP:LS:Arb (with arbitrary periods).

Figure 10.3: Failure ratio of PGEDF vs. RT-OpenMP with different percentages of utilization (speedup) for 14-core task sets with low-slack and high-parallelism.

Comparing Figures 10.1(b) and 10.3(a), we can see the effect of harmonic and arbitrary deadlines. For PGEDF, task sets with arbitrary deadlines are easier to schedule than harmonic deadlines — this is not surprising since many jobs have the same priority (absolute deadlines) when periods are harmonic. GEDF cannot distinguish between jobs having the same deadline but different remaining critical-path length. So, it may decide to delay threads from the job with the largest remaining critical-path length and execute others first. In such cases, after finishing other work, even though all cores are available for that job, the remaining time may not be enough to finish the sequential execution of the remaining critical-path (when speedup is less than the capacity bound of 4), and the job will miss its deadline. On the other hand, RT-OpenMP does not show clear advantage for arbitrary deadlines. Again, this is not surprising, since RT-OpenMP decomposes tasks into sequential subtasks and schedules them using fixed priorities based on their sub-deadlines. Even if the end-to-end tasks are harmonic, these decomposed subtasks are unlikely to be harmonic.



(a) T7:LP:LS:Har (with low-parallelism, low-slack).



(b) T7:HP:HS:Har (with high-parallelism, high-slack).

Figure 10.4: Failure ratio of PGEDF vs. RT-OpenMP with different percentages of utilization (speedup) for 7-core task sets.

Finally, note that there is a significant difference between the simulation results in Chapter 8 and these experiments. In simulation, GEDF required a speedup of at most 2, while here it often requires speedup of 2.5 or more. This is not surprising, since real platforms have overheads that are completely ignored in simulations. In particular, for 14 core experiments on our machines, there is high inter-socket communication overhead of the operating system, which is ignored by theory and is not considered in simulation.

Therefore, we also conduct experiments on 7 cores in the same socket (shown in Figure 10.4(a) and 10.4(b)) also with harmonic periods. We can see that at a speedup of 2, all task sets are schedulable under PGEDF, indicating that inter-socket communication does play a significant role in these experimental results. Both experiments have roughly the same number of tasks per task set. We can see the trend that RT-OpenMP performs a lot worse with low-slack still holds in Figure 10.4. With high-slack, RT-OpenMP has similar failure ratio to PGEDF, while with low-slack, it is much worse than PGEDF.

In conclusion, PGEDF performs better in all experiments and generally requires lower speedup to schedule task sets than RT-OpenMP. In addition, the capacity augmentation bound of 4 for the GEDF scheduler holds for all experiments conducted here.

Chapter 11

Conclusions

In this paper, we have presented the best bounds known for GEDF scheduling of parallel tasks represented as DAGs. In particular, we proved that GEDF provides a resource augmentation bound of $2-1/m$ for sporadic task sets with arbitrary deadlines and a capacity augmentation bound of $4-2/m$ with implicit deadlines. The capacity augmentation bound also serves as a simple schedulability test, namely, a task set is schedulable on m cores if (1) m is at least $4-2/m$ times its total utilization, and (2) the implicit deadline of each task is at least $4-2/m$ times its critical-path length. We also presented another fixed point schedulability test for GEDF.

We simulated randomly generated DAG tasks with a variety of settings. In these simulations, we never saw a required capacity augmentation of more than 2 on randomly generated task sets. For computationally intensive jobs, our experiments indicate that this platform outperforms a previous platform that relies on task decomposition.

There are three possible directions of future work. First, we would like to extend the capacity augmentation bounds to constrained and arbitrary deadline. In addition, while we prove that a capacity augmentation bound of more than $\frac{3+\sqrt{5}}{2}$ is needed, there is still a gap between this lower bound and the upper bound of $(4-2/m)$ for capacity augmentation, which we would like to close. Finally, we would like to conduct experiments on real platform to quantify the performance of GEDF and to measure its overheads, and improve the schedulability test including overheads based on these experiments.

References

[1] Kunal Agrawal, Charles E. Leiserson, Yuxiong He, and Wen Jing Hsu. Adaptive work-stealing with parallelism feedback. *ACM Trans. Comput. Syst.*, 26, September 2008.

[2] James H. Anderson and John M. Calandrino. Parallel real-time task scheduling on multicore platforms. In *RTSS '06*, 2006.

[3] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. In *RTSS '01*, pages 193–202, dec. 2001.

[4] B. Andersson and J. Jonsson. The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%. In *ECRTS '03*, pages 33–40, 2003.

[5] Björn Andersson and Dionisio de Niz. Analyzing global-edf for multiprocessor scheduling of parallel tasks. In *Principles of Distributed Systems*, pages 16–30. 2012.

[6] P. Axer, S. Quinton, M. Neukirchner, R. Ernst, B. Dobel, and H. Hartig. Response-time analysis of parallel fork-join workloads with real-time constraints. In *ECRTS '13*, 2013.

[7] T.P. Baker and S.K. Baruah. Sustainable multiprocessor scheduling of sporadic task systems. In *ECRTS '09*, pages 141 –150, july 2009.

[8] Sanjoy Baruah and Theodore Baker. Schedulability analysis of global edf. *Real-Time Systems*, 38(3):223–235, 2008.

[9] Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Sebastian Stiller. Improved multiprocessor global schedulability analysis. *Real-Time Syst.*, 46(1):3–24, September 2010.

[10] Sanjoy Baruah, Vincenzo Bonifaciy, Alberto Marchetti-Spaccamelaz, Leen Stougiex, and Andreas Wiese. A generalized parallel task model for recurrent real-time processes. In *RTSS '12*, 2012.

[11] S.K. Baruah. Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. *Computers, IEEE Transactions on*, 53(6):781–784, june 2004.

[12] M. Bertogna, M. Cirinei, and G. Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *Parallel and Distributed Systems*, 20(4):553 –566, april 2009.

[13] Marko Bertogna and Sanjoy Baruah. Tests for global edf schedulability analysis. *J. Syst. Archit.*, 57(5):487–497, 2011.

[14] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese. Feasibility analysis in the sporadic dag task model. In *ECRTS '13*, 2013.

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

[16] Björn B. Brandenburg and James H. Anderson. On the implementation of global real-time schedulers. In *RTSS '09*, 2009.

[17] John M. Calandrino and James H. Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. In *ECRTS '09*, 2009.

[18] Felipe Cerqueira and Björn B Brandenburg. A comparison of scheduling latency in linux, preempt-rt, and litmusrt. *OSPERT 2013*, page 20, 2013.

[19] Hoon Sung Chwa, Jinkyu Lee, Kieu-My Phan, Arvind Easwaran, and Insik Shin. Global edf schedulability analysis for synchronous parallel tasks on multicore platforms. In *ECRTS '13*, 2013.

[20] Sébastien Collette, Liliana Cucu, and Joël Goossens. Integrating job parallelism in real-time scheduling theory. *Inf. Process. Lett.*, 106(5):180–187, 2008.

[21] Daniel Cordeiro, Grgory Mouni, Swann Perarnau, Denis Trystram, Jean-Marc Vincent, and Frdric Wagner. Random graph generation for scheduling simulations. In *SIMUTools '10*, 2010.

[22] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comp. Surv.*, 43:35:1–44, 2011.

[23] Xiaotie Deng, Nian Gu, Tim Brecht, and KaiCheng Lu. Preemptive scheduling of parallel jobs on multiprocessors. In *SODA '96*, 1996.

[24] Umamaheswari C Devi. *Soft real-time scheduling on multiprocessors*. PhD thesis, University of North Carolina, 2006.

[25] UmaMaheswari C Devi and James H Anderson. Tardiness bounds under global edf scheduling on a multiprocessor. *Real-Time Systems*, 38(2):133–189, 2008.

[26] Maciej Drozdowski. Real-time scheduling of linear speedup parallel tasks. *Inf. Process. Lett.*, 57(1):35–40, 1996.

[27] J. Erickson, U. Devi, and S. Baruah. Improved tardiness bounds for global edf. In *ECRTS '2010*.

[28] Jeremy P Erickson and James H Anderson. Fair lateness scheduling: Reducing maximum lateness in g-edf-like scheduling.

[29] Frédéric Fauberteau, Serge Midonnet, and Manar Qamhieh. Partitioned scheduling of parallel real-time tasks on multiprocessor systems. *SIGBED Rev.*, 8(3):28–31, sep 2011.

[30] David Ferry, Jing Li, Mahesh Mahadevan, Kunal Agrawal, Christopher Gill, and Chenyang Lu. A real-time scheduling service for parallel tasks. In *RTSS '13*, 2013.

[31] Nathan Fisher, Sanjoy Baruah, and Theodore P. Baker. The partitioned scheduling of sporadic tasks according to static-priorities. In *ECRTS '06*, 2006.

[32] Nathan Fisher, Joël Goossens, and Sanjoy Baruah. Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible. *Real-Time Syst.*, 45(1-2):26–71, 2010.

[33] Gamma. Gamma distribution, Distribution. `http://en.wikipedia.org/wiki/Gamma_distribution`.

[34] Jol Goossens, Shelby Funk, and Sanjoy Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2):187–205, 2003.

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

[36] Huang-Ming Huang, Terry Tidwell, Christopher Gill, Chenyang Lu, Xiuyu Gao, and Shirley Dyke. Cyber-physical systems for real-time hybrid structural testing: a case study. In *ICCPS '10*, 2010.

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

[38] S. Kato and Y. Ishikawa. Gang EDF scheduling of parallel task systems. In *RTSS '09*, 2009.

[39] J. Kim, H. Kim, K. Lakashmanan, and R. Rajkumar. Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *ICCPS '13*, 2013.

[40] JFC Kingman. Inequalities in the theory of queues. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 102–110, 1970.

[41] M. Korsgaard and S. Hendseth. Schedulability analysis of malleable tasks with arbitrary parallel structure. In *RTCSA '11*, 2011.

[42] Oh-Heum Kwon and Kyung-Yong Chwa. Scheduling parallel tasks with individual deadlines. *Theor. Comput. Sci.*, 215(1-2):209–223, 1999.

[43] Karthik Lakshmanan, Shinpei Kato, and Ragunathan (Raj) Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *RTSS '10*, 2010.

[44] Jinkyu Lee and Kang G. Shin. Controlling preemption for better schedulability in multi-core systems. In *RTSS '12*, Dec. 2012.

[45] Wan Yeon Lee and Heejo Lee. Optimal scheduling for real-time parallel tasks. *IEICE Trans. Inf. Syst.*, E89-D(6):1962–1966, 2006.

[46] Juri Lelli, Dario Faggioli, Tommaso Cucinotta, and Giuseppe Lipari. An experimental comparison of different real-time schedulers on multicore systems. *J. Syst. Softw.*, 85(10):2405–2416, October 2012.

[47] Juri Lelli, Dario Faggioli, Tommaso Cucinotta, and Scuola Superiore. An efficient and scalable implementation of global edf in linux. In *OSPERT '11*, 2011.

[48] Hennadiy Leontyev and James H Anderson. Generalized tardiness bounds for global multiprocessor scheduling. *Real-Time Systems*, 44(1-3):26–71, 2010.

[49] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[50] Cong Liu and James H. Anderson. An o(m) analysis technique for supporting real-time self-suspending task systems.

[51] Cong Liu and J.H. Anderson. Supporting soft real-time parallel applications on multicore processors. In *RTCSA '12*, 2012.

[52] J. M. López, J. L. Díaz, and D. F. García. Utilization bounds for edf scheduling on real-time multiprocessor systems. *Real-Time Syst.*, 28(1):39–68, October 2004.

[53] José María López, José Luis Díaz, Joaquín Entrialgo, and Daniel García. Stochastic analysis of real-time systems under preemptive priority-driven scheduling. *Real-Time Systems*, 40(2):180–207, 2008.

[54] Amin Maghareh, Shirley J. Dyke, Arun Prakash, Gregory Bunting, and Payton Lindsay. Evaluating modeling choices in the implementation of real-time hybrid simulation. In *EMI/PMC 2012*, 2012.

[55] G. Manimaran, C. Siva Ram Murthy, and Krithi Ramamritham. A new approach for scheduling of parallelizable tasks in real-time multiprocessor systems. *Real-Time Syst.*, 15(1):39–60, 1998.

[56] Alex F Mills and James H Anderson. A stochastic framework for multiprocessor soft real-time scheduling. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, pages 311–320. IEEE, 2010.

[57] Ingo Molnr. CONFIG_PREEMPT_RT Patch. https://rt.wiki.kernel.org/index.php/Main_Page.

[58] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. Techniques optimizing the number of processors to schedule multi-threaded tasks. In *ECRTS '12*, 2012.

[59] Lus Nogueira and Lus Miguel Pinho. Server-based scheduling of parallel real-time tasks. In *International Conference on Embedded Software*, 2012.

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

[61] Luigi Palopoli, Daniele Fontanelli, Nicola Manica, and Luca Abeni. An analytical bound for probabilistic deadlines. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 179–188. IEEE, 2012.

[62] Constantine D. Polychronopoulos and David J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, 1987.

[63] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. Gill. Parallel real-time scheduling of dags. *IEEE Transactions on Parallel and Distributed Systems*, PP(99):1–1, 2014.

[64] Abusayeed Saifullah, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core real-time scheduling for generalized parallel task models. In *RTSS '11*, 2011.

[65] Anand Srinivasan and James H Anderson. Efficient scheduling of soft real-time applications on multiprocessors. In *ECRTS*, volume 3, pages 51–54, 2003.

[66] Anand Srinivasan and Sanjoy Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information Processing Letters*, 84(2):93 – 98, 2002.

[67] Qingzhou Wang and Kam Hoi Cheng. A heuristic of scheduling parallel tasks and its analysis. *SIAM J. Comput.*, 21(2), 1992.