technische universität
dortmund

Masterthesis

Reservation-Based Federated Scheduling for Parallel Real-Time Tasks

Niklas Ueter

7.3.2018

Supervisors:
Prof. Dr. Jian-Jia Chen
Dipl. Inf. Georg von der Brüggen

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12 (Eingebettete Systeme)
http://ls12-www.cs.tu-dortmund.de

Contents

1 Introduction

"You're unlikely to discover something new without a lot of practice on old stuff, but further, you should get a heck of a lot of fun out of working out funny relations and interesting things" Richard P. Feynman.

1.1 Motivation

Modern real-time systems increasingly facilitate multi-core systems in order to account for the growing computational demands of complex real-time applications e.g., in the domain of autonomous driving or complex sensor-fusion algorithms [49]. Further, physical and technical limitations on clock-frequencies, energy-inefficiency and excessive heat dissipation motivate the transition to multiprocessor platforms in traditional computing systems as well as in cyber-physical systems.

Traditionally, in uniprocessor platforms the execution-demand of sequential real-time tasks is solely modeled by their worst-case execution time, since the processor executes only one job at each point in time. Thus, there is no need to express potential parallel execution paths.

However, multi-core platforms allow for *inter-task parallelism* i.e., to execute sequential programs concurrently and *intra-task parallelism*, i.e., a job of a parallelized task can be executed on multiple processors at the same time. To enable *intra-task parallelism*, applications must be potentially executed in parallel, which must be accounted for by the application design.

As demonstrated by Serrano et al. [55], the asynchronous parallel task model, represented as a directed acyclic graph (DAG), where each thread is represented by a node and the edges denote precedence constraints, can be realized using the untied tasking model of OpenMP[1]. Due to this fact and the increasing OpenMP support by newly developed multi-core processors [55], the DAG task model is a reasonable real-time task model to use for the design of scheduling algorithms.

A recent approach for the scheduling of parallel real-time DAG tasks is *federated scheduling*, where *heavy* tasks i.e., tasks that need to execute on more than one processor concurrently in order to meet the relative deadline are assigned a set of processors exclusively, whilst the *light* tasks are sequentialized and scheduled on the remaining processors [38].

[1] http://www.openmp.org/

A subsequent downside of the exclusive granting of processors to heavy tasks is the non-consideration of whether the exclusively assigned processors could be shared with light tasks and thus potentially wasting system resources. Furthermore, the disparity between a DAG task's density and its utilization poses a challenge.
Nonetheless, the exclusive assignment allows for simple response time analyses due to the absence of inter-task interference and contiguous service provided to the DAG task by the assigned processors. Additionally, the isolation of heavy tasks yields lowered scheduling overheads and cache advantages due to the non-preemption of the heavy tasks.

1.2 Contributions and Structure

To address the aforementioned limitations of federated scheduling whilst maintaining the favorable analytical properties of absent inter-task interference, a novel reservation-based federated scheduling approach for the scheduling of hard real-time arbitrary-deadline DAG task sets is proposed. More precisely, the contributions of this thesis are as follows:

- Proposal of a novel reservation-based federated scheduling approach where each DAG task is assigned a set of dedicated reservation servers that inherit the timing models of the associated DAG task. These reservation servers can then be scheduled as ordinary sequential tasks by any multiprocessor scheduling algorithm that supports such timing models.

- Constraints and design rules for the dimensioning and assignment of the reservation servers such that each DAG task is schedulable if all its reservation servers are feasibly schedulable under any multiprocessor scheduling algorithm.

- Propose algorithms to calculate provably sufficient and heuristically good reservation servers for the scheduling of arbitrary-deadline DAG task sets.

- Resolving the open problem of a non-constant speedup factor for arbitrary-deadline DAG task sets under federated scheduling with respect to any optimal DAG task set scheduling algorithm as pointed out by Chen [20]. More precisely, it can be shown that the speedup factor of reservation-based federated scheduling is at most $3+2\sqrt{2}$ by the design of a specific set of reservation servers that are scheduled under partitioned multiprocessor scheduling or global multiprocessor scheduling.

- Finally, this thesis contains various experiments using different methods for synthetically generated DAG task sets to evaluate and demonstrate the competitiveness of the proposed reservation-based federated scheduling approach with the state of the art.

The remainder of this thesis is structured as follows. In the following Chapter 2 *Real-Time Systems and Fundamentals*, fundamentals of real-time scheduling as well as concepts and techniques, that are required to follow this thesis are introduced.
In the subsequent Chapter 3 *DAG Task Scheduling*, the task and system model as well as related work and their issues are presented and discussed in order to motivate the proposed reservation-based federated scheduling approach.
In Chapter 4 *Reservation-Based Federated Scheduling*, the concepts and fundamental theoretical properties of reservation-based federated scheduling are described.
Followingly, in Chapter 5 *Algorithms for the creation of Reservations* concrete algorithms for the calculation of reservation servers and the analyses thereof are detailed.
Based on these theoretic results, a prototyped realization of reservation-based federated scheduling is presented in Chapter 6 *Simulation and Prototyping of Reservation-Based Federated Scheduling.*
In the following Chapter 7 *Evaluations*, the developed scheduling algorithms and the analyses thereof are compared against the state-of-the-art algorithms using synthetically generated task sets.
Lastly, Chapter 8 *Conclusion, Related and Future work* concludes the results of this thesis and argues the practicability based on an evaluation using a real-time operating system implementation of the introduced reservation-based federated scheduling approach. Further, open problems and future work are discussed.

2 Real-Time Systems and Fundamentals

In this chapter, a short summary and classification of scheduling paradigms and algorithms in the domain of real-time systems is given in order to put this thesis into context and introduce fundamental results and techniques that will be used in this thesis.
To that end, in the first Section 2.1 *Real-Time Systems*, a brief introduction of the most relevant real-time system characteristics is given.
In the following Section 2.2 *Real-Time Scheduling*, the concepts of hard real-time resource arbitration are introduced.
Moreover, in the following Section 2.2.1 *Sporadic Task Model*, the fundamental activation and task models that are relevant in this thesis are presented and motivated.

2.1 Real-Time Systems

In contrast to non-real-time systems that emphasize *functional correctness*, the most important property that must be guaranteed for in real-time systems is *temporal correctness*. In computing systems, a process i.e., a running software application that is executed by a processor sequentially and repeatedly activated according to some activation pattern is considered a task where each instance of a task is considered a job.
Therefore, *temporal correctness* requires that the response time of each task i.e., the time between the activation of a task until the finishing of that task must be no more than the specified relative deadline.
Furthermore, real-time systems can be categorized into *soft*, *firm* and *hard* real-time systems based on the usefulness or rather consequences that a deadline miss and subsequent lateness may cause.
A real-time system is considered *hard*, if a deadline miss may cause catastrophic consequences [19] e.g., a deadline miss in the electronic stability control (ESP) may cause instability of the automobiles trajectory and subsequently cause accidents.
In contrast to that, a real-time system is considered *firm*, if a deadline miss invalidates the computed result [19] in the sense that the computed results are useless, but are no cause of catastrophic consequences.
Lastly, a real-time system is considered *soft*, if the computed result is still useful (despite the lateness), but causes degradation of quality [19]. An example of a soft real-time system

is a video decoder, that based on the lateness may adapt the frame rate or drop the frame altogether, thus leading to a degraded video quality.

Besides *timeliness*, the most important properties a real-time system should posses are *robustness*, *predictability* and *efficiency* [19].

In order to design a predictable system, all relevant application specifications required for the analysis of the consequences of scheduling decisions are mandatory to be modeled. Namely, the worst-case execution time, activation patterns, resource demands or precedence constraints e.g., read / write communications must be properly modeled and accounted for. Furthermore, all system's resources should be managed efficiently by the real-time operating system.

Additionally, robustness with respect to workload variations and model uncertainties must be accounted for by the system design [19]. That is, for example, to account for the context switch or preemption overheads in the worst-case execution time parameters of a task model or by reserving processor utilization for interrupt handling exclusively. In order to achieve the above properties, the system, the hardware architecture and the arbitration algorithms must be designed and verified accordingly.

To put this thesis into context in the domain of real-time systems, this thesis focuses on the design and analysis of hard real-time multiprocessor scheduling algorithms of parallel task systems. In other words, the design and verification of temporal correctness for the system and task model under analysis will be detailed in the remainder of this thesis.

2.2 Real-Time Scheduling

In accordance to the definition by Gorgio Buttazzo, a *problem instance* of a scheduling problem is defined by a set of tasks $\mathbf{T}$, a set of processors $\mathbf{P}$ and a set of resources $\mathbf{R}$ [19]. Based on these definitions, a scheduling algorithm is responsible for the assigning of processors $\mathbf{P}$ and resources $\mathbf{R}$ to the tasks in $\mathbf{T}$ such that all tasks finish under their specific timing constraints e.g., before their relative deadline.

To further illustrate this definition, an exemplary schedule for two sporadic implicit-deadline tasks is shown in Fig. 2.1. In this example, the scheduling algorithm maps the resource, namely processor time, to the ready tasks such that both tasks do not exceed their relative deadlines. In the illustrated example, the scheduling algorithm prioritizes the allocation of processing time to the task with the highest activation frequency.

In general, scheduling algorithms can be categorized into *preemptive* and *non-preemptive*. A preemptive scheduling algorithm may dispatch an executing job whenever the scheduler is invoked, whereas a job always runs to completion in non-preemptive scheduling. Another important categorization is to be made between dynamic and static scheduling algorithms. That is, in dynamic scheduling algorithms, a scheduling decision is based on parameters

that may change during the runtime of the system e.g., prioritizing the tasks according to their remaining time to the deadline (slack).
Conversely, in static scheduling algorithms, the scheduling decisions are based on parameters that are assigned to the tasks before their activation e.g., fixed-priorities. Moreover, scheduling algorithms can be classified according to the supported platform i.e., uniprocessor or multiprocessor systems as well as the supported task models as will be explained in more detail in the following section.

2.2.1 Sporadic Task Model

A scheduling algorithm that is used in hard real-time systems must guarantee that no task in the system exceeds its relative deadline. In order to quantify the workload, that is to be processed by the system, the worst-case arrival patterns must be parameterizable.
A commonly used arrival pattern is the sporadic activation model that is an abstraction of workloads in real-time systems arriving to the system within a variable but bounded rate.
More precisely, if the first job of a sporadic task arrives at time t_0 then the next job may be released as early as $t_0 + T_i$ where T_i denotes the minimal inter-arrival time. Another parameter is the relative deadline that specifies the maximum response time a job is allowed to have.
The relative deadlines themselves are to be specified by the system designer and chosen such that the underlying application constraints are satisfied. Moreover, the worst-case execution C_i of the task must be known in order to model the arriving workload to the system.
In conclusion a sporadic task is specified by the tuple (C_i, D_i, T_i) where given a job release at t_0, C_i amount of work has to be executed over the interval $[t_0, t_0 + D_i)$ and the next job may release as early as $t_0 + T_i$. Based on the relationship of the minimal inter-arrival time T_i and the relative deadline D_i, task sets can be characterized. If for all sporadic

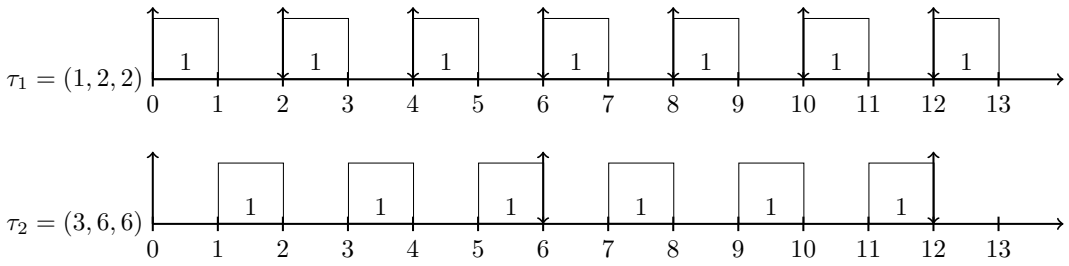


Figure 2.1: An exemplary schedule of two independent sporadic implicit-deadline tasks, that are scheduled according to some fixed-priority scheduling algorithm. The scheduling algorithm maps tasks to the resource (CPU time), which is indicated by the indicator function such that the function support denotes the execution time.

tasks in the task set $D_i \leq T_i$ holds then this task set is considered constrained-deadline. Further, if for all tasks $D_i = T_i$ holds then the task set is called implicit-deadline and arbitrary-deadline otherwise .

2.2.2 Uniprocessor Scheduling

Due to the relevance of uniprocessor scheduling algorithms in this thesis, this section details the underlying scheduling algorithms that are used in this thesis. Additionally, important techniques and definitions, that are necessary to follow the presentation of this thesis are presented.

In uniprocessor scheduling algorithms, at each point in time the scheduling algorithm selects an eligible job from the ready-queue for execution. As previously mentioned, the scheduling algorithms can be categorized into static and dynamic priority algorithms. To that end, fixed-priority scheduling algorithms are static scheduling algorithms since the tasks are prioritized according to fixed parameters that do not change during the runtime. In the category of fixed-priority scheduling algorithms, it could be proved that *deadline-monotonic* (DM) is an optimal preemptive fixed-priority scheduling algorithm in the sense that if a task set can be scheduled using any fixed-priority scheduling algorithm then this task set can be scheduled by deadline-monotonic scheduling as well. At each point in time, the deadline-monotonic scheduling algorithm prioritizes the tasks, amongst all ready tasks, with the smallest relative deadline.

An important dynamic priority uniprocessor scheduling algorithm is earliest-deadline first (EDF) that at each point in time prioritizes the task with the earliest absolute deadline. It could be shown, that the dynamic priority *EDF* algorithm is an optimal preemptive uniprocessor scheduling algorithm [29]. That is, if a task set is schedulable on a uniprocessor platform, then *EDF* is guaranteed to also be able to feasibly schedule that task set.

In the context of hard real-time scheduling, a task set is considered schedulable by a scheduling algorithm if under any legal release pattern of the tasks e.g., periodic job releases, no task misses its relative deadline in the generated schedule (by the scheduling

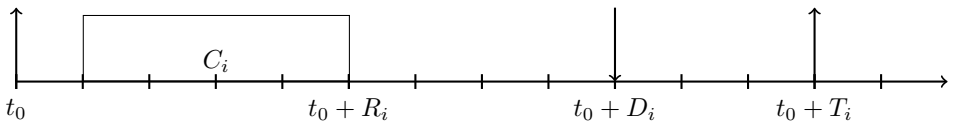


Figure 2.2: A sporadic arbitrary-deadline task is modeled by the tuple $\tau_i = (C_i, D_i, T_i)$. That is, C_i amount of work must be finished during the release and deadline interval $(t_0, t_0 + D_i]$, where new jobs are released as soon as possible i.e., at $t_0 + \ell T_i$ for $\ell \in \mathbb{N}$. Further, the response time R_i denotes the time difference between the arrival t_0 and the finishing time of task τ_i.

algorithm). In order to avoid to enumerate all legal release patterns and thus generated schedules, it is favorable to identify a *worst-case* release pattern to verify schedulability. In fixed-priority uniprocessor scheduling algorithms, this intuition is more precisely described in the following critical instant theorem. A critical instant is defined *[...] to be an instant for a task, at which a request for that task will have the largest response time* [48]. It can be shown that for sporadic constrained-deadline task sets, the critical instant occurs for the first released job, whenever the task under analysis is released synchronously together with all higher-priority tasks [48].
Therefore, it is sufficient to verify that the first job of each task τ_i finishes at any time during the activation and deadline interval $0 < t \leq D_i$ under the assumption that all higher-priority tasks are already verified to be schedulable. In the context of fixed-priority scheduling of sporadic arbitrary-deadline task sets, the request-bound function can be used to formalize each task's generated workload.

2.2.1 Definition (request-bound function [44]). The request-bound function of a sporadic arbitrary-deadline task $\text{RBF}_i(t) \stackrel{\text{def}}{=} \left\lceil \frac{t}{T_i} \right\rceil \cdot C_i$ denotes the largest cumulative amount of work a task τ_i may request over any contiguous interval of length t.

Based on the definition of the request-bound function, the schedulability of each task τ_k that releases synchronously with all higher-priority tasks, denoted by $hp(\tau_k)$, can be verified. That is, relative to some common arbitrary time-offset e.g., $t_0 = 0$, the schedulability can be verified by testing whether $\exists 0 < t \leq D_k$ such that

$$C_k + \sum_{\tau_i \in hp(\tau_k)} \text{RBF}_i(t) \leq t. \tag{2.1}$$

Using the time-demand analysis as illustrated in Eq. (2.1), the response time of task τ_k is given by the earliest point in time $t \in (0, D_k]$ such that equality holds. To calculate the response time, the following fix-point iteration

$$t_n = C_k + \sum_{\tau_i \in hp(\tau_k)} \text{RBF}_i(t_{n-1}) \text{ and } t_0 = C_k \tag{2.2}$$

can be used until either $t_n > D_k$ or $t_n = t_{n-1}$ where the index n denotes the $n - th$ iteration.
If a task has multiple ready jobs pending at a certain point in time then the jobs are dispatched according to their activation e.g., first-come-first-serve. In other words, a pending job can only start executing once all previously activated jobs are finished.
In the case of arbitrary-deadline task sets, the aforementioned critical instant theorem does not hold anymore since it is not sufficient to only verify the first released job since that job does not necessarily exhibit the longest response time.

2.2.2 Definition (busy task [37]). A task τ_k is considered *busy* at time t if at least one job of that task, that arrived before t, has not yet finished execution.

It is to emphasize that a task being *busy* does not imply the execution of that task, but that the task has at least one pending job. Intuitively, if each task's pending jobs are queued according to *first-in-first-out* in a per-task queue then a task stops being *busy* when that queue runs idle.
In the previously discussed constrained-deadline case, the time a task stops being *busy* equals to the task's response time.
Since the execution of pending jobs of a task τ_k depends on the execution demands of higher-priority tasks, further definitions are required to generalize the response time analysis shown in Eq. (2.1) for arbitrary-deadline task sets.

2.2.3 Definition (level-k busy interval [37]). A *level-k busy interval* $[t_0, t_0 + t)$ is an interval such that task τ_k is *busy* during $[t_0, t_0 + t)$ and task τ_k is *not busy* right before t_0.

From the definition it follows that any *level-k busy interval* starts with the release of a job of task τ_k and finishes when all jobs of task τ_k (that have been released before t) have finished.
Another implication is that the response time of any job of task τ_k must be part of *some level-k busy interval*. Moreover, a *level-k busy interval* depends on the execution pattern in a concrete schedule.
It can be proved that the longest *level-k busy interval* occurs when the first job of task τ_k releases synchronously with the first jobs of all higher-priority tasks [43].
Consequently, in order to verify the schedulability of task τ_k, the response times of all jobs, released in the longest *level-k busy interval*, must be no more than the task's relative deadline.
In order to calculate the length of the longest *level-k busy interval*, the smallest time interval L_k such that

$$C_k + \sum_{\tau_i \in hp(\tau_k)} \text{RBF}_i(L_k) = L_k \tag{2.3}$$

holds must be computed. Again, this can be done by using fix-point iteration.
If the longest *level-k busy interval* is denoted by L_k then $h \stackrel{\text{def}}{=} \left\lfloor \frac{L_k}{T_k} \right\rfloor$ many jobs are released in this interval by task τ_k and thus need to be verified. Given a synchronous job release at $t = 0$, the response time of the $\ell - th$ job of task τ_k is given by

$$R_{k,\ell} \stackrel{\text{def}}{=} f_{k,\ell} - (\ell - 1) \cdot T_k \; \forall 1 < \ell \leq h \tag{2.4}$$

where $f_{k,\ell}$ denotes the worst-case finishing time of the $\ell - th$ job, which can be calculated as follows

$$(\ell - 1) \cdot C_k + C_k + \sum_{\tau_i \in hp(\tau_k)} \left\lceil \frac{f_{k,\ell}}{T_i} \right\rceil C_i = f_{k,\ell} \tag{2.5}$$

by using fix-point iteration.

Subsequently, under preemptive fixed-priority scheduling, an arbitrary-deadline task τ_k is schedulable if and only if the largest response time of all jobs of task τ_k that are released during a *level-k busy interval* is no more than the task's relative deadline i.e., $\max_{1 \leq \ell \leq h} R_{k,\ell} \leq D_k$.

While this provides an exact schedulability test, the test has exponential-time complexity. This is due to the reason that the length of the longest *level-k busy interval* can be up to the task set's hyper-period that can be exponential with respect to the input size.

Polynomial-time schedulability test for sporadic arbitrary-deadline task sets

Due to the exponential-time complexity of the exact *time-demand-analysis* in sporadic arbitrary-deadline task sets, more efficient but approximate schedulability tests are required. Since the schedulability analysis by Bini et al. [16] is extensively used in this thesis, the underlying concepts are explained in sufficient detail.

In contrast to other efficient sufficient schedulability tests for sporadic arbitrary-deadline task sets e.g., the analysis proposed by Fisher et al. [9] that use request-bound functions to determine the schedulability, the presented analyses use a task's worst-case idle time to retrieve each task's response time.

2.2.4 Definition (worst-case workload [16]). The *worst-case workload* $W_i(t)$, denotes the maximum amount of time over any contiguous interval of length t during which at least one of the i highest priority tasks is executing:

$$W_i(t) \stackrel{\text{def}}{=} \sum_{k=1}^{i} \min \left\{ t - (T_k - C_k) \left\lfloor \frac{t}{T_k} \right\rfloor, \left\lceil \frac{t}{T_k} \right\rceil C_k \right\}. \tag{2.6}$$

2.2.5 Definition (worst-case idle time [16]). The *worst-case idle time* $H_i(t)$ defines the longest (worst) idle time of the i highest-priority tasks over any interval of length t.

An important property is the *continuity* of the worst-case idle time $H_i(t)$ and the worst-case workload $W_i(t)$ i.e., $t = H_i(t) + W_i(t)$. Based on this property, a lower bound (lb) of the worst-case idle time i.e., the amount of time that the i highest-priority tasks are at least not executing can be expressed by the upper bound of worst-case workload (ub). More formally, $H_i^{lb} \stackrel{\text{def}}{=} t - W_i^{ub} \leq t - W_i(t) = H_i(t)$.

Furthermore, the authors define a pseudo-inverse function

$$X_i(c) = \min \{ t \mid H_i(t) \geq c \} \tag{2.7}$$

where $X_i(c)$ denotes the first instant in time t for which at least c amount of time the i highest-priority tasks are idling. The simple property that for all totally ordered sets S, S' for which $S \subseteq S'$ holds, it must be that $\min(S) \geq \min(S')$ yields

$$X_i^{ub}(c) = \min\{t | H_i^{\ell b}(t) \geq c\} \geq \min\{t | H_i(t) \geq c\}. \tag{2.8}$$

Given that each task executes whenever it has pending workload and is the highest-priority task, the largest response time amongst all ℓ jobs of task τ_k can be expressed as

$$R_k = \max_{\ell \geq 1} \{X_{k-1}(\ell \cdot C_k) - (\ell - 1) \cdot T_k\}. \tag{2.9}$$

Intuitively, this means that if all higher priority tasks of task τ_k are not executing then by the aforementioned execution model τ_k must be executing. Thus, the smallest time interval in which at least $\ell \cdot C_k$ processor time for the completion of ℓ jobs is guaranteed, denotes the finishing time of the $\ell - th$ job.

In order to allow for efficient analyses, a safe approximated upper bound of the largest response time can be given by

$$R_k^{ub} = \max_{\ell \geq 1} \{X_{k-1}^{ub}(\ell \cdot C_k) - (\ell - 1) \cdot T_k\} \geq R_k. \tag{2.10}$$

By linearizing Eq. (2.6), the workload function can be approximated to

$$W_i^{ub}(t) \stackrel{\text{def}}{=} \sum_{j=1}^{i} U_j \cdot t + C_j \cdot (1 - U_j). \tag{2.11}$$

By the discussed properties, the worst-case idle time can be approximated to

$$H_i^{ub}(t) \stackrel{\text{def}}{=} t - \sum_{j=1}^{i} U_j \cdot t + C_j \cdot (1 - U_j), \tag{2.12}$$

which can be inverted to

$$X_i^{ub}(h) = \frac{h + \sum_{j=1}^{i} C_j(1 - U_j)}{1 - \sum_{j=1}^{i} U_j}. \tag{2.13}$$

In conclusion, the upper bound of the response time of task τ_k is thus given by

$$R^{ub} = \max_k \left\{ \frac{kC_i + \sum_{j=1}^{i-1} C_j(1 - U_j)}{1 - \sum_{j=1}^{i-1} U_j} - (k-1)T_i \right\}. \tag{2.14}$$

Bini et al. [16] proved that the response time in Eq. (2.14) is maximized for $k = 1$ and thus

$$C_i + \sum_{j=1}^{i-1} C_j(1 - U_j) \leq D_i(1 - \sum_{j=1}^{i-1} U_j)$$
$$C_i + \sum_{j=1}^{i-1} T_i U_j \leq T_i \tag{2.15}$$

are sufficient conditions for the schedulability of sporadic arbitrary-deadline task sets.
Another approach to upper bound the largest response time of an arbitrary-deadline task is to linearize the previously defined request-bound function i.e.,

$$\text{RBF}_i^*(t) \stackrel{\text{def}}{=} C_i + U_i \cdot t. \tag{2.16}$$

Based on this approximation, it is to verify that for each released $\ell - th$ job of task τ_k in the *level-k busy* interval $\exists t\ (\ell - 1) \cdot T_k < t \leq (\ell - 1) \cdot T_k + D_k$ such that

$$\ell C_k + \sum_{\tau_i \in hp(\tau_k)} C_i + U_i \cdot t \leq t \tag{2.17}$$

is satisfied. As will be further detailed in Section 2.2.3 *Partitioned Scheduling*, this approximated request-bound function can be used to construct a partitioned deadline-monotonic scheduling algorithm.

2.2.3 Multiprocessor Scheduling

In the domain of multiprocessor scheduling algorithms, the three scheduling strategies *Global Scheduling*, *Semi-Partitioned Scheduling* and *Partitioned Scheduling* can be identified. Since an extensive survey of multiprocessor scheduling algorithms is beyond the scope of this thesis, this section gives an introduction to the aforementioned multiprocessor scheduling strategies with an emphasis on *partitioned scheduling* algorithms due to their relevance and predominant use in this thesis.

Global Scheduling

Global scheduling algorithms are characterized by the fact that all m processors in the system share the same *ready-queue*. That is, at each point in time at most m ready tasks that are decided by the scheduling algorithm are executing.
Furthermore, each task may migrate between processors i.e., a job is eligible to execute on any of the m processors. The largest benefit of global scheduling is that optimal real-time scheduling algorithms exist e.g., *pfair* for implicit-deadline task systems [8]. Additionally,

due to the shared ready-queue all workload is implicitly balanced between the available processors.
However, global scheduling suffers from migration costs, inter-processor synchronization and increased cache misses due to task migrations and subsequent *cold caches*.
Additionally, the accesses to the ready-queue must be synchronized between the available processors which can lead to idle processors due to waiting. This problem further increases with the number of processors since the probability for access conflicts increases proportionally.
In principle, the same priority schemes as in the uniprocessor case e.g., *Global-EDF*, *Global-RM* or *Global-DM* are possible strategies, but these algorithms have a least-upper utilization bound of 100% [30] and thus a capacity augmentation factor of $\Omega(m)$.

Partitioned Scheduling

In partitioned scheduling, each processor has its own ready-queue and scheduler i.e., the scheduling decisions are confined to that specific processor and thus independent of one another. In other words, each task is restricted to release jobs to and execute on a designated processor such that at each point in time each processor executes the highest-priority task that is in the processors respective ready-list. Therefore, all tasks must be partitioned onto one of the available processors in the system.
An advantage of partitioned scheduling is that the multiprocessor scheduling problem is reduced to multiple uniprocessor scheduling problems such that all of the theory and results can be used directly. Furthermore, due to the lack of task migrations, the cache hit-rates are improved which leads to overall shorter execution times. Additionally, due to the individual ready-queues of the processors, the schedulers do not need to synchronize before accessing the ready-queues and thus save cpu cycles.
On the flip side, the partitioning of sporadic arbitrary-deadline task sets is NP-hard in the strong sense since the special case of partitioning sporadic implicit-deadline task sets could be shown to be an instance of bin-packing [40], which is proven to be NP-hard in the strong sense. Therefore, unless $P = NP$, no polynomial-time algorithm exists, that solves this partitioning problem optimally and no fully polynomial-time approximation scheme (FPTAS) can exist for this problem.
Thus, in order to solve the partitioning problem efficiently, approximations are used to calculate feasible partitions with provable worst-case performance bounds with respect to optimal solutions.

2.2.6 Definition (Partition). A task set partition $\mathbf{P} \stackrel{\text{def}}{=} \{\mathbf{P_1}, \mathbf{P_2}, .., \mathbf{P_m}\}$ for a task set $\mathbf{T}$ is a set of task sets such that $\mathbf{P_1} \cup \mathbf{P_2} \cup .. \cup \mathbf{P_m} = \mathbf{T}$.

A task set that is sorted according to some property e.g., the task's utilization, period or deadline, is then partitioned to a set of processors successively.

Each partition $\mathbf{P}_\ell$ represents the set of tasks that are assigned to the $\ell - th$ processor.
In order to define a feasible partition, each partition is attributed a *capacity* B_ℓ such that all tasks in a given partition must not exceed that capacity.
The capacity measure itself is dependent on the schedulability test used to verify the feasibility of a given partition.
In the case of the partitioning of implicit-deadline tasks to a set of homogeneous processors that use EDF scheduling, the remaining processor utilization can be used as a capacity measure. That is, each processor has capacity 100% initially and is decremented by the task's utilization C_i/T_i, when task τ_i is partitioned to that processor.
Based on the above definitions, the following common greedy partitioning approaches *First-Fit*, *Best-Fit* and *Worst-Fit* can be identified.

2.2.7 Definition (First-Fit). Each task τ_i that is subject to partitioning is assigned to the first partition $\mathbf{P}_\ell$ such that the assignment of τ_i to $\mathbf{P}_\ell$ does result in the violation of the capacity constraint B_ℓ.

2.2.8 Definition (Best-Fit). Each task τ_i that is subject to partitioning is assigned to the partition $\mathbf{P}_\ell$ such that $\ell \stackrel{\text{def}}{=} \arg\min_{1\leq\ell\leq m}\{B_\ell - \mathit{capacity}(\tau_i)\}$. In other words, the task is assigned to the processor with the smallest capacity left that still has sufficient capacity to take τ_i.

2.2.9 Definition (Worst-Fit). Each task τ_i that is subject to partitioning is assigned to the partition $\mathbf{P}_\ell$ such that $\ell \stackrel{\text{def}}{=} \arg\max_{1\leq\ell\leq m}\{B_\ell - \mathit{capacity}(\tau_i)\}$. In other words, the task is assigned to the processor with the largest capacity left that has sufficient capacity to take τ_i.

Using the above partitioning strategies, the following generic partitioning algorithm Alg. 2.1 can be defined.
Depending on the facilitated uniprocessor scheduling algorithms and associated schedulability tests as well as ordering strategies, concrete partitioned scheduling algorithms can be constructed as will be demonstrated in the remainder of this section.

Partitioned EDF for arbitrary-deadline task sets

Since EDF is an optimal preemptive uniprocessor scheduling algorithm and is used in this thesis for evaluation purposes, the facilitated techniques and concepts of the partitioned EDF algorithm as proposed by Baruah et al. [9] are presented in detail.
In order to formally quantify the workload demand that each task may request, the concept of the demand-bound function is introduced.

Require: Task set $\mathbf{T}$, sorting preference, partitioning preference, capacity measure, number of processors m.
Ensure: Feasible partitions if one could be found.
1: re-index $\mathbf{T}$ by preference
2: **for each** τ_i in $\mathbf{T}$ **do**
3: **if** $\exists \ell \in \{1, 2, .., m\}$ such that $B_\ell \geq capacity(\tau_i)$ **then**
4: choose $\ell \in \{1, 2, .., m\}$ by preference such that $B_\ell \geq capacity(\tau_i)$
5: $\mathbf{P}_\ell \leftarrow \mathbf{P}_\ell \cup \{\tau_i\}$
6: **else**
7: **return** No feasible partition found
8: **end if**
9: **end for**
10: **return** $\mathbf{P_1}, \mathbf{P_2}, .., \mathbf{P}_\ell$

Algorithmus 2.1: A generic partitioned scheduling algorithm that assigns tasks successively according their initial ordering to processors by preference.

2.2.10 Definition (demand-bound function [13]). The demand-bound function

$$\text{DBF}(\tau_i, t) \stackrel{\text{def}}{=} \max \left\{ 0, \left(\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1 \right) \cdot C_i \right\}, t > 0 \tag{2.18}$$

defines the maximum amount of workload that can be released in and must be finished in any contiguous interval of length t.

Using the demand-bound function, it can be proved that a sporadic arbitrary-deadline task set is schedulable on a uniprocessor platform if at no time, the cumulative demand of all tasks is greater than the service that can be provided by the processor.

2.2.11 Theorem (schedulability test [13]). *A task set of sporadic arbitrary-deadline tasks* $\mathbf{T}$ *is schedulable on a uniprocessor platform if*

$$\forall\, t > 0 \sum_{\tau_i \in \mathbf{T}} \text{DBF}(\tau_i, t) \leq t. \tag{2.19}$$

To verify the schedulability according to Eq. (2.19), every discontinuity in any of the task's demand-bound functions that are within the hyper-period H (least-common multiple of the periods $T_1, T_2, .., T_k$)

$$\bigcup_{\tau_i} \{D_i, D_i + T_i, .., D_i + \ell \cdot T_i\} \subseteq [0, H] \tag{2.20}$$

must be tested.
Subsequently, this schedulability test has exponential-time complexity due to the size of the hyper-period that can be at worst exponential in the input size. The hyper-period denotes the smallest time interval such that the schedule is repeated.

A common approach to reduce the complexity of the schedulability test in Eq. (2.19) is to linearize the demand-bound function such that the testing of a single point for each task is sufficient.
More precisely, the demand-bound function can be approximated as follows

$$\text{DBF}^*(\tau_i, t) \stackrel{\text{def}}{=} C_i + U_i \cdot (t - D_i) \text{ if } t \geq D_i \text{ and 0 otherwise.} \tag{2.21}$$

By the property that $\text{DBF}_i^*(t) \geq \text{DBF}_i(t)$ and $\text{DBF}^*(\tau_i, D_i) \leq D_i$, it is implied that $\text{DBF}^*(\tau_i, t) \leq t \; \forall t > 0$.
Therefore, a sporadic arbitrary-deadline task τ_i is schedulable by EDF on a uniprocessor platform if $\text{DBF}^*(\tau_i, D_i) \leq D_i$ holds or consequently the sporadic arbitrary-deadline task set $\mathbf{T}$ is feasibly schedulable by EDF if for each task τ_k

$$C_k + \sum_{\tau_i \in hp_k} \text{DBF}^*(\tau_i, D_k) \leq D_k \tag{2.22}$$

holds where all previously assigned higher-priority tasks hp_k are already verified to be schedulable.
Using the approximated demand-bound function, a partitioning algorithm can be constructed. By Eq. (2.21), it is obvious that the tasks should be partitioned in deadline-monotonic succession i.e., $D_i \leq D_j$ if $i < j$ since then the approximated demand-bound function must not consider the zero case separately which is illustrated in Alg. 2.2.

2.2.12 Theorem (EDF Schedulability [9]). *Let a sporadic arbitrary-deadline task set* $\mathbf{T}$ *be ordered such that* $i < j$ *if* $D_i \leq D_j$*, then* $\mathbf{T}$ *is feasibly schedulable by EDF on a uniprocessor if for each task* τ_k

$$C_k + \sum_{i=1}^{k-1} C_i + U_i \cdot (D_k - D_i) \leq D_k \text{ and}$$
$$C_k + \sum_{i=1}^{k-1} U_i \leq T_k$$

holds under the assumption that all previously assigned tasks are already deemed schedulable.

In conclusion, the algorithm Alg. 2.2 finds an EDF-feasible i.e., schedulable partition of the sporadic arbitrary-deadline task set.

Partitioned DM for arbitrary-deadline task sets

For partitioned deadline-monotonic scheduling, an approach is to use the analyses presented in Section 2.2.2 *Uniprocessor Scheduling* that require to verify the response time of

Require: Sporadic arbitrary-deadline task set $\mathbf{T}$, number of processors m.
Ensure: Feasible partition (if one could be found).
1: re-index $\mathbf{T}$ such that $D_i \leq D_j$ for $i < j$
2: **for each** τ_i in $\mathbf{T}$ **do**
3: **if** $\exists \ell \in \{1, 2, .., m\}$ such that $\mathbf{P}_\ell \cup \{\tau_i\}$ is schedulable according to Thm. 2.2.12 **then**
4: choose $\ell \in \{1, 2, .., m\}$ by preference such that τ_i is schedulable on $\mathbf{P}_\ell$ according to Thm. 2.2.12
5: $\mathbf{P}_\ell \leftarrow \mathbf{P}_\ell \cup \{\tau_i\}$
6: **else**
7: **return** No feasible partition found
8: **end if**
9: **end for**
10: **return** $\mathbf{P_1}, \mathbf{P_2}, .., \mathbf{P}_\ell$

Algorithmus 2.2: Partitioned earliest-deadline first (EDF) scheduling as proposed by Baruah et al. [9], finds a feasible partition (if one could be found).

each job in each task's *level-k busy interval*. As was argued, this test exhibits exponential-time complexity in the worst-case.

In order to reduce the algorithmic complexity, Fisher, Baruah and Baker [34] provided an approximated polynomial-time test, using the linearized request-bound function (cf. Def. 2.2.1)

$$\text{RBF}_i^*(t) \stackrel{\text{def}}{=} C_i + U_i \cdot t. \tag{2.23}$$

With reference to the analysis presented in Section 2.2.2 *Uniprocessor Scheduling* and the above approximation, it is to verify that for each $\ell - th$ released job in the *level-k busy interval* the conditions in Eq. (2.24) $\exists t \ (\ell - 1) \cdot T_k < t \leq (\ell - 1) \cdot T_k + D_k$ such that

$$\ell C_k + \sum_{\tau_i \in hp(\tau_k)} C_i + U_i \cdot t \leq t \tag{2.24}$$

holds.

Fisher et al. [34] proved that if the tasks are partitioned in deadline-monotonic succession i.e., $D_i \leq D_j$ if τ_i is partitioned prior to τ_j then for the partitioning, it is sufficient to test the conditions in Eq. (2.25a) and Eq. (2.25b)

$$C_k + \sum_{\tau_i \in \mathbf{P}_\ell} \left(1 + \frac{D_k}{T_i}\right) C_i \leq D_k \quad \text{and} \tag{2.25a}$$

$$U_k + \sum_{\tau_i \in \mathbf{P}_\ell} U_i \leq 1 \tag{2.25b}$$

which imply the the sufficient schedulability condition in Eq. (2.24) for each task [34].

The condition in Eq. (2.25b) ensures that the workload after D_k is not underestimated when arbitrary deadline task systems are considered.

Require: Sporadic arbitrary-deadline task set $\mathbf{T}$, number of processors m.
Ensure: Feasible partition (if one could be found).
1: re-index $\mathbf{T}$ such that $D_i \leq D_j$ for $i < j$
2: **for each** τ_i in $\mathbf{T}$ **do**
3: **if** $\exists \ell \in \{1, 2, .., m\}$ such that $\mathbf{P}_\ell \cup \{\tau_i\}$ satisfies Eq. (2.25a) and Eq. (2.25b) **then**
4: choose $\ell \in \{1, 2, .., m\}$ by preference such that τ_i satisfies Eq. (2.25a) and Eq. (2.25b) on $\mathbf{P}_\ell$
5: $\mathbf{P}_\ell \leftarrow \mathbf{P}_\ell \cup \{\tau_i\}$
6: **else**
7: **return** No feasible partition found
8: **end if**
9: **end for**
10: **return** $\mathbf{P_1}, \mathbf{P_2}, .., \mathbf{P}_\ell$

Algorithmus 2.3: Partitioned deadline-monotonic scheduling as proposed by Fisher et al. [34], finds a feasible partition (if one could be found).

It is to note that the sufficient schedulability test for arbitrary-deadline task sets detailed in Section 2.2.2 *Polynomial-time schedulability test for sporadic arbitrary-deadline task sets* can be used for the partitioning as well.

Furthermore, the sufficient schedulability test by Bini et al. [16] dominates the here presented partitioning algorithm such that the worst-case performance of this algorithm is an upper bound for the worst-case performance of the schedulability test by Bini et al. [16].

Semi-Partitioned Scheduling

The semi-partitioned scheduling algorithms are a conceptual compromise of global and partitioned scheduling algorithms.

That is, each task in the task set is assigned to a set of processors that it is eligible to execute on. If a task is assigned to a set of processors, then this task may be preempted on the current processor and resume its execution on another processor.

To that end, global scheduling can be interpreted as an extreme case of semi-partitioned scheduling in which each task is assigned to the set of all processors. Likewise, by the restriction of each task to execute on a specific processor, semi-partitioned scheduling degenerates to partitioned scheduling.

Using semi-partitioned scheduling, the improved schedulability of global scheduling and the advantages of partitioned scheduling i.e., improved cache affinity, fewer migrations and fewer synchronization efforts can be balanced by concrete task to processor assignments.

2.3 Real-Time Scheduling Analyses

In hard real-time systems it must be provably guaranteed on the algorithmic level that no admitted task misses its relative deadline upon activation in any concrete schedule generated by the scheduling algorithm under analysis.
Due to the fact that schedulability analyses only prove the timing correctness of the considered models, the models should resemble the real system in sufficient detail.
Furthermore, it is required to design efficient schedulability tests to decide whether a task set (that satisfies model and system constraints) can be scheduled (feasibly) by a specific scheduling algorithm.
From this, it follows that the number of task sets (with respect to the set of all possible task sets) that are admitted to the systems are decided and thus potentially reduced by the test algorithm.
Especially in the context of large-scale produced embedded systems, that are thus energy, resource and cost constrained e.g., automotive systems, the schedulability analyses should preferably be as exact as possible.
Since the workloads or task sets are determined by the application, non-exact schedulability tests may enforce to use more powerful hardware systems in order to guarantee to feasibly host all applications and thus increase costs. Unfortunately, as already discussed in the previous chapters, most exact schedulability analyses for uniprocessor and multiprocessor systems are at least NP-complete and thus practically infeasible to use for large enough problem instances. To that end, approximate schedulability analyses and tests are required, which is more precisely elaborated on in the remainder of this section.
Therefore, the concept of sufficient, necessary and exact scheduling test algorithms are described in Section 2.3.1 *Schedulability tests*.
In the following Section 2.3.2 *Competitiveness Analyses*, defintions and techniques to formalize the worst-case approximation performance i.e., competitiveness of approximate schedulability analyses and tests are presented.

2.3.1 Schedulability tests

A schedulability test $\mathcal{T}$ can be formally described as an algorithm $\mathcal{T}_{\mathcal{A}}(\mathbf{T})$ that decides whether a scheduling algorithm $\mathcal{A}$ can feasibly schedule the input task set $\mathbf{T}$ where the task set consists of tasks that adhere to a specific task model e.g., the aforementioned sporadic constrained-deadline task model.

2.3.1 Definition (Feasibly schedulable). A task set $\mathbf{T}$ is called feasibly schedulable by a scheduling algorithm $\mathcal{A}$, if no task in any concrete schedule, that is generated by the scheduling algorithm $\mathcal{A}$, for any legal release pattern of tasks in $\mathbf{T}$ misses its relative deadline.

If $\mathcal{T}$ deems that $\mathcal{A}$ can feasibly schedule $\mathbf{T}$, then no task in any concrete schedule, that is constructed by $\mathcal{A}$, for any legal release pattern of tasks in $\mathbf{T}$ misses its relative deadline. From this definition it follows, that $\mathcal{T}$ is a binary output algorithm (schedulable, not schedulable). Furthermore, schedulability tests can be classified into sufficient, necessary and exact tests, which are explained in the following definitions.

2.3.2 Definition (Sufficient schedulability test). A schedulability test $\mathcal{T}_\mathcal{A}$ for algorithm $\mathcal{A}$ is considered a sufficient schedulability test if all task sets that are deemed to be schedulable by the schedulability test, are feasibly schedulable by the scheduling algorithm $\mathcal{A}$.

2.3.3 Definition (Necessary schedulability test). A schedulability test $\mathcal{T}_\mathcal{A}$ for algorithm $\mathcal{A}$ is a necessary schedulability test if all task sets that are feasibly schedulable by the scheduling algorithm $\mathcal{A}$, are deemed to be schedulable by the test algorithm.

2.3.4 Definition (Exact schedulability test). A schedulability test $\mathcal{T}_\mathcal{A}$ for algorithm $\mathcal{A}$ is an exact schedulability test if $\mathcal{T}_\mathcal{A}$ is a sufficient and necessary schedulability test.

With these definitions, it is possible to construct and formalize approximation algorithms as well as analyze their respective worst-case performance.

2.3.2 Competitiveness Analyses

It is in general the aim to construct exact schedulability tests such that all task sets, that can be feasibly scheduled by the scheduling algorithm under analysis, are also deemed schedulable by the schedulability test and vice versa.
Unfortunately, most relevant real-time scheduling decision problems and related problems e.g., task partitioning do not admit optimal polynomial-time algorithms.
For example, according to the researches and survey by Ekberg et al. [31] it could be shown that uniprocessor fixed-priority scheduling of ordinary sporadic constrained-deadline task sets is weakly NP-complete and weakly NP-hard for arbitrary-deadline task sets. Further, it could be proved that the dynamic priority scheduling of constrained-deadline and arbitrary-deadline task sets is coNP-complete [31].
As an example for multiprocessor scheduling, Chen proved that even deciding whether an implicit-deadline task set with equal periods on multiprocessor systems is NP-complete in the strong sense [25].
Thus, unless $P = NP$ no polynomial-time complexity solution to these problems exists, which motivates the design of heuristics or approximation algorithms to determine the schedulability of real-time task sets efficiently.
The term *heuristics* classifies algorithms that generate *practically* good, but unquantified worst-case solutions compared to an optimal solution. In contrast to that, approximation

algorithms generate solutions that are proved to have bounded worst-case deviation from an optimal solution.

Due to the fact, that schedulability tests are binary output algorithms, it is unreasonable to approximate the output i.e., to approximate all outputs to *not schedulable* for sufficient schedulability tests or to *schedulable* for necessary schedulability tests.

Therefore, the *speedup factors*, *capacity augmentation factors* and the respective analyses are established metrics and techniques to denote the worst-case performance of an approximate scheduling test algorithm. In other words, the worst-case performance of a schedulability test $\mathcal{T}_1$ is quantified with respect to another scheduling test algorithm $\mathcal{T}_2$ by means of *speedup factors* or *capacity augmentation factors* as detailed in the following.

In the *speedup factor* analysis, the approximation is expressed in terms of processor speeds i.e., the scaling of all parameters that relate to the processor speeds like the worst-case execution time of a task.

If a scheduling test algorithm $\mathcal{T}_1$ has a speedup factor of α with respect to another schedulability test $\mathcal{T}_2$ then it is guaranteed that for all task sets $\mathbf{T}$ that are deemed schedulable by $\mathcal{T}_2(\mathbf{T})$ are also deemed schedulable by $\mathcal{T}_1$ when the processor speeds are increased by α.

Conversely, if a task set $\mathbf{T}$ is not deemed schedulable by $\mathcal{T}_1$ on a unit-speed processor platform, then said task set remains unschedulable by $\mathcal{T}_2(\mathbf{T})$ when the processors are slowed down by $\frac{1}{\alpha}$.

Another proposed technique to quantify the worst-case approximation performance is given by the *capacity-augmentation bound*. In this method, a parametric threshold value is used to bound relevant (to the scheduling) parameters of the input task set. That is for example to bound each task's utilization by a parameter b such that the utilization $U_i \leq \frac{1}{b}$ and the cumulative utilization of the task set is no more than $\sum_{\tau_i} U_i \leq \frac{M}{b}$ [25]. It is to note, that the parametrically bounded task set parameters should reflect or should be entailed in a necessary condition for the schedulability of the task model under analysis and thus the task set parameters to be bounded may vary with the task model.

Depending on the scheduling algorithm and the associated test algorithm, either the *speedup factor* analysis or the *capacity-augmentation factor* analysis is more natural to the problem.

For completeness, it is to emphasize that the speedup factor and capacity-augmentation bound are based on worst-case task sets i.e., both metrics denote the performance of the scheduling test algorithm with respect to a worst-case problem instance, that may never occur in real systems [25, 57]. In order to avoid over-interpretations of the aforementioned metrics, it is possible and may be beneficial to classify i.e., confine the parameter-space of the input task sets to those that can actually occur or are prevalent in real-time systems under consideration.

3 DAG Task Scheduling

In contrast to sequential task models, a parallel real-time task model must allow for the modeling of parallel execution i.e., model intra-task dependencies. A commonly used model to describe these intra-task dependencies is the Directed-Acyclic-Graph (DAG). The scheduling of these parallel real-time tasks is widely researched in various directions using various approaches and techniques.
Therefore, this section briefly surveys the different approaches and details selected algorithms that are relevant to this thesis either conceptually or due to their comparative use in the evaluations.
In the first Section 3.1 *Sporadic DAG task model*, the DAG task model is introduced and detailed.
In the following Section 3.2 *DAG task Scheduling algorithms*, decomposition-based, list scheduling, agnostic scheduling, federated scheduling and semi-federated scheduling algorithms are described.
In the last Section 3.3 *Issues of Federated Scheduling*, the weaknesses and underlying problems of the federated scheduling algorithms are discussed.

3.1 Sporadic DAG task model

Traditionally, the execution demand of each task τ_i is solely described by the worst-case execution time (WCET) C_i as in uniprocessor platforms there is no need to express potential parallel execution paths since the processor only executes one job at one time point.
Multiprocessor platforms on the other hand allow *inter-task parallelism*, which allows for the execution of sequential programs concurrently and *intra-task parallelism*, which allows a parallelized task to be executed in parallel at the same time. A common model to describe the intra-task dependencies is the Directed-Acyclic-Graph (DAG). That is, each node represents the worst-case execution time of a subtask and the set of edges denote the subtasks precedence constraints more formally explained in the following definition.

3.1.1 Definition. A sporadic DAG task τ_i is defined by the tuple (G_i, D_i, T_i) where G_i denotes a Directed-Acyclic-Graph (V_i, E_i) such that each $c_j \in V_i$ denotes the worst-case execution time of the $j-th$ subtask $\tau_{i,j}$ and an edge $(c_j, c_\ell) \in E_i$ denotes the precedence constraints i.e., the activation of subtask $\tau_{i,\ell}$ depends on the finishing of subtask $\tau_{i,j}$.

Further, each DAG task releases an infinite sequence of instances called jobs that are separated by (at least) the minimal inter-arrival time T_i. Whenever a DAG task is released at some time t_r, then all subtasks must finish within the interval $(t_r, t_r + D_i]$.

Based on the DAG structure, the following parameters can be deduced. The total execution time (or work) C_i of task τ_i is defined by the summation of the worst-case execution times of all the subtasks of a task.
Furthermore, the critical path length L_i of task τ_i denotes the length of the critical path in the given DAG, in which each node is characterized by the worst-case execution time of the corresponding subtask of task τ_i. In other words, the critical path length is the worst-case execution time of the task on an infinite number of processors.
Therefore, if $L_i > D_i$ then the DAG task τ_i can not be scheduled by any scheduling algorithm even on an infinite number of processors.
Additionally, the critical path length is no more than the cumulative worst-case execution time i.e., $C_i \geq L_i > 0$ for every task τ_i. The *utilization* $\frac{C_i}{T_i}$ of task τ_i is denoted by U_i. Further, $C_i / \min\{T_i, D_i\}$ denotes the *density* of task τ_i. Furthermore, the response time R_i of the DAG task τ_i denotes the finishing time of the last finishing subtask.
Another variant of the presented DAG task model is to reduce the description to the parameters (C_i, L_i, D_i, T_i) such that this model is completely agnostic of the internal parallelization structure, i.e., how many subtasks exist and how the precedence constraints amongst them are. Subsequently, schedulability tests that use this model allow for the change of the DAG structure during runtime as long as the parameter constraints are met.

3.1.2 Definition. A parametric sporadic DAG task τ_i is defined by the tuple (C_i, L_i, D_i, T_i) where C_i denotes the total execution time and L_i denotes the critical path length. Further, each DAG task releases an infinite sequence of instances called jobs that are separated by (at least) the minimal inter-arrival time T_i. Whenever a DAG task is released at some time t_r, then C_i amount of workload must be finished within the interval $(t_r, t_r + D_i]$.

On the flip side, any schedulability analysis needs to assume the worst-case structure i.e., the sequential execution for L_i amount of time followed by the fully (all available processors) parallelized execution of the remaining $C_i - L_i$ amount of workload in the remaining $D_i - L_i$ amount of time to meet the deadline.

3.2 DAG task Scheduling algorithms

This section briefly surveys related work and the different approaches to the scheduling of DAG task sets, namely *agnostic*, *decomposition-based* and *federated scheduling*. Additionally, the concept of the class of federated scheduling algorithms is presented in more detail and is thoroughly discussed with respect to underlying issues and limitations.

3.2.1 Agnostic

The first approach is to not utilize the DAG structure and parameters of sporadic tasks. That is, whenever a subtask of a given DAG task is released and ready to be executed, the standard global or partitioned multiprocessor scheduling algorithms e.g., Global-RM or Global-DM are used to schedule the subtasks [1, 17, 22, 45].

3.2.2 List Scheduling

Another analytically and conceptually important algorithm that can be used for the scheduling of DAG task sets is list scheduling that was proposed by Graham [35].
In the list scheduling approach, any idle processor can execute any ready subtask of a DAG task.
More precisely, whenever all precedence constraints for any subtask are met, this subtask is inserted into a ready-list. Then, the highest-priority subtask is dispatched from the ready-list and executed (and run to completion) on the first processor that has no remaining workload to execute.
Furthermore, based on the fixed-priority schemes that are used to assign priorities to the subtasks, different list scheduling variants have been proposed.
Under the assumption that at each point in time only a singular DAG task instance is ready, using the analysis proposed by Graham [35], it can be proved that list scheduling has a speedup factor of at most $2 - \frac{1}{m}$ for any homogeneous multiprocessor system of size m irrespective of the facilitated priority schemes.
Unfortunately, even if only a singular DAG task is analyzed, the speedup factor analysis of list scheduling is infeasible for arbitrary-deadlines.
This is due to the fact that in the case of $D_i > T_i$, at most $\lceil D_i/T_i \rceil$ DAG instances might be pending simultaneously. Therefore, this additional workload must be accounted for by the analysis.

3.2.3 Decomposition-based approach

Another approach is to decompose a DAG task into a set of independent sequential tasks with specified relative deadlines and offsets of their release times accordingly. Thus the DAG structure is utilized off-line in order to apply the decomposition. These decomposed sequential tasks are then scheduled accordingly without considering the DAG structure anymore. This approach can be identified in the works like [39, 41, 42, 51, 53].

3.2.4 Federated Scheduling

In federated scheduling approaches, the DAG task set is partitioned into light and heavy tasks. Light tasks are those, that can be completely sequentialized and still fit on one

processor. A task that requires to execute on at least two processors in order to meet its deadline is considered a heavy task. In the original design of federated scheduling for implicit-deadline task systems proposed by Li et al. [46], a light task is solely executed *sequentially* without exploiting the parallelized structure and a heavy task is assigned to its designated processors, that *exclusively* execute only that heavy task.

In addition, the number of exclusively allocated processors for each task is determined by the following equation

$$m_i \stackrel{\text{def}}{=} \left\lceil \frac{C_i - L_i}{T_i - L_i} \right\rceil . \tag{3.1}$$

This number of processors is at least required to provide provably sufficient execution time for any (and thus worst) DAG structure under the task's parameter constraints.

Instead of restating the formal proof, that can be studied in the original paper [46], the following proof sketch or rather intuition is given.

Let t_r denote the time of a job release such that τ_i must finish all its execution demand C_i during the interval $[t_r, t_r + T_i]$. It is obvious, that the processors can execute jobs for at most $m_i \cdot T_i$ amount of time during said release and deadline interval. In order to provide sufficient execution time for any eligible DAG structure under the given parameter constraints, the following worst-case structure must be assumed. That is, the DAG structure of τ_i is such that for the first L_i amount of time only subtasks on the critical path are executed. By these precedence constraints, $m_i - 1$ processors are idle during the interval $[t_r, t_r + L_i]$.

This *waste of resources* can be interpreted as additional interference, which yields the following sufficient schedulability condition

$$(m_i - 1) \cdot L_i + C_i \leq m_i \cdot T_i. \tag{3.2}$$

In other words, a DAG task τ_i must be able to execute for the amount of its worst-case execution time C_i, whilst being subject to additional interference.

Federated Scheduling of sporadic arbitrary-deadline DAG task sets

Baruah [3–5] adopted the concept of federated scheduling for scheduling constrained-deadline and arbitrary-deadline task systems. In order to extend the federated scheduling approach to arbitrary-deadline DAG task sets, Baruah proposed an adaptation that can account for the additional workload, due to pending DAG jobs in arbitrary-deadline task sets [4].

That federated scheduling variant proposes to distinguish between DAG tasks with constrained deadlines i.e., $D_i \leq T_i$ and DAG tasks with strictly larger deadlines than their periods i.e., $D_i > T_i$. To better present this approach, both cases are detailed separately.

In the first case, all tasks are classified into heavy and light tasks. That is, a task is considered light if it does not require to execute in parallel in order to finish before its relative deadline i.e., $C_i \leq D_i$. Conversely, a heavy task requires to be executed in parallel in order to be able to finish before its relative deadline i.e., $C_i > D_i$. In contrast to the original federated scheduling approach, this variant uses the internal DAG structure of the tasks. Therefore, it is possible to retrieve a smaller number of processors that can feasibly schedule a single heavy DAG task using list scheduling.
This is done by iteratively constructing a concrete list schedule for the DAG task under analysis using an increasing number of processors until the response time is no more than the relative deadline. In accordance to the original federated scheduling, all light DAG tasks are partitioned onto the remaining processors.
In the second case ($D_i > T_i$), Baruah proposes to schedule heavy tasks on a set of processors using global EDF instead of list scheduling and the light tasks are scheduled according to partitioned EDF as presented in Section. 2.2.3.
Additionally, the parametric sporadic DAG task model is used. The benefit of EDF in the context of scheduling a single DAG task is that it is a workload conserving algorithm and all already pending DAG jobs are processed according to their absolute deadline and thus release order (first-in first-out).
A scheduling algorithm is considered workload-conserving if it never leaves a processor idle if there is any pending workload i.e., the ready-queue is not empty. The priority ordering is simply given due to the fact that if a job is released at t_0, then the previous job is released at $t_0 - T_i$ with absolute deadline $t_0 - T_i + D_i$, which is earlier than $t_0 + D_i$ and not finished yet due to $D_i > T_i$ and thus prioritized in the analysis window of interest.
In order to quantify the number of required processors, a sufficient condition for the schedulability of a single DAG task τ_i with $D_i > T_i$ on m processors is derived, which is stated in the following theorem.

3.2.1 Theorem (Global EDF schedulability test [4]). *A sporadic DAG task $\tau_i = (G_i, D_i, T_i)$ with $D_i > T_i$ is EDF-schedulable on m unit-speed processors if*

$$D_i U_i + \min\{m \cdot (D_i \bmod T_i), C_i\} + (m-1) \cdot L_i \leq m \cdot D_i. \tag{3.3}$$

Therefore, in this second case a heavy DAG task requires the smallest number of processors m, that satisfies Eq. (3.3).
Instead of restating the complete proof, a brief explanation and intuition shall be given. A system of m processors can at most provide $m \cdot t$ amount of service over any interval of length t, therefore the right-hand side of Eq. (3.3) bounds the maximum amount of service, the system can provide over any interval of length D_i. Conversely, the left-hand side represents an upper bound on the worst-case demand task τ_i can require from the system in order to be feasibly schedulable.

This demand consists of self-interference due to sequential execution, pending DAG jobs and the worst-case execution time of the DAG job under analysis. To illustrate the pending workload, Fig. 3.1 shows an exemplary release pattern of a DAG task, where the deadline is no-less than the period. It is rather easy to see, that at most $\lceil D_i/T_i \rceil$ pending jobs may have deadlines within an analysis window $[t_0, t_0 + D_i]$. This is simply, due to the constraints $t_0 < t_0 + D_i - \ell T_i \leq t_0 + D_i$, which yields the condition $D_i > \ell \cdot T_i$.
Clearly, the demand of each pending DAG jobs may never be more than C_i. Therefore $(1 + D_i/T_i) \cdot C_i$ would be a safe upper bound on the amount of workload.
An important insight is that the pending DAG job with the earliest absolute deadline after t_0 (cf. t_f in Fig. 3.1) overlaps with $[t_0, t_0 + D_i]$ for at most $D_i \ mod \ T_i$ amount of time. Therefore, the demand of this DAG job can be upper bounded by $\min\{m_i \cdot (D_i \ mod \ T_i), C_i\}$, since it could be possible that all m processors are used in the worst-case.
In conclusion, the algorithm distinguishes between heavy and light DAG tasks. The subset of heavy DAG tasks is further distinguished between constrained-deadline tasks and tasks with deadlines strictly larger than the periods i.e., $D_i > T_i$. If the DAG task under analysis is constrained-deadline, then a list schedule is constructed for a concrete number of processors.
If the largest response time is no more than the task's deadline then this number of processors is allocated. Afterwards, all light tasks are partitioned according to partitioned EDF.

Semi-Federated scheduling

In semi-federated scheduling as proposed by Jiang et al. [38], the resource waste of federated scheduling is adressed by computing the required processors more tightly. That is, if federated scheduling requires at least $m_i \geq x$ many processors with $m_i \in \mathbb{N}$, then federated scheduling will allocate $\lceil x \rceil = (x + \epsilon)$-many processors for this task.
This may lead to at most 200% of the required resources since $2 \geq \frac{x+1}{x+\epsilon}, x \geq 1$ for $\lim \epsilon \to 0$.
Conversely, the proposed improvements decrease with the number of required processors.
In summary, if federated scheduling requires $(x + \epsilon)$-many processors exclusively to serve a heavy DAG task, semi-federated scheduling assigns x-many processors to that task

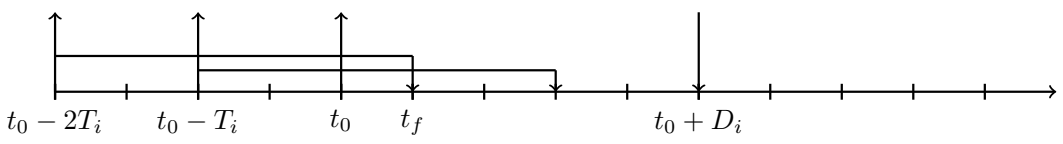


Figure 3.1: An exemplary release-pattern of a sporadic DAG task τ_i with $D_i > T_i$.

exclusively and schedules the remaining fraction of the execution demand ϵ together with the light tasks.
In order to schedule the fractional execution demands along with light tasks, the authors construct sequential *container tasks* that are designated to serve these fractional demands. Intuitively, the container tasks provide fractions of CPU-time e.g., 30% that are interpreted as-if a processor that is slowed down to 0.3 of the original speed was assigned exclusively. A further variant is to allow the fractions to be served by two container tasks.

3.3 Issues of Federated Scheduling

In contrast to implicit-deadline DAG task sets, the difference between a DAG task's deadline and a DAG task's period in constrained-deadline task sets induces potentially unbounded resource waste.
Intuitively, the issue of federated scheduling for constrained-deadline task systems is the exclusive allocation of processors to heavy tasks since the number of assigned processors is at least as large as the DAG tasks density i.e., $m_i \geq \delta_i$ in federated scheduling.
Since the allocation of a designated processor allows for 100% utilization which may never be used due to the task's low utilization, resources are wasted. This waste of resources and sub-optimality formally manifests itself in terms of a non-constant speedup factor.
Chen [20] proved that there exists at least one concrete sporadic constrained-deadline DAG task set for which a feasible schedule exists but that is unschedulable for any given homogeneous multiprocessor system of constant speed by federated scheduling. To better illustrate and motivate the problem, the proposed example by Chen is restated in the following.
Let a multiprocessor system consist of at least two processors i.e., $m \geq 2$ and let K be any arbitrary number with $K \geq 2$. Then, consider a sporadic constrained-deadline DAG task set such that

- $C_1 = m$, $D_1 = 1$, and $T_1 \to \infty$.
- $C_i = K^{i-2}(K-1)M$, $D_i = K^{i-1}$, and $T_i = \infty$ for $i = 2, 3, \ldots, n$.

Furthermore, each task τ_i consists of m subtasks with worst-case execution time $\frac{C_i}{m}$ that have no precedence constraints amongst another. In federated scheduling, all m processors are allocated for task τ_1 exclusively, since τ_1 can only be feasibly scheduled by running on all m processors in parallel i.e., $m_i \geq \delta_i = m$.
Similarly, the semi-federated scheduling approach [38] suffers from the same exclusive allocation problem. Despite, the fact that federated scheduling and semi-federated scheduling can not schedule the example task set, a feasible schedule can be achieved by assigning each subtask of task τ_i to one of the m processors.

By using the above task set, it could be proved that the speedup factor of federated scheduling for constrained-deadline DAG task sets is $\Omega(\min\{m, n\})$ [20].

4 Reservation-Based Federated Scheduling

In this chapter, the basic ideas and concepts of reservation-based federated scheduling for sporadic arbitrary-deadline DAG tasks are presented. One essential property of the proposed reservation server design and analysis is that it is independent of the actual structure of the DAG job. That is, it depends only on the worst-case execution time C_i and critical-path length L_i of the DAG job.
In Section 4.1 *Reservation Server Design*, the general concepts and ideas are detailed.
In the following Section 4.2 *Sufficient Conditions for Reservation Server Design*, sufficient conditions and constraints for the design of reservations are given.

4.1 Reservation Server Design

An inherent difficulty when analyzing the schedulability of DAG task systems is the intra-task dependency in conjunction with the inter-task interference. Federated scheduling avoids this problem by granting a subset of available processors to heavy tasks exclusively and therefore avoiding inter-task interference entirely. A natural generalization of the federated scheduling approach is to reserve sufficient resources for heavy tasks, but not necessarily entire processors.
In order to do so, the minimal amount of resources that are sufficient to schedule a DAG task over a release and deadline interval must be quantified.
In reservation-based federated scheduling, each DAG task τ_i gets m_i sequential reservation instances with $E_{i,1}, .., E_{i,\ell}, .., E_{i,m_i}$ service provisioning. It is mandatory i.e., enforced that the reservation servers are synchronous with the release of a DAG task's job. In particular, each reservation corresponding to a DAG task τ_i has the same deadline and inter-arrival time as τ_i.

4.1.1 Definition. The $\ell - th$ sporadic reservation $\tau_{i,\ell}$ for serving a DAG task τ_i is defined by the tuple $(E_{i,\ell}, D_i, T_i)$ such that $E_{i,\ell}$ amount of service, i.e., computation time, is provided to the DAG task over the interval $[t_0, t_0 + D_i)$ with a minimum inter-arrival time of T_i.

This means, the release pattern of a reservation is inherited from the DAG task and whenever a DAG task releases a job at t_r, the associated service is provided in the release- and deadline-interval $[t_r, t_r + D_i)$. In order to analytically treat a reservation as if it is

a sporadic arbitrary-deadline task, the reservation is enforced to be active, i.e., eligible for scheduling, until the whole runtime-budget is depleted. This also allows to schedule multiple instances of a reservation in parallel instead of sequencing multiple pending jobs of the same tasks in a first-in first-out manner. However, the reservation jobs that are released at time t_r by a reservation for task τ_i are used to serve the DAG job of task τ_i that arrived at time t_r exclusively.

Note that no restrictions about any reservation to service certain subjobs of the DAG task τ_i exclusively are made, but that reservations are eligible to service any subjob (a node in the DAG) as long as they belong to the instance that initiated the activation of the reservations.

Each DAG job is serviced by list scheduling, that is when a reservation of a job is active, it can execute any ready subjob of the DAG job. List scheduling is work-conserving — namely, at every point in time in which the DAG job has pending workload and the system provides service, some subjob is executing.

The exact time of service, i.e., the schedule of the reservations, is determined by the applied scheduling algorithm suited for the scheduling of sporadic arbitrary-deadline tasks on a homogeneous multiprocessor system with m processors. Therefore, the system can use any scheduling algorithm and schedulability test suitable for scheduling sporadic arbitrary deadline sequential tasks to schedule these reservations. The problem of scheduling DAG task sets and the analysis thereof is hence divided into the following two problems:

1. Scheduling of sporadic, arbitrary-deadline task sets.

2. Providing provably sufficient reservation to service a set of arbitrary DAG tasks.

The reservation concept is illustrated in Figure 4.1, where two reservations are partitioned on two different processors and scheduled according to an arbitrary scheduling algorithm.

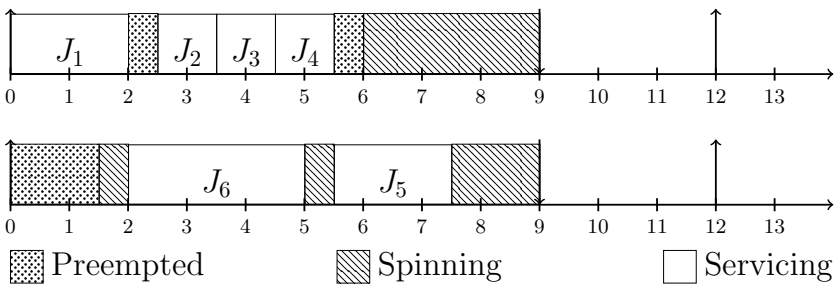


Figure 4.1: An arbitrary schedule of two equal reservations, as computed by the *R-MIN* algorithm. The DAG task shown in Figure 4.2 is serviced according to the *list-scheduling* algorithm by any reservation that does not service an unfinished job at that time. Both reservations provide 7.5 units of service over the interval $[0, 9)$ on two processors in parallel. The white areas denote, that the reservation is active whereas the hatchings denote the preemption or spinning of a reservation.

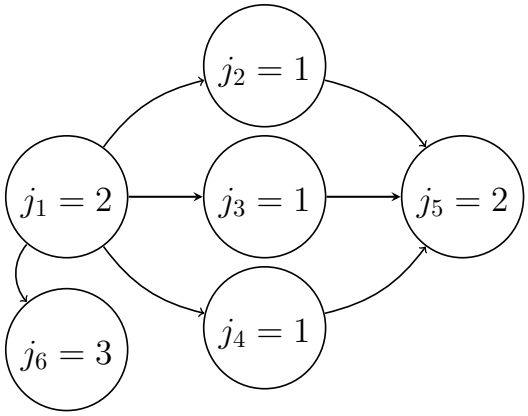


Figure 4.2: A sporadic, constrained-deadline DAG task τ_i with $C_i = 10$, $L_i = 5, D_i = 9, T_i = 12$.

The two reservations provide 7.5 time-units of runtime budget (as computed by the *R-MIN* algorithm, explained later) over the interval $[0, 9)$ in parallel to service the DAG task shown in Figure 4.2. To improve readability, higher priority tasks or reservations in the system are not included in Figure 4.1. Instead, the resulting preemptions and the spinning of the reservations are denoted by the different hatchings. Note that the total reservation of 15 necessarily exceeds the work of the DAG job 10 since the reservation must guarantee that the job completes regardless of how the reservation servers are scheduled by the underlying scheduling algorithm.

4.2 Sufficient Conditions for Reservation Server Design

In this section, suffcient conditions for designing reservation servers for DAG tasks are derived. In particular, it can be shown that any design that satisfies the second condition in the following theorem guarantees that the DAG job will complete by its deadline.

4.2.1 Theorem. *Suppose that m_i sequential instances (jobs) of real-time reservation servers are created and released for serving a DAG task τ_i with execution budgets $E_{i,1}, E_{i,2}, .., E_{i,m_i}$ when a job of task τ_i is released at time t_0. The job of task τ_i arrived at time t_0 can be finished no later than its absolute deadline $t_0 + D_i$ if*

- ***[Schedulability Condition]**: the m_i sequential jobs of the reservation servers can be guaranteed to finish no later than their absolute deadline at $t_0 + D_i$, and*
- ***[Reservation Condition]**: $C_i + L_i \cdot (m_i - 1) \leq \sum_{j=1}^{m_i} E_{i,j}$.*

Proof. Consider an arbitrary execution schedule S of the m_i sequential jobs executed from t_0 to $t_0 + D_i$. Suppose, *for contradiction*, that the reservation condition holds but there is an unfinished subjob of the DAG job of task τ_i at time $t_0 + D_i$ in S. Since the list scheduling algorithm is applied, the schedule for a DAG job is under a certain topological order and

is workload-conserving. That is, unless a DAG job has finished at time t, whenever the system provides service to the DAG job, one of its subjobs is executed at time t.

We define the following terms based on the execution of the DAG job of task τ_i that arrived at time t_0 in S. Let the last moment prior to $t_0 + D_i$ when the system provides service to the DAG job be f_ℓ and c_ℓ is a subjob of task τ_i executed at f_ℓ. Let θ_ℓ be the earliest time in S when the subjob c_ℓ is executed. After θ_ℓ is determined, among the predecessors of c_ℓ, let the one finished *last* be $c_{\ell-1}$. Furthermore, $f_{\ell-1}$ is the finishing time of $c_{\ell-1}$ and $\theta_{\ell-1}$ as the starting time of $c_{\ell-1}$. By repeating the above procedure, one can define θ_1, f_1, c_1, where there is no predecessor of c_1 any more in S. For notational brevity, let f_0 be t_0.

According to the above construction, the sequence $c_1, c_2, \ldots, c_\ell$ is a *path* in the DAG structure of τ_i. Let $exe(c_j)$ be the execution time of c_j. By definition, it must be that $\sum_{j=1}^{\ell} exe(c_j) \leq L_i$. In the schedule S, whenever c_j finishes, it is certain that c_{j+1} can be executed, but there may be a gap between f_j and θ_{j+1}.

Suppose that $\beta_i(x, y, S)$ is the accumulative amount of service provided by the m_i sequential jobs in an interval $[x, y)$ in S. Since the list scheduling algorithm is workload-conserving, if c_j is not executed at time t with $\theta_j \leq t \leq f_j$, then all the services are used for processing other subjobs of the DAG job of task τ_i. Therefore, for $j = 1, 2, \ldots, \ell$, the maximum amount of service that is *provided to the DAG job but not used* in time interval $[\theta_j, f_j)$ in S is at most $(m_i - 1)exe(c_j)$, since each of the m_i reservation servers can only provide its service sequentially. That is, in the interval $[\theta_j, f_j)$ at least $\max\{\beta_i(\theta_j, f_j, S) - exe(c_j) \times (m_i - 1), exe(c_j)\}$ amount of execution time of the DAG job is executed.

Similarly, for $j = 1, 2, \ldots, \ell$, the maximum amount of service that is *provided to the DAG job but not used* in time interval $[f_{j-1}, \theta_j)$ in S is 0; otherwise c_j should have been started before θ_j. Therefore, in the interval $[f_{j-1}, \theta_j)$ at least $\beta_i(f_{j-1}, \theta_j, S)$ amount of execution time of the DAG job is executed.

Under the assumption that the job misses its deadline at time $t_0 + D_i$ and the m_i sequential jobs of the reservation servers can finish no later than their absolute deadline at $t_0 + D_i$ in the schedule S, we know that

$$
\begin{aligned}
C_i &> \sum_{j=1}^{\ell} \beta_i(f_{j-1}, \theta_j, S) + \max\{\beta_i(\theta_j, f_j, S) - exe(c_j)(m_i - 1), exe(c_j)\} \\
&\geq \sum_{j=1}^{\ell} \beta_i(f_{j-1}, \theta_j, S) + \beta_i(\theta_j, f_j, S) - exe(c_j)(m_i - 1) \\
&= \sum_{j=1}^{m_i} E_{i,j} - \sum_{j=1}^{\ell} (m_i - 1) \times exe(c_j) \\
&= \sum_{j=1}^{m_i} E_{i,j} - (m_i - 1) \times L_i \geq C_i
\end{aligned}
$$

Therefore, the contradiction is reached. □

In order to restrict the reservation-budgets to reasonable sizes, the following simple lemma argues that providing a reservation server with $E_{i,j*} < L_i$ is never useful.

4.2.2 Lemma. *If there exists a τ_{i,j^*} with $E_{i,j^*} < L_i$, such a reservation $\tau_{i,j}$ has a negative impact on the condition $\sum_{j=1}^{m_i} E_{i,j} - (C_i + L_i(m_i - 1))$.*

Proof. This comes from simple arithmetics. If so, removing the reservation τ_{i,j^*} leads to $m_i - 1$ reservation servers with better reservations due to $\sum_{j=1}^{m_i} E_{i,j} - (C_i + L_i(m_i - 1)) < (\sum_{j=1}^{m_i} E_{i,j}) - E_{i,j^*} - (C_i + L_i(m_i - 2))$. □

Therefore, in the remainder of this thesis the property in Lemma 4.2.2, i.e., $E_{i,j} \geq L_i \ \forall j$, whenever the reservation condition in Theorem 4.2.1 is used, is implicitly considered.
For further analysis let $E_{i,j} \stackrel{\text{def}}{=} \gamma_{i,j} \cdot L_i$, with $1 < \gamma_{i,j} \leq \frac{D_i}{L_i}$. Subsequently, any reservation system that satisfies the conditions

$$L_i \cdot (m_i - 1) + C_i \leq \sum_{j=1}^{m_i} \gamma_{i,j} \cdot L_i \tag{4.1a}$$

$$\gamma_{i,j} \cdot L_i \leq D_i \ \forall 1 \leq j \leq m_i \tag{4.1b}$$

$$\gamma_{i,j} > 1 \ \forall 1 \leq j \leq m_i \tag{4.1c}$$

satisfies the reservation condition in Theorem 4.2.1 and is thus feasible.

4.2.3 Definition. A reservation system $\gamma_i \stackrel{\text{def}}{=} (\gamma_{i,1}, \gamma_{i,2}, .., \gamma_{i,m_i})$ for a DAG task (C_i, L_i, D_i, T_i) is called feasible if and only if it satisfies the conditions in Eq. (4.1).

Subsequently, the following properties from the Eq. (4.1) can be observed. First, the equation depends only on the task parameters C_i, L_i, D_i as well as the number of reservation servers m_i. Second, as m_i increases, the sum of execution requirements $\sum_{j=1}^{m_i} \gamma_{i,j} \cdot L_i$ also increases. Therefore, there are many possible feasible designs of reservation servers for the same DAG task. The optimal reservation assignment depends on the analysis used to verify schedulability of the reservation servers.
In order to create feasible concrete reservation systems, two different types of algorithms for computing reservations are proposed in this thesis, namely *decoupled* and *coupled* algorithms.

4.2.4 Definition. A *reservation-generating algorithm* that computes a system of reservations for a given sporadic arbitrary-deadline DAG task set is denoted *decoupled*, if the scheduling of the resulting reservations is not considered in the process.

4.2.5 Definition. A *reservation-generating algorithm* that computes a system of reservations for a given sporadic arbitrary-deadline DAG task set is denoted *coupled*, if the scheduling of the resulting reservations is explicitly considered in the process.

5 Algorithms for the creation of Reservations

On the basis of the sufficient design constraints for the generation of reservation systems as introduced in the previous Chapter 4 *Reservation-Based Federated Scheduling*, algorithms that compute concrete feasible reservation systems can be designed.
To that end, this chapter covers the design and analysis of *coupled* and *decoupled* algorithms. As previously mentioned, a *decoupled* algorithm computes the reservation system settings i.e., number of reservations and the budget-sizes irrespective of the consecutive scheduling problem thereof.
In contrast to that, a *coupled* algorithm computes the number of reservations and budget-sizes under consideration of the scheduling problem thereof. Ideally, the increased algorithmic complexity of *coupled* algorithms yields improved schedulability analytically and practically.
The remainder of this chapter is organized as follows. In the first Section 5.1 *Decoupled Algorithms for Calculating Reservations*, the decoupled algorithms are proposed and analyzed with respect to their theoretical properties.
In the following Section 5.2 *Coupled Algorithms for Calculating Reservations*, the developed coupled algorithms are presented respectively.

5.1 Decoupled Algorithms for Calculating Reservations

At first, two decoupled algorithms for the calculation of reservations for DAG tasks are presented. Both these algorithms generalize the federated scheduling core allocation principle and each has its advantages and disadvantages. Furthermore, the second decoupled algorithm *R-EQUAL* will be used to prove, that reservation-based federated scheduling admits a constant speedup factor for sporadic arbitrary-deadline DAG task sets.

5.1.1 R-MIN: Small Number of Reservation Servers

One approach to compute concrete reservations is to enforce *equal-reservations* such that the conditions for feasible reservation systems can be solved analytically to

$$L_i \cdot (m_i - 1) + C_i \leq \gamma_i \cdot m_i \cdot L_i, \tag{5.1}$$

which yields $\frac{C_i - L_i}{L_i \cdot (\gamma_i - 1)} \leq m_i$.
Note that the notation of $\gamma_{i,j}$ changed to γ_i due to the equality of all m_i reservations.
Since the number of reservation servers must be a natural number, it is certain that

$$m_i \stackrel{\text{def}}{=} \left\lceil \frac{C_i - L_i}{L_i \cdot (\gamma_i - 1)} \right\rceil \tag{5.2}$$

is the smallest number of reservation servers required under the equal-reservation constraint. Additionally, due to the fact that there are multiple settings of γ_i that yield the same minimal number of reservation servers, γ_i is defined as follows

$$\gamma_i = \min\{\gamma_i \mid \gamma_i \text{ satisfies Eq. (5.2)}\}. \tag{5.3}$$

5.1.1 Observation. The left-hand side of the above equation (5.2) is minimized if γ_i is maximized, i.e., $m_i = \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$, and the corresponding smallest γ_i that achieves an equally minimal number of reservations is given by $1 + \frac{C_i - L_i}{m_i L_i}$.

This observation motivates the idea behind the *R-MIN* algorithm (c.f. Alg. 5.1). Intuitively, *R-MIN* classifies tasks into light and heavy tasks based on whether a task requires to be serviced by more than one reservation (using the reservation settings from observation 5.1.1) or not. Further, the algorithm assigns each heavy task the least number of reservation servers possible using the equal-reservation constraint.
Therefore, the *R-MIN* has the following properties:

5.1.2 Theorem. *The* R-MIN *algorithm generates a minimal set of sporadic arbitrary-deadline reservations for a given sporadic arbitrary-deadline DAG task set that provide sufficient resources to service their respective DAG tasks.*

Require: Sporadic arbitrary-deadline DAG task set
Ensure: Set of reservations, that can provably service all DAG tasks sufficiently
1: $\mathbf{T}_{\text{HEAVY}} \leftarrow \{\tau_i \in \mathbf{T} \mid C_i > D_i\}$
2: $\mathbf{T}_{\text{LIGHT}} \leftarrow \{\tau_i \in \mathbf{T} \mid C_i \leq D_i\}$
3: $\mathbf{T} \leftarrow \mathbf{T}_{\text{LIGHT}}$
4: **for** each task τ_i in $\mathbf{T}_{\text{HEAVY}}$ **do**
5: $\quad m_i \leftarrow \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$
6: $\quad$ **for** $\ell \leftarrow 1$ to m_i **do**
7: $\quad\quad \tau_{i,\ell} \leftarrow \left(\left(1 + \frac{C_i - L_i}{m_i L_i}\right) \cdot L_i, D_i, T_i \right)$
8: $\quad\quad \mathbf{T} \leftarrow \mathbf{T} \cup \{\tau_{i,\ell}\}$
9: $\quad$ **end for**
10: **end for**
11: **return** $\mathbf{T}$

Algorithmus 5.1: For each DAG task, the *R-MIN* algorithm computes a minimal set of equal reservation servers, that can sufficiently service their designated DAG tasks.

Proof. This is based on the above discussions. □

The first algorithm, *R-MIN*, provides the smallest number of reservation servers m_i to each DAG job. As mentioned earlier, Eq. (4.1) indicates that the total reservation for the DAG task increases as m_i increases; therefore, this strategy also provides the smallest reservation to each job. This reservation scheme is good if the schedulability analysis only considers the cumulative demands of the generated reservations. On the flip side, this strategy is the closest to federated scheduling and inherits its disadvantages. First, each task's reservation is calculated completely independently from all the other jobs in the system. Second, each reservation may be "tight", i.e., $E_{i,j}$ may be very close to D_i. Both these properties may make it difficult to ensure that the task set is schedulable even though individual tasks require their minimum reservations. This strategy is unlikely to provide a constant speedup bound for the same reason that federated scheduling does not admit a constant speedup bound for constrained-deadline tasks. Therefore a different approach is developed and presented in the next section. That strategy potentially increases the total reservation for each task, but each reservation server may have a smaller execution requirement compared to *R-MIN*.

5.1.2 R-EQUAL: Equal "Slack"

As mentioned in the previous section, one of the reasons *R-MIN* may generate unschedulable reservations is that each job calculates its own γ-value which can be as large as possible for that job. Jobs that have large γ-values have little slack (meaning their reservation $E_{i,j} = \gamma_i L_i$ is very close to D_i). To that end, the following algorithm that uses a single common γ-value across all jobs is presented. This algorithm, *R-EQUAL*, is shown in Alg. 5.2. Each DAG task is classified to be either a heavy or a light tasks, i.e., whether a DAG task requires more than one reservation (as generated by the algorithm) to be feasibly serviceable or not, based on a single common γ-value. Since the common γ-value must be valid for all DAG tasks, i.e., $L_i < \gamma_i L_i \leq D_i$, it must be that $1 < \gamma \leq \min_{\tau_i}\{\frac{D_i}{L_i}\}$. Using the largest such γ for a given DAG task set thus minimizes the number of reservation servers under equal γ-constraints.

It will show that *R-EQUAL* provides good analytical results and allows to prove constant speedup bounds. However, it does not lead to heuristically *good* reservations as will be seen in the evaluations. In order to prove theoretical properties of *R-EQUAL* e.g., speedup factors, the over-provisioning of the reservation system with respect to the worst-case execution time of the serviced DAG task must be quantified. To that end, the following theorem proves, that the inflation of any reservation system that generates equal-reservations for each DAG task is parametrically upper bounded by γ.

Require: Sporadic arbitrary-deadline DAG task set and γ-value
Ensure: Set of reservations, that can provably service all DAG tasks sufficiently
1: $\mathbf{T}_{\text{HEAVY}} \leftarrow \{\tau_i \in \mathbf{T} \mid C_i > \gamma \cdot L_i\}$
2: $\mathbf{T}_{\text{LIGHT}} \leftarrow \{\tau_i \in \mathbf{T} \mid C_i \leq \gamma \cdot L_i\}$
3: $\mathbf{T} \leftarrow \mathbf{T}_{\text{LIGHT}}$
4: **for** each task τ_i in $\mathbf{T}_{\text{HEAVY}}$ **do**
5: $\quad m_i \leftarrow \left\lceil \frac{C_i - L_i}{L_i(\gamma-1)} \right\rceil$
6: $\quad$ **for** $\ell \leftarrow 1$ to m_i **do**
7: $\quad\quad \tau_{i,\ell} \leftarrow (\gamma L_i, D_i, T_i)$
8: $\quad\quad \mathbf{T} \leftarrow \mathbf{T} \cup \{\tau_{i,\ell}\}$
9: $\quad$ **end for**
10: **end for**
11: **return T**

Algorithmus 5.2: For each DAG task, the *R-EQUAL* algorithm computes a set of equal reservation servers based on a common γ-value, that can sufficiently service their designated DAG tasks.

5.1.3 Theorem. *Suppose that $\gamma > 1$ is given and there are exactly m_i reservation servers for task τ_i where $m_i \stackrel{def}{=} \left\lceil \frac{C_i - L_i}{L_i(\gamma-1)} \right\rceil$ with $m_i \geq 2$. If $C'_i = \sum_{j=1}^{m_i} E_{i,j} = C_i + (m_i - 1) \cdot L_i$ and $\gamma = \frac{\sum_{j=1}^{m_i} E_{i,j}/L_i}{m_i}$ then $C'_i \leq (1 + \frac{1}{\gamma-1}) \cdot C_i$.*

Proof. By the assumption $L_i > 0$ and $\gamma > 1$, the setting of $m_i = \left\lceil \frac{C_i - L_i}{L_i(\gamma-1)} \right\rceil$ implies that

$$m_i - 1 < \frac{C_i - L_i}{L_i(\gamma - 1)} \leq m_i \tag{5.4}$$

$$\Rightarrow \; (m_i - 1)(\gamma - 1)L_i < C_i - L_i \leq m_i L_i \gamma - m_i L_i \tag{5.5}$$

$$\Rightarrow \; C_i + (m_i - 1)L_i \leq m_i \gamma L_i < C_i + (m_i - 2)L_i + \gamma L_i \tag{5.6}$$

The condition $m_i \gamma L_i < C_i + (m_i - 2)L_i + \gamma L_i$ in Eq. (5.5) implies $(m_i - 1)L_i < \frac{C_i + (m_i-2)L_i}{\gamma}$ because $\gamma > 0$. Since $C'_i = \sum_{j=1}^{m_i} E_{i,j} = C_i + (m_i - 1)L_i$ by definition, yields

$$\begin{aligned} C'_i < \; & C_i + \frac{C_i + (m_i - 2)L_i}{\gamma} \\ <_1 \; & \frac{C_i(\gamma + 1)}{\gamma} + \frac{(m_i - 2)}{\gamma} \left(\frac{C_i}{(m_i - 1)(\gamma - 1)} \right) \\ \leq_2 \; & C_i \left(\frac{\gamma + 1}{\gamma} + \frac{1}{\gamma^2 - \gamma} \right) \\ = \; & \left(1 + \frac{1}{\gamma - 1} \right) \cdot C_i \end{aligned}$$

where $<_1$ is due to $L_i < \frac{C_i}{(m_i-1)(\gamma-1)+1} < \frac{C_i}{(m_i-1)(\gamma-1)}$ by reorganizing the condition in Eq. (5.4) and $\leq_2$ is due to $m_i \geq 2$ and $\frac{m_i - 2}{m_i - 1} \leq 1$. □

This property can thus be used to directly relate the scheduling of sporadic arbitrary-deadline DAG task sets using the proposed reservation-based approach to any other DAG task scheduling algorithm, that quantifies schedulability based on the worst-case execution time.

5.1.3 Partitioned Scheduling for Reservation Servers

Based on the above results, this section explains how the partitioned scheduling of reservation systems, that are created by using the *R-EQUAL* algorithm can be used to prove a constant speedup factor of reservation-based federated scheduling with respect to an optimal DAG task scheduling algorithm.
To that end, this section analyzes the theoretical properties of reservation-based federated scheduling, when the reservation servers are scheduled under multiprocessor partitioned scheduling.
In the domain of ordinary sporadic real-time tasks under partitioned scheduling, extensive speedup factor analyses have been proposed. For scheduling ordinary sporadic real-time tasks, Baruah and Fisher developed a greedy heuristic, namely Deadline-Monotonic-Partitioning (DMP) [9, 10].

5.1.4 Definition. *Deadline-Monotonic Partitioning (DMP)*: The tasks are considered in a non-decreasing order of their relative deadlines. When a task τ_k is considered in this order, if task τ_k and the other *previously assigned* tasks on a processor can be feasibly scheduled under the specified scheduling strategy, then task τ_k is assigned to one of such processors (if there are more than one). Otherwise, task τ_k is assigned to a newly allocated processor.

Chen et al. further analyzed this DMP strategy under different fitting strategies e.g., first-fit, best-fit, worst-fit or arbitrary-fit [21,23,24]. It could be proved, that this strategy allows for a speedup factor of 3 irrespective of the concrete fitting strategies. It is to emphasize, that DMP only specifies the order of the tasks to be considered and partitioned.
More importantly, the underlying uniprocessor scheduling strategy after the task partitioning can be arbitrary, e.g., EDF or deadline-monotonic fixed-priority scheduling. Using these insights, it will be proved that DMP in conjunction with reservation-based federated scheduling can yield constant speedup bounds.
Due to the DMP strategy, all tasks or reservation servers respectively are indexed such that $D_i \leq D_j$ if $i \leq j$. Before the competitiveness of reservation-based federated scheduling under DMP is proved, the schedulability test used for the following partitioned scheduling of sporadic arbitrary-deadline task sets is briefly restated.
Suppose that the partitioning algorithm attempts to assign task τ_k to a processor P_m and hp_k^m is the set of higher-priority tasks that are already assigned on processor P_m.

With reference to the sufficient schedulability analyses of ordinary sporadic arbitrary-deadline task sets presented in Section 2.2.2 *Uniprocessor Scheduling*, the schedulability condition of the reservation servers is given by

$$E_k + \sum_{\tau_i \in hp_k^m} \left(1 + \frac{D_k}{T_i}\right) E_i \leq D_k \quad \text{and} \tag{5.7a}$$

$$U_k + \sum_{\tau_i \in hp_k^m} U_i \leq 1. \tag{5.7b}$$

With the above analysis, the constant speedup factor of reservation-based federated scheduling under DMP is proved.

5.1.5 Theorem. *A system of arbitrary-deadline DAG tasks scheduled by* ***reservation-based federated scheduling*** *under DMP, in which each processor uses deadline-monotonic fixed-priority scheduling for the reservation servers, admits a constant speedup factor of* $3 + 2\sqrt{2}$ *with respect to any optimal scheduler by setting* γ *to* $1 + \sqrt{2}$*.*

Proof. This is proved by adopting *R-EQUAL* with a setting of $\gamma = 1 + \sqrt{2}$ and distinguishing two separate cases. The first case covers all DAG task sets in which for at least one task $\gamma L_i > D_i$ holds, i.e., $(1 + \sqrt{2}) \cdot L_i > D_i$.

In this case, the speedup factor for this task set is $1 + \sqrt{2}$. Hence, in the following second case, $\gamma L_i \leq D_i$ holds for all DAG tasks in the task set and due to the DMP strategy the tasks are indexed such that $D_i \leq D_j$ if $i \leq j$.

Suppose that $\tau_{k,\ell}$ is a reservation task that is not able to be partitioned to any of the given m processors, where $1 \leq \ell \leq m_k$.

Additionally, let $\mathbf{m}_1$ be the set of processors in which Eq. (5.7a) fails and let $\mathbf{m}_2$ be the set of processors in which Eq. (5.7a) succeeds but Eq. (5.7b) fails.

Since $\tau_{k,\ell}$ cannot be assigned on any of the m processors, it must be that $|\mathbf{m}_1| + |\mathbf{m}_2| = m$. Further, the violation of Eq. (5.7a) yields

$$\begin{aligned} &|\mathbf{m}_1| E_{k,\ell} + \sum_{P_m \in \mathbf{m}_1} \sum_{\tau_{i,j} \in hp_k^m} \left(1 + \frac{D_k}{T_i}\right) E_{i,j} > |\mathbf{m}_1| D_k \\ \Rightarrow &|\mathbf{m}_1| \frac{E_{k,\ell}}{D_k} + \sum_{P_m \in \mathbf{m}_1} \sum_{\tau_{i,j} \in hp_k^m} \left(\frac{E_{i,j}}{D_k} + \frac{E_{i,j}}{T_i}\right) > |\mathbf{m}_1|. \end{aligned} \tag{5.8}$$

Conversely, the violation of Eq. (5.7b), yields that

$$|\mathbf{m}_2| \frac{E_{k,\ell}}{T_k} + \sum_{P_m \in \mathbf{m}_2} \sum_{\tau_{i,j} \in hp_k^m} \frac{E_{i,j}}{T_{i,j}} > |\mathbf{m}_2|. \tag{5.9}$$

Due to Eqs. (5.8) and (5.9), the definition $\sum_{j=1}^{m_i} E_{i,j} = C'_i$, and the fact that $\tau_{i,j}$ is assigned either on a processor of $\mathbf{m}_1$ or on a processor of $\mathbf{m}_2$ if $\tau_{i,j}$ is assigned successfully prior to $\tau_{k,\ell}$, is must be that

$$m\frac{E_{k,\ell}}{\min\{T_k, D_k\}} + \sum_{i=1}^{k}\left(\frac{C'_i}{T_i} + \frac{C'_i}{D_k}\right) > m. \tag{5.10}$$

In that case γ satisfies the constraints $L_i < \gamma L_i \leq D_i$ for all tasks by definition, thus represents a feasible setting. Additionally, by the reservation-budget setting of the algorithm ($E_{k,\ell} = \gamma L_k$) and by Theorem 5.1.3, the above inequality can be upper bounded by

$$m\frac{\gamma L_k}{\min\{T_k, D_k\}} + \sum_{i=1}^{k}\left(\frac{(1+\frac{1}{\gamma-1})C_i}{T_i} + \frac{(1+\frac{1}{\gamma-1})C_i}{D_k}\right) > m. \tag{5.11}$$

Let $\alpha \stackrel{\text{def}}{=} \max\left\{\frac{L_k}{\min\{T_k,D_k\}}, \sum_{i=1}^{k}\frac{C_i}{mT_i}, \sum_{i=1}^{k}\frac{C_i}{mD_k}\right\}$, denote a necessary condition for schedulability. Since $D_i \leq D_k$ for $i = 1, 2, \ldots, k$ under DMP, the task system is not schedulable at any speed strictly less than α. Therefore[1]

$$\gamma\alpha + 2\left(1 + \frac{1}{\gamma-1}\right)\alpha > 1 \tag{5.12}$$

$$\Rightarrow \qquad \alpha > \frac{1}{\gamma + \frac{2\gamma}{\gamma-1}} = \frac{\gamma-1}{\gamma^2+\gamma} = \frac{\sqrt{2}}{4+3\sqrt{2}} = \frac{1}{3+2\sqrt{2}} \tag{5.13}$$

In conclusion, the speedup factor is at most $3 + 2\sqrt{2}$. □

By using any setting of γ different from $1 + \sqrt{2}$, the following parametric speedup bound can be given.

5.1.6 corollary. *Based on Theorem 5.1.5, the speedup factor of* ***reservation-based federated scheduling*** *under DMP under a parameter $\gamma > 1$ in the* R-EQUAL *algorithm is*

$$\frac{\gamma^2+\gamma}{\gamma-1}. \tag{5.14}$$

Proof. This follows from the proof in Theorem 5.1.5. □

5.1.7 Theorem. *A system of arbitrary-deadline DAG tasks scheduled by* ***reservation-based federated scheduling*** *under DMP, in which each processor uses EDF for the reservation servers, admits a constant speedup factor of $3+2\sqrt{2}$ with respect to any optimal scheduler by setting γ to $1 + \sqrt{2}$.*

Proof. Since EDF is an optimal uniprocessor scheduling policy with respect to schedulability, the task partitioning algorithm and analysis in Theorem 5.1.5 yields the result directly. □

[1] The setting of γ as $1 + \sqrt{2}$ is in fact to maximize $\frac{\gamma-1}{\gamma^2+\gamma}$.

5.1.4 Global Scheduling for Reservation Servers

In contrast to partitioned scheduling, global scheduling schemes facilitate a single ready queue for all tasks and allow for arbitrary task migrations. Analogously to the problem of calculating reservation servers for partitioned scheduling schemes, there is no globally optimal greedy algorithm to generate reservations for a given DAG task set for global scheduling.

Instead, the optimal reservation assignment depends on the analysis that is used to verify the schedulability. That is, for example if the schedulability analysis only considers the cumulative demands of the generated reservations then the individual settings of the reservations become irrelevant and should thus be chosen such that system of minimal reservation demands, e.g., $C_i + (m_i - 1)L_i = C_i'$ are satisfied as tightly as possible (*R-MIN*). Otherwise, optimization techniques that are suitable for the used schedulability analysis e.g., linear- or quadratic-programming in conjunction with the constraints for feasible reservation systems may be used to calculate reservation servers.

There are extensive results available that analyze the schedulability of global scheduling algorithms for constrained-deadline ordinary sporadic task systems [2, 7, 14, 36] and arbitrary-deadline ordinary sporadic task systems [11, 12, 37, 56].

In general, any of these schedulability tests can be applied for validating the underlying global scheduling algorithm of the reservation servers and thus sporadic DAG task sets. Furthermore, it can be proved that the speedup factor of global deadline-monotonic scheduling for sporadic arbitrary-deadline DAG task sets is the same as in partitioned deadline-monotonic scheduling, namely $3 + 2\sqrt{2}$.

For further analysis, the following definition is introduced in accordance to the definition by Baruah [6].

5.1.8 Definition (load function [6]). The LOAD_i function denotes the highest demand rate, that a task τ_i may exhibit i.e., $\text{LOAD}_i \stackrel{\text{def}}{=} \max_{t>0} \frac{\text{DBF}(\tau_i,t)}{t}$.

It is obvious that $\text{LOAD}_i \leq 1$ is a necessary condition for uniprocessor schedulability i.e., if the task τ_i may demand more computation time than can be provided by the processor, namely $\text{LOAD}_i > 1$ then there exists at least one t' such that $\text{DBF}(\tau_i, t') > t'$.

By the definition of the demand-bound function, this implies that there exists at least one schedule in which task τ_i misses its relative deadline. Using the sufficient schedulability analysis proposed by Chen et al. [26] for scheduling sporadic arbitrary-deadline tasks sets, the following more pessimistic theorem can be derived.

5.1.9 Theorem. *A sporadic arbitrary-deadline task set consistent of tasks $\tau_1, \tau_2, .., \tau_k$ is schedulable by global deadline-monotonic (DM) scheduling on $m > 0$ homogeneous multiprocessors if*

$$2 \cdot \sum_{i=1}^{k} \text{LOAD}(\tau_i) \leq m - (m-1) \cdot \delta_k^{max}.$$

On the basis of this sufficient condition, the following theorem can be proved.

5.1.10 Theorem. *A system of arbitrary-deadline DAG tasks scheduled by* ***reservation-based federated scheduling*** *using global deadline-monotonic scheduling to schedule the generated reservation servers admits a constant speedup factor of at most* $3 + 2\sqrt{2}$ *with respect to any optimal scheduler.*

Proof. By adopting the *R-EQUAL* algorithm Alg. 5.2 with a setting of $\gamma = 1 + \sqrt{2}$, the following two cases can be distinguished. In the first case, all DAG task sets in which for at least one task $\gamma L_i > D_i$ holds, i.e., $(1 + \sqrt{2}) \cdot L_i > D_i$ are covered. In that case, the speedup factor for this task set is $1 + \sqrt{2}$.

In the second case, subsequently $\gamma L_i \leq D_i$ holds for all DAG tasks in the task set. Further, let $\mathbf{T}$ denote the set of sporadic arbitrary-deadline DAG tasks and $\mathbf{T}'$ denote the set of associated reservation servers, that are generated by the *R-EQUAL* algorithm.

Let $\mathbf{T}'$ fail to suffice theorem Thm. 5.1.10, then

$$\begin{aligned} 2 \sum_{\tau_i \in \mathbf{T}'} \text{LOAD}_i &> m - (m-1) \cdot \delta_k^{max} \\ \Rightarrow 2 \sum_{\tau_i \in \mathbf{T}'} \frac{\text{LOAD}_i}{m} &+ \frac{(m-1)}{m} \cdot \delta_k^{max} > 1 \end{aligned} \tag{5.15}$$

must hold. Due to the policy of the *R-EQUAL* algorithm i.e., $E_i = \gamma L_i$ and by theorem Thm. 5.1.3 it follows that

$$2 \cdot (1 + \frac{1}{\gamma - 1}) \sum_{\tau_i \in \mathbf{T}} \frac{\text{LOAD}_i}{m} + \gamma \max_{\tau_i \in \mathbf{T}} \left\{ \frac{L_i}{\min\{D_i, T_i\}} \right\} \cdot (1 - \frac{1}{m}) > 1. \tag{5.16}$$

Let $\alpha \stackrel{\text{def}}{=} \max \left\{ \sum_{\tau_i \in \mathbf{T}} \frac{\text{LOAD}_i}{m}, \max_{\tau_i \in \mathbf{T}} \{ \frac{L_i}{\min\{D_i, T_i\}} \} \right\} \leq 1$ denote a necessary condition for schedulability i.e., either $\sum_{\tau_i \in \mathbf{T}} \text{LOAD}_i > m$ or the existence of at least one DAG task such that $L_i > D_i$ or $L_i > T_i$ implies unschedulability by any algorithm. Therefore,

$$3 + 2\sqrt{2} \geq \frac{\gamma^2 + \gamma}{\gamma - 1} \geq \cdot (1 + \frac{1}{\gamma - 1}) + \gamma \cdot (1 - \frac{1}{m}) > \frac{1}{\alpha}. \tag{5.17}$$

In conclusion, if a sporadic arbitrary-deadline DAG task set is deemed unschedulable using reservation-based federated scheduling in conjunction with global deadline-monotonic scheduling according to the presented sufficient condition, then this DAG task set remains unschedulable by any scheduling algorithm when slowing down all m processors by $\frac{1}{3+2\sqrt{2}}$. □

5.1.5 Dominance over Federated Scheduling

Lastly, it can be shown that reservation-based scheduling is at least as powerful as federated scheduling in the sense that every instance of federated scheduling can be transformed into an instance of reservation-based federated-scheduling using less resources. Since the original version of federated scheduling does not consider arbitrary deadline DAG task sets, the following analysis only considers constrained-deadline task sets. Moreover, the extension of federated scheduling to include arbitrary-deadline DAG task sets as proposed by Baruah [4] is not considered in the following dominance analysis due to the fact, that it considers the concrete DAG structure instead of a purely parametric model.

5.1.11 Lemma. *Reservation-based federated scheduling of constrained-deadline DAG tasks utilizes no more resources than federated scheduling does.*

Proof. In order to prove this theorem, federated scheduling is interpreted as reservation-based federated scheduling, that uses $\left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$ many processors (reservations) of budget T_i. Then, the amount of saved resources is given by

$$\left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil \cdot T_i - \left(\left(\left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil - 1 \right) L_i + C_i \right) \geq 0, \tag{5.18}$$

□

5.1.12 Theorem. *Reservation-based federated scheduling is at least as powerful as federated scheduling in scheduling sporadic, constrained-deadline DAG task sets.*

Proof. Let a sporadic constrained-deadline DAG task set $\mathbf{T}$ be schedulable by federated scheduling i.e., all heavy tasks are assigned to sufficient processors exclusively and all light tasks can be feasibly partitioned according to some partitioned scheduling algorithm for said task model. Since all light tasks in reservation-based federated scheduling use the same amount of resources as in federated scheduling any feasible partition implies a feasible partition for light tasks in reservation-based federated scheduling. Moreover, due to lemma 5.1.11 all heavy tasks in reservation-based federated scheduling use no more resources than heavy tasks in federated scheduling. Therefore, the partitioning of each reservation server to a processor exclusively is feasible, which concludes the proof. □

5.2 Coupled Algorithms for Calculating Reservations

In contrast to the previously presented *decoupled* approaches, this section describes a family of algorithms that include the schedulability tests in the calculation of reservation servers which ideally improve schedulability.

5.2.1 Split-On-Fail Algorithm

One *coupled* approach is presented in the *Split-On-Fail* (SOF) algorithm Alg. 5.4. The *SOF* algorithm attempts to partition all generated reservation servers according to a scheduling test for arbitrary-deadline task systems by preference i.e., worst-fit, first-fit, best-fit.
In order to reduce algorithmic complexity, the partitioning of each group i.e., set of reservation servers that belong to the same DAG task is done on the premise that all previously partitioned groups are already feasibly partitioned and their settings do not change. Additionally, light tasks are excluded from the adaptation process.
Whenever a reservation can not be partitioned, an additional reservation server is added to the group and the individual reservation-budgets are decreased appropriately.

5.2.1 Definition. A rule to generate equal reservation servers for a DAG task τ_i, is given by $E_i(\ell_i) = \frac{C_i}{\ell_i} + (1 - \frac{1}{\ell_i}) \cdot L_i, \ell_i \geq m_i \in \mathbb{N}$.

By this imposed structure, the set of all possible resevation budgets is reduced from a theoretical feasible interval $L_i < E_i \leq D_i$ to the following set of reservations

$$\{E_i \mid E_i = \frac{C_i}{\ell_i} + (1 - \frac{1}{\ell_i}) \cdot L_i, \ \forall \ell_i \geq 1, m_i \in \mathbb{N}\} \cap (L_i, D_i]. \tag{5.19}$$

Moreover, in partitioned scheduling algorithms that use a priority policy such that all reservation servers that serve the same DAG task i.e., belong to the same group are assigned the same priority, yield *non-equal* reservation servers effectively.
For example, rate-monotonic, deadline-monotonic or earliest-deadline-first scheduling are such priority policies.
Due to the fact that if multiple reservations with the same priority are partitioned to the same processor, this behaves as if there was only one reservation server with the individual reservation budgets accumulated. It is to emphasize that since the system of reservation servers is determined upon partitioning, the partitioning strategies i.e., *best-fit*, *worst-fit* and *first-fit* may result in different reservation-budgets effectively.
Without enforcing a specific scheduling algorithm for the partitioned scheduling of the calculated reservation servers and by using the definitions introduced in Section 2.2.3 *Partitioned Scheduling*, the baseline version of the *Split-On-Fail* algorithm Alg. 5.3 is designed.
In the first stage of the *SOF* algorithm, reservation servers are created for each heavy DAG task according to either the *R-MIN* or the *R-EQUAL* algorithm. By that design, the performance of the *SOF* algorithm is lower bounded by the proposed *decoupled* algorithms and automatically exhibits the same theoretical properties.
In the subsequent algorithmic stage, all sequential sporadic arbitrary-deadline tasks are partitioned successively according to the *DMP* ordering i.e., $D_i \leq D_j$ for all $i < j$, under the premise that all prior tasks have already been partitioned and are feasibly schedulable according to the used suffcient schedulability analysis.

Require: An arbitrary-deadline DAG task set $\mathbf{T}$, number of processors m, boundaries $b_1, b_2, .., b_N$.
Ensure: Feasible partition and reservations, that can service $\mathbf{T}$ provably (if one could be found).

1: $\mathbf{T_H}, \mathbf{T_L} \leftarrow$ R-MIN($\mathbf{T}$) or R-EQUAL($\mathbf{T}$)
2: re-index $\mathbf{T_H} \cup \mathbf{T_L}$ such that $D_i \leq D_j$ for $i < j$
3: **for each** $\tau_{i,j}$ in $\mathbf{T_H} \cup \mathbf{T_L}$ **do**
4: **if** $\tau_{i,j}$ in $\mathbf{T_L}$ **then**
5: **if** partition by preference failed **then**
6: **return** Partition Failure
7: **end if**
8: **else**
9: Failure $\leftarrow$ True
10: **while** ℓ_i is no more than b_i **do**
11: **if** partition by preference failed **then**
12: revoke all already partitioned $\tau_{i,1}, \tau_{i,2}, .., \tau_{i,j}$ belonging to DAG task τ_i
13: $\ell_i \leftarrow \ell_i + 1$ ▷ increment the amount of reservations and re-try
14: **for each** j in $\{1, 2, .., \ell_i\}$ **do**
15: $E_{i,j} \leftarrow \frac{C_i}{\ell_i} + (1 - \frac{1}{\ell_i}) \cdot L_i$ ▷ compute evolved reservation budgets
16: **end for**
17: continue
18: **else**
19: Partition $\tau_{i,j}$ to processor as chosen by preference
20: Failure $\leftarrow$ False
21: break
22: **end if**
23: **end while**
24: **if** Failure is True **then**
25: **return** Partition Failure
26: **end if**
27: **end if**
28: **end for**
29: **return** Partition and Reservations

Algorithmus 5.3: The *Split-On-Fail* (SOF) algorithm yields feasible partitions and a feasible system of reservation servers.

Furthermore, if a reservation server $\tau_{k,\ell}$ can not be partitioned then all previously assigned reservation servers that serve the same DAG task i.e., $\tau_{k,1}, \tau_{k,2}, .., \tau_{k,\ell-1}$ are dispatched from their assigned processors. In such a case, the number of reservation servers to service DAG task τ_k is incremented and the associated reservation-budgets are decreased accordingly.

The above procedure is repeated until either a feasible partition is found for all reservation servers in the group or if the associated bound b_k is exceeded.

Due to the fact, that the complexity of the proposed *Split-On-Fail* algorithm is largely impacted by the number of iterations, which also impacts the performance of the algo-

Require: An arbitrary-deadline DAG task set $\mathbf{T}$, number of processors m, boundaries $b_1, b_2, .., b_N$.
Ensure: Feasible partition and reservations, that can service $\mathbf{T}$ provably (if one could be found).

1: $\mathbf{T_H}, \mathbf{T_L} \leftarrow$ R-MIN($\mathbf{T}$) or R-EQUAL($\mathbf{T}$)
2: re-index $\mathbf{T_H} \cup \mathbf{T_L}$ such that $D_i \leq D_j$ for $i < j$
3: **for each** $\tau_{i,j}$ in $\mathbf{T_H} \cup \mathbf{T_L}$ **do**
4: **if** $\tau_{i,j}$ in $\mathbf{T_L}$ **then**
5: **if** partition by preference failed **then**
6: **return** Partition Failure
7: **end if**
8: **else**
9: Failure $\leftarrow$ True
10: **while** ℓ_i is no more than $\max\left\{\left\lceil\frac{C_i}{L_i}\right\rceil, \left\lceil\frac{C_i - L_i}{L_i(\gamma-1)}\right\rceil, b_i\right\}$ **do**
11: **if** partition by preference failed **then**
12: revoke all already partitioned $\tau_{i,1}, \tau_{i,2}, .., \tau_{i,j}$ belonging to DAG task τ_i
13: $\ell_i \leftarrow \ell_i + 1$ ▷ increment the amount of reservations and re-try
14: **for each** j in $\{1, 2, .., \ell_i\}$ **do**
15: $E_{i,j} \leftarrow \frac{C_i}{\ell_i} + (1 - \frac{1}{\ell_i}) \cdot L_i$ ▷ compute evolved reservation budgets
16: **end for**
17: continue
18: **else**
19: Partition $\tau_{i,j}$ to processor as chosen by preference
20: Failure $\leftarrow$ False
21: break
22: **end if**
23: **end while**
24: **if** Failure is True **then**
25: **return** Partition Failure
26: **end if**
27: **end if**
28: **end for**
29: **return** Partition and Reservations

Algorithmus 5.4: The SOF algorithm yields feasible partitions and associated reservations.

rithm i.e., improvements on schedulability, the choice of concrete boundaries are important design choices and thus subject to the following analyses and discussions.

In order to evaluate the *quality* of the algorithm for given boundaries $b_1, b_2, .., b_N$, the worst-case performance i.e., speedup factor analysis is used.

Despite the fact that speedup factor analyses do not reflect the quality of the schedulability test in the average case especially if the speedup factor is not tight, the choice of boundaries should preferably not worsen the analysis. Therefore, the intention of the following analyses is to

1. Identify boundaries that *do not* incur high algorithmic complexity.

2. Identify boundaries that exhibit favorable properties with respect to the speedup factor analysis.

In the remainder of this section, the following analyses are based on the same initial arguments as described in Section 5.1.3 *Partitioned Scheduling for Reservation Servers*. That is, if a reservation server $\tau_{k,\ell}$ fails to be partitioned by the *Split-On-Fail* algorithm, then the following Eq. (5.20)

$$\frac{E_{k,\ell}}{\min\{T_k, D_k\}} + \sum_{i=1}^{k} \frac{C'_i}{mT_i} + \frac{C'_i}{mD_k} \geq \frac{E_{k,\ell}}{\min\{T_k, D_k\}} + \sum_{i=1}^{k-1} \frac{C'_i}{mT_i} + \frac{C'_i}{mD_k} > 1 \tag{5.20}$$

holds. The over-approximation is due to the adding of the remaining reservation servers of the k-th group in order to allow for the comparison against a necessary condition for schedulability.
By definition of the *Split-On-Fail* algorithm, the cumulative worst-case execution time is given by $C_i + (m_i - 1) \cdot L_i$ and all reservation servers for a DAG task are equal in size. Therefore, the above equation can be reformulated to

$$\frac{E_k}{\min\{T_k, D_k\}} + \sum_{i=1}^{k} \frac{C_i}{mT_i} + \frac{C_i}{mD_k} + \sum_{i=1}^{k} \frac{(m_i - 1)L_i}{mT_i} + \frac{(m_i - 1)L_i}{mD_k} > 1$$

$$\Rightarrow (1 - \frac{1}{b_k}) \cdot \frac{L_k}{\min\{T_k, D_k\}} + \frac{C_k \cdot m}{m \cdot b_k \cdot \min\{T_k, D_k\}} + \sum_{i=1}^{k} \frac{C_i}{mT_i} + \frac{C_i}{mD_k} + \sum_{i=1}^{k} (b_i - 1) \cdot \left(\frac{L_i}{mT_i} + \frac{L_i}{mD_k} \right) > 1.$$

There can be two different approaches identified to upper bound Eq. (5.20).
More precisely,

$$... + \sum_{i=1}^{k} (b_i - 1) \cdot \left(\frac{L_i}{mT_i} + \frac{L_i}{mD_k} \right) \leq ... + \max_{i<k} (b_i - 1) \sum_{i=1}^{k} \left(\frac{L_i}{mT_i} + \frac{L_i}{mD_k} \right) \tag{5.21}$$

or

$$... + \sum_{i=1}^{k} (b_i - 1) \cdot \left(\frac{L_i}{mT_i} + \frac{L_i}{mD_k} \right) \leq ... + \left(\sum_{i=1}^{k} b_1 - 1 \right) \cdot \left(\sum_{i=1}^{k} \frac{L_i}{mT_i} + \frac{L_i}{mD_k} \right), \tag{5.22}$$

where the second case is due to the fact, that $\sum_{i=1}^{k} a_i \cdot b_i \leq \sum_{i=1}^{k} a_i \cdot \sum_{i=1}^{k} b_i \; \forall a_i \geq 0, b_i \geq 0$. In the first case illustrated in Eq. (5.21), an approach is to limit each boundary b_i to be no more than some constant multiple of the number of processors in the system i.e., $b_i = \mu m$, which yields the following theorem.

5.2.2 Theorem. *If the number of reservation servers for each DAG task is limited to be no more than some multiple of the processors i.e., $b_i \leq \mu m$ then the* Split-On-Fail *algorithm is guaranteed to find a feasible partition by increasing all processor speeds by $\mathcal{O}(m)$.*

Proof. If *Split-On-Fail* fails to partition the set of reservation servers and $\tau_{k,\ell}$ is the first reseration server, that can not be feasibly partitioned then by the imposed constraints $m_i \leq \mu m$ for all $i < k$ and $b_k = \mu m$. Thus, $3 - \frac{1}{\mu m} + \frac{1}{\mu} + 2 \cdot \mu m > \frac{1}{\alpha}$ holds, which concludes the proof. □

Using the second approach shown in Eq. (5.22), the idea is to limit $\sum_{i=1}^{k} b_i$. By similar arguments as before, the following equation

$$(3 - \frac{1}{b_k}) + \frac{m}{b_k} + 2 \cdot \sum_{i=1}^{k-1} b_i > \frac{1}{\alpha} \tag{5.23}$$

can be formulated.

5.2.3 Theorem. *If the number of reservation servers for each DAG task is limited to be no more than* $m \cdot \sum_{i=1}^{k-1} b_i$*, then the speedup factor of* Split-On-Fail *is at most* $\mathcal{O}(\sum_{i=1}^{k-1} b_i)$ *for a DAG task set of cardinality* k*.*

Proof. By the setting of $b_k = m \cdot \sum_{i=1}^{k-1} b_i$ it follows, that
$(3 - \frac{1}{m \cdot \sum_{i=1}^{k-1} b_i}) + \frac{1}{\sum_{i=1}^{k-1} b_i} + 2 \cdot \sum_{i=1}^{k-1} b_i > \frac{1}{\alpha}$ holds, which concludes the proof. □

Unfortunatly even if the first boundary $b_1 \stackrel{\text{def}}{=} c > 0$ is a constant, by forward substitution it yields that

$$\begin{aligned} b_1 &= c \\ b_2 &= m \cdot c \\ b_3 &= m \cdot (m \cdot c + c) \\ &\ldots \\ b_\ell &= \sum_{i=1}^{\ell-1} c \cdot m^i \end{aligned} \tag{5.24}$$

and thus potentially exponentially large bounds and algorithmic complexity.
Another approach is to enforce the condition that $L_k \cdot b_k \leq C_k$, since then the equation Eq. (5.20) can be simplified to

$$(2 - \frac{1}{b_k}) \cdot \frac{L_k}{\min\{T_k, D_k\}} + \sum_{i=1}^{k-1} \frac{C_i}{mT_i} + \frac{C_i}{mD_k} + \sum_{i=1}^{k-1} (b_i - 1) \cdot \left(\frac{L_i}{mT_i} + \frac{L_i}{mD_k} \right) > 1.$$

In conclusion, if the initialization of the reservation servers is done using the *R-EQUAL* algorithm, the following theorem can be stated.

5.2.4 Theorem. *Let the initial reservation servers be created using the* R-EQUAL *algorithm for some* $\gamma > 1$ *and let* $b_i > 1 \in \mathbb{N}$ *for every DAG task be given, then the speedup factor of* SOF *is at most*

$$\min\left\{\frac{\gamma^2+\gamma}{\gamma-1}, 4 - \min\left\{\frac{L_k}{C_k}, \frac{\gamma L_k - L_k}{C_k - L_k}, \frac{1}{b_k - 1}\right\} + 2 \cdot \max_{i<k}\left\{\max\left\{\frac{C_i}{L_i}, \frac{C_i - L_i}{\gamma L_i - L_i}, b_k - 1\right\}\right\}\right\}$$

Proof. If *SOF* can not find a feasible partition and reservations, then the initial reservation-budgets can not be partitioned feasibly either. Due to Theorem 5.1.7, this yields that the speedup factor is at most $3 + 2 \cdot \sqrt{2}$. Using the same arguments as in Theorem 5.1.5 and deadline-monotonic scheduling, yields

$$(2 - \frac{1}{c_k}) \cdot \frac{L_k}{\min\{T_k, D_k\}} + \sum_{i=1}^{k} \frac{C_i}{mT_i} + \frac{C_i}{mD_k} + \sum_{i=1}^{k}(c_i - 1) \cdot \left(\frac{L_i}{mT_i} + \frac{L_i}{mD_k}\right) > 1. \quad (5.25)$$

Where due to the definition of the boundaries $c_i = \max\left\{\left\lceil\frac{C_i}{L_i}\right\rceil, \left\lceil\frac{C_i - L_i}{L_i(\gamma-1)}\right\rceil, b_i\right\}$, it must be that $c_i \geq \left\lceil\frac{C_i}{L_i}\right\rceil \geq \frac{C_i}{L_i}$. Let

$$\alpha = \max\left\{\frac{L_k}{\min\{T_k, D_k\}}, \sum_{i=1}^{k}\frac{C_i}{mT_i}, \sum_{i=1}^{k}\frac{C_i}{mD_k}, \sum_{i=1}^{k}\frac{L_i}{mT_i}, \sum_{i=1}^{k}\frac{L_i}{mD_k}\right\} \quad (5.26)$$

denote a necessary condition for schedulability, then the following equation

$$4 - \frac{1}{c_k} + 2 \cdot \max_{i<k}\{c_i - 1\} > \frac{1}{\alpha} \quad (5.27)$$

holds. Since,

$$c_i - 1 = \max\left\{\left\lceil\frac{C_i}{L_i}\right\rceil, \left\lceil\frac{C_i - L_i}{L_i(\gamma-1)}\right\rceil, b_i\right\} - 1 \leq \max\left\{\frac{C_i}{L_i}, \frac{C_i - L_i}{L_i(\gamma-1)}, b_i - 1\right\} \quad (5.28)$$

holds, the equation can be reformulated to

$$4 - \min\left\{\frac{L_k}{C_k}, \frac{\gamma L_k - L_k}{C_k - L_k}, \frac{1}{b_k - 1}\right\} + 2 \cdot \max_{i<k}\left\{\max\left\{\frac{C_i}{L_i}, \frac{C_i - L_i}{\gamma L_i - L_i}, b_k - 1\right\}\right\} \quad (5.29)$$

which proves the theorem. □

In conclusion, the *Split-On-Fail* algorithm calculates reservation servers with a worst-case performance guarantee in polynomial-time complexity if the adopted schedulability test exhibits polynomial-time complexity.

6 Simulation and Prototyping of Reservation-Based Federated Scheduling

In the design of hard real-time scheduling algorithms the most important metric is each task's bounded response time, that is guaranteed to be no more than the task's relative deadline.

This guarantee is given by means of scheduling analyses i.e., formal proofs that for given premises and assumptions on the properties of the algorithm, the task- and system model are given. In theoretical models and analyses overheads and problems, that arise in real system implementations are neglected thus optimality statements are statements of the optimality of an algorithm with respect to the model under analysis.

In the paper *Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations* by Björn Brandenburg it was argued, that it is unfavorable to focus on complex and technically challenging to implement scheduling algorithms, if simpler scheduling algorithms perform sufficiently well in practical settings [18].

To that end, metrics like the *number of preemptions*, *number of migrations*, *runtime overheads*, *processor synchronization overheads*, *cache trashing* and the general complexity of implementation are relevant dimensions, that need to be considered when evaluating a scheduling algorithm. In order to evaluate *non-theoretical* properties of the proposed reservation-based federated scheduling algorithm, a prototype is realized within the real-time multiprocessor scheduling simulator *SimSo*[1], that is based on the Discrete-Event Simulator *SimPy*[2]. The simulation based prototyping is a compromise between a purely theoretical analysis and a performance evaluation based on an implementation in a real-time operating system e.g., Litmus-RT[3].

The remainder of this chapter is organized as follows. In the first Section 6.1 *Simulation environment*, the architecture, models and relevant internal workings of the SimSo framework are described.

In the following Section 6.2 *Design and implementation*, the implementation details of the reservation server, the DAG task model and the scheduler are discussed and detailed.

[1] http://projects.laas.fr/simso/
[2] https://simpy.readthedocs.io/en/latest/
[3] https://www.litmus-rt.org/

Lastly, in Section 7.4 *Summary* the achieved is summarized and a short conclusion is drawn.

6.1 Simulation environment

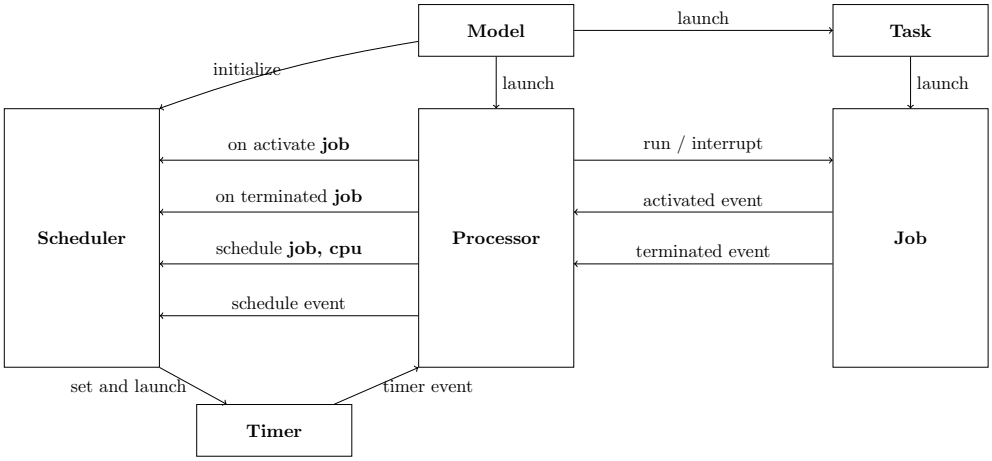


Figure 6.1: Illustration of the architecture of the SimSo Framework (redrawn from [27]).

In this section, the software architecture and structure of the underlying *SimSo* Framework is presented in accordance to Chéramy et al. [27] in order to explain the necessary details to understand the implementation of the proposed scheduling algorithm.
The most relevant SimSo components to model the scheduling system's behavior are the *processor*, *task*, *job*, *model* and *timer* objects as illustrated in Fig. 6.1.
The system's characteristics are defined by a configuration that contains all information about the system, namely the task sets, processors, scheduler and duration of the simulation. Further, the components can be categorized into active processes i.e., components that initiate *events* in the discrete-event simulation and passive processes that are initiated or invoked by active processes.
Due to the fact that the architecture is based on the architecture of real systems, the scheduling algorithm itself is not an active process, but is initiated by the processor. Whenever an event occurs e.g., job activation, termination or a timer event, the processor invokes the corresponding callbacks. For example, when a job is activated, a job *activation event* is placed in the discrete-event queue. Hereafter, the processor dispatches the event and invokes the *on activate* method, that is implemented by the scheduler.
Moreover, each processor in the system is represented by a single processor object that must decide at any given time whether a job is executed, interrupted or the *schedule* callback of the scheduler is executed.
The scheduler can only execute on a single processor at the same simulation time (by default) and must implement the *on terminated*, *on activated*, *init* and *schedule* callbacks

to properly integrate with the framework. Another active process is the timer object, that can be used to initiate events on a processor at specified times.

The task object models the behavior of the simulated task, that is the activation of jobs or the aborting of jobs that exceed their deadlines. Furthermore, the task object maintains a reference to a structure that contains all relevant task descriptions e.g., the identifier, task type, worst-case execution time, average-case execution time, precedence constraints, stack files or a cache model amongst others. A task hence instantiates belonging job objects that simulate the execution on a processor when scheduled.

6.1.1 Execution Time Model

Within the SimSo framework, multiple models of a jobs execution time are realized in a *Execution Time Model* (ETM) [27].

As illustrated in Fig. 6.2, a job *calls* the belonging ETM instance due to a scheduling event e.g., job activation and the remaining execution time of the job is computed by the ETM. Whenever the remaining execution time is depleted, the aforementioned process is repeated until the ETM computes no further remaining execution time. This model allows for the modeling and accounting for various overheads that occur during runtime e.g., context switches and scheduler calls (with fixed time penalties).

The overheads are accounted for on the respective processor on which they occur i.e., the scheduler overhead is accounted for by the processor that called the scheduler.

SimSo execution-time models can be categorized into static and dynamic models, whereas the latter provides job durations (remaining) that change dependent on the context and events that are present during runtime. In this model, increased job durations due to preemptions, migrations, user-specified penalties or cache misses can be described.

The static execution time models can be further divided into a worst-case execution time model and an average-case execution time model. In the worst-case execution time model, each job executes for exactly the specified amount of time, whereas in the average-case execution time model, each job executes for a randomized amount of time that is no more than the specified worst-case execution time.

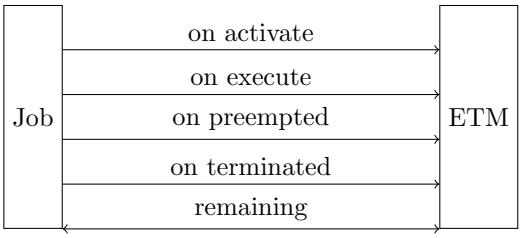


Figure 6.2: Execution Time Model used in SimSo (redrawn from [27]).

6.2 Design and implementation

In this section, the design and implementation of the reservation-based federated scheduling algorithm is presented.
The first Section 6.2.1 *Reservation* covers the implementation details and models of the proposed sporadic reservation server.
Since the current version of *SimSo* only provides sequential, sporadic and arbitrary deadline task models, the design and implementation of the DAG task model is detailed in Section 6.2.2 *DAG task*.
In the last Section 6.2.3 *Scheduler*, the architecture of the scheduler as well as the realization is presented and demonstrated.

6.2.1 Reservation

In the case of partitioned multiprocessor scheduling, each reservation is associated with a designated processor. To that end, each processor is attributed a list to maintain all reservations, that are eligible to service DAG tasks and execute jobs on that processor as illustrated in Fig. 6.3.
Multiple reservations that do not necessarily belong to the same DAG task are partitioned i.e., created on a designated processor. Moreover, each reservation is attributed a processor identifier which memorizes the processor identifier that the respective reservation server is partitioned on. This identification is required for notification purposes which is explained in more detail later.

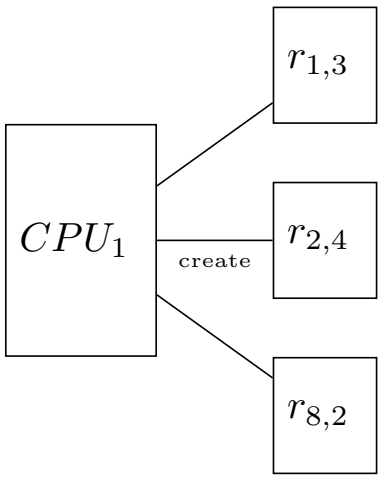


Figure 6.3: Three reservations $r_{1,3}, r_{2,4}, r_{8,2}$, that belong to DAG tasks τ_1, τ_2 and τ_3 respectively are created and initialized on the same designated processor. Further, the runtime environment of the reservation is associated with and maintained by the processor that the reservations are assigned to.

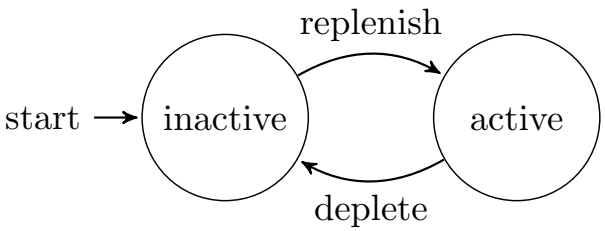


Figure 6.4: The sporadic reservation is modeled by two states *active* and *inactive*, that change upon runtime-budget changes. Each reservation changes state to *active* whenever it is replenished and changes into *inactive* only if the runtime-budget is depleted.

The model of the implemented sporadic reservation can be separated into an *external model* as seen by the scheduling algorithm and an *internal model* of the servicing of a DAG task.

Externally, the reservation is based on the execution model a sporadic arbitrary-deadline task, i.e., whenever the runtime-budget of the reservation is not depleted and the reservation is the highest-priority schedulable entity, it must execute. More precisely, whenever a DAG task is activated (sporadically) the corresponding reservation servers are replenished.

Then, a reservation server either executes an attached job, pulls and attaches a pending job from the associated master-queue or spins if none exists.

The runtime-budget is drained irrespective of whether the reservation is executing or spinning.

The *internal model* consists of the two states *active* and *inactive* as illustrated in Fig. 6.4. Initially, when the reservation is created, the state defaults to *inactive* and changes to *active* only when the reservation is explicitly replenished. The replenishment and activation occurs upon the time of activation of the DAG task. In conclusion, an *active* reservation changes the state into *inactive* if either the relative deadline is expired or the runtime-budget is depleted.

Furthermore, the reservations implement non-preemptive scheduling for the servicing of DAG task jobs. That is, if a DAG subjob is activated and attached to a belonging reservation, then no other job is serviced by that reservation until that DAG subjob is finished. This is guaranteed by the *pull* mechanism of the reservation that is only initiated if no job is attached to the reservation server.

In order to prevent a reservation from *overrunning* i.e., to execute jobs for more than the budget accounts for, the runtime budget must be monitored. As previously mentioned, timer objects (as active components) can be used to invoke methods and thus be used to implement the runtime-budget monitoring. To that end, each reservation is attributed a timer that is set up successively according to the following recurrence shown in Eq. (6.1).

Whenever the timer expires, the budget is either depleted and the reservation thus inactivated or the next possible depletion time is chosen to set up the timer.

$$\begin{aligned} t_n &= t_{n-1} + R_{n-1}, n \geq 1 \\ R_{n-1} &= R_{n-2} - \text{budget drained over the interval } [t_{n-2}, t_{n-1}), n \geq 2. \end{aligned} \tag{6.1}$$

To better illustrate the recurrence, let t_0 denote the replenishment time of a reservation i.e., the reservation is eligible to service jobs from then on for the whole associated budget R_0. The earliest time this budget could possibly be depleted is at time $t_1 = t_0 + R_0$, in the case that the reservation services jobs without any preemption during that interval. By the setting of the timer (cf. Eq. (6.1)), the remaining budget is verified at said point in time and inactivated if the budget is depleted.

In the case that the reservation was preempted during the interval $[t_0, t_1)$ and the budget was only drained by some $\Delta \leq R_0$, then the earliest time the reservation could be depleted is given by $t_0 + R_0 + (R_0 - \Delta) = t_1 + R_0 - \Delta$.

Subsequently, the timer is set successively until either the budget is depleted and the reservation inactivated or the relative deadline is missed.

6.2.2 DAG task

Besides sporadic reservations, DAG tasks are required to be implemented and integrated with the SimSo Framework.

In the following proposed implementation, a DAG task is modeled by a set of subtasks that are subject to precedence constraints. More formally, the tuple $\tau_i \stackrel{\text{def}}{=} (G_i, D_i, T_i)$ defines a sporadic arbitrary-deadline DAG task where $G_i = (V_i, E_i)$ denotes the set of subtasks and precedence constraints.

A first objective in the realization of the DAG task model is to implement the release i.e., activation mechanisms of the subtasks. That is, the first subtask is released according to the sporadic release model and relative deadline, whereas the remaining subtasks are released as soon as all their precedence constraints are met and have an absolute deadline given by $t_0 + D_i$. Here, t_0 denotes the release time of the first subtask.

To simplify the distinction between the aforementioned release mechanisms, each DAG task is enforced to have a unique source subtask. Note that this imposes no further constraints since it is always possible to augment a subtask with zero execution demand that precedes all source subtasks. Another benefit of this unique source model is that the request for replenishment can be coupled with the activation of the source task.

The activations of non-source subtasks $v_\ell \in V_i$ of a given DAG task is hence given by the last finishing time amongst all subtasks $(v_k, v_\ell) \in E_i$. This is stated more formally in the following equation Eq. (6.2)

$$a_{i,\ell} = \max \{ f_{i,k} \mid (\tau_{i,k}, \tau_{i,\ell}) \in E_i \} . \tag{6.2}$$

To implement the above release constraints, each subtask $\tau_{i,\ell}$ is attributed a list of inbound and outbound subtasks as detailed in the following definition.

6.2.1 Definition (inbound-outbound subtasks). A subtask $\tau_{i,k}$ is in $in(\tau_{i,\ell})$ i.e., inbound if and only if $(\tau_{i,k}, \tau_{i,\ell}) \in E_i$. Respectively, a subtask $\tau_{i,k}$ is in $out(\tau_{i,\ell})$ i.e., outbound if and only if $(\tau_{i,\ell}, \tau_{i,k}) \in E_i$.

Whenever a subtask $\tau_{i,\ell}$ finishes its execution, all outbound subtasks are notified of the finishing. Each outbound task, then removes $\tau_{i,\ell}$ from their respective inbound list and releases a job if and only if the inbound list is empty i.e., all precedence constraints are met.
Note that by this design, the inbound- and outbound list must be reset upon each DAG job's activation, as they belong to the runtime environment.

6.2.3 Scheduler

After all relevant components are described, this section elaborates on the realization of the scheduling algorithm.

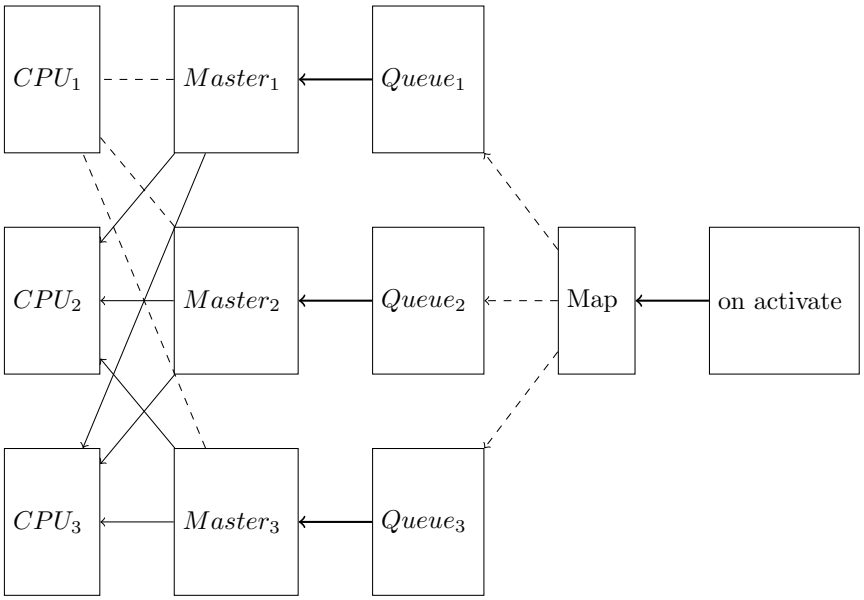


Figure 6.5: The global scheduler delegates jobs to their belonging master-queues and respective processors.

In the proposed reservation-based federated scheduling, it is required to compute the system of sporadic arbitrary-deadline reservation servers that can feasibly service the DAG task set. As illustrated in Fig. 6.5, each DAG task is associated with a unique master. That is, a DAG task τ_i is associated with $master_i$ and releases jobs to the ready-queue of said master (master-queue in the following) exclusively. Thus, if a DAG task τ_i is deemed to be serviced by reservations $\tau_{i,1}, \tau_{i,2}, .., \tau_{i,m_i}$, then these reservation servers service pending jobs from $master_i$'s ready-queue.
Moreover, since all reservation servers are partitioned to specific processors, each master must know all processors to which the DAG task's respective reservation servers are partitioned to. Likewise, each reservation server must know the associated master. In consequence, the partitioning of the reservation servers consists of the creation and connection thereof with the associated master as is further explained in Example. 6.2.2.

6.2.2 Example (Master-queue). Let τ_1 be a DAG task that requires three reservation servers to be sufficiently serviced, namely $\tau_{1,1}, \tau_{1,2}, \tau_{1,3}$.
Further, these reservations are partitioned to the available processors according to some partitioning algorithm.
If $\tau_{1,3}$ is partitioned to processor CPU_2 then a new reservation with the settings of $\tau_{1,3}$ is created and added to the reservation-list of processor CPU_2. Similarly, let $\tau_{1,1}$ and $\tau_{1,2}$ be partitioned to processor CPU_1.
Each reservation server, that services the DAG task τ_1, namely $\tau_{1,1}, \tau_{1,2}, \tau_{1,3}$ knows the corresponding $master_1$ and queue of ready subjobs subsequently.
Thus, whenever a reservation is active, a job can be pulled from the queue of $master_1$.

In order to realize this behavior, the *on activate* callback is implemented as shown in List. 6.1. When a subjob of a DAG task τ_i is activated, the *on activate* callback is invoked.
With reference to Example. 6.2.2, $master_i$ is identified and the job is enqueued into that master-queue. In the case that the released job is the first subjob of the DAG task i.e., a source job, the belonging reservation servers are prompted to replenish their budget.

```
1 def on_activate(self, job):
2     master = self.map_task_master[job.task.identifier]
3     if job.task.trigger is True:
4         now = self.sim.now_ms()
5         for cpu in [x for x in self.processors if x.identifier in master.
              processors]:
6             self.map_cpu_sched[cpu.identifier].replenish(now, master)
7     master.append(job)
8     job.cpu.resched()
```

Listing 6.1: On activation callback, that is called whenever a job is released to the system.

Additionally, after the job is enqueued in the corresponding master-queue, all processors that maintain reservation servers belonging to DAG task τ_i are prompted to *reschedule* and are notified about the existence of pending jobs.

Whenever a job is terminated, the *on terminated* callback is invoked and thus the terminated job is detached from the servicing reservation server and removed from the respective master-queue.

```
1     def on_terminated(self, job):
2         if job.reservation is not None:
3             job.reservation.detach()
4         job.cpu.resched()
```

Listing 6.2: The on terminated callback detaches a job from the servicing reservation.

Based on the implementation details, the proposed scheduling algorithm can be classified as a *semi-partitioned* multiprocessor scheduling algorithm.

That is, each DAG task is eligible to be serviced by a set of reservations that are partitioned to a set of processors.

Therefore, each subtask may execute on all processors that host the DAG task's belonging reservation servers.

The logical structure of the scheduling algorithm is shown in Fig. 6.6. As illustrated, the scheduling algorithm is distributed over all available processors in the system. More precisely, there is one processor designated to execute the *global* scheduler and the remaining processors are used to execute the *local* schedulers.

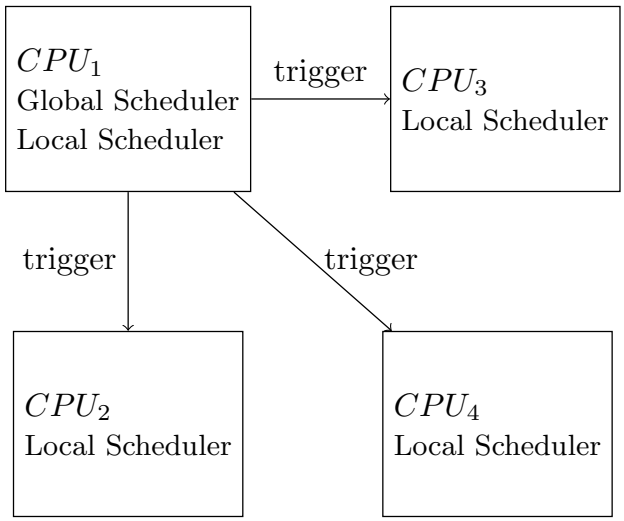


Figure 6.6: Distributed software architecture of the reservation-based federated scheduling algorithm.

```
1  def schedule(self, cpu):
2      master = max(self.master_list, key = lambda x: (1 if any(x.ready_list)
           else 0))
3      if master and master.ready_list:
4          cpu_min = min([cpu for cpu in self.processors if cpu.identifier in
               master.processors], key = lambda x: (1 if x.running else 0))
5          if cpu_min.running is None:
6              return self.map_cpu_sched[cpu_min.identifier].schedule(cpu_min)
```

Listing 6.3: The global scheduler delegates the scheduling decision to the local schedulers of belonging processors.

The global scheduler probes all master-queues for pending jobs as illustrated in List. 6.3. In the case that a master-queue with pending jobs is found, the scheduling decision is delegated to the local schedulers of all processors that are associated with said master-queue and are currently not executing any workload. Subsequently, the global scheduler assures that no processor remains idle if any eligible job is ready for execution. The local scheduler is displayed in List. 6.4, which schedules the highest-priority active reservation and executes the attached job or executes the pulling mechanic or spins at any point in time.

```
def schedule(self, cpu):
    active_reservations = [s for s in self.server_list if not s.depleted]
    now = self.sim.now_ms()
    if self.running_server is not None:
        self.running_server.update_time(now)
        try:
            schedule_next = min(active_reservations, key = lambda x : x.
                deadline)
            self.running_server = schedule_next
            if self.running_server.job is not None and self.running_server.
                job.is_active():
                job = schedule_next.job
                return (job, cpu)
            else:
                try:
                    job = self.running_server.master.ready_list.pop(0)
                    self.running_server.attach(job)
                    return (job, cpu)
                except IndexError:
                    print 'No jobs are pending.'
        except ValueError:
            print 'No reservations are active.'
    else:
        try:
            schedule_next = min(active_reservations, key = lambda x : x.
                deadline)
            self.running_server = schedule_next

            if self.running_server.job is not None and self.running_server.
                job.is_active():
                return (self.running_server.job, cpu)

            else:
                try:
                    job = self.running_server.master.ready_list.pop(0)
                    self.running_server.attach(job)
                    return (job, cpu)
                except IndexError:
                    print 'No jobs are pending.'
        except ValueError:
            print 'No reservations are active.'
```

Listing 6.4: The local scheduler schedules the highest-priority active reservation server.

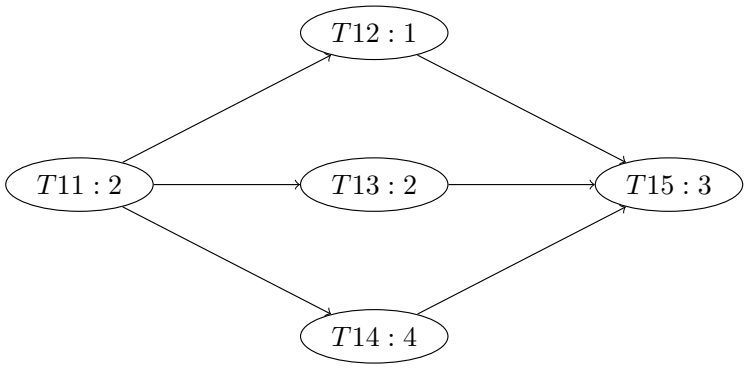


Figure 6.7: A constrained-deadline DAG task with $C_i = 12$, $L_i = 9$, $D_i = 10$ and $T_i = 15$ that belongs to task τ_1 in the generated schedule in Fig. 6.8.

6.3 Summary

In this section, the design and implementation of a prototype of the proposed reservation-based federated scheduling algorithm using partitioned multiprocessor scheduling is described.

Despite the fact, that the simulation does not account for details that must be considered in real system implementations such as synchronizing the access to the master ready-queues, inter-processor communication or discrete time-quanta between scheduling decisions, the prototype proves a possible implementation concept for partitioned scheduling of the reservations. This is due to the fact, that the simulation framework enforces scheduler architectures and interfaces, that are based on real systems.

To demonstrate the workings of the implementation of reservation-based federated scheduling, a schedule for an example DAG task set consisting of three DAG tasks as shown in Table. 6.1 is simulated.

The reservation servers are generated using the *R-MIN* algorithm and partitioned to the processors according to partitioned deadline-monotonic scheduling.

To schedule the given task set, five reservations are created as illustrated in Fig. 6.2. Three reservations are servicing task τ_1 with a budget of 10, period of 15 and a relative deadline of 10. The reservation servers for task τ_2 and τ_3 are parameterized using the task's initial parameters as illustrated in Table. 6.2 since they are classified as light tasks.

Task	WCET	Deadline	Period	Critical path	Reservation Servers (R-MIN)
τ_1	12	10	15	9	3
τ_2	1	30	30	0.9	1
τ_3	1	20	20	0.7	1

Table 6.1: Generated reservation servers that are used in the schedule illustrated in Fig. 6.8.

The reservations of task τ_1 are partitioned to CPU_1, CPU_2 and CPU_3, whereas task τ_2 is partitioned to CPU_2 and τ_3 is partitioned to CPU_3 respectively.
As can be seen in Fig. 6.8, the reservations $\tau_{1,1}, \tau_{1,2}, \tau_{1,3}, \tau_{2,1}$ and $\tau_{3,1}$ are replenished and thus activated synchronously with the release of DAG task τ_1, τ_2 and τ_3 respectively at time $t = 0$.
Since the reservation servers $\tau_{1,1}, \tau_{1,2}, \tau_{1,3}$ have the shortest deadline and thus highest priority, $\tau_{2,1}$ and $\tau_{3,1}$ are preempted until the budget of the reservations of τ_1 are depleted at time $t = 10$.
Therefore, tasks τ_2 and τ_3 are serviced at time $t = 10$ and finish after one unit of time. As can be seen, the reservation servers are replenished whenever a DAG task is activated and remain active until their respective budget is depleted.

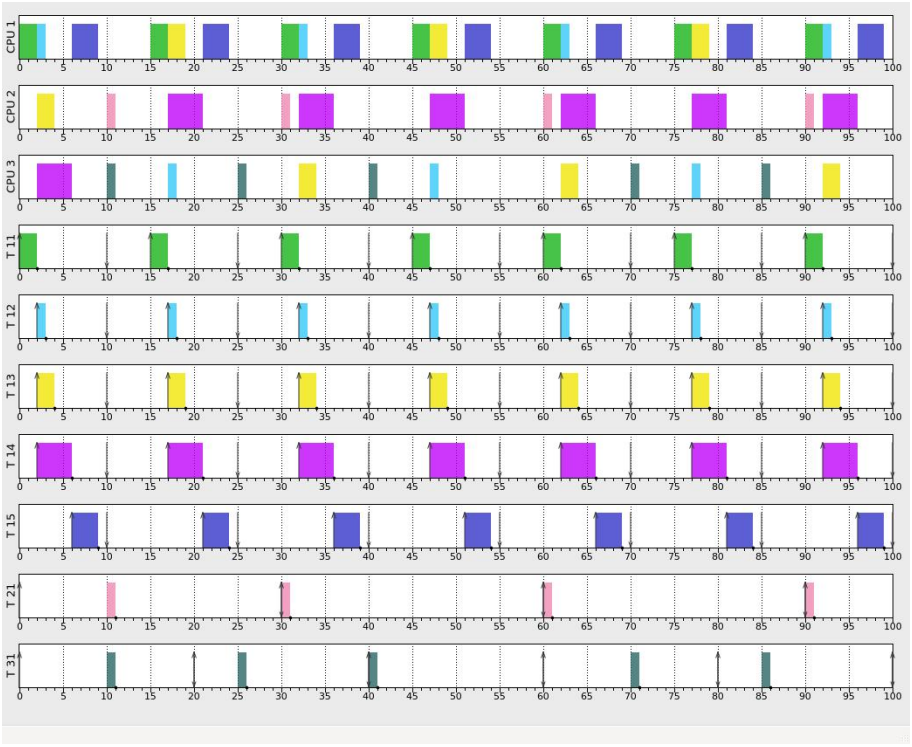


Figure 6.8: Simulated schedule of reservation-based federated scheduling using partitioned deadline-monotonic scheduling and *R-MIN* to generate the reservation servers for the DAG task set listed in Table. 6.1.

Reservation Server	Budget	Deadline	Period	Partition
$\tau_{1,1}$	10	10	15	CPU_1
$\tau_{1,2}$	10	10	15	CPU_2
$\tau_{1,3}$	10	10	15	CPU_3
$\tau_{2,1}$	1	30	30	CPU_2
$\tau_{3,1}$	1	20	20	CPU_3

Table 6.2: Generated reservation servers that are used in the schedule illustrated in Fig. 6.8.

7 Evaluations

Despite the availability of theoretic means to evaluate the performance of scheduling algorithms and scheduling test algorithms by e.g., dominance relationships, speedup factors and capacity augmentation bounds, empirical evaluations using synthetically generated task sets are commonly used to evaluate the performance of scheduling algorithms. Using synthetically generated task sets, the performance of the scheduling test algorithms can be evaluated more accurately with respect to the characteristics of the systems of interest.

In order to allow for a fair and transparent evaluation of scheduling algorithms, the algorithm that generates the synthetic task set must satisfy a couple of prerequisites.

Namely, in the researches of Emberson et al. [32], it is stated that a task set generating algorithm should satisfy the three conditions *efficiency*, *independence* and *being bias-free* [32]. According to the authors, the aspect of efficiency is mandatory in order to allow for the generation of large enough test sets i.e., sets of task sets, to be statistically significant.

Moreover, the independence condition requires that all parameters describing a task set must be variable without restricting other parameters.

Lastly, the bias-free condition requires that the generated task set is unbiased in the sense that no particular task set in the set of all possible task sets (for given parameter constraints) are preferred.

For the evaluation of scheduling test algorithms of uni- and multiprocessor systems, appropriate methods, that satisfy the above conditions have been developed.

Namely, the *uunifast* algorithm [15], that generates a set of specified cardinality with utilizations $0 < u_i \leq 1$ drawn uniformly. Moreover, the utilizations accumulate to a user-specified set utilization.

In order to evaluate multiprocessor scheduling test algorithms, task sets with accumulated utilizations larger than one are required. To that end, the *rand-fixed-sum* algorithm [33] can be used to generate uniformly distributed random vectors of arbitrary dimension, that accumulate to a user-specific value.

Additionally, it is possible to specify an interval of the vector elements i.e., $a < x_i \leq b$ for some $a \leq b \in \mathbb{R}$.

It is therefore possible to use the *rand-fixed-sum* algorithm to generate utilizations for DAG task sets where an individual task may have an utilization larger than one.

However, in those cases where the schedulability test depends on the DAG structure itself, this approach to task generation is infeasible.

To date, there is no algorithm that generates synthetically DAG task sets that satisfy the three conditions *efficiency*, *independence*, being *bias free* and generating a belonging DAG structure. Instead, algorithms either generate DAG task sets to model the internal precedence structure of specific applications e.g., Fast-Fourier Transforms, LU decompositions [52] or randomized DAG structures [28] or specific randomized DAG structures [28].

The remainder of this chapter is organized as follows. In the following Section 7.1 *Evaluated Algorithms* all schedulability tests, that are subject to evaluation are summarized and the rational for their choosing is explained.

Followingly, Section 7.2 *DAG Task Set Generation* details the used DAG task sets generation algorithms and discusses incurring problems and solution approaches.

Lastly, Section 7.3 *Experimental Evaluation* covers the numerical evaluations and concludes the results.

7.1 Evaluated Algorithms

In this section, the algorithms that are subject to evaluation are described. That is, the proposed reservation-based federated scheduling algorithms, namely *Split-On-Fail (SOF)* in the *R-MIN (MIN)* and *R-EQUAL (EQ)* variants as wells as the different partitioning heuristics first-fit (FF), best-fit (BF), and worst-fit (WF) are compared against the semi-federated scheduling approach by Jiang et al. [38] and the federated scheduling variant by Baruah [4].

Furthermore, the evaluation of Jiang et al. [38] suggested the dominance of their proposed semi-federated scheduling compared to the state-of-the-art schedulability analyses of DAG task sets i.e., EDF-based scheduling analyses [39,54] or the response time based analysis for G-EDF by Melani et al. [50]. Therefore, all these schedulability analyses are not explicitly considered in this evaluation.

Additionally, due to the fact that the evaluations of semi-federated scheduling did not suggest a notable difference between the performance of their two proposed algorithms, only the algorithm $SF[X+1]$ was adopted and is referred to as *S-FED* in the following.

In order to allow for a fair evaluation and to make use of the advantage of the generic sporadic task model of the reservation servers, two *SOF* variants are evaluated. That is, one variant uses partitioned EDF (EDF) for arbitrary-deadline reservation servers by using the approximated demand-bound function as proposed by Baruah [3] and the second variant uses partitioned deadline-monotonic (DM) scheduling. Both schedulability analyses and associated schedulability tests are detailed in Chapter 2 *Real-Time Systems and Fundamentals*.

7.2 DAG Task Set Generation

Due to the fact, that the federated scheduling variant for the scheduling of sporadic arbitrary-deadline DAG task sets by Baruah requires the concrete DAG structure, an appropriate generation algorithm is mandatory.
To that end, Section 7.2.1 *DAG Task Generation* details the generation algorithms used to generate such DAG structures and DAG task sets subsequently. Since these so generated task sets exhibit statistical bias and further problems as will be discussed, another parametric generation method is used in addition.
Therefore, Section 7.2.2 *Parametric DAG Task Generation* describes the algorithm to generate purely parametric DAG tasks and task sets respectively.

7.2.1 DAG Task Generation

The parameters of a DAG task are defined by the tuple $\tau_i \stackrel{\text{def}}{=} (G_i, T_i, D_i)$, where $G_i = (V_i, E_i)$ denotes a directed-acyclic graph, T_i the minimal inter-arrival time and D_i the relative deadline.
Further, each vertex in G_i represents a sub task with an associated worst-case execution time. Based on these subtasks and precedence constraints, the critical-path length L_i denotes the largest worst-case execution time, that must be executed sequentially and C_i denotes the overall worst-case execution time of all subtasks.
Due to the parametric dependency of the worst-case execution time C_i and critical-path length L_i on the generated subtasks and precedence constraints, the generation sequence is as follows.

1. Generate n subtasks by sampling n-times from a predefined WCET range
2. Generate edges
3. Compute the critical path length L_i
4. Compute worst-case execution time C_i
5. Generate a deadline such that $D_i > L_i$
6. Generate a period T_i such that $U_i < m$ where m denotes the number of processors.

In the following an adjacency matrix is used in order to represent the precedence constraints of a DAG tasks more formally described in the following definition.

7.2.1 Definition. Let $V = \{v_1, v_2, v_3, .., v_N\}$ denote the set of generated subtasks and let $\mathcal{A}$ denote the relation $\mathcal{A} : V \times V \mapsto \{0, 1\}$ such that $a_{i,j} = 1$ if subtask v_j depends on v_i and 0 otherwise.

To generate a DAG structure efficiently, it is required to identify properties that allow for the fast construction of an adjacency matrix that belongs to a directed-acyclic graph (DAG).
Cordeiro et al. [28] report of the lack of a general efficient method to generate non-isomorphic graphs uniformly. However, it is possible to efficiently construct a subset of adjacency matrices, which provably belong to a DAG as detailed in the following lemma.

7.2.2 Lemma. *Let an adjacency matrix $\mathcal{A}$ be a lower- or upper-triangular (without the diagonal elements) matrix, then the belonging graph is a DAG.*

Proof. Let $\mathcal{A}$ denote an adjacency matrix according to the aforementioned definition, then due to the structure of the lower-triangular matrix (without diagonal elements), for every subtask v_i, the set of subtasks that v_i depends on is at most $v_{i+1}, v_{i+2}, .., v_N$.
Let the belonging graph contain at least one cycle, then there must exist a sequence of subtasks $.., v_1, v_2, v_3, .., v_\ell, v_1$ where at least one index repeats itself. Further, let ℓ be that index and note that v_1 depends on v_ℓ and therefore $\ell > 1$ by assumption. Due to the transitivity of the indexing property, if v_1 is on a path to v_ℓ it must be that $1 > \ell$, which contradicts the assumption. □

It is to emphasize, that the set of all lower- or upper-triangular adjacency matrices are a true subset of all adjacency matrices that belong to a DAG i.e., lemma 7.2.2 is a sufficient condition for the creation of DAG tasks belonging adjacency matrices.
In the *Erdos-Renyi* algorithm Alg. 7.1, an adjacency matrix $\mathcal{A}$ is generated by setting each entry $a_{i,j}$ below (or above) the diagonal to one with probability P_0 and to zero with probability $1 - P_0$, which by lemma 7.2.2 yields a DAG structure.
A further adaptation of the *Erdos-Renyi* algorithm is the *layer-by-layer* algorithm illustrated in Alg. 7.2. In this algorithms, all subtasks are grouped into disjoint layers $L_1, L_2, .., L_k$ such that no two subtasks belonging to the same layer are connected i.e., $a_{i,j} = 0$ if $layer(v_i) = layer(v_j)$.

Require: Probility P_0, number of subtasks n.
Ensure: An adjacency matrix $\mathcal{A}$ such that the associated graph is a DAG.
1: $P \leftarrow P_0$
2: $\mathcal{A} \leftarrow zeros(n, n)$ ▷ $n \times n$ Matrix initialized to zero.
3: **for** $i \leftarrow 2$ **to** n **do**
4: **for** $j \leftarrow 1$ **to** $i - 1$ **do**
5: $a_{i,j} \leftarrow 1$ with probability P_0 and 0 with probability $(1 - P_0)$
6: **end for**
7: **end for**
8: **return** $\mathcal{A}$

Algorithmus 7.1: The Erdos-Renyi algorithm generates randomized precedence constraints, that belong to a DAG structure [28].

Moreover, the partition of the subtasks into k disjoint layers can be efficiently computed by generating a set K, that consists of $k-1$ random natural numbers in the range $1, 2, .., n$. After the the sorting of K (ascending), each subtask v_i is assigned to K_ℓ if $K_{\ell-1} \leq i \leq K_\ell, k \geq \ell \geq 1$ such that the complexity is dominated by either the sorting operation or the linear traversal of all subtasks.

In the researches *Parallel Real-Time Scheduling of DAGs* by Saifullah et al. [54] and *Semi-Federated Scheduling* by Jiang et al. [38] the following strategy to generate periods is chosen.

$$T_i \stackrel{\text{def}}{=} \left(L_i + \frac{C_i}{0.5 \cdot M} \right) \cdot (1 + 0.25 \cdot \Gamma(2, 1)) \tag{7.1}$$

where Γ denotes the gamma distribution.

The authors argue that by this definition each period is valid i.e., $L_i \leq T_i$ for each DAG task. Furthermore, it is argued that in the case of low numbers of processors the generated task set still contains a *reasonable* number of tasks whilst not limiting the average DAG task' s utilization.

Additionally, the generated periods should result in the generation of some high utilization tasks as well as low utilization tasks. Unfortunately, when trying to reproduce the proposed period generation, the resulting task sets did not contain a reasonable mixture of high and low utilization tasks. Instead, in most cases the generated task sets consisted of light tasks i.e., $C_i < T_i$ thus not evaluating the algorithms performance with respect to DAG task sets.

Since the advantages in reservation-based federated scheduling partly come from the possibilities to partition light tasks together with heavy tasks to the same processors, the ratio of heavy to light tasks is an important parameter to control for in the evaluation.

Require: Probility P_0, number of subtasks n, number of layers k.
Ensure: An adjacency matrix $\mathcal{A}$ with k-layers such that the associated graph is a DAG.
1: Partition v_i into $L_1, L_2, .., L_k$ ▷ $layer(v_i) = \ell$ if $v_i \in L_\ell$
2: $P \leftarrow P_0$
3: $\mathcal{A} \leftarrow zeros(n, n)$ ▷ $n \times n$ Matrix initialized to zero.
4: **for** $i \leftarrow 2$ **to** n **do**
5: **for** $j \leftarrow 1$ **to** $i - 1$ **do**
6: **if** $layer(v_j) < layer(v_i)$ **then**
7: $a_{i,j} \leftarrow 1$ with probability P_0 and 0 with probability $(1 - P_0)$
8: **end if**
9: **end for**
10: **end for**
11: **return** $\mathcal{A}$

Algorithmus 7.2: The Layer-By-Layer algorithm generates randomized precedence constraints, that are grouped into layers and belong to a DAG structure [28].

Therefore, in order to better control for the number of heavy DAG tasks generated, the following changes are proposed to generate periods.

7.2.3 Lemma. *A DAG task τ_i with a given worst-case execution time C_i and critical path length L_i is a heavy task with probability $\frac{1}{\alpha}$ and a light task with probability $1 - \frac{1}{\alpha}$ for a period T_i, that is given by*

$$L_i + X_i, \text{ where } X_i \sim U\left(0, \alpha \cdot (C_i - L_i)\right), \alpha \geq 1. \tag{7.2}$$

Proof. A task τ_i is considered heavy if $C_i \geq T_i$ holds and light otherwise. Therefore the probability for τ_i to be a heavy task is given by $Pr\left(T_i \leq C_i\right)$ or $Pr\left(X_i \leq C_i - L_i\right)$ respectively. Subsequently, the probability is given by

$$\frac{1}{\alpha \cdot (C_i - L_i)} \int_0^{C_i - L_i} 1 \; du = \frac{1}{\alpha}. \tag{7.3}$$

Further, the probability of τ_i to be a light tasks is given by $1 - \frac{1}{\alpha}$. □

In order to control for the task sets utilizations, a subset sum approximation algorithm is used. The algorithm selects a subset from a set of generated DAG tasks such that the resulting accumulated utilization of the chosen tasks is equal to a given system utilization within a specified ϵ error-bound.
Additionally, each DAG task set should contain a *reasonable* number of tasks to allow for non-trivial evaluations. In accordance to the proposed algorithms to generate periods, the periods are generated depending on the specified system utilization. That is, a task set with specified normalized utilization of u and m processors needs to surpass a threshold value in order to use the proposed period generation method.
Intuitively, this threshold relates to the number of tasks that a generated task set should at least contain. Using this method it is possible to influence the number of heavy tasks in the generated task set more precisely.
It is however to consider that due to the utilization dependent period generation, an additional bias is introduced into the generated task sets.
In order to make the evaluations as transparent as possible, statistics are generated for all generated task sets that are used to evaluate the scheduling test algorithms in Chapter 8.3 *Appendix*.

7.2.2 Parametric DAG Task Generation

For an improved evaluation using the purely parametric DAG task models i.e., $\tau_i = (C_i, L_i, D_i, T_i)$, the *rand-fixed-sum* algorithm is used [33] to generate utilizations for individual tasks $0 < U_i \leq m$ (where m denotes the number of homogeneous processors) for a given cardinality and cumulative utilization of the to be generated task set.

Furthermore, each task set is characterized by the parameters $0 < \alpha$, $0 < \beta \leq 1$ and T_i that refer to respective realizations of uniformly distributed random variables.

Based on these realizations, the remaining DAG task parameters are given as $C_i = U_i \cdot T_i$, $D_i = \alpha \cdot T_i$, and $L_i = \beta \cdot D_i$, respectively. Moreover, the periods for each task are drawn uniformly from $(0, 100]$ to cover a wide range of characteristics. Additionally, the range of the two parameters α and β for generating deadline and critical-path length is different for different settings. In consequence, the ratio of the critical-path length to the cumulative worst-case execution time i.e., a metric for how much a task can be parallelized, is given by $\frac{L_i}{C_i} = \beta\alpha \cdot T_i$. Consequently, this ratio is the product of the realizations of three independent uniformly distributed random variables.

7.3 Experimental Evaluation

This section presents the evaluation results based on synthetic task sets using reservation-based federated scheduling *SOF* under partitioned deadline-monotonic (DM) and partitioned earliest-deadline-first (EDF) schedulers, semi-federated scheduling *S-FED* and federated scheduling as proposed by Baruah *BARU-FED* as described in Section 7.1 *Evaluated Algorithms*.

The schedulability tests are compared in terms of the *acceptance ratio* where the *acceptance ratio* metric relates the number of *deemed to be schedulable* task sets to the number of *deemed to be unschedulable* task sets for tested task sets of a fixed utilization.

Thus, the *acceptance ratio* of a schedulability test that envelopes the *acceptance ratio* of another schedulability test is considered better in the sense that more of the tested task sets are schedulable by this schedulability test. Note however, that this notion of dominance refers to the characteristics of the generated and thus tested task sets.

The remainder of this section is organized such that Section 7.3.1 *Evaluations Based on DAG Task Sets* details the evaluations using the generation method elaborated on in Section 7.2.1 *DAG Task Generation*.

Respectively, Section 7.3.2 *Evaluations Based on Parametric DAG Task Sets* covers the evaluations of the purely parametric DAG task model as described in Section 7.2.2 *Parametric DAG Task Generation*.

7.3.1 Evaluations Based on DAG Task Sets

In the following sections, experiments for implicit-deadline task sets for 8 and 16 processors are evaluated. The rather small number of experiments is due to the fact, that the generated DAG task sets for a single experiment can amount to as much as 4GiB.

Furthermore, the computation of the critical-path length is rather expensive even when using the highly optimized vectorized *numpy*[1] primitives (cf. Chapter 8.3 *Appendix* Listing. 1) as well as the expensive list-schedule construction for *BARU-FED* (cf. Chapter 8.3 *Appendix* Listing. 2) resulting in long evaluation times.

In the **first experiment**, different DAG task sets with varying settings of edge-probability, heavy and light task probability on 8 homoheneous processors are tested with respect to the *acceptance ratio*. For each normalized utilization (in ten percent steps), 100 task sets are generated where each task set consists of varying numbers of DAG tasks as can be seen in the associated statistics in Chapter 8.3 *Appendix*.

The **second experiment** is conducted in the same manner except for using 16 homogeneous processors.

The results are illustrated in Fig. 7.1 for the first experiment and Fig. 7.2 for the second experiment respectively.

Not surprisingly, the federated scheduling approach by Baruah dominates all other evaluated schedulability tests since it is the only test that takes the concrete DAG structure into account. Therefore, the difference between the parametric schedulability tests and *BARU-FED* may be used as a comparative reference of performance loss due to the loss of information (DAG structure).

Further, it can be observed that *S-FED* is dominated by *SOF-EDF-BF-EQ*, *SOF-EDF-BF-MIN* whenever the edge-probability is high e.g., 90%, which implies a weak degree of parallelism. Expectedly, it is to note that the deadline-monotonic partitioning proves inferior to earliest-deadline-first partitioning.

In the second experiment, the difference between *BARU-FED* and the parametric DAG task set schedulability tests increases substantially. As can be seen in the task set statistics, starting from 60% normalized utilization the DAG task densities exceed 100% thus increasing the difference between the calculated minimal number of processors in *BARU-FED* and the allocated amounts of resources by *SOF* or *S-FED*.

Nonetheless, the proposed *SOF* algorithms is competitive with the *S-FED* approach for the generated task sets.

7.3.2 Evaluations Based on Parametric DAG Task Sets

In the following sections experiments for implicit-, constrained-, and arbitrary-deadline task sets are evaluated.

For each setting, task sets for 8, 16, and 32 processor systems are generated. More precisely, for each normalized utilization (in five percent steps), 100 task sets are generated where each task set consists of 20 DAG tasks. This task set size was selected to be large enough

[1] http://www.numpy.org/

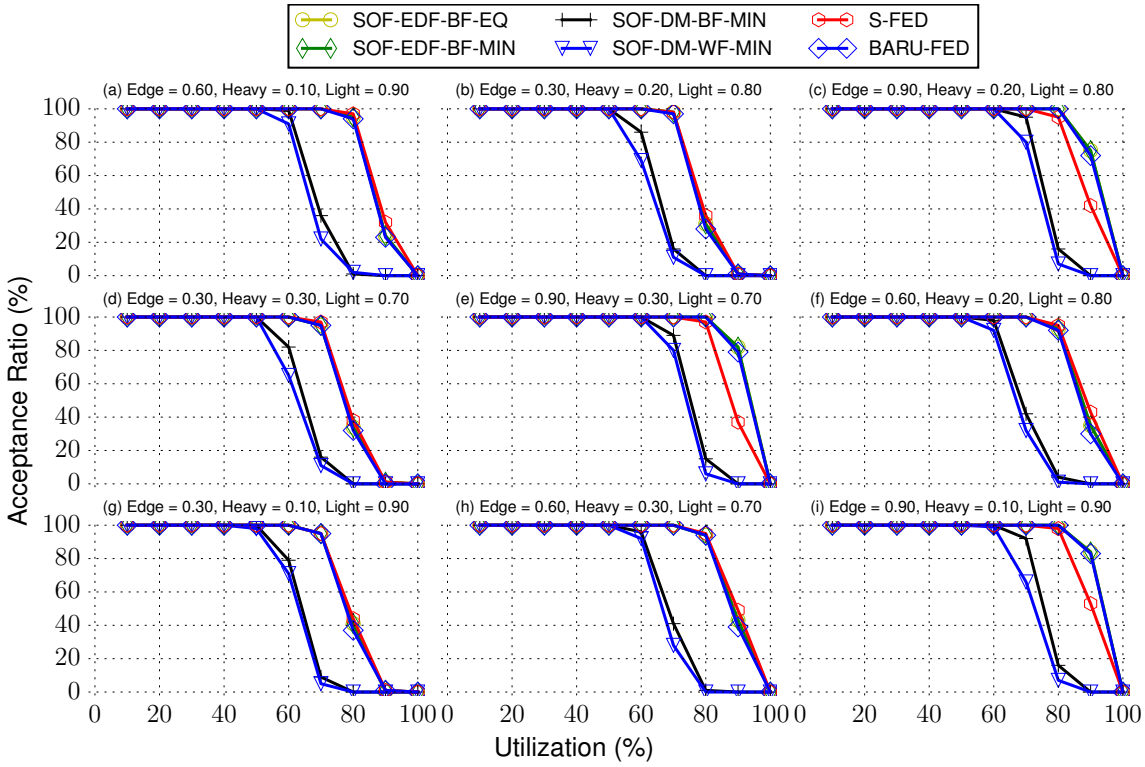


Figure 7.1: Acceptance ratio of the algorithms *SOF*, *S-FED* and *BARU-FED* on 8 homogeneous processors for implicit-deadline DAG task sets. The generation specific parameters edge probability, probability of heavy and light DAG tasks are denoted in the respective titles.

to allow smaller individual utilizations and thus more options for the partitioning, whilst being small enough to still be difficult enough to partition.

For better illustration, the different experimental setups for implicit, constrained and arbitrary-deadline DAG task sets are detailed separately.

Experimental Results on Implicit-Deadline Task Sets

For implicit-deadline DAG task sets, three different experiments were conducted using uniformly drawn critical-path lengths from varying intervals to model different degrees of parallelized DAG tasks.

In the **first experiment** as illustrated in Fig. 7.3, the critical-path length is drawn uniformly from the interval $[0.1Ti, 0.3T_i]$ representing highly parallelized DAG tasks.

For comparison, the **second experiment** as illustrated in Fig. 7.4 represents DAG tasks with critical-path lengths within the interval $[0.3T_i, 0.6D_i]$ i.e., medium parallelized DAG tasks.

In the **third experiment** shown in Fig. 7.5, critical-path lengths are drawn from $[0.6T_i, 0.9T_i]$, thus representing weakly parallelized DAG tasks to demonstrate the sensitivity of the evaluated schedulability tests with respect to this parameter.

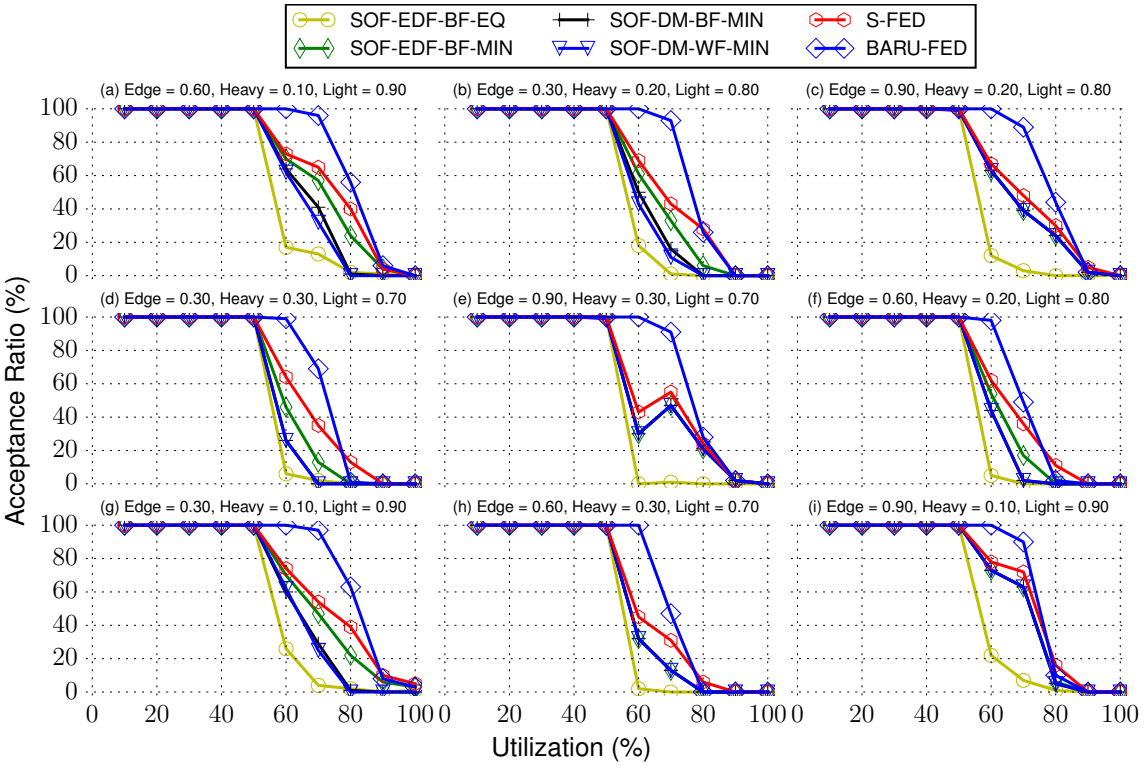


Figure 7.2: Acceptance ratio of the algorithms *SOF*, *S-FED* and *BARU-FED* on 16 homogeneous processors for implicit-deadline DAG task sets. The generation specific parameters edge probability, probability of heavy and light DAG tasks are denoted in the respective titles.

As can be observed, *S-FED* exhibits a distance in cutoff utilization of $10\%, 5\%$ and 0% compared the respective best performing *SOF* variants in the first, second and third experiment respectively. Therefore, the performance differences diminish with lowered degrees of DAG task parallelism. Furthermore, the decline in acceptance ratio of *S-FED* is steeper than for the all *SOF* variants. Additionally, with increasing numbers of processors the performance of all *SOF* variants tend to align in all experiments.

In the third experiment, it can be seen that *S-FED* and the best performing variant *SOF-EDF-BF-MIN* behave similarly up to a cutoff utilization of 60%, 45%, and 35% for 8, 16, and 32 processors. In general, with a slightly slower ratio than the best SOF variants, but the dominance decreases with the number of processors. It can also be observed that the *R-EQUAL* variant compared to the best performing *SOF-EDF-BF-MIN* algorithm exhibits a substantially lowered cutoff utilization. This is due to the fact that *R-EQUAL* generates more reservations initially in comparison to *R-MIN* variants and thus decreases schedulability.

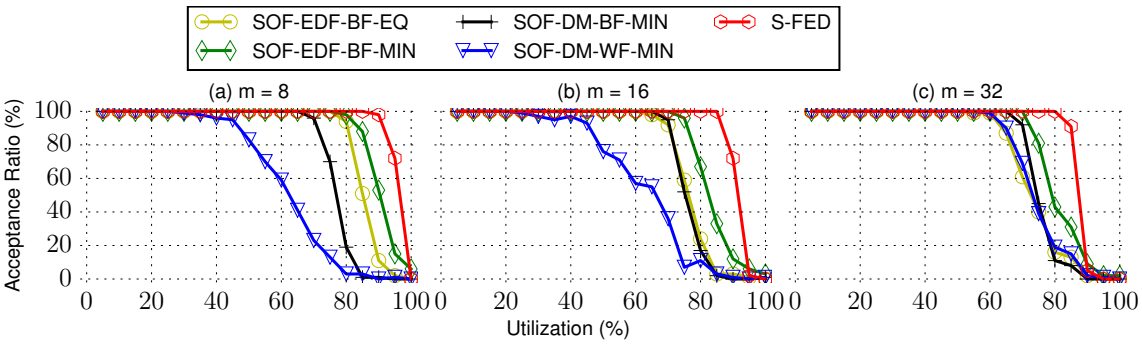


Figure 7.3: Acceptance ratio for normalized utilizations in five percent steps for implicit-deadline DAG task sets on 8, 16, and 32 processors, respectively. The periods are drawn uniformly from $(0, 100]$, the critical-path length is drawn uniformly from $[0.1D_i, 0.3D_i]$.

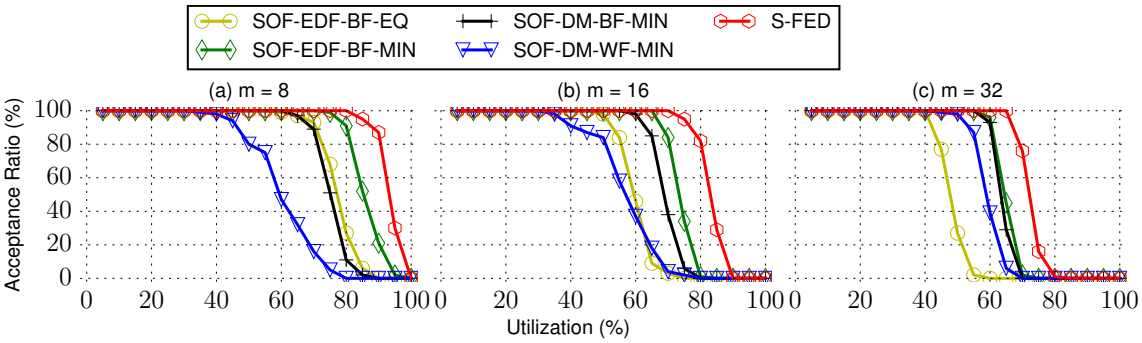


Figure 7.4: Acceptance ratio for normalized utilizations in five percent steps for implicit-deadline DAG task sets on 8, 16, and 32 processors, respectively. The periods are drawn uniformly from $(0, 100]$, the critical-path length is drawn uniformly from $[0.3T_i, 0.6T_i]$.

Experimental Results on Constrained-Deadline Task Sets

For constrained-deadline DAG task sets, three different experiments were conducted using uniformly drawn critical-path lengths and deadlines from varying intervals to model different degrees of parallelized DAG tasks and density to utilization ratios.

In the **first experiment** as illustrated in Fig. 7.6, the deadline is drawn uniformly from the interval $(0, 0.5T_i]$ and the critical-path length is drawn uniformly from $(0, 0.5D_i]$. This setting should represent highly parallelized DAG structures, with high density but low utilization.

In the **second experiment** as illustrated in Fig. 7.7 represents DAG tasks with critical-path lengths drawn uniformly from $(0, 0.5D_i]$ and deadlines drawn from $(0.8T_i, T_i]$ to represent highly parallelized DAG tasks with densities that closely resemble the task's utilizations.

Lastly, in the **third experiment** shown in Fig. 7.8, critical-path lengths are drawn from $[0.4D_i, 0.7D_i]$ and deadlines drawn from $[0.1T_i, T_i]$ thus representing medium parallelized DAG tasks with a wide range of density to utilization ratios.

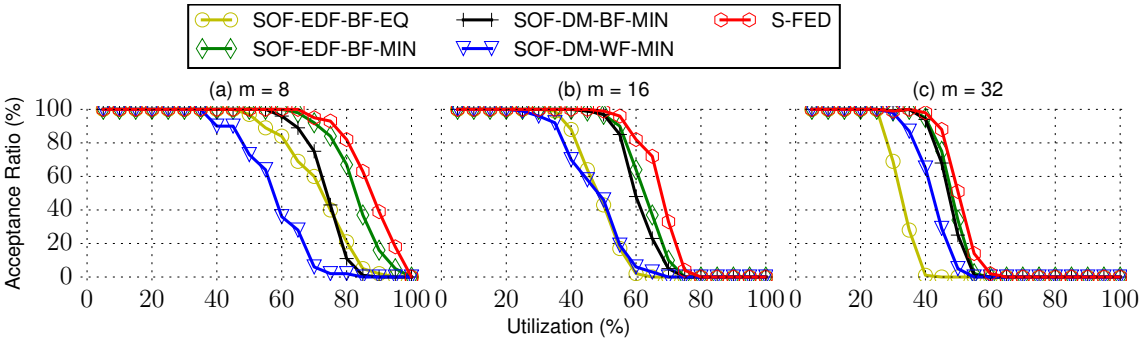


Figure 7.5: Acceptance ratio for implicit-deadline DAG task sets with normalized utilizations in five percent steps on 8, 16, and 32 processors, respectively. The periods are drawn uniformly from $(0, 100]$ and the critical-path length is drawn uniformly from $(0.6T_i, 0.9T_i]$.

It can be seen, that the best *SOF* variant constantly dominates the semi-federated scheduling approach *S-FED* in all tested experiments. Further, with increasing differences between the task's densities and utilizations the dominance of *SOF* grows. Surprisingly, the performance of the EDF and DM variants align in case of constrained-deadline task sets. In general and as demonstrated in the third experiment, the variants *SOF-DM-BF-MIN* and *SOF-EDF-BF-MIN* slightly outperform *S-FED* for 8, 16, and 32 processors whereas the dominance decreases with the number of processors. The *S-FED* approach is especially sensitive to small L_i/D_i and D_i/T_i ratios and performs much worse, which is further illustrated in an extreme-case shown in the first experiment.

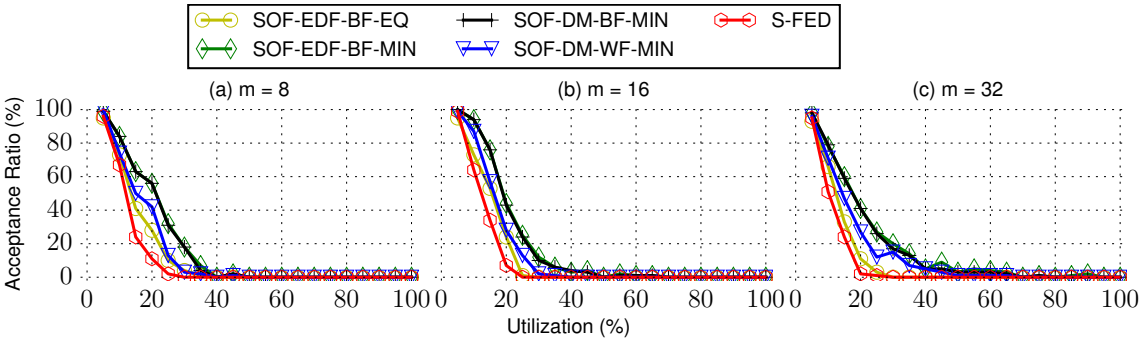


Figure 7.6: Acceptance ratio for normalized utilizations in five percent steps for constrained-deadline DAG task sets on 8, 16, and 32 processors respectively. The periods are drawn uniformly from $(0, 100]$, the deadline is drawn uniformly in $(0, 0.5T_i]$ and the critical-path length is drawn uniformly from $(0, 0.5D_i]$.

Experimental Results on Arbitrary-Deadline Task Sets

Finally, for arbitrary-deadline DAG task sets two different experiments were conducted using uniformly drawn critical-path lengths and deadlines from varying intervals to model different degrees of parallelized DAG tasks and density to utilization ratios. In contrast to

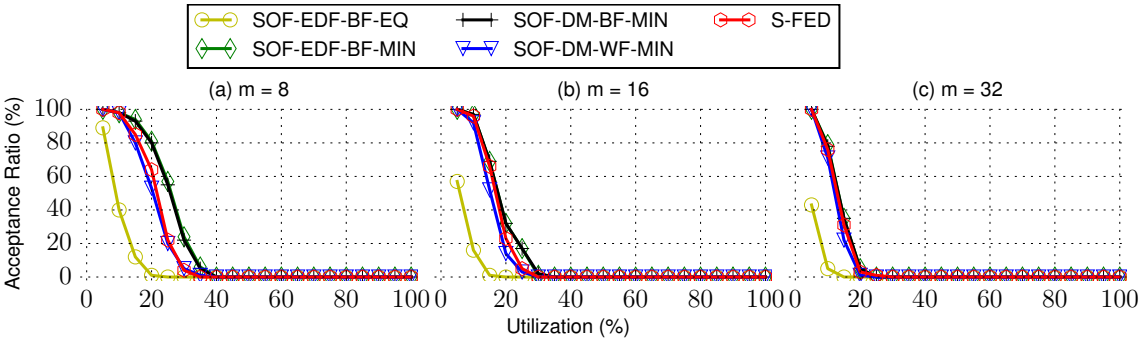


Figure 7.7: Acceptance ratio for normalized utilizations in five percent steps for constrained-deadline DAG task sets on 8, 16, and 32 processors, respectively. The periods are drawn uniformly from $(0, 100]$, the deadline is drawn uniformly in $(0.8T_i, T_i]$ and the critical-path length is drawn uniformly from $(0, 0.5D_i]$.

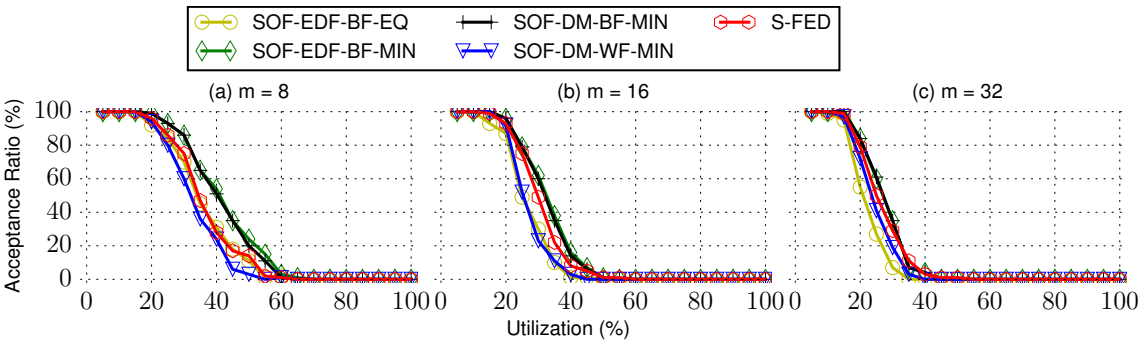


Figure 7.8: Acceptance ratio for normalized utilizations in five percent steps for constrained-deadline DAG task sets on 8, 16, and 32 processors, respectively. The periods are drawn uniformly from $(0, 100]$, the deadline is drawn uniformly in $(0.1T_i, T_i]$ and the critical-path length is drawn uniformly from $(0.4D_i, 0.7D_i]$.

the previous experiments only *SOF* variants are subject to evaluation since *S-FED* does not support arbitrary-deadline DAG task sets.

In the **first experiment** as illustrated in Fig. 7.9, the deadline is drawn uniformly from the interval $(0.1T_i, 10T_i]$ and the critical-path length is drawn uniformly from $(0.4D_i, 0.7D_i]$. This setting should represent a wide range of application characteristics with respect to density to utilization ratios while representing a medium degree of parallelism.

In contrast, the **second experiment** as illustrated in Fig. 7.10 represents DAG tasks with critical-path lengths within the interval $(0, 0.7D_i]$ and deadlines drawn uniformly from the interval $(0.5T_i, 2T_i]$. This experimental setup, represents DAG task sets with wide ranges of parallelism and medium density to utilization ratios.

In the first experiment, all *SOF* variants accept all task sets until 50%, 20%, and 10% cutoff utilizations in 8, 16, and 32 processor platforms respectively. The variants *SOF-DM-BF-MIN* and *SOF-EDF-BF-EQ* exhibit the best performance and behave identical. Additionally, it can be observed that *SOF-DM-WF-MIN* demonstrates the worst performance. The second experiment demonstrates similar findings, but the performance of

SOF-EDF-BF-EQ improves with the number of processors and is even the best algorithm for 16 and 32 processors systems.

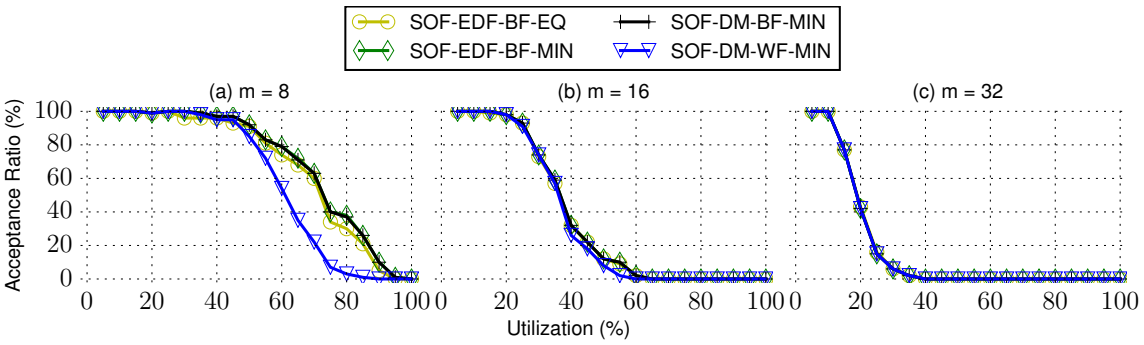


Figure 7.9: Acceptance ratio for normalized utilizations in five percent steps for arbitrary-deadline DAG task sets on 8, 16, and 32 processors respectively. The periods are drawn uniformly from $(0, 100]$, the deadline is drawn uniformly in $(0.1T_i, 10T_i]$ and the critical-path length is drawn uniformly from $(0.4D_i, 0.7D_i]$.

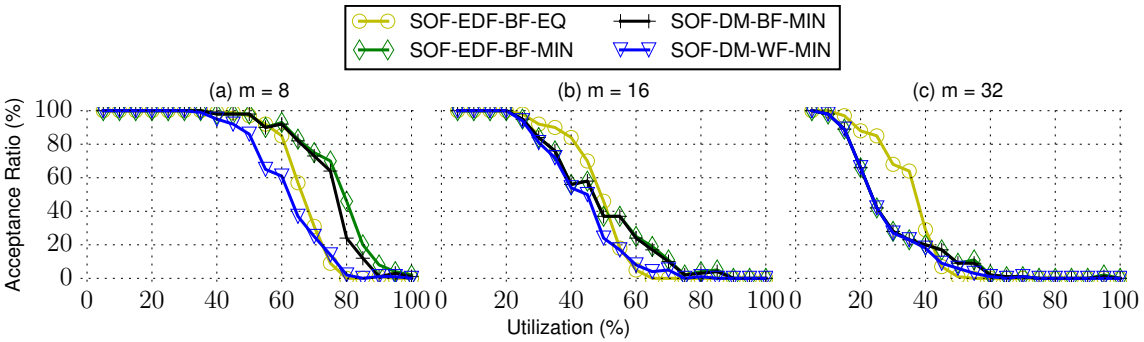


Figure 7.10: Acceptance ratio for normalized utilizations in five percent steps for arbitrary-deadline DAG task sets on 8, 16, and 32 processors, respectively. The periods are drawn uniformly from $(0, 100]$, the deadline is drawn uniformly in $(0.5T_i, 2T_i]$ and the critical-path length is drawn uniformly from $(0, 0.7D_i]$.

7.4 Summary

In summary, the evaluation showcased that reservation-based federated scheduling is competitive with the state-of-the-art of sporadic arbitrary-deadline DAG task scheduling for the parametric model. More precisely, especially in constrained-deadline and arbitrary-deadline cases the reservation-based scheduling demonstrated dominance, robustness with respect to varying parameter ranges and generally good performance.

Despite a slightly dominated acceptance ratio in the case of implicit-deadline DAG task sets, reservation-based federated scheduling exhibited similar behavior except for roughly 10% lowered cutoff utilization in the worst setting (8 processors) that decreased to 0% for 32 processor systems.

8 Conclusion, Related and Future work

This chapter is separated into three sections. In the first Section 8.1 *Conclusion*, the achieved results of this thesis are summarized and evaluated with respect to the motivation of this thesis.
In the following Section 8.2 *Real-Time Operating System Implementation*, an evaluation of an implementation of the proposed reservation-based federated scheduling in a real system is presented.
Lastly, in Section 8.3 *Future Work*, open problems and suggestions for further research are proposed.

8.1 Conclusion

In this master thesis, a novel approach to schedule sporadic arbitrary-deadline DAG task sets was presented. That is, the concept of reservation servers and associated sufficient conditions for the construction of feasible reservation systems have been proposed.
Further, in order to solve the unbounded resource over-provisioning problem in federated scheduling for arbitrary-deadline DAG tasks, several solutions have been proposed. Namely, the *decoupled* algorithms *R-EQUAL* as well as the *coupled* algorithm *Split-On-Fail* that uses *R-EQUAL* for initialization. Additionally, heuristically good algorithms such as *R-MIN* and the *coupled* algorithm *Split-On-Fail* that uses *R-MIN* for initialization have been proposed and demonstrated to perform well.
More precisely, it could be shown that the proposed reservation-based federated scheduling approach is practically competitive with the state of the art for sporadic parametric DAG task sets by comparison of acceptance ratios of synthetically generated task sets.
That is, reservation-based federated scheduling is competitive for scheduling implicit-deadline DAG task sets as well as dominant for constrained-deadline and arbitrary-deadline DAG task sets in terms of acceptance ratios.
The performance of federated scheduling and semi-federated scheduling is notably reduced for constrained-deadline DAG task sets, which is due to the following reasons:

a) Tasks that exhibit high densities require a large number of processors in order to guarantee sufficient service even in peak workload conditions.

b) Tasks that have low utilization, thus rarely use the allocated resources.

To that end, reservation-based federated scheduling demonstrated better resource efficiency for these class of task sets.

In addition, it could be shown that reservation-based federated scheduling admits a constant speedup factor of at most $3 + 2\sqrt{2}$ by using partitioned or global scheduling of a specific set of reservation servers.

In order to asses the practical feasibility e.g., design concepts to realize reservation-based federated scheduling in real-time operating systems, a prototype was designed and realized in a multiprocessor simulation framework.

It could be demonstrated that the concept to implement partitioned scheduling of reservation servers and the subsequent serving of belonging DAG tasks is feasible.

Further testings suggested that the overall number of preemptions is fairly low. It should be noted however, that important overheads e.g., inter-processor-interrupts or the spin-locking of ready-queues was not subject to the implementation and thus evaluation.

8.2 Real-Time Operating System Implementation

During the execution of this master thesis, a collaboration with with Prof. Dr. Jing Li from New Jersey Institute of Technology, USA [1] was done in order to assess the practicability of the proposed reservation-based federated scheduling approach in a real-time operating system. For the completeness of the master thesis, the implementation and evaluation of Prof. Dr. Li are therefore included here, to demonstrate the effectiveness of the proposed approach. Note that the implementation here is solely authored by Prof. Dr. Li.

Further, the following experimental results together with the theoretical analyses presented in this master thesis were the main material of a research paper submitted to Euromicro Conference on Real-Time Systems (ECRTS) 2018.

On the basis of the presented algorithms to calculate feasible systems of reservation servers, Li realized global deadline-monotonic scheduling for scheduling reservation servers. The underlying real-time operating system that was used in the experiments is a Linux kernel with the PREEMPT_RT kernel patch.

The prototype platform that is used for evaluations is a modified federated scheduling platform for OpenMP programs, namely RTCG as detailed in Li et al. [47].

Li identifies the main differences between the reservation-based federated scheduling using global deadline-monotonic scheduling (to schedule the reservation servers) and the original federated scheduling, that are relevant to the implementation to include:

[1] jingli@njit.edu

- To assign deadline-monotonic priorities to the parallel threads of a task.
- To generate different numbers of threads for a task, where each thread corresponds to one reservation server in reservation-based federated scheduling, in contrast to one dedicated core in the original federated scheduling.
- The global execution of the threads of a task, instead of dedicated core assignment using federated scheduling.

Experimental Setup and Task Set Generation

The empirical evaluations were conducted on a 48-core machine composed of 4 AMD Opteron 6168 processors. Moreover, one processor was reserved, i.e., 12 cores, for system tasks, leaving 36 experimental processing cores effectively. Furthermore, single socket 12-core experiments and multi-socket 36-core experiments were conducted.

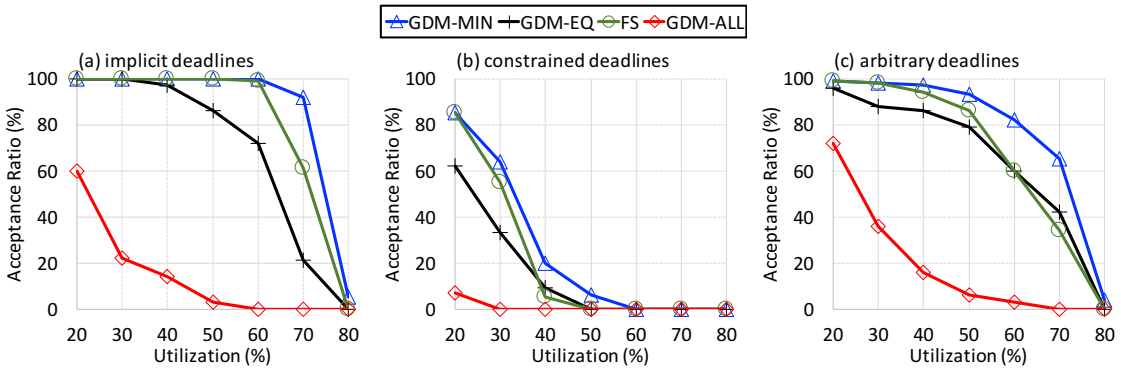


Figure 8.1: Acceptance ratio for normalized utilizations in 10 percent steps for task sets on 12 cores with implicit, constrained and arbitrary deadlines, respectively. For each normalized utilization 100 task sets each with 10 tasks are generated. The maximum critical-path length for each task is drawn uniformly from $(0.6T_i, 0.9T_i]$ for implicit and arbitrary deadlines, and is drawn from $(0.4D_i, 0.7D_i]$ for constrained deadlines. The deadline is drawn uniformly in $(0.1T_i, 10T_i]$ for arbitrary deadlines and in $(0.1T_i, 1T_i]$ for constrained deadlines.

The evaluated task sets were generated using the approach presented in Section 7.2.2 *Parametric DAG Task Generation* where each task τ_i is characterized by $\tau_i = (C_i, L_i, D_i, T_i)$. It is to emphasize that all generated parameters that are used in the presented empirical experiments use the unit milliseconds (ms).

In addition, for each task a period is randomly selected from $4ms, 8ms, 16ms, 32ms, 64ms$ and $128ms$ to form task sets with harmonic periods.

Furthermore, for each task with the generated parameters, a regular OpenMP program is implemented, consisting of sequences of parallel for-loops of varying lengths and numbers of iterations where the number of for-loops was uniformly chosen from $[1, 10]$ and the number of iterations was randomly generated by a lognormal distribution with a mean of

3.5 and variance of 7.6. Additionally, each task set was run for 100 hyper-periods in all presented experiments.

The reservation-based federated scheduling that uses global DM in the *R-MIN* and *R-EQUAL* variants (namely GDM-MIN and *GDM-EQ*, respectively) are compared against an implementation of global DM without any reservation (*GDM-ALL*).

For experiments with implicit deadlines, the original federated scheduling i.e., RTCG platform (*FS*) was additionally used for comparison. To allow for a fair comparison for experiments with constrained and arbitrary deadlines, the federated scheduling approach was modified to use $\min\{D_i, T_i\}$ instead of D_i to calculate the core allocation for each task.

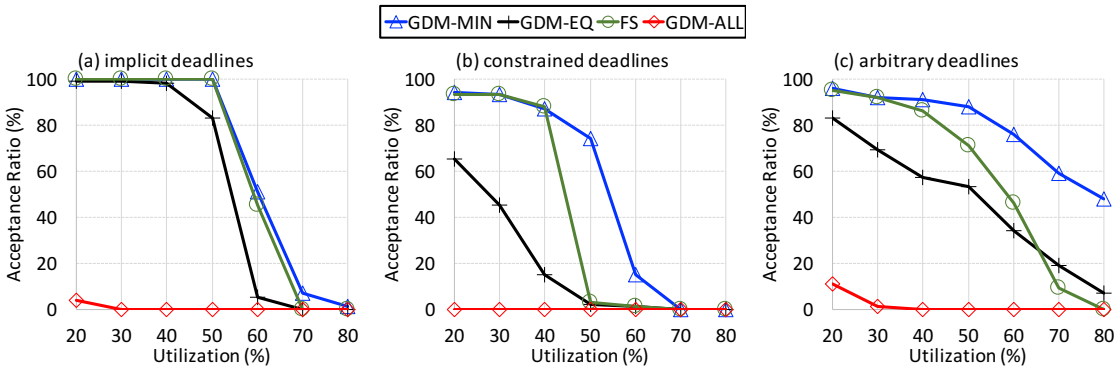


Figure 8.2: Acceptance ratio for normalized utilizations in 10 percent steps for task sets on 36 cores with implicit, constrained and arbitrary deadlines, respectively. For each normalized utilization 100 task sets each with 20 tasks are generated. The maximum critical-path length for each task is drawn uniformly from $(0.6T_i, 0.9T_i]$ for implicit and arbitrary deadlines, and is drawn from $(0.4D_i, 0.7D_i]$ for constrained deadlines. The deadline is drawn uniformly in $(0.1T_i, 10T_i]$ for arbitrary deadlines and in $(0.5T_i, 1T_i]$ for constrained deadlines.

Experimental Results

The experiments on 12 and 36 cores are shown in Fig. 8.1 and 8.2. In all the different settings, it can be observed that *GDM-MIN* outperforms *FS* and *FS* is comparable or even better than *GDM-EQ*.

Additionally, *GDM-EQ* outperforms *FS* since without the dedicated core allocation a core will not idle as long as there are some unfinished jobs pending.

Therefore, *GDM-MIN* can more efficiently utilize the multiprocessor platform.

Finally, *GDM-ALL* performs significantly worse, which underlines the necessity and benefit of calculating the proper reservation demands.

8.3 Future Work

In future work, new approaches are required to reduce the pessimism of the required resources or services. Unfortunately, as is illustrated in the proof for the sufficient service (cf. Section 4.2 *Sufficient Conditions for Reservation Server Design*), the worst-case service pattern matches the worst-case DAG structure i.e., the service pattern (schedule) may degrade any DAG structure into the worst-case DAG structure. Static schedules may alleviate this problem due to the possibility of fitting the service to the demands of the DAG tasks more precisely. Another possible approach to reduce the resource waste is to incorporate *self-suspension* behavior into the reservation server model and thus analysis, where each reservation may suspend for at most L_i amount of time. This might help in the illustrated case, that $m_i - 1$ reservation servers are spinning for at most L_i amount of time.

Appendix

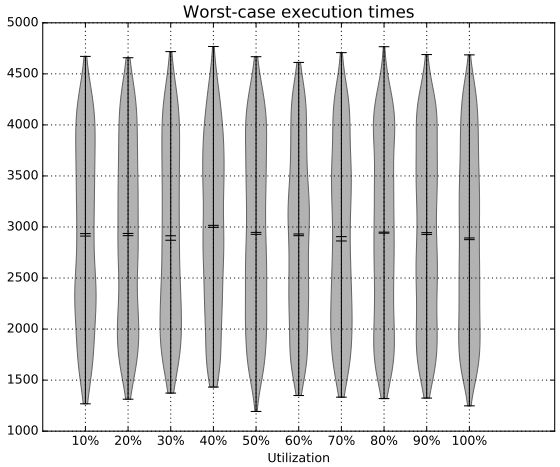
```
def compute_longest(self):
  cost = np.zeros((self.num_nodes, self.num_nodes))
  for k in xrange(self.num_nodes):
    for i in xrange(self.num_nodes):
      if k == 0:
        cost[i,:] = (self.nodes[i] + self.nodes[:])*self.edges[i,:]
        cost[i,cost[i,:] == 0] = -np.inf
      else:
        cost[i,:] = maximum(cost[i,:],cost[i,k]+cost[k,:]-self.nodes[k])
  return max(max(self.nodes), cost.max())
```

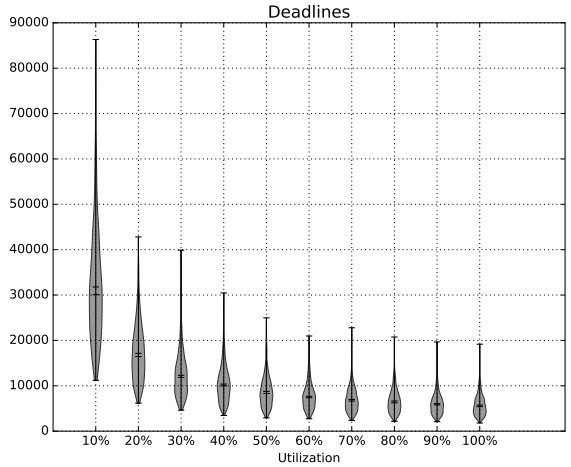
Listing 1: Optimized implementation of Floyd and Warshall.

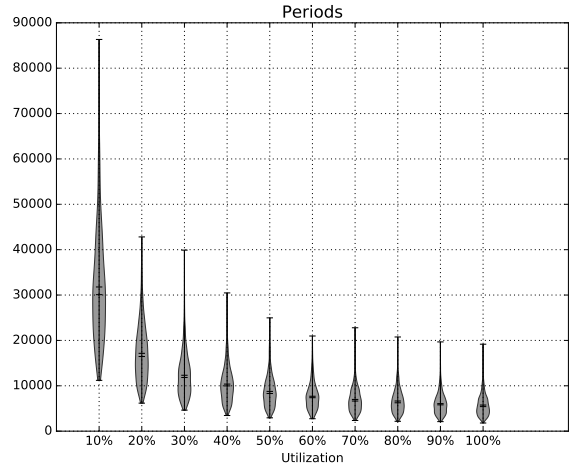
```
def wcrt_list_sched(nodes, edges, processors = 1):
  edge = edges.copy()
  wcet = list(nodes)
  wcrt = [0 for i in range(processors)]
  finished = list(nodes)
  released = [0 for i in range(len(wcet))]
  ready_list = deque()
  for k in reversed(range(len(wcet))):
    if max(edge[i,k] for i in range(len(wcet))) == 0:
      ready_list.append(k)
  while ready_list:
    task = ready_list.popleft()
    selected = np.argmin(wcrt)
    wcrt[selected] = max(wcrt[selected], released[task]) + wcet[task]
    finished[task] = wcrt[selected]

    for j in range(len(wcet)):
      if edge[task, j] == 1:
        edge[task, j] = 0
        released[j] = max(released[j], finished[task])
        if max(edge[k, j] for k in range(len(wcet))) == 0:
          ready_list.append(j)
  return max(wcrt)
```

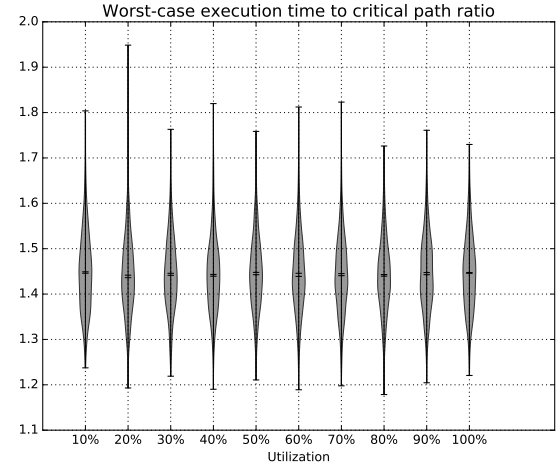
Listing 2: Construct a list-schedule of a given DAG task for the specified number of processors.

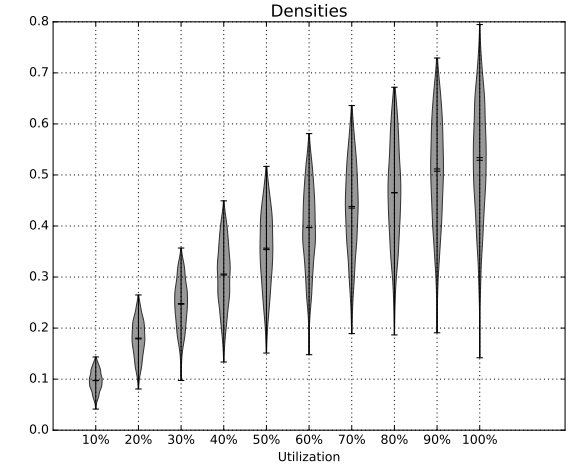
DAG task set statistics (implicit-deadline, 8 processors, P_e = 0.60, P_h = 0.10, P_ℓ = 0.90)

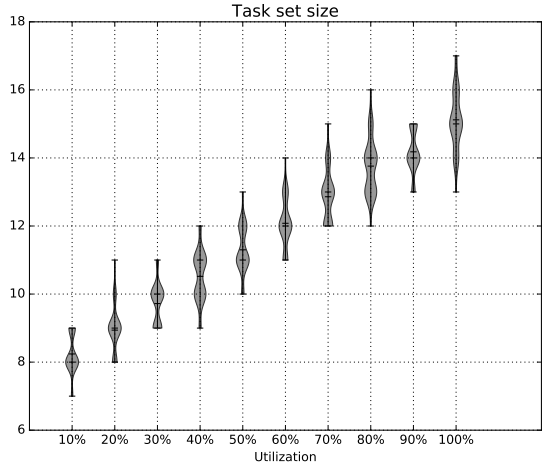




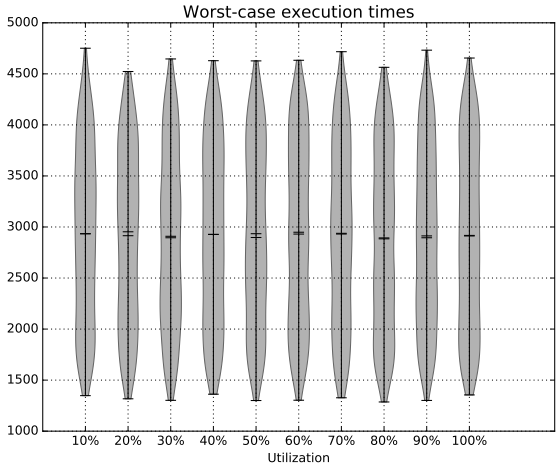


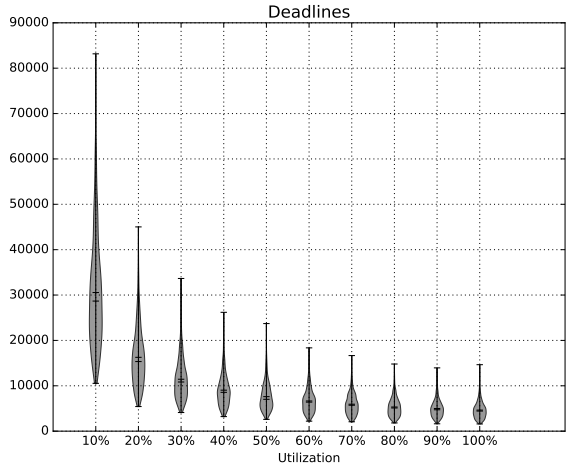


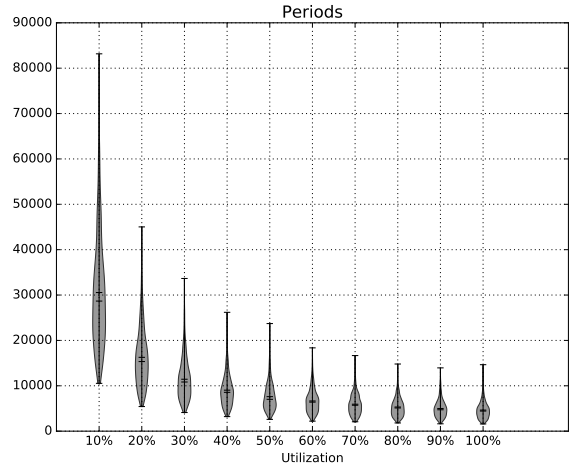


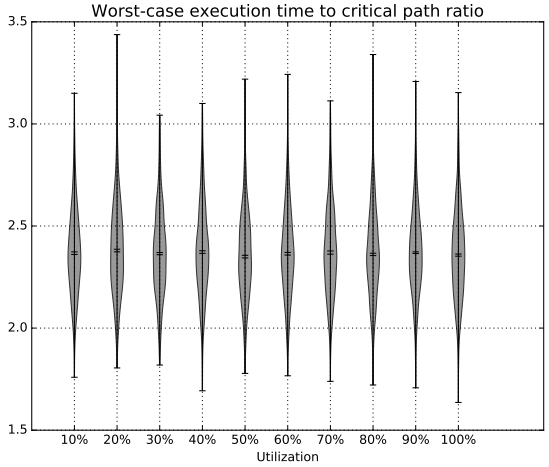


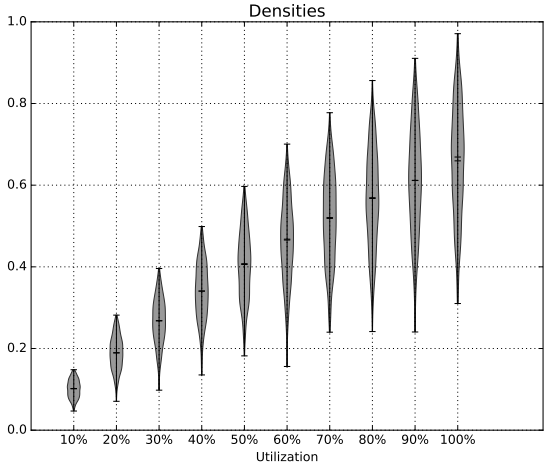
DAG task set statistics (implicit-deadline, 8 processors, P_e = 0.30, P_h = 0.20, P_ℓ = 0.80)

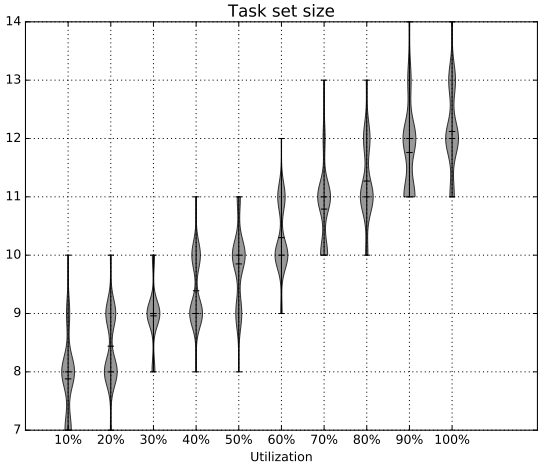




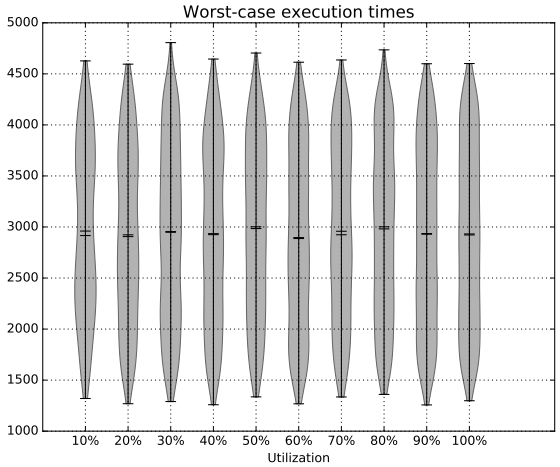


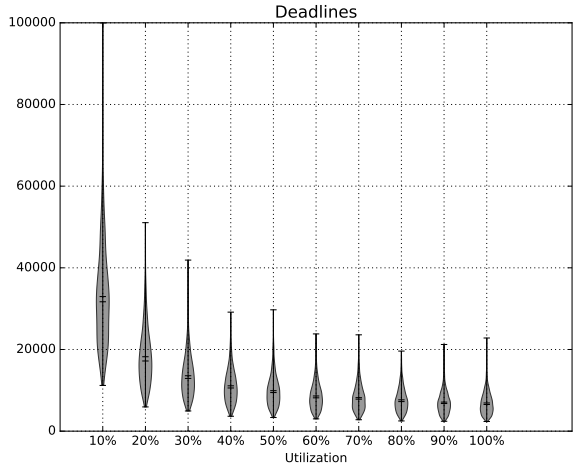


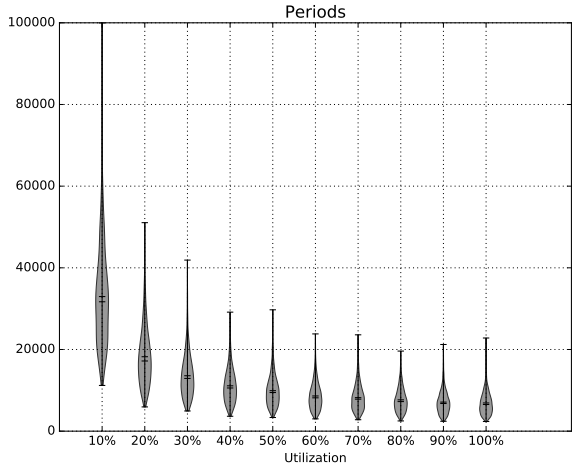


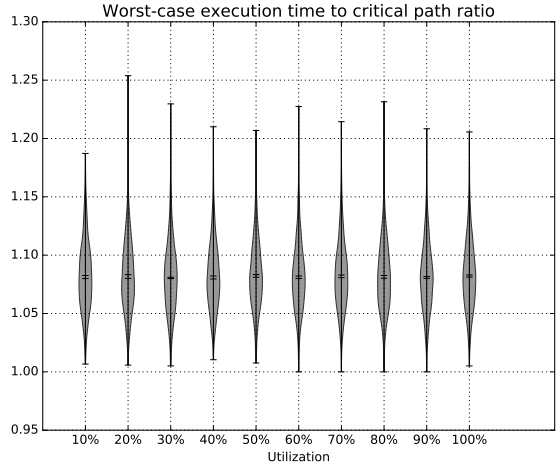


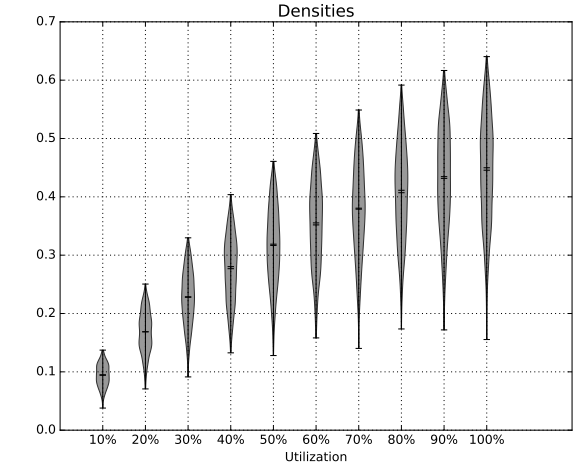
DAG task set statistics (implicit-deadline, 8 processors, P_e = 0.90, P_h = 0.20, P_ℓ = 0.80)

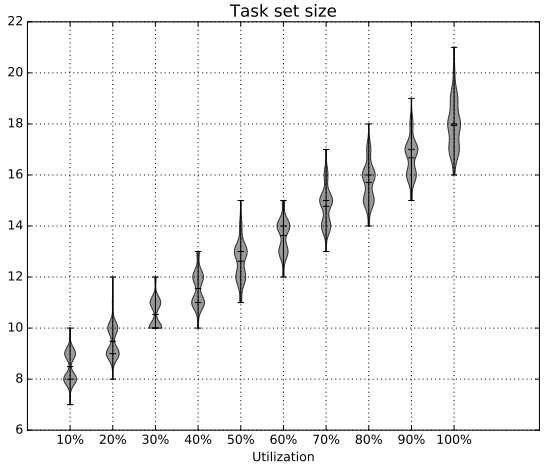




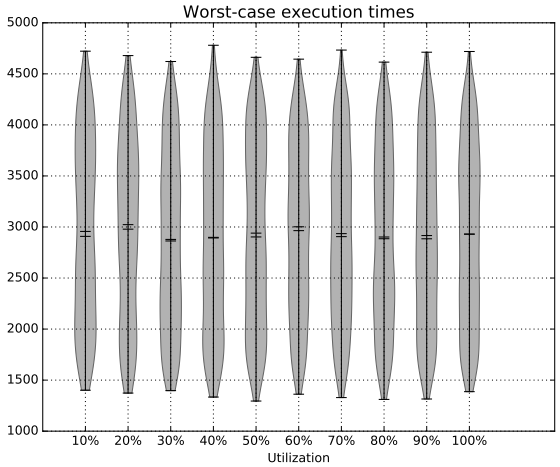


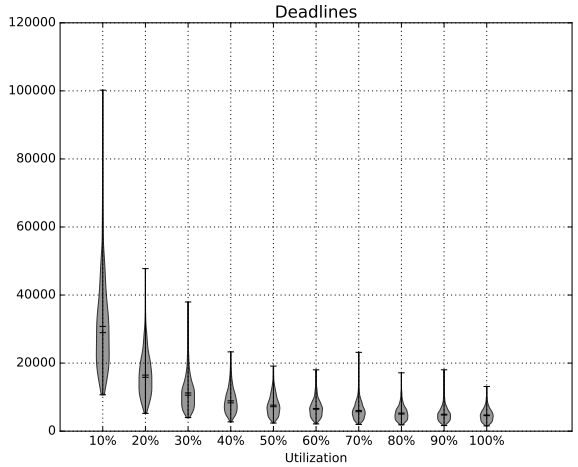


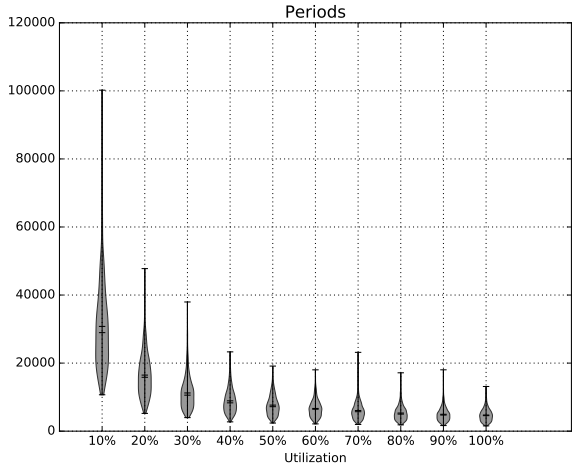


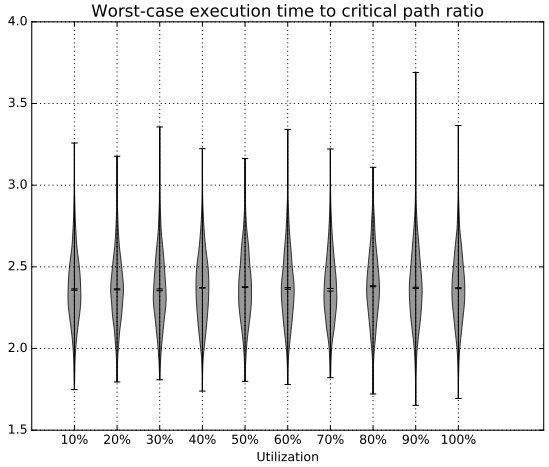


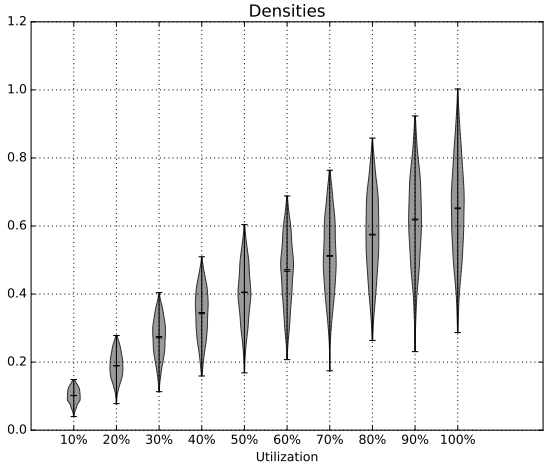
DAG task set statistics (implicit-deadline, 8 processors, P_e = 0.30, P_h = 0.30, P_ℓ = 0.70)

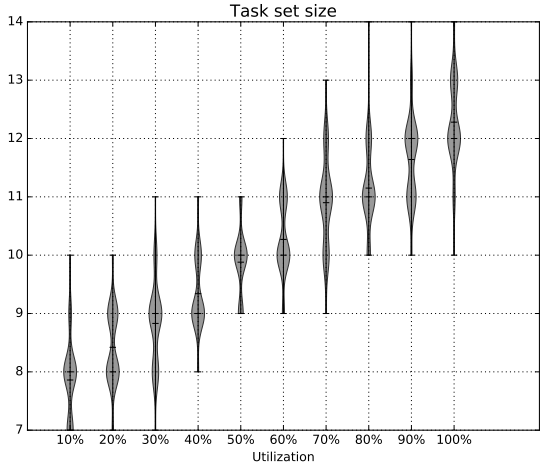




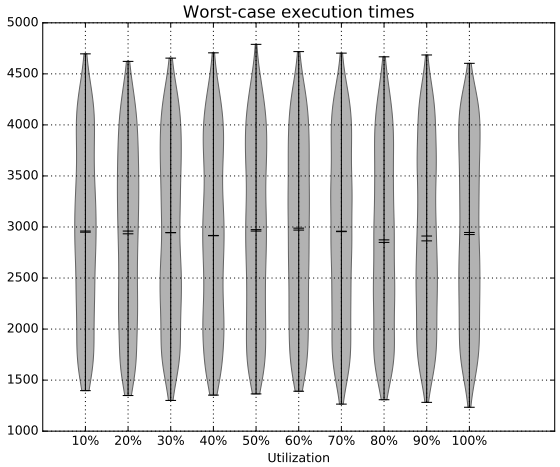


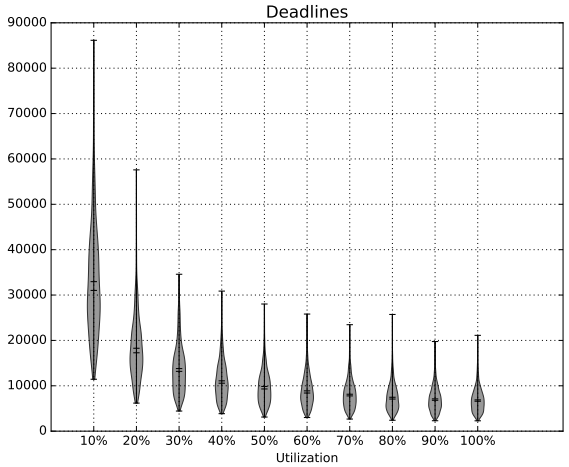


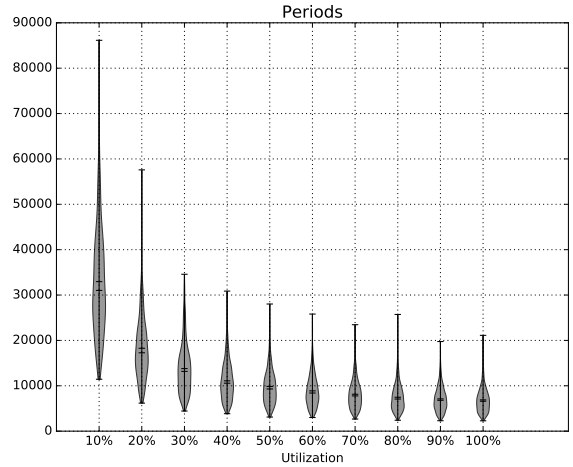


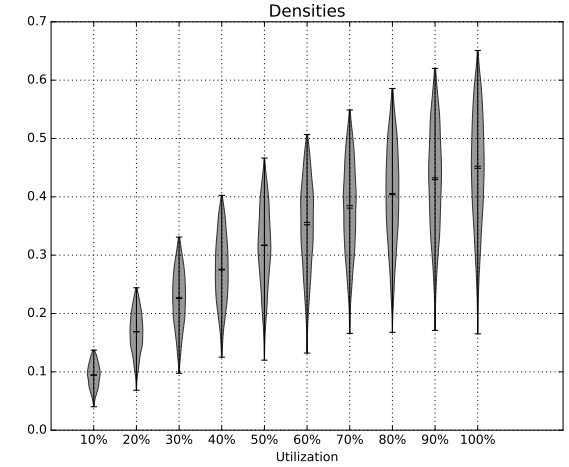


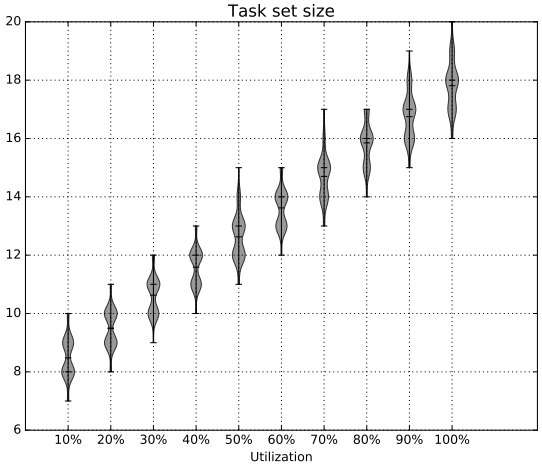
DAG task set statistics (implicit-deadline, 8 processors, P_e = 0.90, P_h = 0.30, P_ℓ = 0.70)



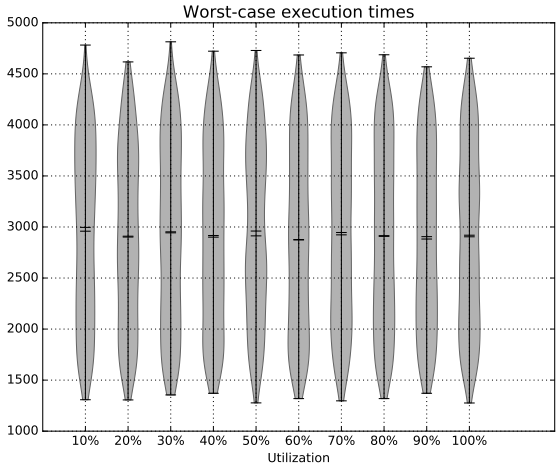


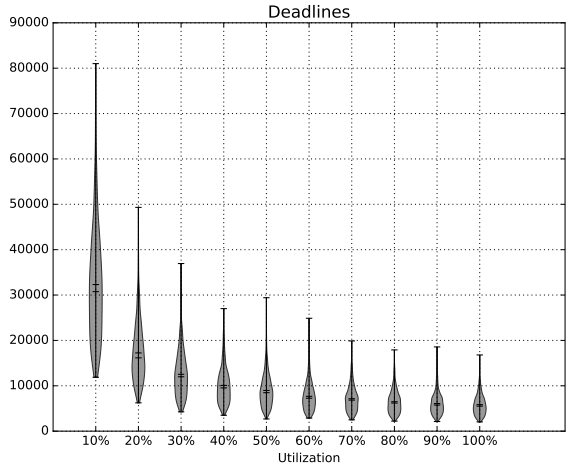


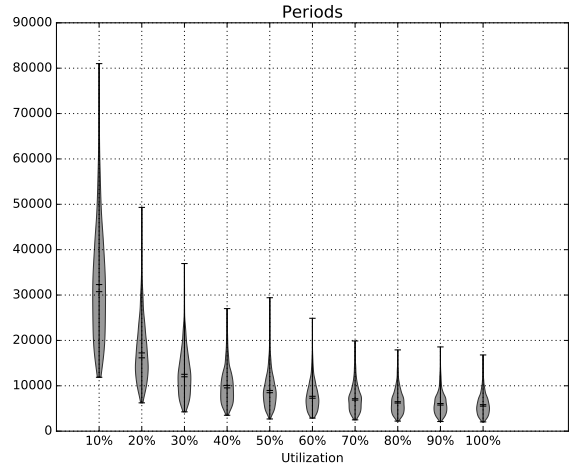


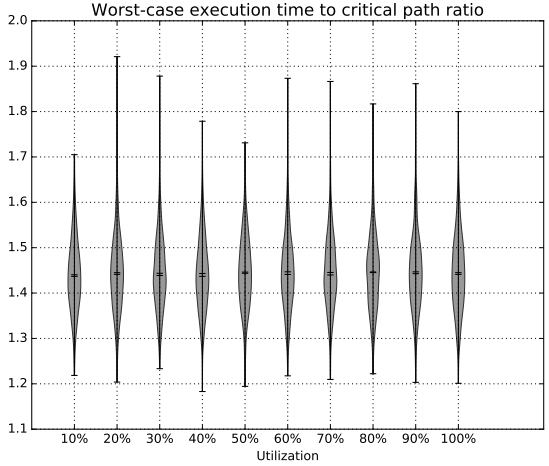


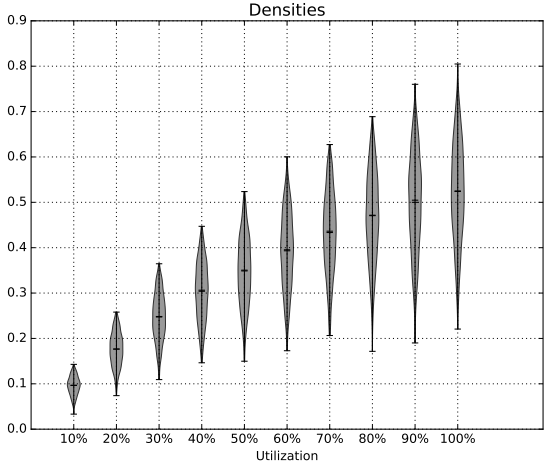
DAG task set statistics (implicit-deadline, 8 processors, P_e = 0.60, P_h = 0.20, P_ℓ = 0.80)

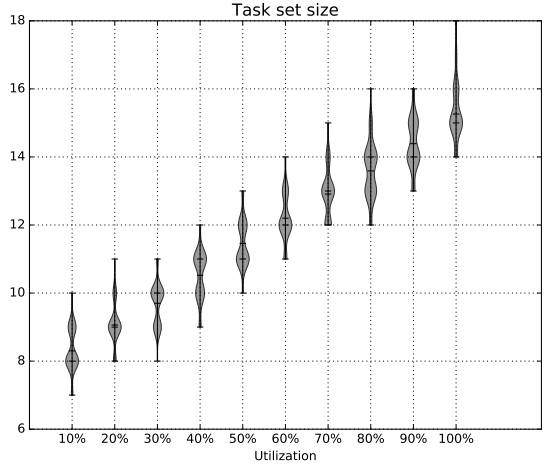


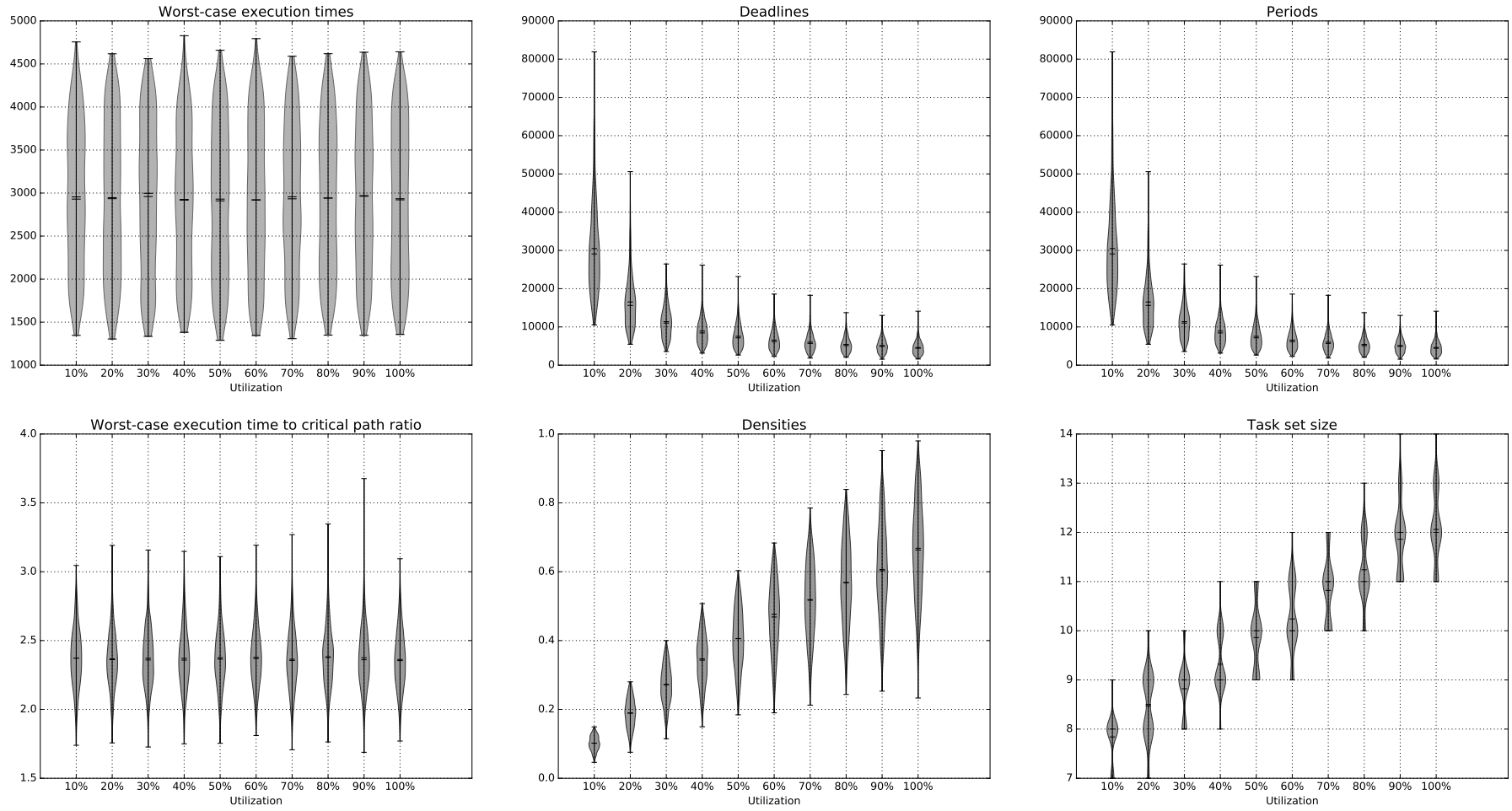




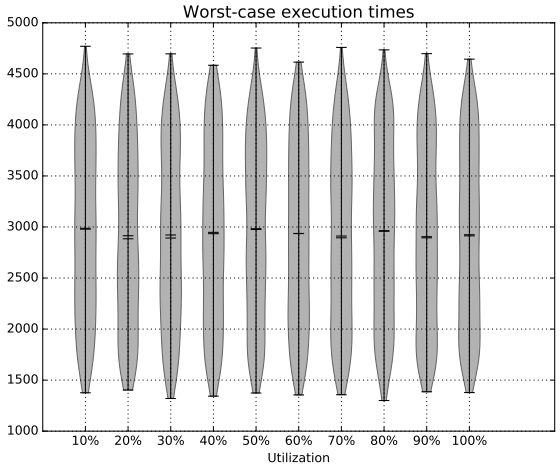


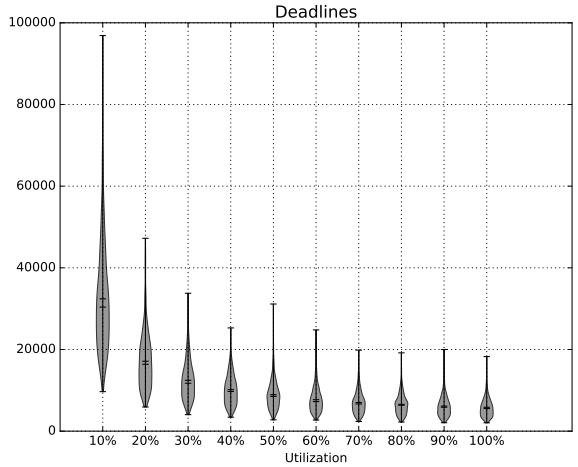


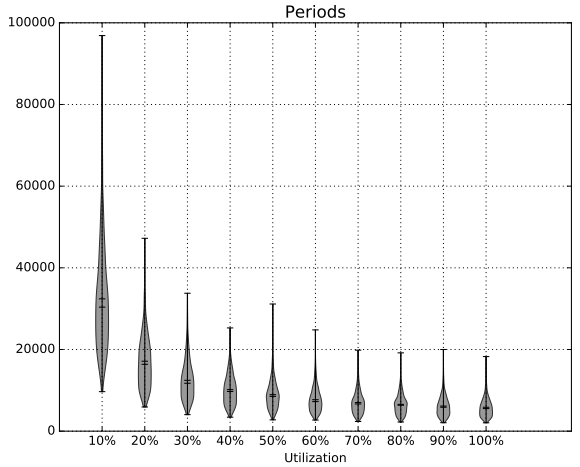


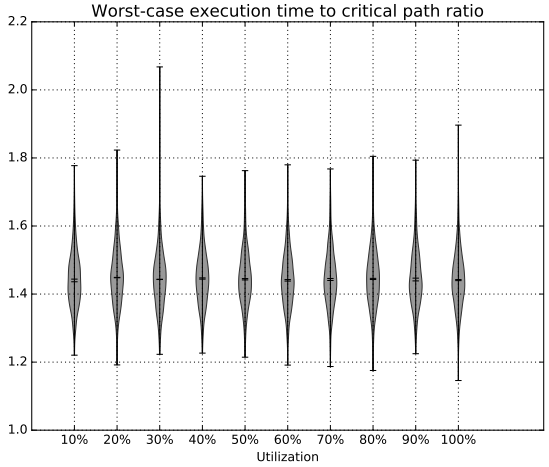

DAG task set statistics (implicit-deadline, 8 processors, P_e = 0.30, P_h = 0.10, P_ℓ = 0.90)
Worst-case execution times
Deadlines
Periods
Worst-case execution time to critical path ratio
Densities
Task set size
Utilization
10%
20%
30%
40%
50%
60%
70%
80%
90%
100%

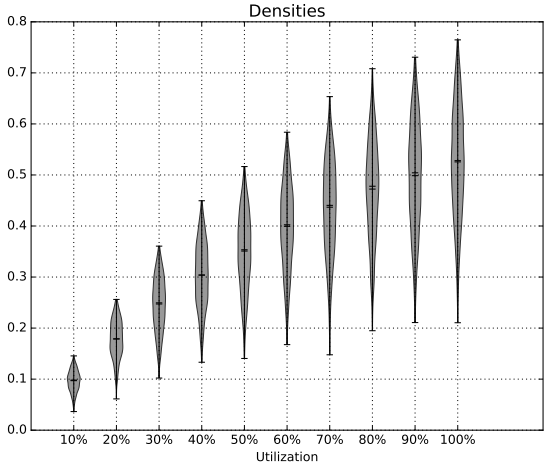
DAG task set statistics (implicit-deadline, 8 processors, P_e = 0.60, P_h = 0.30, P_ℓ = 0.70)

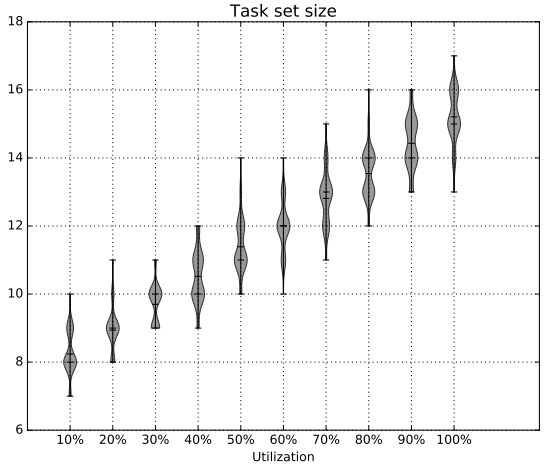




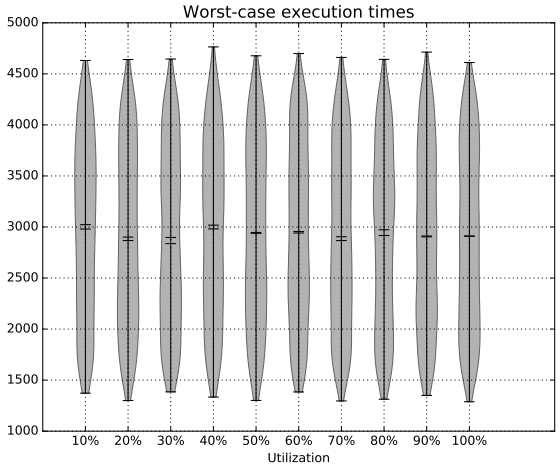


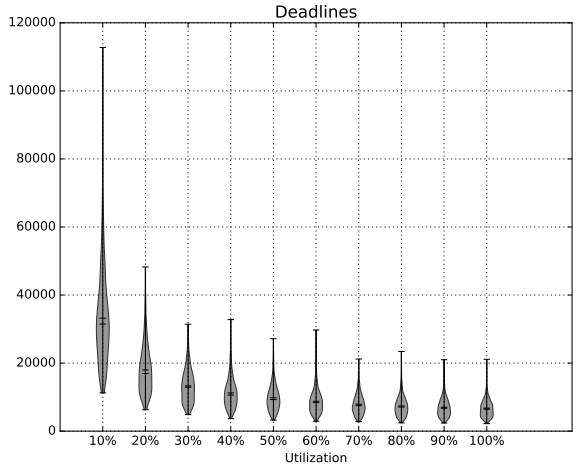


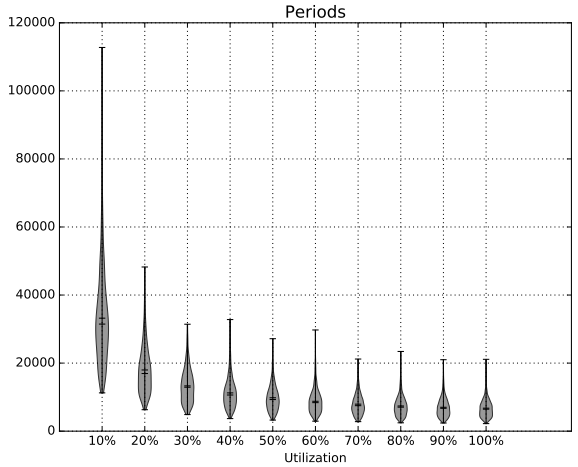


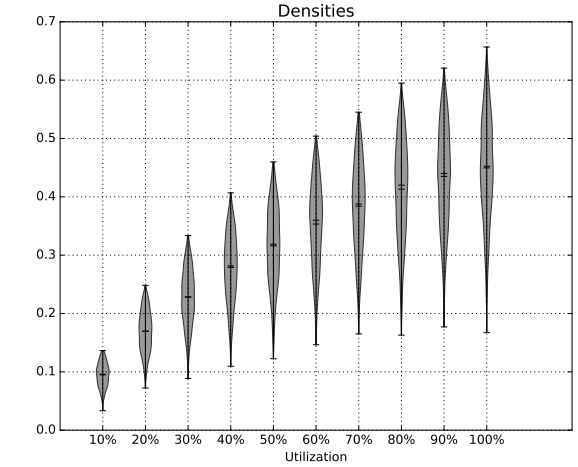


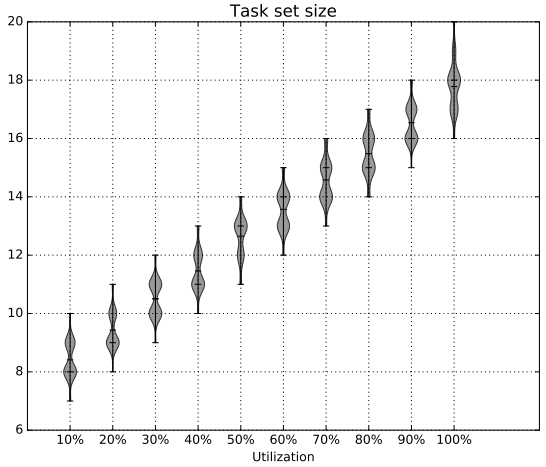
DAG task set statistics (implicit-deadline, 8 processors, P_e = 0.90, P_h = 0.10, P_ℓ = 0.90)



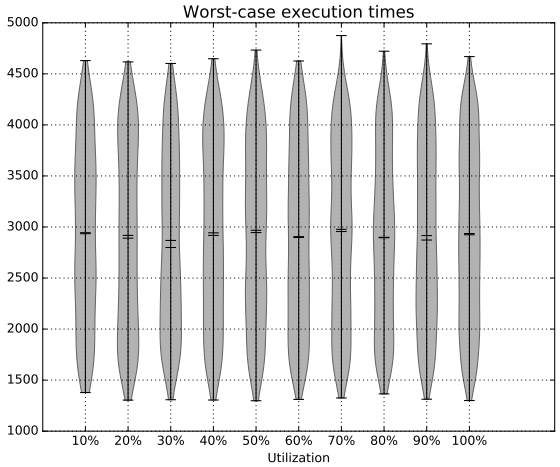


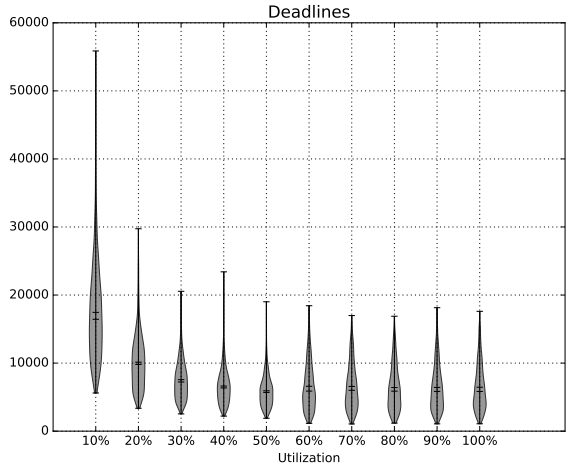


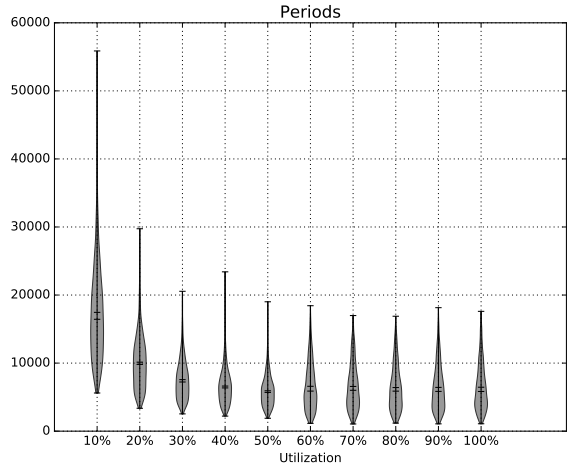


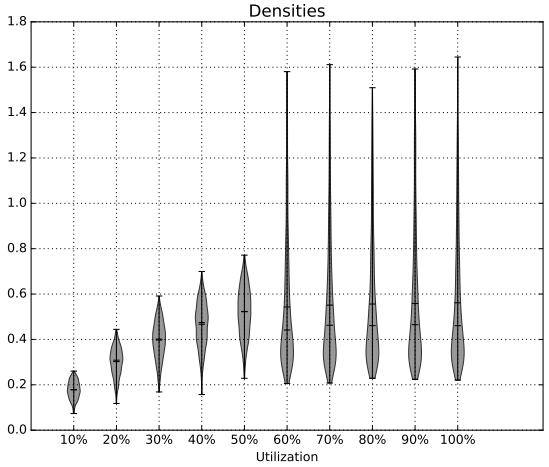


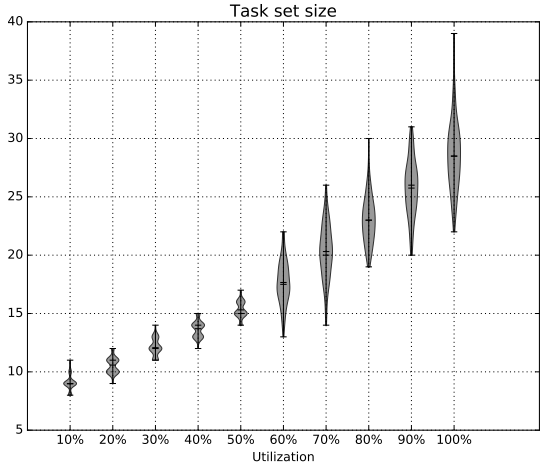
DAG task set statistics (implicit-deadline, 16 processors, P_e = 0.60, P_h = 0.10, P_ℓ = 0.90)



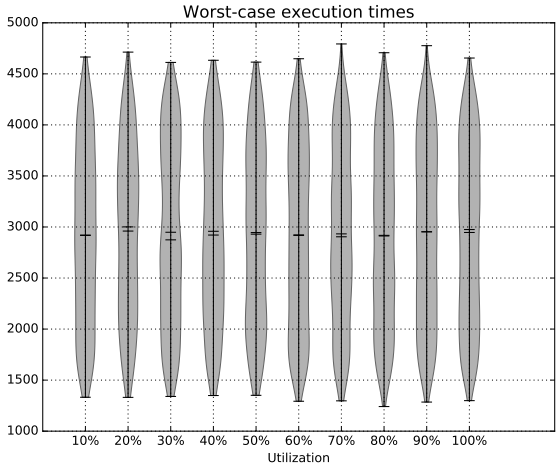


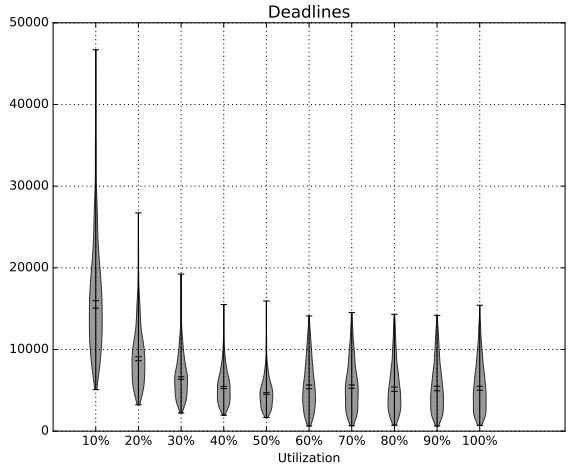


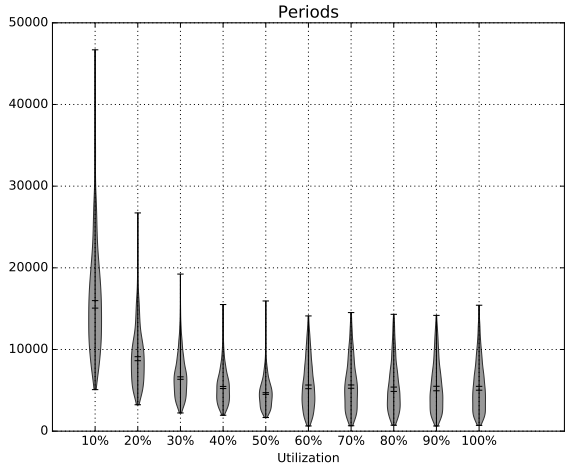


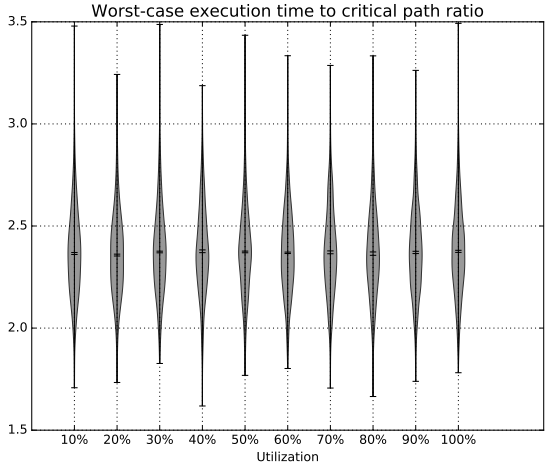


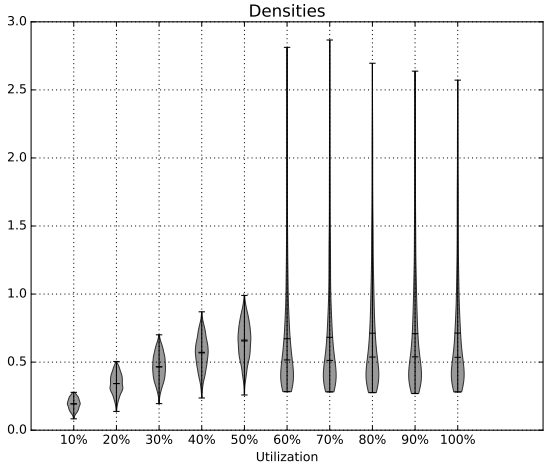
DAG task set statistics (implicit-deadline, 16 processors, P_e = 0.30, P_h = 0.20, P_ℓ = 0.80)

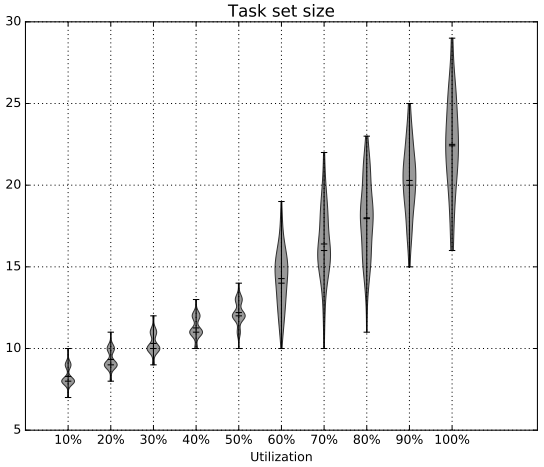




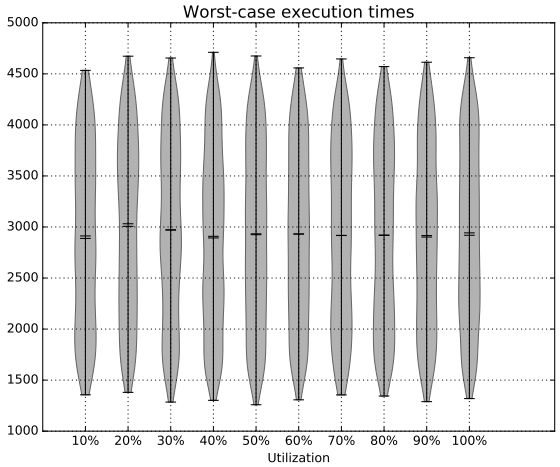


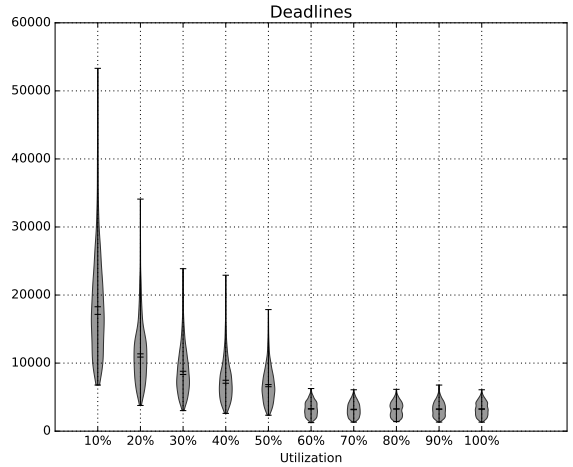


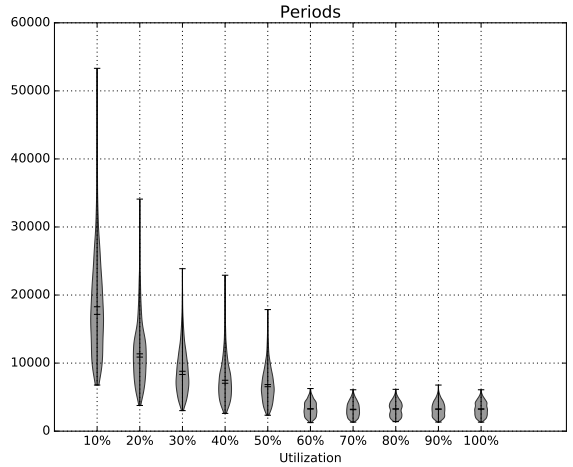


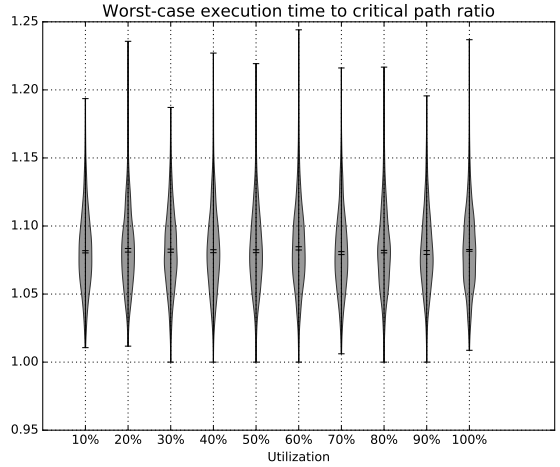


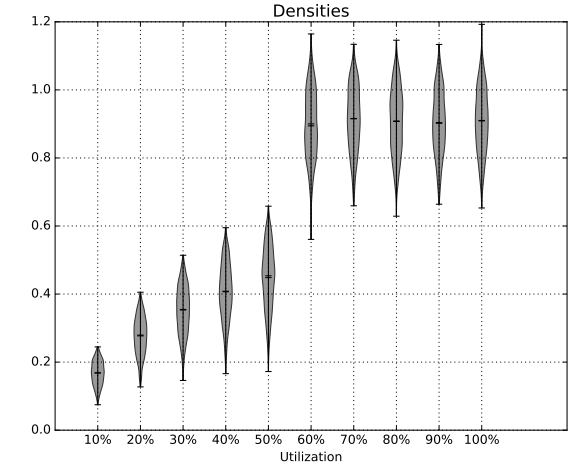
DAG task set statistics (implicit-deadline, 16 processors, P_e = 0.90, P_h = 0.20, P_ℓ = 0.80)

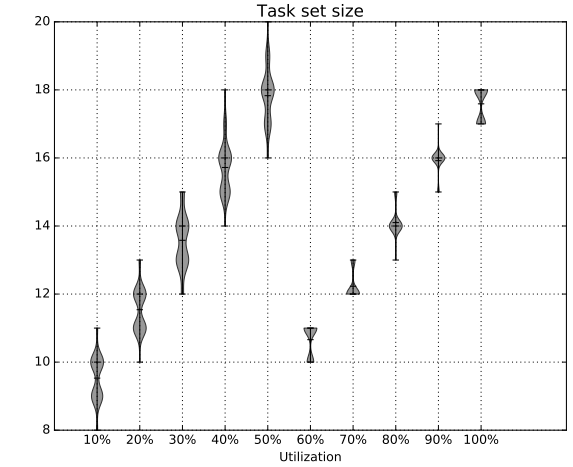




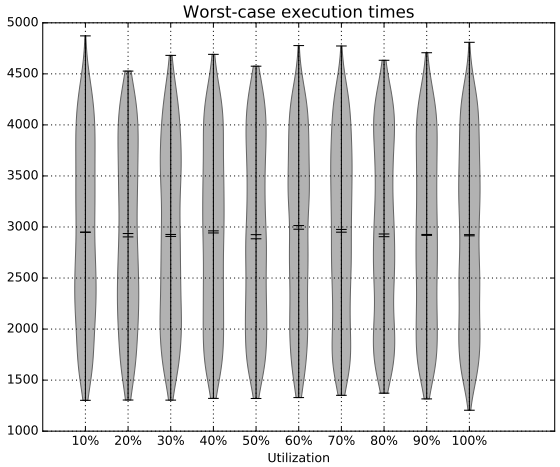


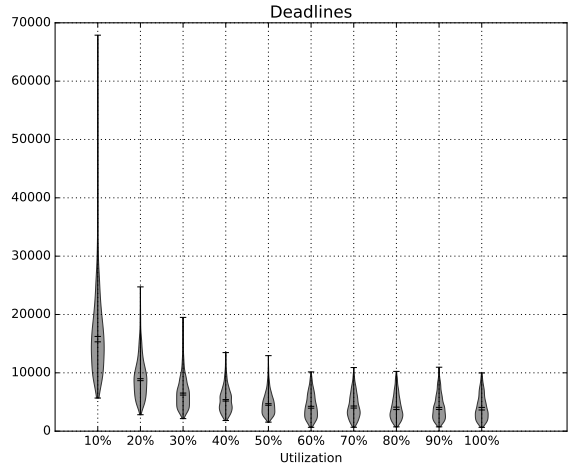


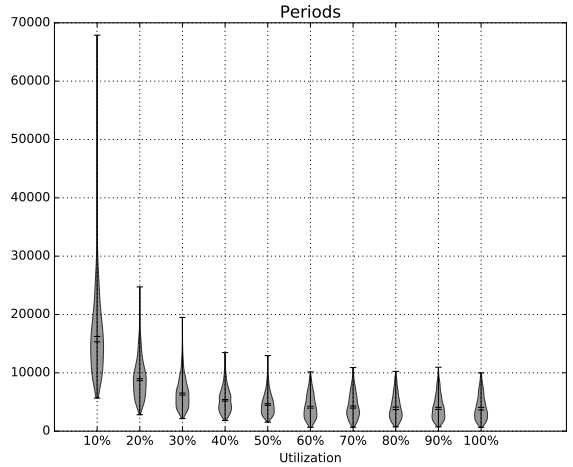


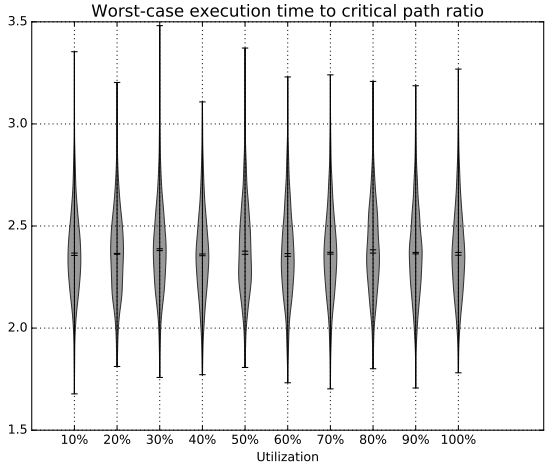


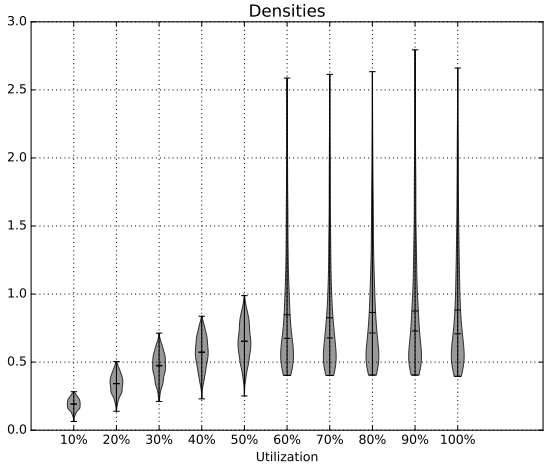
DAG task set statistics (implicit-deadline, 16 processors, P_e = 0.30, P_h = 0.30, P_ℓ = 0.70)

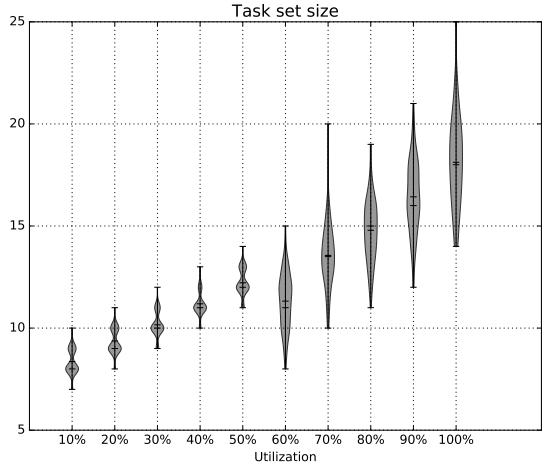




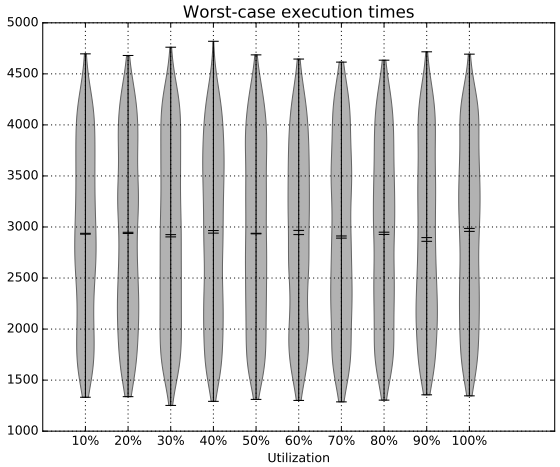


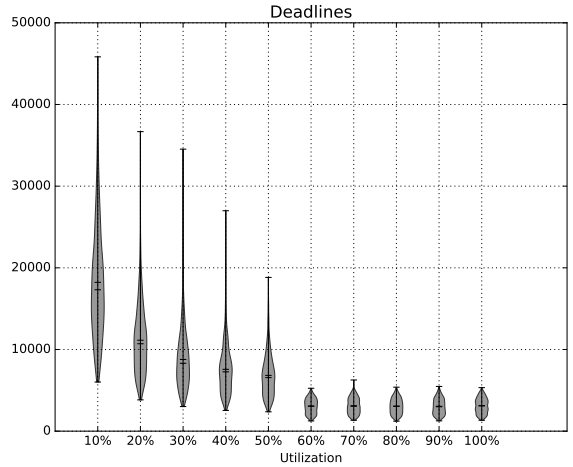


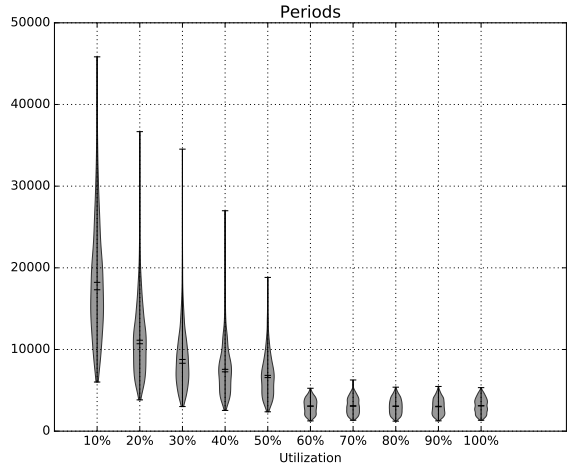


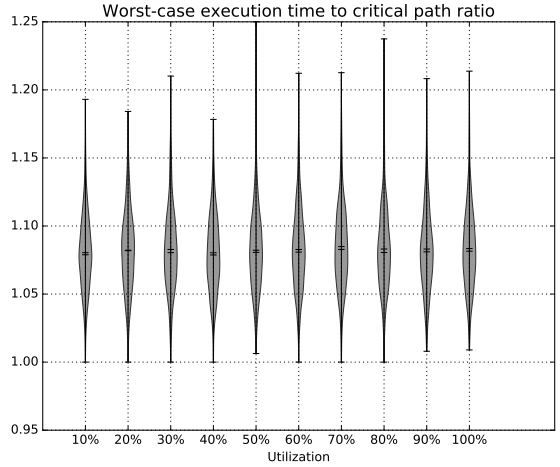


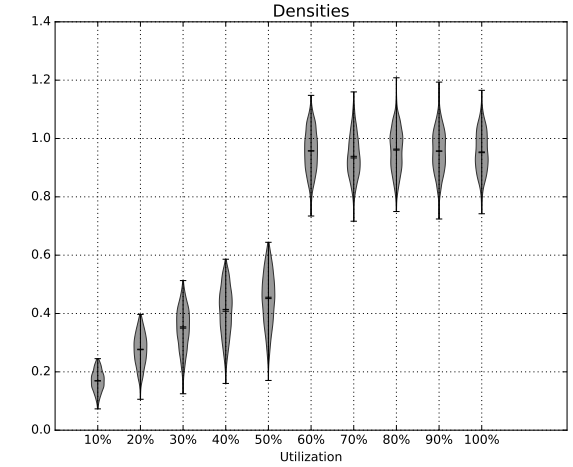
DAG task set statistics (implicit-deadline, 16 processors, P_e = 0.90, P_h = 0.30, P_ℓ = 0.70)

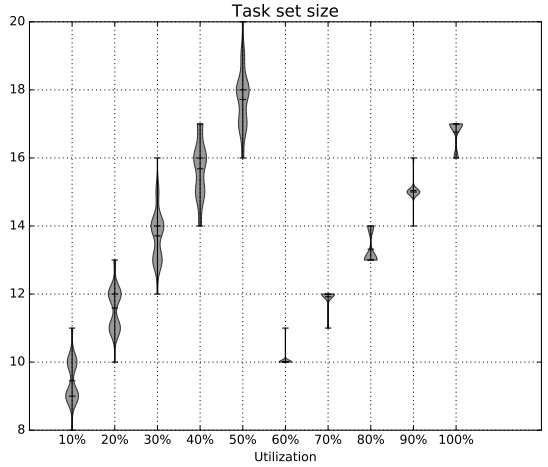




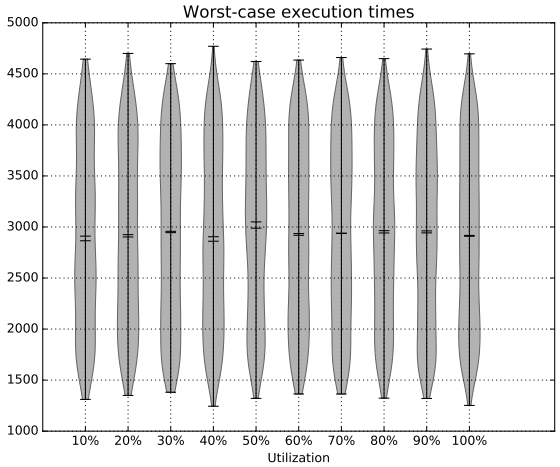


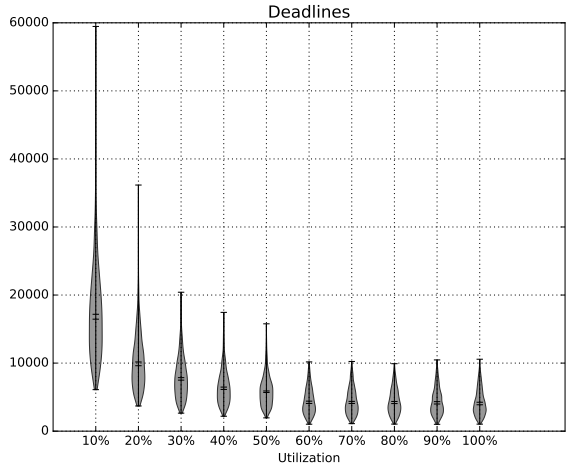


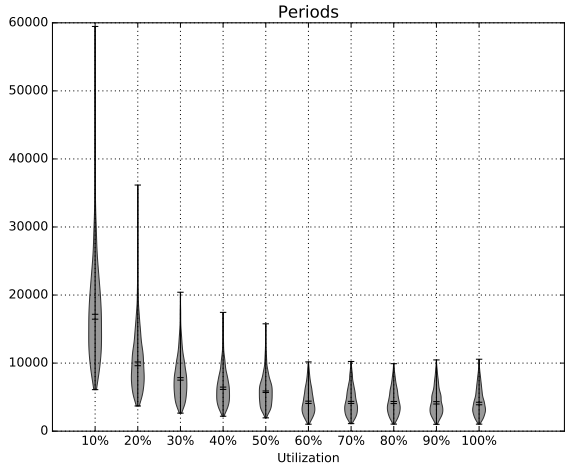


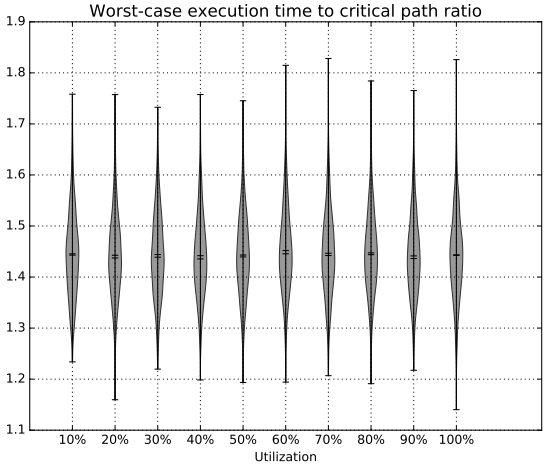


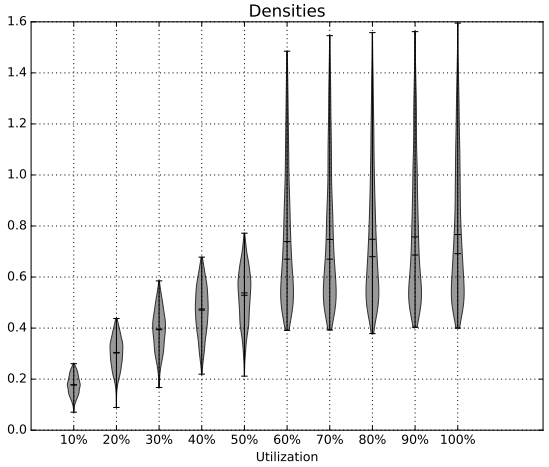
DAG task set statistics (implicit-deadline, 16 processors, P_e = 0.60, P_h = 0.20, P_ℓ = 0.80)

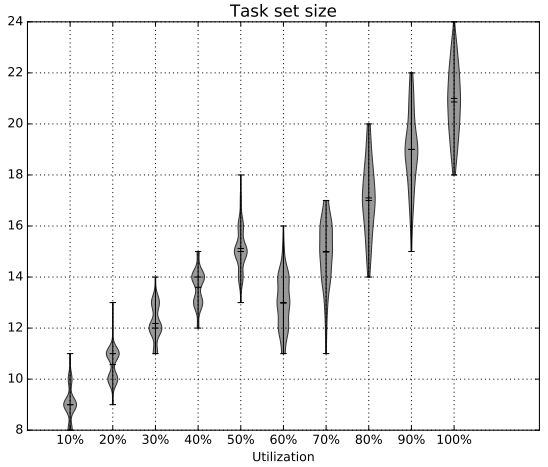




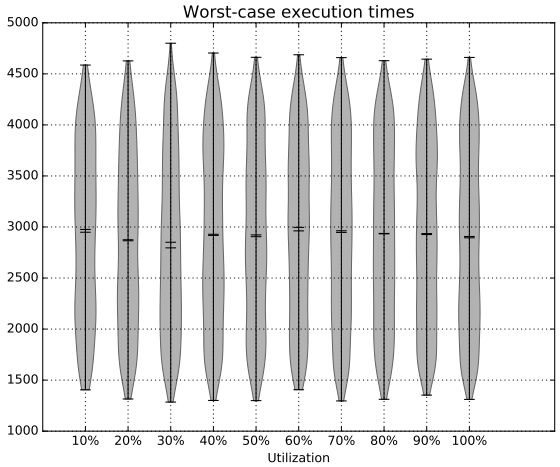


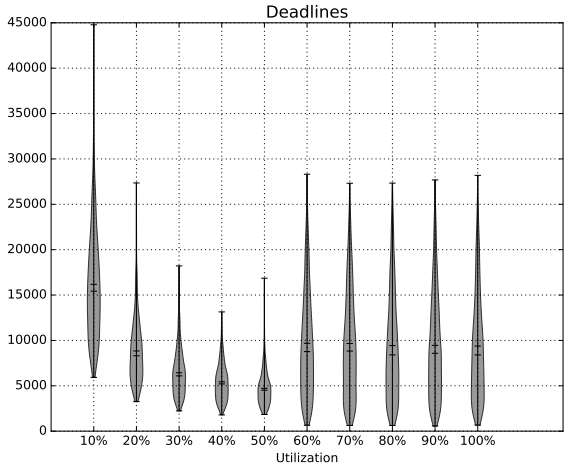


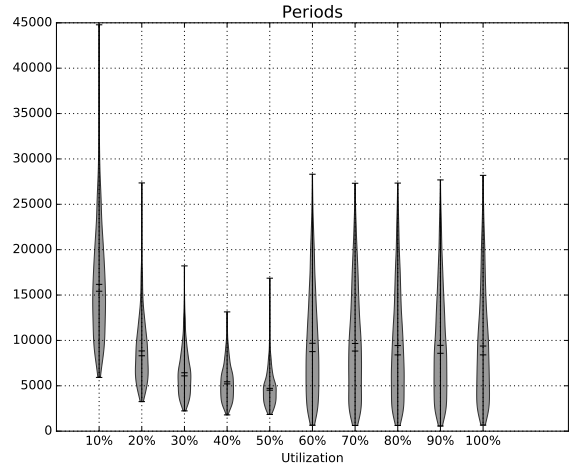


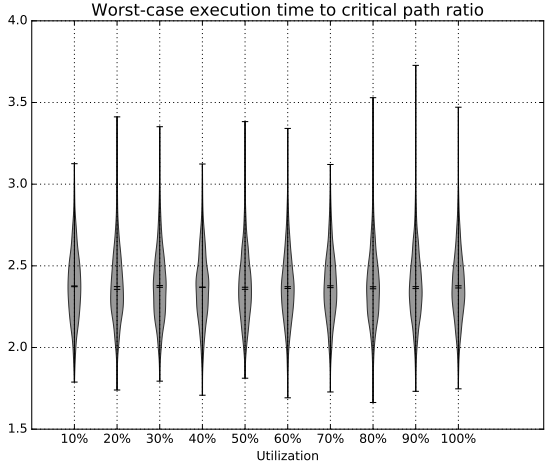


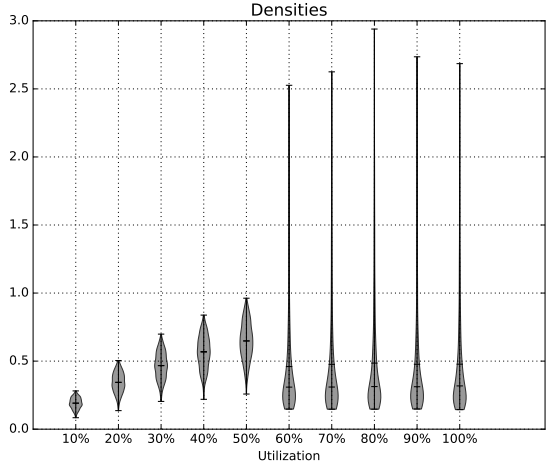
DAG task set statistics (implicit-deadline, 16 processors, P_e = 0.30, P_h = 0.10, P_ℓ = 0.90)

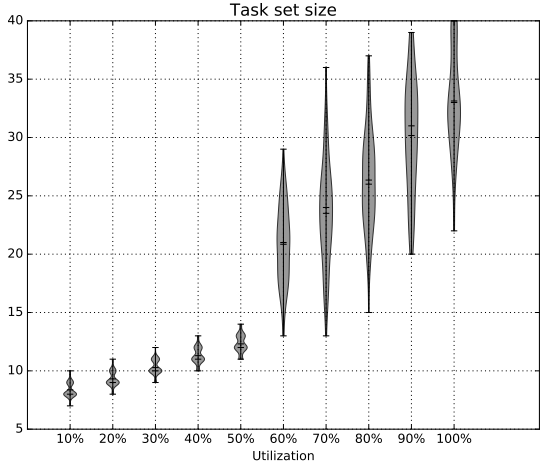




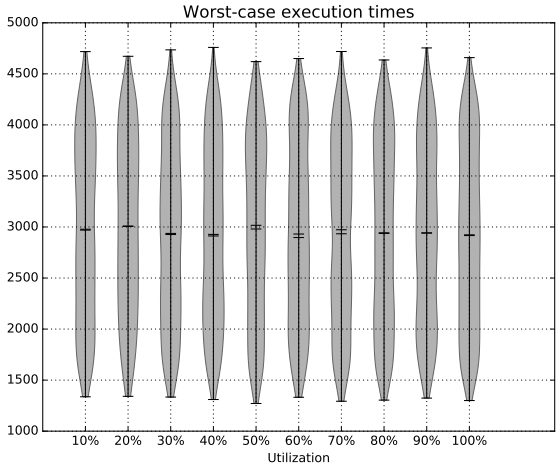


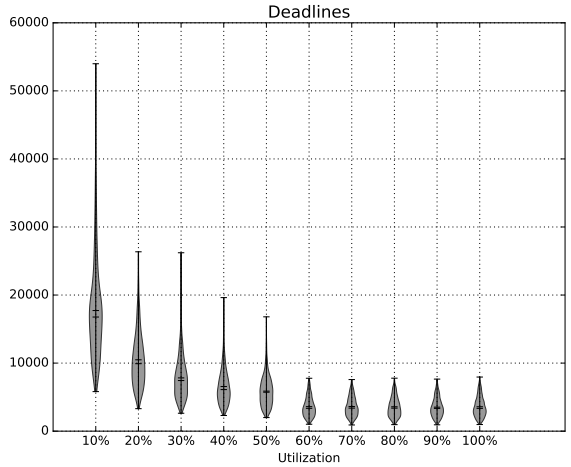


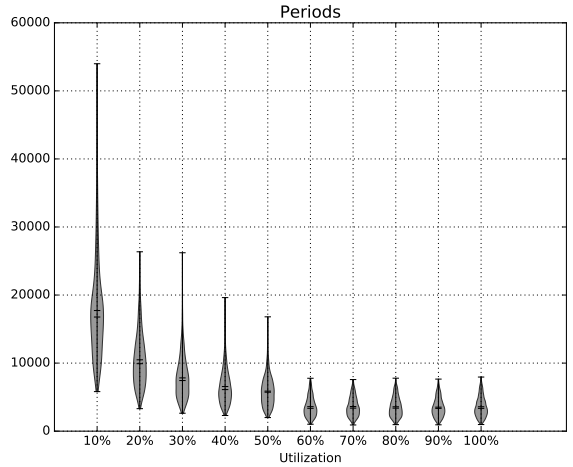


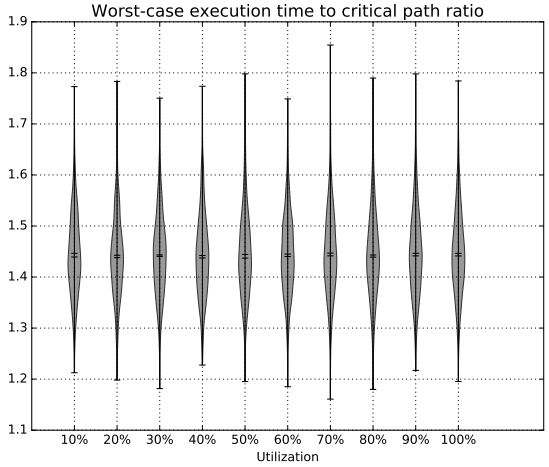


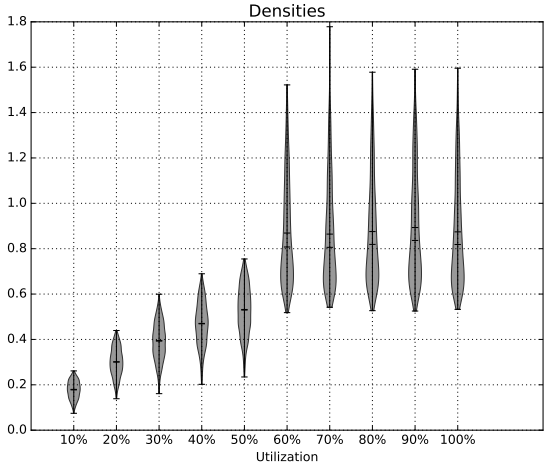
DAG task set statistics (implicit-deadline, 16 processors, P_e = 0.60, P_h = 0.30, P_ℓ = 0.70)

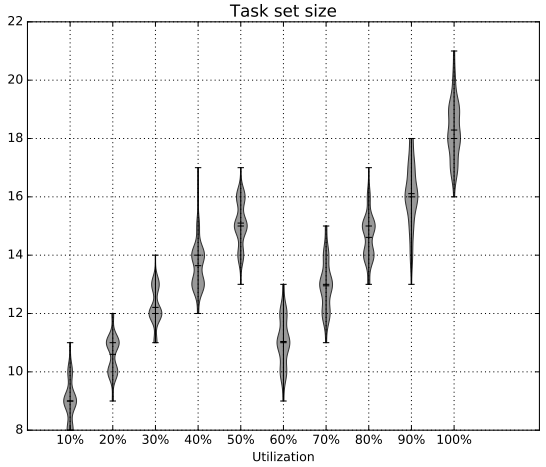




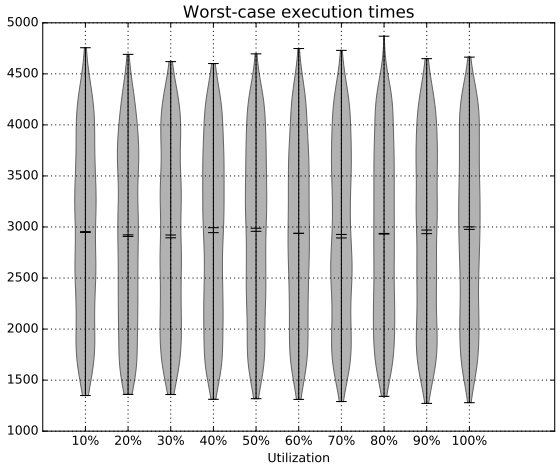


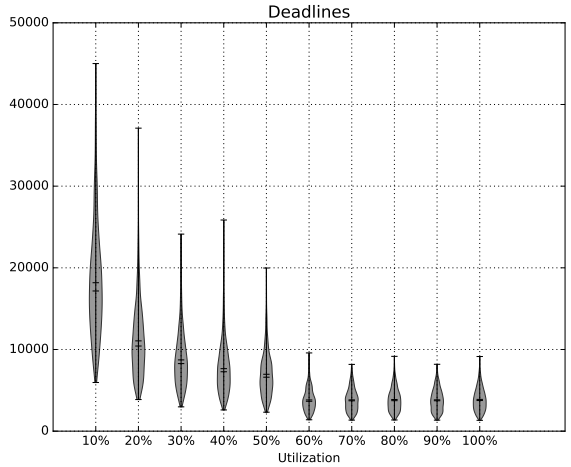


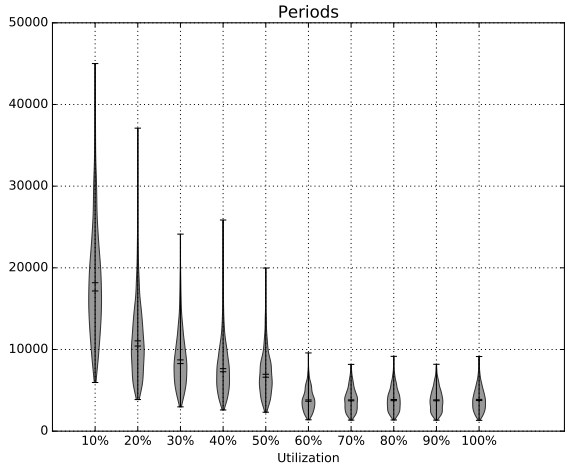


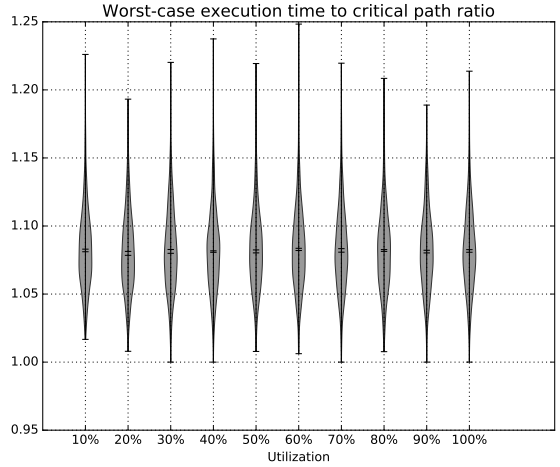


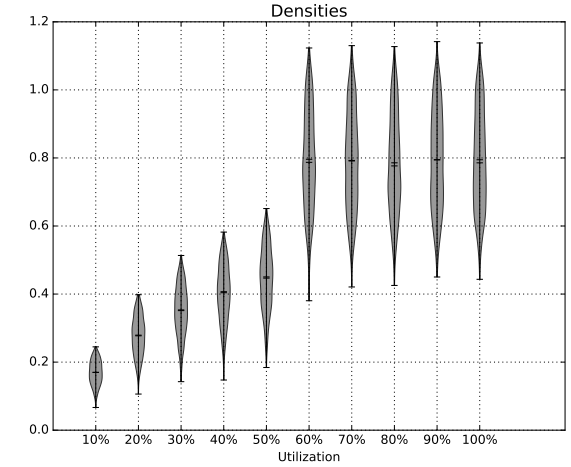
DAG task set statistics (implicit-deadline, 16 processors, P_e = 0.90, P_h = 0.10, P_ℓ = 0.90)

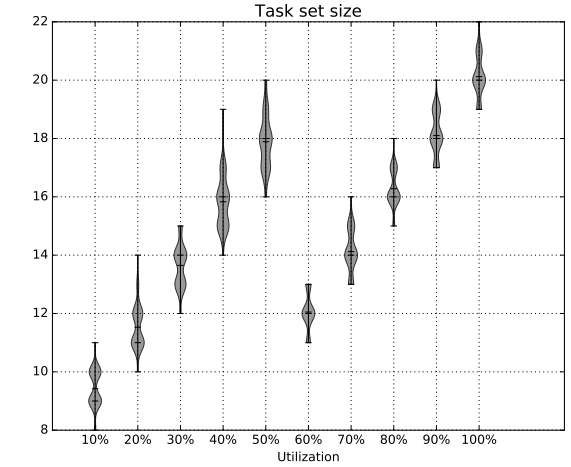












List of Figures

Algorithmenverzeichnis

List of Source Codes

Bibliography

[1] B. Andersson and D. de Niz. Analyzing global-edf for multiprocessor scheduling of parallel tasks. In *Principles of Distributed Systems, 16th International Conference, OPODIS*, pages 16–30, 2012.

[2] S. Baruah. Techniques for multiprocessor global schedulability analysis. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 119–128, 2007.

[3] S. Baruah. The federated scheduling of constrained-deadline sporadic DAG task systems. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition, DATE*, pages 1323–1328, 2015.

[4] S. Baruah. Federated scheduling of sporadic DAG task systems. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS*, pages 179–186, 2015.

[5] S. Baruah. The federated scheduling of systems of conditional sporadic DAG tasks. In *Proceedings of the 15th International Conference on Embedded Software (EMSOFT)*, 2015.

[6] S. Baruah and N. Fisher. Global deadline-monotonic scheduling of arbitrary-deadline sporadic task systems. In *Proceedings of the 11th International Conference on Principles of Distributed Systems*, OPODIS'07, pages 204–216, Berlin, Heidelberg, 2007. Springer-Verlag.

[7] S. K. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, and S. Stiller. Improved multiprocessor global schedulability analysis. *Real-Time Systems*, 46(1):3–24, 2010.

[8] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 345–354, New York, NY, USA, 1993. ACM.

[9] S. K. Baruah and N. Fisher. The partitioned multiprocessor scheduling of sporadic task systems. In *RTSS*, pages 321–329, 2005.

[10] S. K. Baruah and N. Fisher. The partitioned multiprocessor scheduling of deadline-constrained sporadic task systems. *IEEE Trans. Computers*, 55(7):918–923, 2006.

[11] S. K. Baruah and N. Fisher. Global deadline-monotonic scheduling of arbitrary-deadline sporadic task systems. In *Principles of Distributed Systems, 11th International Conference, OPODIS 2007, Guadeloupe, French West Indies, December 17-20, 2007. Proceedings*, pages 204–216, 2007.

[12] S. K. Baruah and N. Fisher. Global fixed-priority scheduling of arbitrary-deadline sporadic task systems. In *Distributed Computing and Networking, 9th International Conference, ICDCN*, pages 215–226, 2008.

[13] S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *proceedings Real-Time Systems Symposium (RTSS)*, pages 182–190, Dec 1990.

[14] M. Bertogna, M. Cirinei, and G. Lipari. New schedulability tests for real-time task sets scheduled by deadline monotonic on multiprocessors. In *Principles of Distributed Systems, 9th International Conference, OPODIS*, pages 306–321, 2005.

[15] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Syst.*, 30(1-2):129–154, May 2005.

[16] E. Bini, T. H. C. Nguyen, P. Richard, and S. K. Baruah. A response-time bound in fixed-priority scheduling with arbitrary deadlines. *IEEE Trans. Computers*, 58(2):279–286, 2009.

[17] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese. Feasibility analysis in the sporadic dag task model. In *ECRTS*, pages 225–233, 2013.

[18] B. B. Brandenburg and M. Gul. Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations. In *2016 IEEE Real-Time Systems Symposium(RTSS)*, volume 00, pages 99–110, Nov. 2016.

[19] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Third Edition*, volume 24 of *Real-Time Systems Series*. Springer, 2011.

[20] J.-J. Chen. Federated scheduling admits no constant speedup factors for constrained-deadline dag task systems. *Real-Time Systems*, 52(6):833–838, November 2016.

[21] J.-J. Chen. Partitioned multiprocessor fixed-priority scheduling of sporadic real-time tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 251–261, 2016.

[22] J.-J. Chen and K. Agrawal. Capacity augmentation bounds for parallel dag tasks under G-EDF and G-RM. Technical Report 845, Faculty for Informatik at TU Dortmund, 2014.

[23] J.-J. Chen and S. Chakraborty. Resource augmentation bounds for approximate demand bound functions. In *IEEE Real-Time Systems Symposium*, pages 272 – 281, 2011.

[24] J.-J. Chen and S. Chakraborty. Resource augmentation for uniprocessor and multiprocessor partitioned scheduling of sporadic real-time tasks. *Real-Time Systems*, 49(4):475–516, 2013.

[25] J.-J. Chen, G. von der Brüggen, W.-H. Huang, and R. I. Davis. On the Pitfalls of Resource Augmentation Factors and Utilization Bounds in Real-Time Scheduling. In M. Bertogna, editor, *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:25, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[26] J.-J. Chen, G. von der Brüggen, and N. Ueter. Push Forward: Global Fixed-Priority Scheduling of Arbitrary-Deadline Sporadic Task Systems. *ArXiv e-prints*, Feb. 2018.

[27] M. Chéramy, P.-E. Hladik, and A.-M. Déplanche. Simso: A simulation tool to evaluate real-time multiprocessor scheduling algorithms. In *5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pages 6–p, 2014.

[28] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner. Random graph generation for scheduling simulations. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, SIMUTools '10, pages 60:1–60:10, ICST, Brussels, Belgium, Belgium, 2010. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[29] M. L. Dertouzos. Control robotics: The procedural control of physical processes. In *IFIP Congress'74*, pages 807–813, 1974.

[30] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[31] P. Ekberg and W. Yi. Fixed-priority schedulability of sporadic tasks on uniprocessors is np-hard. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 139–146, Dec 2017.

[32] P. Emberson, R. Stafford, and R. I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pages 6–11, July 2010.

[33] P. Emberson, R. Stafford, and R. I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, 2010.

[34] N. Fisher, S. K. Baruah, and T. P. Baker. The partitioned scheduling of sporadic tasks according to static-priorities. In *ECRTS*, pages 118–127, 2006.

[35] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, 1969.

[36] N. Guan, M. Stigge, W. Yi, and G. Yu. New response time bounds for fixed priority multiprocessor scheduling. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 387–397, 2009.

[37] W.-H. Huang and J.-J. Chen. Response time bounds for sporadic arbitrary-deadline tasks under global fixed-priority scheduling on multiprocessors. In *RTNS*, 2015.

[38] X. Jiang, N. Guan, X. Long, and W. Yi. Semi-federated scheduling of parallel real-time tasks on multiprocessors. In *Proceedings of the 38nd IEEE Real-Time Systems Symposium, RTSS*, 2017.

[39] X. Jiang, X. Long, N. Guan, and H. Wan. On the decomposition-based global EDF scheduling of parallel real-time tasks. In *Real-Time Systems Symposium (RTSS)*, pages 237–246, 2016.

[40] D. S. Johnson. Fast algorithms for bin packing. *J. Comput. Syst. Sci.*, 8(3):272–314, June 1974.

[41] J. Kim, H. Kim, K. Lakshmanan, and R. Rajkumar. Parallel scheduling for cyber-physical systems: analysis and case study on a self-driving car. In *ACM/IEEE 4th International Conference on Cyber-Physical Systems (with CPS Week 2013), ICCPS*, pages 31–40, 2013.

[42] K. Lakshmanan, S. Kato, and R. R. Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, RTSS '10, pages 259–268, 2010.

[43] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *RTSS*, pages 201–209, 1990.

[44] J. P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium'89*, pages 166–171, 1989.

[45] J. Li, K. Agrawal, C. Lu, and C. D. Gill. Analysis of global EDF for parallel tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3–13, 2013.

[46] J. Li, J.-J. Chen, K. Agrawal, C. Lu, C. D. Gill, and A. Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *26th Euromicro Conference on Real-Time Systems, ECRTS*, pages 85–96, 2014.

[47] J. Li, S. Dinh, K. Kieselbach, K. Agrawal, C. Gill, and C. Lu. Randomized work stealing for large scale soft real-time systems. In *Real-Time Systems Symposium (RTSS), 2016 IEEE*, pages 203–214. IEEE, 2016.

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

[49] S. Liu, J. Tang, Z. Zhang, and J. L. Gaudiot. Computer architectures for autonomous driving. *Computer*, 50(8):18–25, 2017.

[50] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo. Response-time analysis of conditional DAG tasks in multiprocessor systems. In *27th Euromicro Conference on Real-Time Systems, ECRTS*, pages 211–221, 2015.

[51] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. Techniques optimizing the number of processors to schedule multi-threaded tasks. In *24th Euromicro Conference on Real-Time Systems, ECRTS*, pages 321–330, 2012.

[52] A. Olteanu and A. Marin. Generation and evaluation of scheduling dags: How to provide similar evaluation conditions. *Computer Science Master Research*, 1(1):57–66, 2011.

[53] A. Saifullah, K. Agrawal, C. Lu, and C. D. Gill. Multi-core real-time scheduling for generalized parallel task models. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium, RTSS*, pages 217–226, 2011.

[54] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. D. Gill. Parallel real-time scheduling of dags. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3242–3252, 2014.

[55] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quiones. Timing characterization of openmp4 tasking model. In *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 157–166, Oct 2015.

[56] Y. Sun, G. Lipari, N. AGuan, W. Yi, et al. Improving the response time analysis of global fixed-priority multiprocessor scheduling. In *IEEE International Conference*

on Embedded and Real-Time Computing Systems and Applications (RTCSA), pages 1–9, 2014.

[57] G. von der Brüggen, N. Ueter, J.-J. Chen, and M. Freier. Parametric utilization bounds for implicit-deadline periodic tasks in automotive systems. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, RTNS '17, pages 108–117, New York, NY, USA, 2017. ACM.

Eidesstattliche Versicherung

_____ _____

Name, Vorname Matr.-Nr.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem Titel

_____

_____

_____

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

_____ _____

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

_____ _____

Ort, Datum Unterschrift