TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Realzeit-Computersysteme

Verifying and Allocating Real-Time Tasks on Distributed Processing Units

Alejandro Masrur

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr.-Ing. R. Kennel
Prüfer der Dissertation:	1. Univ.-Prof. Dr.-Ing. G. Färber (em.)
	2. Univ.-Prof. Dr. sc. techn. A. Herkersdorf

Die Dissertation wurde am 16.06.2009 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 10.03.2010 angenommen.

Acknowledgements

First of all, I sincerely thank Prof. Färber for his guidance, his unfailing patience and for being always available to me. None of this, neither this Ph.D. thesis nor my stay in Munich, would have ever been possible without his support. I profoundly thank Prof. Chakraborty for his valuable contributions to this thesis and for helping me find motivation towards the end of my Ph.D. For the interesting work and discussions shared, I thank Prof. Herkersdorf to whom I am also particularly grateful for accepting to be my second examiner. I further thank Prof. Kennel for undertaking the chair of the examination board.

To all my colleagues and the staff at RCS, I am deeply grateful for their unconditional help whenever I needed it and, of course, for the pleasant and friendly working atmosphere. Because of them, I will always look back on the time at RCS with great affection.

I would also like to thank my parents and brothers, who ever trusted and supported me. I thank them for respecting my decisions, even when they sometimes did not agree with me. I thank very much my beloved wife Ana not only for her tolerance and sympathy, but also for her contributions to this thesis. I am not sure, if I completely understand her work, but I am certain that she understands mine.

Finally, I gratefully acknowledge the support by the DAAD at the beginning and during the first three and a half years of my Ph.D.

Munich, June 2009

To my wife Ana.

Contents

Abstract

In general, two major issues arise when designing real-time embedded systems upon multiple processors: task allocation and feasibility/schedulability analysis. The task allocation problem is concerned with the assignment of tasks to processors, whereas the feasibility/schedulability analysis deals with testing whether a given set of real-time tasks is schedulable or not. A set of real-time tasks is said to be schedulable or feasible on one or more processors, when all its timing constraints (deadlines) can be guaranteed. Clearly, we cannot allocate real-time tasks to processors that are unable to guarantee their deadlines and, hence, these two problems are interdependent and cannot be handled separately.

In this thesis, we provide an integrated framework for task allocation and feasibility analysis. In particular, new better linear-time feasibility tests are used in conjunction with allocation heuristics. This way, we first analyze the case of allocating independent real-time tasks and then extend algorithms to consider task dependencies. The contributions of this thesis are as follows:

- A novel technique is proposed to perform the feasibility analysis of both fixed and dynamic-priority scheduling policies. This new technique consists in calculating the *maximum loading factor* generated by real-time tasks on a single processor. The concept of loading factor is defined as the total execution demand within a specified time interval divided by the length of this interval. Hence, the maximum loading factor is the upper bound on the loading factor which results from considering every possible time interval. As a consequence, if the maximum loading factor of a given task set is less than or equal to unity on a single processor, the processor will be able to comply with the execution demand of all tasks. Thus, the task set is said to be feasible on that processor.

- Applying the concept of maximum loading factor, linear-time sufficient feasibility tests are presented for the most general case of task having arbitrary deadlines. We analyze both fixed-priority and dynamic-priority scheduling algorithms. Further, the proposed feasibility tests are shown to be more accurate (i.e., less pessimistic) than the known algorithms with same complexity that can be found in the literature.

- The proposed feasibility tests are also combined with well-known bin packing heuristics (e.g., First Fit and First Fit Decreasing) to derive fast polynomial-time allocation algorithms for independent real-time tasks with arbitrary deadlines. By means of a detailed comparison, we further show that these algorithms achieve better task allocations on the average than the ones based on feasibility analysis methods from the literature. This means that the proposed heuristics lead to a bigger reduction of the number of processors, which are necessary to guarantee feasibility for the whole task set.

- The problem of allocating dependent real-time tasks to multiple distributed processors is analyzed. Particularly, we focus on task communication and system constraints. For the case of communicating tasks, different heuristics are presented to reduce the amount of communication between processors during the allocation procedure. Finally, some additional allocation heuristics are proposed to minimize both the number of processors and the amount of communication between them. In contrast to most communication-aware allocation methods from the literature, the proposed algorithms have polynomial complexity and can possibly be adapted to perform an on-line allocation for communicating tasks.

Zusammenfassung

In Bezug auf die Entwicklung eingebetteter Realzeit-Systeme basierend auf mehreren Prozessoren entstehen im Allgemeinen zwei große Herausforderungen: die Taskallokation und der Echtzeitnachweis. Während sich die Taskallokation mit der Zuordnung von Tasks auf Prozessoren beschäftigt, ist das Echtzeitnachweis-Verfahren dafür verantwortlich, die Machbarkeit des Realzeit-Tasksystems zu prüfen. Ein Realzeit-Tasksystem ist nur dann machbar oder realisierbar, wenn garantiert werden kann, dass alle Zeitschranken (Deadlines) eingehalten werden. Dabei soll eine Task keineswegs einem Prozessor zugeordnet werden, der nicht in der Lage ist, sie rechtzeitig auszuführen. Daher können Echtzeitnachweis und Taskallokation nicht getrennt und müssen als ein Ganzes betrachtet werden.

Diese Dissertation befasst sich mit einer ganzheitlichen Betrachtung von Taskallokation und Echtzeitnachweis. Insbesondere werden neue und bessere Echtzeitnachweis-Verfahren linearer Zeit in Verbindung mit Allokationsheuristiken verwendet, wobei die Allokation unabhängiger Realzeit-Tasks zunächst analysiert wird. Darüber hinaus werden die Algorithmen zur Berücksichtigung von Taskabhängigkeiten erweitert. Der wissenschaftliche Beitrag dieser Dissertation kann folgendermaßen zusammengefasst werden:

- Eine neuartige Technik zum Echtzeitnachweis wird eingeführt, die bei Scheduling-Verfahren sowohl fester als auch dynamischer Priorität angewandt werden kann. Die vorgestellte Echtzeitnachweis-Technik basiert auf der Berechnung des *maximalen Belastungsmaßes*, welches das Tasksystem auf einem Einzelprozessor bewirkt. Der Begriff Belastungsmaß wird als das Verhältnis zwischen der gesamten Rechenanforderung innerhalb eines bestimmten Zeitintervalls und der Länge des Zeitintervalls definiert. Das maximale Belastungsmaß ist daher die obere Schranke des Belastungsmaßes, die aus der Betrachtung jedes möglichen Zeitintervalls resultiert. Wenn das maximale Belastungsmaß eines Tasksystems auf einem Einzelprozessor nicht die Einheit übersteigt, dann ist der Prozessor imstande die Rechenanforderung aller Tasks zu entsprechen. Das Tasksystem ist infolgedessen realisierbar auf dem Prozessor.

- Für den allgemeinen Fall beliebiger Deadlines werden hinreichende Echtzeitnachweis-Verfahren linearer Zeit, basierend auf dem Begriff des maximalen Belastungsmaßes, präsentiert. Scheduling-Algorithmen mit festen und mit dynamischen Prioritäten werden analysiert. Des weiteren wird gezeigt, dass die vorgestellten Verfahren genauer (d.h. weniger pessimistisch) sind, als bekannte Algorithmen gleicher Komplexität aus der Literatur.

- Die vorgeschlagenen Echtzeitnachweis-Verfahren werden mit allgemein bekannten Heuristiken für Bin Packing (z.B. First Fit und First Fit Decreasing) kombiniert, um schnelle

Allokationsalgorithmen polynomischer Zeit für Realzeit-Tasks mit beliebigen Deadlines abzuleiten. Im Durchschnitt erzielen diese Algorithmen eine bessere Taskzuteilung als dazu konkurrierende Methoden basierend auf Echtzeitnachweis-Methoden aus der Literatur. Das heißt, die vorgestellten Heuristiken führen zu einer kleineren Anzahl von Prozessoren, die zur Realisierbarkeit des gesammten Tasksystems notwendig sind. Letzteres wird anhand eines ausführlichen Vergleichs veranschaulicht.

- Die Zuteilung von abhängigen Realzeit-Tasks auf mehrere verteilte Prozessoren wird weiterhin analysiert. Insbesondere werden kommunizierende Tasks und systembedingte Randbedingungen in Erwägung gezogen. Für den Fall kommunizierender Tasks werden verschiedene Allokationsheuristiken zur Reduktion des Kommunikationsumfangs zwischen Prozessoren vorgeschlagen. Zum Schluss werden ergänzende Heuristiken dargestellt, die zur Minimierung der beiden Größen Prozessoranzahl und Kommunikationsumfang unter Prozessoren dienen. In Gegensatz zu den meisten in der Literatur vorgeschlagen kommunikationsbewussten Allokationsmethoden zeichnen sich die in dieser Dissertation dargelegten Algorithmen durch ihre polynomische Komplexität aus. Daher eignen sie sich besonders dafür, auf ihrer Basis eine online Taskallokation unter Berücksichtigung von Kommunikation zu realisieren.

1 Introduction

Although the speed of processors has been rapidly increasing in recent decades, there is an even faster growing demand for computation capacity in embedded systems. The use of multiple processors is often the only way to fulfill the computation requirements of many current and upcoming applications. Additionally, most powerful processors are nowadays based on multicore architectures, so that engineers are frequently confronted with designing embedded systems upon multiple processing units. As a consequence, there is a strong interest in developing design methods and techniques for multiprocessor environments.

The problem of executing time-constrained tasks upon multiple processors has been intensively studied in the literature. However, this still remains a very active area of research. Scientific work so far has focused on the analysis of either global or partitioned multiprocessor scheduling for real-time tasks. As discussed later, tasks are globally scheduled when they can be assigned on-line to any available processor. On the other hand, partitioned scheduling is such a one in which tasks are assigned off-line and exclusively to one processor. Clearly, the number of processors is normally known in the case of global scheduling. However, for a partitioned scheduling, the number of processors may be as well an unknown variable that needs to be determined at design time.

The problem of finding an optimal task assignment to processors is known to be intractable [GJ79] (i.e., the running time of an optimal allocation algorithm grows exponentially with the number of tasks to be allocated). For this reason and because of its on-line nature, heuristic methods have been proposed to perform a global scheduling. In general, these global scheduling heuristics are based on simple on-line bin packing heuristic algorithms (like, for example, the well-known First Fit). Additionally, researchers have been able to come up with total utilization bounds for which the feasibility of all real-time tasks can be guaranteed on a given number of processors.

Algorithms to achieve an optimal allocation seem to be preferred with respect to partitioned schedulings of real-time tasks. As mentioned above, the running time of such an optimal algorithm increases exponentially with the number of tasks. However, there are some considerations that argue for optimal algorithms. For example, there are normally few tasks to allocate in most practical situations, so the running time of an optimal allocation algorithm is tolerable in this case. Further, the task assignment is carried out off-line when considering partitioned multiprocessor schedulings, so we normally have sufficient time to look for the optimal assignment. Additionally, an optimal task allocation is more desirable because it reduces costs by resulting in the least possible number of processors. There are some approaches that can be used to solve this problem for an optimal allocation, however, integer programming seems to be the most accepted one in the literature. Here, the allocation problem must be expressed as an integer

program and, then, it can be solved with known integer programming techniques. Further, there exist already a number of efficient solvers which can be used for this purpose.

Several authors have also proposed heuristic methods for allocating tasks to processors under a partitioned scheduling scheme. The main advantage of heuristic approaches is that they are much more faster than optimal algorithms while they deliver fairly good allocations. For instance, if all tasks are independent of each other, some of the well-known bin packing heuristics can also be applied to perform an off-line task allocation. This time, it is not necessary to restrict oneself to on-line algorithms, but off-line bin packing heuristics (e.g., First Fit Decreasing) are certainly more advantageous because of their better performance ratios. Nevertheless, more complex heuristics, for example, based on *genetic algorithms*, have been proposed to allocate dependent tasks. Dependent tasks are considered those that influence the behavior of one another or that interact in some way like, for example, by means of exchanging information between them, by presenting precedence constraints, etc.

In this thesis, heuristic algorithms are presented to perform a task allocation under a partitioned scheduling on multiple distributed processors. For this purpose, we also make use of simple bin packing heuristics, but concentrate on two issues to which it has been paid little attention in the literature:

- Allocating real-time tasks with arbitrary deadlines under fixed and dynamic-priority scheduling policies;
- The allocation of real-time tasks considering communication and system constraints.

The choice of bin packing heuristics is due to the fact that they deliver reasonably good task allocations and that they present polynomial complexity, which is alway desirable in particular for performing on-line allocations.

Although the allocation problem has already been analyzed before on the basis of bin packing heuristics, e.g., in [DL78, LDG04], almost all approaches assume that deadlines are equal to periods. This assumption simplifies substantially the feasibility analysis under both fixed and dynamic priorities, which can thus be carried out in polynomial time. This way, the allocation heuristic, which must compulsorily rely on a feasibility test to guarantee that no deadlines are missed, becomes much more simple.

On the other hand, if deadlines are not restricted to be equal to periods, the feasibility analysis gets more complex; the methods used for deadlines equal to periods are not valid anymore, and a feasibility test can at best be performed with pseudo-polynomial complexity. The combination of these pseudo-polynomial time feasibility tests with bin packing heuristics was studied by Sáez et al. in [SVC98]. However, in this case, we must resign the desired polynomial complexity for the allocation heuristics because of using pseudo-polynomial feasibility tests.

For this reason, we propose sufficient feasibility tests which present polynomial complexity and consequently do not degrade the complexity of the heuristics. The proposed feasibility tests are shown to be more accurate than all known tests of similar complexity. Furthermore, it is shown that allocation heuristics based on these new feasibility tests yield less processors than the respective heuristics from the literature.

Additionally, we extend known bin packing heuristics to consider task communication and system constraints. Even in this case, the resulting heuristics are shown to have polynomial complexity. Finally, novel heuristics are presented and evaluated for reducing communication between tasks. In addition, we consider the problem of optimizing the number of processors and the amount of communication among them simultaneously. For this case, we propose some more heuristics and compare them with the known bin packing heuristics to minimize the number of processors and with the proposed algorithms for reducing communication.

1.1 Related Work

Related work is referenced and discussed all throughout this thesis as it gets necessary. This section gives, however, a general overview of the literature with respect to feasibility analysis on both uniprocessors and multiprocessors. This topic has been intensively studied in recent years and is still a very active research area. A brief overview of methods and techniques for task allocation is further given.

1.1.1 Uniprocessor Scheduling

A feasibility test depends on the scheduling algorithm used to schedule tasks. In general, two different paradigms has been proposed for real-time tasks: fixed priorities and dynamic priorities. An overview of scheduling theory concerning both of these scheduling paradigms is given in [SAr+04].

Feasibility Analysis for EDF

The Earliest Deadline First (EDF) algorithm is a dynamic priority algorithm. EDF is optimal on uniprocessors [Der74], but it is, on the other hand, more complex to implement. An interesting comparison of EDF against the fixed-priority Rate Monotonic (RM) can be found in [But05].

Liu and Layland proved that a set of real-time tasks is feasible on a uniprocessor under EDF if the total processor utilization is less than or equal to 1 (i.e., 100%). To obtain this feasibility condition, Liu and Layland assumed that tasks are fully preemptive, periodic and synchronous and that deadlines (d_i) are all equal to the respective periods (p_i) [LL73]. Tasks are called synchronous when they all are released simultaneously at the beginning of the schedule. Afterwards in [BMR90], Baruah et al. proved that applying the utilization test of Liu and Layland is also valid when $d_i \geq p_i$ holds for all tasks.

When deadlines are allowed to be less than periods, the complexity grows considerably. However, Liu and Layland also showed that if a synchronous task set is not feasible, then in its schedule a deadline is missed without idle time prior to it. Additionally, assuming the processor utilization to be less than 100% also for a synchronous scheduling, Baruah et al. proved in [BRH90, BMR90, BHR93] that if a deadline is missed, this happens before a maximum time

upper limit known as *feasibility bound*. This result allowed Baruah et al. to design a pseudo-polynomial time algorithm for the case where deadlines are not forced to be equal to periods.

Another pseudo-polynomial time algorithm for $d_i \leq p_i$ was presented by Ripoll et al. in [RM96]. Ripoll et al. introduced two better feasibility bounds, which they combined in the same algorithm. On the other hand, George et al. considered in [GRS96] also the case $d_i > p_i$ and got a more general expression of the feasibility bound for arbitrary deadlines.

George's bound reduces to Ripoll's bound if $d_i \leq p_i$ holds for all possible i. A similar feasibility bound was also obtained by Zheng and Shin in [ZS94]. In [MDF08], it was proven that a slight improvement of George's feasibility bound is also possible. This improvement requires at least an additional calculation for any of the tasks and gets maximal when an additional calculation for all tasks is performed; however, the computation complexity of this bound remains $\mathcal{O}(n)$.

The other feasibility bound presented by Ripoll et al. is based on the busy period analysis, whose calculation itself presents pseudo-polynomial complexity. This latter pseudo-polynomial feasibility bound was also independently obtained by Spuri [Spu95, Spu96].

All exact algorithms, including the one of Albers and Slomka [AS05] for $d_i \leq p_i$, present pseudo-polynomial complexity. In order to reach polynomial complexity in feasibility testing for EDF, when d_i can be less than p_i, exactness must be sacrificed. Based on this idea, Liu in [Liu00] and Stankovic et al. in [SSRB98] propose independently the *density test* which has complexity $\mathcal{O}(n)$. The density of a task is defined as the ratio of its execution time over the minimum between p_i and d_i.

A first Fully Polynomial Time Approximation Scheme (FPTAS) for EDF was presented by Chakraborty et al. in [CKT02], where also the concept of optimistic feasibility test was introduced. An optimistic feasibility test does not guarantee that no deadline is missed, but if this happens, the optimistic test makes it sure that the time overflow remains below a given configurable error. These new concepts were applied in the context of packet processing embedded systems in [Cha03].

Assuming that tasks are sorted by non-decreasing deadlines d_i, Devi presented in [Dev03] an $\mathcal{O}(n \log n)$ approach that is better in terms of accepted task sets than the density condition. Albers and Slomka presented in [AS04] a second FPTAS to perform a feasibility test under EDF. Further, they proved in [AS05] that this FPTAS is a more general expression of Devi's test when $d_i \leq p_i$ for all i. Based on the improved feasibility bound from [MDF08], another sufficient polynomial-time test was presented, which is more accurate than Devi's test but has complexity $\mathcal{O}(n^2)$.

All mentioned approaches assume that tasks are synchronous, i.e., that they are released simultaneously at the beginning of the schedule. When tasks may have offsets, i.e., the initial release times (also called phases) for some tasks are not zero, the schedulability conditions may relax with respect to the synchronous case. However, a schedulability relaxation may not happen in some cases, because tasks can still be released together at some later time instant. A task set is called asynchronous when some tasks in it have initial release times different than zero. In general, an asynchronous task set is feasible (no deadlines are missed), if the corresponding synchronous task set (obtained ignoring offsets in the asynchronous one) is feasible. On the

other hand, an asynchronous task set might be feasible, even if its corresponding synchronous task set is infeasible. Goossens considered in [Goo03] the scheduling of task with configurable offsets, i.e., tasks for which offset can be freely chosen. Additionally, Pellizzoni and Lipari presented an approximation algorithm to perform a feasibility test of asynchronous task sets [PL04, PL05, PL07].

Feasibility Analysis for Fixed Priorities

Unlike dynamic-priority scheduling algorithms, the fixed-priority ones are generally not optimal on uniprocessors. However, a fixed-priority scheduling algorithm is easier to implement and integrate in an operating system. On the other hand, if all periods are harmonic, it is possible to reach 100% processor utilization with RM [Liu00], i.e., RM is optimal like EDF in this case. Further, if $d_i \leq p_i$ holds for all tasks, Deadline Monotonic (DM) is the optimal priority assignment [LW82]. In other words, a set of tasks can be feasibly scheduled on one processor according to DM whenever it can be feasibly scheduled by any fixed-priority algorithm, but the inverse does not hold [LW82].

For a set of preemptive, independent, periodic, real-time tasks with deadlines equal to periods, Liu and Layland observed in [LL73] that the worst-case scheduling situation on a uniprocessor happens when tasks are released simultaneously. Further, they presented in [LL73], among others, a utilization bound for the case where tasks are scheduled under RM and $d_i = p_i$ holds for all tasks.

A better utilization upper bound for this case was proposed independently by Liu in [Liu00] and by Bini et al. in [BBB01, BB03]. This latter utilization bound does not depend on the number of tasks as the Liu and Layland bound does. Bini et al. called it hyperbolic bound and showed that it improves the acceptance ratio over Liu and Layland's utilization bound by a factor of $\sqrt{2}$ for a large number of tasks. A similar utilization bound was proposed by Oh et al. in [OS95] to be used in the task allocation problem.

There are some other utilization bounds for the case that $d_i = p_i$ holds for all tasks under RM. Kuo et al. presented in [KM91] a utilization bound that exploits the fact that 100% utilization is possible under RM when tasks have harmonic periods. Further, Burchard et al. presented in [BLOS95] another utilization bound that varies not only with the number of tasks but also with a factor quantifying how close tasks are to having harmonic periods. For arbitrary deadlines, Lehoczky proposed in [Leh90] a utilization bound for RM that does not only depend on the number of tasks but also on the ratio $\frac{d_i}{p_i}$. Lehoczky assumed that this factor is further the same for all tasks.

An exact feasibility test with pseudo-polynomial complexity was presented by Lehoczky et al. in [LSD89] for the case of RM and deadlines equal to periods. Afterwards, in [ABRW91], Audsley et al. considered a DM priority assignment and improved in [ABR+93] Lehoczky's exact feasibility test by observing that tasks' worst-case response times can be found in an iterative manner. Further, Audsley et al. considered deadlines less than or equal to periods and other priority assignments. More recently, Bini and Buttazzo presented in [BB04b] a tunable

feasibility test for the case that $d_i \leq p_i$ holds for all tasks under fixed priorities. Bini and Buttazzo's test allows configuring complexity versus acceptance ratio for the testing.

For arbitrary deadlines and fixed priorities, Lehoczky showed in [Leh90] that the response time of the first job of a task T_i, scheduled synchronously with all higher-priority tasks, is not necessarily the maximum response time for T_i. In order to test T_i's feasibility in this case, the response times of all T_i's jobs within its first busy period must be calculated. Based on this observations, Lehoczky provided in [Leh90] a more general feasibility test for fixed priorities and arbitrary deadlines. T_i's first busy period is the time interval from the synchronous release together with all higher-priority tasks to the time instant at which all previously released jobs finish executing.

In [FB05b, FB05a, FB06], Fisher and Baruah proposed an approximation scheme for the known pseudo-polynomial-time exact feasibility test [LSD89, ABR+93, Leh90]. This approximation has polynomial complexity and is based on the approximation techniques used by Albers and Slomka in [AS04] for the case of an EDF scheduling.

Finally, all approaches discussed for fixed priorities assume that tasks are synchronous, i.e., tasks that are released simultaneously at the beginning of the schedule. The problem of feasibility testing for fixed-priority tasks with offsets has also been studied in the literature [Aud91, Tin94, PGH98, Goo03].

1.1.2 Multiprocessor Scheduling

Considering a real-time task to be a succession of real-time jobs, there are two approaches to schedule these jobs on multiple processors: the partitioned (or off-line) and the global (or on-line) scheduling. Under partitioned scheduling, all jobs of a task are assigned to a particular processor, i.e., they always run on the same processor. On the other hand, under global scheduling, jobs are not restricted to one processor, but they may run on any available processor at a given time instant. In this latter case, on which processor the real-time jobs are going to run is decided on-line by the scheduler.

Global Multiprocessor Scheduling

We know from the previous discussion that EDF is an optimal (on-line) scheduling algorithm on uniprocessors [Der74], however, Hong and Leung showed that no global (on-line) scheduling algorithm can be optimal on multiprocessors [HL88, HL92].

For global multiprocessor scheduling, researchers have been proposing utilization bounds in the same way as Liu and Layland did for single processors [LL73]. The key idea is that a given set of real-time tasks is feasible on a multiprocessor system, if the total utilization on the multiprocessor does not exceed a given bound. Along the years, quite a few utilization bounds for multiprocessors were proposed for different scheduling algorithms; some of these results are also summarized in [SAr+04].

Multiprocessor Utilization Bounds for Fixed Priorities: The algorithm Rate Monotonic First Fit (RMFF) [DL78] was the first proposed approach for performing a multiprocessor scheduling and is based on the combination of RM and the well-known First Fit (FF) heuristic. RMFF can be applied to partition the task set on a multiprocessor and also to perform a global scheduling of tasks. For the use of RMFF as a global scheduling algorithm, different utilization bounds are already known [OB98, LDG01]. Other utilization bounds for fixed priorities appeared in the literature over the years [ABJ01, LDG04]. The effect of the utilization of individual tasks on the multiprocessor utilization bound has also been extensively investigated [PSTW97, ABJ01, FGB01, Lun02, GFB03, Bak03, Bar04, Bak06, BF06a].

Multiprocessor Utilization Bounds for EDF: In what respects to EDF, López et al. introduced in [LGDG00] the analogous to RMFF called Earliest Deadline First-First Fit (EDF-FF), for which they proposed a utilization upper bound on multiprocessors. There exists also a large body of work concerning multiprocessor utilization bounds for EDF [Bak03, Bak05, FB05c, BCL05, BB08].

Partitioned Multiprocessor Scheduling and Task Allocation

Clearly, for the partitioned scheduling, it is first necessary to allocate tasks onto processors. Tasks then remain on the processors they were assigned to and can be scheduled under the known uniprocessor techniques. However, the task allocation problem is not easy to solve. Let us assume, for example, that we have n tasks and that the number of processors in fixed to q. In this case, there are q^n different possible ways of assigning tasks to processors. Further, finding an optimal feasible allocation (i.e., one for which no deadlines are missed) is known to be NP-hard in the strong sense [GJ79], i.e., it has exponential complexity.

If we consider tasks to be independent of one another, there exist already good bin packing heuristics, like the previously mentioned First Fit (FF), that can be successfully applied to solve the allocation problem. An overview of the possible bin packing heuristics was presented by Coffman et al. in [CGJ97], where the authors discuss among others the complexity of the different algorithms and their worst-case performance ratios.

The allocation problem gets harder if there are dependencies among tasks. For this reason, researchers have been trying to come up with good approximation algorithms. For instance, genetic algorithms has been applied in [AD96, Thi00, WYJ+04], but also a technique known as *simulated annealing* has shown to deliver reasonable results [Tin90, TBW92, BNTZ94]. In [AKS+02], greedy heuristics were used to obtain allocations that guarantee a given quality of service.

On the other hand, many researchers opt for optimal solution approaches. Of course, the computational complexity of these approaches is exponential. However, this can be put into perspective by taking into account that we normally have few tasks and that the allocation procedure is performed off-line. There are basically two common approaches to obtain an optimal allocation: integer programming and graph theoretic approach. The graph theoretic approach makes use of graph techniques to obtain the best allocation [ST85, Lat94]. However, the integer

programming method has gained more attention lately and consists in formulating the allocation problem as an integer program [MFHS05, MFHS06, MH06, RGB+08], which can then be solved with well-known techniques for integer programming.

1.2 Underlying Models and Assumptions

In this section, we briefly discuss the models on which all presented methods are based. Additionally, the nomenclature used all along this thesis is introduced. There are, however, some minor details and definitions that will be presented correspondingly as they become necessary so as to ease the exposition.

1.2.1 Task Model

First, we specify the task model and some related notation used in the following chapters. Throughout this thesis, it is assumed that a set of real-time tasks is given. We denote this task set by $\mathbf{T}_n$, where n represents the number of tasks in it. Further, all real-time tasks are assumed to be periodic, independent and fully preemptive. In what follows, we will also use $\mathbf{T}_l$ to denote a subset of the first l tasks in $\mathbf{T}_n$.

We consider that each task T_i in $\mathbf{T}_n$ is characterized by its period of repetition p_i, its relative deadline d_i and its worst-case execution time e_i. An overview of techniques and tools to obtain the worst-case execution time of tasks is given in [WEE+08]. More flexible models for real-time tasks can also be found in the literature [Gre93, MC96, MC97, Bar98, BCGM99, Bar03, ABS06], however, this simple periodic task model (based on Liu and Layland's pioneer work [LL73]) still covers most practical design needs. In case of sporadic tasks, i.e., tasks released with certain non-periodic patterns, p_i may also represent the minimum separation between two consecutive releases of T_i. On the other hand, there are more sophisticated ways of modeling sporadic tasks [Gre93, BCGM99], but this escapes the focus of this thesis.

In principle, a task T_i is an infinite succession of jobs J_{ih}. All jobs of T_i have the same worst-case execution time and relative deadline. Additionally, each job has its own release time t_{ih} and absolute deadline $t_{ih} + d_i$.

For this thesis, we make no assumptions concerning the events that trigger jobs of a task besides that they are periodic or present at least a minimum separation between two occurrences. For this reason, the case where all events occur simultaneously cannot be excluded. As a consequence, in order to consider the worst-case scheduling situation, we need to assume that jobs of all tasks are released together. The simultaneous release of all tasks is often referred to as synchronous case, for which $\mathbf{T}_n$ is called synchronous task set. In this thesis, unless otherwise explicitly stated, we consider that $\mathbf{T}_n$ is a synchronous task set and that jobs of all tasks are released simultaneously at the beginning of the schedule (at time $t = 0$).

The ratio $u_i = \frac{e_i}{p_i}$ is called T_i's task utilization. As discussed later, the task utilization is a key concept in the theory of real-time systems. Further, for the remainder of this thesis, all relative

deadlines d_i are assumed to be arbitrary; they can be less as well as greater than the respective periods p_i for $1 \leq i \leq n$. This assumption has a substantial effect on the feasibility test and is the main difference to the task model applied by Liu and Layland [LL73].

As mentioned above, dependencies among tasks are also going to be considered, so that the task model will have to be adapted to this case. However, the extension of the task model to include task dependencies will not be analyzed until Chapter 4.

1.2.2 Processor Model

Processors are the ones in charge of executing the real-time tasks within a task set $\mathbf{T}_n$. Let us denominate by $\mathbf{P}_q$ the set of processors P_f on which $\mathbf{T}_n$ is going to be executed. Here, q represents the number of processors in $\mathbf{P}_q$.

A given processor type $\mathcal{P}$ is implicitly modeled within the discussed task model by means of the worst-case execution time e_i of tasks. The worst-case execution time depends on the processor type and must be calculated assuming that tasks are going to run on a given processor type $\mathcal{P}$. If tasks then run on another processor type than the originally assumed, their worst-case execution times must be recalculated to consider the new processor type.

Further, when considering multiprocessor systems, there are in principle three different possible scenarios: identical, uniform and heterogeneous processors. In the case of identical processors, a task T_i presents the same worst-case execution time e_i on any processor P_f in $\mathbf{P}_q$. On uniform multiprocessors, the worst-case execution times of tasks in $\mathbf{T}_n$ vary uniformly from processor to processor. Two different tasks T_i and T_j that belong to $\mathbf{T}_n$ have the worst-case execution times e_i and e_j respectively on a processor P_f. The execution time of these tasks on another uniform processor P_g is given by $\alpha_{fg} \cdot e_i$ and $\alpha_{fg} \cdot e_j$ respectively, where α_{fg} is a constant representing the speed-up between P_f and P_g. Clearly, if a multiprocessor platform can be considered to be uniform depends not only on the processor types, but also on the characteristics of tasks. On the other hand, when processors are heterogeneous, the worst-case execution times of tasks on a given P_g cannot be derived from the tasks' execution times on P_f.

In this thesis, in order to perform a task allocation, we focus on identical processors. However, all proposed allocation algorithms can easily be extended to consider uniform processors. In the case of heterogeneous processors, a vector of wost-case execution times instead of a scalar value e_i will be necessary to model a task T_i. Each entry in this vector must contain the worst-case execution time of T_i on a different processor type of the heterogeneous platform.

1.3 Structure of this thesis

This thesis is organized in three main chapters presenting the most important contributions. The next two chapters are concerned with independent real-time tasks, while Chapter 4 considers task communication and system constraints. In Chapter 2, different scheduling policies

are analyzed, for which novel feasibility tests are derived. In particular, we contemplate the most common real-time scheduling algorithms like Deadline Monotonic and Earliest Deadline First. Further, an asynchronous time-triggered scheduling is also considered on the processor. Additionally, for the different scheduling policies, it is shown by means of a thorough statistical comparison that the proposed feasibility tests are more accurate on average than known approaches from the literature.

Chapter 3 is concerned with the allocation of independent real-time tasks. For this purpose, we analyze the simpler case where deadlines are all equal to periods in order to identify the most convenient heuristics independently of the scheduling case. Afterwards, the feasibility test approaches of Chapter 2 are applied for the more general case of arbitrary deadlines. Further, an exhaustive comparison reveals that combining the proposed methods from Chapter 2 with known heuristics results in an improved task allocation, i.e., a bigger reduction of the number of processors can be achieved.

In addition, Chapter 4 considers the modeling of task communication and system constraints. For the case of communicating tasks, some new allocation heuristics are proposed. In contrast to the algorithms of Chapter 3, the presented heuristics are designed in Chapter 4 to reduce the amount of communication between processors and to perform both a reduction of communication and of the number of processors. All heuristics are extensively compared to each other and to the worst-case communication conditions, so as to identify which of these performs the best. Finally, Chapter 5 summarizes the main contributions of this thesis and presents some ideas for future work.

2 Testing Feasibility for Real-Time Tasks

When allocating real-time tasks to processors, we must take special care in ensuring that all tasks meet their deadlines. So, every time a task is assigned to a processor, we have to prove that neither the new assigned task nor the tasks already running on the processor miss their deadlines. Otherwise, the resulting task set will not be feasible (or schedulable) on that processor and correspondingly the whole system may start malfunctioning.

As a consequence, an allocation algorithm for real-time tasks consists of two interlocked components: the allocation procedure itself and the feasibility test. The allocation procedure is the one in charge of deciding to which processor each task should be assigned, whereas the feasibility test proves whether these assigned tasks are schedulable on that processor or not. In this chapter, we analyze how to improve the feasibility tests for real-time tasks, while the allocation procedure will be discussed in later chapters.

In order to test feasibility, we first need to identify the scheduling paradigm under which the tasks are to be scheduled. For the sake of completeness, we analyze both the time-triggered as well as the event-triggered scheduling approaches. The time-triggered scheduling is going to be analyzed assuming that no synchronization takes place, which is rather pessimistic and differs from the common assumptions made [Kop98]. However, the main topic in this thesis is the event-triggered approach, for which fixed-priority and dynamic-priority algorithms are considered.

Second, it is necessary to agree on the model used for describing tasks. In this chapter, we apply the periodic task model already discussed. We further assume that a set $\mathbf{T}_n$ of n periodic tasks is to be scheduled on a processor P_f, where each task T_i is fully preemptive and independent. Unless otherwise stated, the task set $\mathbf{T}_n$ is assumed to be synchronous, i.e., tasks' initial release times are zero for all tasks. Additionally, deadlines are considered to be arbitrary, i.e., they can be less as well as greater than the respective periods.

In this thesis, we focus on allocation algorithms with polynomial complexity for the case of arbitrary deadlines. That is also the reason why we are interested in feasibility tests with polynomial complexity. Unfortunately, for tasks with arbitrary deadlines, forcing the complexity to be polynomial reduces the accuracy of tests, i.e., they become pessimistic. As a consequence, all the polynomial-time tests we can obtain for this case are sufficient but not necessary.

We believe, however, that polynomial complexity is still desirable. This is not only because an exact feasibility test is not really useful when considering some uncertainty in task parameters (which is very common in real-world applications), but also because of the more predictable

and faster running time of algorithms. Moreover, a faster allocation algorithm makes it possible to quickly explore the influence of varying task parameters on system requirements.

The use of the proposed feasibility tests is of course not limited to the allocation problem alone. In particular, algorithms with complexity $\mathcal{O}(n)$, where n is the number of tasks, are also suitable for admission control/on-line testing. This is because testing an additional task on a running system takes only a constant time. Further, they do not need to know all tasks in advance (no presorting is required).

2.1 The Time-Triggered Scheduling Approach

In the time-triggered scheduling approach, jobs of a task T_i cannot simply interrupt the scheduler at an arbitrary point in time, but they must wait for their previously assigned time slots in order to run. The processing time on a processor P_f is normally organized in scheduling cycles of a given length z_f, see Figure 2.1. Additionally, these scheduling cycles are divided into time slots s_i. As already mentioned, processor time is assigned to tasks in form of slots, so that every task T_i running on P_f gets its own time slot.

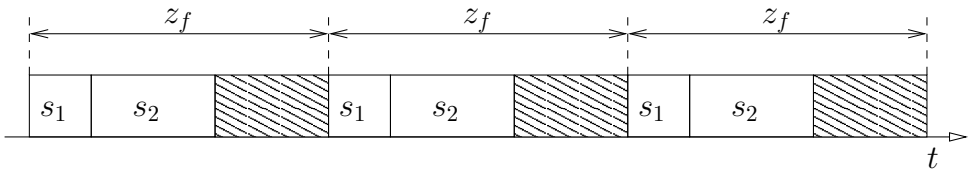


Figure 2.1: Scheduling cycle and time slots on P_f

Whenever J_{ih}, an h-th job of T_i, gets ready, it must wait for its corresponding slot s_i to run, even if the processor gets idle in the meantime. Further, if there is no synchronization between external events and the scheduling on P_f, we need to determine the worst-case release time for jobs of T_i. This latter is important for testing whether T_i's jobs are always going to meet their deadlines or not. Now, the worst-case release time of J_{ih} depends on how the scheduler is implemented. In principle, there are two different possible situations:

1. Jobs of T_i can start executing at any time within the corresponding slot s_i;
2. Jobs of T_i can only start executing at the beginning of the corresponding slot s_i.

In the first situation, it is clear that the worst-case release time for J_{ih} is exactly after s_i has finished. This is because, in this case, J_{ih} will have to wait the most to run: $z_f - s_i$ time units. As a consequence, if $s_i = e_i$ holds (i.e., the slot is equal to the job's execution demand), the following condition must also hold for T_i to be feasible on P_f:

$$e_i \leq z_f \leq \min(d_i, p_i). \tag{2.1}$$

According to Inequality (2.1), the processor scheduling cycle z_f should always be less than or equal to $\min(d_i, p_i)$ of any T_i that runs on it. By considering $\min(d_i, p_i)$ instead of d_i, we assure

that no job backlog is possible, i.e., there is always only one active job of T_i at a time. Recall that the deadline d_i can be longer than the period p_i for every possible T_i.

In the second situation, however, J_{ih} may only start being executed if it is ready by the time at which s_i begins. In such a case, the worst-case release time of J_{ih} will be exactly after s_i has begun because the job will have the maximum waiting time to run: a whole cycle z_f. Now, if $s_i = e_i$ holds, the condition expressed in (2.1) must be modified to the following one:

$$e_i \leq z_f \leq \min(d_i, p_i) - e_i. \tag{2.2}$$

As it can be seen, this latter condition yields $2e_i \leq \min(d_i, p_i)$. In accordance with Inequality (2.2), the processor scheduling cycle z_f should always be less than or equal to $\min(d_i, p_i) - e_i$ of any T_i assigned to it. Again, considering $\min(d_i, p_i)$ instead of d_i allows assuring that there is no job backlog.

The conditions in (2.1) and (2.2) express, respectively according to the scheduler implementation, that the processor P_f must have enough time to execute any J_{ih} before its deadline d_i in order that the task T_i can be feasibly scheduled.

2.1.1 The Minimum Possible Slot

Forcing s_i to be equal to T_i's execution demand e_i allows having only one slot for T_i and reducing in this way the effect of context switches. In this case, the overhead due to context switches can simply be considered in e_i without additional analysis. However, having only one slot for T_i might not necessarily be the most efficient way to proceed, because we use at once too much space in z_f instead of distributing T_i's execution demand over some scheduling cycles when possible. This is the reason why it might be often necessary to find the minimum possible value for s_i.

Let us assume that T_i has to run on P_f which has a scheduling cycle z_f. Again, we have to analyze the two scheduling situations mentioned before in order to determine the length of the time slot s_i for T_i. In the first situation, where jobs can start executing at any time within the slot, we try to find a positive integer κ_i such that the following holds:

$$s_i = \frac{e_i}{\kappa_i}, \tag{2.3}$$

$$\kappa_i \cdot z_f \leq \min(d_i, p_i). \tag{2.4}$$

Notice that Inequality (2.4) is a more general expression of (2.1). Both s_i and κ_i are unknown, however, κ_i can be easily calculated as follows in order to get s_i:

$$\kappa_i = \left\lfloor \frac{\min(d_i, p_i)}{z_f} \right\rfloor. \tag{2.5}$$

Consider the second situation where jobs can only start executing at the beginning of the slot. Now, we have to find a positive integer number κ_i such that this time the following conditions hold:

$$s_i = \frac{e_i}{\kappa_i}, \tag{2.6}$$

$$\kappa_i \cdot z_f + s_i \leq \min(d_i, p_i). \tag{2.7}$$

Here again, Inequality (2.7) is just a more general expression of (2.2). By replacing Equation (2.6) in Inequality (2.7), reshaping and equalizing to zero, we obtain a quadratic equation of κ_i:

$$\kappa_i^2 \cdot z_f - \kappa_i \cdot \min(d_i, p_i) + e_i = 0. \tag{2.8}$$

We first solve Equation (2.8) for its roots. Then, if these roots are real numbers, we need to round up the greatest positive value to obtain κ_i:

$$\kappa_i = \left\lfloor \frac{t_i + \sqrt{t_i^2 - 4z_f \cdot e_i}}{2z_f} \right\rfloor, \tag{2.9}$$

where t_i represents $\min(d_i, p_i)$. Clearly, in the case that none of the roots is a real positive number, T_i is not going to be feasible on P_f.

2.1.2 Context Switches

By choosing the minimum possible slot for T_i, some more context switches are going to happen than for the trivial case $s_i = e_i$. So, we might have sometimes to take the effect of context switches into account if this is not negligible.

Let us consider the overhead due to a context switch to be denoted by σ_f on processor P_f, i.e., σ_f is sum of the time necessary to store and restore a process or job on P_f. Inequality (2.7) must now be modified as follows:

$$\kappa_i \cdot z_f + s_i + (\kappa_i + 1) \cdot \sigma_f \leq \min(d_i, p_i). \tag{2.10}$$

Replacing s_i by $\frac{e_i}{\kappa_i}$ in Inequality (2.10), considering $t_i = \min(d_i, p_i)$ and reshaping, we get:

$$\kappa_i^2 \cdot (z_f + \sigma_f) - \kappa_i \cdot (t_i - \sigma_f) + e_i = 0. \tag{2.11}$$

Finally, the previous quadratic equation can be solved as previously for κ_i:

$$\kappa_i = \left\lfloor \frac{t_i - \sigma_f + \sqrt{(t_i - \sigma_f)^2 - 4(z_f + \sigma_f) \cdot e_i}}{2(z_f + \sigma_f)} \right\rfloor . \tag{2.12}$$

2.1.3 Feasibility Test

The great advantage of a time-triggered scheduling is that it is relatively easy to guarantee its correctness. A task set $\mathbf{T}_n$ can be feasibly scheduled on P_f, if an s_i can be found for every $T_i \in \mathbf{T}_n$ so that either the condition of (2.1) or (2.2) holds—according to the scheduler implementation—and the following inequality also holds true where n is the number of tasks in $\mathbf{T}_n$:

$$\begin{aligned} S_f = \sum_{i=1}^{n} s_i &\leq z_f, \\ \frac{S_f}{z_f} = \frac{\sum_{i=1}^{n} s_i}{z_f} &\leq 1. \end{aligned} \tag{2.13}$$

2.2 The Event-Triggered Scheduling Approach

Another possibility for scheduling real-time tasks is the event-triggered approach. In the event-triggered approach, jobs interrupt the scheduler, whenever they get ready to run or they finish executing. This way, every time an interrupt occurs, the scheduler decides which one among all ready jobs is executed next. The scheduler takes decisions based on the jobs' priority, so that the job with the highest priority is the one that is executed first. From an intuitive point of view, the event-triggered approach is more efficient than the time-driven approach because the processor cannot idle as long as there are ready jobs to execute.

The concept of priority is not relevant in time-driven schedulings, but it is crucial when the scheduling is driven by events. If jobs of task T_i present always the same priority along the schedule, i.e., the priority assignment does not change, we have a so-called fixed-priority scheduling. On the other hand, we have a dynamic-priority scheduling, when the priority assignment may change throughout the schedule. That is when jobs of a task T_i may assume different priorities along the schedule. As an example for dynamic-priority scheduling algorithms, we are going to consider EDF (Earliest Deadline First), whereas we consider a combination of DM (Deadline Monotonic) and RM (Rate Monotonic) for fixed priorities as explained later.

Although dynamic-priority scheduling algorithms are optimal on uniprocessors (the optimality on a uniprocessor was proven by Dertouzos for the case of EDF in [Der74]), fixed-priorities are normally preferred in the real world. A possible reason for this preference is certainly that fixed-priority scheduling algorithms are easier to implement and to integrate into an operating system. Another reason is probably that their behavior seems to be more predictable. An interesting

comparison between RM and EDF was presented by Buttazzo in [But05]. Buttazzo considered here characteristics of these algorithms with respect to overload and context switches among others.

There are also other dynamic-priority algorithms, e.g., the LST algorithm (Least Slack Time First). The LST algorithm is like EDF optimal on uniprocessors [Liu00]. However, EDF is unlike LST a job-level fixed-priority algorithm, which makes it more efficient with respect to context switches. This latter decisive property of EDF is the reason why we chose it for our analysis on real-time scheduling.

As already mentioned, we chose a combination of DM and RM for analyzing fixed-priority schedulings. The reason for this choice is that DM is the most efficient fixed-priority algorithm, i.e., the fixed-priority algorithm that allows a higher processor utilization when $d_i \leq p_i$ for all tasks [LW82]. As we have arbitrary deadlines, we believe it is more meaningful to use RM priorities for tasks with $d_i > p_i$. So, tasks that are triggered more frequently get higher priority when their deadlines are longer than their periods. In this way, we combine DM and RM so that priorities are assigned according to DM for $d_i \leq p_i$ and according to RM for $d_i > p_i$. The result is an algorithm which assigns priorities according to $\min(d_i, p_i)$, that is, the task T_i with the minimum $\min(d_i, p_i)$ is the one with the highest priority and so forth.

2.2.1 The EDF Scheduling

The Earliest Deadline First algorithm assigns priorities to individual jobs of a task T_i according to the jobs' absolute deadlines. At any point in the schedule, the job with the nearest deadline has the highest priority. As the priority assigned to a job of T_i may change along the schedule, EDF is a dynamic-priority scheduling algorithm.

Although exact feasibility tests are already known for arbitrary deadlines, e.g., [BMR90] and [GRS96], they all present pseudo-polynomial complexity. For the purpose of this thesis, we prefer, however, a polynomial complexity because of the more predictable and faster running time.

As stated before, in order to achieve polynomial complexity when testing feasibility for arbitrary deadlines under EDF, some accuracy must be sacrificed. Following this principle, some polynomial-time tests have been already proposed. Among them, we can mention the density test [SSRB98], [Liu00] and Devi's test [Dev03].

The density test is with $\mathcal{O}(n)$ the less complex polynomial-time test known. On the other hand, by performing an initial sorting of tasks, Devi proved in [Dev03] that an interesting accuracy improvement over the density test is also possible. The complexity of Devi's test becomes $\mathcal{O}(n \log n)$ because of sorting tasks at the beginning.

In our case, since the feasibility test should be part of an allocation algorithm, we are normally interested in keeping the complexity as low as possible. As a consequence, we focus on increasing accuracy of feasibility tests for tasks with arbitrary deadlines under EDF while the

complexity remains $\mathcal{O}(n)$. We further propose two algorithms, both with complexity $\mathcal{O}(n)$, that outperform the density test while they do not require an initial sorting of tasks.

The Principles

In this section, we discuss the principles on which the proposed feasibility tests are based. For the sake of clarity, we reproduce at this point some relevant related work. First, let us recall that an exact feasibility test can be performed as follows in polynomial time, when tasks are synchronous, scheduled preemptively on a uniprocessor and the deadline d_i is equal to the period p_i for every task [LL73]:

$$U_n = \sum_{i=1}^{n} \frac{e_i}{p_i} \leq 1. \tag{2.14}$$

Here, n is the number of tasks, e_i is the worst-case execution time and U_n is called processor utilization. For arbitrary deadlines, however, all exact feasibility tests present pseudo-polynomial complexity. In order to reach polynomial complexity for this case, exactness must be sacrificed. Based on this idea, Liu in [Liu00] and Stankovic et al. in [SSRB98] propose independently the density test, which results by replacing p_i by $\min(d_i, p_i)$ in Inequality (2.14):

$$\delta_n = \sum_{i=1}^{n} \frac{e_i}{\min(p_i, d_i)} \leq 1. \tag{2.15}$$

For the sake of clarity, the following two lemmas restate theorems presented respectively by George et al. [GRS96] and Devi [Dev03]. Before starting, let us represent by $h_n(t)$ the demand bound function of $\mathbf{T}_n$ under EDF, which was defined by Baruah et al. in [BMR90]:

$$h_n(t) = \sum_{i=1}^{n} \max\left(0, \left\lfloor \frac{t - d_i}{p_i} \right\rfloor + 1\right) \cdot e_i. \tag{2.16}$$

The demand bound function of Equation (2.16) gives the maximum execution requirement of $\mathbf{T}_n$ under EDF at a given point in time t. As a consequence, the task set $\mathbf{T}_n$ is going to be feasible if $h_n(t) \leq t$ holds, for every possible t. That is, for $\mathbf{T}_n$ to be feasible, the execution demand of $\mathbf{T}_n$ must always be less than or equal to the available time.

LEMMA 1 *If the schedule of a given synchronous task set $\mathbf{T}_n$ is not feasible, i.e., a deadline is missed at t_{miss}, then $t_{miss} < I_n$ holds, where $I_n = \frac{\sum_{i=1}^{n}(p_i - \min(d_i,p_i))\cdot u_i}{1-U_n}$ is George's bound and $u_i = \frac{e_i}{p_i}$ [GRS96].*

Proof: We know from [LL73] that if a deadline is missed at t_{miss} in the synchronous case, there is no idle time previous to t_{miss}. Further, because a deadline is missed at t_{miss}, the total

execution demand of $\mathbf{T}_n$ at t_{miss} is greater than the available time t_{miss}, i.e., $t_{miss} < h_n(t_{miss})$ where $h_n(t)$ is $\mathbf{T}_n$'s demand bound function under EDF [BMR90]:

$$t_{miss} < \sum_{i=1}^{n} \max\left(0, \left\lfloor \frac{t - d_i}{p_i} \right\rfloor + 1\right) \cdot e_i.$$

Considering $\min(p_i, d_i)$ instead of d_i, we obtain:

$$t_{miss} < \sum_{i=1}^{n} \left(\left\lfloor \frac{t_{miss} - \min(p_i, d_i)}{p_i} \right\rfloor + 1\right) \cdot e_i.$$

Further, removing the floor function, we reach:

$$t_{miss} < \sum_{i=1}^{n} \left(\frac{t_{miss} - \min(p_i, d_i)}{p_i} + 1\right) \cdot e_i.$$

Reshaping this inequality to obtain t_{miss}, we get:

$$t_{miss} < \frac{\sum_{i=1}^{n} (p_i - \min(d_i, p_i)) \cdot u_i}{1 - U_n}, \tag{2.17}$$

where $u_i = \frac{e_i}{p_i}$ and U_n is the total utilization. The right member of Inequality (2.17) is known as George's feasibility bound I_n. As it can be observed, George's bound is always greater than the time instant at which a deadline is missed. □

LEMMA 2 *Let $\mathbf{T}'_n$ be a task set of n preemptive, asynchronous, periodic tasks, with arbitrary relative deadlines, arranged in order of non-decreasing relative deadlines. $\mathbf{T}'_n$ is schedulable using an optimal scheduling algorithm if the following inequality holds for all l for which $1 \leq l \leq n$ [Dev03]:*

$$\sum_{i=1}^{l} \frac{e_i}{p_i} + \frac{1}{d_l} \sum_{i=1}^{l} \left(\frac{p_i - \min(p_i, d_i)}{p_i} \right) \cdot e_i \leq 1. \tag{2.18}$$

We do not include the proof of Lemma 2, for which the reader is referred to [Dev03]. The following lemma is a generalization of the one presented in [MDF08] and demonstrates the tight relation between Devi's condition and George's feasibility bound.

LEMMA 3 *Assuming that tasks in the asynchronous $\mathbf{T}'_n$ are sorted according to non-decreasing relative deadlines, Devi's condition is equivalent to an iterative calculation of George's feasibility bound for the corresponding synchronous $\mathbf{T}_n$, for which phases of $\mathbf{T}'_n$ were simply set to zero. The task set $\mathbf{T}'_n$ is feasible, if George's bound for $\mathbf{T}_n$'s first l tasks is less than or equal to d_l for every possible l, where $1 \leq l \leq n$ holds.*

Proof: It is assumed that tasks are sorted according to non-decreasing relative deadlines, so if $i < j$ holds, $d_i \leq d_j$ will also hold. Devi's condition states that $\mathbf{T}'_n$ is feasible if the following inequality holds for every possible l, where $1 \leq l \leq n$ [Dev03]:

$$\sum_{i=1}^{l} \frac{e_i}{p_i} + \frac{1}{d_l} \sum_{i=1}^{l} \left(\frac{p_i - \min(p_i, d_i)}{p_i} \right) \cdot e_i \leq 1.$$

Reordering terms, we get $\sum_{i=1}^{l} \left(\frac{p_i - \min(p_i, d_i)}{p_i} \right) \cdot e_i \leq \left(1 - \sum_{i=1}^{l} \frac{e_i}{p_i} \right) \cdot d_l$, and finally:

$$\frac{\sum_{i=1}^{l} (p_i - \min(p_i, d_i)) \cdot u_i}{1 - U_l} \leq d_l.$$

Where $U_l = \sum_{i=1}^{l} \frac{e_i}{p_i}$ and $u_i = \frac{e_i}{p_i}$, i.e., the left-hand side of this inequality is George's bound I_l for the first l tasks of the corresponding synchronous $\mathbf{T}_n$ where the initial release times (or phases) in $\mathbf{T}'_n$ are ignored. The lemma follows. □

Notice that Lemma 3 shows another way of proving Devi's condition, i.e., through the iterative calculation of George's feasibility bound. The following lemma proves that Devi's condition remains valid even if the task set is not sorted according to non-decreasing relative deadlines.

LEMMA 4 *Given a task set $\mathbf{T}'_n$ of n preemptive, asynchronous, periodic tasks, with arbitrary relative deadlines. $\mathbf{T}'_n$ is feasible under EDF if the following inequality holds for all l for which $1 \leq l \leq n$:*

$$\sum_{i=1}^{l} \frac{e_i}{p_i} + \frac{1}{d_l} \sum_{i=1}^{l} \left(\frac{p_i - \min(p_i, d_i)}{p_i} \right) \cdot e_i \leq 1. \tag{2.19}$$

Proof: Notice that this lemma assumes no particular order for the tasks in $\mathbf{T}'_n$. Let us first consider what happens for $l = 1$. If Inequality (2.19) holds, Lemma 2 guarantees that the task subset of only T_1 is feasible.

Suppose that Inequality (2.19) holds for $l = 2$. It is clear that the task subset of T_1 and T_2 is feasible if $d_1 \leq d_2$ because of Lemma 2. On the other hand, we know from Lemma 3 that Devi's condition is equal to the gradual calculation of George's feasibility bound for the corresponding synchronous $\mathbf{T}_n$. So, if Inequality (2.19) holds for $l = 2$, we have that George's bound I_2 for T_1 and T_2 is less than or equal to d_2. Additionally, Lemma 1 guarantees that $t_{miss} < d_2$. As a consequence, if $d_1 > d_2$ holds, there are no deadline in $(0, d_2)$ that could be missed and the task subset of T_1 and T_2 is also feasible in this case.

For a third task, if now Inequality (2.19) holds, George's bound I_3 for T_1, T_2 and T_3 is less than or equal to d_3, what implies $t_{miss} < d_3$. There is no deadline of T_3 in $(0, d_3)$ that could be missed. Additionally, whatever value d_3 may have, the task subset of T_1 and T_2 alone was proven feasible in the previous step. Consequently, the task subset of T_1, T_2 and T_3 is also feasible regardless of the order of tasks.

Proceeding as previously, let us assume that Inequality (2.19) holds for $1 \leq l \leq n-1$, George's bound $I_l \leq d_l$ holds for $1 \leq l \leq n-1$. If Inequality (2.19) holds for $l = n$ too, $I_n \leq d_n$ also holds. In accordance with Lemma 1, if a deadline miss occurs, $t_{miss} < d_n$ holds. Further, $\mathbf{T}_n$ is feasible because the first $n-1$ tasks were proven to be feasible in the previous steps and there are no deadlines of T_n in $(0, d_n)$ that could be missed. The lemma follows. □

As it can be noticed, Lemma 4 allows reducing the complexity of Devi's test from $\mathcal{O}(n \log n)$ to $\mathcal{O}(n)$. However, Lemma 4 results in a feasibility test that is extremely pessimistic as it is shown later.

In general, an asynchronous task set $\mathbf{T}'_n$ is feasible if its corresponding synchronous task set $\mathbf{T}_n$ is feasible where the initial release times in $\mathbf{T}'_n$ are not considered [SSRB98]. As a consequence, the remainder of this section concentrates on synchronous task sets.

The following lemma is about finding the maximum loading factor for a group of two tasks scheduled under EDF. As already mentioned, the loading factor is defined as the total execution demand in a given time interval over the length of this interval. Further, the maximum loading factor is the upper bound on the loading factor. Two tasks are feasible together if their maximum loading factor is less than or equal to 1.

LEMMA 5 *Let $\mathbf{T}_{xy}$ be a subset of two arbitrary tasks T_x and T_y from $\mathbf{T}_n$ scheduled under EDF, where $t_y = \min(d_y, p_y)$, $t_x = \min(d_x, p_x)$ and $t_y \leq t_x$ hold. The loading factor $\rho_{xy}(t)$ of the subset $\mathbf{T}_{xy}$ is bounded above by $\hat{\rho}_{xy} = \max\left(\frac{e_y}{t_y}, \frac{e_{xy}}{t_x}, \frac{(p_x - tx)\cdot\frac{e_x}{p_x} + (p_y - ty)\cdot\frac{e_y}{p_y}}{t_{yk}} + \frac{e_x}{p_x} + \frac{e_y}{p_y}\right)$, where $e_{xy} = e_x + k \cdot e_y$, $t_{yk} = \min(d_y, p_y) + k \cdot p_y$ and k is given by $\lfloor \frac{t_x - t_y}{p_y} \rfloor + 1$.*

Proof: The loading factor of $\mathbf{T}_{xy}$ is defined as the ratio $\rho_{xy}(t) = \frac{h_{xy}(t)}{t}$, i.e., the subset's execution demand in the interval $(0, t]$ over the interval's length t:

$$\rho_{xy}(t) = \frac{\max\left(0, \lfloor \frac{t-d_x}{p_x} \rfloor + 1\right) \cdot e_x + \max\left(0, \lfloor \frac{t-d_y}{p_y} \rfloor + 1\right) \cdot e_y}{t}.$$

By considering $t_y = \min(d_y, p_y)$ and $t_x = \min(d_x, p_x)$, we can get rid of the $\max$ function:

$$\rho_{xy}(t) \leq \frac{\left(\lfloor \frac{t-t_x}{p_x} \rfloor + 1\right) \cdot e_x + \left(\lfloor \frac{t-t_y}{p_y} \rfloor + 1\right) \cdot e_y}{t}. \tag{2.20}$$

As $t_y \leq t_x$ holds, the right member of Inequality (2.20) is zero for $t < t_y$. So, the first interval that needs to be checked is $(0, t_y]$ in order to find an upper bound for $\rho_{xy}(t)$. Additionally, between t_y and t_x, there can only be jobs of T_y. It is clear that T_y's loading factor cannot exceed $\frac{e_y}{\min(d_y, p_y)}$, so deadlines of T_y in (t_y, t_x) do not need to be checked. The second interval that needs to be tested is clearly $(0, t_x]$:

$$\rho_{xy}(t_x) \leq \frac{e_x + \left(\lfloor \frac{t_x - t_y}{p_y} \rfloor + 1\right) \cdot e_y}{t_x}.$$

The numerator in the previous expression is what we denominated e_{xy} in this lemma. Now, in order to find an upper bound on $\rho_{xy}(t)$, it is required to calculate Inequality (2.20) for all possible time intervals. However, we can approximate $h_{xy}(t)$ in Inequality (2.20):

$$\begin{aligned} \rho_{xy}(t) &\leq \frac{\left(\frac{t-t_x}{p_x} + 1\right) \cdot e_x + \left(\frac{t-t_y}{p_y} + 1\right) \cdot e_y}{t}, \\ &\leq \frac{(p_x - t_x) \cdot \frac{e_x}{p_x} + (p_y - t_y) \cdot \frac{e_y}{p_y}}{t} + \frac{e_x}{p_x} + \frac{e_y}{p_y}. \end{aligned} \tag{2.21}$$

From Inequality (2.21), it can be concluded that our approximation of $\rho_{xy}(t)$ increases as the interval $(0, t]$ decreases. As we have already analyzed intervals up to t_x, we can choose the shortest interval greater than t_x without committing any error. This shortest interval greater than t_x is given by the next deadline of T_y after t_x—considering $t_y = \min(d_y, p_y)$ instead of d_y: $t_{yk} = t_y + k \cdot p_y$, where $k = \lfloor \frac{t_x - t_y}{p_y} \rfloor + 1$. Although there may be jobs of T_x between t_x and t_{yk}, the loading factor for these T_x's jobs cannot exceed the loading factor at $t_x = \min(d_x, p_x)$.

Finally, the greatest of the three $\frac{e_y}{t_y}$, $\frac{e_{xy}}{t_x}$ and $\frac{(p_x - tx) \cdot \frac{e_x}{p_x} + (p_y - ty) \cdot \frac{e_y}{p_y}}{t_{yk}} + \frac{e_x}{p_x} + \frac{e_y}{p_y}$ is going to determine the upper bound $\hat{\rho}_{xy}$ for the loading factor of $\mathbf{T}_{xy}$ and the lemma follows. □

Further, the next two lemmas give upper bounds for the loading factor of several synchronous tasks scheduled under EDF. We will see later that calculating the upper bound on the loading factor of several tasks results in more accurate feasibility tests.

LEMMA 6 *Let $\mathbf{T}_{l-1}$ be the subset of the first $l-1$ tasks from $\mathbf{T}_n$ scheduled under EDF, for which we know the maximum loading factor $\hat{\rho}_{l-1}$. If task T_l, also from $\mathbf{T}_n$, is added to $\mathbf{T}_{l-1}$, the loading factor of the resulting subset $\mathbf{T}_l$ is going to be bounded above by* $\max\left(\hat{\rho}_{l-1}, \frac{\sum_{i=1}^{l}(p_i - \min(d_i, p_i)) \cdot \frac{e_i}{p_i}}{\min(d_l, p_l)} + \sum_{i=1}^{l} \frac{e_i}{p_i}\right)$.

Proof: The loading factor of $\mathbf{T}_{l-1}$ plus T_l is defined as the ratio $\rho_l(t) = \frac{h_l(t)}{t}$.

$$\begin{aligned} \rho_l(t) &= \frac{\sum_{i=1}^{l} \max\left(0, \lfloor \frac{t-d_i}{p_i} \rfloor + 1\right) \cdot e_i}{t}, \\ &\leq \frac{\sum_{i=1}^{l} \left(\frac{t - \min(d_i, p_i)}{p_i} + 1\right) \cdot e_i}{t}. \end{aligned}$$

Finally, we can reshape the previous expression to obtain:

$$\rho_l(t) \leq \frac{\sum_{i=1}^{l} (p_i - \min(d_i, p_i)) \cdot \frac{e_i}{p_i}}{t} + \sum_{i=1}^{l} \frac{e_i}{p_i}. \tag{2.22}$$

If we choose t in Inequality (2.22) to be the least possible, we can get an approximation for the loading factor of $\mathbf{T}_l$. As $\hat{\rho}_{l-1}$ is the maximum loading factor of $\mathbf{T}_{l-1}$, the loading factor for intervals previous to $\min(d_l, p_l)$ is already considered in $\hat{\rho}_{l-1}$. The value of t in Inequality (2.22) can consequently be set to $\min(d_l, p_l)$. Further, the upper bound on $\mathbf{T}_l$'s loading factor is the maximum of the two $\hat{\rho}_{l-1}$ and $\frac{\sum_{i=1}^{l}(p_i - \min(d_i,p_i)) \cdot \frac{e_i}{p_i}}{\min(d_l,p_l)} + \sum_{i=1}^{l} \frac{e_l}{p_l}$. □

LEMMA 7 *Let $\mathbf{T}_{l-1}$ be the subset of the first $l-1$ tasks from $\mathbf{T}_n$ scheduled under EDF, for which we know the maximum loading factor $\hat{\rho}_{l-1}^{sup}$ and that $\min(d_i, p_i) \geq t_{sup}$ holds for all $1 \leq i \leq l-1$. If task T_l, also from $\mathbf{T}_n$, is added to $\mathbf{T}_{l-1}$, where $t_{sup} \geq \min(d_l, p_l)$, the loading factor of the resulting subset is going to be bounded above by $\hat{\rho}_{l-1}^{sup} + \max\left(\frac{k \cdot e_l}{t_{sup}}, \frac{(p_l - \min(d_l,p_l)) \cdot \frac{e_l}{p_l}}{t_{lk}} + \frac{e_l}{p_l}\right)$ for all $t \geq t_{sup}$, where $k = \lfloor \frac{t_{sup} - \min(d_l,p_l)}{p_l} \rfloor + 1$ and $t_{lk} = \min(d_l, p_l) + k \cdot p_l$.*

Proof: The loading factor of $\mathbf{T}_{l-1}$ is at maximum $\hat{\rho}_{l-1}^{sup}$. That is, $\rho_{l-1}(t) = \frac{h_{l-1}(t)}{t} \leq \hat{\rho}_{l-1}^{sup}$ holds for all possible t, what yields $h_{l-1}(t) \leq t \cdot \hat{\rho}_{l-1}^{sup}$. Hence, we can approximate the loading factor of $\mathbf{T}_{l-1}$ plus T_l as shown below:

$$\begin{aligned} \rho_l(t) &\leq \hat{\rho}_{l-1}^{sup} + \frac{\max\left(0, \lfloor \frac{t-d_l}{p_l} \rfloor + 1\right) \cdot e_l}{t}, \\ &\leq \hat{\rho}_{l-1}^{sup} + \frac{\left(\lfloor \frac{t-\min(d_l,p_l)}{p_l} \rfloor + 1\right) \cdot e_l}{t}. \end{aligned} \tag{2.23}$$

As we are interested in finding an upper bound for $\rho_l(t)$ from t_{sup} onwards. The first interval for which we need to calculate Inequality (2.23) is $(0, t_{sup}]$. That is:

$$\rho_l(t_{sup}) \leq \hat{\rho}_{l-1}^{sup} + \frac{\left(\lfloor \frac{t_{sup}-\min(d_l,p_l)}{p_l} \rfloor + 1\right) \cdot e_l}{t_{sup}}.$$

Where $\lfloor \frac{t_{sup}-\min(d_l,p_l)}{p_l} \rfloor + 1$ was denoted by k in this lemma. Further, by getting rid of the floor function in Inequality (2.23) and reordering, we reach:

$$\rho_l(t) \leq \hat{\rho}_{l-1}^{sup} + \frac{(p_l - \min(d_l, p_l)) \cdot \frac{e_l}{p_l}}{t} + \frac{e_l}{p_l}. \tag{2.24}$$

From Inequality (2.24), we know that we need to choose the shortest possible t to get the upper bound on $\rho_l(t)$. However, we do not need to calculate Inequality (2.24) until $t_{lk} =$

$\min(d_l, p_l) + k \cdot p_l$ for $k = \lfloor \frac{t_{sup} - \min(d_l,p_l)}{p_l} \rfloor + 1$. This is because the loading factor for jobs in (t_{sup}, t_{lk}) cannot exceed the one at t_{sup}. Thus, the upper limit for $\rho_l(t)$ and $t \geq t_{sup}$ is given by $\hat{\rho}_{l-1}^{sup} + \max\left(\frac{k \cdot e_l}{t_{sup}}, \frac{(p_l - \min(d_l,p_l)) \cdot \frac{e_l}{p_l}}{t_{lk}} + \frac{e_l}{p_l}\right)$. □

Notice that the order of tasks does neither affect the validity of Lemma 6 nor of Lemma 7, so that these lemmas can be used to design $\mathcal{O}(n)$ feasibility tests for EDF. The following section shows how the different lemmas can be combined for this purpose.

Feasibility Tests for EDF

In this section, we explain two better linear-time feasibility tests for EDF based on the principles already discussed. Both algorithms calculate the maximum loading factor $\hat{\rho}_n$ of the whole task set $\mathbf{T}_n$. So that if $\hat{\rho}_n \leq 1$ holds, $\mathbf{T}_n$ is feasible. In what follows, we assume that there is already a set of $l-1$ tasks running on processor P_f, where $1 \leq l \leq n$, and that a new task T_l should be tested for feasibility on P_f. Clearly, the complexity in this case is only $\mathcal{O}(1)$ because only the new task T_l must be tested.

Figure 2.2 presents the flow chart of the first algorithm we propose, which will be called EDFTest1. In principle, EDFTest1 applies Lemma 5 to compute the loading factor of two tasks together. If one task was accepted, EDFTest1 keeps the task parameters until the next task arrives. This way, the loading factor of the previous and the current task can be calculated. EDFTest1 uses the boolean variable called *previous task* to indicate that the previous task parameters are available. When a new task T_l arrives, EDFTest1 checks up whether previous task parameters are available at the moment. If *previous task* is true, i.e., the previous task parameters are available, EDFTest1 applies Lemma 5 to calculate the maximum loading factor $\hat{\rho}_{xy}$ of previous accepted and the current task. If the maximum loading factor of all other accepted tasks denoted by $\hat{\rho}_{sum}$ plus $\hat{\rho}_{xy}$ is not greater than one, the new task T_l is accepted, $\hat{\rho}_{sum}$ is set to $\hat{\rho}_{sum} + \hat{\rho}_{xy}$ and *previous task* is set to false. On the other hand, if *previous task* is false, this algorithm uses the density condition, i.e., Inequality (2.15), to get the maximum loading factor $\frac{e_l}{\min(d_l,p_l)}$ of the current task T_l alone. If further $\hat{\rho}_{sum} + \frac{e_l}{\min(d_l,p_l)} \leq 1$ holds, T_l is accepted. In this case, T_l's parameters are stored for later calculations, *previous task* is set to true, but $\hat{\rho}_{sum}$ is not updated until another task arrives.

The flow chart for our second algorithm is shown in Figure 2.3. This latter algorithm is called EDFTest2 and applies Lemma 6 and Lemma 7 to calculate the maximum loading factor $\hat{\rho}_n$ of $\mathbf{T}_n$. A particularity of this latter algorithm is that it divides the loading factor into three. In this way, $\hat{\rho}_{inf}$ is the upper bound on the loading factor between t_{inf} and t_{sup} for $t_{inf} \leq t_{sup}$, whereas $\hat{\rho}_{sup}$ is the maximum loading factor from t_{sup} onwards. The loading factor for intervals less than t_{inf} is zero, because $t_{inf} \leq \min(d_l, p_l)$ holds for $1 \leq l \leq n$. So, the maximum of the two $\hat{\rho}_{inf}$ and $\hat{\rho}_{sup}$ is the maximum loading factor of tasks. Before we start discussing this algorithm, let us denote by $\tilde{\rho}_{sup}$, $\tilde{\rho}_{inf}$, $\tilde{t}_{sup}$ and $\tilde{t}_{inf}$ temporal variables for the previously mentioned parameters. These variables are used to store temporal calculations of $\hat{\rho}_{sup}$, $\hat{\rho}_{inf}$, t_{sup} and t_{inf} that are then discarded if the new task is not accepted.

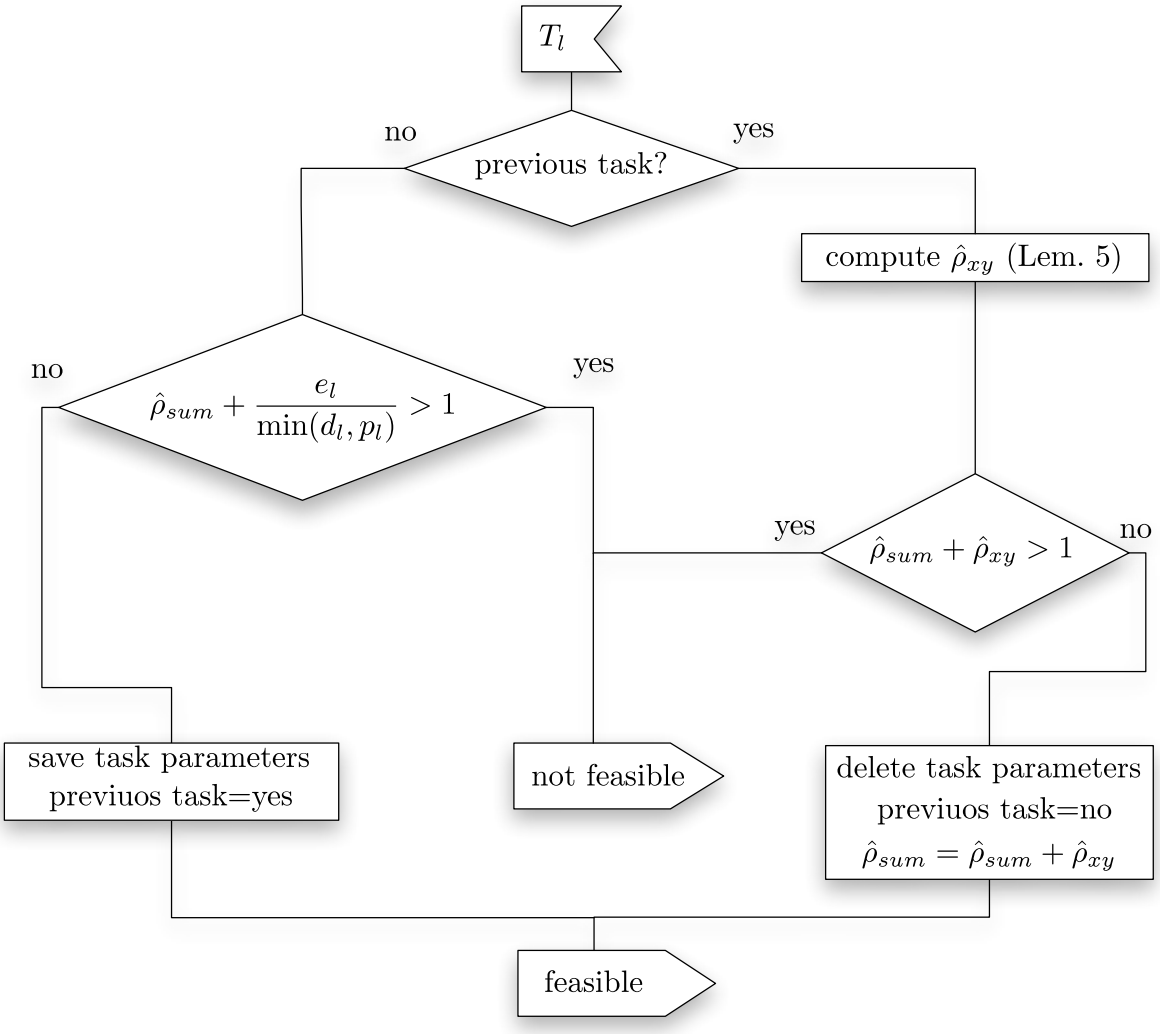


Figure 2.2: Algorithm EDFTest1

For an arriving T_l, if $\min(d_l, p_l) > t_{sup}$ holds, Lemma 6 is applied to obtain $\hat{\rho}_l$ and considering all previously accepted tasks. In this case, $\hat{\rho}_{l-1}$ from Lemma 6 is set to the current value of $\hat{\rho}_{sup}$. Because $\min(d_l, p_l) > t_{sup}$ holds, T_l has only influence on $\hat{\rho}_{sup}$, but not on $\hat{\rho}_{inf}$. For the moment, the maximum value of the two $\hat{\rho}_l$ and $\hat{\rho}_{sup}$ is stored in the temporal variable $\tilde{\rho}_{sup}$.

On the other hand, if $\min(d_l, p_l) < t_{inf}$ holds, i.e., $\min(d_l, p_l)$ is the shortest until now, T_l influences both $\hat{\rho}_{sup}$ and $\hat{\rho}_{inf}$. Consequently, Lemma 7 can be applied to obtain $\hat{\rho}_{inf}$ because $\min(d_l, p_l) < t_{inf}$ holds and $\hat{\rho}_{inf}$ is the maximum loading factor between t_{inf} and t_{sup}. In this case, $\hat{\rho}^{sup}_{l-1}$ from Lemma 7 is set to the current value of $\hat{\rho}_{inf}$ and $\hat{\rho}_{inf}$ is temporarily stored in $\tilde{\rho}_{inf}$. Further, $\hat{\rho}_{sup}$ can be calculated using Lemma 7 because $\min(d_l, p_l) < t_{inf} \leq t_{sup}$ holds and $\hat{\rho}_{sup}$ is the maximum loading factor from t_{sup} on. Now, $\hat{\rho}^{sup}_{l-1}$ from Lemma 7 is set to the current value of $\hat{\rho}_{sup}$ instead and $\hat{\rho}_{sup}$ is stored in the temporal variable $\tilde{\rho}_{sup}$. Further, $\tilde{t}_{inf}$ is set to $\min(d_l, p_l)$, which is the shortest up to the moment, and $\tilde{\rho}_{inf}$ must be set to maximum between $\tilde{\rho}_{inf}$ and the maximum loading factor $\frac{e_l}{\min(d_l, p_l)}$ of the current task T_l alone.

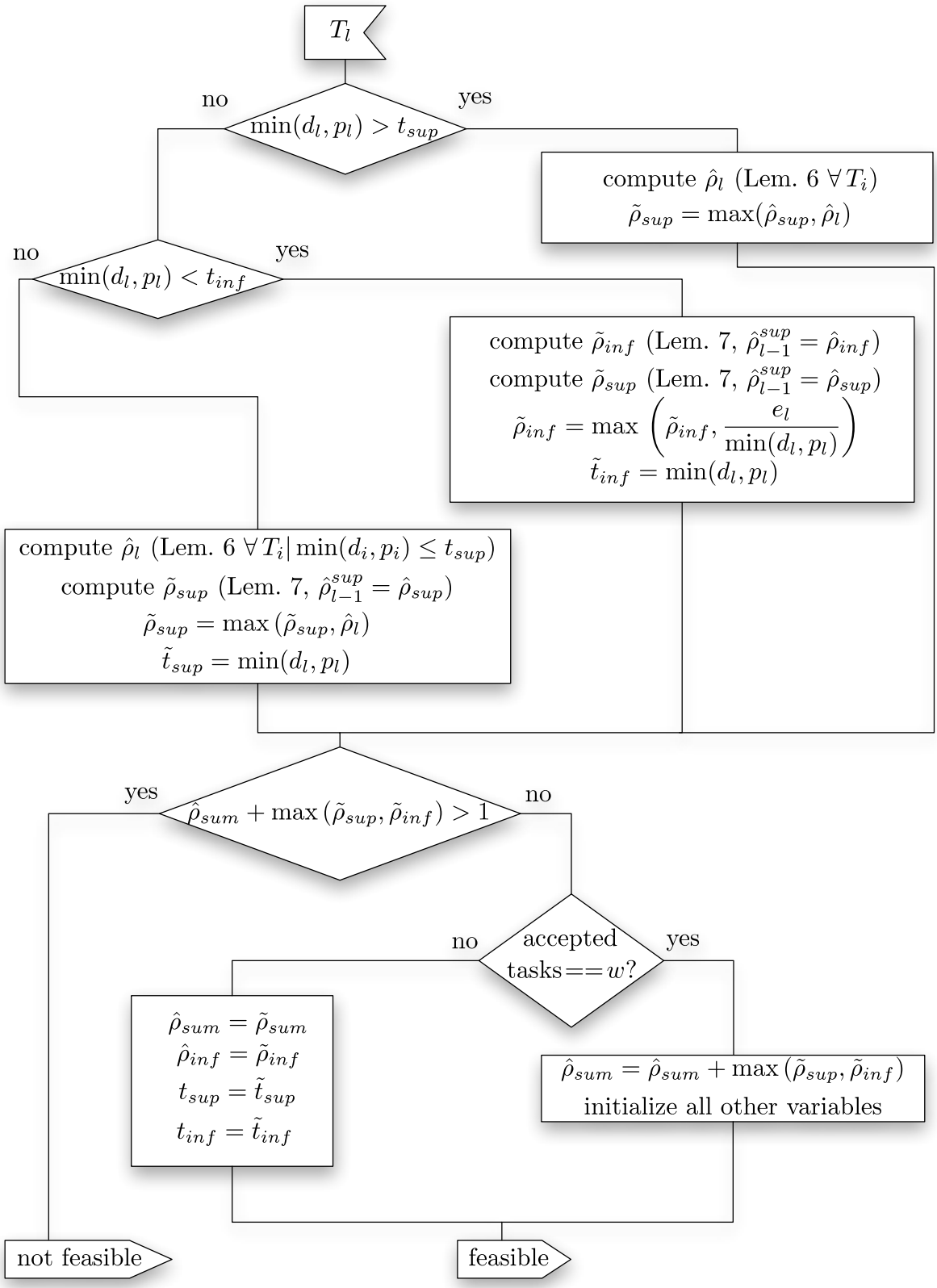


Figure 2.3: Algorithm EDFTest2

In case that $\min(d_l, p_l)$ is between t_{inf} and t_{sup}, $\tilde{t}_{sup}$ is going to be set to $\min(d_l, p_l)$. So, $\hat{\rho}_{sup}$ is the only one which must be recalculated. For this purpose, Lemma 6 is applied to get $\hat{\rho}_l$ considering $\hat{\rho}_{l-1} = \hat{\rho}_{inf}$ and only the tasks for which $\min(d_i, p_i) \leq t_{sup}$ holds. Further, Lemma 7 is used in order to find $\hat{\rho}_{sup}$ considering for this $\hat{\rho}_{l-1}^{sup} = \hat{\rho}_{sup}$ and storing it temporarily in $\tilde{\rho}_{sup}$. As mentioned, $\tilde{t}_{sup}$ is set to $\min(d_l, p_l)$ and $\tilde{\rho}_{sup}$ must be set to the maximum between $\hat{\rho}_l$ and $\tilde{\rho}_{sup}$.

Now, if the maximum loading factor of all other accepted tasks denoted by $\hat{\rho}_{sum}$ plus the maximum of the two $\tilde{\rho}_{sup}$ and $\tilde{\rho}_{inf}$ is not greater than 1, the new task T_l is accepted and the values of temporal variables can be adopted. The computation of $\hat{\rho}_{sup}$ and $\hat{\rho}_{inf}$ can be performed in the proposed way for all tasks. However, for most cases, it results better to calculate them for smaller task groups of w tasks each. When $\hat{\rho}_{sup}$ and $\hat{\rho}_{inf}$ have been calculated for w tasks, the maximum between these two is accumulated in $\hat{\rho}_{sum}$. All variables with exception of $\hat{\rho}_{sum}$ must be initialized at this point in order to start calculating $\hat{\rho}_{sup}$ and $\hat{\rho}_{inf}$ for the next group of w tasks. The variable w can be set to any arbitrary integer value in $1 \leq w \leq n$. The technique used in EDFTest2 can further be extended as shown in [MCF10b] to improve the accuracy of the resulting algorithm (i.e., its capability of detecting schedulable task sets).

Some Experimental Results

In this section, we present and evaluate some experiments comparing the proposed algorithms against the known feasibility tests of the same complexity $\mathcal{O}(n)$. Hence, we compare the two new algorithms with the density test and with Devi's test according to Lemma 4, i.e., Devi's test without initial sorting of tasks. For EDFTest2, we use in this section $w = 5$, i.e., we compute the maximum loading factor for groups of 5 tasks each. Additionally, we include Baruah's pseudo-polynomial-time exact feasibility test from [BMR90] denoted by Baruah's PPT in this comparison.

The mentioned algorithms are contrasted with respect to their accuracy versus utilization for different numbers of tasks per task set. In order to achieve a meaningful comparison, random task sets were uniformly generated for different processor utilizations as recommended in [BB04a] and [BB05]—*UUniFast* was used to generate a set of tasks utilizations u_i.

For each of the presented curves, test data was obtained as follows: Once generated a set of task utilizations u_i as mentioned above, we created periods p_i also in a random way with uniform distribution. Consequently, we got the worst-case execution times by $e_i = u_i \cdot p_i$. The relative deadlines d_i were uniformly chosen from the range $[e_i, p_i]$. Additionally, we sampled the utilization axis at 24 points, for which 1000 different task sets were generated each time.

Figures 2.4 to 2.6 present schedulability curves for 5, 10 and 100 tasks per task set. Remember that the Devi's test in this curves is the one without initial sorting. For tens of tasks Figure 2.4 and 2.5, the proposed algorithms reach around 20% more accepted task sets than the density test in the utilization interval $(30\%, 70\%)$. As the number of tasks grows, the performance of all polynomial-time algorithms, the proposed and the known ones, decays rapidly. Where for 10 tasks the polynomial-time algorithms detect feasible task sets up to 90% utilization, they are

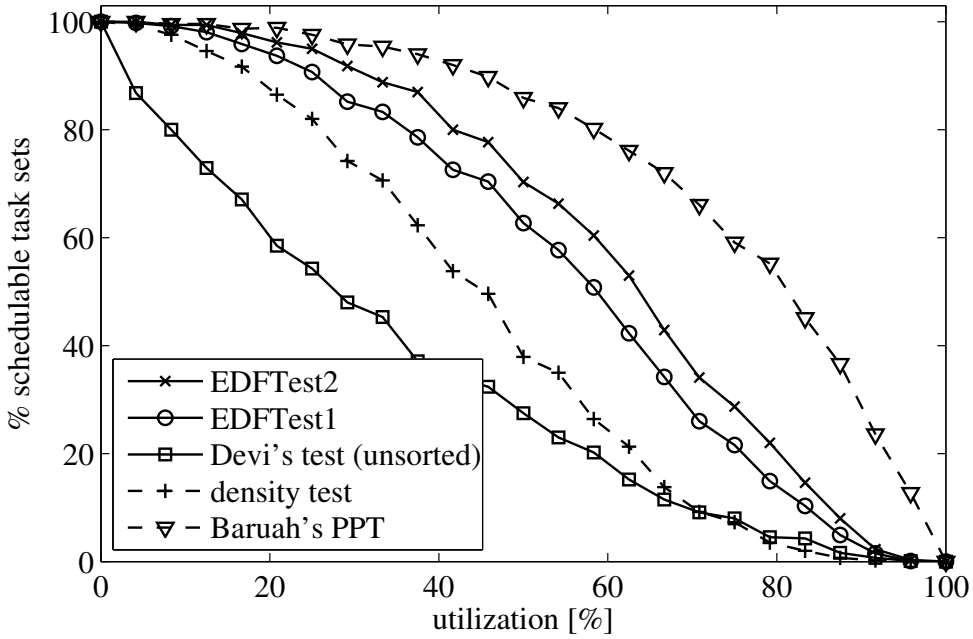


Figure 2.4: Schedulability vs. utilization for 5 tasks

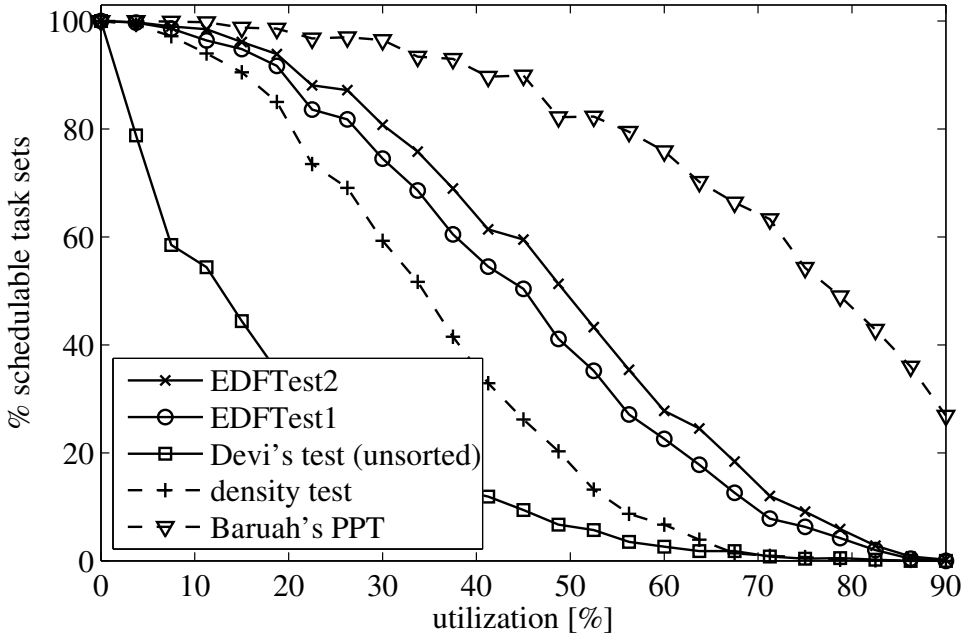


Figure 2.5: Schedulability vs. utilization for 10 tasks

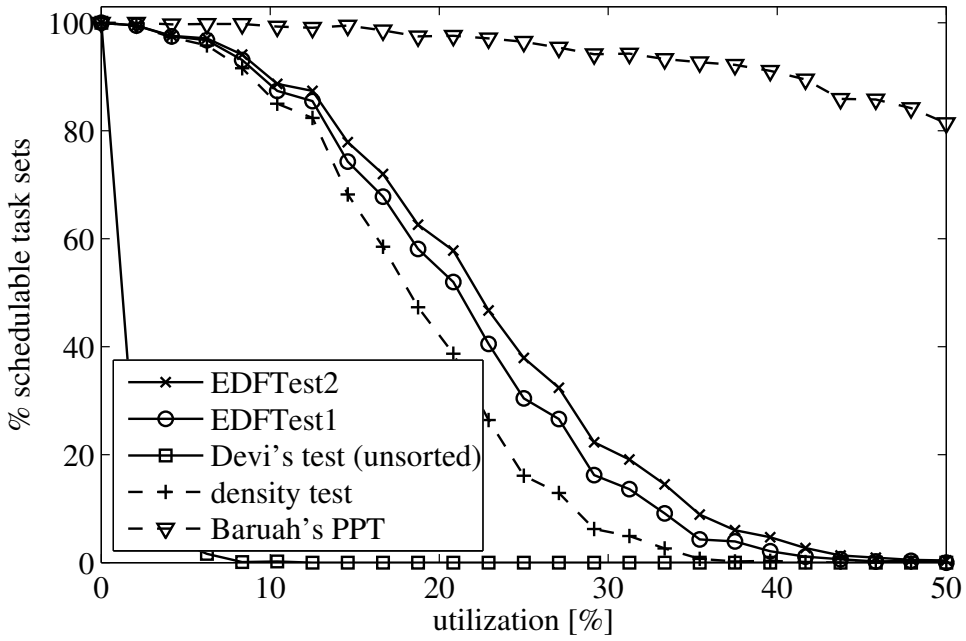


Figure 2.6: Schedulability vs. utilization for 100 tasks

unable to find schedulable task sets for a utilization over 50% and 100 tasks per task set. For 100 tasks, the performance improvement over the density test goes down to around 10% in the utilization range $(15\%, 35\%)$. The algorithm EDFTest2 is always around 5% or 10% better than EDFTest1. The performance of EDFTest1 and EDFTest2 with respect to the exact algorithm can be improved as discussed in [MCF10b].

2.2.2 The DM/RM Scheduling

The combination of Deadline Monotonic (DM) and Rate Monotonic (RM) that is used assigns priorities according to $\min(d_i, p_i)$. A task T_i receives a higher priority than another task T_j if $\min(d_i, p_i) < \min(d_j, p_j)$ holds. For this, DM is used to assign priorities when $d_i \leq p_i$ holds whereas RM is used to assign priorities when $d_i > p_i$ holds.

The DM algorithm assigns priorities according to tasks' relative deadlines. That is, all jobs of the task with the shortest relative deadline are going to have the highest priority throughout the schedule and so on. Under the RM algorithm, priorities are given to tasks according to their rate (the inverse of the period). The task with the highest rate is the one with the highest priority and so forth. Notice that the DM scheduling reduces to RM when deadlines are equal to periods for all tasks. Priorities assigned like this do not change along the schedule, so we talk about fixed priorities.

An exact feasibility test with pseudo-polynomial complexity is also known for arbitrary deadlines and fixed priorities, i.e., [Leh90]. However, as stated previously, we are interested in

polynomial complexity because of the more predictable and faster running time. To reach polynomial complexity when testing feasibility for arbitrary deadlines under fixed priorities, some accuracy gets lost. Analogously to EDF, it is possible to come up with algorithms of complexity $\mathcal{O}(n)$ and $\mathcal{O}(n \log n)$ too.

Here again, we are normally interested in keeping the complexity as low as possible because the feasibility test is going to be part of an allocation algorithm. As a consequence, we focus on increasing accuracy for $\mathcal{O}(n)$ feasibility tests and tasks with arbitrary deadlines under the suggested DM/RM scheduling. For this purpose, we extend the concepts presented for EDF in the previous section so that they can be applied for the DM/RM scheduling as well.

The Principles

In [LL73], Liu and Layland also gave a utilization upper bound for RM schedulings and deadlines equal to periods, which has the following expression:

$$U_{RM}(n) = n \cdot (2^{1/n} - 1), \tag{2.25}$$

where n is the number of tasks. So, if $d_i = p_i$ holds for all tasks and the total utilization is not greater than the bound of Equation (2.25), a task set scheduled under Rate Monotonic is going to be feasible. Unlike Equation (2.14) for EDF, this is a sufficient but not necessary test for the case $d_i = p_i$ and Rate Monotonic. A better utilization upper bound for the given case was presented in [Liu00, BBB01], which additionally does not depend on the number of tasks as the previous one:

$$\prod_{i=1}^{n} \left(1 + \frac{e_i}{p_i}\right) \leq 2. \tag{2.26}$$

Kuo and Mok presented another utilization bound for $d_i = p_i$ and RM schedulings [KM91], however, this latter bound needs to sort tasks in order to exploit the higher utilization that gets possible for harmonic periods. As any sorting yields a complexity of at least $\mathcal{O}(n \log n)$, this bound of Kuo and Mok is less interesting for the purpose of this thesis. In [BLOS95], Burchard et al. proposed a better utilization bound also for $d_i = p_i$ and RM that quantifies how close tasks are to having harmonic periods and does not require any sorting. For arbitrary deadlines, Lehoczky et al. proposed in [Leh90, LS86] a utilization bound that does not only depend on the number of tasks but also on the ratio $\frac{d_i}{p_i}$. Lehoczky et al. assumed this ratio to be same for all T_i.

On the other hand, we can proceed as for EDF and come up with a sort of density test for the DM/RM algorithm, for example, using the bound of Inequality (2.26):

$$\prod_{i=1}^{n} \left(1 + \frac{e_i}{\min(d_i, p_i)}\right) \leq 2. \tag{2.27}$$

The validity of latter test follows immediately from the validity of Inequality (2.26) and needs no further explanation. In the same manner, it is also possible to extend the Liu and Layland bound (i.e., Equation (2.25)), but Inequality (2.27) results in a less pessimistic test. Although we do not go deeper into details, experimental results with a large number of synthetic tasks have shown that the test of Inequality (2.27) is even less pessimistic than Lehoczky's utilization bound [Leh90, LS86] and than Burchard's bound [BLOS95]. So Inequality (2.27) is the best possible linear-time feasibility test for the DM/RM policy, which we can obtain based on techniques from the literature.

The following lemma presents a new polynomial-time test for the DM algorithm, which can also be applied without changes in the context of the DM/RM policy. The presented test is based on calculating the maximum loading factor for each individual task T_i. If the sum of all maximum loading factors is less than or equal to unity, the task set is feasible. First, let us denote by $h_n(t)$ the demand bound function of $\mathbf{T}_n$ under whichever fixed-priority scheduling (DM, RM, DM/RM, etc.) [Leh90]:

$$h_n(t) = \sum_{i=1}^{n} \left\lceil \frac{t}{p_i} \right\rceil \cdot e_i. \tag{2.28}$$

The demand bound function of Equation (2.28) gives the maximum execution requirement of $\mathbf{T}_n$ under fixed priorities at a given point in time t. The task set $\mathbf{T}_n$ is going to be feasible under fixed priorities if $h_n(t) \leq t$ holds for every possible t, where t can be any deadline of the tasks in $\mathbf{T}_n$. In other words, for $\mathbf{T}_n$ to be feasible, the execution demand of $\mathbf{T}_n$ must always be less than or equal to the available time just as in the case of EDF.

LEMMA 8 *Let $\mathbf{T}_n$ be a task set of n preemptive, synchronous, periodic tasks, with arbitrary relative deadlines. $\mathbf{T}_n$ is schedulable under Deadline Monotonic priorities if the following inequality holds:*

$$\sum_{i=1}^{n} \max\left(\frac{e_i}{\min(d_i, p_i)}, 2\frac{e_i}{p_i}\right) \leq 1. \tag{2.29}$$

Proof: The loading factor is given by the execution demand over the available time. So, it is clear that $\mathbf{T}_n$ is going to be feasible, if the loading factor is not greater than 1 throughout its schedule. Further, if $\hat{\rho}_n$ is $\mathbf{T}_n$'s maximum loading factor, i.e., the upper bound for $\mathbf{T}_n$'s loading factor, $\mathbf{T}_n$ is going to feasible if $\hat{\rho}_n \leq 1$ holds.

In order to obtain $\hat{\rho}_n$, we proceed calculating an upper bound on the loading factor of individual tasks. With this purpose, let us consider a single task T_i from $\mathbf{T}_n$. It is clear that the upper bound on T_i's loading factor is given by $\frac{e_i}{\min(d_i,p_i)}$. Furthermore, if $\frac{e_i}{\min(d_i,p_i)} \leq 1$ holds, T_i alone will be feasible/schedulable.

However, T_i is scheduled together with other tasks under DM; it will be preempted by higher priority tasks and it will preempt lower priority tasks. So, some more analysis is required to find an upper bound on T_i's loading factor considering the influence of concurrent tasks.

T_i's execution demand under DM is given by $\lceil \frac{t}{p_i} \rceil \cdot e_i$ [Leh90]. So that T_i's loading factor will be given by: $(\lceil \frac{t}{p_i} \rceil \cdot e_i)/t$. In order to obtain an upper bound on this expression, we remove the ceiling function and reorder terms:

$$\begin{aligned} \frac{\lceil \frac{t}{p_i} \rceil \cdot e_i}{t} &\leq \frac{(\frac{t}{p_i} + 1) \cdot e_i}{t}, \\ &\leq \frac{e_i}{t} + \frac{e_i}{p_i}. \end{aligned} \tag{2.30}$$

From Inequality (2.30), it can be concluded T_i's loading factor gets maximum as t decreases to the minimum possible. Since we assume that the synchronous scheduling of $\mathbf{T}_n$ begins at $t = 0$, there can only be deadlines that are less than d_i in the time interval $(0, d_i)$. These shorter deadlines belong to higher-priority tasks, which cannot be preempted by T_i. As a consequence, T_i's loading factor does not need to be calculated for time intervals that are less than d_i. Furthermore, as we have already discussed, the maximum loading factor due to T_i at $t = d_i$ is clearly given by $\frac{e_i}{\min(d_i, p_i)}$.

On the other hand, tasks with longer deadlines than T_i will be preempted by this latter. From the point of view of lower-priority tasks, T_i's jobs are going to start running as soon as they get ready. Consequently, we need to calculate Inequality (2.30) at $t = p_i$ where the next T_i's job gets ready, which results in $2\frac{e_i}{p_i}$. Hence, the upper bound on T_i's loading factor is given by the maximum value between $\frac{e_i}{\min(d_i, p_i)}$ and $2\frac{e_i}{p_i}$. Finally, if the sum of all maximum loading factors for every T_i is not greater than one, $\mathbf{T}_n$ is going to be feasible and the lemma follows. □

The algorithm of Lemma 8 can be improved as shown in [MCF10a]. Now, if we use the DM/RM policy, Lemma 8 is still valid. For this reason and because we focus on DM/RM, we referred to the test of Lemma 8 as DM/RMTest1 in the remainder of this thesis.

For fixed-priority schedulings and arbitrary deadlines, Lehoczky showed in [Leh90] that the response time of the first job of a task T_l, scheduled synchronously with all higher-priority tasks, is not necessarily the maximum response time for T_l. In order to test T_l's feasibility in this case, the response time of all T_l's jobs within its first busy period must be calculated. T_l's first busy period is the time interval from the synchronous release together with all higher priority tasks to the time instant at which all previously released jobs finish executing.

Considering that tasks in $\mathbf{T}_n$ are sorted according to non-increasing priorities, so that if $i < j$ holds, T_i has priority higher than or equal to T_j. The busy period for T_l can be found in the following way [Leh90]:

$$t = \sum_{i=1}^{l} \left\lceil \frac{t}{p_i} \right\rceil \cdot e_i. \tag{2.31}$$

Equation (2.31) can be solved iteratively starting from $t^{(1)} = \sum_{i=1}^{l} e_i$ and until $t^{(k+1)} = t^{(k)}$ for some $k \geq 1$. This $t^{(k)}$ is the upper bound of T_l's busy period. The number of steps for solving Equation (2.31) is finite if the utilization is not greater than 100% [Leh90].

In [FB05a], Fisher and Baruah proposed an approximation of Lehoczky's exact test. This approximation has polynomial complexity and is based on the approximation techniques used by Albers and Slomka in [AS04] for the case of EDF.

Based on the Fisher and Baruah's work, it is possible to propose an $\mathcal{O}(n \log n)$ sufficient but not necessary feasibility test for fixed priorities and arbitrary deadlines. This test is the analogous to Devi's test for EDF and is presented by the following lemma.

LEMMA 9 *Let $\mathbf{T}'_n$ be a task set of n preemptive, asynchronous, periodic tasks, with arbitrary relative deadlines, arranged in order of non-increasing priorities. $\mathbf{T}'_n$ is schedulable using any fixed-priority scheduling algorithm if the following inequality holds for all l for which $1 \leq l \leq n$:*

$$\sum_{i=1}^{l} \frac{e_i}{p_i} + \frac{\sum_{i=1}^{l} e_i}{d_l} \leq 1. \tag{2.32}$$

Proof: It is assumed that tasks are sorted according to non-increasing priorities, so if $i < j$ holds, T_i's priority is higher than or equal to the priority of T_j. As $\mathbf{T}'_n$ is asynchronous and because we cannot find the worst-case release of tasks in polynomial time, we assume that a synchronous release happens at time $t = 0$. In this way, we just reduce the $\mathbf{T}'_n$ to a synchronous task set $\mathbf{T}_n$ for which the initial release times of tasks are ignored. Further, as already known, if $\mathbf{T}_n$ can be shown to be feasible, $\mathbf{T}'_n$ is also feasible.

In order to guarantee feasibility for arbitrary deadlines and fixed priorities, we know from Lehoczky's exact feasibility test [Leh90] that all jobs of a task T_l, where $1 \leq l \leq n$, must be verified within the task's first busy period starting from the synchronous release onwards. We can get the upper bound of a T_l's busy period by solving Equation (2.31) as stated in [Leh90]. Let us further denote by $\hat{t}_l$ the upper bound of T_l's busy period. By definition, the following equation obtained replacing t by $\hat{t}_l$ in Equation (2.31) must hold:

$$\hat{t}_l = \sum_{i=1}^{l} \left\lceil \frac{\hat{t}_l}{p_i} \right\rceil \cdot e_i.$$

However, instead of solving this equation in an exact manner, we just consider the following approximation and remove the ceiling function:

$$\hat{t}_l \leq \sum_{i=1}^{l} \left(\frac{\hat{t}_l}{p_i} + 1 \right) \cdot e_i.$$

Reshaping the previous inequality and considering that $U_l = \sum_{i=1}^{l} \frac{e_i}{p_i}$ is the utilization due to the first l higher-priority tasks including T_l, we get an estimation of $\hat{t}_l$:

$$\hat{t}_l \leq \frac{\sum_{i=1}^{l} e_i}{1 - U_l}. \tag{2.33}$$

Inequality (2.33) gives an approximated upper bound for T_l's busy period. If we now calculate the response times of all T_l's jobs up to the bound given by Inequality (2.33), we will have an exact feasibility test. The only difference to Lehoczky's test is that we might have to test more jobs of T_l because of overestimating $\hat{t}_l$.

Furthermore, because T_l's busy period $\hat{t}_l$ is the time instant at which all previously released jobs—belonging to the first l higher-priority tasks including T_l—finish executing, we can conclude that $\mathbf{T}_n$ and consequently $\mathbf{T}'_n$ are feasible if $\hat{t}_l \leq d_l$ holds for $1 \leq l \leq n$. This latter is not an exact feasibility test anymore, because it does not verify all T_l's jobs within its busy period. On the contrary, this test concludes that T_l is feasible only if all jobs in T_l's busy period, all T_l's jobs and all higher-priority jobs, finish executing before the deadline of the first T_l's job. Instead of calculating $\hat{t}_l$ in an exact way, we can use the approximation given by Inequality (2.33) as before. Finally, we have that the task set is feasible if the following holds for all $1 \leq l \leq n$:

$$\frac{\sum_{i=1}^{l} e_i}{1 - U_l} \leq d_l.$$

Recalling that $U_l = \sum_{i=1}^{l} \frac{e_i}{p_i}$ and rearranging terms, we obtain the following expression:

$$\sum_{i=1}^{l} \frac{e_i}{p_i} + \frac{\sum_{i=1}^{l} e_i}{d_l} \leq 1.$$

Hence, this lemma holds true. □

The test of Lemma 9 is a sufficient but not necessary test, which is the analogous for fixed priorities to Devi's test for EDF. Notice at this point that the complexity of this test is also $\mathcal{O}(n \log n)$, because tasks in $\mathbf{T}'_n$ must be sorted according to non-increasing priorities (e.g., non-decreasing relative deadlines for DM). For the case of the DM policy, the following lemma proves that the test of Lemma 9 remains valid even if the task set is not sorted according to non-decreasing relative deadlines.

LEMMA 10 *Given a task set $\mathbf{T}'_n$ of n preemptive, asynchronous, periodic tasks, with arbitrary relative deadlines. $\mathbf{T}'_n$ is feasible under the Deadline Monotonic algorithm if the following inequality holds for all l for which $1 \leq l \leq n$:*

$$\sum_{i=1}^{l} \frac{e_i}{p_i} + \frac{\sum_{i=1}^{l} e_i}{d_l} \leq 1. \tag{2.34}$$

Proof: Notice that this lemma assumes no particular order for the tasks in $\mathbf{T}'_n$. Let us first consider what happens for $l = 1$. If Inequality (2.34) holds, Lemma 9 guarantees that the task subset of only T_1 is feasible.

Suppose that Inequality (2.34) holds for $l = 2$. It is clear that the task subset of T_1 and T_2 is feasible if $d_1 \leq d_2$ because of Lemma 9. On the other hand, if $d_1 > d_2$ holds, we know

from Lemma 9 that $\hat{t}_2$, the first busy period for synchronous T_1 and T_2, is less than d_2. As a consequence, T_1 and T_2 are feasible under DM in this case too.

Considering a third task, if Inequality (2.34) holds, the first busy period for the synchronous T_1, T_2 and T_3 is less than or equal to d_3. That is $\hat{t}_3 \leq d_3$ holds. This implies further that T_3 is feasible independently of the value of d_3, because all jobs released previous to $t = d_3$ finish executing before d_3. Additionally, whatever value d_3 may have, the tasks T_1 and T_2 were proven feasible in the previous steps. For example, if d_3 were the shortest deadline (T_3 had the highest priority), the fact that $\hat{t}_3 \leq d_3$ holds guarantees the feasibility of T_1 and T_2, which have longer deadlines. Consequently, the task subset of T_1, T_2 and T_3 is also feasible regardless of the order of tasks.

Proceeding as previously, let us assume that Inequality (2.34) holds for $1 \leq l \leq n-1$, so T_l's first synchronous busy period $\hat{t}_l$ is less than or equal to d_l for $1 \leq l \leq n-1$. If Inequality (2.34) holds for $l = n$ too, $\hat{t}_n \leq d_n$ also holds. This further implies that T_n is feasible independently of the value of d_n, because all synchronous jobs released previous to $t = d_n$ finish executing before d_n. Additionally, whatever value d_n may have, the task subset of the first $n-1$ tasks was proven feasible in the previous steps. Furthermore, even if d_n were the shortest deadline in $\mathbf{T}'_n$ (T_n has the highest priority among all tasks), the fact that $\hat{t}_n \leq d_n$ holds guarantees the feasibility of the first $n-1$ tasks, which have all longer deadlines than d_n. As a result, $\mathbf{T}'_n$ is feasible independently of the order of tasks. Hence, the lemma follows. □

Lemma 10 allows us to reduce the complexity of Lemma 9's test from $\mathcal{O}(n \log n)$ to $\mathcal{O}(n)$ in case DM is used for assigning priorities. It is easy to see that Lemma 10 is also valid for the DM/RM policy.

In general, an asynchronous task set $\mathbf{T}'_n$ is always feasible if its corresponding synchronous task set $\mathbf{T}_n$ is feasible (where phases of $\mathbf{T}'_n$ are not considered) [SSRB98]. As a consequence, we focus on synchronous task sets in the remainder of this section.

The following lemma is about finding the maximum loading factor for a group of two tasks scheduled under the DM/RM algorithm discussed previously. That is, priorities are assigned according to $\min(d_i, p_i)$, so that the T_i with the shortest $\min(d_i, p_i)$ is the one with the highest priority and so forth. The next lemma is the analogous to Lemma 5 for EDF.

LEMMA 11 *Let $\mathbf{T}_{xy}$ be a subset of two arbitrary tasks T_x and T_y from $\mathbf{T}_n$ scheduled under the DM/RM algorithm, for which $t_y = \min(d_y, p_y)$, $t_x = \min(d_x, p_x)$ and $t_y \leq t_x$ hold. The loading factor $\rho_{xy}(t)$ of the subset $\mathbf{T}_{xy}$ is bounded above by $\hat{\rho}_{xy} = \max\left(\frac{e_y}{t_y}, 2\frac{e_y}{p_y}, \frac{e_{xy}}{t_x}, \frac{e_x+e_y}{t'_x} + \frac{e_x}{p_x} + \frac{e_y}{p_y}\right)$, where $e_{xy} = e_x + \lceil \frac{t_x}{p_y} \rceil \cdot e_y$ and t'_x is given by $\min\left(p_x, t_y + (\lfloor \frac{t_x - t_y}{p_y} \rfloor + 1) \cdot p_y\right)$.*

Proof: We know from Lemma 8 that an upper bound on T_y's loading factor is given by $\hat{\rho}_y = \max\left(\frac{e_y}{t_y}, 2\frac{e_y}{p_y}\right)$, where t_y is equal to $\min(d_y, p_y)$. Because $t_y \leq t_x$ holds, T_x cannot preempt T_y under the DM/RM algorithm. So, $\mathbf{T}_{xy}$'s loading factor is at maximum equal to $\hat{\rho}_y$ for $t < t_x$.

For greater time intervals, we use the definition of loading factor as the ratio $\rho_{xy}(t) = \frac{h_{xy}(t)}{t}$, i.e., the subset's execution demand in the interval $(0, t]$ over the interval length t:

$$\rho_{xy}(t) = \frac{\lceil \frac{t}{p_x} \rceil \cdot e_x + \lceil \frac{t}{p_y} \rceil \cdot e_y}{t}. \tag{2.35}$$

Calculating Equation (2.35) for $t_x = \min(d_x, p_x)$, we get the following if $d_x < p_x$ holds:

$$\rho_{xy}(t_x) = \frac{e_x + \lceil \frac{t_x}{p_y} \rceil \cdot e_y}{t_x}.$$

The numerator of the previous expression is what we denoted by e_{xy} in this lemma. Now, in order to find an upper bound on $\rho_{xy}(t)$, we would need to calculate Equation (2.35) for all possible time intervals. However, we can approximate $h_{xy}(t)$ in Equation (2.35) as follows:

$$\begin{aligned} \rho_{xy}(t) &\leq \frac{\left(\frac{t}{p_x} + 1\right) \cdot e_x + \left(\frac{t}{p_y} + 1\right) \cdot e_y}{t}, \\ &\leq \frac{e_x + e_y}{t} + \frac{e_x}{p_x} + \frac{e_y}{p_y}. \end{aligned} \tag{2.36}$$

From Inequality (2.36), we can conclude that our estimation of $\rho_{xy}(t)$ increases as the interval $(0, t]$ decreases. As we have already analyzed intervals up to t_x assuming $d_x < p_x$, we can choose the shortest interval greater than t_x without committing any error. This shortest interval greater than t_x can be given by p_x in case $d_x \geq p_x$ holds or the next deadline of T_y after t_x—considering $t_y = \min(d_y, p_y)$ instead of d_y. The minimum of these two will have to be considered, i.e., $t'_x = \min\left(p_x, t_y + (\lfloor \frac{t_x - t_y}{p_y} \rfloor + 1) \cdot p_y\right)$.

Finally, the greatest of the four $\frac{e_y}{t_y}$, $2\frac{e_y}{p_y}$, $\frac{e_{xy}}{t_x}$, and $\frac{e_x + e_y}{t'_x} + \frac{e_x}{p_x} + \frac{e_y}{p_y}$ is going to determine the upper bound $\hat{\rho}_{xy}$ for the loading factor of $\mathbf{T}_{xy}$ under the DM/RM scheduling algorithm. □

The following two lemmas give upper bounds for the loading factor of several tasks scheduled under DM/RM. As in the case of EDF, calculating the maximum loading factor for a group of several tasks leads to more accurate feasibility tests. These next two lemmas are analogous for DM/RM to Lemma 6 and Lemma 7 for EDF.

LEMMA 12 *Let $\mathbf{T}_{l-1}$ be the subset of the first $l-1$ tasks from $\mathbf{T}_n$ scheduled under the DM/RM algorithm, for which we know the maximum loading factor $\hat{\rho}_{l-1}$. If task T_l, also from $\mathbf{T}_n$, is added to $\mathbf{T}_{l-1}$, the loading factor of the resulting subset $\mathbf{T}_l$ is going to be bounded above by $\hat{\rho}_{l-1} + \max\left(\frac{e_l}{\min(d_l, p_l)}, 2\frac{e_l}{p_l}\right)$.*

Proof: It follows immediately from Lemma 8 observing that an upper bound for T_l's loading factor is given by $\max\left(\frac{e_l}{\min(d_l, p_l)}, 2\frac{e_l}{p_l}\right)$. □

LEMMA 13 *Let $\mathbf{T}_{l-1}$ be the subset of the first $l-1$ tasks from $\mathbf{T}_n$ scheduled under the DM/RM algorithm, for which we know the maximum loading factor $\hat{\rho}_{l-1}^{sup}$ and that $\min(d_i, p_i) \geq t_{sup}$ holds for all $1 \leq i \leq l-1$. If task T_l, also from $\mathbf{T}_n$, is added to $\mathbf{T}_{l-1}$, where $t_{sup} \geq \min(d_l, p_l)$, the loading factor of the resulting subset is going to be bounded above by $\hat{\rho}_{l-1}^{sup} + \max\left(\frac{k \cdot e_l}{t_{sup}}, \frac{e_l}{t_{lk}} + \frac{e_l}{p_l}\right)$ for all $t \geq t_{sup}$, where $k = \lceil \frac{t_{sup}}{p_l} \rceil$ and $t_{lk} = k \cdot p_l$.*

Proof: The loading factor of $\mathbf{T}_{l-1}$ is at maximum $\hat{\rho}_{l-1}^{sup}$. That is, $\rho_{l-1}(t) = \frac{h_{l-1}(t)}{t} \leq \hat{\rho}_{l-1}^{sup}$ holds for all possible t, what yields $h_{l-1}(t) \leq t \cdot \hat{\rho}_{l-1}^{sup}$. As a result, we can approximate the loading factor of $\mathbf{T}_{l-1}$ plus T_l in the following way:

$$\rho_l(t) \leq \hat{\rho}_{l-1}^{sup} + \frac{\lceil \frac{t}{p_l} \rceil \cdot e_l}{t}. \tag{2.37}$$

As we are interested in finding the upper bound for $\rho_l(t)$ from t_{sup} onwards. The first interval for which we need to calculate Inequality (2.37) is $(0, t_{sup}]$. That is:

$$\rho_l(t_{sup}) = \hat{\rho}_{l-1}^{sup} + \frac{\lceil \frac{t_{sup}}{p_l} \rceil \cdot e_l}{t_{sup}},$$

where $\lceil \frac{t_{sup}}{p_l} \rceil$ was denoted by k in this lemma. Further, getting rid of the ceiling function in Inequality (2.37) and reordering, we reach:

$$\rho_l(t) \leq \hat{\rho}_{l-1}^{sup} + \frac{e_l}{t} + \frac{e_l}{p_l}. \tag{2.38}$$

From Inequality (2.38), we need to choose the shortest possible t so as to get the upper bound of $\rho_l(t)$. However, we do not need to calculate Inequality (2.38) until $t_{lk} = k \cdot p_l$ for $k = \lceil \frac{t_{sup}}{p_l} \rceil$. This is because the loading factor for jobs in (t_{sup}, t_{lk}) cannot exceed the one at t_{sup}. Thus, the upper limit for $\rho_l(t)$ and $t \geq t_{sup}$ is given by $\hat{\rho}_{l-1}^{sup} + \max\left(\frac{k \cdot e_l}{t_{sup}}, \frac{e_l}{t_{lk}} + \frac{e_l}{p_l}\right)$. □

As it can be seen, the order of tasks in $\mathbf{T}_n$ does neither matter for Lemma 12 nor of Lemma 13. The following section is concerned with applying these lemmas in order to design feasibility tests with complexity $\mathcal{O}(n)$ for the DM/RM scheduling algorithm.

Feasibility Tests for the DM/RM algorithm

In this section, we explain two better linear-time feasibility tests for the DM/RM algorithm, which are analogous to the ones presented for EDF. Both algorithms calculate the maximum loading factor $\hat{\rho}_n$ of the whole task set $\mathbf{T}_n$. If $\hat{\rho}_n \leq 1$ holds, $\mathbf{T}_n$ is feasible. Again, we assume that there is already a set of $l-1$ tasks running on processor P_f, where $1 \leq l \leq n$ holds, and that the feasibility of a new task T_l should be tested in constant time on P_f.

Figure 2.7 presents the flow chart of a further algorithm proposed, which will be called DM/RMTest2. In principle, DM/RMTest2 applies Lemma 11 to obtain the loading factor of two tasks together under the DM/RM policy. If one task was accepted, DM/RMTest2 maintains the task parameters until the next task arrives. This way, the loading factor of the previous and the current task can be calculated. Analogously to EDFTest1, DM/RMTest2 uses a boolean variable called *previous task* to indicate that the previous task parameters are available. When a new task T_l arrives, DM/RMTest2 verifies whether previous task parameters are available. Then, if *previous task* is true, DM/RMTest2 applies Lemma 11 to calculate the maximum loading factor $\hat{\rho}_{xy}$ of previous accepted and the current task under DM/RM. If the maximum loading factor of all other accepted tasks denoted by $\hat{\rho}_{sum}$ plus $\hat{\rho}_{xy}$ is not greater than one, the new task T_l is accepted, $\hat{\rho}_{sum}$ is set to $\hat{\rho}_{sum} + \hat{\rho}_{xy}$ and *previous task* is set to false. Otherwise, if *previous task* is false, this algorithm uses Lemma 8 to get the maximum loading factor $\max\left(\frac{e_l}{\min(d_l,p_l)}, 2\frac{e_l}{p_l}\right)$ of the current task T_l alone. If $\hat{\rho}_{sum} + \max\left(\frac{e_l}{\min(d_l,p_l)}, 2\frac{e_l}{p_l}\right) \leq 1$ holds, T_l is accepted. In this case, T_l's parameters are stored for later calculations, *previous task* is set to true, but $\hat{\rho}_{sum}$ is not updated until another task arrives.

The flow chart of our last algorithm for DM/RM is shown in Figure 2.8. This algorithm is called DM/RMTest3 and applies Lemma 12 and Lemma 13 to compute the maximum loading factor $\hat{\rho}_n$ of $\mathbf{T}_n$. Like EDFTest2, this algorithm divides the loading factor into three. In this way, $\hat{\rho}_{inf}$ is the upper bound on the loading factor between t_{inf} and t_{sup} for $t_{inf} \leq t_{sup}$, whereas $\hat{\rho}_{sup}$ is the maximum loading factor from t_{sup} onwards. The loading factor for intervals less than t_{inf} is zero, because $t_{inf} \leq \min(d_l, p_l)$ holds for $1 \leq l \leq n$. So, the maximum of the two $\hat{\rho}_{inf}$ and $\hat{\rho}_{sup}$ is the maximum loading factor of all tasks. Before we start discussing this algorithm, let us denote by $\tilde{\rho}_{sup}$, $\tilde{\rho}_{inf}$, $\tilde{t}_{sup}$ and $\tilde{t}_{inf}$ temporal variables for the previously mentioned parameters. In the same way as in EDFTest2, these temporal variables are used to store temporal calculations of $\hat{\rho}_{sup}$, $\hat{\rho}_{inf}$, t_{sup} and t_{inf} that are then discarded if the task cannot be accepted.

For an arriving T_l, if $\min(d_l, p_l) > t_{sup}$ holds, Lemma 12 is applied to obtain $\hat{\rho}_l$. In this case, $\hat{\rho}_{l-1}$ from Lemma 12 is set to the current value of $\hat{\rho}_{sup}$. Because $\min(d_l, p_l) > t_{sup}$ holds, T_l has only influence on $\hat{\rho}_{sup}$, but not on $\hat{\rho}_{inf}$. The maximum value of the two $\hat{\rho}_l$ and $\hat{\rho}_{sup}$ is then stored in $\tilde{\rho}_{sup}$.

On the other hand, if $\min(d_l, p_l) < t_{inf}$ holds, i.e., $\min(d_l, p_l)$ is the shortest so far, T_l affects both $\hat{\rho}_{sup}$ and $\hat{\rho}_{inf}$. Consequently, Lemma 13 can be applied to obtain $\hat{\rho}_{inf}$ because $\min(d_l, p_l) < t_{inf}$ holds and $\hat{\rho}_{inf}$ is the maximum loading factor between t_{inf} and t_{sup}. In this case, $\hat{\rho}_{inf}$ is temporarily stored in $\tilde{\rho}_{inf}$. The variable $\hat{\rho}_{l-1}^{sup}$ of Lemma 13 is here set to the current value of $\hat{\rho}_{inf}$. Further, $\hat{\rho}_{sup}$ can be calculated using Lemma 13 because $\min(d_l, p_l) < t_{inf} \leq t_{sup}$ holds and $\hat{\rho}_{sup}$ is the maximum loading factor from t_{sup} onwards. Now, $\hat{\rho}_{sup}$ is stored in the temporal variable $\tilde{\rho}_{sup}$. This time, $\hat{\rho}_{l-1}^{sup}$ of Lemma 13 is set to the current value of $\hat{\rho}_{sup}$. Further, $\tilde{t}_{inf}$ is set to $\min(d_l, p_l)$, the shortest at the moment, and $\tilde{\rho}_{inf}$ must be assigned the maximum between $\tilde{\rho}_{inf}$ and the maximum loading factor of the current task T_l: $\max\left(\frac{e_l}{\min(d_l,p_l)}, 2\frac{e_l}{p_l}\right)$.

When $\min(d_l, p_l)$ is between t_{inf} and t_{sup}, $\tilde{t}_{sup}$ is going to be set to $\min(d_l, p_l)$. As a result, only $\hat{\rho}_{sup}$ needs to be recalculated. For this purpose, Lemma 12 is applied to get $\hat{\rho}_l$ considering $\hat{\rho}_{l-1} = \hat{\rho}_{inf}$. Further, Lemma 13 is used in order to find $\hat{\rho}_{sup}$ considering for this $\hat{\rho}_{l-1}^{sup} = \hat{\rho}_{sup}$

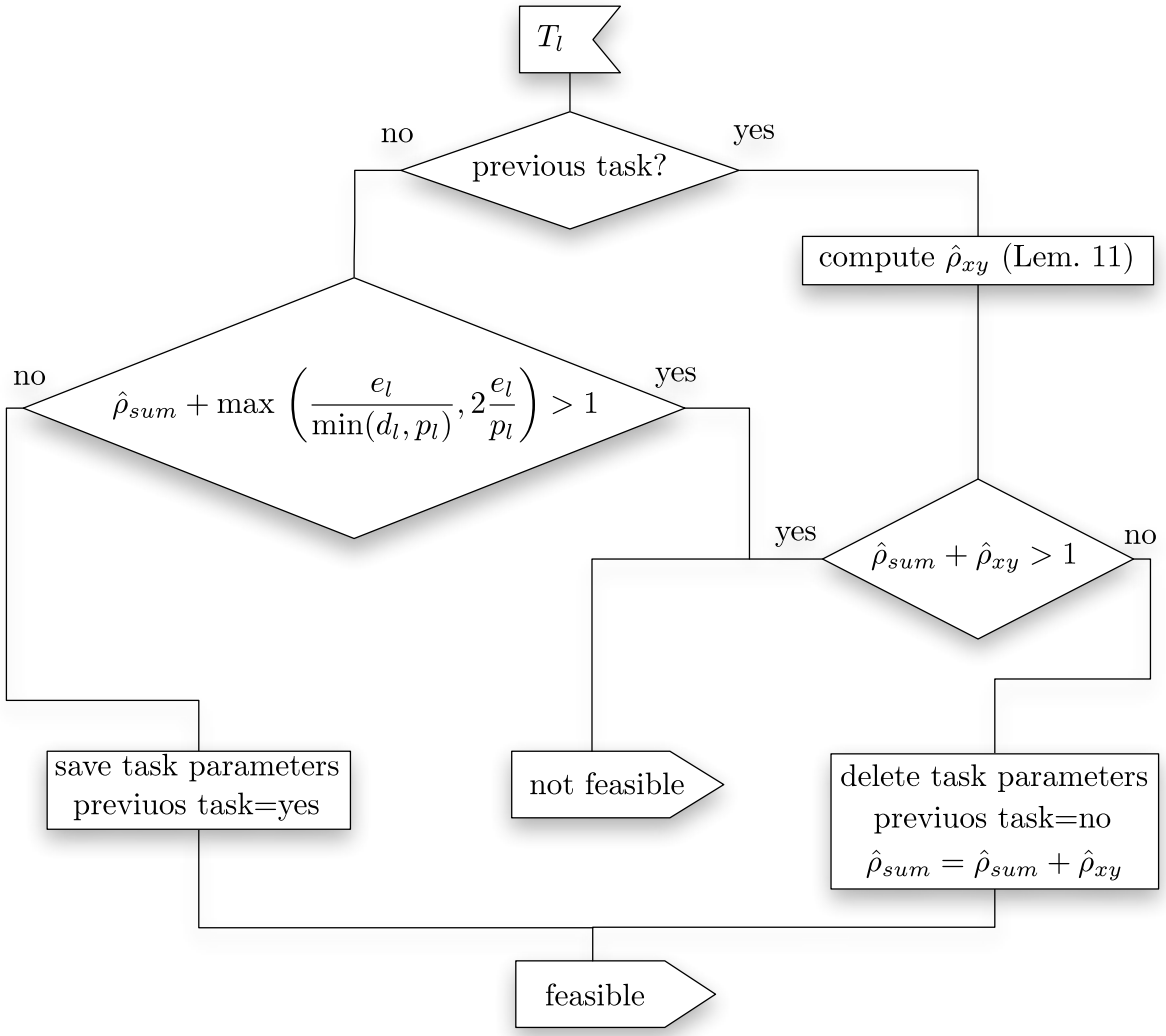


Figure 2.7: Algorithm DM/RMTest2

and storing it temporarily in $\tilde{\rho}_{sup}$. As it was mentioned before, $\tilde{t}_{sup}$ is set to $\min(d_l, p_l)$ and the temporal variable $\tilde{\rho}_{sup}$ is assigned the maximum between $\hat{\rho}_l$ and the value that is currently contained in $\tilde{\rho}_{sup}$.

Now, if the maximum loading factor of all other accepted tasks denoted by $\hat{\rho}_{sum}$ plus the maximum of the two $\tilde{\rho}_{sup}$ and $\tilde{\rho}_{inf}$ is not greater than 1, the new task T_l can be accepted in the system. Then, the values stored in temporal variables need to be adopted for later calculations.

The computation of $\hat{\rho}_{sup}$ and $\hat{\rho}_{inf}$ can be performed in the proposed way for all tasks in $\mathbf{T}_n$. For most cases, it results better to calculate them for smaller task groups of w tasks each. When $\hat{\rho}_{sup}$ and $\hat{\rho}_{inf}$ were computed for w tasks, the maximum between these two is accumulated in $\hat{\rho}_{sum}$. Then, all variables with exception of $\hat{\rho}_{sum}$ must be set to zero at this point in order to start calculating $\hat{\rho}_{sup}$ and $\hat{\rho}_{inf}$ for the next group of w tasks. The variable w can assume any integer value between 1 and n.

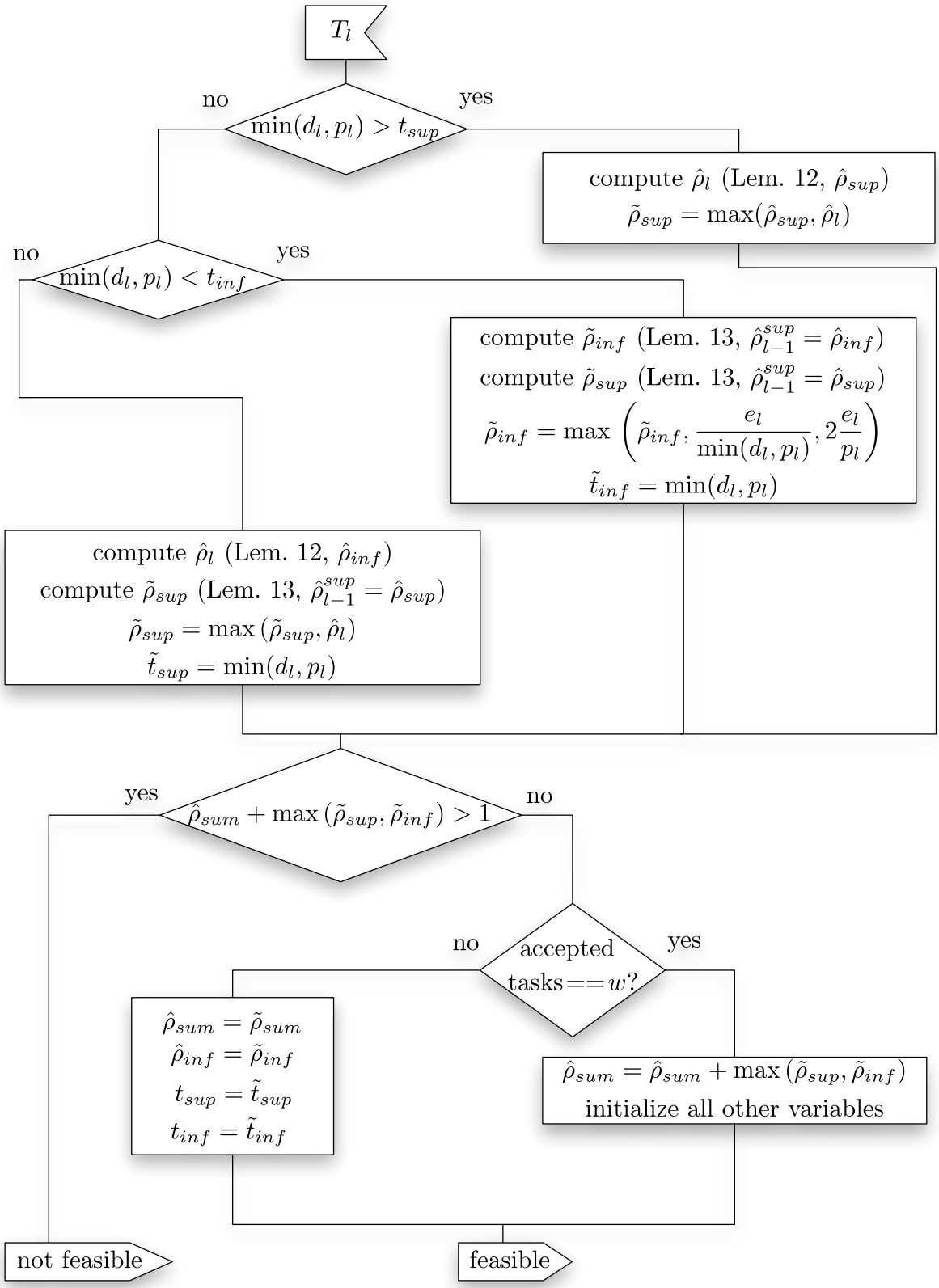


Figure 2.8: Algorithm DM/RMTest3

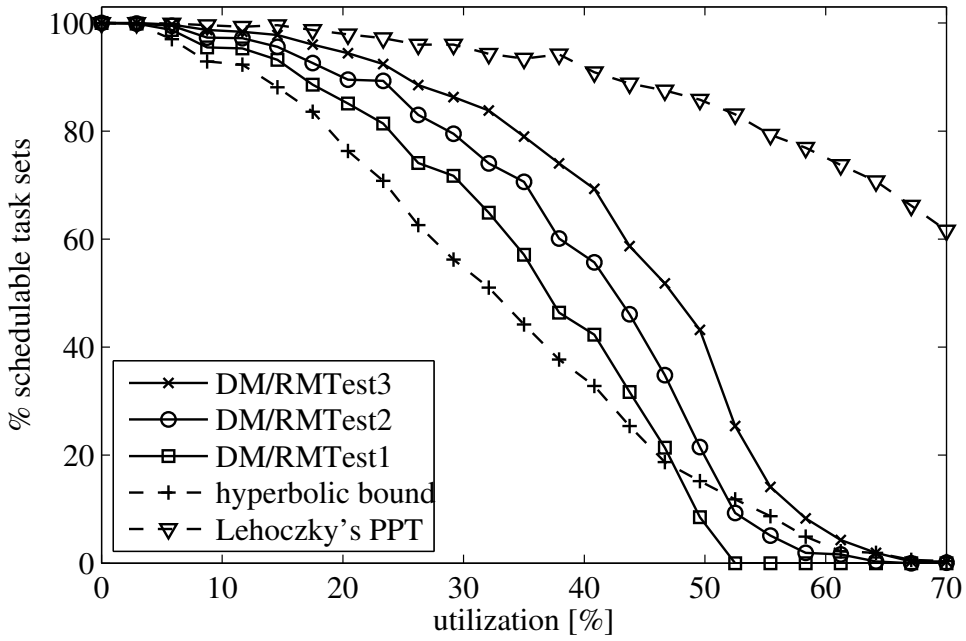


Figure 2.9: Schedulability vs. utilization for 5 tasks

Some Experimental Results

In this section, we evaluate some experiments comparing the proposed algorithms against the best known feasibility tests of the same complexity $\mathcal{O}(n)$. Hence, we compare the three new algorithms DM/RMTest1 from Lemma 8, DM/RMTest2 and DM/RMTest3 with the hyperbolic bound of Inequality (2.27). For DM/RMTest3, the size of the task groups was chosen to be $w = 5$, i.e., the maximum loading factor is calculated for groups of 5 tasks each. Additionally, we include Lehoczky's pseudo-polynomial-time exact feasibility test from [Leh90] denoted by Lehoczky's PPT in this comparison.

The mentioned algorithms are contrasted with respect to their accuracy versus utilization for different numbers of tasks per task set. As in the case of EDF, random task sets were uniformly generated for different processor utilizations as recommended in [BB04a] and [BB05]—*UUniFast* was used to generate a set of task utilization u_i. Further, once generated a set of u_i this way, we created periods p_i also randomly with uniform distribution. Consequently, we got $e_i = u_i \cdot p_i$. The relative deadlines d_i were uniformly chosen from the range $[e_i, p_i]$. Additionally, the utilization axis was uniformly sampled at 24 points, for which 1000 different task sets were generated each time.

Figures 2.9 to 2.11 present schedulability curves for 5, 10 and 100 tasks per task set. For 5 tasks per task set, the proposed algorithms reach around 20% to 40% more accepted task sets than the hyperbolic bound in the utilization interval $(20\%, 40\%)$. For a utilization of approximately 45% and 5 tasks, the hyperbolic bound test has a better performance than the DM/RMTest1 and than DM/RMTest2. As the number of tasks grows, the performance of all polynomial-time

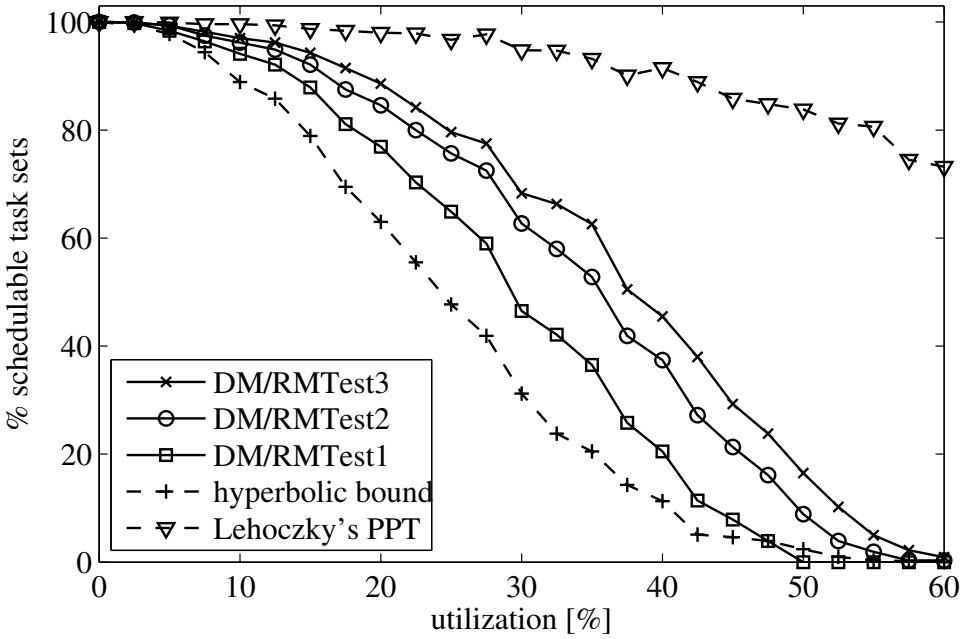


Figure 2.10: Schedulability vs. utilization for 10 tasks

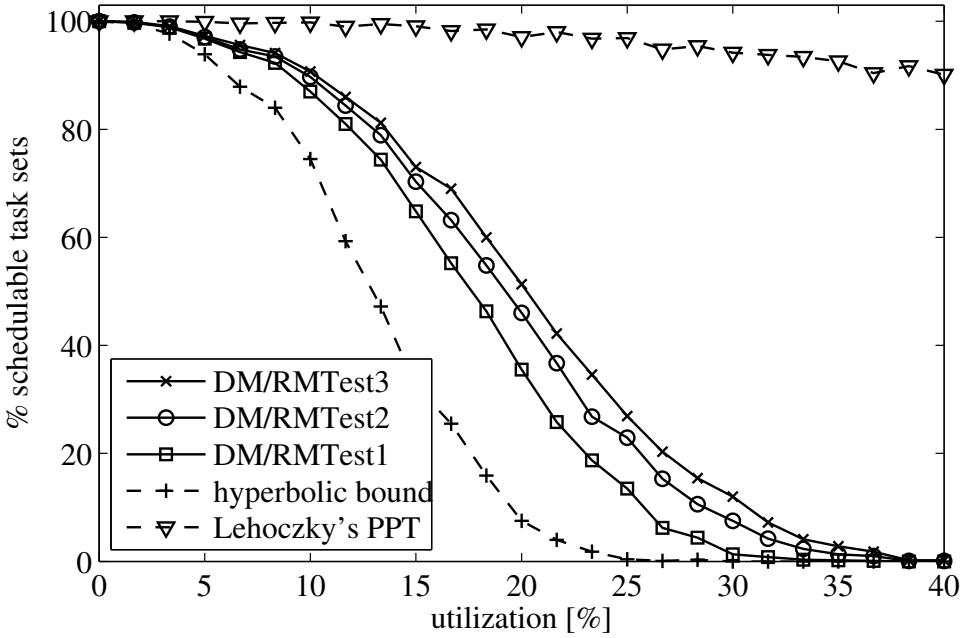


Figure 2.11: Schedulability vs. utilization for 100 tasks

algorithms, the proposed and the known ones, decays rapidly. For 10 tasks the polynomial-time algorithms detect feasible task sets up to 60% utilization, whereas they are unable to find schedulable task sets for a utilization over 40% and 100 tasks per task set. For an increasing number of tasks, the performance improvement of the three proposed algorithms over the hyperbolic bound gets better: around 50% more accepted tasks sets for 100 tasks and 20% utilization. On the other hand, the proposed tests perform similarly to one another also as the number of tasks increases.

2.2.3 Context Switches

It is clear that tasks' priority and consequently the one of jobs is fixed for DM and RM. On the other hand, the priority of tasks may change dynamically under EDF, i.e., jobs of the same task may assume different priorities along the schedule. However, once a job of any task is released under EDF, it gets a priority that will not change as long as the job exists. So, EDF, DM and RM are called job-level fixed-priority scheduling algorithms.

For job-level fixed priorities, in oder to take the effect of context switches into account, we apply the method proposed in [Liu00]. This method consists in adding σ_f to the worst-case execution time e_i of jobs, where σ_f is the overhead due to a context switch on the processor P_f (i.e., the sum of the time for storing and restoring a job on P_f). In this way, every time a job is preempted by another one, the overhead to store and restore a job will be considered. Our previous analysis on EDF and on fixed priorities remains unchanged.

2.3 Considering Soft Real-Time

For some applications, for which the system functioning does not degrade drastically if a deadline is missed, it is possible to allow a controlled deadline violation. Of coarse, these applications cannot be considered to be hard real-time, but they are quite common in today's world, e.g., multimedia applications, etc. If a deadline is missed when processing video streams, there will be some quality loss, but the system can continue running. As long as this deadline violation does not make the quality loss to be intolerable, nobody will notice the difference.

Allowing a controlled deadline violation can relax the system requirements and reduce costs, so that it is worth to research on techniques for making this possible. Chakraborty et al. introduced in [CKT02] the concept of optimistic feasibility test. An optimistic feasibility test is such that considers a task set to be feasible if the deadline violations do not exceed a given configurable upper limit. In this section, we use the concept of optimistic feasibility test to propose polynomial-time tests for EDF and fixed priorities. The difference to the tests of [CKT02] is that we sacrifice accuracy for the sake of faster running times.

The following lemma is about finding a time upper bound under EDF after which an overflow cannot be greater than a given limit ϵ. This follows similar principles to the ones presented in [MF06a, Mas07].

LEMMA 14 *If a time overflow occurs at t_{miss} when scheduling a synchronous task set $\mathbf{T}_n$ under EDF, $\epsilon < h_n(t_{miss}) - t_{miss}$ holds for a given ϵ, if and only if $t_{miss} \leq I_{n,\epsilon}$ also holds, where $I_{n,\epsilon} = \frac{\sum_{i=1}^{n}(p_i - \min(d_i,p_i))\cdot u_i - \epsilon}{1-U_n}$, $h_n(t) = \sum_{i=1}^{n} \max\left(0, \left\lfloor \frac{t-d_i}{p_i} \right\rfloor + 1\right) \cdot e_i$, $u_i = \frac{e_i}{p_i}$ and $U_n = \sum_{i=1}^{n} \frac{e_i}{p_i}$ is the total utilization.*

Proof: Because a deadline is missed at t_{miss}, the total execution demand of $\mathbf{T}_n$ at t_{miss} is greater than the available time t_{miss}, i.e., $t_{miss} < h_n(t_{miss})$ where $h_n(t)$ is the demand bound function under EDF [BMR90]. Now, if the time overflow is greater than a given ϵ, we will have $\epsilon < h_n(t_{miss}) - t_{miss}$:

$$\epsilon < \sum_{i=1}^{n} max\left(0, \left\lfloor \frac{t_{miss} - d_i}{p_i} \right\rfloor + 1\right) \cdot e_i - t_{miss}.$$

Considering $\min(p_i, d_i)$ instead of d_i, we obtain:

$$\epsilon < \sum_{i=1}^{n} \left(\left\lfloor \frac{t_{miss} - \min(p_i, d_i)}{p_i} \right\rfloor + 1\right) \cdot e_i - t_{miss}.$$

Further, removing the floor function, we reach:

$$\epsilon < \sum_{i=1}^{n} \left(\frac{t_{miss} - \min(p_i, d_i)}{p_i} + 1\right) \cdot e_i - t_{miss}.$$

Reshaping this inequality to obtain t_{miss}, we get:

$$t_{miss} < \frac{\sum_{i=1}^{n} (p_i - \min(d_i, p_i)) \cdot u_i - \epsilon}{1 - U_n}, \tag{2.39}$$

where $u_i = \frac{e_i}{p_i}$ and U_n is the total utilization. The right member of Inequality (2.39) was previously denoted by $I_{n,\epsilon}$. The lemma follows. □

Based on Lemma 3 and on Lemma 14, it is possible to come up with the following $\mathcal{O}(n \log n)$ feasibility test for soft real-time tasks where the time overflow is bounded to ϵ. The proof is straight forward and requires no further analysis.

LEMMA 15 *Let $\mathbf{T}_n$ be a task set of n preemptive, synchronous, periodic tasks, with arbitrary relative deadlines, arranged in order of non-decreasing relative deadlines. In $\mathbf{T}_n$'s schedule under EDF, there is no time overflow greater than a given ϵ, if the following inequality holds for all l for which $1 \leq l \leq n$:*

$$\sum_{i=1}^{l} \frac{e_i}{p_i} + \frac{1}{d_l} \sum_{i=1}^{l} \left(\frac{p_i - \min(p_i, d_i)}{p_i} \right) \cdot e_i - \frac{\epsilon}{d_l} \leq 1. \tag{2.40}$$

The following lemma is the analogous to Lemma 14 for fixed priorities. That is, a time upper bound is given after which an overflow under fixed priorities cannot exceed a given limit ϵ.

LEMMA 16 *If a time overflow occurs at t_{miss} when scheduling a synchronous task set $\mathbf{T}_n$ under fixed priorities, $\epsilon < h_n(t_{miss}) - t_{miss}$ holds for a given ϵ, if and only if $t_{miss} \leq I_{n,\epsilon}$ also holds, where $I_{n,\epsilon} = \frac{\sum_{i=1}^{n} e_i - \epsilon}{1-U_n}$, $h_n(t) = \sum_{i=1}^{n} \left\lceil \frac{t}{p_i} \right\rceil \cdot e_i$, $u_i = \frac{e_i}{p_i}$ and $U_n = \sum_{i=1}^{n} \frac{e_i}{p_i}$ is the total utilization.*

Proof: Because a deadline is missed at t_{miss}, the total execution demand of $\mathbf{T}_n$ at t_{miss} is greater than the available time t_{miss}, i.e., $t_{miss} < h_n(t_{miss})$ where $h_n(t)$ is the demand bound function under fixed priorities [Leh90]. If the time overflow is greater than a given ϵ, we will have $\epsilon < h_n(t_{miss}) - t_{miss}$:

$$\epsilon < \sum_{i=1}^{n} \left\lceil \frac{t_{miss}}{p_i} \right\rceil \cdot e_i - t_{miss}.$$

Now, removing the ceiling function, we obtain:

$$\epsilon < \sum_{i=1}^{n} \left(\frac{t_{miss}}{p_i} + 1 \right) \cdot e_i - t_{miss}.$$

Reshaping this inequality to obtain t_{miss}, we get:

$$t_{miss} < \frac{\sum_{i=1}^{n} e_i - \epsilon}{1 - U_n}, \tag{2.41}$$

where U_n is the total utilization. The right member of Inequality (2.41) was previously denoted by $I_{n,\epsilon}$. The lemma follows. □

Finally, as in the case of EDF, we can apply Lemma 9 and Lemma 16 to derive an $\mathcal{O}(n \log n)$ feasibility test for soft real-time tasks scheduled under fixed priorities. As for the test of Lemma 15, the time overflow is bounded to ϵ. The proof of Lemma 17 is also straight forward and needs no further attention.

LEMMA 17 *Let $\mathbf{T}_n$ be a task set of n preemptive, synchronous, periodic tasks, with arbitrary relative deadlines, arranged in order of non-increasing priorities. In the $\mathbf{T}_n$'s schedule under fixed priorities, there is no time overflow greater than a given ϵ, if the following inequality holds for all l for which $1 \leq l \leq n$:*

$$\sum_{i=1}^{l} \frac{e_i}{p_i} + \frac{\sum_{i=1}^{l} e_i - \epsilon}{d_l} \leq 1. \tag{2.42}$$

2.4 Key Findings

Based on the concept of maximum loading factor, we have derived linear-time feasibility tests for the case of arbitrary deadlines. These tests are more accurate than known tests with the same complexity (e.g., the density condition in case of EDF and the hyperbolic bound in case of the DM/RM policy). The most accurate linear-time feasibility test for EDF resulted to be EDFTest2, whereas DM/RMTest3 (the analogous of EDFTest2) is the best linear-time feasibility test proposed for the DM/RM policy. Both of these tests compute the maximum loading factor for groups of several tasks. The number of tasks in these groups can be configured by a parameter denoted w.

3 Allocating Independent Real-Time Tasks to Processors

When allocating real-time tasks, we are normally interested in finding the minimum possible number of processors that guarantees feasibility. That is, we are concerned with the partitioning of tasks onto processors, so that they all can meet their deadlines while the number of processors is reduced to the minimum possible. Even in its simplest form, the task allocation problem is an extremely complex one. It still remains very complex when tasks are independent of one another, i.e., even if there are no interactions among tasks: no precedence order and no communication.

In the case of independent tasks and considering the most favorable scheduling conditions (deadlines equal to periods under EDF), the task allocation reduces to the bin packing problem [GJ79]. The bin packing problem has been proven to be NP-hard in the strong sense [GJ79], i.e., it is an intractable problem. Intuitively, this means that an algorithm with exponential complexity will be necessary to find an optimal solution to this problem. In other words, even for the simplest form of the allocation problem, the running time of an optimal algorithm grows exponentially with the number of tasks to be allocated.

On the other hand, heuristic approximation methods, e.g., simulated annealing and genetic algorithms, have been proposed to solve the allocation problem [Thi00]. These methods aim at achieving a good but not optimal solution while they present less computational complexity. A deeper analysis of these methods goes, however, beyond the scope of this thesis—for more information see [ACG+03].

The approach chosen for this thesis is based on sequential approximation algorithms [ACG+03], which are also of heuristic nature. However, in contrast to other heuristic methods, sequential algorithms do not attempt to optimize the allocation, but they just build task partitions sequentially a task after the other. The greatest advantage of sequential algorithms over all other approaches is their simplicity. We can further differentiate between on-line and off-line sequential algorithms. On-line sequential algorithms perform no initial sorting of tasks, so they do not require all tasks to be known at the beginning. On the other hand, off-line sequential algorithms do perform an initial sorting of tasks according to a given criterion and require consequently to know all tasks before running.

In this chapter, we focus on sequential algorithms for allocating independent real-time tasks to processors. These algorithms will be extended to consider task communication and system constraints in the following chapter. We first discuss known sequential algorithms for bin packing and compare their efficiency, in terms of reducing the number of processors, by means of an

extensive statistical comparison. As stated above, the bin packing problem is the simplest form of task allocation where tasks are scheduled under EDF and deadlines are equal to periods. In order to consider more general scheduling conditions, we apply the feasibility tests from Chapter 2 to extend known sequential algorithms for bin packing. The resulting algorithms are able to deal with the allocation of real-time tasks with arbitrary deadlines under both EDF and the DM/RM scheduling policy.

3.1 Bin Packing and Task Allocation

The bin packing problem consists in partitioning a given finite set of items, which have rational sizes in $[0, 1]$, into disjoint subsets, such that the sum of the sizes of the items in each subset is no more than 1 and such that the number of subsets is as small as possible [GJ79].

Clearly, the task allocation problem can directly be expressed as a bin packing problem if the processor utilization can be as high as 1 (i.e., 100%) without missing deadlines. However, this is only possible under an optimal scheduling algorithm like EDF when tasks are periodic, independent and fully preemptive and their deadlines are equal to periods [LL73]. (A 100% utlization is also possible for harmonic periods under RM, however, we consider periods to be arbitrary in this thesis.) Further, tasks should be scheduled on identical processors [GJ79], so that their utilizations do not change from processor to processor. If all these conditions hold, the tasks T_i can be considered to be the items to partition in a bin packing problem, where task utilizations $u_i = \frac{e_i}{p_i}$ are the corresponding item sizes and processors are the bins where items should be placed.

As previously mentioned, sequential algorithms are applied to solve the task allocation problem. In this section, our aim is to compare the performance of these algorithms in the context of the bin packing problem before we start analyzing more complex allocation scenarios. For this purpose, an extensive statistical comparison is presented in contrast to the known worst-case performance metrics [CGJ97], which are most of the time too pessimistic.

3.1.1 Sequential Algorithms for Bin Packing

For the sake of the comparison, we analyzed four of the most common sequential algorithms for bin packing: Next Fit (NF), First Fit (FF), Best Fit (BF), First Fit Decreasing (FFD) and Best Fit Decreasing (BFD). NF, FF and BF are on-line algorithms, i.e., they do not require to know all tasks at the beginning. On the other hand, FFD and BFD perform an initial sorting of tasks and are consequently off-line algorithms [CGJ97].

Before describing the different algorithms, let us define by *open processor* a processor to which tasks can still be allocated. In the same way, a *closed processor* is a processor to which no additional task can be allocated. Normally, algorithms have different strategies to change the state of a processor from open to closed (to close a processor) and to add a new processor (to open a processor).

- Next Fit (NF) has only one open processor at a time and allocates tasks to it as long as the processor utilization is less than or equal to 1. The processor is closed whenever its utilization reaches 1 or a task does not fit into it (i.e., the available processor utilization does not suffice for the new task). Once the processor has been closed, it cannot be accessed anymore and a new empty one must be opened.

- First Fit (FF) can have more than one open processor at a time. It tries to allocate tasks to open processors in the order in which they were opened. A task can be allocated to a given processor if the processor utilization including the new task is less than or equal to 1. If a task cannot be allocated to any open processor, a new processor is opened for this task. Processors are only closed if their corresponding utilization is equal to 1, i.e., if they have run out of capacity.

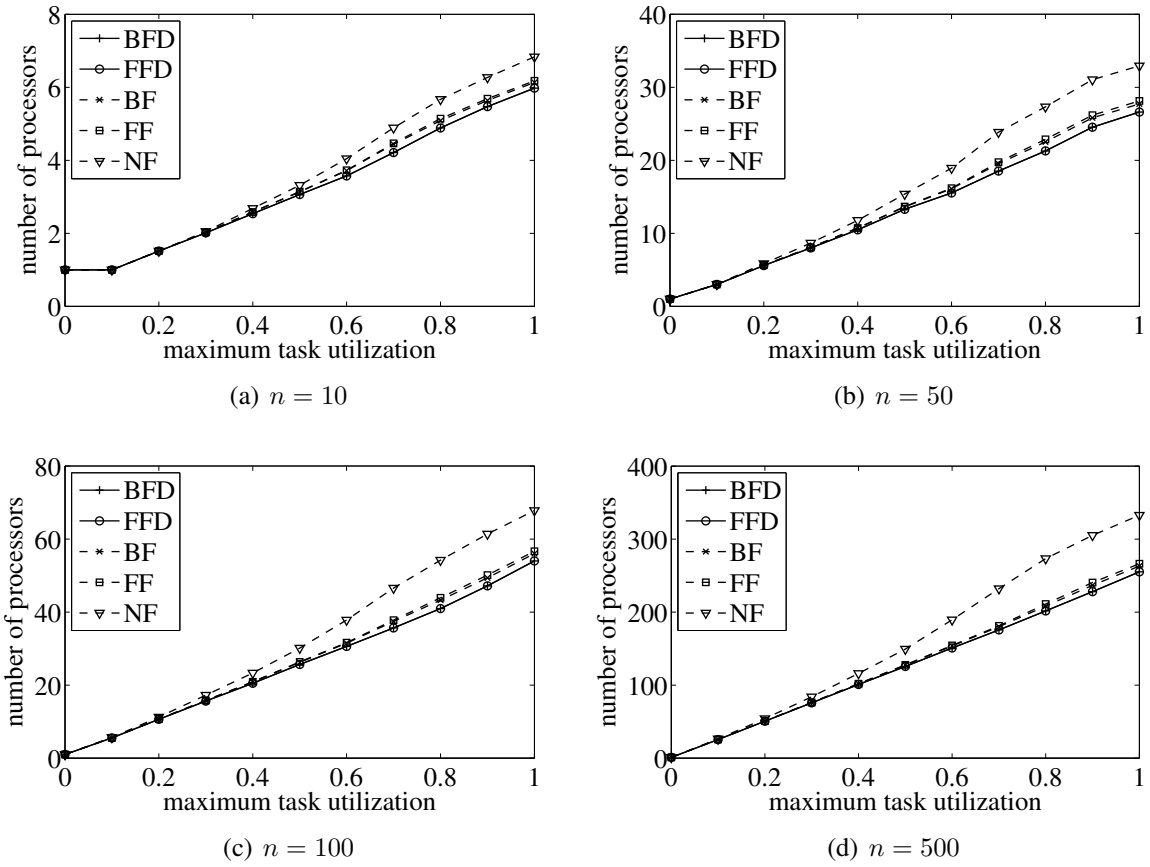
- Best Fit (BF) can also have more than one open processor at a time. However, it tries to allocate tasks to open processors in increasing (non-decreasing) order of available processor utilization. That is, BF first tries to allocate a task to the processor with the least available utilization. If this is not possible (the processor utilization with the new task is greater than 1), it tries to allocate the task to the processor with the second least available utilization and so forth. If a task cannot be allocated to any open processor, a new processor is opened for this task. Like FF, BF closes a processor only if its total utilization is equal to 1 (when there is no remaining capacity on the processor).

- First Fit Decreasing (FFD) sorts tasks according to decreasing (non-increasing) task utilization u_i and then proceeds as described for FF (i.e., it allocates tasks to processors in the order in which processors were opened).

- Best Fit Decreasing (BFD) also sorts tasks according to decreasing (non-increasing) task utilization u_i and then proceeds as described for BF (i.e., it assigns tasks to processors in order of increasing available processor utilization).

The complexity of NF is clearly $\mathcal{O}(n)$, while all other algorithms can be implemented with complexity $\mathcal{O}(n \log n)$ [CGJ97]. In case of BF and BFD, for example, when a new processor is opened, it can be added to a sorted list according to its available utilization. This can be performed in $\mathcal{O}(\log n)$, because the number of open processors, and consequently the number of entries in the sorted list, can be as large as the number of tasks n. Every time a task must be allocated, the sorted list of open processors can be sought for the appropriate one also in $\mathcal{O}(\log n)$. Further, whenever a task is allocated to an open processor, the resorting of the processor in the sorted list can also be performed in $\mathcal{O}(\log n)$. As a consequence, the whole BF algorithm presents a complexity $\mathcal{O}(n \log n)$. On the other hand, FF and FFD require using an appropriate data structure [Joh73].

All described algorithms process tasks sequentially until all of them are allocated, so they always provide a solution for the bin packing problem. Let us analyze how good this solution is for most cases, i.e., by means of a statistical comparison.



(a) $n = 10$ (b) $n = 50$

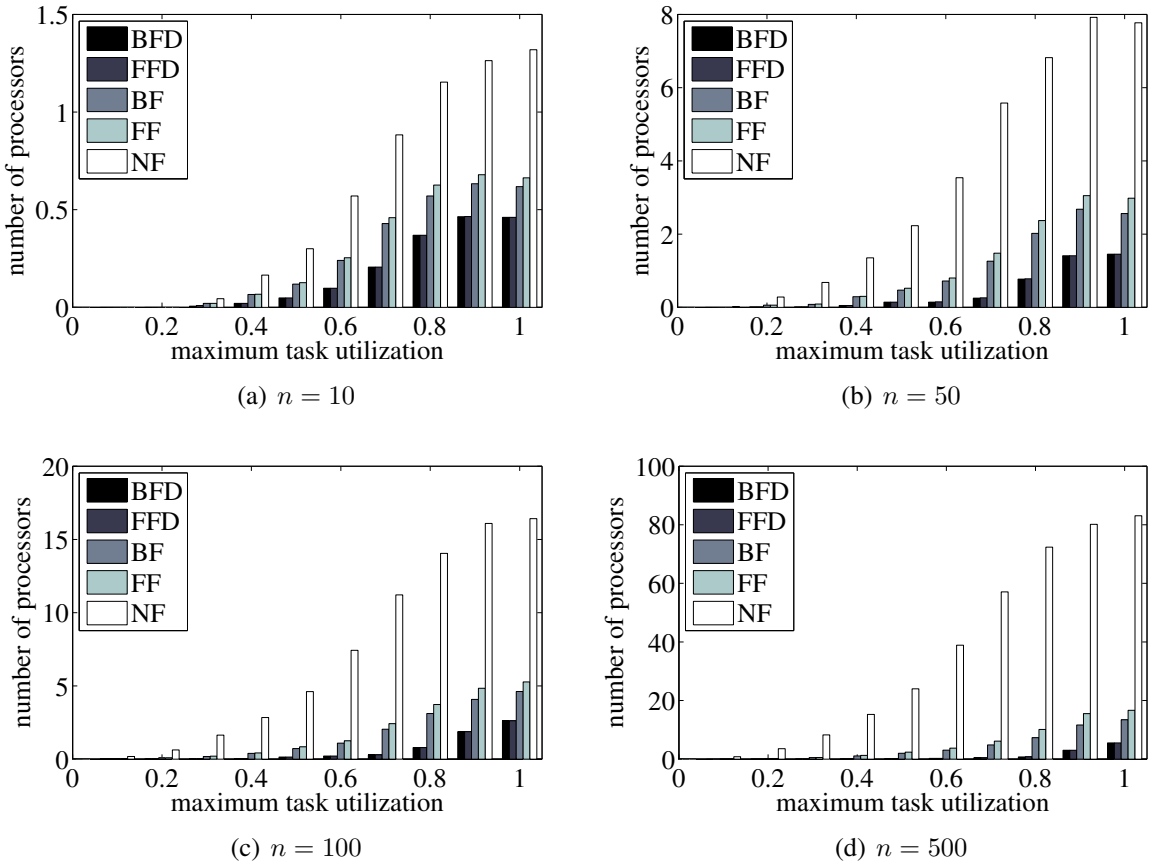
(c) $n = 100$ (d) $n = 500$

Figure 3.1: Bin packing: average number of processors vs. maximum task utilization

3.1.2 Statistical Performance Comparison

An extensive statistical comparison is provided for all previously described bin packing algorithms. Further, let us consider a *pseudo-optimal* bin packing algorithm (we refer to as BestBP) that returns the smallest possible number of processors at all. BestBP calculates the number of processors as the ceiling of the sum of all task utilizations: $\lceil \sum_{i=1}^{n} u_i \rceil$. Clearly, BestBP returns a total of processors that is always less than or equal to the one resulting from an optimal bin packing. However, we chose to use BestBP for the statistical comparison, because an optimal bin packing algorithm presents exponential complexity and, consequently, a huge running time for large numbers of tasks.

Figure 3.1 shows how algorithms behave, for which task sets were randomly generated for different numbers of tasks n. The maximum task utilization represented on the x-axis is the upper bound of the uniform distribution, i.e., task utilizations within task sets were uniformly generated between 0 and a varying maximum task utilization. Further, we increased the maximum task utilization from 0 to 1 in 10 steps of size 0.1 each. Every time the maximum task utilization was increased, 1000 different task sets of n tasks each were created. Curves in Figure 3.1 were plotted for 10, 50, 100 and 500 tasks. NF is as expected the on-line algorithm with the



(a) $n = 10$ (b) $n = 50$

(c) $n = 100$ (d) $n = 500$

Figure 3.2: Bin packing: average number of additional processors with respect to BestBP vs. maximum task utilization

worst performance, whereas FF and BF show similar performance. On the other hand, FFD and BFD behave exactly the same and are more efficient at reducing the number of processors than on-line algorithms. In other words, allocating tasks from the biggest (the one with the highest utilization) to the smallest leads to a more significant reduction of the number of processors.

In addition, Figure 3.2 shows the algorithms' performance compared to BestBP in a more illustrative manner. The length of bars represents the average distance to BestBP, which provides an estimation of how many additional processors are going be necessary because of using a non-optimal bin packing algorithm. In general, the average number of additional processors falls as the maximum task utilization within the task set decreases. Further, the number of additional processors decreases with an increasing number of tasks. The following table summarizes this for FFD and BFD, which behave almost identically and perform better than the other algorithms we discussed.

NF is the algorithm with the least complexity, however, it results in a rather pessimistic task allocation. On the other hand, FF and BF perform much better than BF. From a statistical point of view, both FF and BF produce quite similar task allocations, although it is possible to find

maximum task utilization	$n = 10$	$n = 50$	$n = 100$	$n = 500$
1	8.87%	6.14%	4.67%	2.32%
0.6	2.76%	1.04%	0.68%	0.02%

Table 3.1: Bin packing: percent average number of additional processors with respect to BestBP for FFD and BFD

cases for which FF is better than BF and vice versa. In the same way, FFD presents almost the same statistical behavior as BFD. As a consequence, in the remainder of this chapter, we focus on solutions based on FF and FFD to perform a task allocation.

3.1.3 Bin Packing for RM

Before continuing, let us briefly consider what happens when RM is used instead of EDF for tasks with deadlines equal to periods. In this case, it will generally not be possible to have processor utilizations as high as 100%. This is only possible under RM if all periods are harmonic [Liu00]. So, the bin packing algorithms must be adapted to utilize processors up to a given bound less than 1.

In general, the described bin packing algorithms can be extended to use Liu and Layland's utilization bound of Equation (2.25). Following this idea, an algorithm called Rate Monotonic First Fit (RMFF) was proposed by Dhall and Liu in [DL78]. Using a similar feasibility test to the hyperbolic bound from Equation (2.26), Oh and Son presented other two algorithms based on FF [OS95]. Further, Burchard et al. prosed two algorithms that exploit the fact that a higher processor utilization is possible when periods are closer to being harmonic [BLOS95].

3.2 Task Allocation for Arbitrary Deadlines

In this section, we extend the described bin packing algorithms to the case of arbitrary deadlines. For this purpose, the feasibility tests from Chapter 2 are applied. Because EDF and the DM/RM scheduling require different feasibility tests, we have to analyze them separately.

For both mentioned scheduling policies, possible allocation algorithms are presented and compared statistically for a large number of synthetic task sets. All curves presented in the following sections were generated as before, i.e., using a uniform distribution to generate n tasks utilizations between 0 and a given maximum task utilization. The maximum task utilization (x-axis) was increased in steps of length 0.1 until reaching 1, for which 1000 different sets of utilizations were created each time. Once a set of utilizations u_i was obtained, we generated periods p_i also with a uniform distribution in the interval $(0, 1]$. Further, the worst-case execution times were obtained ($e_i = u_i \cdot p_i$) and deadlines were uniformly generated in the interval $[e_i, p_i]$.

3.2.1 Algorithms for EDF

Clearly, it is possible to use the density test together with FFD or BFD to perform a task allocation when deadlines are not restricted to be equal to periods. On the other hand, it is also possible to combine Devi's test with FF or BF [BF05, BF06b, MF06b, BF07], which results in a better allocation. Further, we can use EDFTest1 and EDFTest2 to design allocation algorithms in combination with FFD or BFD.

Let us now compare how these possible algorithms perform with respect to each other and to BestBP. The following four algorithms are considered in this section: DensityFFD, DeviFF, EDFTest1FFD and EDFTest2FFD.

- DensityFFD results from combining the density test and the FFD heuristic, where tasks are sorted according to decreasing (non-increasing) density $\frac{e_i}{\min(d_i,p_i)}$ to consider arbitrary deadlines.
- DeviFF combines Devi's test [Dev03] and the FF heuristic as proposed in [MF06b]. Although Devi's test remains valid when tasks are not sorted—see Lemma 4, it performs better if tasks are arranged according to non-decreasing (increasing) deadlines. In order to exploit this, DeviFF sorts tasks according to increasing deadlines and uses FF to perform an allocation.
- EDFTest1FFD uses EDFTest1 of Figure 2.2 together with the FFD heuristic, where tasks are sorted in order of decreasing density. In contrast to the density test, EDFTest1 computes the maximum loading factor $\hat{\rho}_{xy}$ (see Lemma 5 in Chapter 2) for two tasks together.
- EDFTest2FFD results from applying EDFTest2 of Figure 2.3 and the FFD heuristic, where tasks are also sorted in order of decreasing density. EDFTest2 computes the maximum loading factor for groups of several tasks using Lemma 6 and Lemma 7 from Chapter 2. The maximum loading factors $\hat{\rho}_{sup}$ and $\hat{\rho}_{inf}$ are calculated for groups of $w = 10$ tasks in Figure 2.3.

Notice that all these algorithms have the same polynomial complexity as FFD and BFD, i.e., $\mathcal{O}(n \log n)$. So, applying the new feasibility tests does not degrade the complexity of the original heuristics and results consequently in very fast allocation algorithms. Because of needing tasks to be sorted according to deadlines, Devi's test can only be used for an off-line allocation. On the other hand, the density test, EDFTest1 and EDFTest2 can also be combined with on-line heuristics (e.g., NF, FF and BF) to design on-line algorithms for arbitrary deadlines under EDF.

Figure 3.3 to Figure 3.6 show how the different algorithms behave for different numbers of tasks and an increasing maximum task utilization. The heuristic DensityFFD is the most pessimistic of all, whereas DeviFF has a good performance in terms of reducing the number of processors when the maximum task utilization is not greater than 0.4. On the other hand, EDFTest1FFD performs very well when the maximum task utilization is greater than 0.4 and independently of the number of tasks n.

EDFTest2FFD is the algorithm that has the smoothest curve giving good results all along the x-axis and independently of n. For $n = 100$ and a maximum task utilization of 0.8 in Figure 3.5,

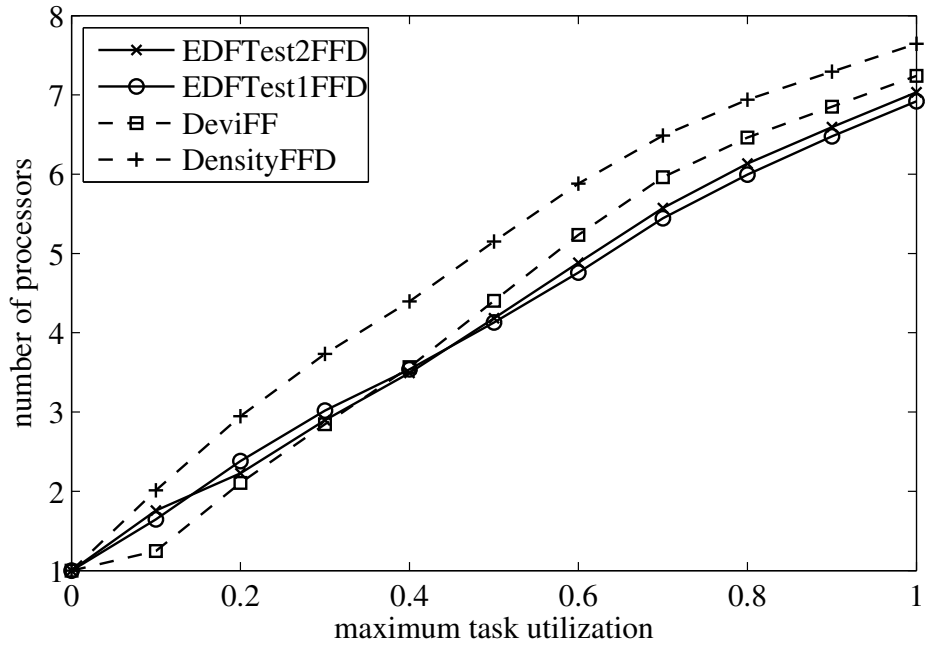


Figure 3.3: Arbitrary deadlines under EDF: average number of processors vs. maximum task utilization, $n = 10$

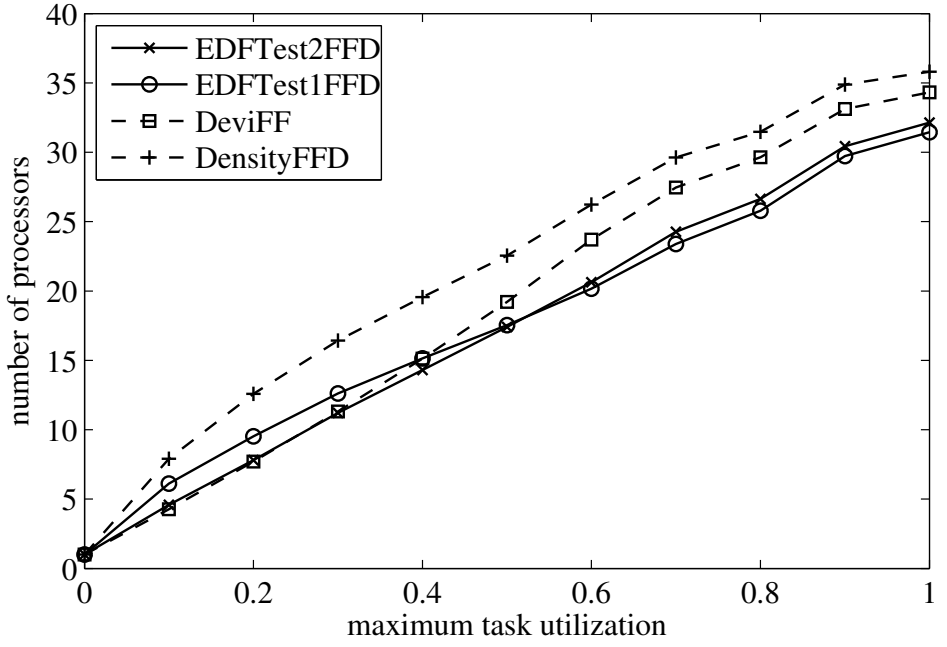


Figure 3.4: Arbitrary deadlines under EDF: average number of processors vs. maximum task utilization, $n = 50$

both algorithms EDFTest1FFD and EDFTest2FFD lead to a reduction of around 10 additional processors when compared to the other heuristics. In Figure 3.6 for $n = 500$ and also a maximum task utilization of 0.8, the additional reduction of the number of processors produced by these algorithms is around 50 processors.

The distance of these algorithms to BestBP is shown in Figure 3.7 to Figure 3.10, where the length of bars represents the number of additional processors compared to BestBP that results from the different algorithms. As discussed above, BestBP yields an estimation of the optimal number of processors in the case that tasks have deadlines equal to periods. So, contrasting the heuristics of this section with BestBP leads to a rather pessimistic comparison. However, these plots give us an idea of how the different algorithms perform with respect to each other and of how their performance degrades regarding the case where deadlines are equal to periods. In Figure 3.9, for $n = 100$ and a maximum task utilization of 0.8, we can notice that EDFTest1FFD yields 10 more processors than BestBP and that DeviFF results in approximately 20 additional processors with respect to BestBP. Figure 3.10 shows that, for $n = 500$ and also a maximum task utilization of 0.8, EDFTest1FFD leads to around 40 additional processors compared to BestBP whereas DeviFF results in over 80 more processors than BestBP.

3.2.2 Algorithms for the DM/RM Scheduling

When considering the DM/RM scheduling, there are also few algorithms that can be designed using feasibility tests from Section 2.2.2. Let us consider the following allocation algorithms: HBFFD, PTASFF, DM/RMTest1FFD, DM/RMTest2FFD and DM/RMTest3FFD. These heuristics apply respectively the hyperbolic bound, the feasibility test of Lemma 9, and the three new feasibility tests proposed for DM/RM.

- HBFFD (Hyperbolic Bound FFD) results from combining the hyperbolic bound from Equation (2.27) and the FFD heuristic, where tasks are sorted according to decreasing (non-increasing) $1 + \frac{e_i}{\min(d_i,p_i)}$ to take arbitrary deadlines into account.
- PTASFF (Polynomial Time Approximation Scheme FF) combines the feasibility test of Lemma 9 and the FF heuristic. The test of Lemma 9 is the analogous for fixed priorities to Devi's test. This test requires tasks to be sorted according to decreasing priority, i.e., according to non-decreasing (increasing) $\min(d_i, p_i)$ under the DM/RM policy. PTASFF first arranges tasks according to increasing $\min(d_i, p_i)$ and then uses the FF heuristic to perform an allocation.
- DM/RMTest1FFD applies DM/RMTest1 of Lemma 8 in Chapter 2 together with the FFD heuristic, where tasks are sorted in order of decreasing $\max\left(\frac{e_i}{\min(d_i,p_i)}, 2\frac{e_i}{p_i}\right)$.
- DM/RMTest2FFD uses DM/RMTest2 of Figure 2.7 together with the FFD heuristic, where tasks are sorted in order of decreasing $\max\left(\frac{e_i}{\min(d_i,p_i)}, 2\frac{e_i}{p_i}\right)$. In contrast to DM/RMTest1, DM/RMTest2 computes the maximum loading factor $\hat{\rho}_{xy}$ (see Lemma 11 in Chapter 2) for groups of two tasks.

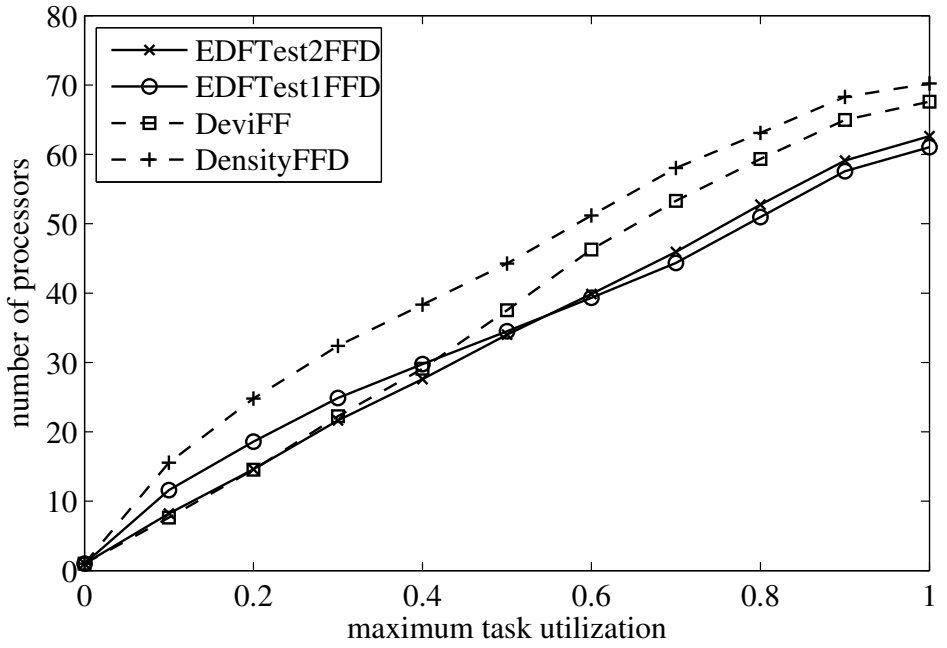


Figure 3.5: Arbitrary deadlines under EDF: average number of processors vs. maximum task utilization, $n = 100$

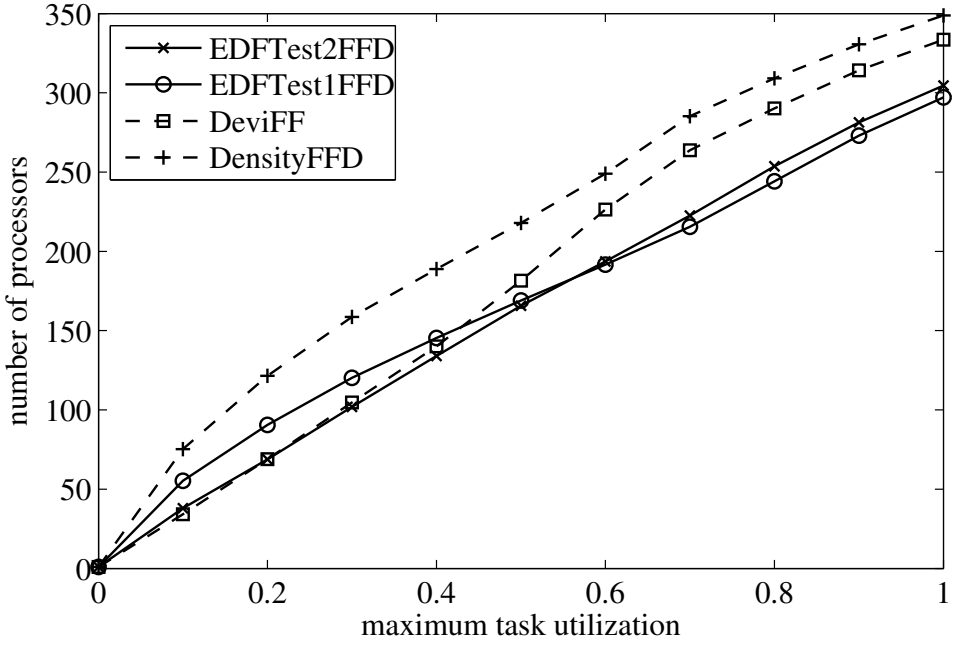


Figure 3.6: Arbitrary deadlines under EDF: average number of processors vs. maximum task utilization, $n = 500$

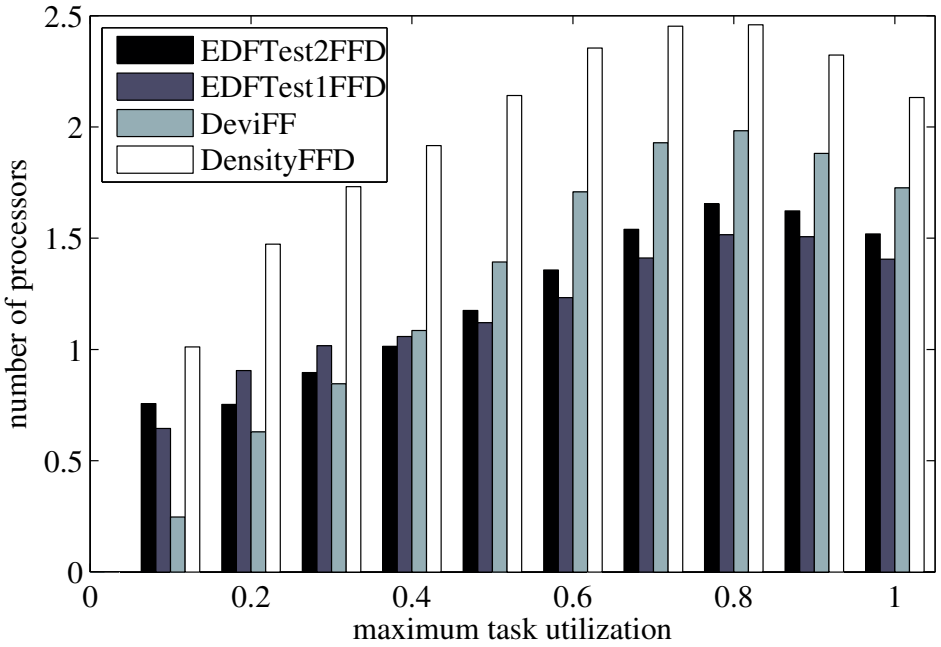


Figure 3.7: Arbitrary deadlines under EDF: average number of additional processors with respect to BestBP vs. maximum task utilization, $n = 10$

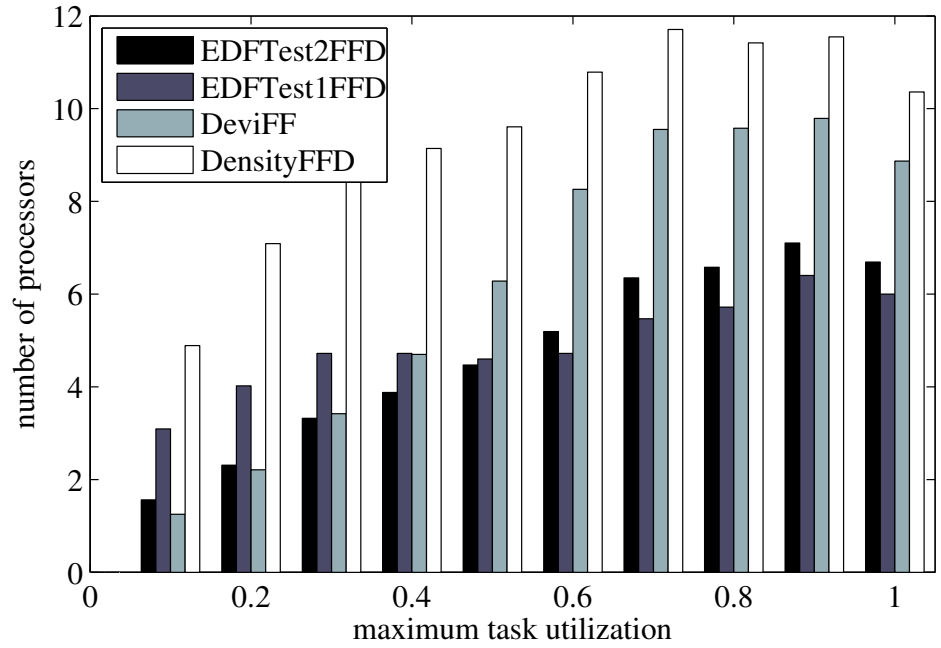


Figure 3.8: Arbitrary deadlines under EDF: average number of additional processors with respect to BestBP vs. maximum task utilization, $n = 50$

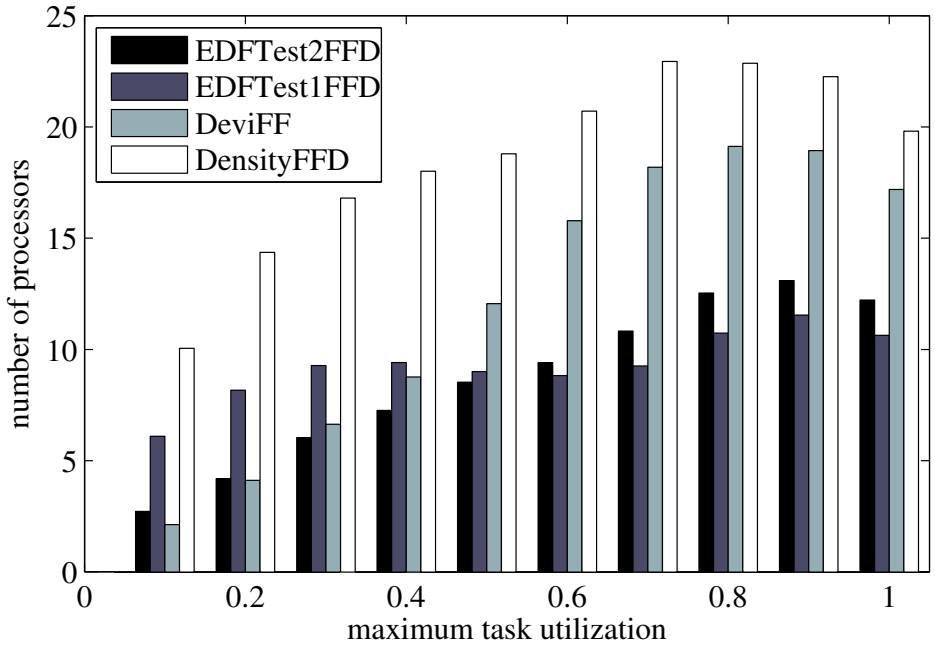


Figure 3.9: Arbitrary deadlines under EDF: average number of additional processors with respect to BestBP vs. maximum task utilization, $n = 100$

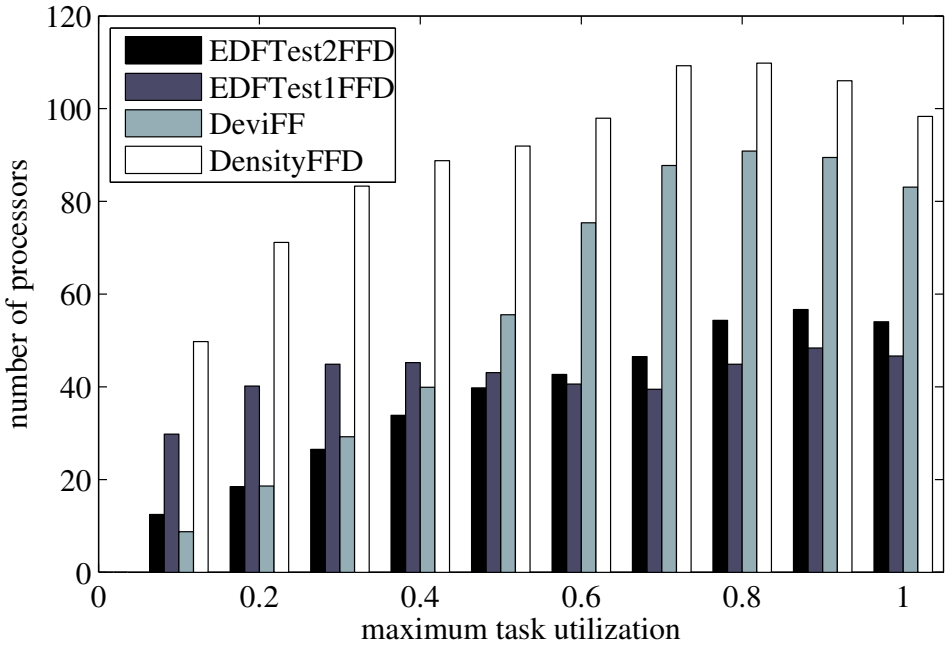


Figure 3.10: Arbitrary deadlines under EDF: average number of additional processors with respect to BestBP vs. maximum task utilization, $n = 500$

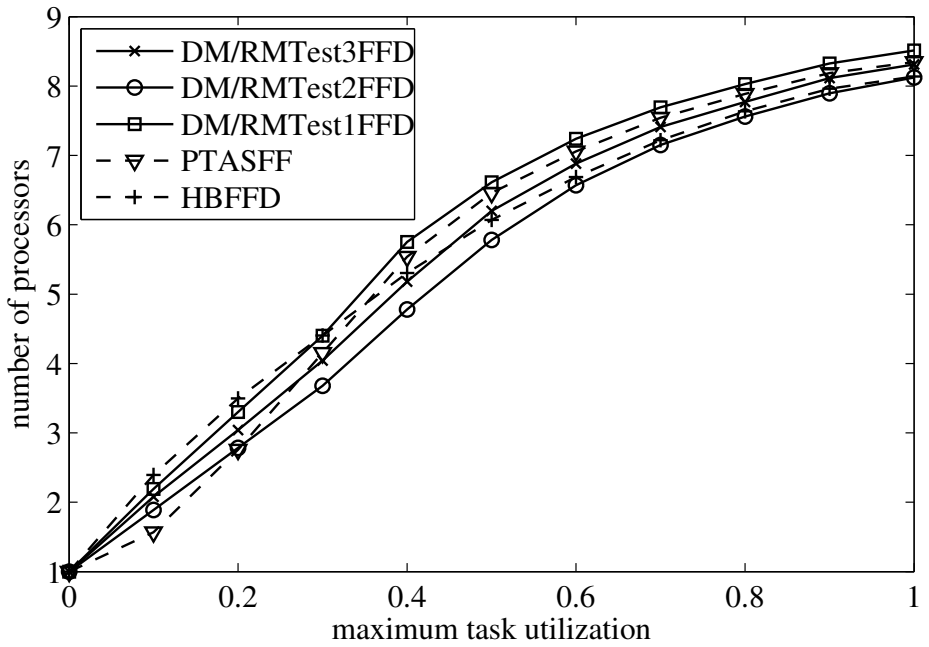


Figure 3.11: Arbitrary deadlines under the DM/RM policy: average number of processors vs. maximum task utilization, $n = 10$

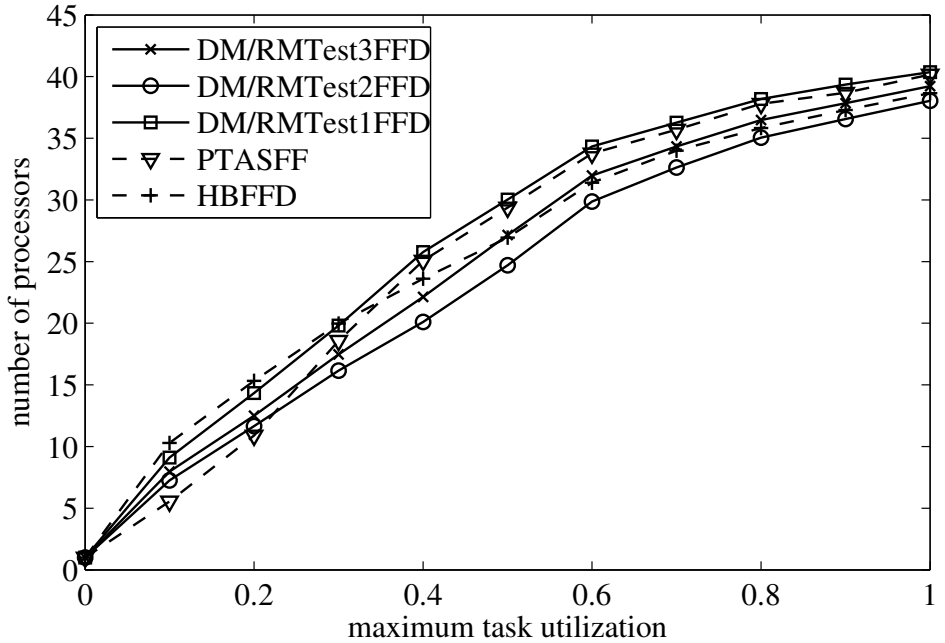


Figure 3.12: Arbitrary deadlines under the DM/RM policy: average number of processors vs. maximum task utilization, $n = 50$

- DM/RMTest3FFD is the combination of DM/RMTest3 shown in Figure 2.8 with the FFD heuristic. Here, tasks are again sorted in order of decreasing $\max\left(\frac{e_i}{\min(d_i,p_i)}, 2\frac{e_i}{p_i}\right)$. The test DM/RMTest3 allows computing the maximum loading factor for groups of several tasks. For this purpose, DM/RMTest3 applies the presented Lemma 12 and Lemma 13 from Chapter 2. As discussed previously, DM/RMTest3 can be configured to compute the maximum loading factor for groups of w tasks. For the case of DM/RMTest3FFD, we chose $w = 10$, i.e., the maximum loading factors $\hat{\rho}_{sup}$ and $\hat{\rho}_{inf}$ are computed for groups of 10 tasks each (see Figure 2.8).

All these algorithms have also polynomial complexity $\mathcal{O}(n \log n)$ like FFD and BFD. So, they neither degrade the complexity of the original heuristics. Because of needing tasks to be sorted according to priorities, the feasibility test of Lemma 9, like Devi's test, can only be used for an off-line allocation. On the other hand, the hyperbolic bound, DM/RMTest1, DM/RMTest2 and DM/RMTest3 can also be combined with on-line heuristics (e.g., NF, FF and BF) to design on-line algorithms for arbitrary deadlines under the DM/RM policy.

Figure 3.11 to3.14 illustrate how the described algorithms perform for different numbers of tasks and an increasing maximum task utilization. They all behave almost the same, however, DM/RMTest2FFD is slightly better than all other algorithms for a maximum task utilization over 0.3 independently of the number of tasks n. For a maximum task utilization less than 0.3, PTASFF is the algorithm leading to a more significant reduction of processors. For $n = 100$ and considering a maximum task utilization of 0.6 in Figure 3.13, DM/RMTest2FFD leads to a reduction of around 5 additional processors compared to the others. For $n = 500$ in Figure 3.14 and also a maximum task utilization of 0.6, the additional reduction of the number of processors produced by this algorithm is around 25 processors.

The distance to BestBP is shown in Figure 3.15 to Figure 3.18, for which the length of bars represents the number of additional processors compared to BestBP that result from the different algorithms. BestBP gives an estimation of the optimal number of processors under EDF and for deadlines equal to periods. As a consequence, comparing the DM/RM allocation heuristics with BestBP is more pessimistic than comparing to an optimal allocation algorithm for DM/RM. However, an optimal allocation algorithm for DM/RM has exponential complexity. It is not practicable to perform a statistical comparison including such an optimal algorithm, so that we use BestBP instead. On the other hand, a comparison of the DM/RM allocation algorithms with BestBP shows, on one hand, how the different algorithms perform with respect to each other. On the other hand, it gives us an idea of the performance degradation regarding the case where processors can be utilized up to 100% (i.e., with respect to EDF and deadlines equal to periods as mention above).

For $n = 100$ in Figure 3.17, DM/RMTest2FFD leads to over 25 more additional processors than BestBP when considering a maximum task utilization of 0.6. In this case, HBFFD, for example, needs over 30 more additional processors than BestBP. For $n = 500$ in Figure 3.18 and also considering a maximum task utilization of 0.6, DM/RMTest2FFD requires around 125 more additional processors than BestBP while HBFFD results in approximately 150 more additional processors with respect to BestBP. The other algorithms introduced in this section show approximately the same behavior.

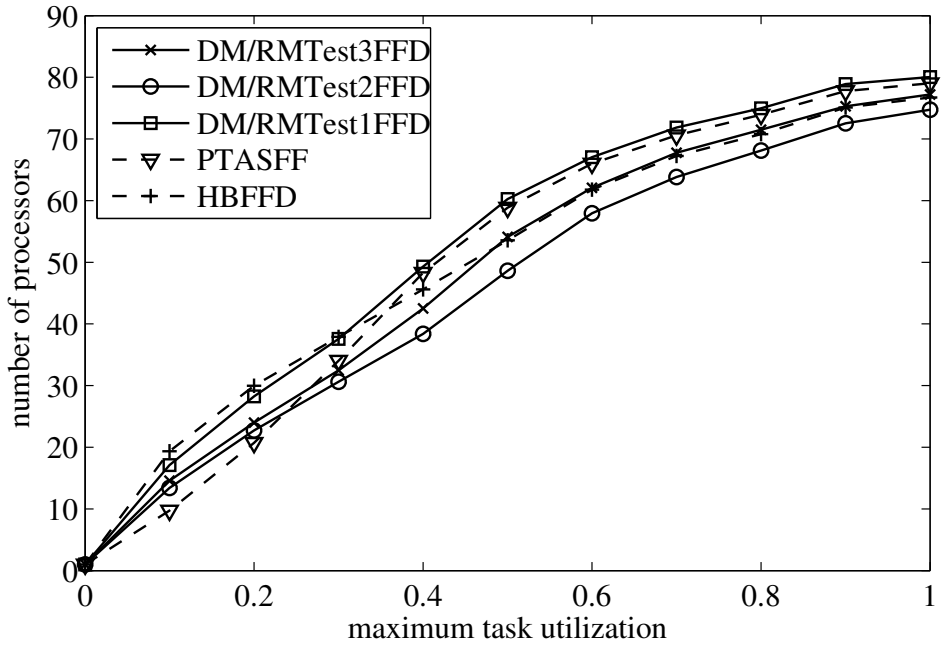


Figure 3.13: Arbitrary deadlines under the DM/RM policy: average number of processors vs. maximum task utilization, $n = 100$

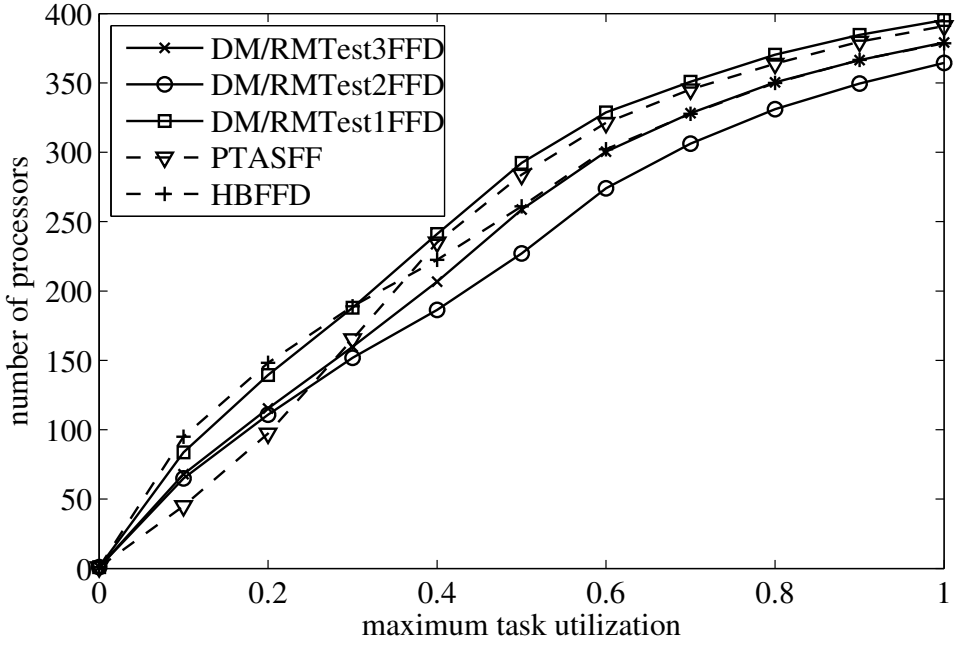


Figure 3.14: Arbitrary deadlines under the DM/RM policy: average number of processors vs. maximum task utilization, $n = 500$

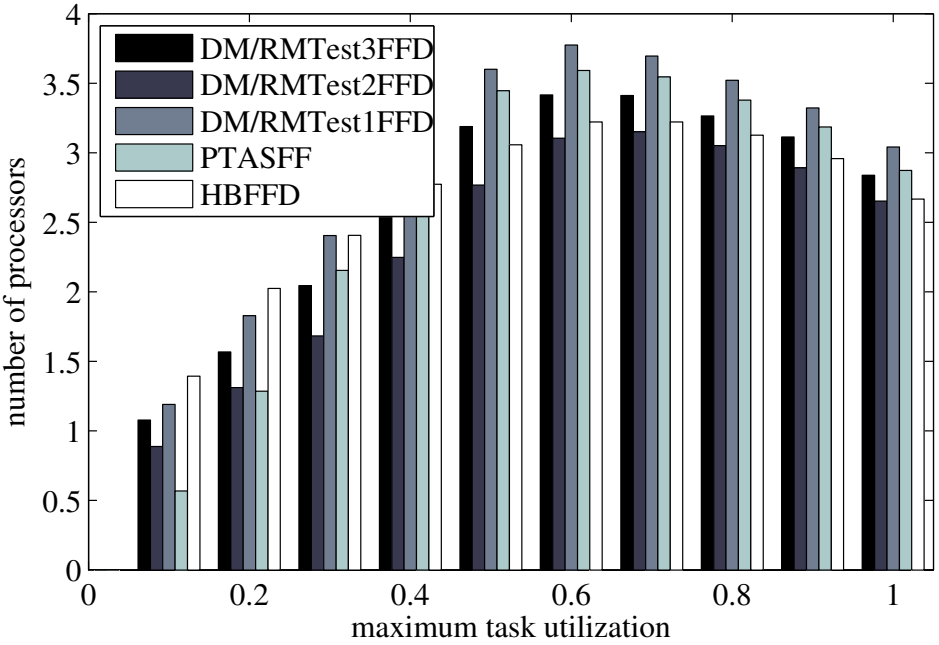


Figure 3.15: Arbitrary deadlines under the DM/RM policy: average number of additional processors with respect to BestBP vs. maximum task utilization, $n = 10$

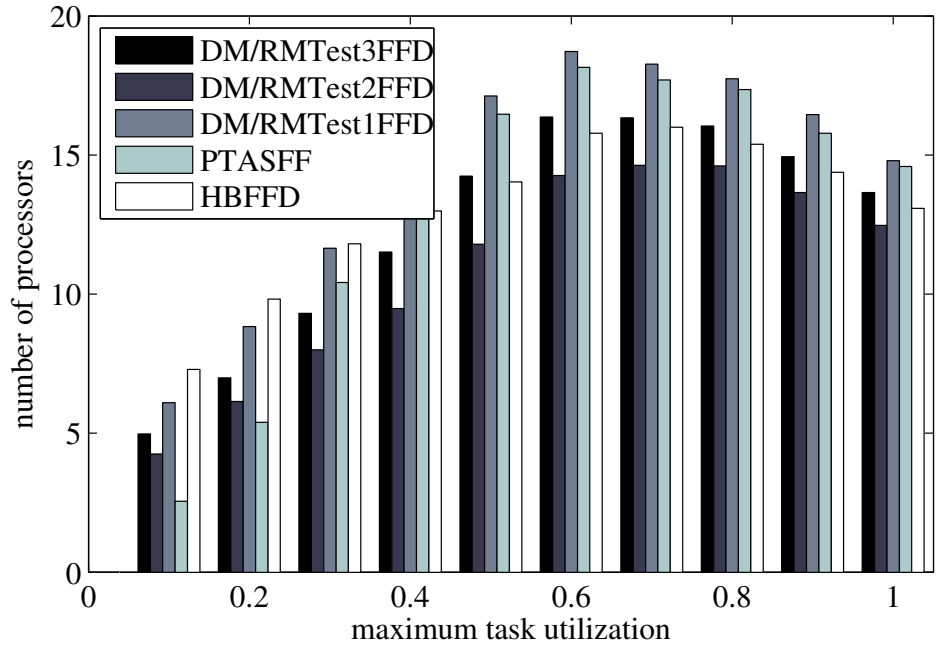


Figure 3.16: Arbitrary deadlines under the DM/RM policy: average number of additional processors with respect to BestBP vs. maximum task utilization, $n = 50$

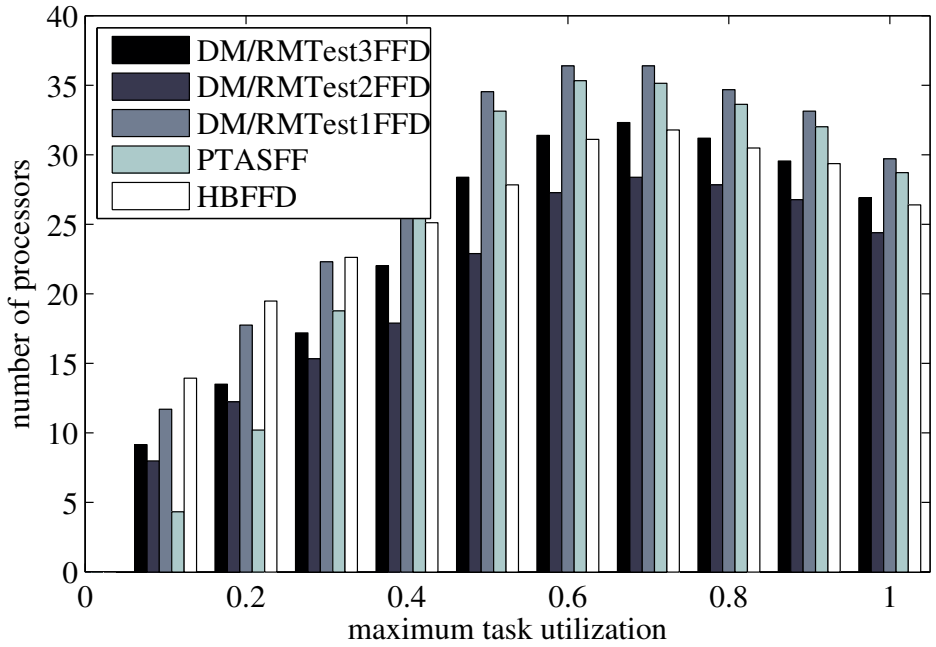


Figure 3.17: Arbitrary deadlines under the DM/RM policy: average number of additional processors with respect to BestBP vs. maximum task utilization, $n = 100$

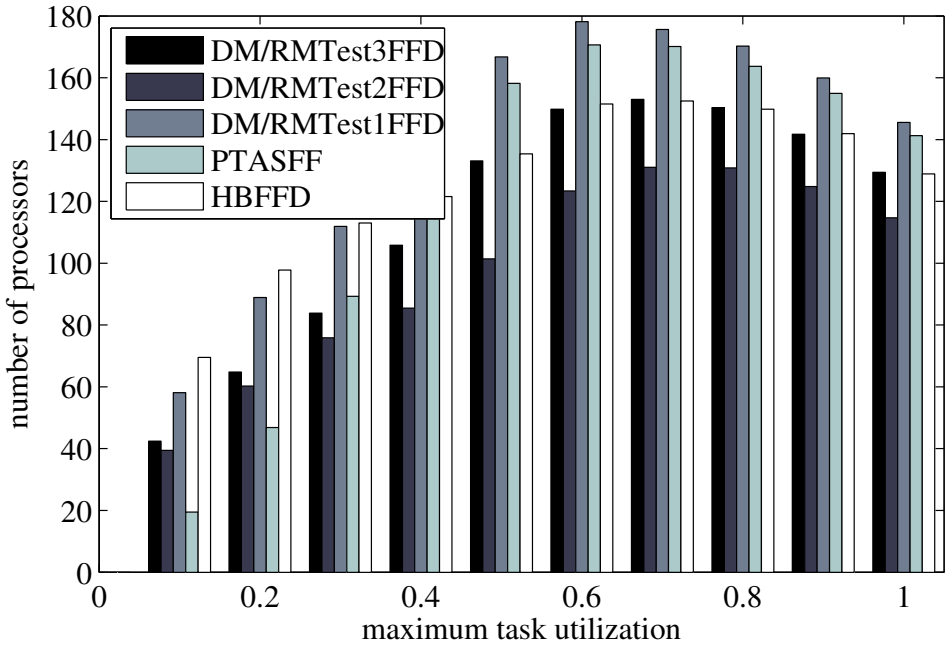


Figure 3.18: Arbitrary deadlines under the DM/RM policy: average number of additional processors with respect to BestBP vs. maximum task utilization, $n = 500$

3.3 Key Findings

In this chapter, we have applied the feasibility tests proposed in Chapter 2 to come up with allocation heuristics for real-time tasks with arbitrary deadlines. In case of an EDF scheduling, we have seen that the heuristic EDFTest2FFD yields a better task allocation, i.e., EDFTest2FFD reduces more the number of processors than the other heuristics for EDF. EDFTest2FFD is the combination of EDFTest2 and FFD. For the DM/RM policy, DM/RMTest2FFD is the heuristic which presents the better results. This latter is the combination of DM/RMTest2 and FFD.

4 Communication and System Constraints

In many practical situations, tasks are not really independent as assumed previously but they interact with each other in some way. In general, there are two types of task dependencies that cover practical design needs to a large extent and that are worth a more detailed analysis: communication and system constraints.

Tasks communicate if they exchange some kind of information. This is the case, for example, when tasks exchange messages containing necessary data for their execution, when there is a given order of execution and some synchronization is needed, etc.

On the other hand, system constraints are preliminary design restrictions, which we have to take into account. For instance, if we were to include some redundancy in order to increase the reliability of the system, it is important not to place two redundant tasks into the same processor. If we were to do so and the processor fails, both redundant tasks will not be available anymore and the redundancy will not comply with its function. This latter necessary mutual exclusion of redundant tasks is what we call a system constraint.

In order to allocate dependent real-time tasks to processors, the task model should be able to describe all dependencies between tasks. For this reason, the task model applied in previous chapters will be first extended to consider communication and system constraints. Afterwards, we adapt the heuristics discussed in Chapter 3 to make use of the extended task model. For the sake of simplicity, we focus on the case under EDF where deadlines are equal to periods (i.e., the case for which the task allocation reduces to the bin packing problem). However, all methods presented in this chapter are general enough and their extension to other discussed scheduling cases is straight forward.

Further, to illustrate the average performance of the different algorithms, an extensive statistical comparison will be presented. In particular, we are interested in analyzing the ability of the different allocation heuristics to reduce the amount of communication between processors.

4.1 Modeling Task Dependencies

In the previous chapters, we have assumed that a task T_i is modeled by a 3-tuple (p_i, d_i, e_i) (i.e., period, deadline and worst-case execution time, where p_i can also be seen as the minimal inter-arrival time between two jobs of T_i). This task model was used to describe the execution

demand of tasks and is still valid in this chapter. However, we have to add some more parameters to be able to capture other task properties like, for example, the amount of communication that it produces, with which other tasks it communicates, etc. The following sections introduce the model used in this thesis to characterize these other task properties.

4.1.1 Communication

In principle, we assume that tasks can send messages to and/or receive messages from any other task in the task set. This establishes a communication dependency between two communicating tasks. For the sake of generality, no restrictions are imposed to the number of communication dependencies that a task may have. So, theoretically, a task can send messages to and/or receive messages from every other task in the system. In this case, the task system is a fully connected and every task sends information to and receives information from every other task in the system.

For every communication dependency, we need to distinguish which two tasks communicate, the amount of information (in bits or bytes) they exchange, the direction of the information exchange (incoming or outgoing messages) and how often messages are sent or received. All this data is contained in the communication matrix $\mathbf{C}_{n\times n}$ where n is the number of tasks in $\mathbf{T}_n$ ($n = |\mathbf{T}_n|$).

The Communication Matrix

As stated above, we need to identify which task sends and which task receives which messages. This information is provided by the communication matrix $\mathbf{C}_{n\times n}$. Columns and rows in the communication matrix are given by tasks T_i in the task set $\mathbf{T}_n$, so that the i-th row and the i-th column correspond to T_i. An element c_{ij} of $\mathbf{C}_{n\times n}$ contains the amount of bits (or bytes) per time unit that are sent from T_i to T_j. In the same way, the element c_{ji} is the amount of bits (or bytes) per time unit that T_i receives from T_j. That is, the i-th row from $\mathbf{C}_{n\times n}$ contains all outgoing messages of T_i, whereas the i-th column contains T_i's incoming messages. To illustrate the use of the communication matrix, let us consider the following example of four tasks:

- T_1 sends $b_{1,2}$ bits to T_2 every p_1 seconds and receives $b_{4,1}$ bits from T_4;
- T_2 receives $b_{1,2}$ bits from T_1 and sends $b_{2,3}$ to T_3 and $b_{2,4}$ to T_4 every p_2 seconds;
- T_3 receives $b_{2,3}$ bits from T_2, $b_{4,3}$ bits from T_4 and sends no message;
- T_4 receives $b_{2,4}$ bits from T_2 and sends $b_{4,1}$ and $b_{4,3}$ bits respectively to T_1 and T_3 every p_4 seconds.

The first row of the communication matrix $\mathbf{C}_{4\times 4}$ represents T_1's outgoing messages. The second row represents T_2's outgoing messages and so forth. Because T_1 sends $b_{1,2}$ bits only to T_2 every p_1 seconds, $c_{1,2} = \frac{b_{1,2}}{p_1}$ while all other entries in T_1's row will be zero. On the other hand, T_2 sends $b_{2,3}$ and $b_{2,4}$ bits to T_3 and T_4 respectively and every p_2 seconds, so $c_{2,3} = \frac{b_{2,3}}{p_2}$, $c_{2,4} = \frac{b_{2,4}}{p_2}$

and all other entries in the second row will be zero. As T_3 sends no messages at all, the third row will have all entries equal to zero. Finally, T_4 sends $b_{4,1}$ and $b_{4,3}$ bits every p_4 seconds to T_1 and T_3, consequently, $c_{4,1} = \frac{b_{4,1}}{p_4}$ and $c_{4,3} = \frac{b_{4,3}}{p_4}$ whereas other entries in the fourth row are zero. The communication matrix for the given example is the following one:

$$\mathbf{C}_{4\times4} = \begin{bmatrix} 0 & \frac{b_{1,2}}{p_1} & 0 & 0 \\ 0 & 0 & \frac{b_{2,3}}{p_2} & \frac{b_{2,4}}{p_2} \\ 0 & 0 & 0 & 0 \\ \frac{b_{4,1}}{p_4} & 0 & \frac{b_{4,3}}{p_4} & 0 \end{bmatrix}.$$

Notice that the main diagonal elements in the communication matrix are always zero ($c_{ij} = 0$ for all $i = j$ where $1 \leq i \leq n$ and $1 \leq j \leq n$), because they would otherwise represent messages from the tasks to themselves. As already mentioned, columns contain the incoming messages per task. However, the information contained in the columns results clearly from filling in the outgoing messages into rows.

Communication Signatures

The communication matrix can be implemented in form of communication signatures. In this case, it is necessary to index all individual messages sent by the tasks, so that each of these messages gets a univocal index. Let us further denote by $\mathbf{M}_m$ the set of messages, where m is the number of messages in the system. Then, every task T_i is assigned a sequence of m bits that we denominate communication signature C_i. Every bit in C_i represents a message in $\mathbf{M}_m$. This way, the k-th bit in C_i refers to the k-th message M_k and will be set to 1 only if T_1 sends the message M_k, otherwise it is set to zero. In our previous example, let us make the following assumptions:

- M_1 denotes the message from T_1 to T_2;
- M_2 and M_3 are the messages from T_2 to respectively T_3 and to T_4;
- M_4 is the message sent from T_4 to T_1 and M_5 the message from T_4 to T_3.

The communication matrix can be rewritten as follows:

$$\mathbf{C}_{4\times4} = \begin{bmatrix} 0 & M_1 & 0 & 0 \\ 0 & 0 & M_2 & M_3 \\ 0 & 0 & 0 & 0 \\ M_4 & 0 & M_5 & 0 \end{bmatrix}.$$

The communication signature C_4 for task T_4 is given by the m-bit sequence "00011", where the first bit represents M_1, the second bit denotes M_2 and so forth. Notice that only the bits 4 and 5 are set to 1 in C_4 because T_4 sends only the messages M_4 and M_5. All other bits in C_4 are consequently zero.

4.1.2 System Constraints

Besides the case of mutual exclusion mentioned above, there are also other system constraints that might be useful to consider. For instance, it might sometimes be interesting to bind one or more tasks to a given processor type presenting certain characteristics like, e.g., an A/D converter, a DMA channel, etc. The procedure we use to model mutual exclusion can be extended to other system constraints as well, so we use the mutual exclusion example to illustrate it.

In order to take mutual exclusion into account when performing a task allocation, we need to identify which tasks should not be placed together on the same processor. This information is provided by the mutual exclusion matrix discussed next. Like the communication matrix, the mutual exclusion matrix $\mathbf{X}_{n\times n}$ is a square one where n is the number of tasks in $\mathbf{T}_n$.

The Mutual Exclusion Matrix

Columns and rows in the mutual exclusion matrix $\mathbf{X}_{n\times n}$ are given by tasks in $\mathbf{T}_n$, so that the i-th row and the i-th column correspond to T_i. An element x_{ij} of $\mathbf{X}_{n\times n}$ is set to 1 if T_i excludes T_j (i.e., if T_i and T_j cannot be placed on the same processor), otherwise it will be set to zero. Clearly, if T_i excludes T_j, T_j also excludes T_i, so the mutual exclusion matrix is symmetric. To illustrate the construction of the mutual exclusion matrix, we consider the following example of four tasks:

- T_1 excludes T_3;
- T_2 does not exclude any task;
- T_3 excludes T_1 (because T_1 excludes T_3) and T_4;
- T_4 excludes T_3 (because T_3 also excludes T_4).

The first row of $\mathbf{X}_{4\times 4}$ represents T_1, the second one represents T_2 and so forth. In the same way, the first column represents T_1, T_2 is represented by the second column and so on. As T_1 and T_3 exclude each other mutually, $x_{1,3} = x_{3,1} = 1$. The other entries for T_1 will be zero while $x_{3,4} = 1$ because T_3 excludes T_4. On the other hand, T_2 does not exclude any tasks so all elements of the second row and column are zero. The mutual exclusion matrix for the previous example is the following one:

$$\mathbf{X}_{4\times 4} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

In the mutual exclusion matrix, like in the communication matrix, elements of the main diagonal are all zero. This is because a task T_i cannot exclude itself.

Mutual Exclusion Signatures

The mutual exclusion matrix can also be implemented in form of binary signatures. This way, every task T_i is assigned a bit sequence denominated mutual exclusion signature X_i, which corresponds to the i-th row of the mutual exclusion matrix and has a length of n bits. Every bit in this signature represents a different task from the task set $\mathbf{T}_n$, so that the j-th bit refers to T_j and so forth. In our previous example, the mutual exclusion signature for task T_3 will be given by the bit sequence "1001", where the first bit represents T_1, the second bit represents T_2 and so on.

4.2 Allocating Dependent Real-Time Tasks

In this section, we extend the algorithms discussed in Chapter 3 to consider communication and mutual exclusion—this latter as an example of system constraints. Other possible system constraints can be modeled as shown for the mutual exclusion case and included in the allocation algorithms as well.

The algorithms presented in this section differentiate from those of Chapter 3 in that they utilize the communication and mutual exclusion matrix. Before we can start discussing how to integrate these two matrices in the allocation procedure, it is necessary to introduce the concept of allocation matrix and matrix of resulting communication. These two matrices contain the information about which task is assigned to which processors and the resulting outgoing messages per processor respectively.

4.2.1 The Allocation Matrix

The algorithms presented in this chapter use an allocation matrix $\mathbf{A}_{q\times n}$ to identify which processors the different tasks have been assigned to. The dimensions of this matrix are given by the number of processors q and the number of tasks n. The number of processors q is increased dynamically during the allocation procedure. At the beginning, there is no processor ($q = 0$), and then processors are dynamically added as they get necessary to reach a feasible task allocation (i.e., the number of rows in $\mathbf{A}_{q\times n}$ changes at run time). The f-th row in $\mathbf{A}_{q\times n}$ corresponds to processor P_f independently of whether P_f is open or closed at the moment. On the other hand, columns in $\mathbf{A}_{q\times n}$ relate to tasks, so that i-th column represents T_i, the j-th T_j and so on. An element a_{fi} in $\mathbf{A}_{q\times n}$ is set to 1 if the task T_i was assigned to the corresponding processor P_f, otherwise it is set to zero.

Let us analyze briefly the following example to clarify how the allocation matrix is built:

- T_1 and T_2 were assigned to processor P_1;
- T_3 was allocated to P_2;
- T_4 was assigned to a third processor P_3.

The first row of the allocation matrix $\mathbf{A}_{3\times4}$ represents the processor P_1, the second row represents P_2 and so forth. Because T_1 and T_2 run on P_1, $a_{1,1}$ and $a_{1,2}$ are both set to 1 whereas all other entries in the first row will be zero. On the other hand, T_3 and T_4 were assigned respectively to P_2 and P_3, so $a_{2,3}$ and $a_{3,4}$ are toggled to 1 and all other elements in these two rows are zero. The resulting allocation matrix $\mathbf{A}_{3\times4}$ is the following one:

$$\mathbf{A}_{3\times4} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Clearly, a task T_i can only be assigned once to only one processor, so if $a_{fi} = 1$ holds, all other elements of the i-th column must be zero.

Allocation Signatures

We can implement the allocation matrix by means of binary signatures as well. Each processor P_f is given a bit sequence denominated allocation signature A_f, which corresponds to the f-th row of the allocation matrix and has a length of n bits. Every bit in this signature represents a different task from the task set $\mathbf{T}_n$, so that the i-th bit refers to T_i, the j-th bit to T_j and so forth. In the previous example, the allocation signature for P_1 will be given by the bit sequence "1100", where the first bit represent T_1 and so on.

4.2.2 The Matrix of Resulting Communication

For each processor, it is necessary to keep track of which messages are sent out towards other processors. This is the function of the matrix of resulting communication $\mathbf{R}_{q\times n}$. Rows and columns in the this matrix are given by processors and tasks respectively. As for the case of the allocation matrix, the f-th row corresponds to processor P_f while the i-th column corresponds to task T_i. An element r_{fi} of $\mathbf{R}_{q\times n}$ contains the amount of bits (or bytes) per time unit that is sent from the tasks on P_f to T_i, if T_i runs on another processor. That is, the i-th entry of the f-th row from $\mathbf{R}_{q\times n}$ contains the sum of all outgoing messages from P_f towards T_i, if this latter is located on another processor.

To illustrate the use of the resulting communication matrix, let us put together the examples of Section 4.1.1 and 4.2.1:

- T_1 runs on P_1, sends $b_{1,2}$ bits to T_2 every p_1 seconds and receives $b_{4,1}$ bits from T_4;
- T_2 is executed on P_1, it receives $b_{1,2}$ bits from T_1 and sends $b_{2,3}$ to T_3 and $b_{2,4}$ to T_4 every p_2 seconds;
- T_3 is executed on P_2, it receives $b_{2,3}$ bits from T_2 and $b_{4,3}$ bits from T_4, but it sends no message;

- T_4 runs on P_3, receives $b_{2,3}$ bits from T_2 and sends $b_{4,1}$ and $b_{4,3}$ bits respectively to T_1 and T_3 every p_4 seconds.

Because T_1 and T_2 are both on P_1, the message $\frac{b_{1,2}}{p_1}$ will not be sent out of P_1 and $r_{1,2}$ is zero. The only messages going out of P_1 are $\frac{b_{2,3}}{p_2}$ from T_2 to T_3 that runs on P_2 and $\frac{b_{2,4}}{p_2}$ from T_2 to T_4 that runs on P_3. As a consequence, $r_{1,3} = \frac{b_{2,3}}{p_2}$, $r_{1,4} = \frac{b_{2,4}}{p_2}$ and all other elements of the first row are zero. On the other hand, all elements of the second row are zero because T_3 sends no messages to other tasks. Finally, $r_{3,1} = \frac{b_{4,1}}{p_4}$ and $r_{3,3} = \frac{b_{4,3}}{p_4}$ in P_3's row, because T_4 sends messages to T_1 and T_3 that are both executed on other processors.

$$\mathbf{R}_{3\times4} = \begin{bmatrix} 0 & 0 & \frac{b_{2,3}}{p_2} & \frac{b_{2,4}}{p_2} \\ 0 & 0 & 0 & 0 \\ \frac{b_{4,1}}{p_4} & 0 & \frac{b_{4,3}}{p_4} & 0 \end{bmatrix}.$$

Notice that the sum of all elements from the resulting communication matrix constitutes the total amount of communication taking place in the system.

Resulting Communication Signatures

In the same way as discussed above for the other matrices, the resulting communication matrix can be implemented in form of bit sequences. In this case, we also need to identify individual messages that are sent by tasks, so that each of these messages gets a univocal index. If $\mathbf{M}_m$ denotes the set of messages, where m is the number of messages in the system. Then, every processor P_f is assigned a sequence of m bits that we denominate resulting communication signature R_f. Every bit in R_f represents a message in $\mathbf{M}_m$. Just like before, the k-th bit in R_f refers to the k-th message M_k and will be set to 1 only if M_k is sent out of P_f, otherwise it will be zero. As in Section 4.1.1, we can index messages in our previous example as follows:

- M_1 denotes the message from T_1 to T_2;
- M_2 and M_3 are the messages from T_2 to respectively T_3 and to T_4;
- M_4 is the message sent from T_4 to T_1 and M_5 the message from T_4 to T_3.

The matrix of resulting communication can further be rewritten as follows:

$$\mathbf{R}_{3\times4} = \begin{bmatrix} 0 & 0 & M_2 & M_3 \\ 0 & 0 & 0 & 0 \\ M_4 & 0 & M_5 & 0 \end{bmatrix}.$$

Finally, the resulting communication signature R_3 for processor P_3 is given by the m-bit sequence "00011", where the first bit represents M_1, the second bit denotes M_2 and so forth. Notice that only the fourth and fifth bit are set to 1 in R_3 because only the messages M_4 and M_5 are sent out of P_3. All other bits in R_3 are consequently zero.

```
    //Input: T_n=tasks, C_{n×n} and X_{n×n}
    //Output: P_q=processors, A_{q×n} and R_{q×n}

    sort T_n according to a heuristic criterion;
5:  n=size(T_n); //number of tasks
    for i = 1 to n
      q=size(P_q); //number of processors
      if q > 0 //if there are open processors
        for f = 1 to q
10:       if !(A(f,1:n) & X(i,1:n))
            if T_i fits on P_f
              update P_f parameters;
              A(f,i) = 1;
              R(f,1:n) = (R(f,1:n) + C(i,1:n)) .* !A(f,1:n);
15:           break; //exits the inner for-loop
            end if
          elseif f == q
            open P_{q+1};
            update P_{q+1} parameters;
20:         A(q+1,i) = 1;
            R(q+1,1:n) = C(i,1:n);
          end if
        end for
        sort P_q according to heuristic criterion;
25:   else
        open P_1;
        update P_1 parameters;
        A(1,i) = 1;
        R(1,1:n) = C(i,1:n);
      end if
    end for
    return(P_q, A_{q×n}, R_{q×n});
```

Figure 4.1: General pseudo-code for the allocation algorithms

4.3 Allocation Algorithms

As mentioned above, the allocation algorithms presented in this section use the four matrices $\mathbf{C}_{n\times n}$, $\mathbf{X}_{n\times n}$, $\mathbf{A}_{q\times n}$ and $\mathbf{R}_{q\times n}$ to take task communication and mutual exclusion (as an example for system constraints) into account. To illustrate how these four matrices are integrated into the allocation procedure, let us consider the pseudo-code of Figure 4.1. This is a general pseudo-code that can be applied for all heuristics in Chapter 3.

In Figure 4.1, the following input parameters are required: the task set T_n, the communication matrix $\mathbf{C}_{n\times n}$ and the mutual exclusion matrix $\mathbf{X}_{n\times n}$. Notice that other system constraints can be modeled by means of a matrix as in the mutual exclusion case. If so, this system constraint matrix must also be passed to the algorithms as a separate input parameter.

Algorithms based on the pseudo-code of Figure 4.1 populate a set of processors $\mathbf{P}_q$ dynamically together with $\mathbf{A}_{q\times n}$ and $\mathbf{R}_{q\times n}$, which contain information about the processors needed to schedule all tasks. A new processor is added to $\mathbf{P}_q$ when there are no open processors $q = 0$ (lines 25 to 29) and whenever a task T_i cannot be assigned to any open processor P_f (lines 17 to 21), e.g., because it is excluded from every processor. As discussed in the previous chapters, the decision of whether a task fits (i.e., whether it is feasible/schedulable) on a given processor depends on the scheduling. In case that tasks are scheduled under EDF and deadlines are equal to periods for all tasks, we only have to test that the total utilization on the processor P_f does not exceed 100% for $1 \leq f \leq q$. This is a sufficient and necessary condition [LL73]. However, other tests like the ones proposed in Chapter 2 must be applied when no restrictions can be imposed to deadlines or whenever other scheduling algorithms are used (e.g., the DM/RM scheduling policy).

Now, before testing whether a task T_i fits on P_f, it should be checked that no mutual exclusion exists between T_i and other tasks already allocated to P_f. The information concerning all tasks already assigned to P_f is contained in the f-th row of the allocation matrix $\mathbf{A}_{q\times n}$, whereas mutual exclusion restrictions for T_i are given by the i-th row in $\mathbf{X}_{n\times n}$. As a consequence, an element-wise AND (line 10) is performed between this two rows. If this results in a vector of zeroes, there are no excluding tasks already allocated to P_f and T_i's schedulability can be tested in line 11. If including T_i does not result in any deadline miss, T_i can be assigned to P_f (lines 12 to 15). If the vector resulting from line 10 has at least one non-zero entry, there is a task on P_f conflicting with T_i and T_i must be placed somewhere else on another processor.

Every time a new task T_i is assigned to a given processor P_f, the corresponding i-th entry in the f-th row of the allocation matrix must be set to 1 to reflect that T_i is now allocated to P_f (lines 13, 20 and 28). Besides updating P_f's parameters (e.g., the remaining available utilization on P_f or similar depending on the scheduling case), the resulting communication must be calculated for P_f. This can be achieved by summing the i-th row of the communication matrix $\mathbf{C}_{n\times n}$ and the f-th row of the matrix of resulting communication $\mathbf{R}_{q\times n}$ (line 14). The i-th row of $\mathbf{C}_{n\times n}$ contains all outgoing messages that T_i sends to other tasks, while the f-th row of $\mathbf{R}_{q\times n}$ contains all outgoing messages that are sent out from P_f to tasks on other processors. The sum of this two rows is going to result in the amount of communication sent out from P_f when the new task T_i is placed on P_f. However, messages that are sent to tasks that are already allocated to P_f must be omitted. So, the sum of these two communication rows must be then multiplied element by element (this is represented by the operator "$.*$") by the negated f-th row of the allocation matrix. This way, messages being sent to tasks that are already on P_f will be set to zero (line 14).

Sorting the task set $\mathbf{T}_n$ (line 4) and the processor set $\mathbf{P}_q$ (line 24) is something that varies from heuristic to heuristic as discussed in Chapter 3. For example, NF and FF do not perform any kind of sorting. On the other hand, FFD and BFD sort the task set according to decreasing

task utilization (which is the analogous to the item size in the bin packing terminology) and BF and BFD also arrange processors according to increasing available utilization (available space). Apart from this, all mentioned heuristics can be easily derived from Figure 4.1.

When a processor is closed, e.g., when no further task would fit into it, it cannot be accessed anymore and the inner for-loop becomes shorter. However, closed processors' entries in $\mathbf{A}_{q\times n}$ and $\mathbf{R}_{q\times n}$ are still available, so some more elaborated indexing must be implemented to access the right rows in these matrices. This more complex indexing is not really relevant and makes algorithms more difficult to understand, so it was excluded from the pseudo-code of Figure 4.1.

Finally, all algorithms based this pseudo-code return the resulting set of processors $\mathbf{P}_q$, the allocation matrix $\mathbf{A}_{q\times n}$ and the matrix of resulting communication $\mathbf{R}_{q\times n}$. From $\mathbf{P}_q$ and $\mathbf{A}_{q\times n}$, we know where the different tasks are going to run. $\mathbf{R}_{q\times n}$ contains the information concerning the amount of communication that originates from the performed task allocation, which is necessary to dimension the communication medium.

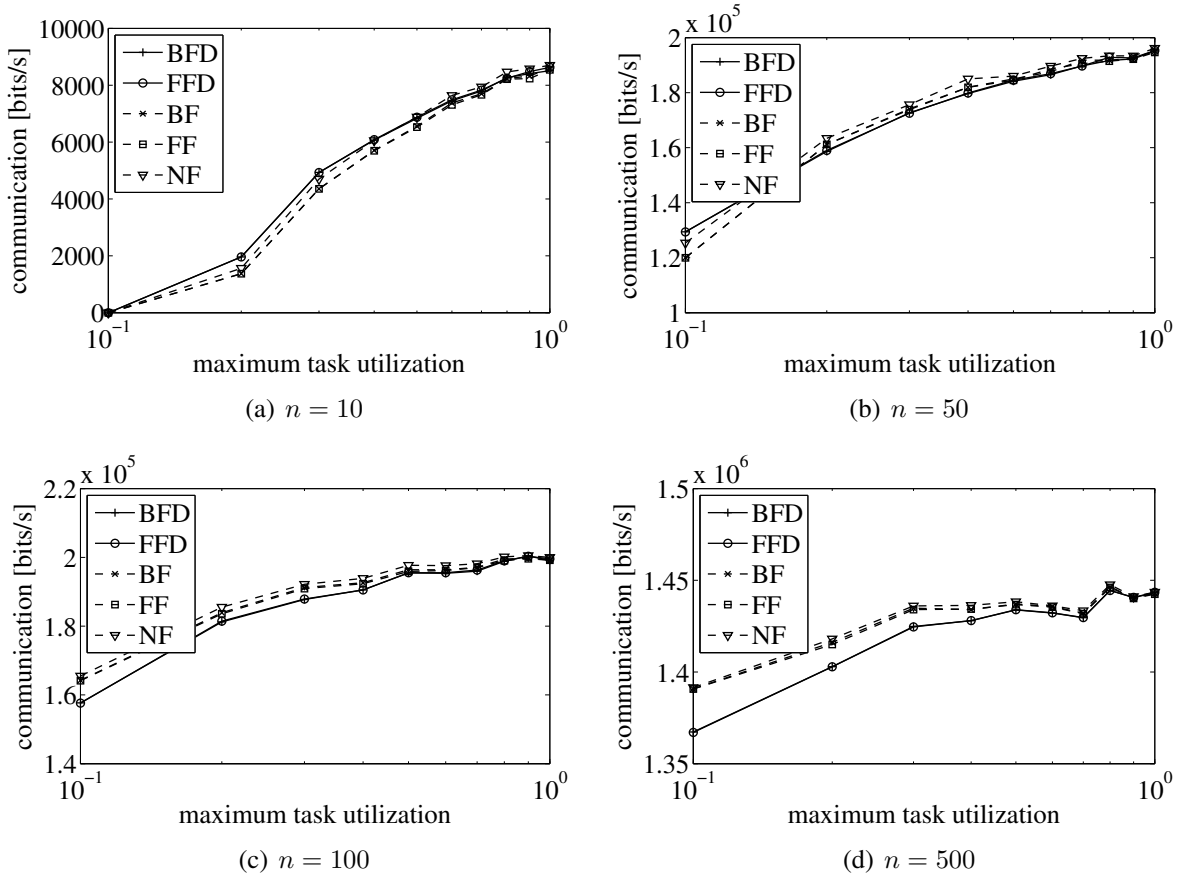
Complexity of the Algorithms

Clearly, ignoring all matrix operations, the pseudo-code presented in Figure 4.1 presents a complexity $\mathcal{O}(n^2)$ in the number of tasks. This is because we have two nested for-loops and the limit of the inner for-loop, i.e., the number of processors q, can be as large as n. We chose to present this pseudo-code with $\mathcal{O}(n^2)$ to ease its exposition, however, algorithms based on it can also be implemented with complexity $\mathcal{O}(n \log n)$. The only exception is NF which can be implemented with linear complexity $\mathcal{O}(n)$.

Now, if we use the correct hardware all these matrix operations (which can be easily parallelized) can be perform in $\mathcal{O}(1)$. Furthermore, programming languages like Matlab are also very efficient to perform matrix operations. On the other hand, we can also use the discussed signatures to implement matrices. This way, all matrix operations reduce to logical operations that can be performed in $\mathcal{O}(1)$. The number of bits in the signatures is still going to depend on the number of tasks (and messages in the case of the communication signature), but this can be limited to a certain upper bound achieving in this way a constant complexity.

When using binary signatures instead of matrices, a bit-wise AND can be performed between the allocation signature A_f of processor P_f and the mutual exclusion signature X_i of T_i in order to verify whether T_i is allowed to run on P_f or not (see line 10 in Figure 4.1). Additionally, every time that messages must be sent out from P_f, the corresponding bits that represent messages in the resulting communication signature R_f will be toggled to 1. In the same way, bits in R_f will be toggled to 0 if messages are not necessary anymore, because the communicating tasks could be placed together on P_f. The activating of messages can be carried out by a bit-wise OR between R_f and the communication signature C_i of task T_i (line 14 in Figure 4.1). Further, messages can be deactivated by performing a bit-wise AND between the result of the previous OR operation and the negated allocation signature A_f (also line 14 in Figure 4.1).

Once all tasks have been allocated to processors, we obtain the set $\mathbf{P}_q$ of necessary processors to guarantee feasibility. Every processor $P_f \in \mathbf{P}_q$ has a communication signature R_f which



(a) $n = 10$ (b) $n = 50$

(c) $n = 100$ (d) $n = 500$

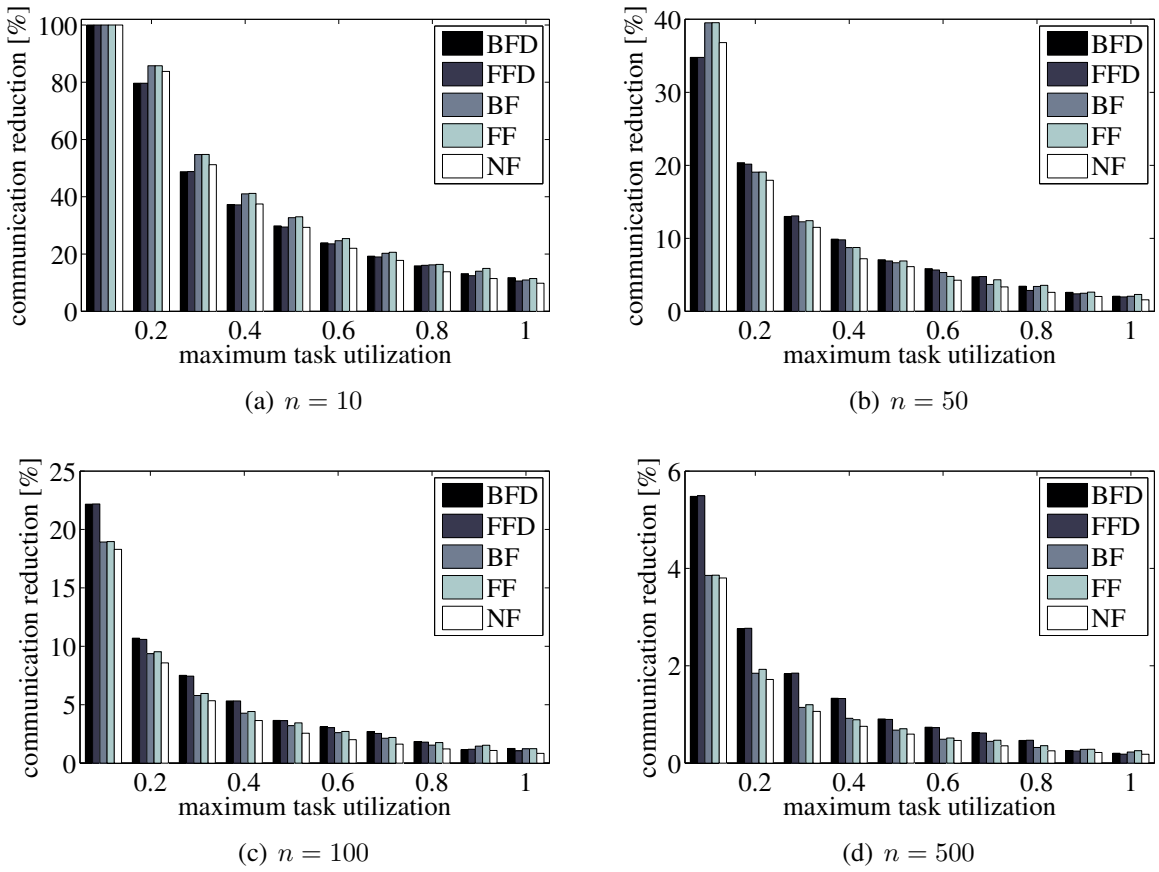
Figure 4.2: Bin packing: average amount of communication vs. maximum task utilization

results from the allocation reached. Which messages must finally be transmitted on the bus can be obtained by performing an bit-wise OR of all R_f and then using the information from the message set $\mathbf{M}_m$ to compute the amount of communication.

4.4 Amount of Communication between Processors

In this section, we analyze how the different bin packing heuristics perform at reducing the amount of communication between processors. Recall that, in this chapter, we focus on the case under EDF where deadlines are all equal to periods, i.e., the case for which the task allocation reduces to the bin packing problem. For this case, we also consider the heuristics previously discussed in Section 3.1.1: NF, FF, BF, FFD and BFD.

For the purpose of obtaining meaningful test data, a large number of task sets were randomly generated for 10, 50, 100 and 500 tasks respectively as described in Section 3.1.2 (i.e., task utilizations within task sets were chosen uniformly between 0 and a varying maximum task utilization, where 1000 different task sets were generated every time the maximum task utilization was increased).



(a) $n = 10$ (b) $n = 50$

(c) $n = 100$ (d) $n = 500$

Figure 4.3: Bin packing: average communication reduction compared to WorstCP vs. maximum task utilization

In order to generate communication matrices for the different task sets, it was assumed that a task can only communicate (send and/or receive messages) at maximum with only one other task in the task set. Further, tasks are supposed to send messages at the end of their execution time, which yields that messages present the same period as the respective tasks. For the presented curves, periods were uniformly distributed in the range of $(0, 1]$ seconds. These were generated first and remain the same for all task sets in each plot, so that the effect of increasing the maximum task utilization on the amount of communication can be illustrated more easily. (Only the periods are uniformly chosen once at the beginning of each plot, all other parameters are randomly created every time a new task set is generated.) The pairs of communicating tasks were chosen randomly with a uniform distribution. Then, for every pair of communicating tasks, random messages were created also uniformly in the range $[0, 1024]$ bits, where 1024 bits is the maximum length assumed for a message (longer messages have an effect on the scale, but not on the curves' shape).

Figure 4.2 shows the amount of communication in bits per second that results from applying the different bin packing heuristics to reduce the number of processors. In this figure, a logarithmic scale was used for the x-axis representing the maximum task utilization, so as to increase the

separation between curves. Figure 4.2 shows that FF and BF result in a greater reduction of the amount of communication than other heuristics for 10 tasks, however, FFD and BFD reduce communication the most when the number of tasks grows. In general, heuristics that are more efficient reducing the number of necessary processors are statistically better to reduce communication. This is because they can place more tasks in one processor producing that more messages can be omitted on average.

Further, let us consider an algorithm referred to as WorstCP (Worst Communication Packing) that returns the worst-case communication scenario. WorstCP calculates the amount of communication as the sum of all messages in the communication matrix: $\sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} c_{ij}$. Clearly, WorstCP is the most pessimistic communication scenario at all, however, we chose to use it as reference for this statistical comparison: the greater the distance to WorstCP, the better the heuristic. The reason for this is that an optimal allocation algorithm to obtain the optimal communication case presents exponential complexity and consequently a huge running time for large numbers of tasks. Performing a comparison to WorstCP further gives a notion of how much the different algorithms reduce the communication between processors when allocating tasks. Figure 4.3 shows the communication reduction achieved by the different algorithms in comparison to WorstCP for 10, 50, 100 and 500 tasks per task set. The length of bars represents how much communication was reduced: a longer bar means a greater communication reduction and consequently a better performance of algorithms. As it can be seen in Figure 4.3, all heuristics produce a communication reduction of at least 40% when the maximum task utilization is less than 0.5 and for $n = 10$. However, they are unable to reduce communication over 5% for a maximum task utilization of 0.5 and $n = 100$.

4.5 Reducing Communication between Processors

The known bin packing heuristics compared in the previous section were designed to reduce the number of processors (the number of bins in the bin packing terminology). However, performing a task allocation to reduce the number of processors results most of the time in an increased communication flow between processors. A greater communication flow produces additional delay because tasks must wait for messages to arrive. Moreover, in embedded systems, it is normally associated with more costs because adding a new communication bus might be required. For these reasons, four additional heuristics are further proposed to reduce the amount of communication between processors. These new heuristics can also be derived from the pseudo-code of Figure 4.1 and are based on the communication volume matrix described in the following section.

4.5.1 The Communication Volume Matrix

The communication volume between two tasks is given by total amount of bits per second that they exchange with each other. For example, if a task T_i sends b_{ij} bits to T_j every p_i seconds and T_j sends T_i an amount of b_{ji} bits every p_j seconds, the communication volume between

T_i and T_j is given by the sum $\frac{b_{ij}}{p_i} + \frac{b_{ji}}{p_j}$. Now, the sum of all messages from T_i directed to T_j is contained in the entry c_{ij} of the communication matrix $\mathbf{C}_{n\times n}$ discussed in Section 4.1.1. Similarly, c_{ji} contains the sum of all messages from T_j to T_i, so that the communication volume between T_i and T_j is given by $v_{ij} = v_{ji} = c_{ij} + c_{ji}$. As a consequence, the communication volume matrix is a symmetric matrix and can be obtained summing the communication matrix and its transpose: $\mathbf{V}_{n\times n} = \mathbf{C}_{n\times n} + \mathbf{C}'_{n\times n}$.

4.5.2 Heuristics to Reduce Communication

As already stated, the heuristics proposed in this section make use of the communication volume matrix $\mathbf{V}_{n\times n}$ in the way explained next.

- The Maximum Communication First Fit Decreasing heuristic (maxCFFD) is based on FF. Similar to FFD, maxCFFD sorts tasks at the beginning. However, the initial sorting in maxCFFD is done according to the decreasing maximum communication volume of tasks and not according to decreasing task utilization as FFD. That is, the tasks T_i in $\mathbf{T}_n$ are sorted according to decreasing values of $\max_{j=1}^{n}(v_{ij})$, i.e., the maximum value in the i-th row of $\mathbf{V}_{n\times n}$. Then, FF is used to perform an allocation to processors.
- The Total Communication First Fit Decreasing heuristic (totCFFD) is also based on FF and sorts tasks according to the decreasing total communication volume that they exchange with all other tasks. The total communication volume of tasks can be obtained by summing up all elements in the communication volume matrix row by row: $\sum_{j=1}^{n} v_{ij}$. After sorting tasks with respect to this parameter, FF is applied to allocate them to processors.
- The Maximum Communication Best Fit Decreasing heuristic (maxCBFD) is based on BF. As maxCFFD, it sorts tasks according to decreasing maximum communication volumes, but then it uses a *best fit* technique to perform an allocation. In contrast to the original BF, processors are not sorted according to remaining available utilization, but they are sorted according to outgoing communication. This means that maxCBFD tries to allocate a task T_i on the processor whose allocated tasks communicate the most with T_i. If this is not possible, it continues with the next processor having the greatest communication towards T_i and so on.
- The Total Communication Best Fit Decreasing heuristic (totCBFD) is also based on BF. The only difference between totCBFD and maxCBFD is that totCBFD sorts tasks according to decreasing total communication volumes like totCFFD, but then it proceeds as discussed for maxCBFD.

Let us further define *task connectivity* as the maximum number of tasks with which any task in $\mathbf{T}_n$ is allowed to communicate (receive messages from and/or send messages to). So, two comparison scenarios can be taken into account. The first one consists in increasing the maximum task utilization with a minimum connectivity between tasks, i.e., for the case where tasks communicate with only one other task in the task set. The second comparison scenario is about increasing the task connectivity from 0 to n for a fixed maximum task utilization.

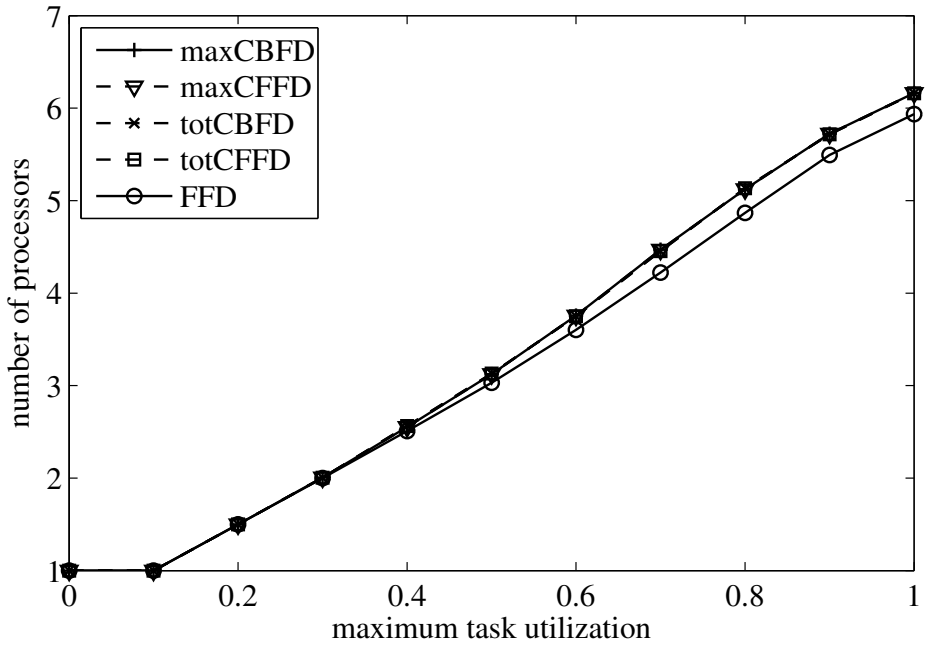


Figure 4.4: Heuristics to reduce communication: average number of processors vs. maximum task utilization, $n = 10$

Before comparing the proposed heuristics with respect to their ability to minimize communication, we first analyze how good these heuristics are in reducing the number of processors.

4.5.3 Processors versus Maximum Task Utilization

All plots in this section are generated as for Chapter 3 for a large number of synthetic task sets and different numbers of tasks per task set. In Figure 4.4 to Figure 4.7, the proposed heuristics are compared against FFD with respect to the number of processors they obtain. As expected, the heuristics to reduce communication are not as effective as FFD when compared with regard to the number of processors that they result in. The proposed algorithms to reduce communication perform similar to FF and BF because of sorting tasks in another order than the decreasing order of task utilization.

Figure 4.8 to Figure 4.11 show the average difference of the new algorithms to BestBP, i.e., the best bin packing algorithm discussed in Chapter 3. Clearly, the proposed heuristics to reduce communication yield more additional processors than FFD. When compared to BestBP, FFD results in around 1 additional processor for $n = 100$ and a maximum task utilization of 0.8—see Figure 4.10. In this case, the proposed heuristics to reduce communication lead to approximately 4 additional processors compared to BestBP. In Figure 4.11, for $n = 500$ and a maximum task utilization of 0.8, FFD still results in 1 additional processor. This time, however, the proposed heuristics are unable to reduce the number of additional processors to less than 10.

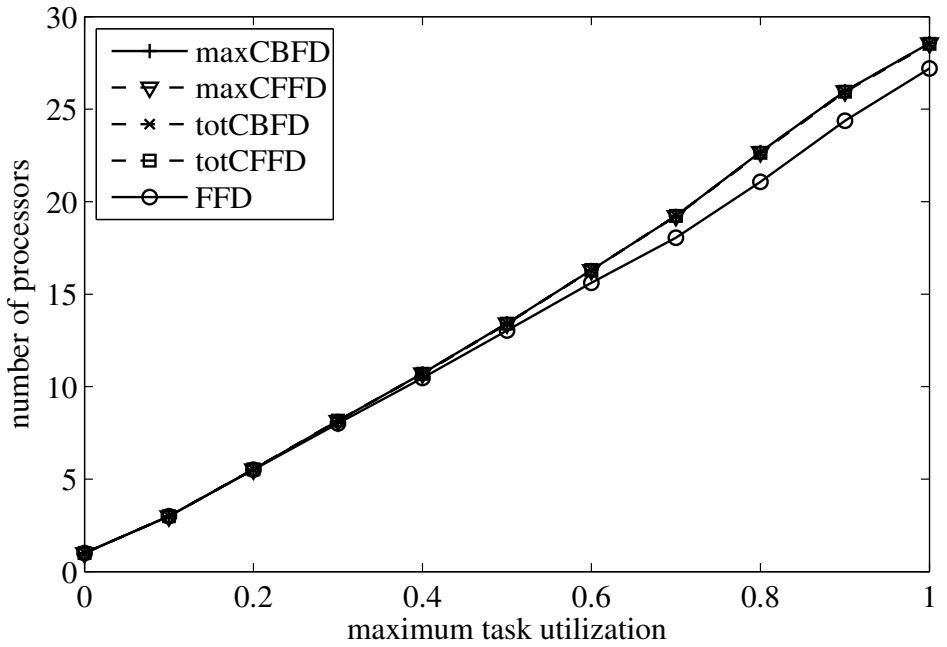


Figure 4.5: Heuristics to reduce communication: average number of processors vs. maximum task utilization, $n = 50$

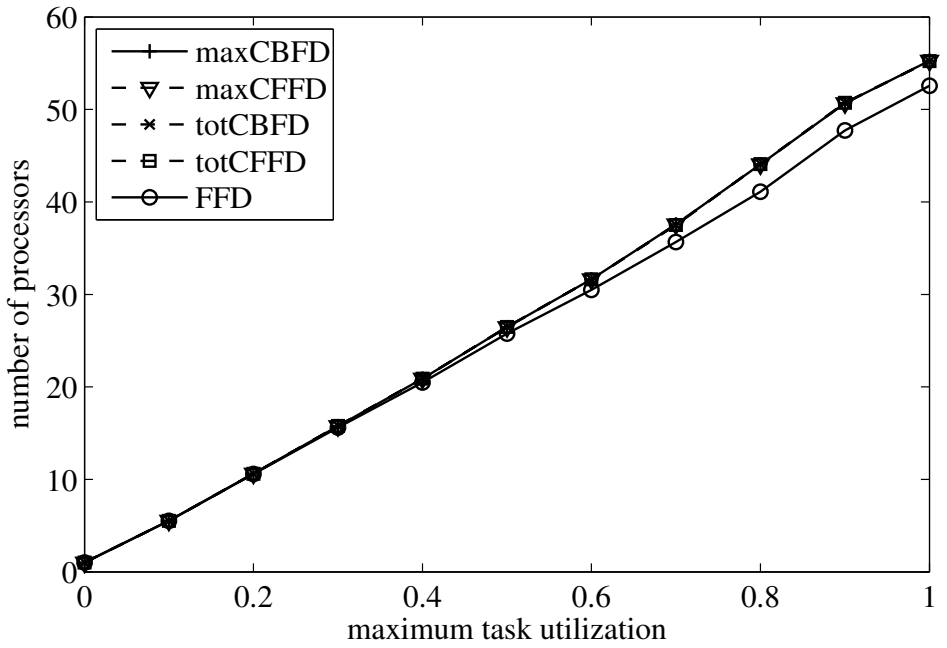


Figure 4.6: Heuristics to reduce communication: average number of processors vs. maximum task utilization, $n = 100$

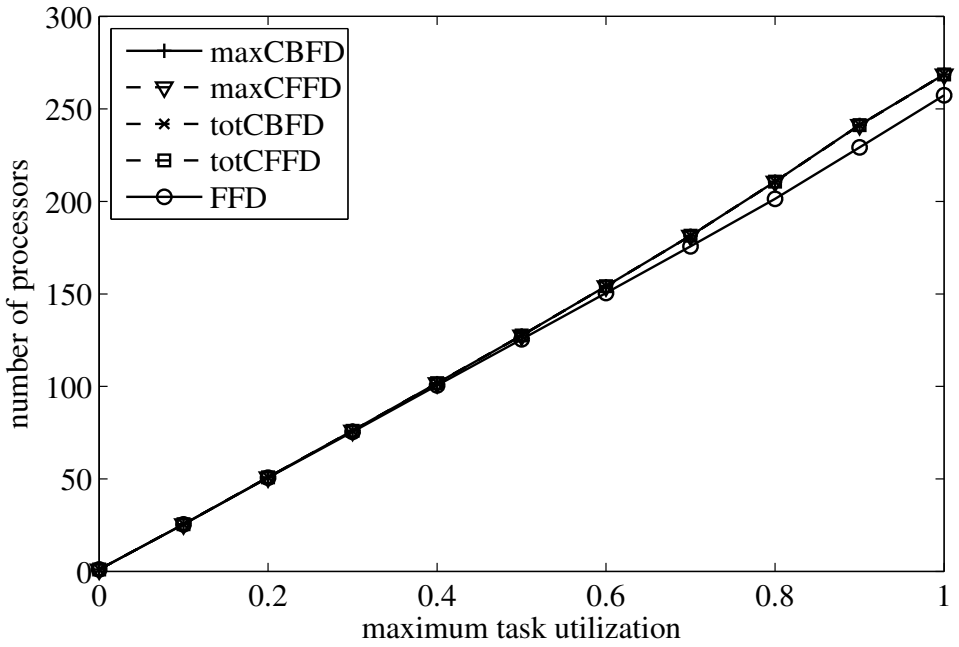


Figure 4.7: Heuristics to reduce communication: average number of processors vs. maximum task utilization, $n = 500$

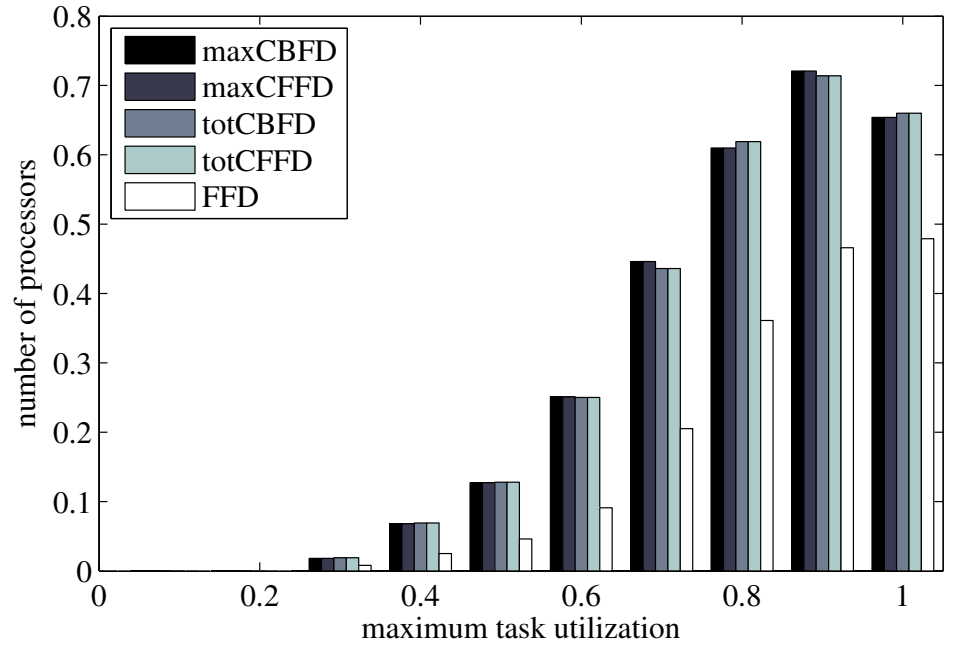


Figure 4.8: Heuristics to reduce communication: average additional number of processors compared to BestBP vs. maximum task utilization, $n = 10$

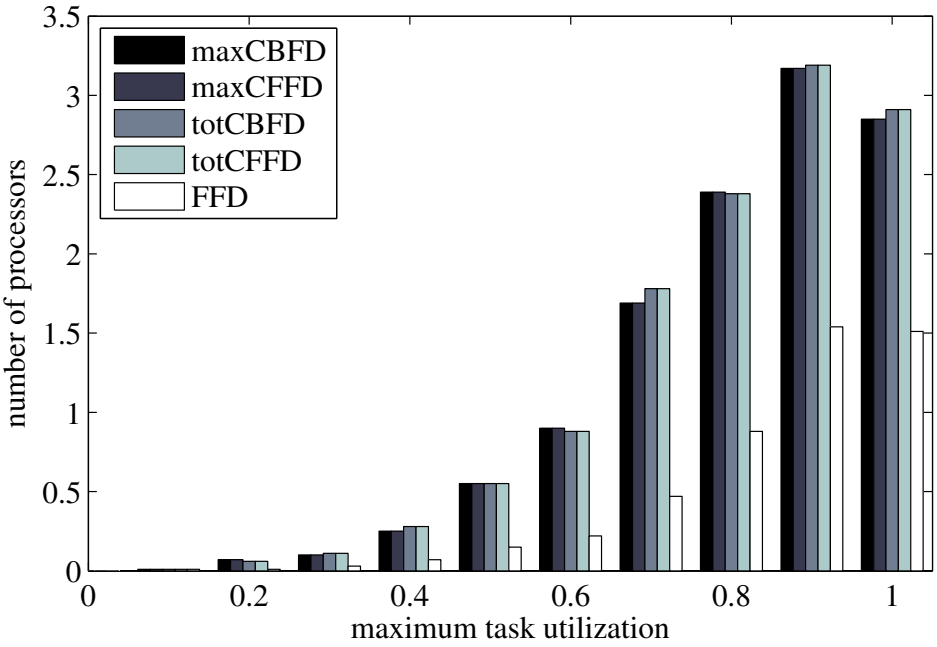


Figure 4.9: Heuristics to reduce communication: average additional number of processors compared to BestBP vs. maximum task utilization, $n = 50$

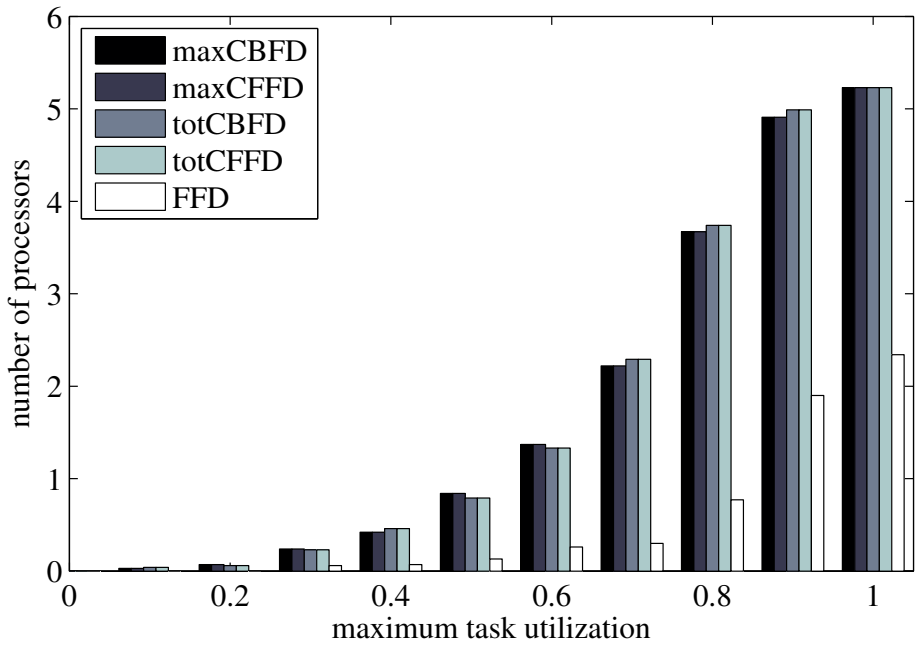


Figure 4.10: Heuristics to reduce communication: average additional number of processors compared to BestBP vs. maximum task utilization, $n = 100$

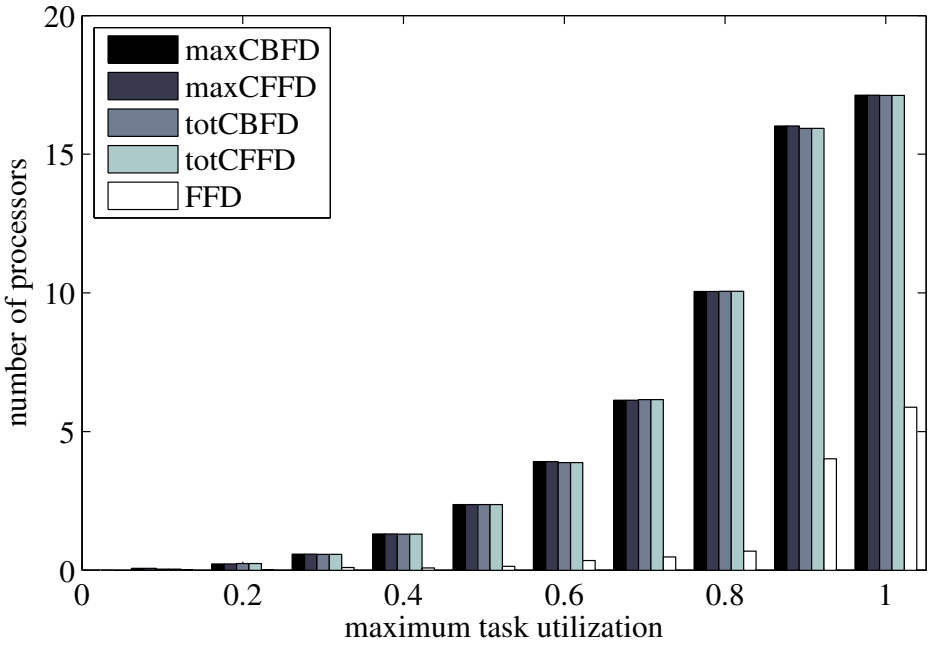


Figure 4.11: Heuristics to reduce communication: average additional number of processors compared to BestBP vs. maximum task utilization, $n = 500$

4.5.4 Communication versus Maximum Task Utilization

In this section, we compare the proposed heuristics to reduce communication with the FFD heuristic. For this purpose, a large number of task sets were generated exactly as for Section 4.4 and also for 10, 50, 100 and 500 tasks respectively.

Communication matrices for the different task sets were generated assuming that a task can communicate with only one other task in the task set. Further, tasks are supposed to send messages at the end of their execution times, so messages present the same period as the originating tasks. In addition, periods were uniformly generated in the range of $(0, 1]$ seconds and remain the same for all task sets in each plot. This way, the effect of increasing the maximum task utilization on the amount of communication can be observed more clearly. The pairs of communicating tasks were chosen randomly with a uniform distribution, for which random messages were created also uniformly in the range $[0, 1024]$ bits.

Figure 4.12 to Figure 4.15 show the amount of communication in bits per second that results from applying the different heuristics to allocate tasks. From this figure, it can be observed that the proposed heuristics to reduce communication present a remarkable improvement over FFD and consequently over all previously discussed bin packing heuristics. Both, maxCFFD and maxCBFD perform exactly the same, i.e., sorting processors as in maxCBFD does not present much advantage from a statistical point of view. In the same way, totCFFD and totCBFD behave very similar to each other. Additionally, the two heuristics based on the maximum communication volume (maxCFFD and maxCBFD) are slightly better than totCFFD and totCBFD, i.e.,

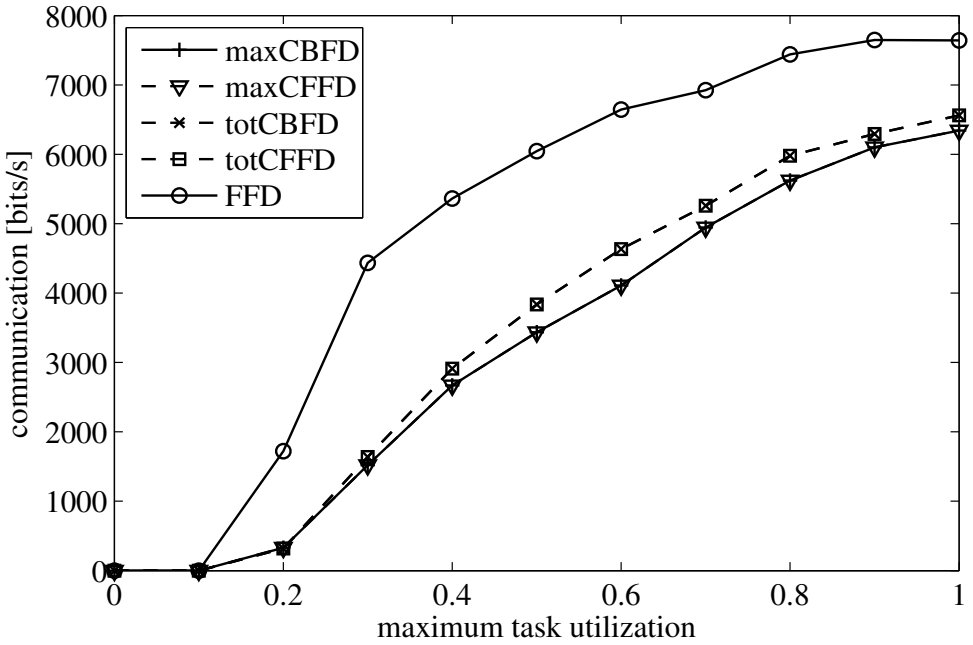


Figure 4.12: Heuristics to reduce communication: average amount of communication vs. maximum task utilization, $n = 10$

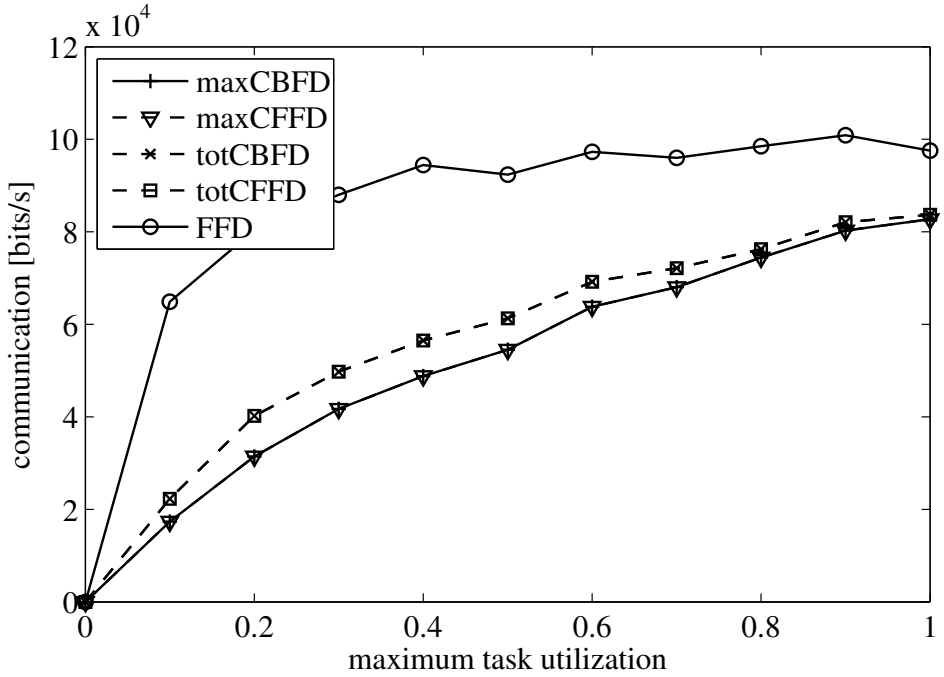


Figure 4.13: Heuristics to reduce communication: average amount of communication vs. maximum task utilization, $n = 50$

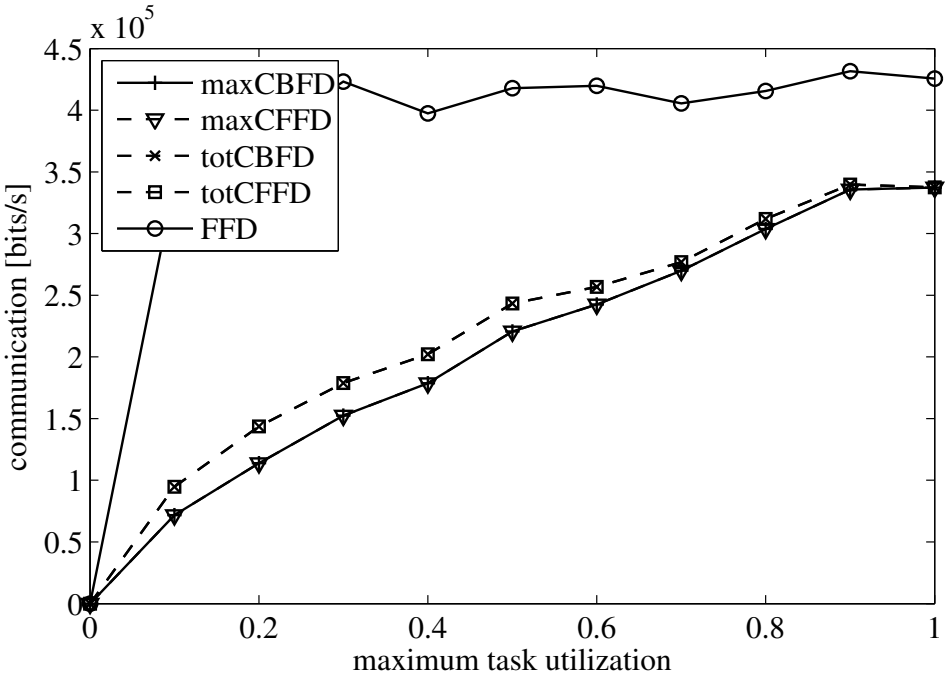


Figure 4.14: Heuristics to reduce communication: average amount of communication vs. maximum task utilization, $n = 100$

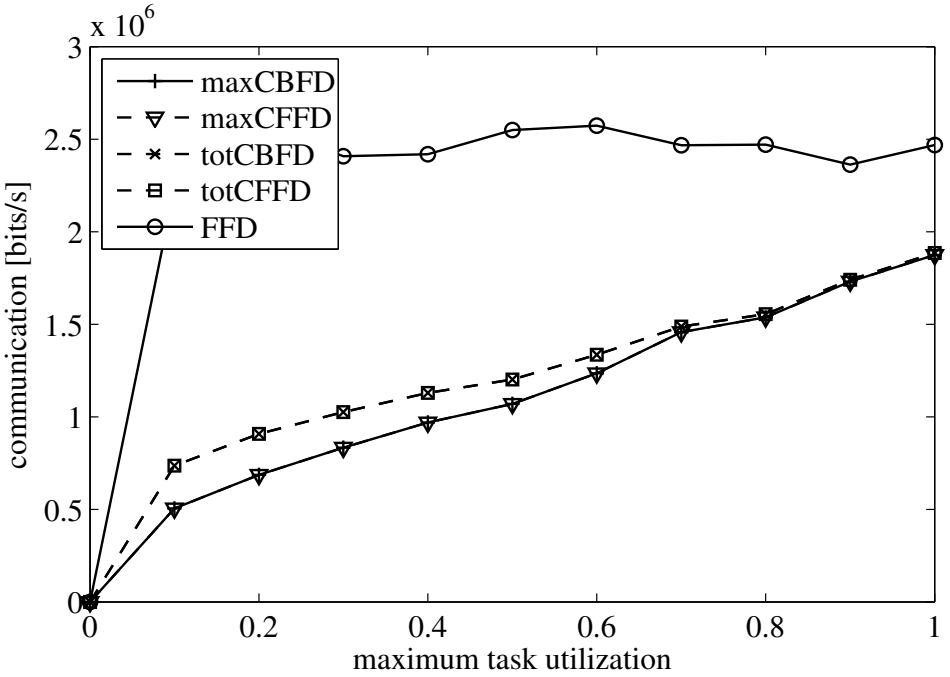


Figure 4.15: Heuristics to reduce communication: average amount of communication vs. maximum task utilization, $n = 500$

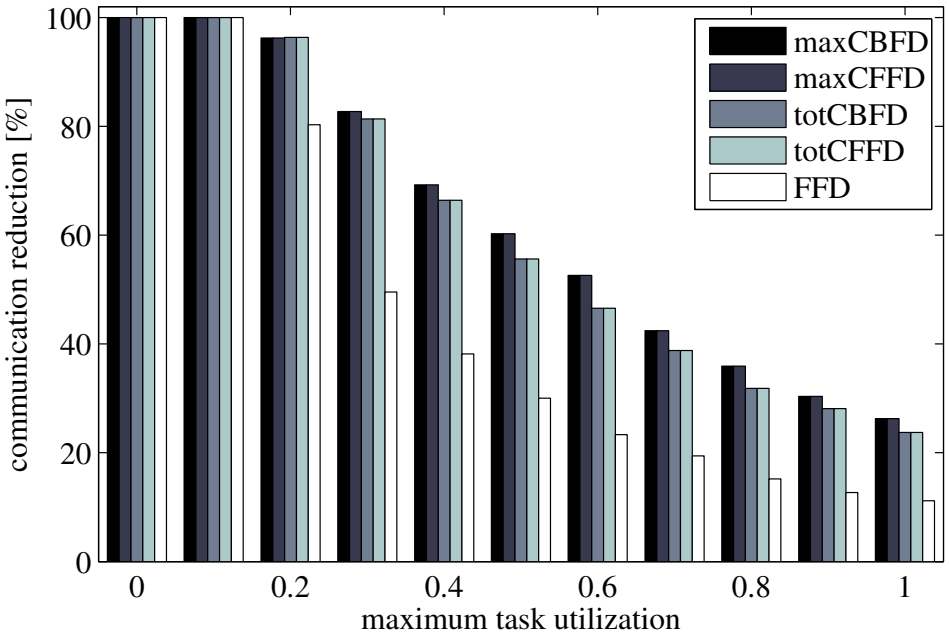


Figure 4.16: Heuristics to reduce communication: average communication reduction compared to WorstCP vs. maximum task utilization, $n = 10$

they result in less communication flow. In Figure 4.14 for $n = 100$ and a maximum task utilization of 0.6, the proposed algorithms lead to 250 Kbps, whereas FFD is unable to reduce the amount of communication more than 420 Kbps. On the other hand, the proposed algorithm reach 1.2 Mbps for $n = 500$ and a maximum task utilization of 0.6 (FFD delivers 2.5 Mbps, see Figure 4.15). Clearly, as the number of tasks grows, the communication flow is also going to grow. (Where in the case of 100 tasks, we have a communication flow of Kbps, we have Mbps for 500 tasks.) This is because every task is allowed to send a message in this comparison. As a consequence, if the number of tasks increases, there will be also more messages that need to be exchanged among processors.

From Figure 4.16 up to Figure 4.19, algorithms are contrasted with respect to WorstCP (the worst-case communication case) for different numbers of tasks. Comparing to WorstCP gives an idea of the communication reduction that can be achieved by the different algorithms. The length of bars represents how much communication was reduced: longer bars mean a greater communication reduction and consequently a better performance of algorithms. As it can be seen in these figures, all heuristics produce a communication reduction of at least 40% compared to WorstCP, when the maximum task utilization is less than 0.5 and independently of the number of tasks. As expected, maxCFFD and maxCBFD are slightly better than totCFFD and totCBFD, and all of them outperform FFD in that they produce a more significant communication reduction. Whereas the communication reduction resulting from FFD degrades drastically as the number of tasks grows, the proposed algorithms allow for a more stable and meaningful reduction of communication.

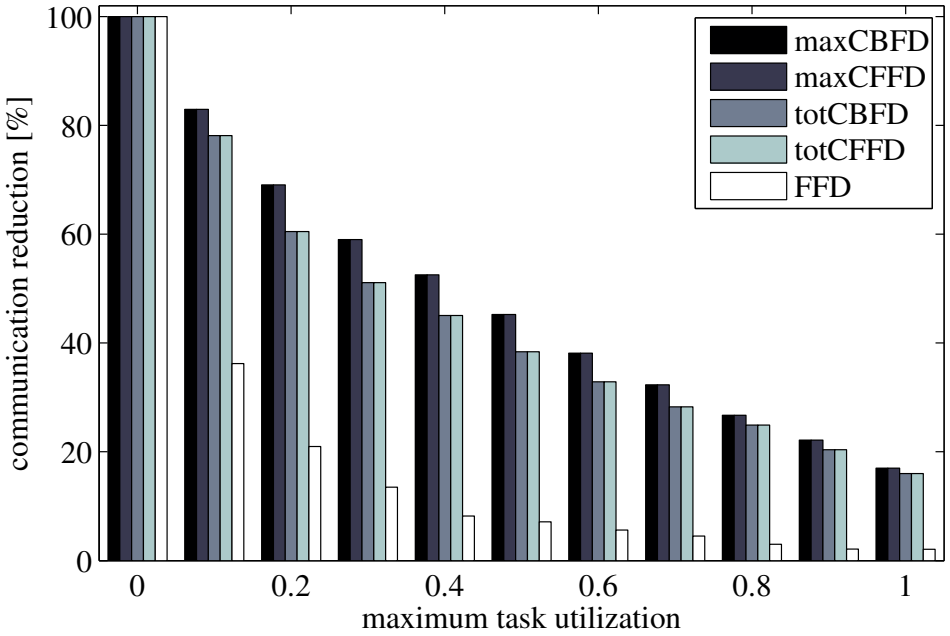


Figure 4.17: Heuristics to reduce communication: average communication reduction compared to WorstCP vs. maximum task utilization, $n = 50$

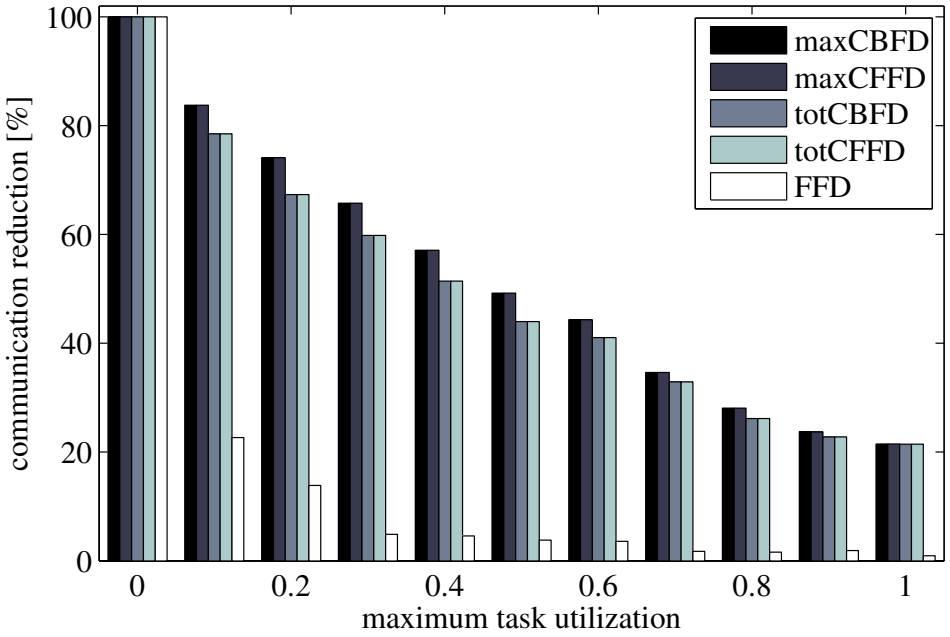


Figure 4.18: Heuristics to reduce communication: average communication reduction compared to WorstCP vs. maximum task utilization, $n = 100$

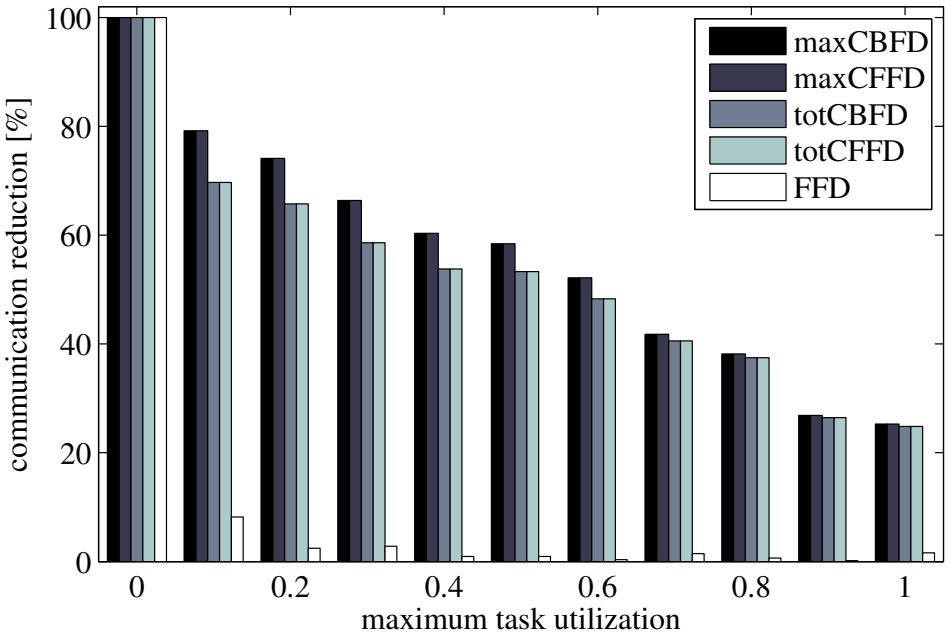


Figure 4.19: Heuristics to reduce communication: average communication reduction compared to WorstCP vs. maximum task utilization, $n = 500$

4.5.5 Communication versus Maximum Task Connectivity

Now, Figure 4.20 to Figure 4.23 show the amount of communication that results as the maximum task connectivity is increased. The task connectivity is defined as the number of tasks with which each task in $\mathbf{T}_n$ is allowed to communicate. The maximum task connectivity will be increased from 0 to n tasks. The case where the maximum task connectivity is 1 was already studied in the previous section and corresponds to a loosely connected task set. On the other hand, when the maximum task connectivity is n, all tasks are allowed to send and receive messages from all other tasks, i.e., we have a fully connected task set.

A large number of task sets were created randomly for 10, 50, 100 and 500 tasks respectively. This time, the maximum task utilization within task sets was fixed to 0.5, i.e., task utilizations were uniformly taken from $(0, 0.5]$. The maximum task connectivity was then increased from 0 to n in uniform steps of $\frac{n}{10}$ tasks each.

Every time the maximum task connectivity was increased, communication matrices were generated for the different task sets. For this, it was also assumed that tasks send messages at the end of their execution times, such that messages have the same periods as the tasks they come from. For the presented curves, periods were uniformly distributed in the range of $(0, 1]$ seconds. These were generated first and remain the same for all task sets in each plot to emphasize the effect of an increasing task connectivity on the resulting amount of communication. (Here again, only the periods are chosen once at the beginning of each plot, all other parameter are randomly generated every time a new task set is created.) The pairs of communicating tasks

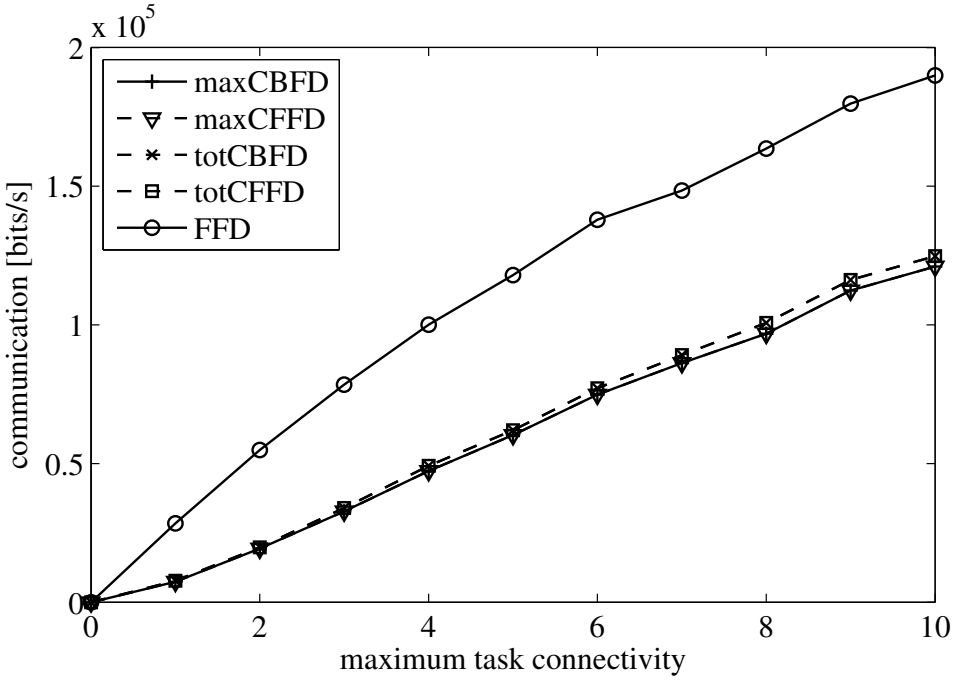


Figure 4.20: Heuristics to reduce communication: average amount of communication vs. maximum task connectivity, $n = 10$

were chosen randomly with a uniform distribution as well. Then, for every pair of communicating tasks, random messages were created also uniformly in the range $[0, 1024]$ bits, where 1024 bits is the maximum length assumed for a message (longer messages have no effect on the shape of curves).

In Figure 4.20 to Figure 4.23, the resulting amount of communication for an increasing maximum task connectivity is compared. As expected, the performance of all algorithms gets very similar as the number of tasks grows. Where for $n = 10$ all heuristics to reduce communication present a interesting improvement over FFD, this practically disappears for $n = 500$. However, this drastic change is also because the task connectivity is increased in steps of 1 task for $n = 10$, whereas the connectivity grows in steps of 50 tasks for $n = 500$. For $n = 10$ and a maximum task connectivity of 6 in Figure 4.20, the proposed heuristics to reduce communication deliver 60 Kbps, while FFD reaches 140 Kbps. In the case of $n = 100$ and maximum task connectivity of 60, FFD achieves 55 Mbps and the new heuristics yield 45 Mbps (see Figure 4.22).

A comparison to WorstCP for an increasing task connectivity is given in Figure 4.24 to Figure 4.27. As expected in Figure 4.24, algorithms produce around 60% communication reduction with respect to WorstCP when $n = 10$ and tasks are allowed to communicate with the half of the other tasks in $\mathbf{T}_n$ (with 5 tasks in this case). On the other hand, in Figure 4.27 for $n = 500$, algorithms reach only between 2% and 1.5% communication reduction when tasks are connected to the half of the other tasks in $\mathbf{T}_n$ (i.e., with 250 tasks).

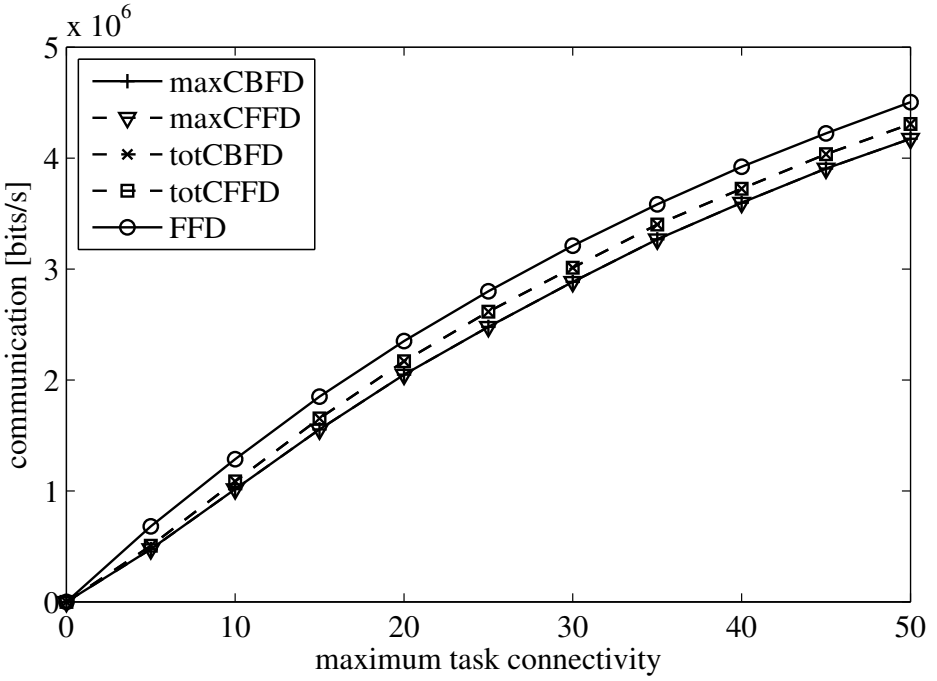


Figure 4.21: Heuristics to reduce communication: average amount of communication vs. maximum task connectivity, $n = 50$

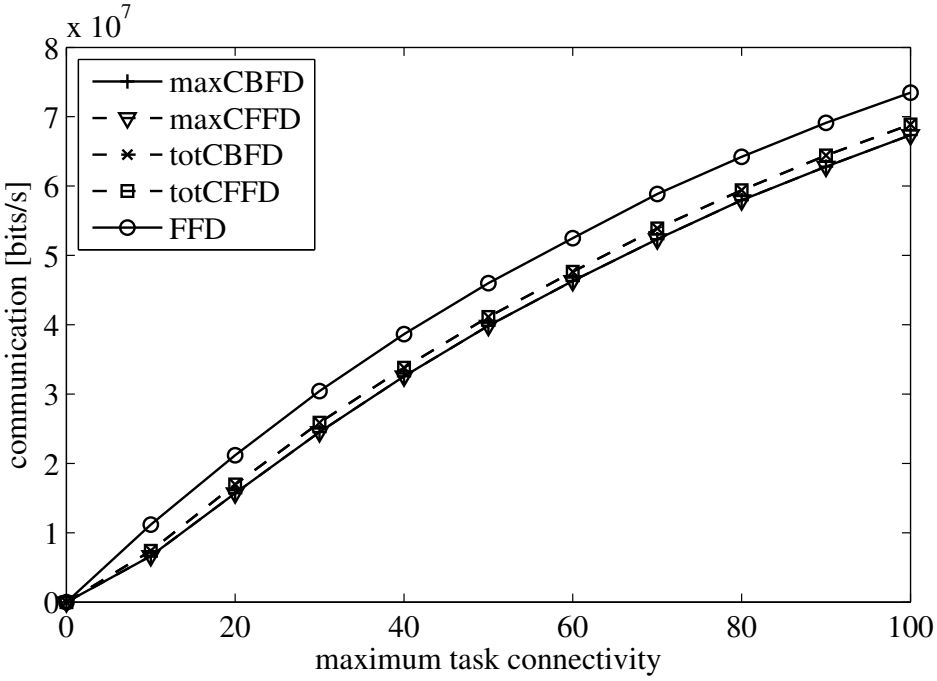


Figure 4.22: Heuristics to reduce communication: average amount of communication vs. maximum task connectivity, $n = 100$

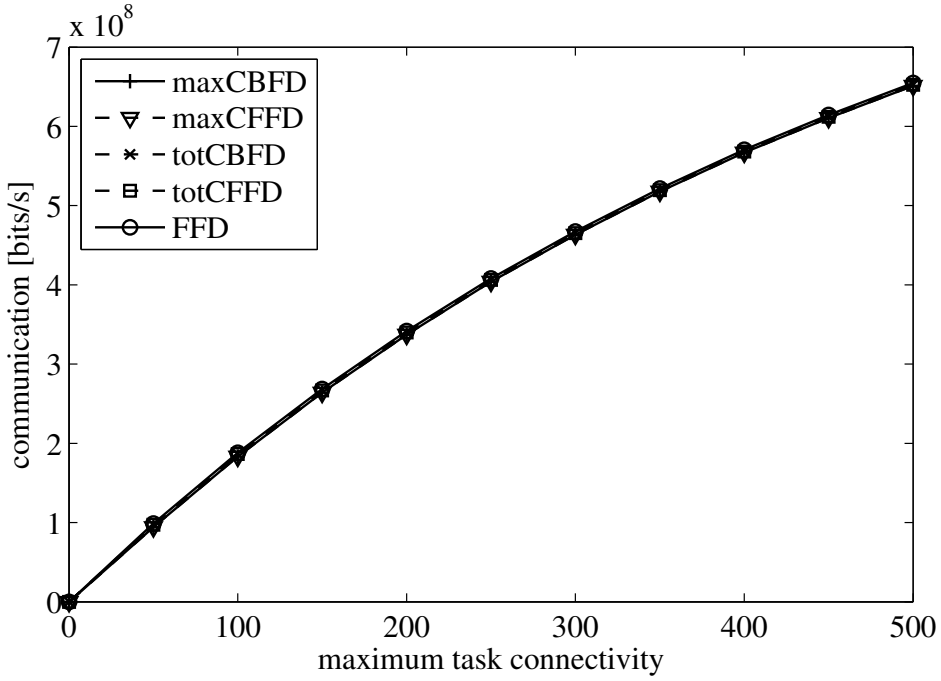


Figure 4.23: Heuristics to reduce communication: average amount of communication vs. maximum task connectivity, $n = 500$

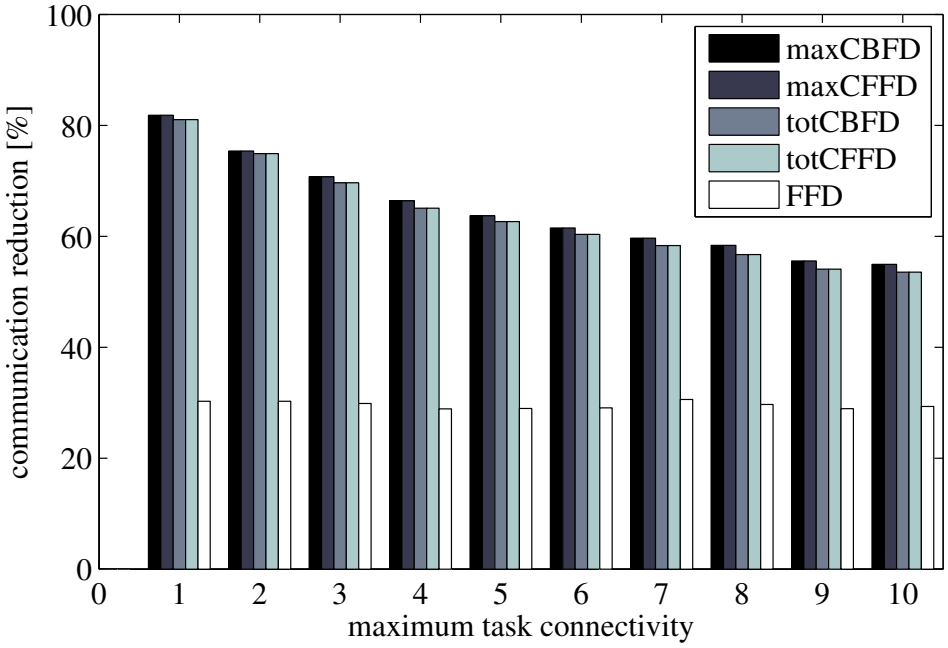


Figure 4.24: Heuristics to reduce communication: average communication reduction compared to WorstCP vs. maximum task connectivity, $n = 10$

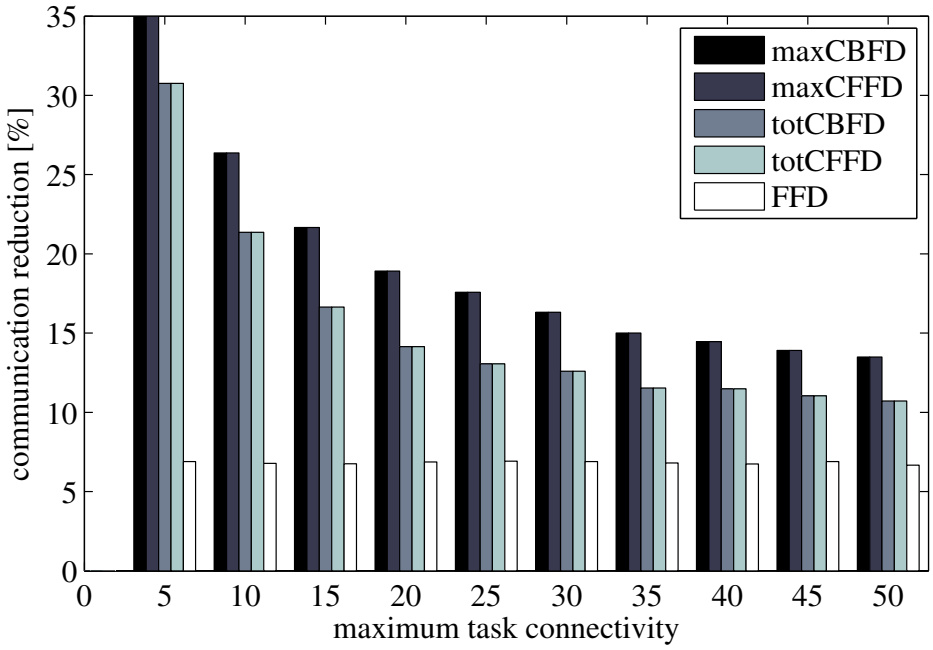


Figure 4.25: Heuristics to reduce communication: average communication reduction compared to WorstCP vs. maximum task connectivity, $n = 50$

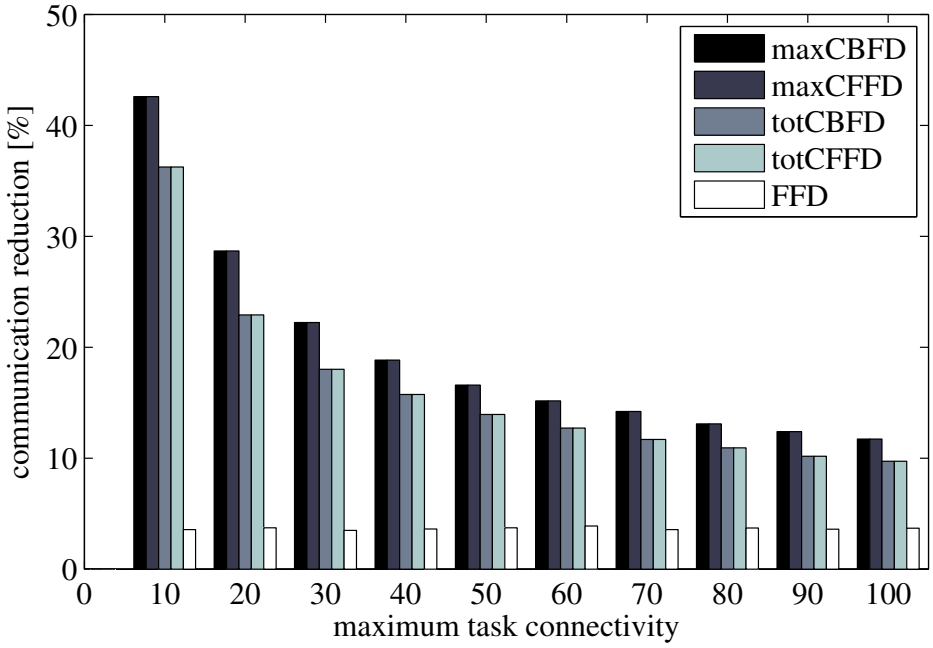


Figure 4.26: Heuristics to reduce communication: average communication reduction compared to WorstCP vs. maximum task connectivity, $n = 100$

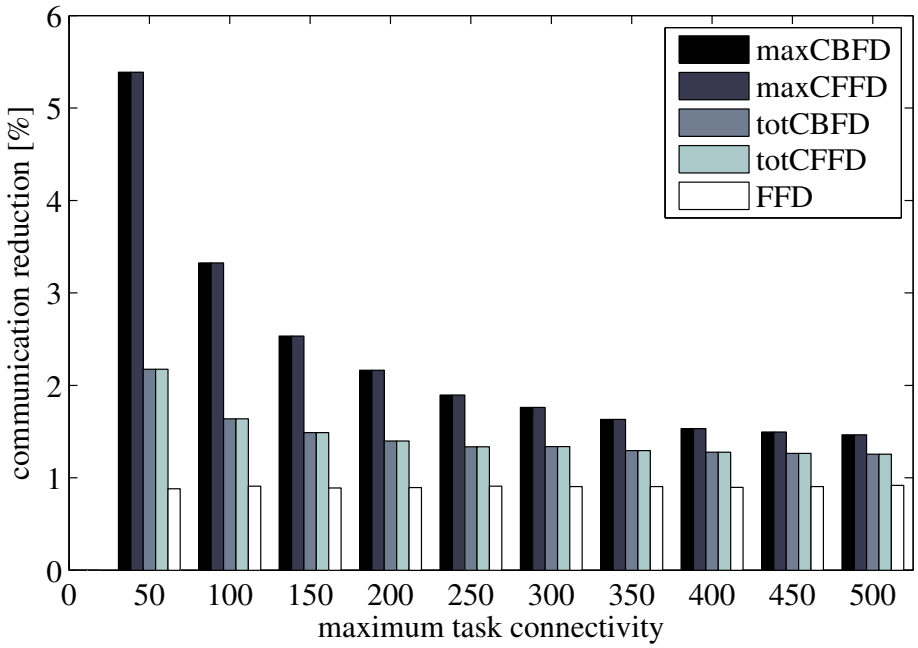


Figure 4.27: Heuristics to reduce communication: average communication reduction compared to WorstCP vs. maximum task connectivity, $n = 500$

4.6 Heuristics to Reduce Processors and Communication

Finally, let us consider two more heuristics that aim at reducing both the total of processors and the communication flow between them. In general, it will not be possible to reduce the number of processors much more than what FFD does (at least with sequential algorithms) [CGJ97]. Nevertheless, we know from Chapter 3 that the number of processors given by FFD is quite close to the optimum (i.e., to BestBP) from a statistical point of view.

On the other hand, it is desirable to combine FFD with the much more better communication reduction obtained by the proposed maxCFFD and totCFFD. With this purpose, we propose two other heuristics described below.

- The Maximum Load First Fit Decreasing heuristic (maxLFFD) is based on FF. Similar to FFD, the maxLFFD heuristic sorts tasks at the beginning. However, the initial sorting in maxLFFD is performed according to decreasing order of a parameter we call maximum load of tasks. The maximum load takes the maximum communication volume and the task utilization into account to express the load produced by tasks on the system. We know from our previous discussion that the maximum communication volume of a task T_i is given by $\max_{j=1}^{n}(v_{ij})$. Let us further represent by v_{max} the maximum of the maximum communication volumes of all tasks, i.e., $v_{max} = \max_{i=1}^{n}\left(\max_{j=1}^{n}(v_{ij})\right)$. Now, the max-

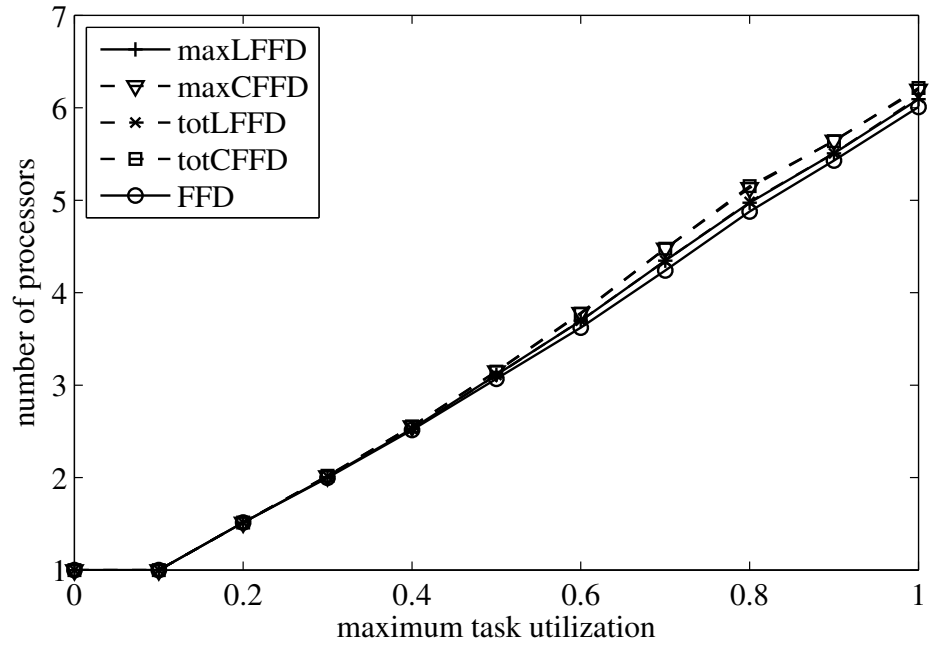


Figure 4.28: Heuristics to reduce processors and communication: average number of processors vs. maximum task utilization, $n = 10$

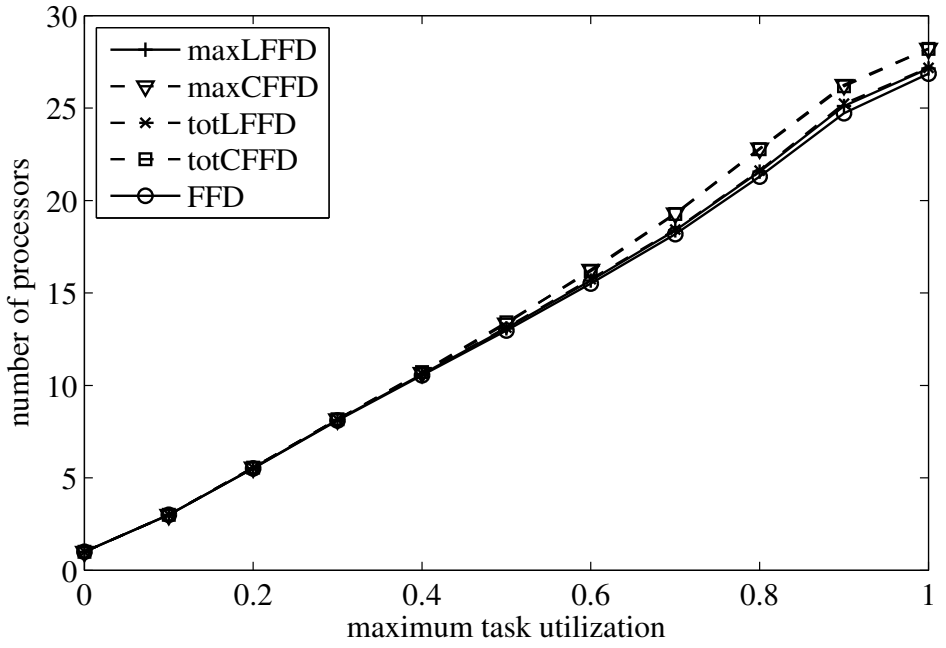


Figure 4.29: Heuristics to reduce processors and communication: average number of processors vs. maximum task utilization, $n = 50$

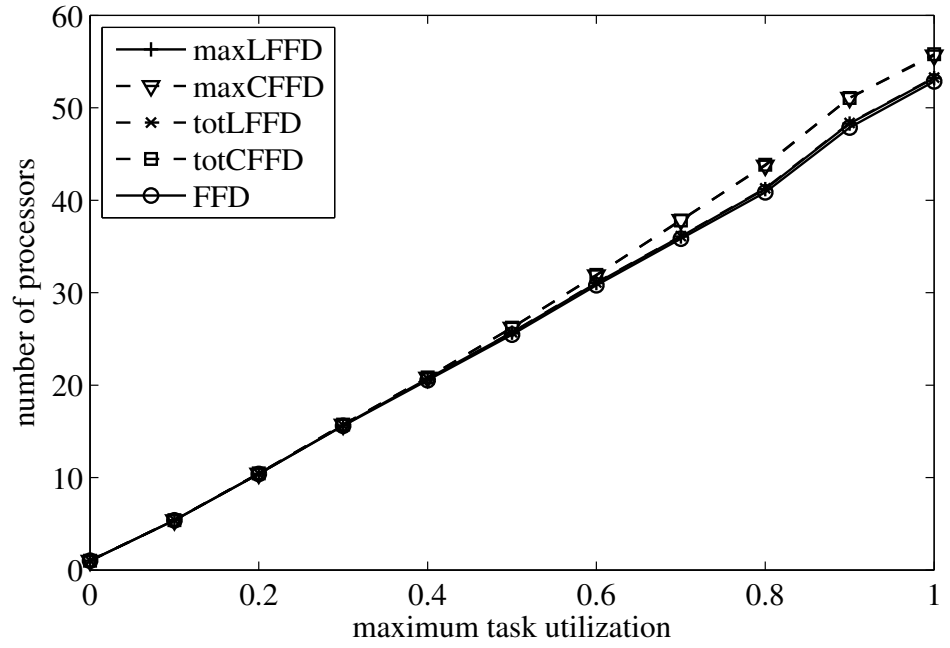


Figure 4.30: Heuristics to reduce processors and communication: average number of processors vs. maximum task utilization, $n = 100$

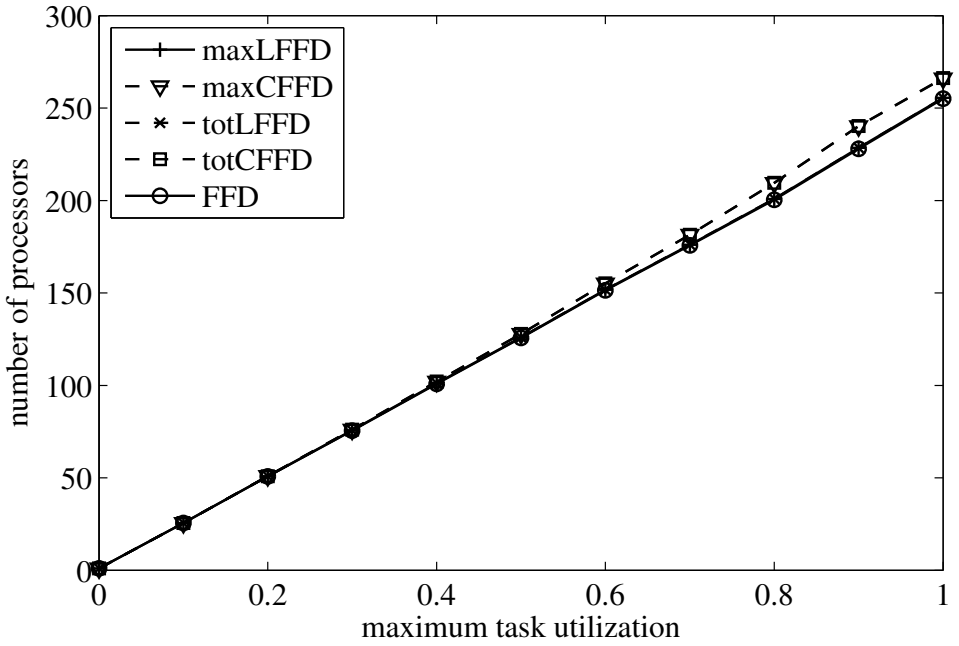


Figure 4.31: Heuristics to reduce processors and communication: average number of processors vs. maximum task utilization, $n = 500$

imum load of T_i will be given by $\frac{\max_{j=1}^{n}(v_{ij})}{v_{max}} + u_i$. This is the maximum communication volume of T_i normalized with respect to v_{max} plus T_i's utilization. Thus, maxLFFD sorts all tasks T_i in $\mathbf{T}_n$ according to decreasing values of this parameter and then performs an allocation based on FF.

- The Total Load First Fit Decreasing heuristic (totLFFD) is also based on FF. This heuristic sorts tasks according to decreasing order of a parameter called total load of tasks. This parameter is similar to the previously defined maximum load, however, it uses total communication volume to come up with a measure of the load produced by tasks on the system. The total communication volume of a task T_i is given by $\sum_{j=1}^{n} v_{ij}$ and v_{tot} is the maximum of the total communication volumes of all tasks, i.e., $v_{tot} = \max_{i=1}^{n}\left(\sum_{j=1}^{n} v_{ij}\right)$. Now, the total load of T_i is defined as $\frac{\sum_{j=1}^{n} v_{ij}}{v_{tot}} + u_i$. That is, the total communication volume of T_i normalized to v_{tot} plus T_i's utilization. Finally, totLFFD sorts all tasks T_i in $\mathbf{T}_n$ according to decreasing values of total load and then performs an allocation based on the FF heuristic.

The following sections are concerned with comparing these two heuristics with FFD and the previously presented maxCFFD and totCFFD. FFD is together with BFD the best heuristic we have discussed so far to reduce the number of necessary processors. On the other hand, maxCFFD and totCFFD present better results than FFD in reducing communication between processors.

4.6.1 Processors versus Maximum Task Utilization

For generating curves in this section, we proceeded as in Chapter 3. A huge number of task sets were created for different numbers of tasks per set, where task utilizations were uniformly chosen between 0 and a variable maximum task utilization. Figures 4.28 to 4.31 compare the proposed heuristics against FFD, maxCFFD and totCFFD with respect to the number of processors they obtain. The proposed heuristics to reduce both the number of processors and the amount of communication perform better than maxCFFD and totCFFD, i.e., they obtain less processors. As expected, these heuristics are not as effective as FFD regarding the number of processors that they deliver for a small n. These new algorithms get, however, closer to FFD when n grows.

In Figure 4.32 to Figure 4.35, the average distance of algorithms to BestBP, i.e., the best bin packing, is shown. The proposed heuristics to reduce processors and communication result in a number of additional processors that is quite close to FFD and they outperform maxCFFD and totCFFD. For $n = 100$ and a maximum task utilization of 0.8, the three FFD, maxLFFD and totLFFD produce around 1 additional processor, whereas maxCFFD and totCFFD can only reduce the number of additional processors to approximately 4—see Figure 4.34. In the case of 500 tasks and also a maximum utilization of 0.8, maxLFFD and totLFFD still produce around 1 additional processor like FFD. However, maxCFFD and totCFFD come to 10 additional processors compared to BestBP (Figure 4.35).

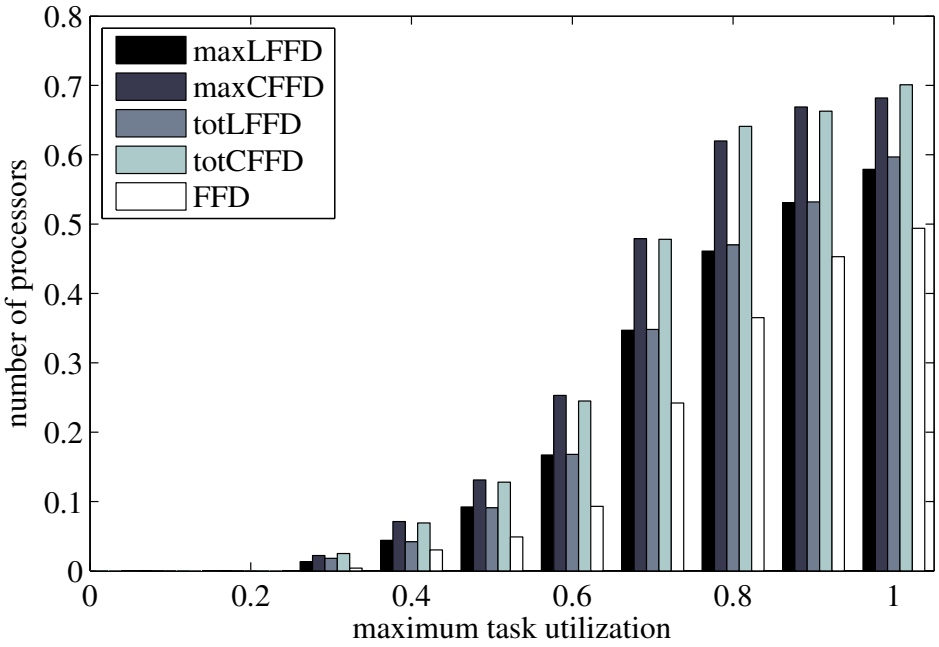


Figure 4.32: Heuristics to reduce processors and communication: average additional number of processors compared to BestBP vs. maximum task utilization, $n = 10$

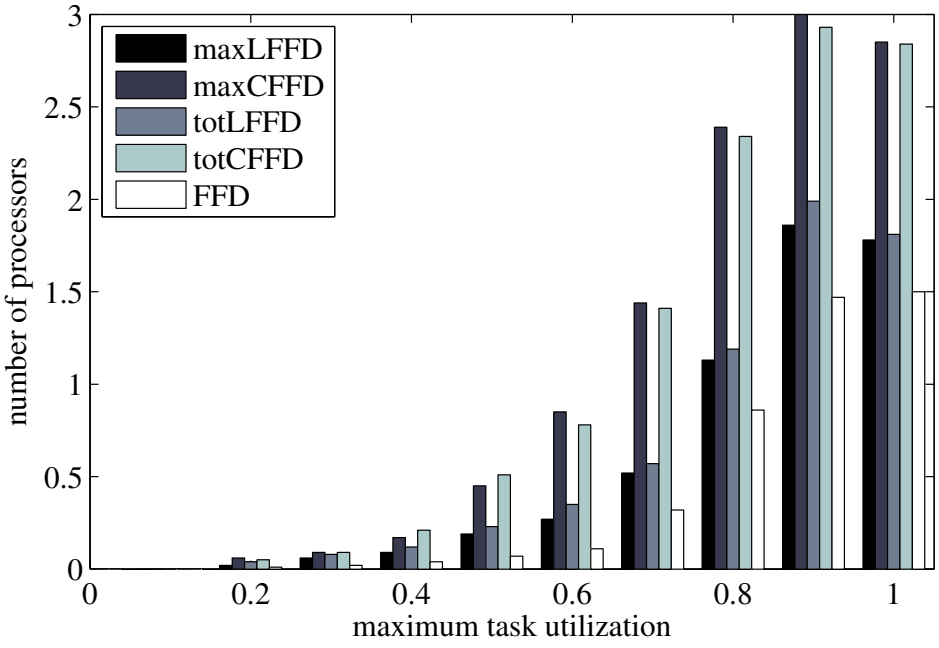


Figure 4.33: Heuristics to reduce processors and communication: average additional number of processors compared to BestBP vs. maximum task utilization, $n = 50$

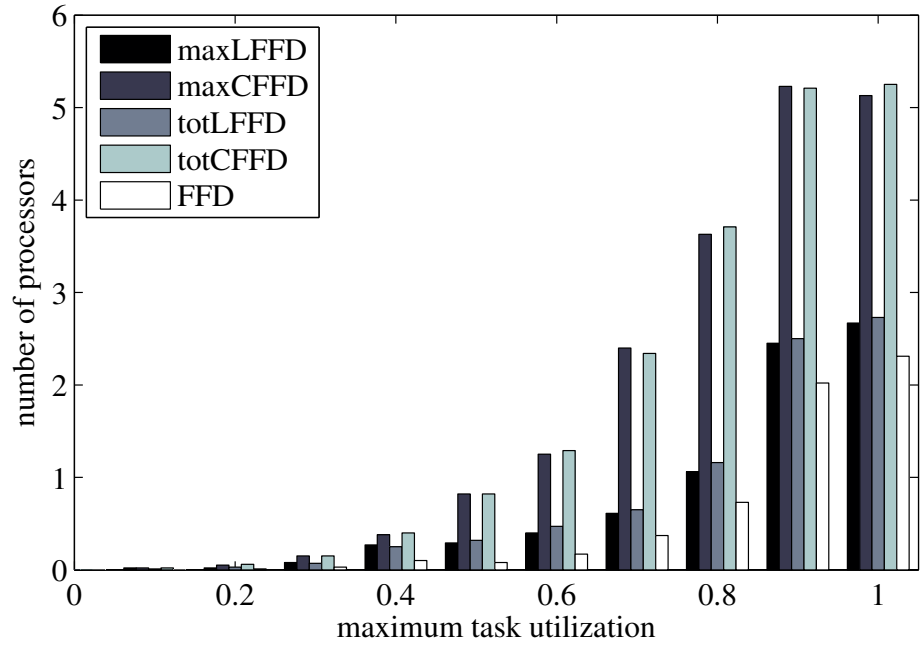


Figure 4.34: Heuristics to reduce processors and communication: average additional number of processors compared to BestBP vs. maximum task utilization, $n = 100$

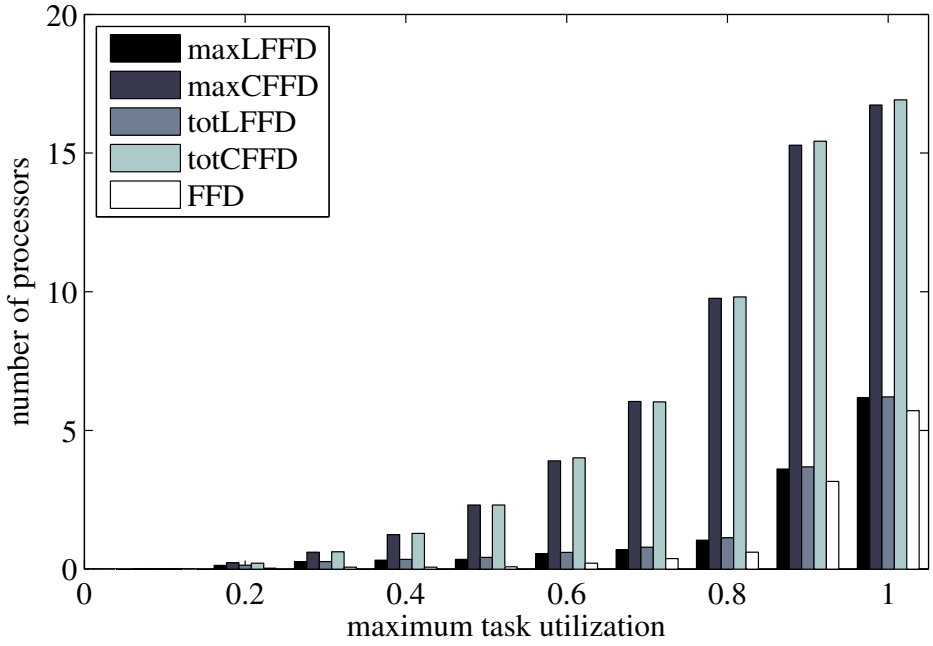


Figure 4.35: Heuristics to reduce processors and communication: average additional number of processors compared to BestBP vs. maximum task utilization, $n = 500$

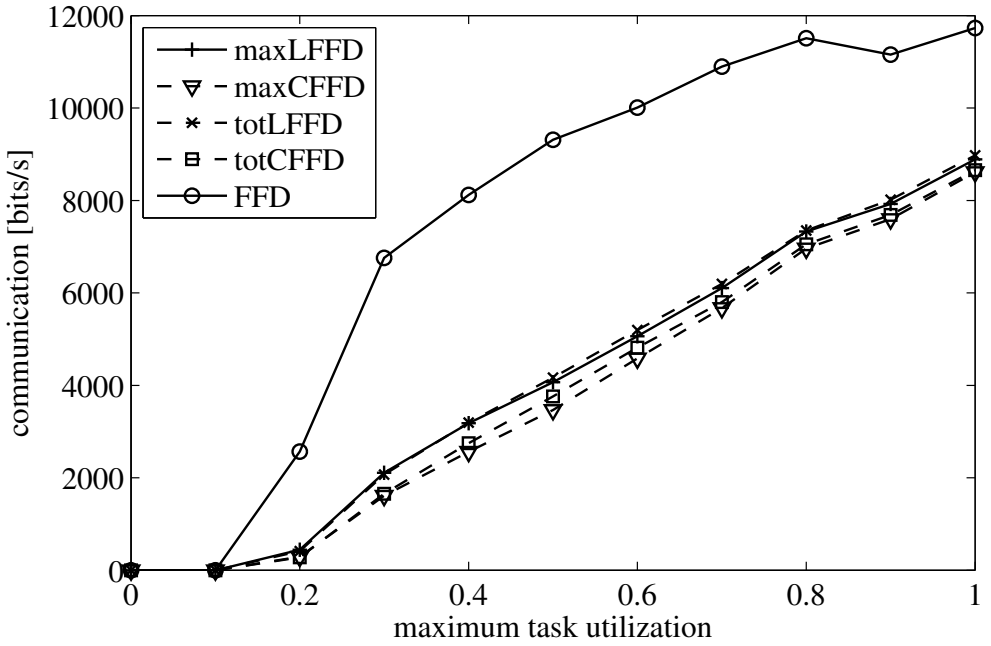


Figure 4.36: Heuristics to reduce processors and communication: average amount of communication vs. maximum task utilization, $n = 10$

4.6.2 Communication versus Maximum Task Utilization

In this section, the proposed heuristics maxLFFD and totLFFD are compared to FFD, maxCFFD and totCFFD concerning the amount of communication they lead to. For this purpose, a large number of task sets were generated exactly as for Section 4.5.4.

Communication matrices were randomly generated considering that a task can only communicate with only one task at a time (a loosely connected task system). Further, messages present the same period as the originating tasks. These periods were uniformly generated in the range of $(0, 1]$ seconds and remain the same for all task sets in each plot to make the effect of increasing the maximum task utilization more clear. The pairs of communicating tasks were also chosen randomly with a uniform distribution, for which random messages were created also in the range $[0, 1024]$ bits.

Figures 4.36 to 4.39 show the amount of communication in bits per second that results from the different heuristics. It can be seen that maxLFFD and totLFFD are not as efficient as maxCFFD and totCFFD to reduce communication. However, they still show a remarkable improvement over FFD. Both, maxLFFD and totLFFD behave approximately the same for the different numbers of tasks per task set considered. For 100 tasks in Figure 4.38, maxLFFD and totLFFD reach 140 Kbps for a maximum task utilization of 0.4. This is around 30 Kbps more than maxCFFD and approximately 60 Kbps less than FFD. In Figure 4.39, for $n = 500$, maxLFFD and totLFFD reach 1.4 Mbps for a maximum task utilization of also 0.4. In this case, they achieve around 300 Kbps more than maxCFFD but also approximately 700 Kbps less than FFD.

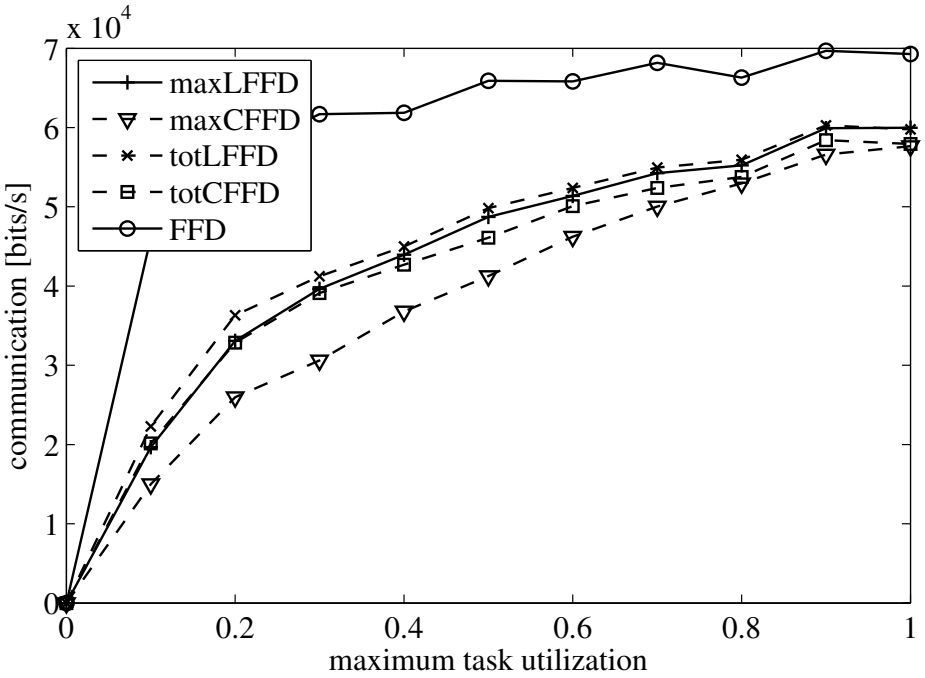


Figure 4.37: Heuristics to reduce processors and communication: average amount of communication vs. maximum task utilization, $n = 50$

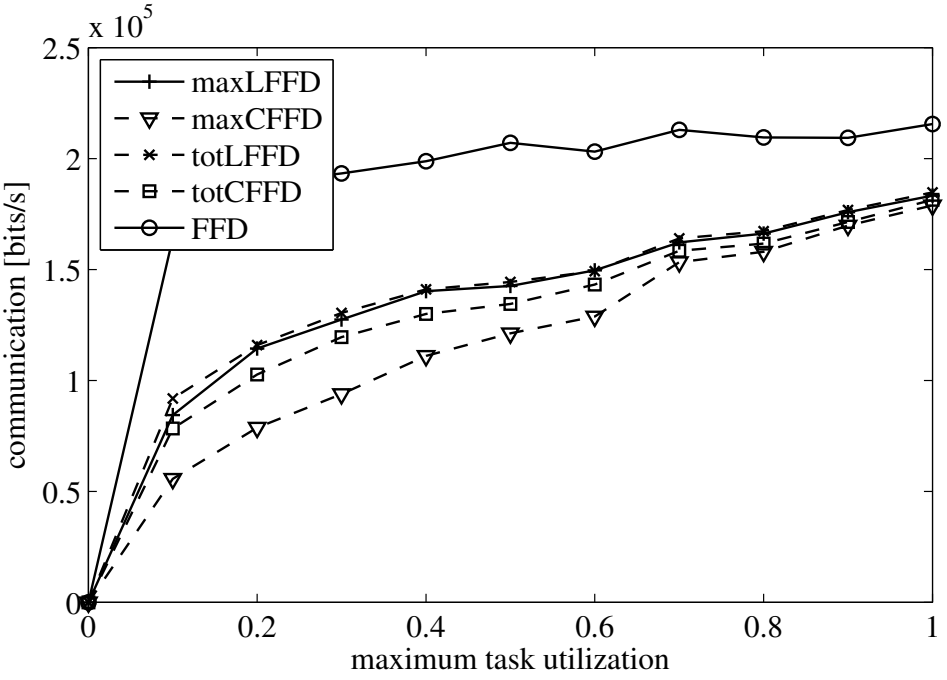


Figure 4.38: Heuristics to reduce processors and communication: average amount of communication vs. maximum task utilization, $n = 100$

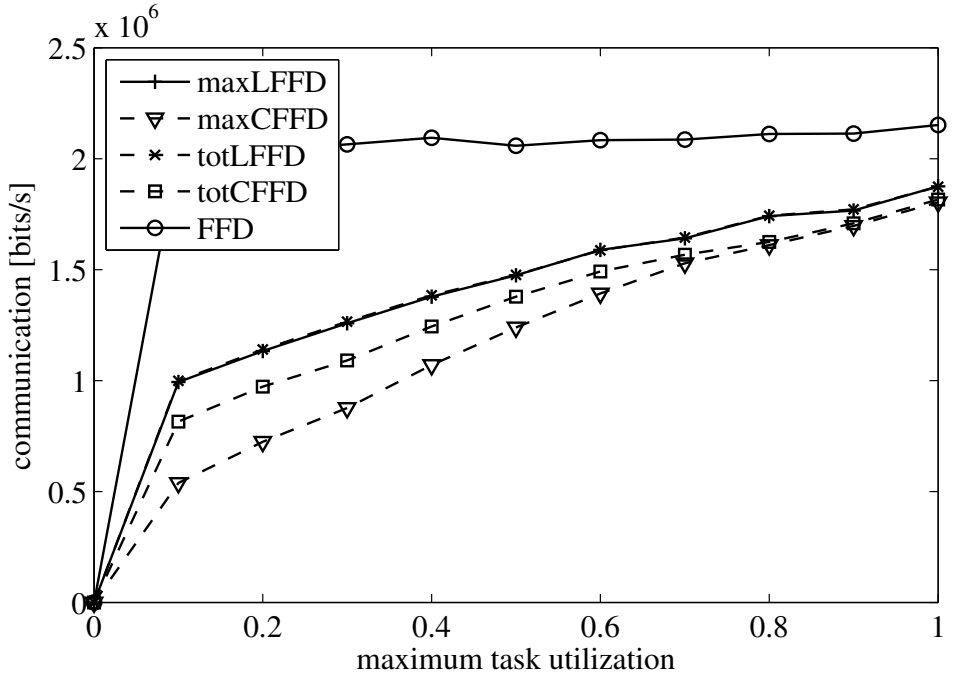


Figure 4.39: Heuristics to reduce processors and communication: average amount of communication vs. maximum task utilization, $n = 500$

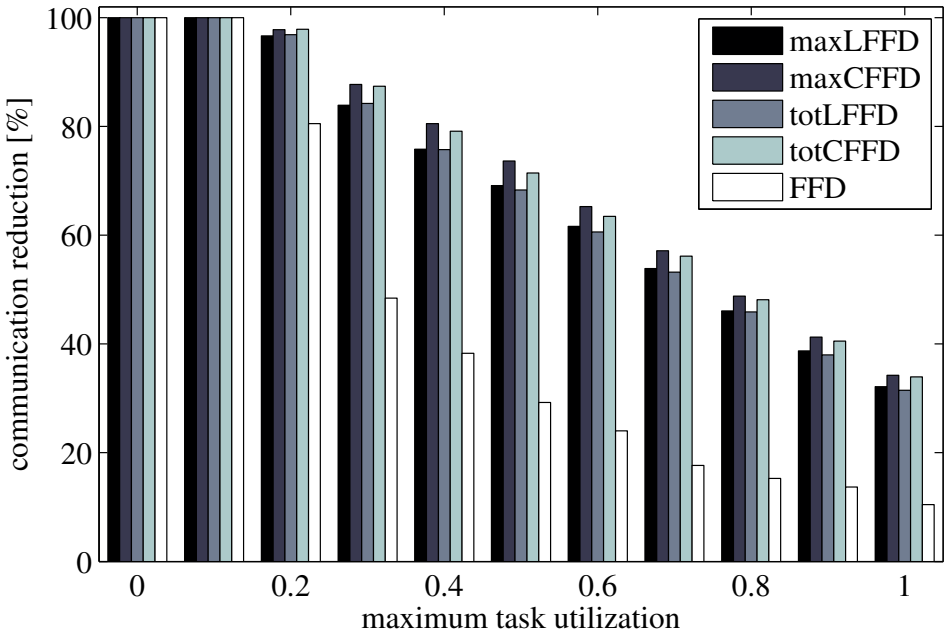


Figure 4.40: Heuristics to reduce processors and communication: average communication reduction compared to WorstCP vs. maximum task utilization $n = 10$

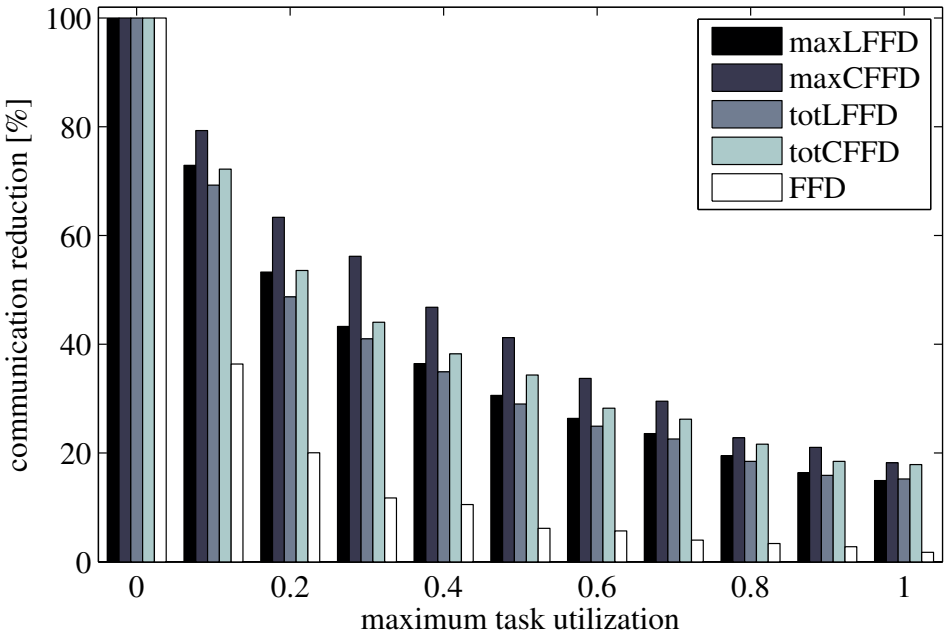


Figure 4.41: Heuristics to reduce processors and communication: average communication reduction compared to WorstCP vs. maximum task utilization $n = 50$

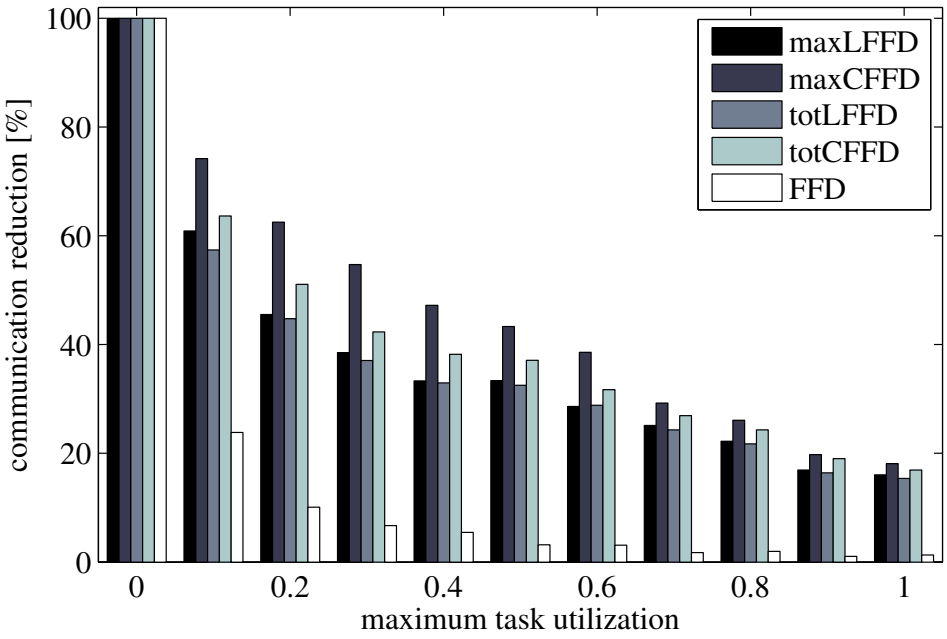


Figure 4.42: Heuristics to reduce processors and communication: average communication reduction compared to WorstCP vs. maximum task utilization $n = 100$

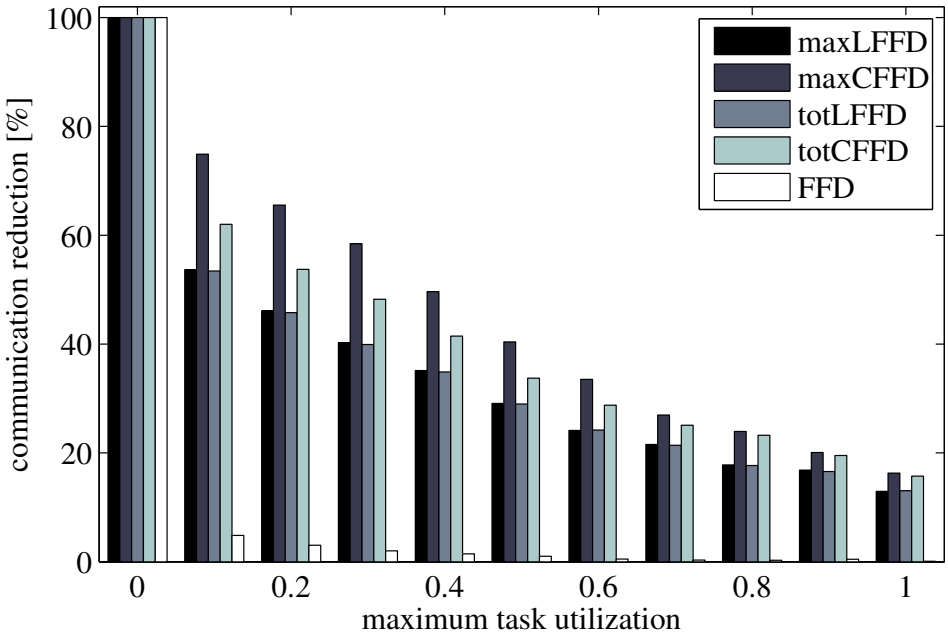


Figure 4.43: Heuristics to reduce processors and communication: average communication reduction compared to WorstCP vs. maximum task utilization $n = 500$

In Figure 4.40 to Figure 4.43, algorithms are compared with respect to WorstCP, i.e., the worst-case communication case. Comparing to WorstCP gives an idea of the communication reduction that algorithms are able to achieve. The length of bars represents how much communication was avoided. From Figure 4.40, it can be observed that maxLFFD and totLFFD result in only 5% less communication reduction than maxCFFD when the maximum task utilization is less than 0.5 and $n = 10$. However, this difference increases to $10 - 20\%$ less communication reduction for $n = 500$ in Figure 4.43.

4.6.3 Communication versus Maximum Task Connectivity

Figures 4.44 to 4.47 present the amount of communication that results as the maximum task connectivity is increased. The task connectivity stands for the number of tasks with which each task communicates. The maximum task connectivity will be increased from 0 to n tasks, i.e., we go from an unconnected to a fully connected task set.

A large number of task sets were created randomly for different numbers of tasks per task set. As in Section 4.5.5, the maximum task utilization within task sets was also fixed to 0.5. The maximum task connectivity was then increased from 0 to n in uniform steps of $\frac{n}{10}$ tasks each time.

Every time the maximum task connectivity was increased, communication matrices were generated for the different task sets as described in Section 4.5.5. The periods for the messages

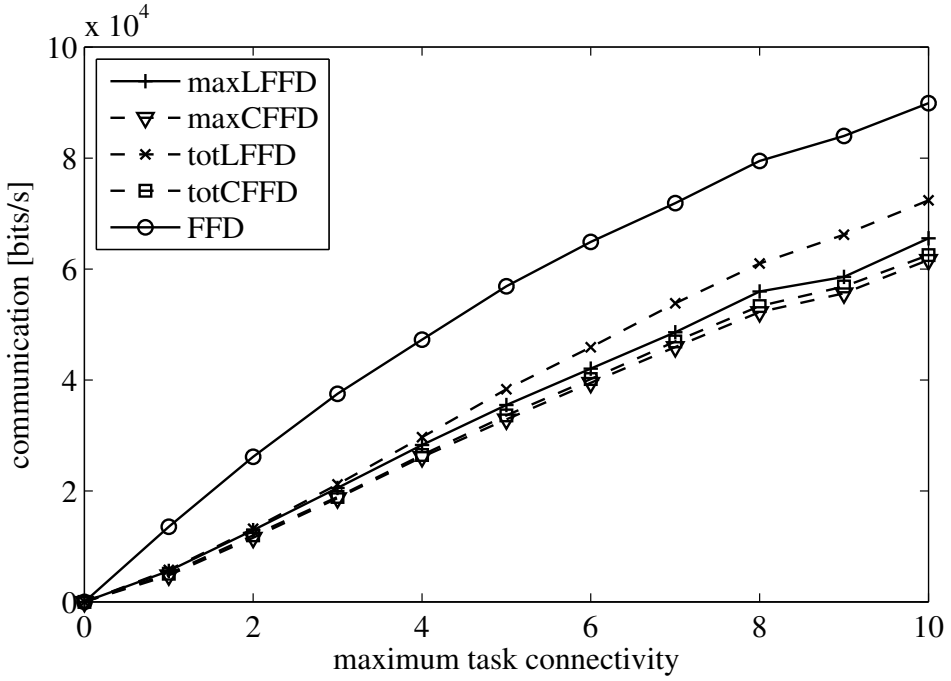


Figure 4.44: Heuristics to reduce processors and communication: average amount of communication vs. maximum task connectivity, $n = 10$

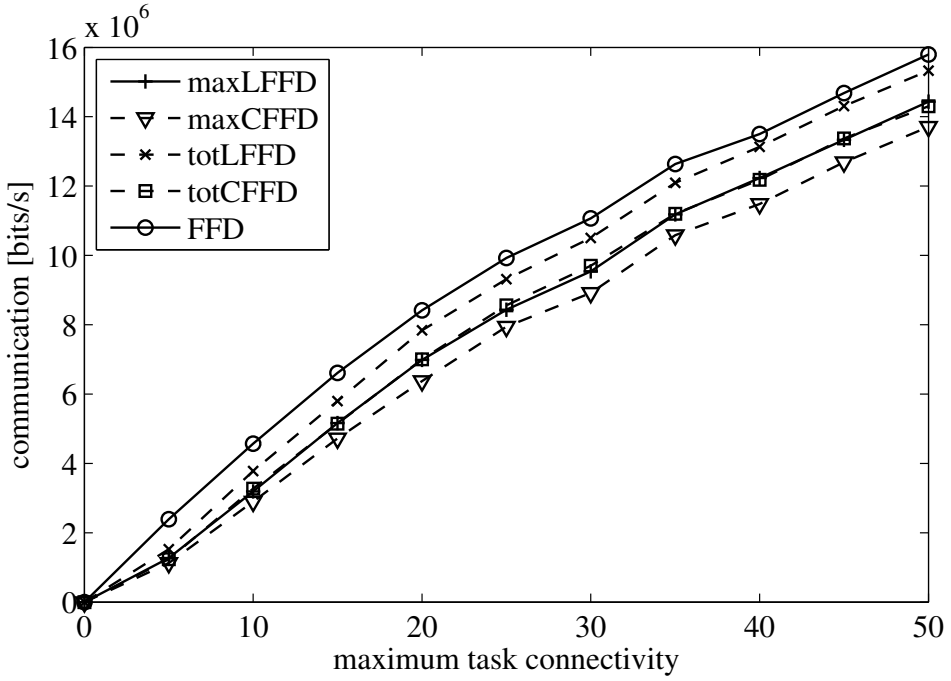


Figure 4.45: Heuristics to reduce processors and communication: average amount of communication vs. maximum task connectivity, $n = 50$

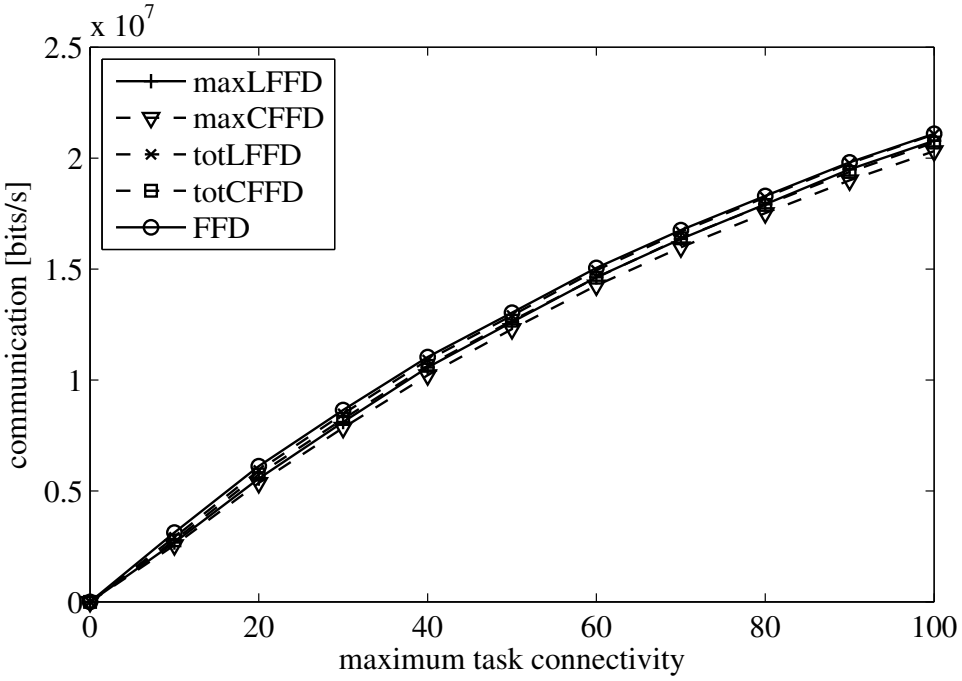


Figure 4.46: Heuristics to reduce processors and communication: average amount of communication vs. maximum task connectivity, $n = 100$

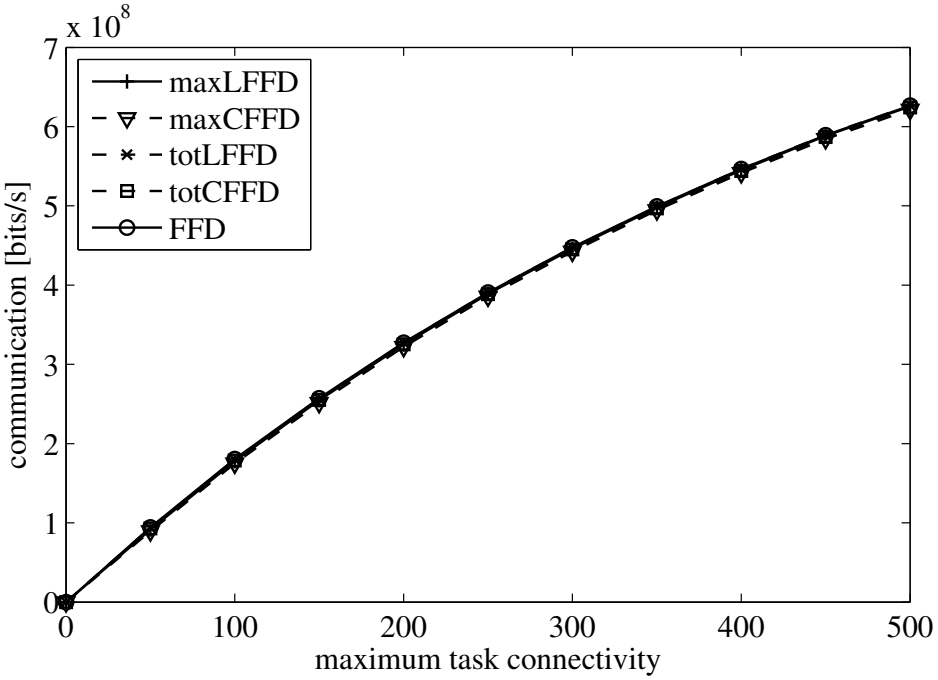


Figure 4.47: Heuristics to reduce processors and communication: average amount of communication vs. maximum task connectivity, $n = 500$

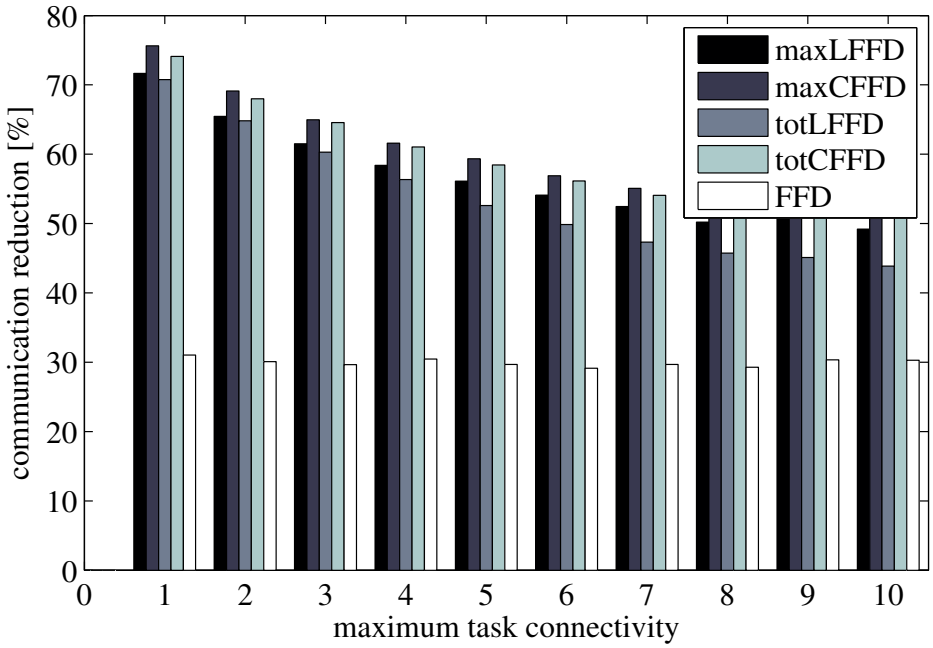


Figure 4.48: Heuristics to reduce processors and communication: average communication reduction compared to WorstCP vs. maximum task connectivity, $n = 10$

were uniformly distributed in the range of $(0, 1]$ seconds and generated once at the beginning of each plot, so that the effect of an increasing task connectivity can be emphasized. Further, for every pair of communicating tasks, random messages were created also uniformly in the range $[0, 1024]$ bits.

The resulting amount of communication for an increasing task connectivity is shown in Figure 4.44 to Figure 4.47. As expected, the performance of all algorithms becomes similar as the number of tasks increases. In Figure 4.44 for $n = 10$, maxLFFD and totLFFD show an interesting improvement over FFD, however, maxCFFD and totCFFD are still better. For $n = 10$, maxLFFD is slightly better than totLFFD, i.e., it results in less communication flow. For $n = 500$, the improvement of all new heuristics over FFD decreases drastically—see Figure 4.47. This behavior is partially due to the different step sizes with which the task connectivity is increased: 1 task for $n = 10$ and 50 tasks for $n = 500$.

Algorithms are compared to WorstCP under these conditions in Figure 4.48 to Figure 4.51. The heuristics maxLFFD and totLFFD produce around 55% communication reduction with respect to WorstCP when $n = 10$ and tasks are allowed to communicate with 5 of the other tasks in $\mathbf{T}_n$—see Figure 4.48. These two heuristics are outperformed in around 5% by maxCFFD. On the other hand, for $n = 500$ in Figure 4.51, maxLFFD and totLFFD arrive only to a communication reduction of 1.5% to 1% when tasks are connected to 250 of the remaining tasks. However, maxCFFD and totCFFD are not much better at this point. From Figure 4.48 to Figure 4.51, it can be observed that maxLFFD is slightly better than totLFFD in reducing the amount of communication between processors.

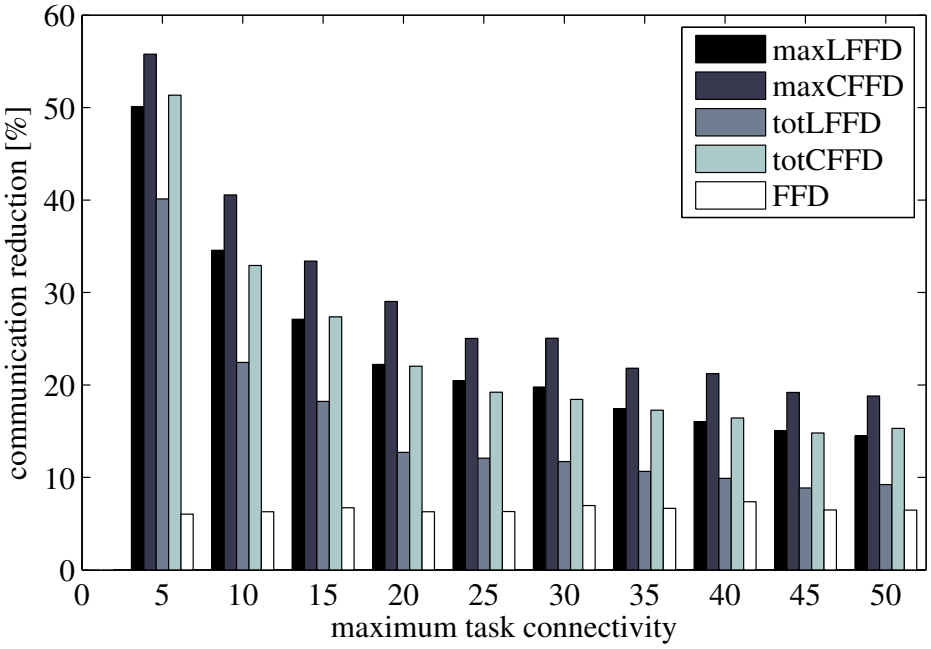


Figure 4.49: Heuristics to reduce processors and communication: average communication reduction compared to WorstCP vs. maximum task connectivity, $n = 50$

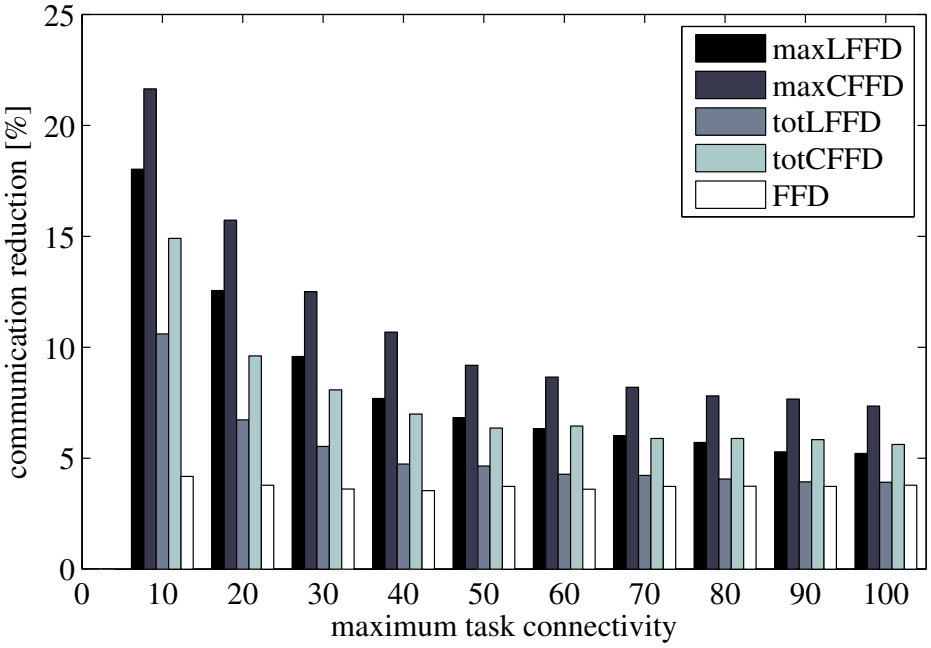


Figure 4.50: Heuristics to reduce processors and communication: average communication reduction compared to WorstCP vs. maximum task connectivity, $n = 100$

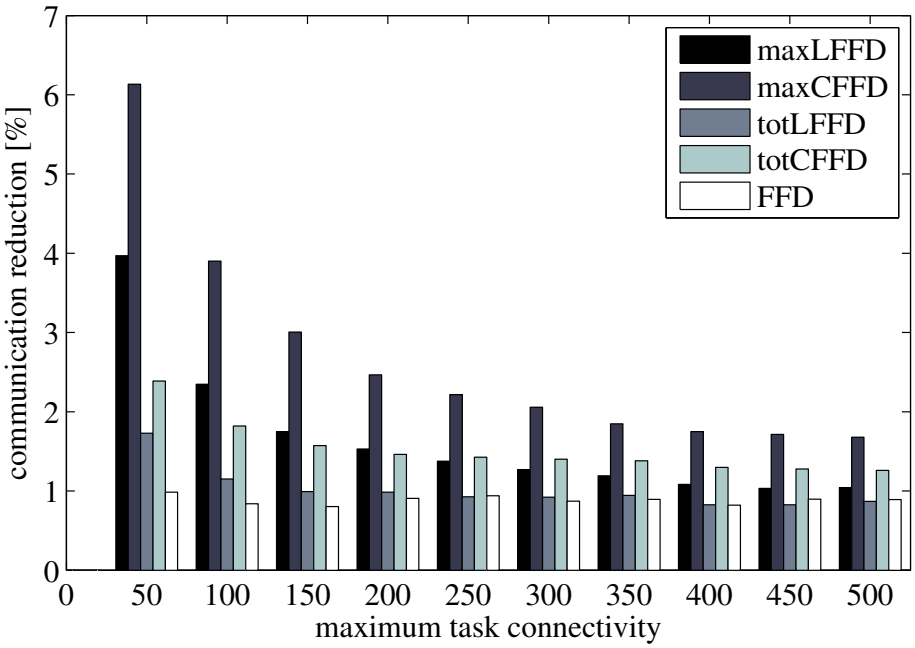


Figure 4.51: Heuristics to reduce processors and communication: average communication reduction compared to WorstCP vs. maximum task connectivity, $n = 500$

4.7 Key Findings

The problem of allocating dependent real-time tasks to processors was addressed. We focused on the allocation of communicating tasks to multiple processors, for which several possible heuristics were analyzed. In general, maxCFFD (Maximum Communication First Fit Decreasing) is the most suitable heuristic for minimizing the amount of communication between processors. This heuristic consists in sorting tasks according to their maximum communication volume. Concerning the minimization of both the number of processors and the amount of communication, maxLFFD (Maximum Load First Fit Decreasing) has been shown to be the more efficient than the other heuristics. The heuristic maxLFFD sorts tasks according to a parameter we called maximum load, which represents the load generated by tasks on the system. The maximum load parameter takes both the communication and the computation demand into account.

5 Concluding Remarks

In this thesis, we have analyzed the problem of allocating real-time tasks onto distributed processing units. For this purpose, we concentrated on allocation algorithms with polynomial complexity that are based on well-known techniques for bin packing like First Fit (FF) and First Fit Decreasing (FFD). In particular, we focused on the analysis of real-time tasks with arbitrary deadlines and considered also task communication and system constraints.

We have seen that an allocation algorithm for real-time tasks is composed of two interdependent parts: an allocation procedure and a feasibility test. Whilst the allocation procedure is the one deciding how tasks are going to be placed onto processors, the feasibility test is in charge of proving that no deadlines are missed when a new task is assigned to a processor.

In the literature, it is often assumed that deadlines are equal to periods so as to simplify the feasibility test when allocating real-time tasks to processors. In this context, there has been very little work with respect to the more general case of tasks with arbitrary deadlines. The problem is that, when considering arbitrary deadlines, the known methods for feasibility analysis have either a pseudo-polynomial complexity or a poor accuracy.

Clearly, the complexity of an allocation algorithm degrades if a feasibility test with higher complexity is used in combination with it. As we concentrate on allocation algorithms with polynomial complexity, we cannot make use of feasibility tests with pseudo-polynomial complexity. On the other hand, the use of simple polynomial-time feasibility tests (e.g., the density test under EDF [SSRB98, Liu00]) results in less efficient allocation algorithms. In order to retain the polynomial complexity of bin packing heuristics and, at the same time, to obtain more efficient allocations, we proposed a novel technique for the feasibility analysis of real-time tasks. This technique consists in calculating the maximum loading factor produced by tasks on a given processor and can be applied to both fixed-priority as well as dynamic-priority scheduling policies.

The loading factor is defined as the total execution demand of tasks in a specified time interval over the length of this interval. Further, the maximum loading factor is the upper bound for the loading factor. If the maximum loading factor is less than or equal to 1, the processor has always sufficient time to execute all real-time tasks within their deadlines and the tasks are said to be feasible or schedulable on that processor.

This way, two new feasibility tests were proposed for the EDF scheduling, which we called respectively EDFTest1 and EDFTest2. Furthermore, three additional tests were also introduced for tasks scheduled under the DM/RM policy: DM/RMTest1 to DM/RMTest3. All these tests are based on calculating the maximum loading factor produced by tasks on a given processor. The test DM/RMTest1 is the analogous for DM/RM to the density test for EDF [SSRB98,

Liu00]. DM/RMTest1 is concerned with the computation of the maximum loading factor of individual tasks. The tests DM/RMTest2 and EDFTest1 compute the maximum loading factor for a group of two tasks achieving this way a better accuracy. A further accuracy improvement can be obtained if the maximum loading factor is calculated for several tasks. This latter is what DM/RMTest3 and EDFTest2 deal with.

These proposed feasibility tests were combined with some of the well-known bin packing heuristics in order to come up with polynomial-time allocation algorithms for real-time tasks with arbitrary deadlines. The heuristic EDFTest2FFD, i.e., the combination of the mentioned EDFTest2 with the FFD heuristic, has shown to be better at reducing the number of processors under an EDF scheduling. In the same way, DM/RMTest2FFD is the combination of DM/RMTest2 and FFD and resulted to be the best heuristic for allocating tasks under the DM/RM policy.

On the other hand, we have also seen that simple bin packing heuristic perform very well on the average for the case under EDF of independent tasks with deadlines equal to periods (i.e., for the case where the task allocation reduces to the bin packing problem). However, as already stated, very little work have been done so far on extending these methods to other more interesting allocation scenarios, where, for example, tasks present communication between them. On the contrary, researchers have rather focused on other approaches to solve the allocation problem in more complex contexts like the one mentioned. In this thesis, we intended to bridge this gap by proposing new heuristics based upon the known bin packing heuristics and that consider task communication and system constraints. The main advantage of the proposed heuristics over the methods from the literature is that they all have polynomial complexity.

Four new heuristics were presented to reduce the amount of communication when allocating tasks to processors: maxCFFD, maxCBFD, totCFFD and totCBFD. As the known FFD, these heuristics are based on sorting tasks according to a given criterion and then performing a sequential assignment to processors one task after the other. The criteria to sort tasks are concerned with the communication characteristics among them. This way, maxCFFD and maxCBFD sort tasks according to the maximum communication volume they exchange with other tasks, whereas totCFFD and totCBFD arrange tasks according to their total communication volume. The maximum communication volume of a task was defined to be the maximum message exchange (i.e., incoming and outgoing messages) with any other task. On the other hand, a task's total communication volume is the sum of all incoming and outgoing messages associated with it. All four proposed heuristics were shown by means of a thorough statistical comparison to achieve a substantial reduction of the communication among processors. However, maxCFFD leads to the maximum communication reduction among all proposed heuristics.

The heuristics to reduce communication perform similar to FF in minimizing the number of processors. This is because they do not sort tasks according to their utilizations as, for example, the better FFD does (the task utilization is the analogous to the item size in the bin packing problem). However, we know from [CGJ97] that sorting task according to their utilizations results in a better allocation in the sense that less processors will be necessary. As a consequence, two further heuristics were presented to reduce both the number of resulting processors and the amount of communication between them: maxLFFD and totLFFD. These heuristics

also sort task according to given criteria. This time, the sorting criteria depend not only on the communication characteristics of tasks, but also on their respective utilizations. As shown by means an extensive comparison, these two other heuristics achieve fairly good allocations in what respects to both the reduction of communication and processors, whereupon maxLFFD is slightly better than totLFFD.

Finally, the proposed feasibility tests are general enough and can of course also be used in other contexts. Particularly, because the proposed tests present all linear complexity $\mathcal{O}(n)$, they are suitable for performing an (on-line) admission control. In this case, whether a new task can be accepted on a system of already running tasks can be determined more accurately in constant time $\mathcal{O}(1)$ by the new algorithms.

In addition, all presented heuristics were conceived to be used as off-line allocation algorithms. This is because they sort tasks according to given criteria and, consequently, they need to know all tasks in advance. However, if tasks are sorted as they arrive it may also be possible to use these communication-aware heuristics to performs an on-line global scheduling upon multiprocessors. In this case, the task sorting might have to be restricted so as to be performed more efficiently in on-line operation.

Bibliography

[ABJ01] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS)*, pages 193–202, December 2001. 7

[ABR+93] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993. 5, 6

[ABRW91] N. Audsley, A. Burns, M. Richardson, and A. Wellings. Hard real-time scheduling: The deadline-monotonic approach. *Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems and Software*, pages 133–137, May 1991. 5

[ABS06] K. Albers, F. Bodmann, and F. Slomka. Hierarchical event streams and event dependency graphs: A new computational model for embedded real-time systems. *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 97–106, July 2006. 8

[ACG+03] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer Verlag, Berlin, Germany, 2003. 47

[AD96] I. Ahmad and M. Dhodhi. Multiprocessor scheduling in a genetic paradigm. *Parallel Computing*, 22(3):395–406, 1996. 7

[AKS+02] S. Ali, J.-K. Kim, H. Siegel, A. Maciejewski, Y. Yu, S. Gundala, S. Gertphol, and V. Prasanna. Greedy heuristics for resource allocation in dynamic distributed real-time heterogeneous computing systems. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 519–530, June 2002. 7

[AS04] K. Albers and F. Slomka. An event stream driven approximation for the analysis of real-time systems. *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS)*, June-July 2004. 4, 6, 32

[AS05] K. Albers and F. Slomka. Efficient feasibility analysis for real-time systems with edf scheduling. *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, March 2005. 4

[Aud91] N. Audsley. Optimal priority assignment and feasibility of static priority tasks with

arbitrary start times. *Technical Report YCS 164, University of York, Deparment of Computer Science*, 1991. 6

[Bak03] T. Baker. Multiprocessor edf and deadline monotonic schedulability analysis. *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS)*, pages 120–129, 2003. 7

[Bak05] T. Baker. An analysis of edf schedulability on a multiprocessor. *IEEE Transactions on Parallel Distributed Systems*, 16(8):760–768, 2005. 7

[Bak06] T. Baker. An analysis of fixed-priority schedulability on a multiprocessor. *Real-Time Systems*, 32(1-2):49–71, 2006. 7

[Bar98] S. Baruah. A general model for recurring real-time tasks. *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, pages 114–122, December 1998. 8

[Bar03] S. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems*, 24(1):93–128, 2003. 8

[Bar04] S. Baruah. Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. *IEEE Transaction on Computers*, 53(6):781–784, 2004. 7

[BB03] E. Bini and G. Buttazzo. Rate monotonic analysis: the hyperbolic bound. *IEEE Transactions on Computers*, 52(7):933–942, July 2003. 5

[BB04a] E. Bini and G. Buttazzo. Biasing effects in schedulability measures. *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS)*, June-July 2004. 26, 40

[BB04b] E. Bini and G. Buttazzo. Schedulability analysis of periodic fixed priority systems. *IEEE Transactions on Computers*, 53(11):1462–1473, November 2004. 5

[BB05] E. Bini and G. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005. 26, 40

[BB08] S. Baruah and T. Baker. Schedulability analysis of global edf. *Real-Time Systems*, 38(3):223–235, 2008. 7

[BBB01] E. Bini, G.C. Buttazzo, and G.M. Buttazzo. A hyperbolic bound for the rate monotonic algorithm. *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS)*, June 2001. 5, 29

[BCGM99] S. Baruah, D. Chen, S. Gorinsky, and A. Mok. Generalized multiframe tasks. *Real-Time Systems*, 17(1):5–22, 1999. 8

[BCL05] M. Bertogna, M. Cirinei, and G. Lipari. Improved schedulability analysis of edf on multiprocessor platforms. *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 209–218, July 2005. 7

[BF05] S. Baruah and N. Fisher. The partitioned multiprocessor scheduling of sporadic

task systems. *Proceedings of the 26th Real-Time Systems Symposium (RTSS)*, pages 321–329, December 2005. 53

[BF06a] S. Baruah and N. Fisher. The feasibility analysis of multiprocessor real-time systems. *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 85–96, July 2006. 7

[BF06b] S. Baruah and N. Fisher. The partitioned multiprocessor scheduling of deadline-constrained sporadic task systems. *IEEE Transactions on Computers*, 55(7):918–923, 2006. 53

[BF07] S. Baruah and N. Fisher. The partitioned dynamic-priority scheduling of sporadic task systems. *Real-Time Systems*, 36(3):199–226, 2007. 53

[BHR93] S. Baruah, R. Howell, and L. Rosier. Feasibility problems for recurring tasks on one processor. *Theoretical Computer Science*, 118(1):3–20, 1993. 3

[BLOS95] A. Burchard, J. Liebeherr, Y. Oh, and S.H. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Transactions on Computers*, 44(12):1429–1442, December 1995. 5, 29, 30, 52

[BMR90] S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. *Proceedings of the 11th IEEE Real-Time Systems Symposium (RTSS)*, pages 182–190, December 1990. 3, 16, 17, 18, 26, 43

[BNTZ94] A. Burns, N. Nicholson, K. Tindell, and N. Zhang. Allocating and scheduling hard real-time tasks on a parallel processing platform. *Technical Report YCS 238, University of York, Deparment of Computer Science*, 1994. 7

[BRH90] S. Baruah, L. Rosier, and R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic real-time tasks on one processor. *Real-Time Systems*, 2(4):301–324, November 1990. 3

[But05] G. Buttazzo. Rate monotonic vs. edf: Judgment day. *Real-Time Systems*, 29(1):5–26, January 2005. 3, 16

[CGJ97] E. Coffman, M. Garey, and D. Johnson. Approximation algorithms for bin packing: A survey. *Approximation Algorithms for NP-Hard Problems by D. Hochbaum*, pages 46–93, March 1997. Book Chapter. 7, 48, 49, 93, 110

[Cha03] S. Chakraborty. *System-Level Timing Analysis and Scheduling for Embedded Packet Processors*. Eidgenössische Technische Hochschule (ETH) Zürich, Zurich, Switzerland, 2003. Ph.D. Thesis. 4

[CKT02] S. Chakraborty, S. Künzli, and L. Thiele. Approximate schedulability analysis. *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, pages 159–168, December 2002. 4, 42

[Der74] M. Dertouzos. Control robotics: The procedural control of physical processes. *Information Processing*, 74:807–813, 1974. 3, 6, 15

[Dev03] M. Devi. An improved schedulability test for uniprocessor periodic task systems. *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 23–30, July 2003. 4, 16, 17, 18, 19, 53

[DL78] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978. 2, 7, 52

[FB05a] N. Fisher and S. Baruah. A fully polynomial-time approximation scheme for feasibility analysis in static-priority systems with arbitrary relative deadlines. *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 117–126, July 2005. 6, 32

[FB05b] N. Fisher and S. Baruah. A polynomial-time approximation scheme for feasibility analysis in static-priority systems with bounded relative deadlines. *Proceedings of the 13th International Conference on Real-Time Systems*, April 2005. 6

[FB05c] S. Funk and S. Baruah. Task assignment on uniform heterogeneous multiprocessors. *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 219–226, July 2005. 7

[FB06] N. Fisher and S. Baruah. A fully polynomial-time approximation scheme for feasibility analysis in static-priority systems with bounded relative deadlines. *Journal of Embedded Computing*, 2(3,4):291–299, 2006. 6

[FGB01] S. Funk, J. Goossens, and S. Baruah. On-line scheduling on uniform multiprocessors. *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS)*, pages 183–192, December 2001. 7

[GFB03] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2-3):187–205, 2003. 7

[GJ79] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., San Francisco, USA, 1979. 1, 7, 47, 48

[Goo03] J. Goossens. Scheduling of offset free systems. *Real-Time Systems*, 24(2):239–258, 2003. 5, 6

[Gre93] K. Gresser. *Echtzeitnachweis ereignisgesteuerter Realzeitsysteme*. VDI-Verlag, Düsseldorf, Germany, 1993. Ph.D. Thesis (in German). 8

[GRS96] L. George, N. Rivierre, and M. Spuri. Preemptive and non-preemptive real-time uniprocessor scheduling. *Rapport de Recherche RR-2966, INRIA*, 1996. 4, 16, 17

[HL88] K. Hong and J. Leung. On-line scheduling of real-time tasks. *Proceedings of the 9th IEEE Real-Time Systems Symposium (RTSS)*, pages 244–250, December 1988. 6

[HL92] K. Hong and J. Leung. On-line scheduling of real-time tasks. *IEEE Transactions on Computers*, 41(10):1326–1331, October 1992. 6

[Joh73] D. Johnson. *Near-Optimal Bin Packing Algorithms*. Massachusetts Institute of Technology (MIT), Department of Mathematics, Cambridge, USA, 1973. Ph.D. Thesis. 49

[KM91] T.-W. Kuo and A. Mok. Load adjustment in adaptive real-time systems. *Proceedings of the 12th IEEE Real-Time Systems Symposium (RTSS)*, pages 160–170, December 1991. 5, 29

[Kop98] H. Kopetz. The time-triggered model of computation. *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, December 1998. 11

[Lat94] S. Latifi. Task allocation in the star graph. *IEEE Transactions on Parallel Distributed Systems*, 5(11):1220–1224, 1994. 7

[LDG01] J. López, J. Díaz, and D. García. Minimum and maximum utilization bounds for multiprocessor rm scheduling. *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS)*, June 2001. 7

[LDG04] J. López, J. Díaz, and D. García. Minimum and maximum utilization bounds for multiprocessor rate monotonic scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 15(7):642–653, 2004. 2, 7

[Leh90] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. *Proceedings of the 11th IEEE Real-Time Systems Symposium (RTSS)*, pages 201–209, December 1990. 5, 6, 28, 29, 30, 31, 32, 40, 44

[LGDG00] J. López, M. García, J. Díaz, and D. García. Worst-case utilization bound for edf scheduling on real-time multiprocessor systems. *Proceedings of the 12th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 25–33, June 2000. 7

[Liu00] J. Liu. *Real-Time Systems*. Prentice Hall, New Jersey, USA, 2000. 4, 5, 16, 17, 29, 42, 52, 109, 110

[LL73] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in hard real-time environments. *Journal of the Association for Computing Machinery*, 20(1):40–61, 1973. 3, 5, 6, 8, 9, 17, 29, 48, 73

[LS86] J. Lehoczky and L. Sha. Performance of real-time bus scheduling algorithms. *Proceedings of the ACM SIGMETRICS joint international conference on computer performance modeling, measurement and evaluation*, pages 44–53, May 1986. 29, 30

[LSD89] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. *Proceedings of the 10th IEEE Real-Time Systems Symposium (RTSS)*, pages 166–171, December 1989. 5, 6

[Lun02] L. Lundberg. Analyzing fixed-priority global multiprocessor scheduling. *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–153, September 2002. 7

[LW82] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982. 5, 16

[Mas07] A. Masrur. Feasibility conditions for edf. *Technical Report, Technische Universität München (TUM), Institute for Real-Time Computer Systems*, September 2007. 42

[MC96] A. Mok and D. Chen. A multiframe model for real-time tasks. *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS)*, pages 22–29, December 1996. 8

[MC97] A. Mok and D. Chen. A multiframe model for real-time tasks. *IEEE Transaction on Software Engineering*, 23(10):635–645, October 1997. 8

[MCF10a] A. Masrur, S. Chakraborty, and G. Färber. Constant-time admission control for deadline monotonic tasks. *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, March 2010. 31

[MCF10b] A. Masrur, S. Chakraborty, and G. Färber. Constant-time admission control for partitioned edf. *Proceedings of 22nd Euromicro Conference on Real-Time Systems (ECRTS)*, July 2010. 26, 28

[MDF08] A. Masrur, S Drössler, and G. Färber. Improvements in polynomial-time feasibility testing for edf. *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, March 2008. 4, 18

[MF06a] A. Masrur and G. Färber. Ideas to improve the performance in feasibility testing for edf. *Proceedings of the WiP Session of the 18th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2006. 42

[MF06b] A. Masrur and G. Färber. An off-line variable-size bin packing for edf. *Technical Report, Technische Universität München (TUM), Institute for Real-Time Computer Systems*, July 2006. 53

[MFHS05] A. Metzner, M. Fränzle, C. Herde, and I. Stierand. Scheduling distributed real-time systems by satisfiability checking. *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 409–415, August 2005. 8

[MFHS06] A. Metzner, M. Fränzle, C. Herde, and I. Stierand. An optimal approach to the task allocation problem on hierarchical architectures. *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS)*, April 2006. 8

[MH06] A. Metzner and C. Herde. Rtsat– an optimal and efficient approach to the task allocation problem in distributed architectures. *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS)*, pages 147–158, December 2006. 8

[OB98] D.-I. Oh and T. Bakker. Utilization bounds for n-processor rate monotone schedul-

ing with static processor assignment. *Real-Time Systems*, 15(2):183–192, 1998. 7

[OS95] Y. Oh and S.H. Son. Allocating fixed-priority periodic tasks on multiprocessor systems. *Real-Time Systems*, 9(3):207–239, 1995. 5, 52

[PGH98] J Palencia and M. González Harbour. Schedulability analysis for tasks with static and dynamic offsets. *In Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, pages 26–37, December 1998. 6

[PL04] R. Pellizzoni and G. Lipari. A new sufficient feasibility test for asynchronous real-time periodic task sets. *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 204–211, June-July 2004. 5

[PL05] R. Pellizzoni and G. Lipari. Feasibility analysis of real-time periodic tasks with offsets. *Real-Time Systems.*, 30(1-2):105–128, 2005. 5

[PL07] R. Pellizzoni and G. Lipari. Holistic analysis of asynchronous real-time transactions with earliest deadline scheduling. *Journal of Computer and System Sciences*, 73(2):186–206, 2007. 5

[PSTW97] C. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC)*, pages 140–149, May 1997. 7

[RGB+08] M. Ruggiero, A. Guerri, D. Bertozzi, M. Milano, and L. Benini. A fast and accurate technique for mapping parallel applications on stream-oriented mpsoc platforms with communication awareness. *International Journal on Parallel Programming*, 36(1):3–36, 2008. 8

[RM96] A. Ripoll, I. Crespo and A. Mok. Improvement in feasibility testing for real-time tasks. *Real-Time Systems*, 11(1):19–39, 1996. 4

[SAr+04] L. Sha, T. Abdelzaher, K. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, J. Caccamo, C. Lehoczky, and A. Mok. Real- real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2-3):101–155, November 2004. 3, 6

[Spu95] M. Spuri. *Earliest Deadline Scheduling in Real-Time Systems*. Scuola Superiore Sant'Anna, Pisa, Italy, 1995. Ph.D. Thesis. 4

[Spu96] M. Spuri. Analysis of deadline scheduled real-time systems. *Rapport de Recherche RR-2772, INRIA*, 1996. 4

[SSRB98] J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo. *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. Kluwer, Dordrecht, The Netherlands, 1998. 4, 16, 17, 20, 34, 109, 110

[ST85] C.-C. Shen and W.-H. Tsai. A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion. *IEEE Transactions on Computers*, C-34(3):197–203, March 1985. 7

[SVC98] S. Sáez, J. Vila, and A. Crespo. Using exact feasibility tests for allocating real-time tasks in multiprocessor systems. *Proceedings of the 10th Euromicro Workshop on Real-Time Systems*, pages 53–60, June 1998. 2

[TBW92] K. Tindell, A. Burns, and A. Wellings. Allocating hard real time tasks + (an np-hard problem made easy). *Real Time Systems*, 4(2):145–165, 1992. 7

[Thi00] H. Thielen. *Optimierte Auslegung verteilter Realzeitsysteme*. Technische Universität München (TUM), Intitute for Real-Time Computer Systems, München, Germany, 2000. Ph.D. Thesis (in German). 7, 47

[Tin90] K. Tindell. Allocating hard real time tasks + (an np-hard problem made easy). *Technical Report YCS 149, University of York, Deparment of Computer Science*, 1990. 7

[Tin94] K. Tindell. Adding time-offsets to schedulability analysis. *Technical Report YCS 221, University of York, Deparment of Computer Science*, 1994. 6

[WEE+08] R. Wilhelm, J. Engblom, A. Ermedahl, S. Holsti, N.and Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, 2008. 8

[WYJ+04] A. Wu, H. Yu, S. Jin, K.-C. Lin, and G. Schiavone. An incremental genetic algorithm approach to multiprocessor scheduling. *IEEE Transactions on Parallel Distributed Systems*, 15(9):824–834, 2004. 7

[ZS94] Q. Zheng and K. Shin. On the ability of establishing real-time channels in point-to-point packet-switched networks. *IEEE Transactions on Communications*, 42(2-3-4), 1994. 4