Fixed-Priority Scheduling Algorithms with Multiple Objectives in Hard Real-Time Systems

Armando Aguilar-Soto

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science.

The University of York
Department of Computer Science

2006

Para Mama, Papa, Paty, Tere, Gabriel ...

... y todas las personas que amo!

Abstract

In the context of Fixed-Priority Scheduling in Real-Time Systems, we investigate scheduling mechanisms for supporting systems where, in addition to timing constraints, their performance with respect to additional QoS requirements must be improved. This type of situation may occur when the worst-case resource requirements of all or some running tasks cannot be simultaneously met due to task contention.

Solutions to these problems have been proposed in the context of both fixed-priority and dynamic-priority scheduling. In fixed-priority scheduling, the typical approach is to artificially modify the attributes or structure of tasks, and/or usually require non-standard run-time support. In dynamic-priority scheduling approaches, utility functions are employed to make scheduling decisions with the objective of maximising the utility. The main difficulties with these approaches are the inability to formulate and model appropriately utility functions for each task, and the inability to guarantee hard deadlines without executing computationally costly algorithms.

In this thesis we propose a different approach. Firstly, we introduce the concept of relative importance among tasks as a new metric for expressing QoS requirements. The meaning of this importance relationship is to express that in a schedule it is desirable to run a task in preference to other ones. This model is more intuitive and less restrictive than traditional utility-based approaches. Secondly, we formulate a scheduling problem in terms of finding a feasible assignment of fixed priorities that maximises the new QoS metric, and propose the DI and DI+ algorithms that find optimal solutions.

By extensive simulation, we show that the new QoS metric combined with the DI algorithm outperforms the rate monotonic priority algorithm in several practical problems such as minimising jitter, minimising the number of preemptions or minimising the latency. In addition, our approach outperforms EDF in several scenarios.

Acknowledgements

- The most gratitude to Dr Guillem Bernat for supervising my research and for his invaluable support, friendship and encouragement.
- Special thanks to Professor Andy Wellings for his valuable opinion and suggestions.
- Kind appreciation to Professor Alan Burns for his support, and all members of the Real-Time Systems Group at the University of York for their friendship and feedback.
- Thanks to my sponsorship, the National Council of Science and Technology of Mexico (CONACYT).

Author Declaration

- The research presented in this thesis was undertaken from October 2002 to September 2006. Unless otherwise is indicated, all ideas and text are the author's own.
- Chapter 5 and parts of chapter 4 and 7 were published in the paper: A. Aguilar-Soto and G. Bernat, *Bicriteria Fixed-Priority Scheduling in Hard Real-Time Systems: Deadline and Importance*, 4th International Conference on Real-Time and Networks Systems RTNS 2006.

Contents

Abstract i

1 Introduction 1

1.1 Real-Time Systems . . . 2

1.2 Motivation: Problems with Deadlines and QoS . . . 3

1.2.1 Example: Problem with Deadlines and Output Jitter . . . 4

1.2.2 Discussion . . . 8

1.3 Thesis Aims . . . 10

1.3.1 Contributions . . . 11

1.3.2 Research Approach . . . 11

1.3.3 Thesis Organisation . . . 13

2 A Review On Preemptive Fixed Priority Scheduling 16

2.1 Real-time Systems . . . 16

2.1.1 Process Model . . . 17

2.2 Scheduling Tasks . . . 21

2.2.1 Table-Driven Scheduling . . . 22

2.2.2 Priority-Driven Scheduling . . . 22

2.2.3 Discussion . . . 24

2.3 Fixed-Priority Scheduling . . . 25

2.3.1 Rate-Monotonic Analysis . . . 26

2.3.2 Deadline-Monotonic Analysis . . . 27

2.3.3 Dependent Tasks . . . 29

2.3.4 Release Jitter . . . 32

2.3.5 Arbitrary Deadlines . . . 33

2.3.6 Aperiodic Tasks . . . 35

2.3.7 Weakly-Hard Tasks 36
2.4 Summary 39

3 A Review On Scheduling with QoS **40**
3.1 Introduction 40
3.2 From Requirements to Scheduling Problems 41
3.2.1 Requirements 41
3.2.2 Design 43
3.2.3 The Scheduling Problem 45
3.2.4 Summary 47
3.3 Types of Utility 47
3.3.1 Levels of Abstraction 48
3.3.2 Levels of Perspective 49
3.3.3 Summary 51
3.4 Dynamic-Priority Scheduling with QoS 51
3.4.1 Discussion 54
3.5 Fixed-Priority Scheduling with QoS 54
3.5.1 Discussion 56

4 Importance **58**
4.1 Introduction 58
4.2 Process Model 59
4.3 Tasks Orderings 60
4.4 Relative Importance Statements 61
4.5 Defining Importance 63
4.5.1 Task Importance 63
4.5.2 Index of Importance Z_I 64
4.5.3 Tasks Equally Important 66
4.6 Scheduling Problems 67
4.6.1 Problem: Deadlines and Importance 68
4.6.2 Problem: Deadlines and Preemptions 68
4.6.3 Problem: Deadlines and Absolute Output Jitter 69
4.6.4 Problem: Deadlines and Relative Output Jitter 70
4.6.5 Problem: Deadlines and Maximum Latency 70

4.6.6 Problem: Deadlines and Relative Maximum Latency 71
4.6.7 Problem: Deadlines and Average Response-Time 72
4.7 Representing Bicriteria Problems 73
4.7.1 Deadlines Metric 74
4.8 Using Importance in Scheduling Problems 75
4.8.1 Importance at Task Level 75
4.8.2 Importance at Application Level 81
4.8.3 Summary 86

5 FPS with Deadlines and Importance: The DI Algorithm 88

5.1 Introduction 88
5.2 Process Model 89
5.3 Problem Description 89
5.4 Solving the Deadlines and Importance Problem 92
5.4.1 Organizing the Search Space 92
5.4.2 The Algorithm DI 94
5.4.3 An Example 99
5.4.4 Summary 99

6 FPS with Deadlines and Conditional Importance: The DI+ Algorithm 100

6.1 Introduction 100
6.2 Conditional Relative Importance 101
6.2.1 Task Conditional Importance 102
6.2.2 Index of Conditional Importance $Z_{\bar{I}}$ 103
6.2.3 Problem: Deadlines and Conditional Importance 104
6.3 Conditional Importance in Scheduling Problems 104
6.3.1 Λ-constraints 104
6.3.2 Precedence Relationships 105
6.3.3 Example 106
6.4 The DI+ Algorithm 107
6.4.1 Modifications to DI 108
6.4.2 Example 111
6.5 Extending the DI+ Algorithm 113
6.5.1 An Example 114

6.6 Summary 116

7 Evaluation **117**

7.1 Introduction 117

7.2 Experimental Setup 118

7.2.1 Task Sets 119

7.2.2 Window of Time 119

7.2.3 Number of Task Sets 120

7.2.4 Characterizing the Feasibility of the Task Sets Generated 125

7.3 Finding Heuristics for Assigning Importance 126

7.3.1 Total Number of Preemptions 128

7.3.2 Output Jitter 129

7.3.3 Latency 130

7.3.4 Maximum Relative Average Response-Time 130

7.4 Evaluating the DI Algorithm with Heuristics 131

7.4.1 Problem: Deadlines and Total Number of Preemptions 139

7.4.2 Problem: Deadlines and Absolute Output Jitter 143

7.4.3 Problem: Deadlines and Relative Output Jitter 147

7.4.4 Problem: Deadlines and Maximum Latency 151

7.4.5 Problem: Deadlines and Relative Maximum Latency 155

7.4.6 Problem: Deadlines and Average Response-Time 159

7.4.7 Summary 163

7.5 DI Algorithm vs Swapping Algorithm 164

7.5.1 Maximising the Importance Metric 165

7.5.2 Maximising/Minimising a QoS Metric 166

7.6 DI+ Algorithm: QoS and Priority Constraints 168

8 Conclusions and Further Work **172**

8.1 Review of Contributions 172

8.1.1 Importance 173

8.1.2 DI and DI+ Algorithms 174

8.1.3 Solutions to Multicriteria Problems 175

8.2 Suggestions for Further Work 177

8.3 Final Remark 178

Appendix A. Notation 179

Appendix B. Bibliography 181

List of Figures

1.1 Precedence relationships 5

1.2 Jitter of the subset $S_8^J = \{x, y, z\}$ for the $8!$ priority assignments. 7

3.1 Attribute Taxonomy for Performance 44

3.2 From Requirements to Tasks 45

3.3 Utility functions 52

4.1 A Bicriteria Solution Space for a task set under criteria $Z1$ and $Z2$. Pareto-optimal solutions are indicated by Greek small letters. The solution s is not pareto-optimal because it is strictly dominated by at least another solution; e.g. $Z1(\omega) < Z1(s)$ and $Z2(\omega) < Z2(s)$. 73

4.2 Plotting all priority orderings of task set S_5. S^I is infeasible ($Z_D(S^I) = 229/80 = 2.86$). S^D is feasible but has low importance index ($Z_I(S^D) = 119$). The *Z_I-optimal* solution is $S^* = \langle b\, e\, a\, d\, c \rangle$ located in $(Z_D(S^*), Z_I(S^*)) = (0.88, 43)$ 78

4.3 Total Number of Preemptions and Relative Output Jitter of subset $\{a, b\} \subset S_5$ computed during its hyperperiod for all $5!$ priority orderings. The *$\mathcal{P}$-optimal* and *$\mathcal{J}^{rel}$-optimal* solutions are indicated. 80

4.4 Total Number of Preemptions computed during the hyperperiod for all $5!$ priority orderings of S_5. The *$\mathcal{P}$-optimal* solution is indicated. The results of some simple rules for assigning priorities are also illustrated as well as the result of simulating S_5 under EDF 82

4.5 All priority assignments of set S_5 ordered lexicographically starting from the ordering $\langle a\, b\, c\, d\, e \rangle$ (i.e. the LC rule). Lines illustrate that following the lexicographic order the first feasible ordering found is the optimal S^*. . . 84

4.6 All priority assignments of set S_5 ordered lexicographically starting from the ordering $\langle c\,b\,d\,a\,e \rangle$ labelled as $\langle 5\,4\,3\,2\,1 \rangle$ (i.e. the C/T rule). $S^* = \langle c\,e\,b\,d\,a \rangle = \langle 5\,1\,4\,3\,2 \rangle$ and the *P-optimal* is $\langle b\,e\,d\,a\,c \rangle = \langle 4\,1\,3\,2\,5 \rangle$ 86

5.1 Plotting all priority orderings of S_5. S^I is not feasible. S^D is feasible but has low importance metric. S^A is the Swapping algorithm. The optimal bicriteria S^* is $\langle b\,e\,a\,d\,c \rangle$ 90
5.2 Task set S_5 and priority orderings S^D (DMPA), S^I (Importance), S^A (swapping algorithm with S^I as input), and the optimal S^*. The lexicographic order is $S^I \succ S^* \succ S^A \succ S^D$ 90
5.3 Similar to Figure 5.1 but the priority orderings are connected according to their lexicographic order. Circles enclose groups of either feasible or near feasible orderings distributed around the indices 23, 47, 71, 95 and 119. The orderings corresponding to these indices have a similar pattern: a task plus a suffix ordered by DMPA. 92
5.4 DI applied to S_5 showing the sequence of operations. 98

6.1 Precedence relationships 106
6.2 5! Priority orderings of set S_W ordered lexicographically. There are 32 weakly-hard feasible priority orderings 115

7.1 Variation on number of preemptions, absolute and relative output jitter with respect to the time window simulated 121
7.2 Variation on maximum latency, relative maximum latency, and relative average response time with respect to the time window simulated 122
7.3 Variation of results of different metrics with respect to the number of task sets simulated. When more than 100 task sets are simulated, low variation is observed 123
7.4 Number of feasible priority assignments. Under RMPA all tasks sets are feasible. The heuristics get worse as the utilisation increase 124
7.5 Total Number of Preemptions under different priority assignments. Only RMPA and EDF guarantee deadlines. The best heuristic is LC. Note that in (c) RMPA is close to the worst possible value (i.e. MAX) 132

7.6 Total Number of Preemptions Normalized under different priority assignments. Only RMPA and EDF guarantee deadlines. The best heuristic is LC. Note that in (c) RMPA is close to the worst possible value 133
7.7 Absolute Output Jitter under different heuristics. The best ones are T/C and 1/C 134
7.8 Relative Output Jitter under different heuristics. Assigning priorities by shorter deadlines is by far the best option. 135
7.9 Maximum Latency under different priority assignments. It seems that LT, T/C and LC are the best heuristics 136
7.10 Relative Maximum Latency under different priority assignments. In general, among the heuristic, the best is $1/C$. 137
7.11 Maximum Relative Average Response-Time under different priority assignments. Among the heuristics, the best ones are T/C and $1/C$. 138
7.12 PLOTS A TYPE. Guaranteeing Deadlines and Minimizing the Total Number of Preemptions. $\mathsf{FP}_{\mathsf{opt}}$ computed by exhaustive search is the best. In general, *DI(LC)* is the best tractable solution excepting when $U > 0.8$ and the metric includes all tasks. In this case EDF is better 140
7.13 PLOTS B TYPE. Guaranteeing Deadlines and Minimizing the Total Number of Preemptions for different task sets 141
7.14 PLOTS C TYPE. Comparing DI(LC) against EDF for the Deadlines and Total Number of Preemptions problem. While on average one outcome is better than the other one, it is not necessarily true for a number of solutions 142
7.15 PLOTS A TYPE. Guaranteeing Deadlines and Minimising the Absolute Output Jitter. DI(1/C) and D(T/C) produce good results. 144
7.16 PLOTS B TYPE. Guaranteeing Deadlines and Minimising the Absolute Output Jitter 145
7.17 PLOTS C TYPE.Comparing DI(1/C) against EDF for the Deadlines and Absolute Output Jitter problem. While on average one outcome is better than the other one, it is not necessarily true for a number of solutions . . 146
7.18 PLOTS A TYPE. Guaranteeing Deadlines and Minimising the Relative Output Jitter. Assigning priorities by shorter period is the best solution in FPS 148

7.19 PLOTS B TYPE. Guaranteeing Deadlines and Minimising the Relative Output Jitter 149
7.20 PLOTS C TYPE. RMPA against EDF for the Deadlines and Relative Output Jitter problem. While on average one outcome is better than the other one, it is not necessarily true for a number of solutions 150
7.21 PLOTS A TYPE. Guaranteeing Deadlines and Minimising the Maximum Latency under different priority assignments. Our best solution is DI(LC) 152
7.22 PLOTS B TYPE. Guaranteeing Deadlines and Minimising the Maximum Latency 153
7.23 PLOTS C TYPE. Comparing DI(LC) against EDF for the Deadlines and the Maximum Latency problem. While on average one outcome is better than the other one, it is not necessarily true for a number of solutions . . 154
7.24 PLOTS A TYPE. Guaranteeing Deadlines and Minimising the Relative Maximum Latency. In all cases, both DI(1/C) and DI(T/C) outperform EDF. Note that DI(1/C) is a near-optimal solution. 156
7.25 PLOTS B TYPE. Guaranteeing Deadlines and Minimising the Relative Maximum Latency 157
7.26 PLOTS C TYPE. Comparing DI(1/C) against EDF for the Deadlines and the Relative Maximum Latency problem. Observe the excellent performance of DI(1/C) 158
7.27 PLOTS A TYPE. Guaranteeing Deadlines and Minimising the Maximum Relative Average Response-Time. In all cases, both DI(1/C) and DI(T/C) are near-optimal solutions and outperform EDF 160
7.28 PLOTS B TYPE. Guaranteeing Deadlines and Minimising the Maximum Relative Average Response-Time 161
7.29 PLOTS C TYPE. Comparing DI(T/C) against EDF for the Deadlines and the Maximum Relative Average Response-Time problem. Observe the excellent performance of DI(T/C) 162
7.30 For high utilisations, the number of feasible tasks sets decreases and therefore, the distance between S^D and S^* gets smaller. The distance between S^A and S^* increases slightly. 165
7.31 The solutions S^D and S^A move dramatically away from S^* conforming the number of tasks increases. 166

7.32 Comparing DI(LC) against A(LC) results for the problem of minimising the total number of preemptions 169
7.33 Guaranteeing Deadlines and Priority Constraints $\{P_7 > P_1, P_6 > P_5\}$ for different QoS metrics. Note that constraining the freedom for assigning priorities reduces the chance for improving the QoS metric. The DI solution does not meet the priority constraints 170
7.34 Guaranteeing Deadlines and Priority Constraints $\{P_7 > P_1, P_6 > P_5\}$ for different QoS metrics. Note that constraining the freedom for assigning priorities reduces the chance for improving the QoS metric. The DI solution does not meet the priority constraints 171

List of Tables

1.1 Task set S_8 ordered in non-increasing order of deadlines. It is feasible under DMPA. The relative output jitter (ROJ) is shown. Note that the subset of interest $\{x , y , z\}$ suffers jitter. The utilisation factor is 0.69. . 5

1.2 Two alternative feasible solutions for task set S_8. Both minimise ROJ but only the second one meets the precedence constraints 8

4.1 When in $\{a, b, c\}$ all tasks have different importance, there are 6 different indices of importance; when *a* and *b* have the same importance, there are only 3 different indexes 66

4.2 Task set S_5 with importance values I. a is the highest importance task and e is the lowest one. Note that is is feasible under DMPA. 76

4.3 Relative output jitter ($\jmath^{rel}$) and total number of preemptions ($\mathcal{P}$) for tasks a and b obtained by the simulating of the orderings S^D, S^I and S^* (smallest figures are better) 77

4.4 Simple rules for assigning priorities. For example, under the rule 1/C, priorities assigned to tasks are inversely proportional to the length of C. Its contrary is LC, where priorities assigned to tasks are proportional to the length of C 81

5.1 Executing the swapping algorithm with input $S^I = \langle a\,b\,c\,d\,e \rangle$. The result is the ordering $S^A = \langle e\,a\,b\,d\,c \rangle$ 91

6.1 Task set S_8 with their precedence constraints. It is ordered by DMPA. . . 106

6.2 DI algorithm with input $S_8^{\bar{I}}$ labelled by importance $\langle 8\,7\,6\,5\,4\,3\,2\,1 \rangle$. The solution $\langle 8\,7\,6\,5\,3\,2\,4\,1 \rangle$ is found in 10 steps from a universe of 8! 107

6.3 DI+ algorithm with input $S_8^{\bar{I}}$ labelled by importance $\langle 8\,7\,6\,5\,4\,3\,2\,1 \rangle$ with constraints $\Lambda = \{P_6 > P_3, P_4 > P_3, P_8 > P_5, P_8 > P_7\}$ 112

6.4 Task set S_W with Weakly-Hard Constraints 114

7.1 Simple rules for assigning priorities 126

7.2 Classification according to the performance. 1 is the best and 4 the worst scheduling solution. 164

7.3 Comparing solutions obtained with DI and swapping algorithms for different utilisation with heuristics LC and T/C. Note how similar the results are. From 1000 priority orderings, the column "Identical" shows the number of identical ones. 167

Chapter 1

Introduction

Scheduling on computing systems has been studied for around fifty years. The first techniques were adopted from results developed in the operational research area and since then it has evolved almost independently. While operational research scheduling is concerned with the utilisation of people, equipment and raw materials, computing systems scheduling is concerned with the utilisation of processors, programs, memory, I/O devices, and so on.

Scheduling can be defined as "*a plan for performing work or achieving an objective, specifying the order and allotted time for each part*" [1]. Another definition is termed by Pinedo [2], who states that *"scheduling deals with the allocation of scarce resources to tasks over time. It is a decision-making process with the goal of optimising one or more objectives"*. The scarce resources can be CPU's, I/O devices or energy sources. The tasks can be application programs, execution threads or messages travelling through a network. Each task can have different properties that distinguish it among others such as processing times, release dates and deadlines. Examples of objectives are the minimisation of deadlines missed, maximisation of resources, or minimisation of energy consumption.

Three main parts are identified: *resources*, *tasks* and *objectives*. These three elements together define a *scheduling problem* and the research domain dealing with these problems is *scheduling theory*.

Theoretically, computer systems can execute a number of different algorithms provided that sufficient time and storage space are available. Computing programs spend

time performing the calculations, and memory to store the program and related data. In this context, a program is realizable if the program can both be fitted in memory and give valid results within reasonable time. The memory issue does not permit ambiguities, the program and all related data must reside in memory (primary and/or secondary). Contrarily, the time issue can be interpreted in several ways depending on the circumstances. Precisely, the time issue has split up computer systems in two well defined areas: general-purpose computing systems, where average time performance is sufficient and real-time systems where specific timing constraints must be fulfilled.

Thus, on conventional computing systems the results are satisfactory whether they are obtained in a unspecified but finite time. On the other hand, in real-time systems time is essential; the results are valid only if they are always produced within specific time delays.

1.1 Real-Time Systems

A Real-Time System is a computing system that must react within precise timing constraints to events in its environment [3]. Timing constraints are expressed as a function of committed completion times called *deadlines*. Real-time systems can be categorized according to the criticality of their deadlines. In a *hard real-time system*, it is compulsory that responses occur within specific deadlines [3]. In a *soft real-time systems*, the timing constraints can be satisfied acceptably well with acceptable predictability depending on application specific requirements [4]. More recently, there have been defined systems where it is tolerable to miss some deadlines, if it happens in a predictable way. For instance, some systems allow that a task misses any n in m deadlines in a time interval. These are called *weakly-hard* real-time systems and they are characterized by the distribution of their met and missed deadlines during a window of time[5]. Thus, real time systems can be classified according to their timing constraints as hard, weakly-hard or soft.

As in any other engineering discipline, a Real-Times System has a life cycle model divided in different phases such as requirements, design, implementation, testing, operations and maintenance.

During the specification of requirements, several Quality of Service (QoS) require-

ments (e.g. safety, reliability, performance) are defined. These requirements form a multidimensional set of interrelated requisites that must be conveyed throughout all stages of the development cycle. Normally, conflicts among requirements exist and then tradeoffs have to be defined to help the designers with their decisions. Assigning importances to the requirements is a mechanism to specify such tradeoffs. In effect, not all requirements are equally important; some may be essential while others may be desirable, and therefore each requirement should be rated according to importance to make these differences clear and explicit [6].

During the design phase a real-time system can be structured as a set of concurrent tasks with deadlines to meet. The tasks share some scarce computer resources that must be allocated optimally by a scheduling algorithm in order to obtain the greater benefit possible. A single task provides some benefit to the system when it achieves all or part of one or more QoS requirements. Thus, *the scheduling problem consists of how to meet the deadlines while maximising the QoS.*

1.2 Motivation: Problems with Deadlines and QoS

Scheduling problems where simultaneously timing and QoS requirements have to be fulfilled have been widely studied in the context of non-critical systems [7] [8] [9]. Most approaches rely on on-line scheduling algorithms that try to maximise an overall system value metric. Unfortunately, the applicability of such results is not possible in commercial systems due to the lack of support for on-line scheduling facilities.

On the other hand, almost all commercial real-time kernels (e.g. QNX, PDOS, VxWorks), real-time operating systems (e.g. RT-Linux, RT-Mach), run-time environments for languages (e.g. Ada95, Real-Time Specification for Java) and industry standards (e.g. POSIX, OSEK) provide support for fixed-priority scheduling only. *Fixed-Priority Scheduling* (FPS) is a framework formed by policies for assigning static priorities, feasibility tests to decide if a priority assignment can fulfil the timing constraints, and run-time support algorithms to execute the tasks according to their priorities such that at all times the task with the highest priority is always executing. FPS is considered an industry standard providing good performance, predictability and flexibility. Many government agencies

and system integrators recommend it as the method of choice [10] [11].

In this thesis we argue that fixed-priority scheduling is a robust and proven technology and that significant improvements in quality of additional performance and QoS metrics can be achieved by adequate priority assignment of tasks.

When all deadlines have to be guaranteed, the only priority assignment that is usually considered is the Deadline Monotonic Priority Assignment (DMPA) (equivalent to Rate Monotonic Priority Assignment when deadline is equal to period). However, among all the possible feasible priority assignments there may be other feasible assignments that may be better according to other QoS metrics. When there are at least two different optimality criteria, it is said that it is a multicriteria problem [12].

In this thesis we focus on multicriteria scheduling problems where all tasks need to meet all their deadlines, and where tasks have additional QoS requirements that result in different perception of the quality of a system when tasks do complete by their deadline, without modifying both the base scheduling mechanism and the task characteristics. In contrast some research in the literature concentrates on increasing some QoS measure by modifying the two above system properties. The next section clarifies this idea.

1.2.1 Example: Problem with Deadlines and Output Jitter

Consider the problem of a system where the QoS relies only on a subset of the tasks which are sensitive to output jitter. The *output jitter* is the variation in the inter-completion times of successive instances of the same task [13]. It is known that in standard real-time scheduling approaches the presence of output jitter causes degradation in control performance and may lead to instability. In multitasking systems, scheduling algorithms introduce some forms of jitter such as sample jitter (i.e. variations in their release time), output jitter, or a combination of both [14]. Let us concentrate on the output jitter problem such that in the rest of the example, jitter, means output jitter.

In FPS, any particular task suffers jitter due to the variation in the interference produced from high priority tasks. Thus, it may be the case that under a particular priority assignment algorithm such as DMPA, some of the tasks that are sensitive to jitter are as-

task	*C*	*T=D*	*R*	*Description*	*ROJ*
a	2	10	2	Alarm	0
x	1	16	3	Pilot instruction	0.125
y	2	16	5	Flaying gears	0.125
b	1	16	6	Cabin temp & press.	0.125
z	3	32	9	Temp. sensor	0.0625
c	2	32	13	Pressure sensor	0.0625
d	1	32	14	Speed control	0.0625
e	3	56	23	Human interface	0.357

Table 1.1: Task set S_8 ordered in non-increasing order of deadlines. It is feasible under DMPA. The relative output jitter (ROJ) is shown. Note that the subset of interest $\{x , y , z\}$ suffers jitter. The utilisation factor is 0.69.

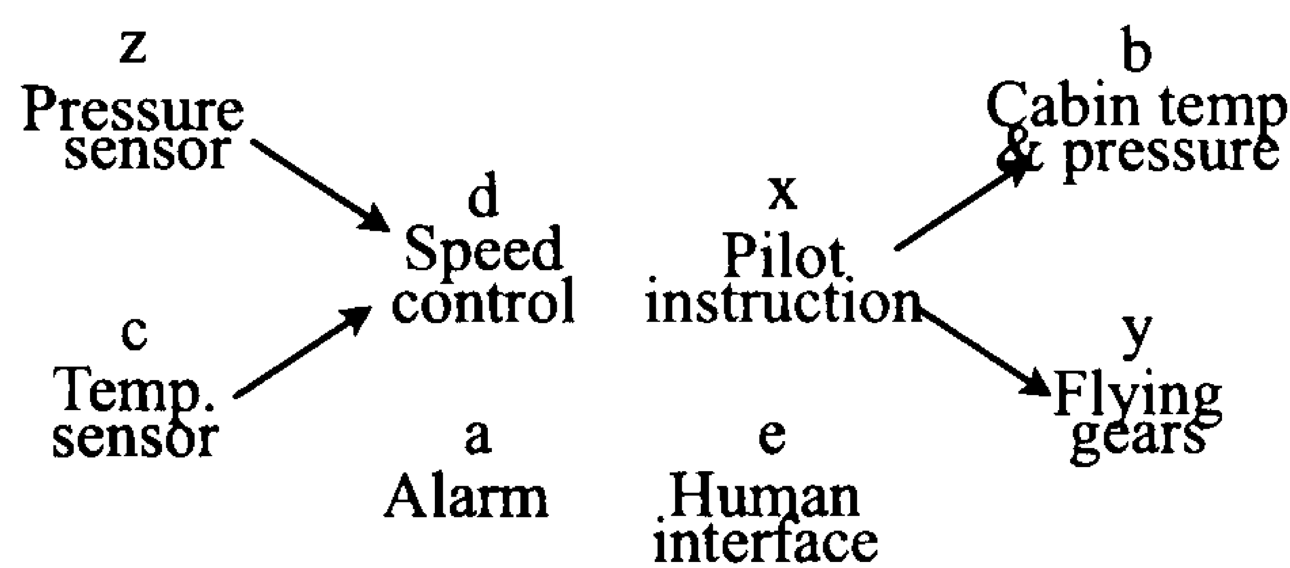


Figure 1.1: Precedence relationships

signed lower priorities and therefore suffer from jitter. Moreover, the interference of high priority tasks can be related to other QoS issues in tasks with low priorities.

Jitter problems are dealt with by techniques such as designing special purpose task models, using jitter compensation schemes in feedback controllers, optimising the selection of task attributes (e.g. deadlines, offsets), or a mixture of them [15] [14] [16]. Those techniques are powerful as they can be used to effectively reduce jitter. However, it is our thesis that there exist priority assignments that are both feasible and are able to reduce jitter.

Consider the task set S_8 of Table 1.1. This task set is taken from [16] and corresponds to an aircraft control system. All tasks have to complete by their deadlines and the tasks in subset $S_8^J = \{x, y, z\}$ have jitter requirements. In addition, the tasks have precedence relationships $z \rightarrow d$, $c \rightarrow d$, $x \rightarrow b$ and $x \rightarrow y$, where, for instance, $z \rightarrow d$ means that z must finish before d.

Note that the task set is feasible under DMPA and fulfills the precedence constraints but

its QoS with respect to the output jitter could be improved. This example is solved in [16] by introducing offsets (using a technique with exponential complexity), re-assigning deadlines to maintain the schedulability under DMPA and finally, testing feasibility by simulation. Instead of modifying the task characteristics, a simple but alternative approach consists of performing an exhaustive search on the 8! possible priority assignments looking for a one that fulfills timing and precedence requirements, and minimises the jitter. The hyperperiod of S_8 is small (1120 time units) and therefore, the 8! priority assignments can be simulated and the jitter for the subset S_8^J computed and stored.

The results of such simulation are plotted in Figure 1.2. The simulation uses the worst-case computation time C all the time. For each fixed-priority assignment, (1) a feasibility test is performed and (2) a simulation to measure the jitter is executed:

1. A priority assignment is feasible if it passes the response time test, which consists of computing the response time R_j of each task and comparing it with the respective deadline D_j such that if $R_j \leq D_j$, $\forall j$, the priority assignment is feasible [3].

2. The relative output jitter (ROJ) of a particular task j is the variation in the inter-completion times of the successive instances of the same task divided by its period [13].

A point (x, y) on the graph corresponds to a different priority assignment as follows:

- $x = \max\{R_j/D_j\}$ $\forall j \in S_8$. This metric indicates whether the priority assignment guarantees or does not guarantee the deadlines of all tasks (see section 4.7.1 (page 74)). If $x \leq 1$ the priority assignment is feasible; otherwise it is infeasible. When $x = 1$ or very close to 1, it indicates at least one task finishes very close to its deadline. We use this metric because it is more expressive than a simple yes/no answer.

- $y = \max\{\text{ROJ}_j\}$ $\forall j \in S_8^J$: This metric indicates how much jitter this priority assignment produces. It is the maximum Relative Output Jitter of any task in the subset S_8^J under a particular priority assignment (see definition 4.6.5, page 70).

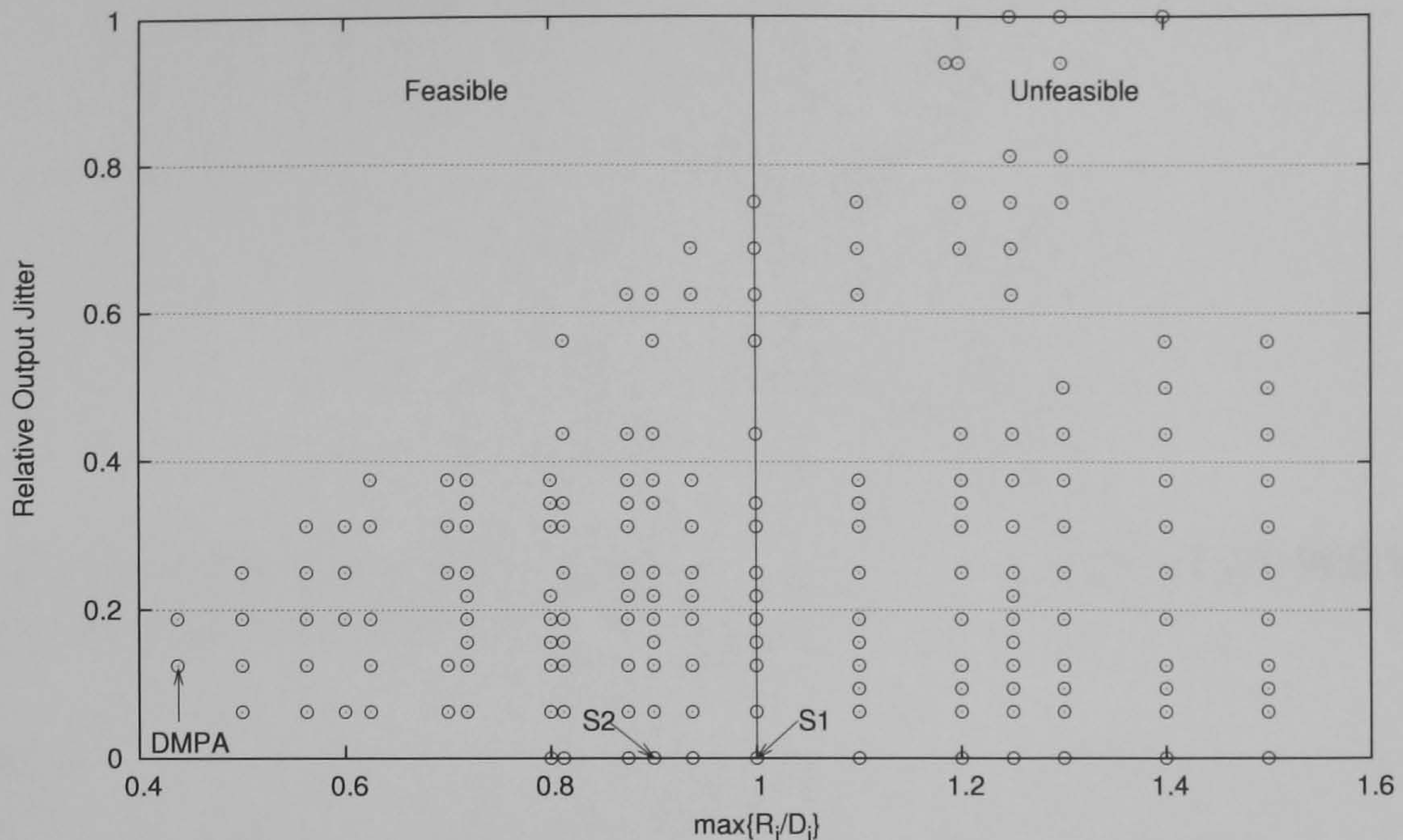


Figure 1.2: Jitter of the subset $S_8^J = \{x, y, z\}$ for the 8! priority assignments.

For instance, the task set in Table 1.1 is ordered by DMPA. Applying both above metrics gives:

$$\text{DMPA}_{\max\{R_j/D_j\}} = \max\{R_j/D_j\},\ \forall j \in S_8$$
$$= 0.4375$$

and

$$\text{DMPA}_{ROJ} = \max\{\text{ROJ}_k\},\ \forall k \in S_8^J$$
$$= 0.125$$

Thus, the priority assignment given by DMPA is located at $(0.4375, 0.125)$ in Figure 1.2.

Though there are $8! = 40320$ points in Figure 1.2, most of them are overlapped indicating that a number of priority assignments produce the same jitter result and the same maximum response time. Observe that in the bottom end of the graph, on the feasible side, there are six points. They correspond to feasible priority assignments with zero output jitter. For example, Table 1.2 shows two assignments with zero output jitter but only solution *S2* fulfills the precedence constraints.

S1	*R*	*ROJ*
x	1	0
y	3	0
z	6	0
b	7	0.187
d	8	0
a	10	0.8
c	14	0.0625
e	23	0.357

S2	*R*	*ROJ*
x	1	0
y	3	0
z	6	0
b	7	0.187
a	9	0.218
c	13	0.2
d	14	0.0625
e	23	0.357

Table 1.2: Two alternative feasible solutions for task set S_8. Both minimise ROJ but only the second one meets the precedence constraints

Summary

This example shows that scheduling problems with hard deadlines and QoS requirements expressed as quantitative metrics can be solved not only using complicated algorithms but also with pure fixed-priority assignment approaches. Unfortunately, current analysis for FPS lacks mechanisms for finding such priority assignments and, naturally, enumerating all assignments is in the general case impracticable. Consequently, finding algorithms that convert this intractable problem into a tractable one is a challenge.

In this thesis we present a general mechanism that solves problems like that illustrated above in pseudo-polynomial time (instead of trying all $N!$ different priority orderings). This makes such problems tractable even for large task sets. Even though we use the example of output jitter, the approach presented here is generic and can be applied to a wide variety of scheduling problems in fixed-priority systems.

1.2.2 Discussion

Though quantitative metrics can be defined to express several QoS requirements, it seems clear that not all system requirements can be expressed quantitatively. In fact, there exist qualitative requirements that depend on particular desires of users or system designers that are difficult to formulate and model appropriately.

For example, assume that the system described above is developed with the priority assignment of Table 1.1, and only during the testing phase it is discovered that improve-

ments in quality are required. A first improvement consists in the reduction of output jitter and therefore, the priority assignment S2 (see Table 1.2) is optimal in that sense. However, for any decision maker, S2 could not be the best solution, for instance:

1. users could not be satisfied with the responsiveness of the human interface; different levels of satisfaction may be perceived by different users.
2. a system designer may note that the worst-case response time of the Alarm task is very close to its deadline and, in an overload situation, its deadline could be missed; thus, he may prefer a higher priority for task a.

Thus, the first issue is how to express appropriately such qualitative QoS requirements and afterwards, the next issue is how to solve the scheduling problem.

Expressing QoS

Traditional scheduling theory in real-time systems use utility functions to describe the satisfaction obtained when completing some computations. This satisfaction may be qualitative and/or quantitative. A limitation is that an appropriated utility function has to be formulated for each task. Clearly, it cannot be determined easily. In fact, there does not exists a methodology for their design [8] and the assignment of utility functions to tasks is usually imprecise. These disadvantages have been noted in other areas. For example, instead of utility functions, pure ordinal relationships that specify which of any two bundles of commodities individuals prefer are utilised in economics. The relationship reflects the extent to which a thing is preferred to others. *Ordinal preference relationships is a concept that may be introduced in real-time systems for expressing QoS requirements.*

Scheduling with QoS

Assuming that QoS may be expressed appropriately, a scheduling problem may be formulated and its solution may be proposed. Naturally, less expensive solutions are the preferred ones. For example, with regard to the two above observations to the solution S2 proposed, increasing computing resources or performing some optimisations to the code

could be costly solutions. On the other hand, an inexpensive alternative consists in manipulating the priorities for raising "as higher as possible" the priorities of tasks without modifying the improvements already obtained. How to achieve such priority assignment is the main idea of our research: *In fixed-priority scheduling there exist feasible priority assignments that meet additional QoS requirements. The problem consists on how to find such priority assignments without performing exhaustive search.*

1.3 Thesis Aims

The following statement synthesizes the thesis proposal:

In the context of fixed priority scheduling of real-time systems, multicriteria problems can be solved combining novel and established techniques based on assignment of priorities. Optimal or near-optimal solutions can be proposed and evaluated. A solution consists of a feasible priority assignment that maximises/minimises a novel QoS metric based on simple ordinal importance relationships. The solution proposed is generic and can be applied to a wide variety of scheduling problems in hard real-time systems.

We investigate the problem of finding priority orderings that simultaneously satisfy QoS and timing (deadline) requirements. The problem of finding an assignment of priorities that maximises a set of QoS constraints and satisfies all deadlines is know to be NP-Hard. We argue that the intrinsic complexity of the problem stems from the value system usually utilised to face the problem, and in particular due to the scale of measurement and the intrinsic comparison of value between tasks. Nevertheless, the assignment of values to tasks is usually imprecise and complex utility-based scheduling algorithms based on these values are therefore based on imprecise input data. Instead, we propose a weaker notion of QoS, a simple importance relationship between tasks rather than an absolute value attached to each task. The meaning of this importance relationship is to express that in a schedule it is desirable to run a task in preference to another one. This simplified model captures even complex QoS metrics between tasks. Thus, the aim is to develop algorithms based on the concept of importance to express QoS requirements, such that combined with well-established feasibility test and other techniques, scheduling problems with deadlines and QoS requirements can be solved.

1.3.1 Contributions

In this thesis we present a general mechanism that allows solutions to several scheduling problems based on the simple assignment of fixed-priorities to tasks in pseudo-polynomial time. The main contributions are:

- The development of a metric for expressing QoS requirements based on the concept of relative importance among tasks.
- The development of algorithms that find optimal solutions for problems with hard deadlines and the metric of relative importance.
- A set of heuristics based on simple rules for assigning importance to tasks. These heuristics combined with our algorithms, provide solutions to problems with hard deadlines and QoS requirements such as jitter, preemptions and latency.

1.3.2 Research Approach

Relative importance among tasks is a preference for executing a task with regard to the other tasks in the system. By associating different importance to tasks, importance orderings can be defined. For instance, the ordering where the first task has the highest value of importance, the second task has the second highest importance, the third one has the third highest importance and so on, is the highest importance ordering.

We observe that such orderings can be arranged lexicographically and indexed by importance. The lexicographic index permits to define a QoS metric as the *lexicographic distance between any ordering and the highest importance one.* This importance metric expresses a distance from a level of quality to the maximum one. Thus, minimising this distance maximises the QoS. Based on these concepts, a scheduling problem is formulated as *finding a feasible priority ordering that maximises the metric of importance.*

In order to solve the above scheduling problem, we present the DI (Deadline and Importance) algorithm that finds in a polynomial number of steps, an optimal priority ordering that meets hard deadlines ($D \leq T$) and that satisfies the importance metric. Optimality in this context means that there is no other feasible schedule with higher importance. The

DI algorithm is posteriorly extended in the DI+ algorithm which provides support for precedence constraints, arbitrary deadlines ($D \leq T$ or $D > T$) and weakly-hard timing constraints. The correctness of the algorithm is proved.

The DI and DI+ algorithms find an optimal solution in $O((N^2 + N)/2)$ steps (in the worst-case) where N is the number of tasks. This makes the problem tractable even for large task sets. However, the complexity of the solution depends on the complexity of the feasibility test due to performing a feasibility test at each step. For instance, the complexity of DI is pseudo-polynomial when the feasibility test is the response time analysis [3].

We also observe that the concept of importance is semi-abstract but representable by a specific metric (i.e. the importance metric); consequently, it could be used to solve other problems if analogies with the relative importance concept are found. In this context, we propose that *if a particular problem with QoS metric Z_{QoS} can be related with a problem with Importance metric Z_I, then solving the problem with Z_I also will solve the problem with Z_{QoS}.*

In order to demostrate the validity of the above statement a case study is presented in chapter 7. It is separated in two parts:

- In the first one, we simulate different simple priority assignments to observe how good or bad they are with respect to different QoS metrics. Examples of priority orderings tested are shortest deadline first, largest deadline first, shortest worst-case computation time first; examples of metrics are minimising the output jitter, minimising the number of preemptions, and minimising the latency. The results of the experiments show that some of these assignments of priorities are particularly good with respect to the QoS metrics but, naturally, they do not guarantee the deadlines.
- Those priority assignments that are good with regard to a particular metric are used as assignments of importance and then, by applying the DI algorithm, we find good or near-optimal solutions for such QoS and deadline problems. The case study also compares the solutions found with the corresponding Earliest Deadline First scheduling solutions. Remarkably, it is shown that contrary to previously published results[17], the fixed-priority scheduling solutions obtained outperform EDF in a number of scenarios.

This thesis provides algorithms that produce fixed-priority orderings of superior quality than those obtained by current FPS theory. The QoS is improved without modifying the installed base of fixed-priority systems.

1.3.3 Thesis Organisation

The remainder of this thesis is organized as follows:

Chapter 2. A Review On Preemptive Fixed Priority Scheduling

This chapter reviews the main concepts on preemptive fixed-priority scheduling for uniprocessor real-time systems utilised in our research. In fixed-priority scheduling the priority assignment and the feasibility analysis are the two main areas of research. With respect to priority assignments, our review covers from Rate Monotonic Priority Assignment (to solve problems with deadlines equal to periods) to swapping algorithms (to solve problems with arbitrary deadlines or weakly-hard constraints). With respect to feasibility analysis, it covers from the utilisation-based test to the weakly-hard analysis.

Chapter 3. A Review On Scheduling with QoS

In real-time systems the concept of importance among tasks is usually defined in terms of utility functions. In this chapter, we briefly make a review of some scheduling mechanisms for improving the QoS measured in terms of utility. It starts with a concise review of how the system requirements are mapped from the specification to the design. Afterwards, a summary of how the utility is perceived in a system at different levels of abstraction is presented. Finally, a review of scheduling approaches for improving QoS is presented.

Chapter 4. Importance

In this chapter we propose a model for expressing QoS requirements in real-time systems based on the concept of relative importance among tasks. We define importance in terms of predilection for executing tasks in preference to other tasks in a system as well as

predilection for executing tasks in specific orders. The concept is based on preferential statements of the form "α is more important than β". We define a QoS metric using this concept and propose its use in the context of fixed-priority scheduling. In addition, we define some scheduling problems in terms of fixed-priorities; solutions to these problems are proposed in the following chapters applying our model of importance.

Chapter 5. Fixed-Priority Scheduling with Deadlines and Importance: The DI Algorithm

In this chapter we present the DI algorithm that solves the scheduling problem where hard deadlines and importance criterion have to be optimised. The problem is formulated as finding an optimal priority ordering that maximises the importance criterion. Optimality in this context means that there is no other feasible schedule with higher importance. The DI algorithm finds such optimal priority assignment in $O((N^2 + N)/2)$ where N is the number of tasks.

Chapter 6. Fixed-Priority Scheduling with Deadlines and Conditional Importance: The DI+ Algorithm

In this chapter the concept of conditional relative importance is added to our model of importance. Conditional relative importance statements express preferences for executing a task with regard to other ones in the system subject to conditional clauses. This concept allows implementing priority relationships constraints such as precedence constraints and allows including importance as a hard requirement. We formulate the scheduling problem with deadlines and conditional importance and introduce the DI+ algorithm for solving it. DI+ is DI with some modifications such that the feasibility test can be substituted by other tests. In particular, by including feasibility tests for tasks with arbitrary deadlines or weakly-hard constraints and including the swapping algorithm for finding feasible priority orderings, DI+ is enabled to cope with such problems.

Chapter 7. Evaluation

In this chapter we evaluate different heuristics for assigning importance such that when used with the DI algorithm, solutions to multicriteria problems are presented. In order to improve the quality of a system, we assume that each task is characterized not only by a deadline but also by an importance value which is correlated with a quantitative QoS metric. A heuristic assigns importance to tasks pursuing to minimise (or maximise) a QoS metric and then the DI algorithm finds a feasible solution. The performance of this approach is then evaluated. The best combination DI algorithm with heuristic is proposed as a fixed-priority solution for improving the quality of the system. Metrics considered are for the total number of preemptions, output jitter, latency and average response time.

Chapter 8. Conclusions and Future Work

The final comments about the research results are given in this chapter. In addition, some directions for further research are also presented.

Chapter 2

A Review On Preemptive Fixed Priority Scheduling

This chapter is an introduction to the concepts in preemptive fixed-priority scheduling for uniprocessor real-time systems utilised in our research. It is not our intention to cover all topics but only those required to understand the subsequent chapters. It includes Rate Monotonic scheduling, Deadline Monotonic scheduling and scheduling in weakly-hard systems.

2.1 Real-time Systems

A real-time system performs some activities and responds within timing constraints to events in its environment. Timing constraints are expressed in terms of committed completion times called deadlines. We use the terms deadline and timing constraint interchangeably. Usually, real-time systems can be classified according to the semantics of the deadlines. A *hard real-time system* has critical deadlines that must be met; otherwise a catastrophic system failure can occur. A *soft real-time system* has non-critical deadlines and hence missing some deadlines occasionally is not catastrophic but it is an undesirable effect [3]. In a *weakly-hard real-time* system [5] some deadlines can be missed but their number and/or frequency must be bounded and well determined. Naturally, a system can have a mixture of tasks with critical or non-critical deadlines.

An *embedded* system is a special-purpose computer system built into a larger sys-

tem, which does not interact directly with humans in a regular manner. For instance, a car controller system only receives inputs from sensors and sends outputs to actuators. Real-time systems are typically embedded. In addition, it is said that a real-time system is *synchronous* when their activities are coordinated by a clock that keeps track of time. An *asynchronous* system interacts with the world at any time. Clearly, a mixture of synchronous and asynchronous activities can coexist [3].

A real-time system is usually designed as a set of activities to be executed concurrently and continuously on a single or multiple processors. Each of these activities models a part of the system that interacts with each other and with the environment. These activities are usually called the *process model.* In our context, it will assumed that the process model corresponds to a single processor system only.

2.1.1 Process Model

A process model is represented as a *task set* $S = \{a, b, c, \ldots\}$ with cardinality N. For scheduling purposes, a task set can be ordered totally in different ways. Such an *ordering* is usually a totally ordered task set. An activity or *task* is either a computer process or a single thread of control that is executed by a single processor. Each task has a number of distinctive features many of which are described next; the subscript j usually refers to a task.

A task becomes ready for execution at its *release time* (r) , executes its first computation at beginning time (b) , and concludes at its *completion time* (c) . The difference between its completion time and its release time is called the *response-time.* The amount of processor time required to complete, if it executes alone, is called the task *execution time.* The execution time is not always the same, it varies depending on different circumstances but the *worst-case execution time* (C) must be calculated for deterministic systems. On the system life-time, a task has an *initial release time* (r_0) at time zero and afterwards it is released (or invoked) for execution a number of times along the time axis. Each release is called an instance and is separated from the previous ones by a time interval; this time interval can be random, quasi-random but with a minimal separation known, or constant. For instance, when the time interval is a constant T, it is said that the task is periodic with

period T invoked at times kT with invocations $k = 0, 1, 2, \ldots$. The longest response-time of all instances of a task is called the *worst-case response-time* (R) . In addition, sometimes it is useful to specify a task release in terms of an initial time plus a constant O called *offset* such that the task is invoked at times $kT + O$.

At each release, it is expected that a task completes before its deadline (D) . A *hard deadline* is one that is critical for the system functionality and therefore a task missing one is considered a system failure. A *soft deadline* is non-critical for the system functionality and therefore a task missing one occasionally is tolerated depending on different circumstances. For instance, a task with a soft deadline may typically produce useful results even if it finishes late; however, there are tasks where their results are useless when a deadline is missed; these soft deadlines are called *firm deadlines*. More general soft deadlines can be expressed as a value-function where a task has a *value* when it completes before its deadline and such value changes (usually the value decreases) as its lateness increases. Commonly, tasks with soft deadlines are expected to complete as early as possible.

In addition, some tasks can tolerate missing deadlines not in occasional terms but in a predictable way. This kind of timing constraints is called weakly-hard. In effect, a *weakly-hard constraint* is an exact description (in a window of time) of a required pattern of missed/met deadlines that a task must fulfil in the worst-case. For instance, it is said that a task "meets any *m* in *k* deadlines" if in any window of *k* consecutive instances of the task, there are at least *m* instances that meet their deadline [18].

In this context, tasks can be classified orthogonally according to their release and timing constraints as follows:

Task Categorization			
Arrival	Hard Deadline	Soft Deadline	Weakly-Hard Constraint
Periodic	*hard periodic task*	*soft periodic task*	*wh periodic task*
Minimal known	*sporadic task*	*aperiodic task*	*wh aperiodic task*
Random	N/A	*aperiodic task*	*wh aperiodic task*

A *periodic task* has a period T and can have a hard, soft, or weakly-hard constraint; an *aperiodic task* does not have a period and can have a soft or weakly-hard constraint; in the literature, hard aperiodic tasks are not defined because there is no way of guaranteeing such hard deadlines; *sporadic tasks* have a minimal inter-arrival time (also denoted by T) and a

hard deadline. Note that adding columns for a wider classification of deadlines increases the above categorization. In general, several classifications can be defined depending on the semantics of task releases and deadlines, but this is the usual one.

Finally, tasks can be interrupted or even not released depending on the application. A task is *rescindable* if it can be revoked or not released at any time; when a task has to be executed until completion it is called a *non-rescindable* task.

Preemptions

When executing, a computer task can be suspended at any time, at specific points or not allowed to suspend at all. A task that can be suspended at any time and resumed from the point of suspension without affecting its behavior is called a *preemptive task.* A task is *non-preemptive* when it is not allowed to suspend it. A task is *self-preemptive* when it is organized in non-preemptive blocks such that at the end of each of these blocks the task offers a de-scheduling request to the system; if a high priority task is ready it start its execution, otherwise the next block of the current task is executed [19]. The process of saving and restoring sufficient information of a task such that it can be resumed after being interrupted is called a *context switching.*

Release Jitter

Normally, it is assumed that a task is released as soon as it arrives (i.e. when it is able to run). However, it may occur that the task is subject to a bounded delay between its arrival and release, and then it is said to exhibit release jitter. *Release jitter* J is the worst-case delay between the arrival time and the release time of a task [19].

Release jitter takes place due to system implementation techniques as, for example, a tick-driven scheduler. A tick-driven scheduler periodically polls task arrivals at fixed time ticks; at these times the scheduler performs its scheduling decisions. It could be the case that a task arrives just a tick later from the last poll checking such that the task arrival will be noted at the next scheduler period when the task will be released (i.e. put on the run queue). For a periodic task, release jitter can be avoided by choosing its period to be

multiple of the period of the scheduler. For a sporadic task, it cannot be avoided as its release time cannot be exactly determined; its release jitter is determined by the polling period of the process that awakes it or the polling period of the scheduler [3].

Dependency Relationships

It is said that two tasks are *independent* when the results of their computations do not depend on each other; otherwise, they are *dependent*; for instance, dependent tasks require communicating data or synchronizing their computations.

Some dependent tasks interact accessing mutually exclusive resources for both communication and synchronization purposes. Tasks that share common resources have to use synchronization primitives such as semaphores, monitors or protected objects to gain exclusive access to the resources [3]. Thus, a task will experience *blocking* when it is stopped in a semaphore waiting for the release of the resource. In real-time systems, the *worst-case blocking time* must be determined.

Other aspect of dependent tasks is when a task cannot start until another task has completed. In this case it is said that a *precedence relationship* exists. Data is often passed between these tasks but they never execute together and therefore, mutual exclusion over this data does not need be enforced.

Value

Sometimes the notion of value is attached to a task. The *value* V is a scalar that expresses the weight or the intrinsic benefit obtained by running a task [20]; in other words, value is an artifact that expresses a measure of quality and conveys how valuable is the task for the system. Note that value is not a value-function as defined in the literature [4] as a value-function combines urgency (deadlines) and benefit (value) in a single expression.

Other Properties

In the above description, a task executes at each release a single stream of computations with a processing time, inter-arrival time, deadline and so on. However, in some applications tasks can have multiple deadlines in the same release, or can have different computation times, or periods from release to release [19].

Furthermore, in more complex schemes (as described in [21]), a task x can be broken down into a series of operations or optional sub-tasks, where each one can have a processing time, inter-arrival time, deadline and so on. A mono-operation task consists on one operation and a multi-operation task consists on several operations x_k with $k = 1, 2, \ldots$. The operations can be

- mutually inclusive: $x = (x_1$ and x_2 and $x_3 \ldots)$,
- mutually exclusive: $x = (x_1$ or x_2 or $x_3 \ldots)$,
- or a mixture of both, for instance: $x = (x_1$and $(x_2$ or $x_3)$and $\ldots)$.

For instance, in many real-time systems the set of functions that a system is required to provide may change over time. Mode change protocols allow currently running tasks to be deleted or changed, or new tasks to be added [22] [23]. Mode change protocols use tasks with mutually exclusive operations. Other example is the mandatory/optional task model [24] where mandatory operations have critical deadlines and optional operations can be rescindable. These are mutually inclusive operations.

2.2 Scheduling Tasks

Real-time systems must guarantee in some degree that the timing constraints of all tasks are met. The scheduling algorithm is a fundamental tool for guaranteeing such constraints. In real-time systems scheduling algorithms are executed off-line, on-line or a combination of both. In general, scheduling algorithms can be categorized as table-driven approaches or priority-driven approaches.

Table-driven approaches create a schedule off-line, which is stored in a static table to be posteriorly executed. *Priority-driven* approaches create a schedule assigning priorities such that at run-time a dispatching mechanism executes the tasks according to their priorities. Priority-driven is divided in two major schemes named fixed-priority and dynamic-priority. In *fixed-priority* the execution's order is based on off-line priority assignments, and in *dynamic-priority* the execution's order is based on on-line priority assignments (see Liu [24], Burns and Wellings [3] and Buttazo [25]).

2.2.1 Table-Driven Scheduling

In the table-driven approach, all tasks characteristics must be known a priori because the scheduling decisions are taken off-line. In effect, prior knowledge of all data allows computing one or more possible schedules, which can be stored in a table. At run-time, an initialisation phase allocates all the resources required by the tasks and the first task in the table is selected for execution. When it finishes the rest of the tasks in the table are executed sequentially and this sequence is repeated continuously. Some variations of this paradigm allows identifying free time slots that are used for implementing some mechanism to dispatch aperiodic tasks. Table-driven scheduling allows "complex and sophisticated algorithms because the schedule is computed off-line" [24]. Thus, different scheduling solutions can be tried and more than one objective can be pursued. For example, if more than one feasible plan is found, different heuristics can be applied looking to minimize other criteria such as minimizing the average response time. Examples of table-driven approaches are the clock-driven [24] and the cyclic executive [3].

Although table-driven scheduling is a highly predictable approach that solves many kinds of problems, it is mainly used for small, simple systems which do not need dynamic capabilities [26]; moreover, it is very complex to maintain and it is inflexible since any change to the tasks may require rebuilding the table [3].

2.2.2 Priority-Driven Scheduling

In the context of scheduling, a *priority* establishes an order of importance or urgency with respect to the timing constraints. In the priority-driven scheduling approach, the task

execution order is based on priorities assigned to tasks such that at all times the task with the highest priority is always executing. This paradigm has three main components:

1. A *priority assignment algorithm.* An algorithm which creates a plan by assigning priorities to tasks.

2. A *feasibility test.* An algorithm which checks conditions that are necessary and/or sufficient for a task set to be scheduled.

3. A *run-time dispatching algorithm.* An algorithm which executes the plan by dispatching the tasks in priority order.

Thus, the priority assignment establishes the order at which the tasks will be executed, and the feasibility test checks whether the timing constraints will be fulfilled; finally the dispatching algorithm schedules the tasks.

A task set ordered according to their priorities forms a *priority ordering.* We will use the terms priority assignment and priority ordering interchangeably. For a given task set, a priority ordering is *feasible* if the timing constraints of all tasks are met. A task set is *schedulable* by a specific scheduling algorithm if such algorithm produces a feasible priority ordering. When all task sets for which a feasible priority ordering exists are schedulable by an scheduling algorithm, it is said that such an algorithm is *optimal* [25].

The priority-driven approach is divided into fixed-priority and dynamic-priority scheduling. In fixed-priority scheduling static priorities are assigned and they do not change over time. In dynamic-priority scheduling priorities might change from invocation to invocation.

Fixed-priority scheduling. In the fixed-priority scheduling approach, the execution order of the tasks is based on off-line static assignment of priorities. The *Rate Monotonic Priority Assignment* (RMPA) algorithm [27] is the best known algorithm for assigning fixed-priorities. In RMPA, the priorities are assigned according to the rule "tasks with shorter periods have higher priorities". RMPA is optimal when tasks deadlines are equal to their respective periods (i.e. $T = D$). In the *Deadline Monotonic Priority Assignment* (DMPA) algorithm [28], the priorities are assigned according to the rule "tasks with shorter

deadlines have higher priorities". DMPA is optimal when tasks deadlines do not exceed their respective periods (i.e. $D \leq T$). DMPA subsumes RMPA. Optimality implies that if a feasible priority ordering over a task set exists, then DMPA is also feasible. On some process models where DMPA is not optimal, more complex algorithms for finding feasible priority assignments are utilised such that the one presented by Audsley in [29]. This *Swap algorithm* starts with an infeasible priority assignment and performs swaps of priorities until a feasible one is found.

Dynamic-priority scheduling. In the dynamic-priority scheduling approach, the execution order of the tasks is based on dynamic assignment of priorities, such that at run-time the task with the highest priority is always executing. The *Earliest Deadline First* (EDF) algorithm [27] forms the base for all dynamic-priority scheduling in real-time systems. In EDF, the priorities are assigned on-line according to the rule "the ready task with the nearest deadline has the highest priority". EDF is an optimal dynamic preemptive algorithm on single processor systems that can theoretically utilise all the processor capacity; for the simple process model defined in [27], all deadlines (with $D = T$) will be met under EDF if $\sum_{i=1}^{N}(C_i/T_i) \leq 1$. In addition, other dynamic-priority algorithms are also found in the literature such as the Least Laxity First algorithm (LLF) where "the task with the smallest laxity has the higher priority" [30]; the Value-Density algorithm where the priorities are assigned according to the ratio $\frac{V(t)}{C}$ (i.e. value at time t divided by computation time) [31]. In the Maximum Urgency First algorithm (MUF) the task with the maximum urgency has the higher priority; in this approach, urgency is defined as a tuple consisting of two fixed priorities and a dynamic priority; one of the fixed priorities has precedence over the dynamic priority and the dynamic one has precedence over the other fixed priority; at run-time, the algorithm described in [32] computes the maximum urgency as a function of the three priorities.

2.2.3 Discussion

Depending upon the application, scheduling schemes exhibit different advantages and disadvantages which are debated in the community (see Locke [33], Xu and Parnas [10], Buttazzo [17]). In terms of predictability and flexibility, fixed-priority driven scheduling is localized in the middle between table driven (the most predictable) and dynamic-priority

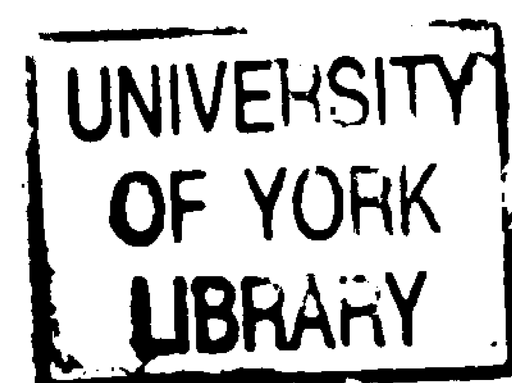
driven (the most flexible) schemes. This particularity is appreciated by many government agencies and system integrators who recommend it as the method of choice [10]. As a consequence, the majority of commercial real time operating systems and languages and standards support fixed-priorities only. The main focus of this thesis is fixed-priority scheduling in real-time systems which is now presented on detail.

2.3 Fixed-Priority Scheduling

Liu and Layland proved in [27] a set of fundamental results known as the rate monotonic analysis that form the base of the fixed-priority scheduling theory. Rate Monotonic imposes some restrictions to the task model that have been posteriorly relaxed for allowing tasks to have deadlines less than their periods [34], deadlines greater than their periods [35], arbitrary offsets [36], precedence constraints [19], variations in computation time and period [19], shared resources [37], etc. The literature in this area is large and for such reason, only the fundamental concepts will be reviewed starting from the basic rate-monotonic analysis.

Basic task model. A task set S consists on N independent preemptive tasks to be scheduled in a single processor system where:

- Each task consists of an infinite number of invocation requests, each one separated by a minimum time T. For a periodic task it defines its period and for a sporadic task it defines its minimal inter-arrival time.
- Tasks have deadlines D relative to the actual release such that if the task is invoked at time t, it should have finished by $t + D$.
- The worst-case computation time C is known for each task (with $C < D$) and during this time the task does not suspend itself.
- All tasks share a common release r_0.
- Tasks have a priority P where P_{max} is the highest and 1 is the lowest priority one; without loss of generality we assume that two tasks do not share the same priority.


UNIVERSITY OF YORK LIBRARY

- All tasks are non-rescindable; e.g. they cannot be aborted or skipped.

All periodic tasks are assumed to be released together at time zero. The tasks will be released together again at their hyperperiod. The hyperperiod is the least common multiple of the periods of the tasks and it is denoted by H such that $H = \text{lcm}\{T_j\} \ \forall j \in S$.

2.3.1 Rate-Monotonic Analysis

The Rate-Monotonic analysis applies to a set S of N periodic tasks with the above basic task model and where all tasks have a deadline equal to their period ($D = T$). Liu and Layland proved in [27] the following results.

Theorem 2.3.1. *The longest response time of a task occurs when it is invoked simultaneously with all higher priority tasks.*

The time when all tasks are invoked simultaneously is called a *critical instant*. A critical instant occurs when all tasks are released at time zero.

Theorem 2.3.2. *A fixed-priority assignment is feasible provided that the task deadlines at the critical instant are met.*

Thus, for a fixed-priority assignment to be feasible, only the first deadline of each task must be met when all tasks are scheduled at time zero.

Liu and Layland proposed the Rate-Monotonic Priority Assignment (RMPA) algorithm, which consists on assigning fixed-priorities assigned inversely proportional to the tasks periods. The RMPA algorithm is optimal among all static priority assignment algorithms. In addition, the Utilisation-Bound test for checking the feasibility of the assignment is also introduced in [27].

Definition 2.3.3 (Rate-Monotonic Priority Assignment). *A task set S is in rate-monotonic priority ordering if*

$$P_a > P_b \Leftrightarrow T_a < T_b$$

for any two tasks a, b in S.

Theorem 2.3.4 (Utilisation-Bound test). *For a set S of N tasks, the Rate Monotonic Priority Assignment yields a feasible priority ordering if*

$$U < N(2^{1/N} - 1)$$

where $U = \sum_{j=1}^{N} C_j/T_j$ is known as the utilisation factor of the task set.

The bound converges to [27]:

$$\lim_{N\to\infty}(N(2^{1/N} - 1)) = \ln 2 \approx 0.69314$$

However, experimental results with randomly generated task sets show that this bound can be as higher as 0.85 [3]. The utilisation-bound tests is *sufficient* (i.e. any task set passing the test will be schedulable) but *not necessary* (i.e. failing the test does not imply it is unschedulable).

The feasibility analysis of the RMPA algorithm can also be performed using a sufficient but not necessary test called the Hyperbolic bound [38]. This test has the same complexity as the above test but it is less pessimistic allowing to accept task sets that would be rejected using the Utilisation-bound approach.

Theorem 2.3.5 (Hyperbolic-Bound test). *For a set S of N tasks, the Rate Monotonic Priority Assignment yields a feasible priority ordering if*

$$\prod_{j=1}^{N}(U_j + 1) \leq 2$$

where $U_j = C_j/T_j$ is the utilisation factor of task j.

With the Rate-Monotonic analysis, Liu and Layland founded the basis of fixed-priority scheduling in real-time systems by proving that giving higher priorities to tasks with shorter periods, scheduling problems with deadlines can be solved optimally. Over the years, the assumptions of strict periodic tasks and $D = T$ were relaxed causing the next milestone in the area: the Deadline-Monotonic approach [34].

2.3.2 Deadline-Monotonic Analysis

The Deadline-Monotonic Analysis extends the basic task model allowing tasks with deadlines less than or equal to the periods, and sporadic tasks. The approach consists of the

Deadline-Monotonic Priority Assignment (DMPA) for assigning fixed-priorities and the Response-Time test for checking the feasibility of the assignment. The Response-Time test is based on the exact analysis for finding the worst-case response time of a task when it is released at its critical instant assuming that $D \leq T$. This assumption permits to incorporate sporadic tasks without alteration to the task model [34]. The next results are proved in [28], [34], [39], [35].

Leung and Whitehead [28] proved that the RMPA is no longer optimal when $D \leq T$. However, assigning higher priorities to tasks with shorter deadlines is still optimal among fixed-priority algorithms. This assignment policy is known as the *Deadline-Monotonic Priority Assignment* (DMPA) algorithm.

Definition 2.3.6 (Deadline-Monotonic Priority Assignment). *A task set S is in deadline-monotonic priority ordering if*

$$P_a > P_b \Leftrightarrow D_a < D_b$$

for any two tasks a, b in S.

The *Response-Time test* (or completion-time test) is a sufficient and necessary (i.e. the set is unschedulable if it fails the test) schedulability test based on the response-time equation defined [34], [39]:

Definition 2.3.7 (Response-Time equation). *The Response-Time equation computes the worst-case response time R_j of any task j when it is released at its critical instant:*

$$\begin{aligned} W_j^0 &= C_j \\ W_j^{n+1} &= C_j + \sum_{i \in hp(j)} \left\lceil \frac{W_j^n}{T_i} \right\rceil C_i \end{aligned} \tag{2.3.1}$$

where $i \in hp(j)$ is the set of higher priority processes than j. This recurrence equation finishes when either $W_j^{n+1} = W_j^n$ or $W_j^{n+1} > D_j$. If $W_j^{n+1} = W_j^n$, then the response time R_j is W_j^n; otherwise, the response time of task j will be greater that its deadline. The convergence of the response-time equation is guaranteed [35] if $U \leq 1$.

Theorem 2.3.8 (Response-Time test). *Given a task set S where $D_j \leq T_j \; \forall j$, a fixed-priority ordering is feasible if and only if $R_j \leq D_j$ for all j.*

In the Response-Time test note that:

- It is independent of the priority assignment; i.e. it can be used to determine the feasibility of any fixed-priority assignment.
- The condition $R_j \leq D_j$ must be verified for each task.
- The test has pseudo-polynomial complexity because the response-time equation is pseudo-polynomial; i.e. the number of steps to compute the response time depends not only on the number of tasks but also on the values of C, T and D, and hence the number of steps cannot be determined in advance.

Motivated by the necessity of fulfilling more realistic system requirements, the analysis based on fixed-priorities has been extended to include dependent tasks, release jitter, arbitrary deadlines, aperiodic tasks and more. Some of those extensions are reviewed next.

2.3.3 Dependent Tasks

Two tasks are dependent if the activities conducted by one depend on the initiation or termination of activities conducted by the other task. Dependent tasks could involve inter-task communication or synchronization, or precedence constraints.

Tasks Synchronisation

In practice, tasks interact as for example when accessing mutually exclusive resources (e.g. shared protected data). Task synchronization primitives can be used to control access to such critical resources. However, when applying synchronisation primitives (e.g. semaphores) deadlocks or priority inversion phenomenons can occur [3]. *Deadlock* is a situation wherein two or more tasks competing for a resource are waiting for the other to finish but it never does, and then they wait forever. *Priority inversion* is produced when a higher priority task is prevented from executing by a lower priority one. Priority inversion leads to unbounded blocking times and therefore deadlines cannot be guaranteed. These problems can be controlled by the set of priority inheritance protocols [37].

Priority Inheritance Protocol. Under the *Priority Inheritance Protocol*, when a high priority task is blocked by a lower priority one, the lower priority task inherits the priority of the higher priority one, thus preventing middle priority tasks from executing. However, this protocol suffers from several problems such as tasks having transitive blocking and the protocol does not prevent deadlocks [37].

Priority Ceiling Protocol. In the *Priority Ceiling Protocol* (PCP), each resource is assigned a priority ceiling, which is a priority equal to the highest priority of any task which may lock the resource [37]. At run-time, there exists a system ceiling that is the highest priority ceiling of any currently locked semaphore. In addition to its static priority, a task has also a dynamic priority that is the maximum of its static priority and the one that it inherits due to blocking of higher priority tasks. Thus, during the protocol operation, a given task can only lock the resource when its dynamic priority is strictly higher than the system ceiling. If so, its priority is temporarily raised to the system ceiling such that no task that may lock the resource is able to get scheduled. Let B_a be the duration of the longest critical section executed by a task of lower priority than a task a, guarded by a semaphore whose priority ceiling is greater than or equal to the priority of a. Sha et al. [37] proved that:

1. Task a can be blocked by a lower priority task for at most B_a time units.
2. The priority ceiling protocol is deadlock free.
3. Transitive blocking is prevented.

In addition, the protocol provides mutual exclusion access to the resource. The main problem of PCP is its implementation cost and run-time overhead.

Immediate Priority Ceiling Protocol. In the *Immediate Priority Ceiling Protocol* (IPCP), when a task access a resource, IPCP raises immediately its priority up to ceiling of the resource and lowers its priority only when the task leaves the resource [37]. IPCP also prevents deadlocks, transitive blocking and provides mutual exclusion access to the resource. IPCP is easier to implement and has lower run-time overhead than the PCP. IPCP is the currently policy in programming languages as Real-Time Java and Ada, and in standards as POSIX [3].

Response-Time Equation with Blocking. Let K be the number of critical sections (resources) under PCP or IPCP, the maximum time B_j a task j can be blocked is equal to the execution time $C(k)$ of the longest critical section k in any of the lower priority tasks that are accessed by higher priority tasks [3]:

$$B_j = \max_{k=1}^{K}\{\Upsilon(k,j)C(k)\}$$

where

- $\Upsilon(k,j) = 1$: if k is used by at least one task with a priority lower than P_j, and at least one task with a priority greater or equal to P_j.
- $\Upsilon(k,j) = 0$: otherwise.

In order to take into account task synchronisation, the response time equation 2.3.1 is updated as follows [3]:

$$\begin{aligned} W_j^0 &= C_j \\ W_j^{n+1} &= C_j + B_j + \sum_{i \in hp(j)} \left\lceil \frac{W_j^n}{T_i} \right\rceil C_i \end{aligned} \qquad (2.3.2)$$

Note that maximum blocking will not occur in some scenarios. For example, when a set of periodic tasks with the same period are released at the same time, then no preemption will take place and hence blocking will not occur. Consequently, the feasibility test with the response-time computed with equation 2.3.2 is pessimistic [3].

Precedence Constraints

Another kind of dependence between tasks is when a task b depends on some results of a task a (e.g. task b cannot start until task a has completed). It this case it is said that "a precedes b" and we denote it as $a \rightarrow b$.

Precedence relationships are important hard constraints in many systems where tasks involved in an activity are dependent on other ones. Simple precedence relationships can

be implemented for tasks with the same period by modifying tasks priorities. In effect, in priority scheduling, in order to ensure precedence of a over b (with $T_a = T_b$) is necessary to consider their task release time r_a and r_b such that [16]:

1. if $r_a \leq r_b$ then P_a must be greater than P_b, or
2. if $r_a < r_b$ then P_a is must be equal to or greater than P_b.

In [19], a simple transformation technique for assigning priorities to tasks with precedence requirements is explained. The technique applies to tasks that share the same period and consists of modifying their deadlines such that the priorities are enforced to the order required when DMPA is utilised.

2.3.4 Release Jitter

Release jitter J is the worst-case delay between the arrival time and the release time of a task [19]. In the above analysis, it is assumed that all tasks are periodic with periods which are exact multiples of the scheduler period. Thus, they can be released as soon as they arrive and hence, they do not suffer release jitter. However, there are situations where tasks could suffer release jitter. For example: (i) periodic tasks non-synchronized with the scheduler period and (ii) sporadic tasks which are not released as soon as the event on which they are waiting has occurred (see [3]).

To take into account task release jitter, the response-time equation 2.3.2 is updated as follows [3]:

$$
\begin{aligned}
W_j^0 &= C_j \\
W_j^{n+1} &= C_j + B_j + \sum_{i \in hp(j)} \left\lceil \frac{W_j^n + J_i}{T_i} \right\rceil C_i
\end{aligned} \qquad (2.3.3)
$$

where J_i is the release jitter of higher priority tasks than j. Finally, the response-time is calculates as $R_j = W_j^{n+1} + J_j$.

2.3.5 Arbitrary Deadlines

Arbitrary deadlines refers to tasks with $D \leq T$ or $D > T$; i.e. deadlines can be greater than their periods. When $D > T$, the critical instant assumption of a task meeting its first deadline will meet all the successive deadlines does not hold. Consequently, neither the DMPA nor RMPA are optimal anymore even if a feasible priority exist. In this case, the extended response-time equation provides an exact analysis for computing the worst-case response time. In addition, a feasible priority ordering can be determined by a priority swapping algorithm [3].

Extended Response-Time Equation

In order to compute the worst-case response time of a task j when $D_j > T_j$, a number of releases must be considered. In this case, there are potentially overlapping releases of different instances of j and then the analysis given in [35] must be utilised. Assuming that an overlapping release will be delayed until any previous one has completed, for any potentially overlapping release a separate window $W(q)$ is defined, where $q = 0, 1, 2, \ldots$ identifies a particular window of time. Thus, the response time equation 2.3.3 is extended as follows [3]:

$$
\begin{aligned}
W_j^0(q) &= C_j \\
W_j^{n+1}(q) &= (q+1)C_j + B_j + \sum_{i \in hp(j)} \left\lceil \frac{W_j^n(q) + J_j}{T_i} \right\rceil C_i
\end{aligned}
\tag{2.3.4}
$$

For each q, a stable value that stops the loop can be found (i.e. $W_j^{n+1}(q) = W_j^n(q)$) and then the response time for each q is given by

$$R_j(q) = W_j^{n+1}(q) - qT_j + J_j$$

Thus, different releases are calculated until a q is found such that $R_j(q) \leq T_j$. Finally, the worst-case response time is the maximum value of all $R_j(q)$ computed:

$$R_j = \max \{R_j(q) \text{ for } q = 0, 1, 2, \ldots\}$$

Algorithm 1 Swapping Algorithm

```
Require: α is a priority ordering
 1: ok := false
 2: for j := N − 1 to 0 do
 3:    for next := j to 0 do
 4:       Swap(α,j,next)
 5:       if F(α, j) = true then
 6:          ok := true
 7:          exit inner loop
 8:       end if
 9:    end for
10:    if not ok then
11:       exit loop {infeasible}
12:    end if
13: end for
Ensure: ok is true
```

Assignment of Priorities

When tasks have arbitrary deadlines, DMPA is not optimal [3]. In this case the the Swapping algorithm may be used. This algorithm is based on the following theorem proved in [29]:

Theorem 2.3.9. *If a task j is feasible at the lowest priority level, then if a feasible priority ordering exists for the complete task set, an ordering exists with j assigned at this lowest priority level.*

The *Swapping algorithm* receives as input an infeasible priority assignment and finds a feasible one (if such priority assignment exists) by swapping pairs of task priorities. Assuming a priority ordering α where index $\alpha[0]$ corresponds to the highest priority task and $\alpha[N-1]$ to the lowest priority one, the swapping algorithm (see algorithm 1) tries to accommodate feasible tasks starting from the $N-1$ to the 0 position. The function $F(\alpha, j)$ returns true when a task at j^{th} position is feasible. Note that this function may be implemented using any of the above recurrent response-time equations. When a task in the $N-1$ position is found feasible it is fixed in that position, and then the algorithm tries to accommodate another task in the $N-2$ position and so on, until it accommodates successfully all tasks. However, when a task in a jth position is infeasible, it is swapped with the $(j-1)^{th}$ task, and if it is also infeasible with the $(j-2)^{th}$ and so on; when

no task can be fitted then there does not exist a feasible ordering and the algorithm ends unsuccessfully.

When the feasibility test $F(\alpha, j)$ is exact (i.e. necessary and sufficient), the priority ordering is optimal. Otherwise the priority ordering is as good as the quality of the test [19].

2.3.6 Aperiodic Tasks

Aperiodic tasks have random release times and therefore a deterministic worst-case analysis cannot be done; consequently, only aperiodic tasks with soft deadlines can be handled. In FPS, scheduling problems where in addition to hard tasks, soft aperiodic tasks have to be included in the plan, incorporate at least two objectives: (1) to guarantee that all hard deadlines and (2) to minimize the response time of the aperiodic tasks. Thus, the problem is how to schedule aperiodic tasks without causing that hard ones to miss their deadlines. Server algorithms such as the Sporadic Server [40], Extended Priority Exchange Server [41] or slack stealing algorithms such as Dual Priority Scheduling [42] accomplish these two objectives. Of special interest is the Sporadic Server because for FPS analysis purposes, it can be treated as a single periodic task, and consequently it can be incorporated into the FPS analysis.

Conceptually, a server is a periodic task with a *capacity* (execution time) and a *replenish time* (period). The capacity is set to the maximum possible such that the task set and the server remain schedulable. When the server is executed (*released*), its capacity is utilised to service aperiodic tasks. These tasks are stored in a queue and if it is nonempty when the server is released, the tasks are executed. At the beginning of specific times (e.g each period), the capacity is *replenished* (a computation time is assigned). When the server executes a task, it consumes its capacity and it becomes exhausted when the capacity is zero. The server is suspended when either its capacity is exhausted or the queue is empty. The simplest server is the Polling Server which works as above and has a main drawback: if the server is released when the queue is empty, the server will be suspended and its capacity will be lost. Therefore, the bandwidth-preserving servers algorithms have been developed to preserve its capacity when they are released [24].

The *Sporadic Server* is a the bandwidth-preserving server. When the queue becomes

empty, the sporadic server preserves its capacity deferring its assigned time slot. If a task arrives and the server has capacity, the task will be executed. The sporadic server replenishes just the capacity consumed at sporadic times (variable times). In contrast, other servers such as the Deferrable Server replenishes its whole capacity at the start of each period (fixed times) [24].

2.3.7 Weakly-Hard Tasks

A weakly-hard real-time system is a system that tolerates occasional losses of deadlines if the distribution of its met and lost deadlines is precisely bounded within a window of time [5]. Thus, while in the above analysis task timing constraints are represented by a single deadline, in weakly-hard systems timing task constraints have to be expressed by more complex artifacts. It implies the development of new priority assignment algorithms and feasibility tests.

A *weakly-hard constraint* is an exact description in a window of time of a required pattern of miss/met deadlines that in the worst-case a task must fulfil. Given a task system, a pattern of miss/met deadlines of any task in a window of time can be computed either by simulation or analytically. This pattern can be represented by a binary sequence called μ-pattern [18], where for each task release: 1 represents a deadline met and 0 represents a deadline missed.

μ-pattern

Definition 2.3.10 (μ-pattern [18]). *A μ-pattern is the worst-case pattern of met and lost deadlines of a task j and is defined as*

$$\mu_j(k) = \begin{cases} 1 & \text{if } R_j(k) \le D_j \\ 0 & \text{otherwise} \end{cases}$$

where k is the kth activation of j.

Thus, a μ-pattern is a sequence of $1/0$ indicating that at any particular release a task either met or lost its deadline. For instance, the μ-pattern of a task that never miss a deadline is a sequence of 1s; a task that miss just one deadline and it occurs only at critical

instant has a μ-pattern $\mu = \overbrace{01111\ldots1}^{L}$ of length L, where the length is the number of activations during its hyperperiod.

Computing the μ-pattern of a task requires computing the worst-case response time $R_j(k)$ at each invocation k, within the first hyperperiod at its priority level. This can be done by either simulation or analysis.

In [18], several algorithms to compute analytically a μ-pattern are described. First, the *worst-case finalization time* $F_j(k)$, at invocation k, within the first hyperperiod is computed. $F_j(k)$ is the sum of three factors:

1. the computation time of the first k invocations
2. the time that higher priority tasks have been computing before $F_j(k)$
3. the time the processor has not been used by higher priority tasks

The algorithms for computing $F_j(k)$ are rather complex and can be consulted in [18]. Once that $F_j(k)$ is computed, the worst-case response time is

$$R_j(k) = F_j(k) - r_j(k)$$

where $r_j(k)$ is the kth release. Finally, for each release k its worst-case response time can be calculated and the μ-pattern is obtained applying definition 2.3.10.

Weakly-Hard Constraints

In some systems the effect of missed deadlines can be different depending on whether they are missed consecutively or non-consecutively; thus, the tolerance to missed deadlines is established within a window of m consecutive invocations of a task. In this context, four types of weakly-hard constraints are defined in [5].

Definition 2.3.11 (Weakly-Hard Constraints). *Let m ($m \geq 1$) be any window of consecutive invocations of a task, and n ($0 \leq n \leq m$) the number of invocations in such window:*

- (n, m): *a task meets any n in m deadlines if there are at least n invocations in any order that meet the deadline.*

- $\langle n, m \rangle$: *a task meets row* n *in* m *deadlines if there are at least* n *consecutive invocations that meet the deadline.*

- $(\overline{n, m})$: *a task misses any* n *in* m *deadlines if no more than* n *deadlines are missed.*

- $\langle \overline{n, m} \rangle$: *a task misses row* n *in* m *deadlines if it is never the case that* n *consecutive invocations miss their deadline.*

The first two weakly-hard constraints express task sensitivity to the consecutiveness of deadlines met, and the last two to the consecutiveness of deadlines missed. Note that a task that must met all their deadlines (i.e. a hard task) is a particular case of a weakly-hard task. This task is called a *strong weakly-hard task* and it has a μ-pattern of 1s.

Feasibility Test

A weakly-hard real-time system is schedulable if it can be guaranteed that at run-time all weakly-hard temporal constraints are always satisfied. Given a task set ordered by priorities, the feasibility analysis requires:

1. Computing the worst-case μ-pattern of each task at its corresponding priority level.

2. Checking if the μ-pattern satisfies the weakly-hard constraint.

Naturally, a task set can have tasks with either hard deadlines or weakly-hard constraints. All tasks must be tested with the response-time equation but only the infeasible ones with weakly-hard constraints require the two above steps. A function $W(\alpha, k)$ that returns true when the jth task in the priority ordering α meets the weakly-hard constraints is described in [18]. Function $W(\alpha, k)$ starts checking the strict feasibility of the jth task according to the FPS feasibility test;

- if feasible , it will not miss any deadline and therefore it is also weakly-hard feasible;

- if infeasible and with a strict hard deadline, it is infeasible;

- otherwise, the task miss some deadlines but it has a weakly-hard constraint and then the μ-pattern of task j is computed and it is checked against its constraint. True is returned if the μ-pattern fulfills the weakly-hard constraint.

Priority Assignment

Neither DMPA nor RMPA priority assignment algorithms are optimal for task sets with weakly-hard constraints. Fortunately, the Swapping algorithm (see page 34) may be used to find a feasible priority assignment by substituting the boolean function $F(\alpha, j)$ with $W(\alpha, j)$ [5].

2.4 Summary

On Fixed Priority Scheduling (FPS), the assignment of priorities and the feasibility analysis are the two main areas of work. The former establishes the order in which the tasks will be executed and the latter checks whether the time constraints will be fulfilled. RMPA, DMPA and the Swapping algorithm are the most known priority assignment algorithms. On the other hand, the utilisation-based test and the response-time analysis test (with all its variations for computing the worst-case response time) are the most known feasibility tests. In addition, complex feasibility tests for weakly-hard constraints exist. All these algorithms allow solving problems with complex task models when only timing constraints are involved. However, when the objective includes other quality requirements, FPS exhibits some weaknesses that are discussed in the next chapter.

Chapter 3

A Review On Scheduling with QoS

3.1 Introduction

In real-time systems researchers have been investigating scheduling mechanisms for supporting systems where, in addition to timing constraints, their performance with respect to additional QoS requirements must be improved. This type of situation can occur when the worst-case resource requirements of all or some running tasks cannot be simultaneously met.

In some scenarios, systems are split up into tasks that are designed and tested separately. When the system is integrated, it may be the case that the resource requirements of tasks cannot be simultaneously met due to task contention. In this case, the scheduling mechanism must allocate greater resources to more important tasks. This allocation can be off-line or on-line. In other scenarios, worst-case resource requirements cannot be determined in advance and then, it is the responsibility of the scheduler to allocate the resources to the most important tasks at run-time.

Thus, allocating greater resources to important tasks is essential to improve the QoS of a system. In real-time systems, the concept of importance among tasks is usually defined in terms of utility functions. In this chapter, we briefly make a review of some scheduling mechanisms for improving the QoS measured in terms of utility. In addition, a concise review of how the system requirements are mapped from the specification to the design is also presented. It conveys the idea that scheduling problems in real-time systems are

inherently multicriteria and include both timeliness and QoS requirements.

3.2 From Requirements to Scheduling Problems

Real-time systems are not different from other computer applications; they have a life cycle divided in different phases starting from the requirements specification to maintenance. During the specification of requirements QoS requirements are defined. During the design phase, the system is structured as a set of activities that achieve one or more requirements and then a scheduling problem is formulated; further, the scheduling problem must be solved.

3.2.1 Requirements

The requirements specification phase is the period of time during which the requirements for the software product are defined and documented [43]. Requirements are usually separated into *Functional* and *Non-Functional.* Functional requirements describe “what” the system should do (e.g. a car air bag must protect the driver in the case of a collision), while any other requirement is considered non-functional as, for example, “how well” a functional requirement shall be accomplished (e.g. the car air bag must be always triggered within 30 milliseconds after the crash sensor detects a collision at high speed). In general, any requirement not considered functional is dichotomously considered non-functional [44].

The IEEE Standard Glossary of Software Engineering Terminology [43] distinguishes on the one hand, functional requirements, and on the other hand design, implementation, interface and performance requirements. According to Gilb [45], functional requirements are of less use when designing because “it is usually fairly self-evident what functionality is required. It is actually the other categories that determine the choice of design”. Therefore, non-functional requirements are determinant during the design phase and consequently, in the definition of the scheduling problem.

Non-Functional Requirements

Several classifications of requirements can be found in the literature. The IEEE standard 830-1998 on Software Requirements Specification [6] sub-classifies non-functional (*NF*) requirements into external interface, performance, attributes and design constraints, where attributes are qualities such as reliability, availability, portability, maintainability, etc. It is noted that requirements can have different levels of importance or stability because typically not all requirements are equally important. "Some requirements may be essential, especially for life-critical applications, while others may be desirable" [6]. Naturally, it implies the existence of preferences to achieve more important requirements first with respect to the least important ones.

In [44], it is noted that some requirements can be objective or subjective. When measurable, the requirements can be expressed in a quantitative form that is precise, unambiguous and verifiable. Nevertheless, sometimes it is also wanted to state the requirements in a qualitative form, as for example when stating achievement goals ("the system shall achieve good performance"). Such goals can be verified only after deployment, or with the help of prototypes such that stakeholders can subjectively judge whether or not the requirement is fulfilled. Alternatively, they can be "verified indirectly by decomposing it or by deriving metrics that (hopefully) are highly correlated with the given requirement" [44].

In general, NF requirements have some fundamental principles [45], [46], [44]:

- They are either quantitative or qualitative, and testable for presence in their implementation. If a requirement is quantitative, it is quantifiable in at least one scale of measure.
- Some are binary (hard) and some are specified in a scale of measure (soft).
- They reflect objective and/or subjective importances.
- They can be in conflict with others for shared resources and therefore, priorities must be assigned to resolve the conflicts.
- They change as the necessities, values and priorities change during the life cycle.

In hard real-time systems, some quality requirements such as reliability and safety have higher importance with respect to other requirements. Indeed, they must be considered as hard, and they must be specified in a quantitative form to be precise, unambiguous and verifiable. Other quality requirements (e.g. portability and maintainability) and performance requirements (e.g. throughput) can be considered secondary. These requirements could be hard or soft, and could be expressed quantitatively or qualitatively. For example, consider a requirement enunciated as a goal: "the system shall achieve good performance". It is a soft requirement expressed in a qualitative form. It can be verified only after deployment, or with the help of prototypes to judge whether or not the requirement is fulfilled [47].

Note that hard or soft requirements are not necessarily related directly to the concepts of hard or soft real-time systems. For example, consider a requirement enunciated as: "the system has to execute in the X, Y and Z hardware configurations" (e.g. a piece of software for a mobile phone). It is a mandatory portability requirement which is satisfied or not satisfied (i.e. hard) for a soft real-time system.

On summary, NF requirements form a multidimensional set of interrelated necessities that vary with time. NF requirements must be conveyed throughout all stages of the life cycle development; first to the design phase, after to the implementation and so on. At some point of the design, the system is structured as concurrent tasks that posteriorly will be scheduled. Thus, the formulation of the scheduling problem depends on how the requirements are mapped to the design.

3.2.2 Design

Design refers to the process of defining the architecture, components and other characteristics of a system intended to satisfy the requirements [43]. Design can be divided in two stages:

- *Architectural design.* In this stage, the system is structured into its constituent components. Choices are made about hardware and software. A systematic breakdown of the system into smaller parts is performed and each part is described using a method of documentation [3].

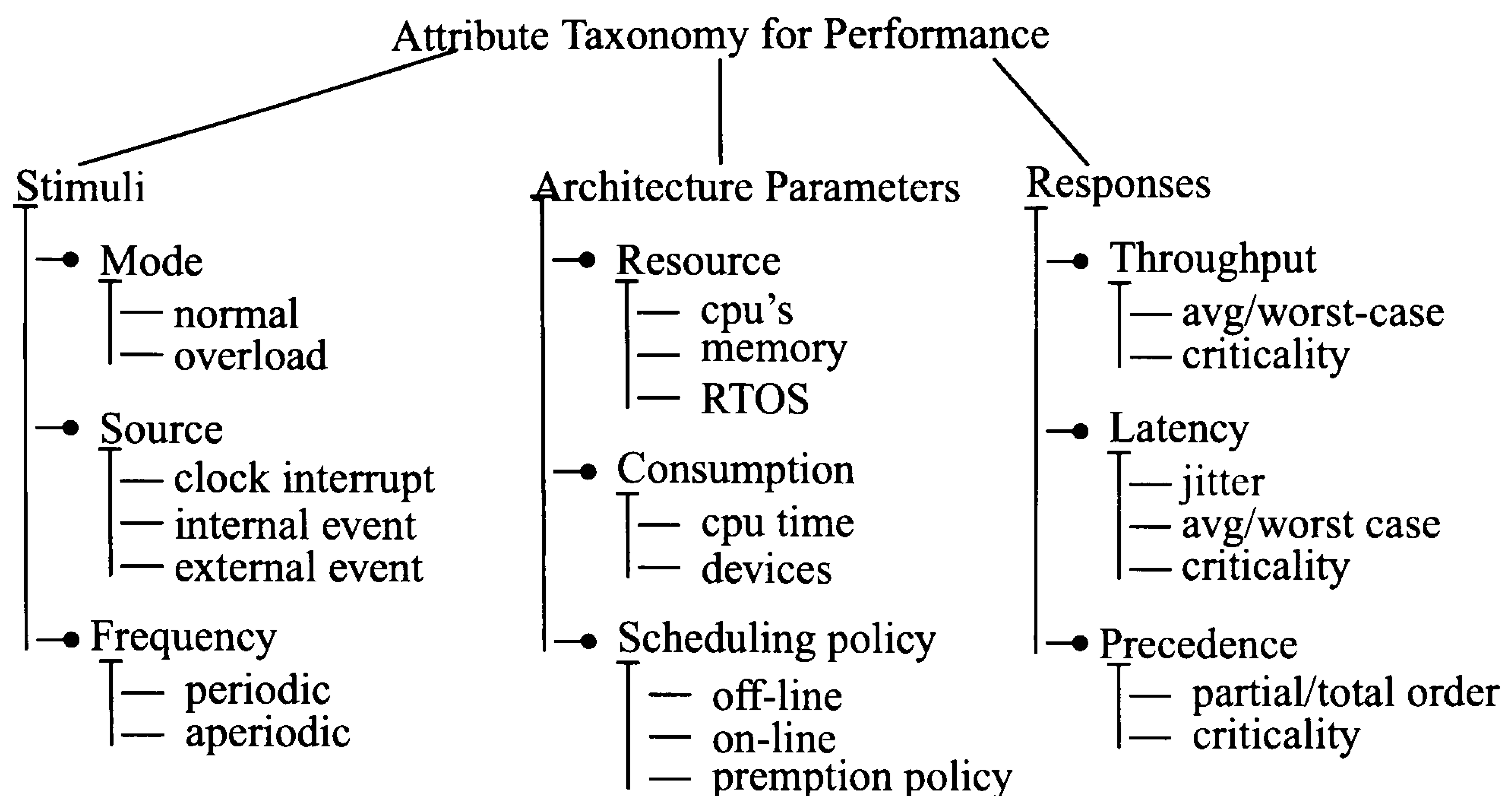


Figure 3.1: Attribute Taxonomy for Performance

- *Detailed design.* This stage incorporates some crucial objectives such as structuring the system in concurrent tasks, supporting the development of reusable components through information hiding, and analysing the performance of the design to determine its real-time properties [3]. In this stage the scheduling problem is defined.

Naturally, the architecture and detailed design are not sequential activities but they can be considered as coroutines. NF requirements are inputs for the architectural design. The architectural design makes decisions that must satisfy the NF requirements, while it deals with sensitive points (i.e. decisions that affect one of the requirements, if changed) and tradeoff points (i.e. decisions that affect more than one requirement, if changed) [48].

In order to elicit architectural information relevant to NF requirements, taxonomies of the requirements can be depicted. Figure 3.1 shows a modified version of a taxonomy presented in [48]. A performance requirement is broken down into their attributes. Three categories of attributes are identified:

- *Stimuli* are the events that cause the architecture to respond or change.
- *Architectural parameters* are components that will affect a response.
- *Responses* are quantities measurable or observable related with the requirement.

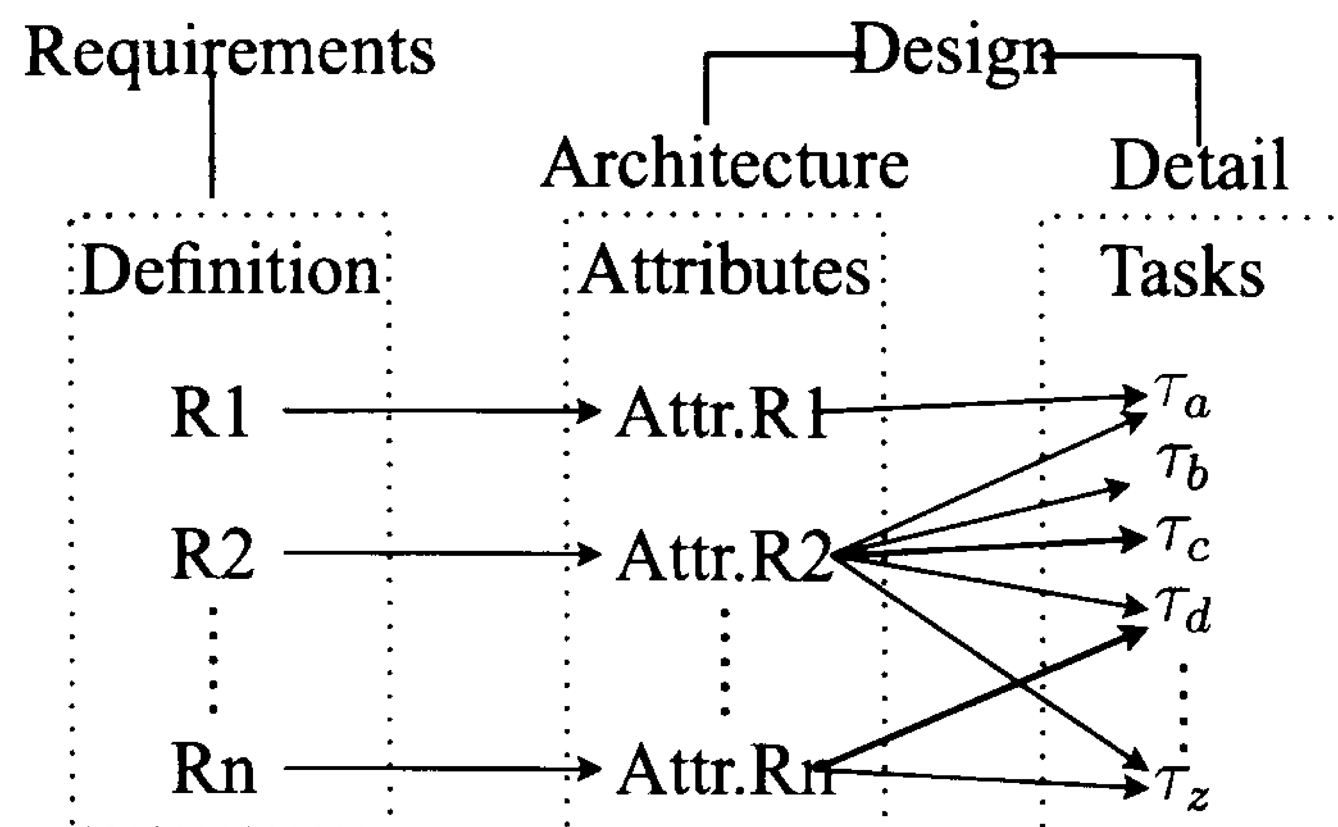


Figure 3.2: From Requirements to Tasks

Taxonomies are helpful to understand how the requirements can be mapped to the tasks and other components during the detailed design stage. For instance, one aspect of the response such as the *latency* (i.e. the length of time a task takes to respond to an event) is a function of the stimuli (e.g. external events) and the architectural parameters (e.g. cpu's, cpu time consumption).

Outcomes of the architectural design form the inputs for the detailed design. During this phase, the system is structured as a set of concurrent tasks. Clearly, a single requirement can be mapped into one or more tasks (see Figure 3.2). A task is modelled according to the architecture defined such that it achieves all or part of one or more requirements when it completes; this achievement constitutes the benefit or utility that the task will give to the system. Naturally, it is desirable that the benefit can be detected quantitatively but, as it was argued in 3.2.1, some requirements are expressed qualitatively such that they can only be verified after deployment or indirectly by correlated metrics.

3.2.3 The Scheduling Problem

Understandably, the tasks will provide the maximum benefit if sufficient resources exist. However, these resources are normally scarce and the tasks must compete for them, resulting in a scheduling problem. Problems where the optimality is expressed by a single criterion are called single criterion problems. In a *multicriteria scheduling problem* the optimality is expressed using two or more criteria.

Evidently, solving scheduling problems with multiple criteria is not simple. Some of

the difficulties refer to the complexity involved in their solution and in the meaning of optimality. The theory shows that a scheduling problem for an arbitrary set of periodic tasks is NP-hard. Thus, adding a second criterion increases its complexity and hence, it also is a NP-hard problem. On the other hand, maximising one criterion could decrease the value of the other criteria in the problem. Scheduling solutions to problems where it is not possible to increase the value of one objective without decreasing the value of the other are called Pareto-Optimal solutions [2].

It is accepted [12] that when solving multicriteria problems, no simple algorithm can be applied and instead, the use of a combination of approaches such as procedures based on dispatching rules, branch and bound techniques, and local search techniques must be implemented. With respect to optimality, if tradeoffs between criteria can be established, optimal solutions can be defined. Otherwise, the concept of pareto-optimality has to be applied.

In the scheduling literature, NF requirements are usually referred as QoS requirements. In a problem where timing and QoS requirements must be satisfied, the scheduling mechanism must optimize the QoS requirements without violating the timing constraints. During the design, QoS requirements are mapped to the task set that constitute the system. This mapping will be uniform if all task are responsible for achieving all QoS requirements. Clearly it is not always true. Instead, individual tasks or subsets of tasks could be responsible for achieving specific goals. In this context, QoS requirements can be considered as local or global with respect to tasks. Consider the following examples:

Local QoS: Consider a control system composed by several tasks where a specific one may implement the control algorithm. Control algorithms are sensitive to *output jitter* (i.e. the variation in the inter-completion times of successive instances of the same task). Its large variation could degrade the control performance or even cause instability of the system affecting other QoS requirements such as efficiency, reliability and even safety. Clearly, the scheduling algorithm must allocate more resources to this highly important task.

Global QoS: Consider a preemptive priority-based system with maximum requirements of performance. A timely but slow or inefficient system cannot be considered viable.

Worst-case and best-case performance are reciprocally related to the worst-case and best-case execution times [49]. The system influence the task execution time due to the context switching. Context switching wastes processor time and disrupt task execution affecting some performance attributes such as *latency* (i.e. the length of time it takes to respond to an event) and *throughput* (i.e. the number of events responses that have been completed over a given observation interval). Therefore, a system with lower number of preemptions is more efficient than one with higher number of preemptions.

3.2.4 Summary

The past sections have shown that NF requirements are a multidimensional set of interrelated requisites that must be fulfilled throughout all stages of the cycle development. Since conflicts among requirements exist, tradeoffs must be allowed to help the designers with their decisions. Assigning importances to the requirements is a mechanism to specify such tradeoffs. NF requirements are usually referred as QoS requirements.

During the design phase, a system can be structured as a set of concurrent tasks where a single task provides some benefit to the system when it achieves all or part of one or more requirements. In real-time systems, the benefit is mainly measured with respect to the timing requirements. However, it is clear that there exist a number of additional QoS requirements that must be also fulfilled. Moreover, QoS requirements can be local or global with respect to tasks achievements.

Thus, while timeliness is defined as a function of deadlines, how to measure QoS is still a matter of discussion and research. In real-time systems QoS is usually expressed in terms of utility (or benefit or value). Different types of utility exist depending on how it is defined and used.

3.3 Types of Utility

In real-time systems the concept of utility captures the importance of different tasks being executed. This utility can be perceived in different ways depending on how the utility is

defined or employed. In this sense, types of utility can be identified in terms of [50]:

- the level of abstraction; (e.g. utility as a function of time or resources).
- the perspective from which the utility is perceived (e.g. task or user perspective).

3.3.1 Levels of Abstraction

Utility can refer to one of several different levels of abstraction, ranging from a low-level view of the system in terms of tasks characteristics (e.g. deadlines) to a high-level view of the system in terms of applications, user goals and resource allocations. The level of abstraction influences how utility is defined. For instance, utility can be expressed as a function of task completion time (task-level abstraction), as a function of resource allocation (resource-level abstraction), or as a function of the quality of the system observed by the user (output-level abstraction) [50].

Task-Level Abstraction

At the task-level abstraction, utility is described by time-utility functions [31]. These functions express the utility obtained by tasks when they complete at a particular distance from their deadlines. In addition, utility can also be described by weakly-hard constraints [5]. These constraints express the task utility in terms of a level of tolerance for missing some deadlines in specific patterns.

Resource-Level Abstraction

At the resource-level abstraction, utility is described by complex models where several resources (e.g. processors, devices) have associated utility-functions. The utility can vary with respect to time (i.e. utility increases/decreases with the time), or with respect to the resources available (i.e. utility increases/decreases with the availability of resources). Models described in [51] [52] [53] use this concept.

Output-Level Abstraction

At the output-level abstraction, utility is described as a function of the output quality of the system as observed by the user and/or the environment [50]. QoS metrics are defined to indicate at which level of quality an application is executing. In addition, the implementation of different algorithms in an application may produce different output quality. Thus, at the output-level abstraction, utility may not necessarily be a function of a QoS metric but it may be a function of the system implementation.

For instance, in [13] two EDF-based scheduling algorithms for minimising jitter are proposed. One has pseudo-polynomial complexity and provides better results than the other one which has polynomial complexity. If high processing resources are available to the application, the pseudo-polynomial algorithm can be used to obtain high output quality; if medium processing resources are available the polynomial algorithm can be used; otherwise EDF is a good option.

3.3.2 Levels of Perspective

Utility can refer to one of several different levels of perspective from which the utility is perceived by tasks, applications or users [50]. The level of perspective influences how the utility is employed. For instance, utility can be used as a priority for executing tasks (task perspective), as a policy for assigning resources to applications (application perspective), or as a measure of the necessities of users of a system (user perspective).

Task Perspective

Task perspective reflects the necessities of each task in an application. From this perspective, each task is greedy with respect to processor and other resources; the utility for completing all or part of its computation is measured individually. For instance, utility can be used to determine which task executes at any given time or which task can be admitted or discarded to maximise the utility. Value-based scheduling theory is based on this concept. In section 3.4, we describe this approach in more detail.

Application Perspective

Application perspective reflects the necessities of applications in a system where each application may be constituted by one or more tasks. From this perspective, applications are greedy with respect to system resources; the utility is a function of the amount of resources allocated to each application. In some real-time systems the utility is relatively fixed as long as the timing deadlines are met, while in other systems the utility may increase or decrease according to the available resources. Changes in tasks may or may not modify the utility perceived by the application. When dealing with applications, utility of individual tasks could not be considered.

For instance, in [53] a complex approach for maximising overall system utility and satisfying multi-resource constraints in the context of a phased-array radar system is described. The idea is to maintain at run-time a complex model of QoS requirements that includes utility-functions, weights and other data such that an algorithm can allocate resources to tasks. The approach determines the resource settings of the tasks and attempts to maximize the global utility. Afterwards, an admission control algorithm runs a schedulability test. If the task set is not schedulable, the resources available to tasks are reduced and the admission control algorithm is executed again. This iterative process runs until a feasible task set is found.

User Perspective

The user perspective reflects the needs of users of the system. From this perspective, utility is the degree of user satisfaction obtained when one or more applications are running. Naturally, different users have different perception levels and the perception can be quantitative or qualitative.

When quantitative, the utility perceived can be verified by QoS metrics. Typical QoS metrics that alter the user perception are latency, throughput, and jitter [54]. Other system properties that also have influence on the quality perceived include preemptions, fault tolerance, and energy consumption. When qualitative, the utility may be verified indirectly by deriving metrics that are correlated with a particular requirement.

3.3.3 Summary

The concept of utility varies depending of the level of abstraction and the perspective from which it is perceived. Naturally, a system includes both types of utility in some degree. Utility captures the importance of different activities such that when the resources are scarce a scheduling algorithm can allocate greater resources to important tasks with the objective of fulfill a set of QoS requirements. In general, this scheduling problem is NP-Hard. In the literature a number of scheduling algorithms based on dynamic-priority and fixed-priority paradigms with different tradeoffs between timeliness and QoS are found. The next section reviews some of them.

3.4 Dynamic-Priority Scheduling with QoS

When dynamic-priority scheduling algorithms include QoS requirements, such that algorithms trade-off timeliness against QoS at run-time. When deadlines must be guaranteed, most of the algorithms are too complex to be implemented in real applications. Thus, these algorithms are more oriented to soft real-time systems although their complexity is still very high. In general, most of soft real-time systems use time-utility functions to express the QoS.

Time-utility functions describe the utility obtained when a task completes; the utility can remain static or can vary with the time [7]. Time-utility functions can be interpreted as quality functions, reward functions, or penalty functions; such a function may either be maximised or minimised depending on its definition. Metrics such as the utility accrued (or value accrued) during a window of time are usually utilised. Thus, a scheduling problem is formulated in terms of *meeting the timing requirements and maximising the utility* [55]. Including timeliness and utility into the scheduling decisions is sought by the value-based or reward-based schemes.

Under value-based schemes, a task has a function $V(t)$ that defines the value to the system for completing the task at time t (see Figure 3.3). Value-based schemes are founded in two approaches: the Value-Density Scheduling algorithm where the utility is used as a priority and the Best-Effort algorithm where the utility is used as an admission policy.

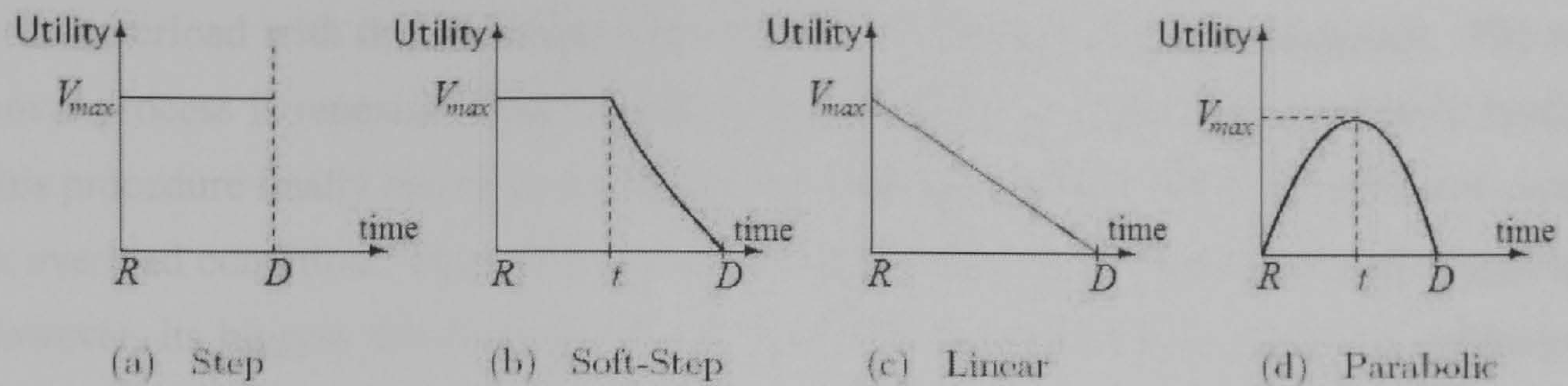


Figure 3.3: Some utility-functions (source [56])

These two above schemes apply for sets of independent tasks which are preemptive, periodic or aperiodic, with worst-case computation time, deadline and $V(t)$ known. These schemes rely on the next observations [31]:

1. In non-overload conditions, if all deadlines can be met by some schedule, the EDF algorithm meets all deadlines.
2. In overload conditions, a schedule organized in decreasing order by value-density, produces a total value at each point in time that is at least as high as any other schedule.
3. Most value functions have their highest value prior to the deadline.

Value-Density Scheduling Algorithm [57]: The Value-Density Scheduling (VDS) algorithm executes tasks according to "the highest value-density, the highest the priority" rule. The *value-density* is computed as V/C, where V is usually the maximum value assigned to the task (e.g. V_{max} in figure 3.3) and C is the expected worst-case execution time. Thus, a priority $P = V/C$ is assigned to a task when it is submitted to the system and the task with the highest priority is executed. The priority P remains fixed until task completion. In [31] is showed that the performance of VDS varies depending on the value function chosen but in general terms, it performs better than other scheduling algorithms under overload situations, in addition to having low overhead.

The Best Effort Scheduling algorithm [57]: The Best Effort Scheduling (BES) algorithm creates a sequence of the available tasks ordered by the EDF rule, which is then sequentially checked for its probability of overload using a heuristic. At any point in the sequence, where this overload probability passes a user-defined threshold, the task prior

to the overload with the minimum value-density is removed from the sequence. The removal process is repeated until the overload probability is reduced to acceptable levels. This procedure finally results in a sequence of processes in EDF order that does not cause an overload condition. Thus, BES achieves a high total utility under overload situations. However, its biggest disadvantage lies in their lack of predictability and sub optimality since it may discard tasks, which will be able to be scheduled under a clairvoyant algorithm [58]. In addition, experimental results in [59] showed that executing the BES algorithm on the same processor as application tasks may result in very large overheads under overload conditions. For example, under a load of 200% or more, over 80% of the processors time was spent in the scheduler deciding which task to run next [59].

In general, value-based schemes are oriented to enhance the system performance under normal operating loads and to reduce any performance degradation due to overload conditions [9]. Studies of the impact of different utility priority assignments have been published in the context of overloaded systems [60] [9] and real-time systems databases [61]. Different algorithms that improve the original BES algorithm are presented in [62] [63]; some recent advances in all these topics are reviewed in [55].

When utility is a function of resource allocation, there are best-effort schemes with complex admission policies designed to trade-off quality and timeliness at run-time. Models described in [51] [52] [53] assume a system with multiple concurrent applications each operating at different levels of quality based on resources available. In the so-called QoS negotiation model [64] [65], services must guarantee predictable performance under specified load and failure conditions, and ensure graceful degradation when these conditions are violated. In [66] a best-effort scheme include energy savings objectives [66].

When utility is a function of output quality, the EDF algorithm provides good results with respect to jitter, latency, and total number of preemptions [17]. However, the biggest disadvantage of EDF is its lacks of predictability on some situations such as when a task misses a deadline. In effect, when a task misses its deadline may cause all subsequent tasks to miss their deadlines (i.e. the so-called domino effect) causing unpredictable output quality results. Variations of the EDF algorithm specifically designed for reducing jitter can be found in [13] [67] [14].

3.4.1 Discussion

Scheduling schemes based on utility functions have several drawbacks namely:

1. They are heuristics.
2. They do not guarantee hard timing constraints, beyond the special case when it is reduced to a EDF scheduler.
3. There is not a methodology for the design of utility functions [8].
4. It is assumed, a priori, that arithmetic operations can be performed with the utility functions, which it is not necessarily true [68].
5. They have higher overhead than fixed-priority based algorithms.

Drawbacks (1) and (2) have relegated utility-based scheduling into the domain of soft dynamic-priority real time systems. Points (3) and (4) are related with the meaning of value. Prasad et al. [68] have shown that the assignment and the use of values are not separate issues but are linked. Any assignment of values must conform to a *scale* of measurements and the scheduling scheme must be cognizant of the scale utilized to perform only meaningful operations.

3.5 Fixed-Priority Scheduling with QoS

Fixed-priority scheduling algorithms that include QoS requirements meet timeliness and fulfill QoS as a secondary objective. They guarantee hard deadlines and hence they are oriented to hard real-time systems; since that the priority assignment is off-line, the run-time costs are minimal. However, most of these algorithms require non-standard fixed-priority run-time support. Consequently, most of them are not directly implementable in standard fixed-priority systems.

In general terms, utility functions are not used in fixed-priority scheduling due to the high complexity of computing a set fixed-priorities that fulfill deadlines and maximise the sum of utilities.

When utility is a function of output quality perceived, there are some scheduling algorithms in the literature. For example, in [69] a fixed-priority assignment algorithm improves system fault resilience in fault-tolerant hard real-time systems. This approach is very attractive because the fault resilience of task sets is maximized only with the adequate priority assignment.

In [70], fixed-priorities are computed using a branch and bound algorithm with the objective of both meeting hard deadlines and minimising a function of the response time. The task model assumes periodic tasks with a weight that models the importance of the task in terms of worst-case response time. According to these weights, feasible fixed-priorities are computed such that the mean weighted response time of the tasks is minimised. Unfortunately, the algorithm is inefficient; for instance, given a time limit of one hour for finding a priority ordering, no solution was found for some sets with 18 tasks; moreover, no solution at all was found for task sets with more than 28 tasks.

From the FPS perspective, solutions based on pure fixed-priority assignments (as in [69]) are ideal because they improve the QoS at zero cost for the implemented scheduling mechanism. Unfortunately, most of the approaches concentrate on increasing some QoS measure by modifying either the base scheduling mechanism or the task characteristics. For instance, approaches for reducing the number of preemptions in FPS have been published but such solutions introduce additional problems such as requiring non-standard runtime support [71], or multiplying the number of tasks to be scheduled by at least a factor of two [72]. Output jitter problems are dealt with using techniques such as designing special purpose task models, using jitter compensation schemes in feedback controllers, optimising the selection of task attributes, or a mixture of all them [15] [14] [16]. In the energy consumption problem, the use of energy must be bounded to guarantee stability and/or extended the lifetime of a system. In this case, the system includes specialized hardware (e.g. Dynamic Voltage Scaling processor [73] or I/O Device Power States [74]) and the solutions are pairs *(priority,power-state)* such that at run-time both the priority and the power-state are applied. A number of papers related to this problem have been published and some solutions can be found in [75] and [76].

3.5.1 Discussion

Scheduling with QoS requirements has been extensively studied in the context of dynamic-priority real-time systems. However, the solutions proposed are too complex to be implemented in real applications. Moreover, the dynamic scheduling paradigm lacks support of commercial real-time operating systems and programming languages.

In the context of fixed-priority scheduling, solutions have been proposed although mostly relying on modifications to the FPS base scheduling mechanism or changes to the task set characteristics. Therefore, such solutions cannot be implemented directly in standard FPS systems.

In general terms, utility functions offer a generic paradigm for expressing QoS requirements. Unfortunately, they have not been included in FPS. Indeed, utility-based scheduling is synonymous of dynamic scheduling in soft real-time systems.

We argue that utility functions have not been included in FPS because they are too complex for analysis, and such complexity is an obstacle to formulate appropriately the scheduling problem with deadlines and QoS requirements. This difficulty has been recognized in other areas such as economics and game theory, where utility functions have been used for several decades

- *Economics:* In economics, utility-functions assign numbers on a cardinal scale to represent preferences of individuals (e.g. pleasure, happiness) toward goods [77]. This concept is called *cardinal utility*. Clearly, the difficulty is how to establish the relation between object and subject. Another problem is that although comparisons between goods normally make sense, it is difficult to make comparison between individuals. By contrast, modern theory starts with a preference ordering as its primitive. This preference is an ordinal relation between any two bundles of commodities that specifies which one the individual prefers. This concept is called *ordinal utility*. The utility reflects the extent to which a thing or an action is preferred to others [78] [79].
- *Game Theory:* In game theory, each point in the space of a decision can be mapped to a payoff (utility) function. This limits its applicability to problems where the pay-

off functions can be mathematically defined for all players. Furthermore, in many problems the main difficulty lies in the ability to formulate a model appropriate for the payoff functions. Instead of calculating payoffs, in *theory of ordinal games* the players may have preferences that can be expressed as a *rank-ordering* of the various options available to them. Moreover, these preferences may be completely subjective in nature rejecting certain biases or experiences [80].

In effect, although utility is conceptually simple, there are inherent difficulties (see [68]) in defining an adequate utility-function for each task, that posteriorly has to be combined with other ones to make a meaningful decision. Therefore, instead of attaching a utility-function to each task, we may observe the problem defining ordinal preference relationships among tasks to measure the QoS. These preferences are called importance relationships.

Chapter 4

Importance

4.1 Introduction

In order to improve the QoS of a system, scheduling algorithms allocate larger resources to the most important tasks. In real-time systems, the concept of importance among tasks is usually defined in terms of utility functions that, although powerful, can be very difficult to both derive and use.

In this chapter, we propose a different approach. We define importance in terms of predilection for executing tasks in preference to other tasks in a system as well as predilection for executing tasks in specific orders. Thus, importance expresses that in a schedule it is desirable to run a task or groups of tasks in preference to other ones without specifying how much this preference would be. This concept is based on preferential statements of the form

"it is more important to execute α than β"

This model is more intuitive and less restrictive than approaches based on the concept of utility. We define a QoS metric based on this concept and propose its use in the context of fixed-priority scheduling in two specific levels of abstraction:

1. For improving QoS at task level by assigning higher priorities to higher importance tasks while the deadlines are guaranteed. For instance, tasks sensitive to preemp-

tions are assigned higher priorities than other ones if such an assignment guarantees deadlines.

2. For improving QoS at application level by finding feasible priority assignments that improve output quality. For instance, if it is known that a particular assignment of priorities $\mathcal{P}$ is effective for reducing the total number of preemptions but such an assignment is infeasible, then finding a feasible one $\mathcal{P}^*$ similar to $\mathcal{P}$ not only will guarantee the deadlines but also will potentially reduce the total number of preemptions.

Thus, in the next sections we define importance among tasks and importance among orderings of priorities. Afterwards, we define a metric to measure the importance and formulate the scheduling problem with deadlines and importance. In addition, we also define some scheduling problems with deadlines and a secondary QoS metric that can be empirically correlated with the concept of importance such that solving a problem with importance produce similar results as solving the problem with the corresponding QoS metric. We present some examples to clarify this idea since solutions to these problems will be proposed in the following chapters.

4.2 Process Model

We consider an extension of the traditional process model utilised in FPS, where a task set must be scheduled on a single processor system. In order to increase the quality of a system, we assume that tasks are characterized not only by a deadline but also by an importance value.

The tuple (C, T, D, P, B, J, I) characterises a task j where C is the worst case computation time, T is the period or the minimal inter-arrival time between two consecutive releases, depending whether it is periodic or sporadic; D is the deadline of the task relative to the actual release; the deadline is less than or equal to the period ($D \leq T$); P is the priority of the task where 1 is the lowest priority; without loss of generality we assume that two tasks do not share the same priority. The blocking factor B is the maximum interference that a task may suffer from lower priority tasks due to a sharing resource protocol

such as the priority ceiling protocol. The release jitter J is the maximum elapsed time between the programmed initial release time and the real ready-to-run time, which is usually zero for periodic tasks. The tasks are preemptive, they are released at time zero and they do not suspend themselves.

Finally, I is a natural number which represents the importance of the task with regard to the other tasks in the system. We assume that all I parameters are different and hence, a task set can be totally ordered with respect to I. Moreover, I is a number in the range of 1 to N where N is the number of tasks and therefore, it is used for indexing tasks in S. Refer to section 4.5.1 for a further definition of importance and refer to section 4.5.3 for a discussion about some issues when two tasks share the same importance.

4.3 Tasks Orderings

For a given task set $S = \{a, b, c, \ldots\}$, an *ordering* on S with relation $\lhd$ (*"precedes"*) is

$$S^{\lhd} = \langle a\, b\, c\, \ldots \rangle \text{ such that } a \lhd b \lhd c \lhd \ldots$$

For instance, given the relation $D =$*"has a shorter deadline than"*, and a task set $S = \{a, b, c\}$ with $D_a > D_b > D_c$, its ordering under this relation is $S^D = \langle c\, b\, a \rangle$. We use $\{\ \}$ to denote a task set and $\langle\ \rangle$ to denote an ordering. In general, Greek small letters usually denote orderings or sub-orderings. Note that orderings can be indexed as a vector. Following these conventions:

- For a given task set S, we denote $\hat{S}$ the set of all possible $N!$ orderings of S, and $\hat{S}_F$ the set of all feasible orderings of S. Note that $\hat{S}_F \subseteq \hat{S}$
- The ordering defines a priority assignment over the tasks such that

  $$\text{if } \langle\, a\, b \,\rangle \text{ then } P_a > P_b$$

  Therefore, an assignment of priorities and a priority ordering will be interchangeable terms.
- S^D is the ordering by DMPA (the deadline monotonic priority assignment).

- S^I is the ordering with priorities assigned according to the rule *"the higher the importance, the higher the priority"*. For instance, in a task set $\{a, b, c\}$ with $I_b > I_c > I_a$, the ordering S^I is $\langle b\,c\,a \rangle$. We will refer to S^I as *the highest importance ordering.*

- All orderings in $\hat{S}$ are organized as the ordering of dictionaries using the lexicographic rule. For example, if the first task in an ordering α is more important than the first task in an ordering β then α will be more important than β and therefore, α will precede β in the dictionary. In case of ties, the second task in both orderings will be compared, and so on. The operator $\succ$ is used to denote lexicographic precedence between tasks as well as between orderings (see section 4.5).

4.4 Relative Importance Statements

In areas such as Economics and Game Theory it has been recognized that expressing benefit with utility functions is too complex and it is not always possible to attach an utility function to each individual. However, by defining ordinal relationships, similar results can be achieved with the advantage that conceptually it is simpler.

One way for expressing benefit in terms of ordinal relationships is using preferential statements. An example of preferential statement is "I prefer the value x_0 for variable X if $Y = y_0$ and $Z = z_0$". Such statements are used in, for example, Artificial Intelligence applications for eliciting user preferences from a set of possible actions. The rationale is that for making good decisions, it must be possible to assess and compare different alternatives. The alternatives are given by users that probably do not have time, the knowledge, or the expertise required to specify complex multi-attribute utility functions. Therefore, qualitative information should be obtained from users by simple preferential representation [81].

Observe that a preferential statement as stated above is not so simple. In fact, complex frameworks as described in [82] are required to gather sets of preferential statements to reduce required information to derive conclusions. However, there exists a class of simpler preferential statements of the form

"it is more important to me that the value of X be higher than the value of Y";

these statements are called *relative importance statements.*

The simplest relative importance statements are the following [81]:

- *relative importance*: "X is more important than Y"
- *conditional relative importance*: "X is more important than Y if some conditions are fulfilled"

In this chapter we will concentrate only in relative importance. We consider apart conditional relative importance in chapter 6. Relative importance statements can also be integrated in complex frameworks as in [81]; nevertheless they are more intuitive than general preference statements. Relative importance statements could be used to express subjective preferences derived from qualitative analysis (i.e. based on opinions, behaviours or experiences).

For instance, in some safety-critical real-time database applications, both timely execution of transactions and data temporal consistency are important requirements. Unfortunately, preemptions produced by priority based scheduling may cause the data read by the transactions to become temporally inconsistent by the time the data is used. Such inconsistencies may compromise the application safety (i.e. its ability to keep operating free of catastrophic consequences to the environment, equipment or people) and reliability (i.e. its ability to keep operating according to its specification). Thus, assuming that a task x implements a safety-critical transaction and a task y implements something else, it is reasonable to elicit that safety and reliability will be jeopardized if x is preempted frequently by y; the statement "executing x is more important than executing y" expresses this as a QoS requirement without necessity of specifying how much safety or reliability are compromised.

Thus, an alternative form of expressing QoS requirements is using these relative importance statements. Instead of defining a utility function for each task, a simple ordinal number representing the importance of a task will be defined. This number will identify the relative importance of a task with respect to the relative importance of other tasks in

the system. Moreover, different orderings by importance can be defined and compared following the lexicographic rule. This permits to define the relative importance of an ordering with respect to other orderings. The next section defines these concepts.

4.5 Defining Importance

4.5.1 Task Importance

Definition (Relative Importance). *Relative importance is a binary relation $\succ$ in a task set S such that for any pair of tasks x and y,*

"$x \succ y$" is interpreted as *"task x is more important than task y"*

The relation $\succ$ has the following properties. For all x, y and z in S:

- It is not the case that $x \succ x$ (irreflexive).
- If $x \succ y$ then it is not the case that $y \succ x$ (asymmetric).
- If $x \succ y$ and $y \succ z$ then $x \succ z$ (transitive).
- Exactly one of $x \succ y$ or $y \succ x$ is true.

Thus, the relative importance relation defines a *strict total order* on S. It permits to compare any pair of tasks by importance and to assign a different natural number I to each task such that the relation is preserved.

Definition 4.5.1 (Task Importance). *Task importance is a natural number I such that for any pair of tasks x and y, there exist task importance I_x and I_y respectively (with $I_x \neq I_y$) such that*

$$\textit{if } x \succ y \Rightarrow I_x > I_y$$

The number I expresses a preference for executing a task with respect to the other tasks in the system. Thus, tasks can be compared and ordered totally according to their relative importance.

4.5.2 Index of Importance Z_I

We note that by labeling tasks according to their importance, the set of $N!$ possible permutations of S form a set $\hat{S}$, which can be ordered lexicographically and indexed by importance. For example, for a task set $S_3 = \{a, b, c\}$ where a is the most important and c the least one such that $I_a > I_b > I_c$, the 3! orderings in lexicographic order are:

$$\hat{S}_3 = \{\langle a\,b\,c\rangle\ \langle a\,c\,b\rangle\ \langle b\,a\,c\rangle\ \langle b\,c\,a\rangle\ \langle c\,a\,b\rangle\ \langle c\,b\,a\rangle\}$$

Note that the ordering arranged in order of importance $\langle a\,b\,c\rangle$ is the highest importance one when compared with the other ones. The least important ordering is $\langle c\,b\,a\rangle$ because it is contrary to the highest importance one. The rest of the orderings can be compared following the lexicographic rule as follows:

Definition 4.5.2 (Lexicographic comparison of orderings). *Let α and β be two orderings in $\hat{S}$. We say that "α* is more important than *β" if comparing each element $\alpha[k]$ with $\beta[k]$, starting from the leftmost to the rightmost, the first difference is in the kth task and the importance of $\alpha[k]$ is greater than the importance of $\beta[k]$. We denote it as $\alpha \succ \beta$ and then*

"$\alpha \succ \beta$" is interpreted as *"the ordering α is more important than the ordering β"*

In other words, if $\alpha \succ \beta$ then we say that α precedes β lexicographically. This is similar to comparing two strings.

Note that the operator $\succ$ is used for comparing tasks as well as orderings. In addition, observe that

- The lexicographic rule orders totally to $\hat{S}$ (i.e. the set of $N!$ orderings of S). For the sake of simplicity, we will assume that $\hat{S}$ is always ordered lexicographically.
- There exists a least element in $\hat{S}$, which precedes the other orderings ones. This is the task set S ordered in strict order of importance. We denote it as S^I.

Consequently, $\hat{S}$ is well-ordered and a function for indexing orderings in $\hat{S}$ can be defined as follows:

Definition 4.5.3 (Lexicographic Index). *For all orderings in $\hat{S}$, its lexicographic order defines $Z_I : \hat{S} \longrightarrow \mathbb{N} \cup 0$ which is a function that indexes any element in $\hat{S}$, such that $\forall \alpha, \beta \in \hat{S}$*

$$\alpha \succ \beta \Rightarrow Z_I(\alpha) < Z_I(\beta)$$

Observe that the index of the highest importance ordering is zero (i.e. $Z_I(S^I) = 0$). For instance, in $\hat{S}_3$ the orderings are compared as follows:

$$S^I = \langle a\,b\,c \rangle \succ \langle a\,c\,b \rangle \succ \langle b\,a\,c \rangle \succ \langle b\,c\,a \rangle \succ \langle c\,a\,b \rangle \succ \langle c\,b\,a \rangle$$

and then

$$Z_I(\langle a\,b\,c \rangle) = 0,\ Z_I(\langle a\,c\,b \rangle) = 1,\ \ldots, Z_I(\langle c\,a\,b \rangle) = 5$$

Thus, orderings can be identified unambiguously by its lexicographic index where the ordering with index 0 is more important than the ordering with index 1 and so on. The lexicographic index ranks all orderings with respect to the highest importance one, and therefore:

Definition 4.5.4 (Index of Importance Z_I). *The lexicographic index defines an index of importance Z_I among all orderings in $\hat{S}$.*

The index of importance expresses a preference for executing a particular ordering with respect to the other orderings in the set of possibilities $\hat{S}$.

A significant property of the lexicographical order is that it preserves well-orders, that is, the Cartesian product of two well-ordered sets is also a well-ordered set. Thus, the lexicographic order has been used in areas such as software engineering for ranking a set of user requirements, or in artificial intelligence for ordering preferences in a system of reasoning. When different user requirements or preferences coexist, the lexicographic rule is a way of combining them into a single lexicographic order [83]. This single lexicographic order can be directly mapped to our model of importance.

Note that the index of importance is unique; i.e. $\forall \alpha, \beta \in \hat{S},\ \alpha \neq \beta \Rightarrow Z_I(\alpha) \neq Z_I(\beta)$. Thus, orderings can be compared and organized in unique levels of importance that reflects a level of QoS where index 0 is the best and the quality decreases as the index increases.

Index	$I_a > I_b > I_c$
1	a b c
2	a c b
3	b a c
4	b c a
5	c a b
6	c b a

Index	$I_a = I_b > I_c$
1	a a c
2	a c a
1	a a c
2	a c a
3	c a a
3	c a a

Table 4.1: When in $\{a, b, c\}$ all tasks have different importance, there are 6 different indices of importance; when a and b have the same importance, there are only 3 different indexes

4.5.3 Tasks Equally Important

When in a task set the assignment of importance is unique each one of the $N!$ priority orderings in $\hat{S}$ is different. However, when at least two tasks have the same importance, the orderings are not unique and can be divided into equivalent subsets that are totally ordered.

For example, consider the set S_3 and their permutations $\hat{S}_3$ (see Table 4.1). When all tasks have different importance, each element in $\hat{S}_3$ maps to just one index of importance; however, when $I_a = I_b$ the lexicographic order maps pairs of orderings to a single importance index. This defines three equivalent subsets:

1. $S_{3,1} = \{\langle a, b, c\rangle, \langle b, a, c\rangle\}$
2. $S_{3,2} = \{\langle a, c, b\rangle, \langle b, c, a\rangle\}$
3. $S_{3,3} = \{\langle c, a, b\rangle, \langle c, b, a\rangle\}$

Clearly the orderings in $S_{3,1}$ are more important than the rest but it is undefined which one is preferred. Note that if both orderings in $S_{3,1}$ are feasible, an a posteriori conclusion must be taken to decide which one is the more convenient. However if just one of them is feasible, it is clear that it is preferred. If both are infeasible and the solution is in $S_{3,2}$, then it is necessary to test for schedulability both orderings in $S_{3,2}$ and perform the same analysis as above. This process can be performed by exhaustive search enumerating all feasible orderings in an equivalent subset.

Thus, when some tasks have the same importance, $\hat{S}$ will not have a single highest importance ordering S^I but a set of highest importance ones $\{S_i^I, i = 1, 2, \ldots\}$. Which S_i^I is the best one depends on the application.

Therefore, if a conflict exists for tasks equally important, its resolution is left at the discretion of the system implementer.

4.6 Scheduling Problems

In the context of fixed-priority scheduling problems, a *solution* is a particular assignment of priorities. In multicriteria fixed-priority scheduling, a single optimal solution that satisfies all the criteria normally does not exist. Instead, a set of pareto-optimal solutions is computed. A solution is pareto-optimal if it is not possible to improve such a solution with respect to one criterion without affecting negatively on the other criteria. It implies the need for making tradeoffs among criteria.

In hard real-time systems there is no tradeoff between timeliness and other QoS criteria; meeting deadlines is mandatory and other criteria are considered secondary. In this sense, all solutions in the pareto-optimal set have to be feasible. Consequently, tradeoffs among QoS criteria can be done but only in the subset $\hat{S}_F$ of feasible solutions ($\hat{S}_F \subseteq \hat{S}$).

Recalling that a fixed-priority algorithm is optimal when it produces a feasible priority assignment if a feasible one exists, we define optimality with respect to meeting deadlines and meeting secondary criteria Z as follows:

Definition 4.6.1 (Z*-optimality*). *An assignment of fixed-priorities is* Z*-optimal if the assignment is feasible and maximises/minimises criteria Z in the set* $\hat{S}_F$ *of feasible solutions.*

With this definition of optimality, in the following sections we formulate several bicriteria scheduling problems where the objective is to find *Z-optimal* fixed-priority assignments. Posteriorly in section 4.8 we show examples of how the concept of importance can be applied to such bicriteria problems.

4.6.1 Problem: Deadlines and Importance

In hard real-time systems meeting the timing constraints is fundamental while any importance requirement can be considered as a soft requirement. If a task set with priorities assigned proportionally to task importance is feasible, then it is an optimal solution for both hard deadlines and importance criteria. However, if it is infeasible, finding the ordering lexicographically closer to the highest importance one can be considered a *Z-optimal* solution in the sense that there is not other feasible solution with higher importance.

For a task set S, let α be a priority ordering in the set of all possible priority orderings $\hat{S}$. The *relative importance* of α with respect to S^I (i.e. the ordering with the highest importance) is given by its index of importance $Z_I(\alpha)$ (see definition 4.5.4).

Since *Z-optimality* is defined in the set $\hat{S}_F$ of feasible orderings (definition 4.6.1), the *z_I-optimal* solution S^* that maximises the importance in $\hat{S}_F$ is defined as the minimum index of importance as follows:

$$Z_I(S^*) = \min_{\forall \alpha \in \hat{S}_F} \{Z_I(\alpha)\} \tag{4.6.1}$$

In other words, maximising the importance implies in finding the feasible ordering lexicographically closest to S^I:

Scheduling Problem 1. *Given a task set S and an ordering S^I, to find S^* which is an ordering of fixed-priorities that is feasible and is the closest one to S^I.*

4.6.2 Problem: Deadlines and Preemptions

In real-time systems, preemptions may cause undesired high processor utilization or even infeasibility. Preemptions impact cache-related activities, database transactions and introduce additional run-time overhead.

For a task set S and with respect to the schedule under a particular priority ordering α, let $\mathcal{P}_j(\alpha)$ be the total number of preemptions of all instances of a task j in a fixed window

of time. The *Total Number of Preemptions* of all tasks in $A \subseteq S$ is defined as

$$\mathcal{P}(\alpha) = \sum_{\forall j \in A} \mathcal{P}_j(\alpha) \tag{4.6.2}$$

Observe that by defining the metric in a subset A and not in S, this metric can be used for all tasks or for a subset of tasks. All the metrics in the following sections are defined in the same way.

Thus, the $\mathcal{P}$*-optimal* solution $\alpha^{\mathcal{P}}$ that minimises the total number of preemptions in the set $\hat{S}_F$ of feasible solutions is defined as

$$\mathcal{P}(\alpha^{\mathcal{P}}) = \min_{\forall \alpha \in \hat{S}_F} \{\mathcal{P}(\alpha)\}$$

Scheduling Problem 2. *Given a task set S and $A \subseteq S$, to find $\alpha^{\mathcal{P}}$ which is an ordering of fixed-priorities that is feasible and minimises the total number of preemptions in A.*

4.6.3 Problem: Deadlines and Absolute Output Jitter

It is known that in real-time control applications, the presence of jitter causes degradation in control performance and may lead to instability of the controlled system. In multitasking systems, scheduling algorithms introduce some forms of output jitter. Depending on the application, systems can be sensitive to the absolute output jitter or the relative output jitter or both.

For a task set S and with respect to the schedule under a particular priority ordering α, let $T_j^{\min}$ and $T_j^{\max}$ denote the minimum and maximum separation between successive completions of a task j. The absolute output jitter is $\mathcal{J}_j(\alpha) = \max\{T_j^{\max} - T_j, T_j - T_j^{\min}\}$; it measures the worst variation between successive completions of j under the schedule generated by α in a fixed window of time. The *Absolute Output Jitter* of all tasks in $A \subseteq S$ is defined to be the largest $\mathcal{J}_j(\alpha)$ as follows [13]:

$$\mathcal{J}(\alpha) = \max_{\forall j \in A}\{\mathcal{J}_j(\alpha)\} \tag{4.6.3}$$

and the $\mathcal{J}$*-optimal* solution $\alpha^{\mathcal{J}}$ that minimises the absolute output jitter in the set $\hat{S}_F$ of

feasible solutions is defined as

$$\jmath(\alpha^{\jmath}) = \min_{\forall \alpha \in \hat{S}_F} \{\jmath(\alpha)\} \tag{4.6.4}$$

Scheduling Problem 3. *Given a task set S and $A \subseteq S$, to find $\alpha^{\jmath}$ which is an ordering of fixed-priorities that is feasible and minimises the absolute output jitter in A.*

4.6.4 Problem: Deadlines and Relative Output Jitter

For a task set S and with respect to the schedule under a particular priority ordering α, the relative output jitter of a task is $\jmath_j^{rel}(\alpha) = \jmath_j(\alpha)/T_j$. It is its absolute output jitter measured as a fraction of its period during a fixed window of time. For instance, a schedule with a relative output jitter of 20% guarantees that the variation of the output of j is in $[0.8T_j, 1.2T_j]$. The *Relative Output Jitter* of all tasks in $A \subseteq S$ under a particular priority ordering α is defined to be the largest $\jmath_j^{rel}(\alpha)$ as follows [13]:

$$\jmath^{rel}(\alpha) = \max_{\forall j \in A}\{\jmath_j^{rel}(\alpha)\} \tag{4.6.5}$$

and the $\jmath^{rel}$*-optimal* solution $\alpha^{\jmath^{rel}}$ that minimises the relative output jitter in the set $\hat{S}_F$ of feasible solutions is defined as

$$\jmath(\alpha^{\jmath^{rel}}) = \min_{\forall \alpha \in \hat{S}_F} \{\jmath^{rel}(\alpha)\} \tag{4.6.6}$$

Scheduling Problem 4. *Given a task set S and $A \subseteq S$, to find $\alpha^{\jmath^{rel}}$ which is an ordering of fixed-priorities that is feasible and minimises the relative output jitter in A.*

4.6.5 Problem: Deadlines and Maximum Latency

The input-output latency is an important parameter in control systems. Assuming that a control task gets inputs at the beginning of each instance and deliver outputs at the end, minimising the latency improves the responsiveness of the system. Depending on the application, the system may be sensitive to *maximum latency*, *relative maximum latency* or both.

For a task set S and with respect to the schedule under a particular priority ordering α, the difference between the completion time and the beginning time of a task j is the *input-output latency* (i.e. $\mathsf{c}_j - \mathsf{b}_j$). Different instances k of the same task have different latencies. Thus, lets define $\mathcal{L}_j(\alpha) = \max\{\mathsf{c}_{j,k} - \mathsf{b}_{j,k}\}$ as the maximum latency of different instances of a task j in a fixed window of time and under the priority ordering α. The *Maximum Latency* of all tasks in $A \subseteq S$ is defined to be the largest $\mathcal{L}_j(\alpha)$ as follows [17]:

$$\mathcal{L}(\alpha) = \max_{\forall j \in A}\{\mathcal{L}_j(\alpha)\} \tag{4.6.7}$$

and the $\mathcal{L}$*-optimal* solution $\alpha^{\mathcal{L}}$ that minimises the maximum latency in the set $\hat{S}_F$ of feasible solutions is defined as

$$\mathcal{L}(\alpha^{\mathcal{L}}) = \min_{\forall \alpha \in \hat{S}_F}\{\mathcal{L}(\alpha)\} \tag{4.6.8}$$

Scheduling Problem 5. *Given a task set S and $A \subseteq S$, to find $\alpha^{\mathcal{L}}$ which is an ordering of fixed-priorities that is feasible and minimises the maximum latency in A.*

4.6.6 Problem: Deadlines and Relative Maximum Latency

For a task set S and with respect to the schedule under a particular priority ordering α, the relative maximum latency of a task j in a fixed window of time is its maximum latency divided by its worst-case computation time, i.e. $\mathcal{L}_j^{rel}(\alpha) = \mathcal{L}_j(\alpha)/C_j$. Thus, the relative maximum latency measures the worst input-output latency of a task in proportion to its computation time. The *Relative Maximum Latency* of all tasks in $A \subseteq S$ is defined to be the largest $\mathcal{L}_j^{rel}(\alpha)$ as follows:

$$\mathcal{L}^{rel}(\alpha) = \max_{\forall j \in A}\{\mathcal{L}_j^{rel}(\alpha)\} \tag{4.6.9}$$

and the $\mathcal{L}^{rel}$*-optimal* solution $\alpha^{\mathcal{L}^{rel}}$ that minimises the relative maximum latency in the set $\hat{S}_F$ of feasible solutions is defined as

$$\mathcal{L}^{rel}(\alpha^{\mathcal{L}^{rel}}) = \min_{\forall \alpha \in \hat{S}_F}\{\mathcal{L}^{rel}(\alpha)\} \tag{4.6.10}$$

Scheduling Problem 6. *Given a task set S and $A \subseteq S$, to find $\alpha^{\mathcal{L}^{rel}}$ which is an ordering of fixed-priorities that is feasible and minimises the relative maximum latency in A.*

4.6.7 Problem: Deadlines and Average Response-Time

Improving the responsiveness of a system improves the quality perceived for its users. At system level, reducing the response-time of a task implies to reduce the time between the arrival and completion times; consequently tasks spend less time in the ready queue accelerating the flow time and increasing the system performance. The responsiveness of a task during a period of time can be measured by averaging the response-time of all its instances.

For a task set S and with respect to the schedule under a particular priority ordering α, the average response-time of a task j is the sum of all the response-times divided by its number of releases, i.e. $\mathcal{R}_j(\alpha) = \sum_{k=0}^{M-1} R_{j,k}/M$ where $R_{j,k}$ is the response-time of the kth release of j and M is the number of releases in a fixed window of time.

We define the *relative average response-time* as $\mathcal{R}_j(\alpha)/C_j$; it indicates how well a task completes soon. For example, in our task model, all instances of the highest priority task should complete at C_j (assuming no interference by low priority tasks) and therefore, its relative average response-time is 1. The rest of tasks have values higher than 1; larger ratio $\mathcal{R}_j(\alpha)/C_j$ indicates that task j stays active on the system for longer time.

We define the *Maximum Relative Average Response-Time* as the maximum $\mathcal{R}_j(\alpha)/C_j$ of all tasks in $A \subseteq S$:

$$\mathcal{R}(\alpha) = \max_{\forall j \in A}\{\mathcal{R}_j(\alpha)/C_j\} \tag{4.6.11}$$

The *$\mathcal{R}$-optimal* solution $\alpha^{\mathcal{R}}$ that minimises the maximum relative average response-time in the set $\hat{S}_F$ of feasible solutions is defined as

$$\mathcal{R}(\alpha^{\mathcal{R}}) = \min_{\forall \alpha \in \hat{S}_F}\{\mathcal{R}(\alpha)\} \tag{4.6.12}$$

Scheduling Problem 7. *Given a task set S and $A \subseteq S$, to find $\alpha^{\mathcal{R}}$ which is an ordering of fixed-priorities that is feasible and minimises the maximum relative average response-time in A.*

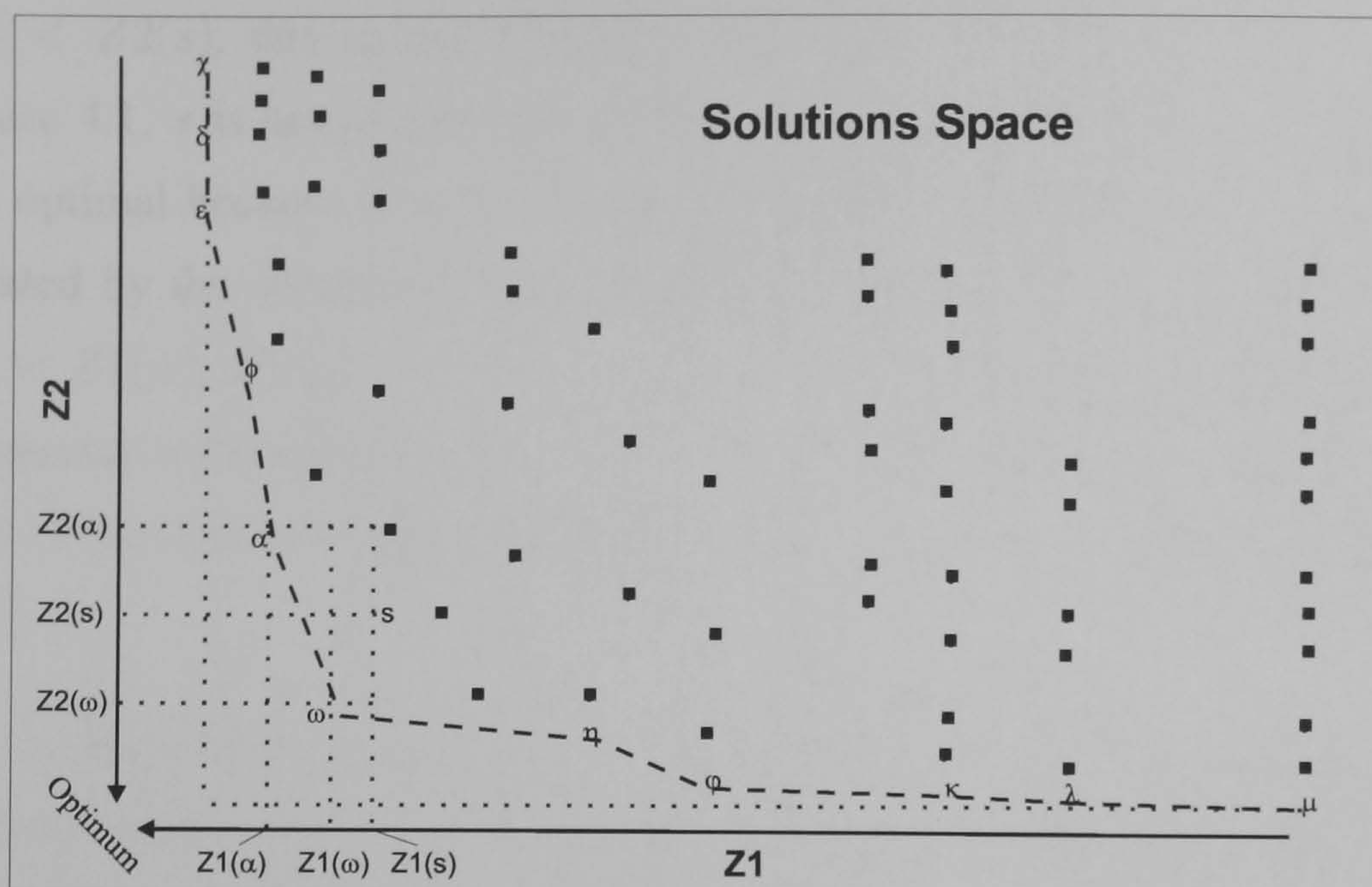


Figure 4.1: A Bicriteria Solution Space for a task set under criteria $Z1$ and $Z2$. Pareto-optimal solutions are indicated by Greek small letters. The solution s is not pareto-optimal because it is strictly dominated by at least another solution; e.g. $Z1(\omega) < Z1(s)$ and $Z2(\omega) < Z2(s)$.

4.7 Representing Bicriteria Problems

Multicriteria scheduling problems are better understood when they are represented graphically. In effect, graphical representation of multicriteria problems assists both, to illustrate the problem and to identify clues to solve it. When only two criteria are involved in a scheduling problem, all pareto-optimal solutions can be represented in a Cartesian plane such that all tradeoffs between criteria can be shown.

For example, lets assume that for a task set S two criteria $Z1$ and $Z2$ are defined such that the optimum tends to zero. By evaluating all orderings in $\hat{S}$ using both criteria, the space of possible solutions is obtained. This is illustrated in Figure 4.1 that shows all tradeoffs between criteria. A point $(Z1(\alpha), Z2(\alpha))$ represents a particular priority assignment α evaluated by both metrics. With respect to $Z1$, points $\{\chi, \delta, \epsilon\}$ at the left side are the optimal solutions. With respect to $Z2$, the point $\{\mu\}$ is the optimal one.

The pareto-optimal set is formed by finding all the solutions not strictly dominated

by other solution. A solution ω strictly dominates a solution s, if $Z1(\omega) < Z1(s)$ and $Z2(\omega) < Z2(s)$, this is, the solution ω is better than s in both criteria. For instance, in Figure 4.1, s is not pareto-optimal because it is dominated by ω. Observe that ω is pareto optimal because it is no strictly dominated. In effect, with respect to $Z1$, ω is dominated by the solutions at its left-side; for instance $\{\alpha, \chi. \delta, \epsilon, \phi\}$ dominate ω (i.e. $Z1(\alpha) < Z1(\omega)$, $Z1(\chi) < Z1(\omega)$, ..., $Z1(\phi) < Z1(\omega)$); however, all these solutions do not dominate with respect to $Z2$ (i.e. $Z2(\alpha) \not< Z2(\omega)$, $Z2(\chi) \not< Z2(\omega)$, ..., $Z2(\phi) \not< Z2(\omega)$). Note that the single point that minimises simultaneously both criteria does not exist.

Naturally, in order to represent a problem in this form, we need expressive metrics. In this sense, the yes/no answer given by the feasibility test is useless. With the only purpose of illustrating graphically bicriteria problems with deadlines, we introduced the next metric.

4.7.1 Deadlines Metric

On FPS, any fixed-priority ordering α is feasible if and only if $R_j \leq D_j \ ,\forall j \in \alpha$. Therefore, for all j in α we have

$$R_j \leq D_j \ \Rightarrow \ \frac{R_j}{D_j} \leq 1 \ \Rightarrow \ \frac{R_j}{D_j} \leq \max_{\forall j \in \alpha} \left\{ \frac{R_j}{D_j} \right\} \leq 1$$

where $\max\{\frac{R_j}{D_j}\}$ corresponds to the task j with the tightest deadline in α.

Introducing

$$Z_D(\alpha) = \max_{\forall j \in \alpha} \left\{ \frac{R_j}{D_j} \right\}$$

we are able to define:

Definition 4.7.1. *An ordering α is feasible iff $Z_D(\alpha) \leq 1$ where*

$$Z_D(\alpha) = \max_{\forall j \in \alpha} \left\{ \frac{R_j}{D_j} \right\} \tag{4.7.1}$$

Note that, if $Z_D \leq 1$ the ordering α is feasible; otherwise it is infeasible. Furthermore, when $Z_D = 1$ or very close to 1, it indicates at least one task finishes very close to its deadline. This metric give us more expressiveness that a simple yes/no answer.

4.8 Using Importance in Scheduling Problems

In this section we present examples of how the model of importance introduced can be applied to practical scheduling problems in real-time systems. For illustrative purposes, we assume a hypothetical real-time system composed of a task set of 5 periodic tasks (see Table 4.2). Observe that this set is biased in the sense that periods, deadlines and computation times are monotonically correlated. However, this does not affect the purpose of the example. In the evaluation chapter, we perform our experiments with unbiased task sets.

4.8.1 Importance at Task Level

In order to illustrate the concept of importance in a problem, let us consider a hypothetical hard real-time system with several QoS requirements, which must be implemented in a fixed-priority real-time operating system. The requirements are to reduce the relative output jitter and to reduce the number of preemptions.

The system has five tasks. The tasks have deadlines less than or equal to their periods. Table 4.2 shows the tasks set with their respective worst-case response times R_j, computed with the response time analysis equation.

- Task a implements a control algorithm which is essential for the stability of the system and consequently, it is highly sensitive to output jitter effects; the control algorithm is also affected by the context switching caused by preemptions.
- Task b reads inputs from sensors and stores them in a database. The database represents the external environment and then, the data freshness is essential for reliability. Task b is sensitive to preemptions because frequent interruptions in the middle of a

τ	T	D	C	R	I	R/D
e	100	80	13	13	1	0.16
d	240	240	37	50	2	0.21
c	330	330	55	118	3	0.36
b	350	350	56	174	4	0.50
a	480	400	68	292	5	0.73

Table 4.2: Task set S_5 with importance values I. a is the highest importance task and e is the lowest one. Note that is is feasible under DMPA.

transaction could cancel it. Since sometimes the sensor readings are used by the control algorithm, task b is moderately sensitive to jitter.

- Similar reasoning can be given to tasks c, d, e; for example, we can say task c is moderately sensitive to preemptions, and task e is indifferent to both jitter and preemptions effects. For the sake of simplicity, we omit additional description because in this specific problem we are only interested in improving QoS for tasks a and b.

Firstly, the tasks will be rated by importance. Observe that, in terms of dependability (i.e. the ability to deliver service that can justifiably be trusted [3]), the above description suggests that tasks a and b are more important than tasks c, d, e and hence, I_a and I_b will be greater than I_c, I_d and I_e.

Afterwards we decide whether a is more important than b or vice versa. Lets assume that reducing jitter is more important than reducing the number of preemptions for several reasons; for example, we can say that output jitter jeopardizes some aspects of dependability such as safety (i.e absence of catastrophic consequences on the users and/or the environment), or reliability (i.e continuity of correct service). Under this reasoning, "a is more important than b" and hence, we assign importance $I_a = 5$ and $I_b = 4$ to tasks a and b respectively. From the subset $\{c, d, e\}$, task c seems more important than both d and e because c is sensitive to preemptions, and then $I_c = 3$. Finally, task e is indifferent to both jitter and preemptions effects and hence, we can assign the lowest importance to e (i.e. $I_e = 1$). Consequently $I_d = 2$ (see Table 4.2).

In order to determine the best solution, we compute $\hat{S}_5$ (i.e. the 5! priority orderings of task set S_5) and simulate all the priority orderings (using the maximum C) during the hyperperiod (184800 time units). During the simulation, the relative output jitter ($\jmath^{rel}$)

Orderings	S^D		S^I		S^*	
	$\mathcal{J}^{rel}$	$\mathcal{P}$	$\mathcal{J}^{rel}$	$\mathcal{P}$	$\mathcal{J}^{rel}$	$\mathcal{P}$
task a	0.435	490	0.0	0	0.171	275
task b	0.3	514	0.19	55	0.0	0

Table 4.3: Relative output jitter ($\mathcal{J}^{rel}$) and total number of preemptions ($\mathcal{P}$) for tasks a and b obtained by the simulating of the orderings S^D, S^I and S^* (smallest figures are better)

and the total number of preemptions ($\mathcal{P}$) is registered for each task. The results of the simulation will be analysed from the perspective of the Deadlines and Importance problem, the Deadlines and Preemptions problem and, the Deadlines and Relative Output Jitter problem.

Importance vs Deadlines. Figure 4.2 shows the plot of all possible priority orderings of task set S_5. Points (x, y) are computed by enumerating all 120 priority orderings and evaluating each ordering α in both the metric Z_D and the index of importance Z_I obtaining $(Z_D(\alpha), Z_I(\alpha))$. Thus, a point indicates if α is feasible and which is the distance of α to the highest importance ordering S^I. In other words, the plot illustrate the space of solutions for the Deadlines and Importance scheduling problem (see 4.6.1).

$S^D = \langle e\,d\,c\,b\,a\rangle$ is the priority ordering by deadlines. This ordering is feasible and hence, from the point of view of the timing constraints, it is satisfactory. However, both tasks suffer jitter and high number of preemptions. Table 4.3 shows that for task a, $\mathcal{J}_a^{rel} = 0.435$ and $\mathcal{P}_a = 490$; for task b, $\mathcal{J}_b^{rel} = 0.3$ and $\mathcal{P}_b = 514$. From the point of view of the QoS requirements, it is disappointing.

$S^I = \langle a\,b\,c\,d\,e\rangle$ is the priority ordering by importance. From the point of view of the QoS requirements, it is rather satisfactory because task a suffers zero jitter and preemptions, and task b suffers low jitter and preemptions (see Table 4.3). Unfortunately S^I is infeasible because the task e misses its deadline. In effect, under S^I, R_e is 229 and then, $D_e \not< R_e$. In terms of the metric Z_D we have $Z_D(S^I) = 229/80 = 2.86 \not< 1$.

Performing exhaustive search over the set of all feasible priority assignments, it can be shown that there are 32 feasible ones. Among them none assigns the highest or the second one higher priority to a. However, there are four priority assignments where the highest

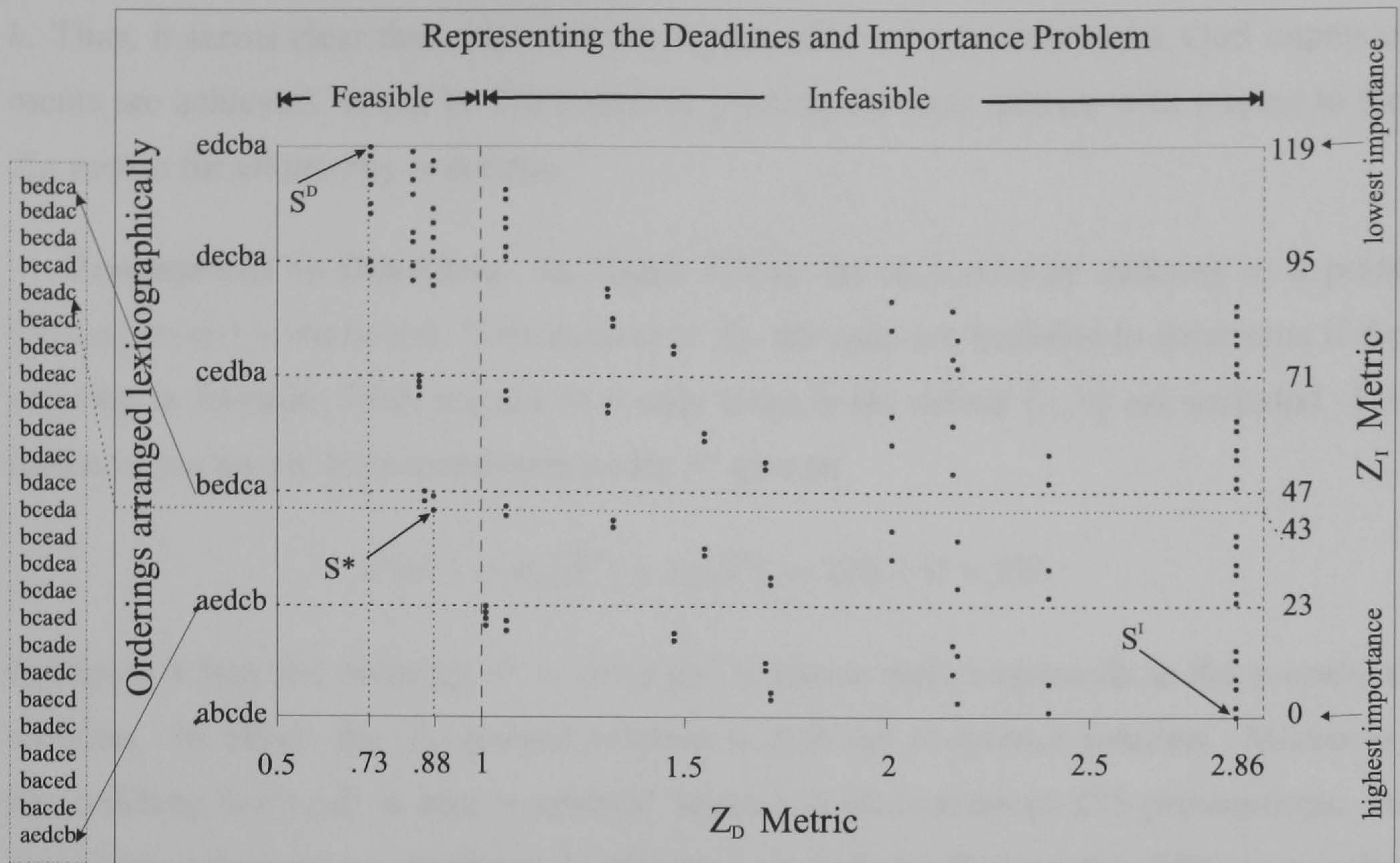


Figure 4.2: Plotting all priority orderings of task set S_5. S^I is infeasible ($Z_D(S^I) = 229/80 = 2.86$). S^D is feasible but has low importance index ($Z_I(S^D) = 119$). The Z_I*-optimal* solution is $S^* = \langle b\,e\,a\,d\,c \rangle$ located in $(Z_D(S^*), Z_I(S^*)) = (0.88, 43)$

priority is assigned to b:

$$S^I = \langle a\,b\,c\,d\,e \rangle \succ \ldots \langle b\,e\,a\,d\,c \rangle \succ \langle b\,e\,c\,d\,a \rangle \succ \langle b\,e\,d\,a\,c \rangle \succ \langle b\,e\,d\,c\,a \rangle \succ \ldots S^D$$

Note that $\langle b\,e\,a\,d\,c \rangle$ is the lexicographically closest one to S^I; thus, $S^* = \langle b\,e\,a\,d\,c \rangle$. Therefore, this priority ordering is the Z_I*-optimal* solution for the scheduling problem with deadlines and importance. This solution is represented graphically in Figure 4.2; it corresponds to the point $(Z_D(S^*), Z_I(S^*)) = (0.88, 43)$.

In Figure 4.2, S^I is infeasible but it is the solution with the highest importance. The DMPA solution S^D is feasible but its index of importance is 119, which is the worst. This example is an extreme case where S^I is completely opposite to S^D. The Z_I*-optimal* is $S^* = \langle b\,e\,a\,d\,c \rangle$ which has the highest importance in the set of feasible solutions.

By observing Table 4.3, we note that S^* is not only the Z_I*-optimal* solution with respect to the deadlines and importance problem but also it is a good solution with respect to minimising the total number of preemptions and the relative output jitter for tasks a and

b. Thus, it seems clear that only with raising priorities of important tasks, QoS improvements are achieved. It can be illustrated by plotting the QoS metrics with respect to the Z_D metric for all priority orderings.

Preemptions vs Deadlines. In Figure 4.3(a), for each priority ordering α, a point $(Z_D(\alpha), \mathcal{P}(\alpha))$ is evaluated. With respect to Z_D all tasks are included to determine if the ordering is feasible. With respect to $\mathcal{P}$ only tasks in the subset $\{a, b\}$ are included. For instance, the metric for preemptions under S^* give us

$$\mathcal{P}(S^*) = \mathcal{P}_a(S^*) + \mathcal{P}_b(S^*) = 275 + 0 = 275$$

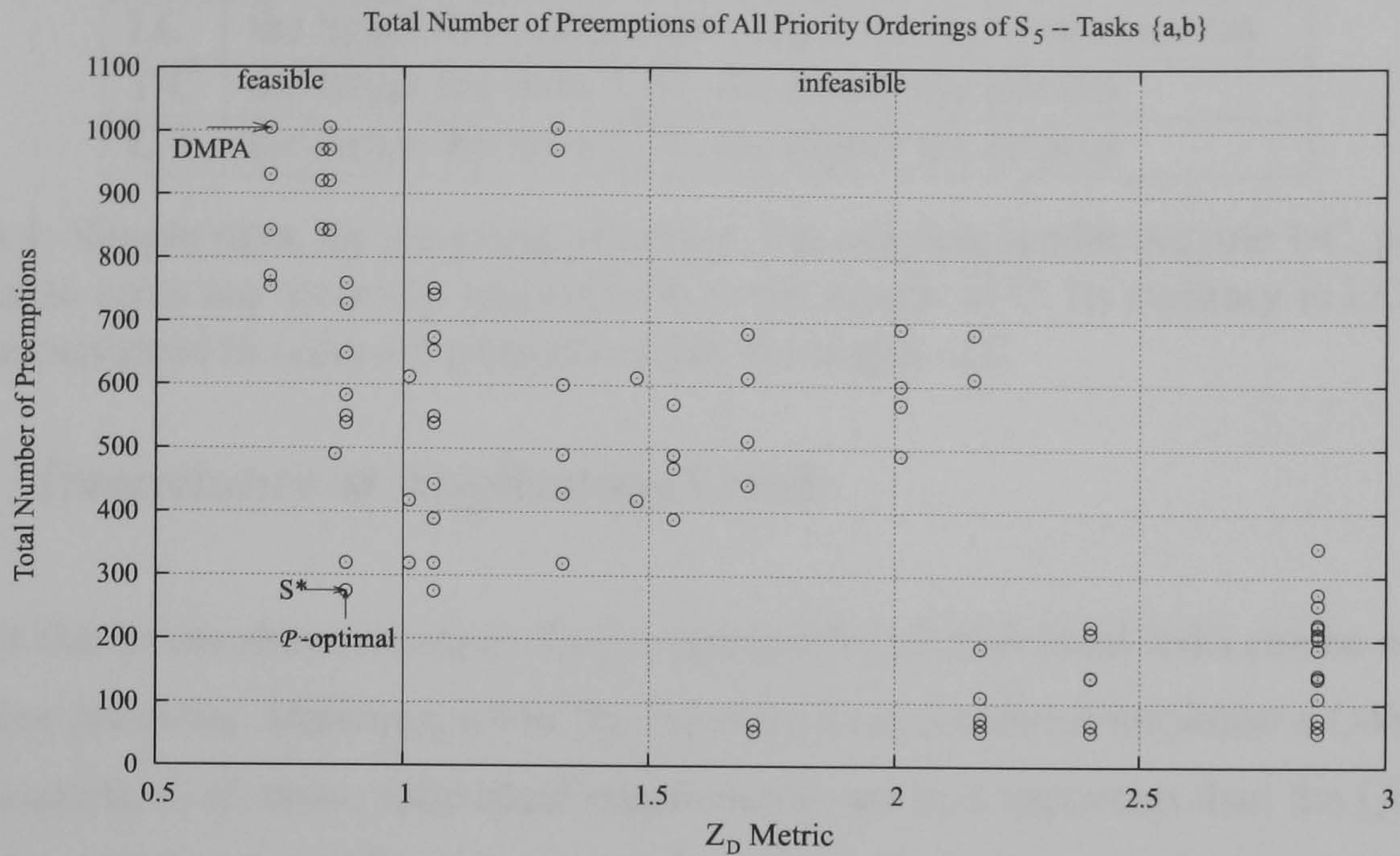
In Figure 4.3(a), the ordering $S^* = \langle b\, e\, a\, d\, c \rangle$ is shown and corresponds to the $\mathcal{P}$*-optimal* solution. In effect, the Z_I*-optimal* solution is also the $\mathcal{P}$*-optimal* solution. Moreover, the ordering $\langle b\, e\, a\, c\, d \rangle$ is also $\mathcal{P}$*-optimal* because it also produces 275 preemptions. In fact many solutions are overlapped indicating equivalent solutions for different priority orderings.

Relative Output Jitter vs Deadlines. In Figure 4.3(b), for each priority ordering α, a point $(Z_D(\alpha), \mathcal{J}^{rel}(\alpha))$ is evaluated. As above, with respect to Z_D all tasks are included to determine if the ordering is feasible. With respect to $\mathcal{J}^{rel}$ only tasks in the subset $\{a, b\}$ are included. For instance, the metric for relative output jitter under S^* give us

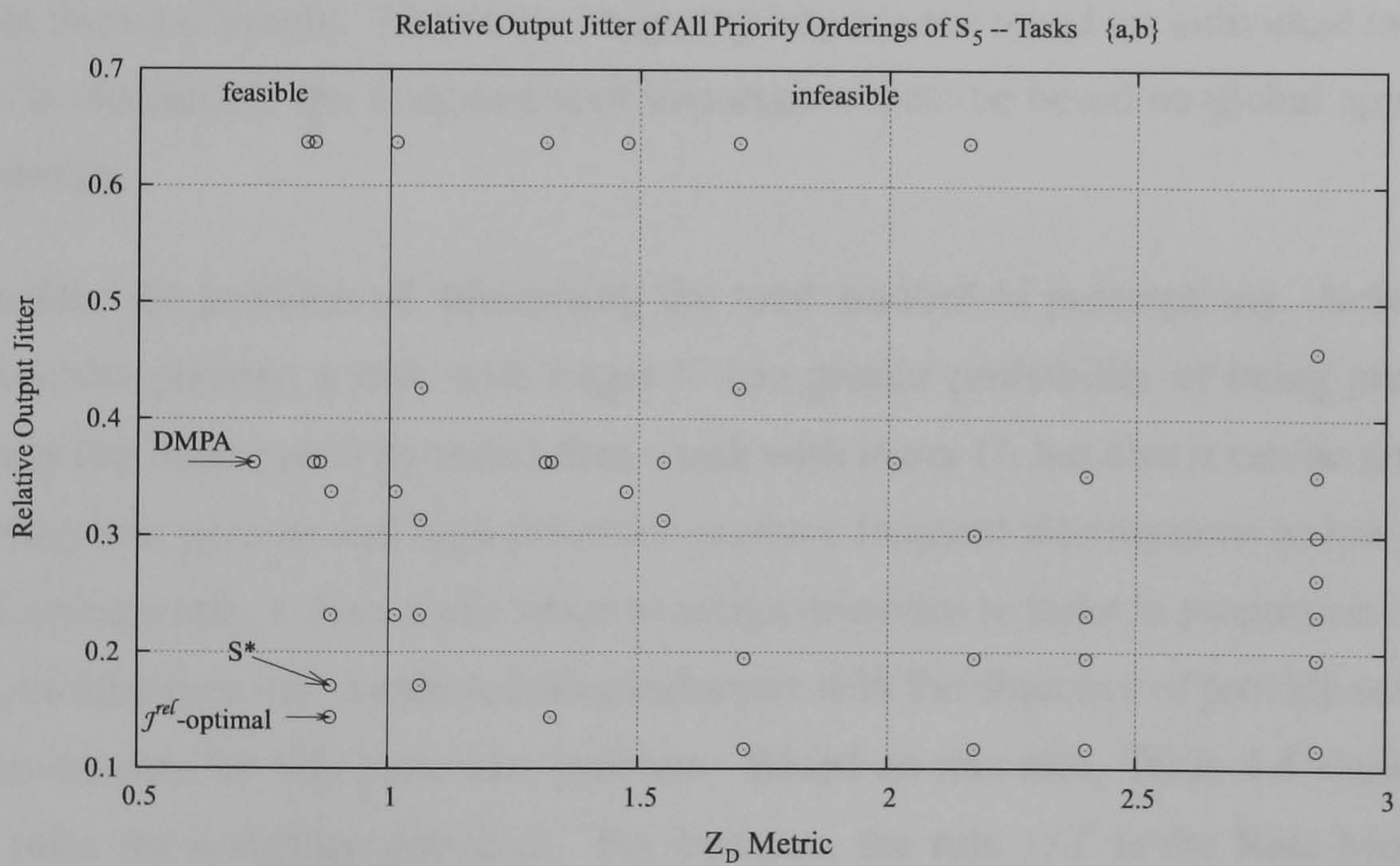
$$\mathcal{J}^{rel}(S^*) = \mathcal{J}_a^{rel}(S^*) + \mathcal{J}_b^{rel}(S^*) = 0.171 + 0 = 0.171$$

In Figure 4.3(b), the $\mathcal{J}^{rel}$*-optimal* solution is indicated and corresponds to four different orderings: $\langle e\, d\, b\, a\, c \rangle$, $\langle e\, b\, d\, a\, c \rangle$, $\langle d\, e\, b\, a\, c \rangle$, $\langle b\, e\, d\, a\, c \rangle$. Note that S^* is very close to the $\mathcal{J}^{rel}$*-optimal*. As above, many solutions are overlapped indicating equivalent solutions for different priority orderings.

Therefore, a solution for the problem where a subset of tasks has different QoS requirements is obtained by assigning higher importance to this subset and solving the scheduling problem with deadlines and importance.



(a) The $\mathcal{P}$*-optimal* solution is indicated and it coincides with S^*



(b) The $\mathcal{J}^{rel}$*-optimal* solution is indicated and it is close to S^*

Figure 4.3: Total Number of Preemptions and Relative Output Jitter of subset $\{a, b\} \subset S_5$ computed during its hyperperiod for all 5! priority orderings. The $\mathcal{P}$*-optimal* and $\mathcal{J}^{rel}$*-optimal* solutions are indicated.

PA	Priority Assignment Rule
1/T	the shorter the period, the higher the priority
1/C	the shorter the computation time, the higher the priority
LT	the larger the period, the higher the priority
LC	the larger the computation time, the higher the priority
T/C	the larger the ratio T/C, the higher the priority
C/T	the larger the ratio C/T, the higher the priority

Table 4.4: Simple rules for assigning priorities. For example, under the rule 1/C, priorities assigned to tasks are inversely proportional to the length of C. Its contrary is LC, where priorities assigned to tasks are proportional to the length of C

4.8.2 Importance at Application Level

Observe that in the above example, QoS requirements of individual tasks can be achieved by raising priorities. However, when the objective is to maximise/minimise a QoS metric which include to all tasks, individual requirements are less important than the QoS metric. In fact, raising the priority of a particular task could impact negatively on other tasks reducing the total benefit. Therefore, assigning importance based on individual task preferences is ineffective; the assignment of importance must be based on global application requirements.

Consider the problem of minimising the total number of preemptions. Note that at any particular priority, a task with larger C has greater probability of being preempted frequently (by higher priority tasks) than a task with lower C; but also it can be noted that tasks with short periods and high priorities produce frequent interruptions to low priority tasks. Consequently, it does make sense to assign priorities to tasks in proportion of larger C or T, or functions that combine both parameters with the objective of provide some kind of improvements for this particular problem. Based on this idea, Table 4.4 shows some simple rules for assigning priorities. For instance, the rule $1/T$ is the Rate Monotonic Priority Assignment algorithm, while the rule LT is its contrary; i.e. under the rule LT, the task with the largest period gets the highest priority and the task with the shortest period gets the lowest priority.

In order to observe the effect of assigning fixed-priorities with those rules, we perform an experiment considering the task set S_5 and the problem of minimising the total number of preemptions. We perform the experiment by computing the $5!$ priority orderings of S_5

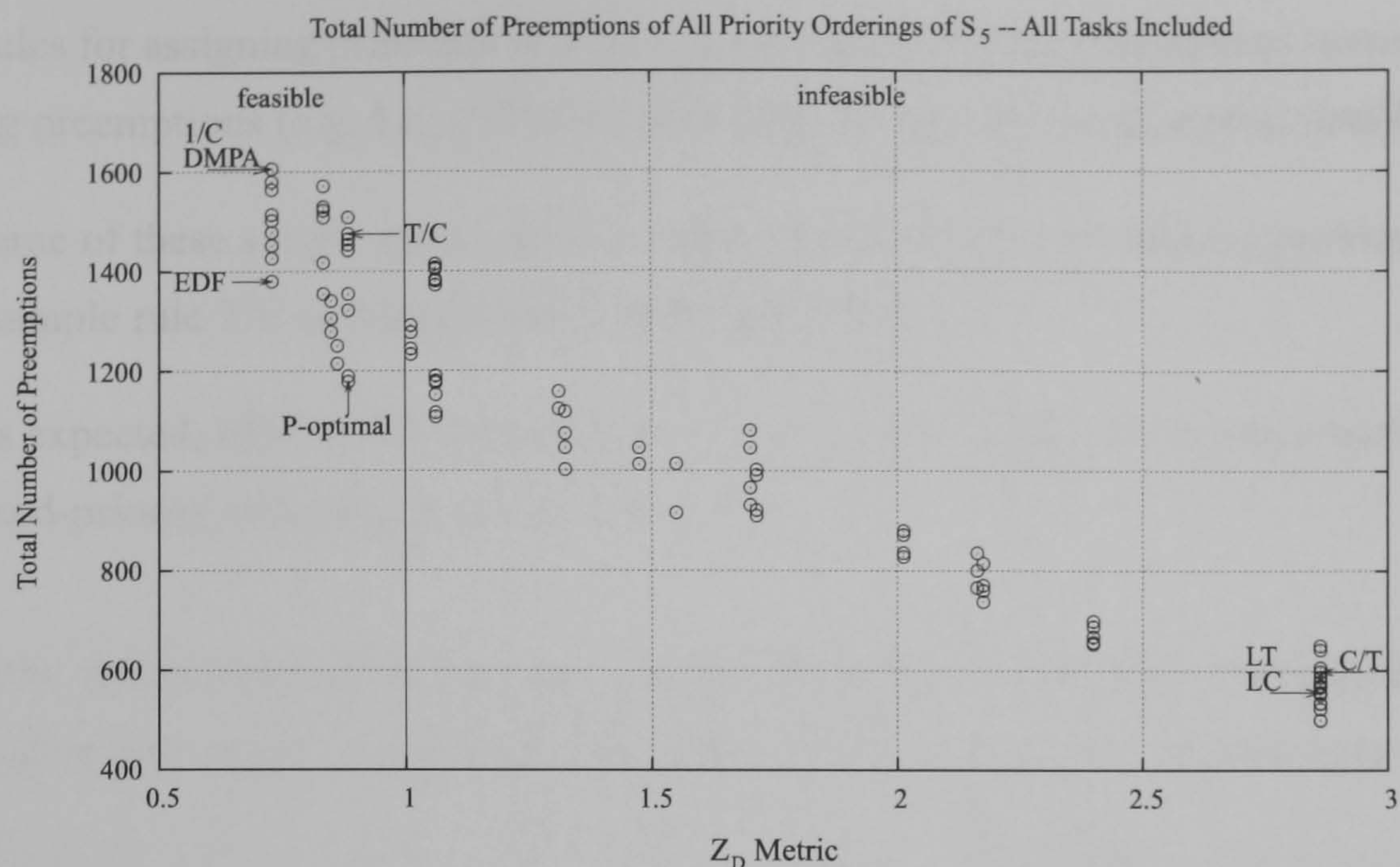


Figure 4.4: Total Number of Preemptions computed during the hyperperiod for all 5! priority orderings of S_5. The $\mathcal{P}$*-optimal* solution is indicated. The results of some simple rules for assigning priorities are also illustrated as well as the result of simulating S_5 under EDF

and simulating all the priority orderings during the hyperperiod. In addition, for comparative purpose we simulate the task set under the EDF scheduling algorithm. During the simulation, the total number of preemptions is registered for all tasks. The results of the experiment are shown in Figure 4.4. Several points are overlapped indicating equivalent solutions for different priority orderings.

Preemptions vs Deadlines. In Figure 4.4, a point $(Z_D(\alpha), \mathcal{P}(\alpha))$ is evaluated for each priority ordering α. Both metrics Z_D and $\mathcal{P}$ are applied to all tasks in the system. There is just one $\mathcal{P}$*-optimal* solution and this corresponds to $\langle c\,e\,b\,a\,d \rangle$; its total number of preemptions is 1178. The priority orderings of the rules in Table 4.4 are indicated in the plot. Observe that for this particular task set:

- The worst solution is DMPA (1606 preemptions).
- Due to the biased task set, the rules 1/T, 1/C and DMPA produce the same priority ordering; it also applies for the rules LT and LC.
- When comparing against DMPA, there exist a number of feasible priority orderings with better performance with respect to the total number of preemptions.

- Rules for assigning priorities may produce excellent results with respect to minimising preemptions (e.g. LC, C/T) but unfortunately they do not guarantee deadlines.
- Some of these simple rules can be good heuristics for this scheduling problem. For example rule T/C is feasible and is better than DMPA.
- As expected, EDF (1380 preemptions) is better than DMPA . However, a number of fixed-priority orderings are better than EDF.

Clearly, the problem of finding the $\mathcal{P}$*-optimal* solution is NP-hard. Nevertheless, in the context of our model of importance heuristics can be proposed to approximate such a solution.

Assigning Importance by Heuristics

In our model of importance, we assume that a higher importance ordering exists and consequently, it is the objective to be achieved. We also assume that the index of importance of a particular priority ordering reflects a QoS requirement. Thus, in order to apply our model to a particular QoS metric, such metric should have a structure similar to the importance model; i.e. there should exist a priority ordering S^I which should be the one that maximises/minimises the metric, and the rest of the priority orderings should decrease/increase the values of the metric following the lexicographic order. In general, finding a QoS metric conforming these assumptions is unrealistic.

Nevertheless, we do some experiments which show that in some cases good approximations to these assumptions can be achieved depending on how the S^I ordering is selected. For instance, continuing with the above example, we follow the next steps:

1. A priority ordering is nominated as the most important with respect to the preemptions metric, and then importance is assigned to tasks according to this ordering. This will be the S^I ordering.
2. The scheduling problem with deadlines and importance is solved; i.e. the Z_I*-optimal* solution S^* is found.

3. The scheduling problem with deadlines and preemptions metric is solved by simulation; i.e. the $\mathcal{P}$*-optimal* solution is found by simulating the $N!$ orderings.

4. Finally, the goodness of the approach is measured comparing how far is Z_I*-optimal* from $\mathcal{P}$*-optimal* with respect to the total number of preemptions.

In Figure 4.4, note that there exist some priority orderings that achieve the minimum number of preemptions; one of such orderings is a logical candidate to be S^I. However, since we do not know how to find it we will use the orderings obtained by the rules LC and C/T as heuristics for approximating them.

Rule LC as the assignment of importance. We assign importance to tasks according to the rule LC and nominate it as the highest importance ordering; i.e. $S^I = \langle a\,b\,c\,d\,e \rangle$. The Z_I*-optimal* solution S^* is computed by testing the feasibility of the priority orderings following the lexicographic order starting from S^I until the first feasible ordering is found. It is illustrated in Figure 4.5.

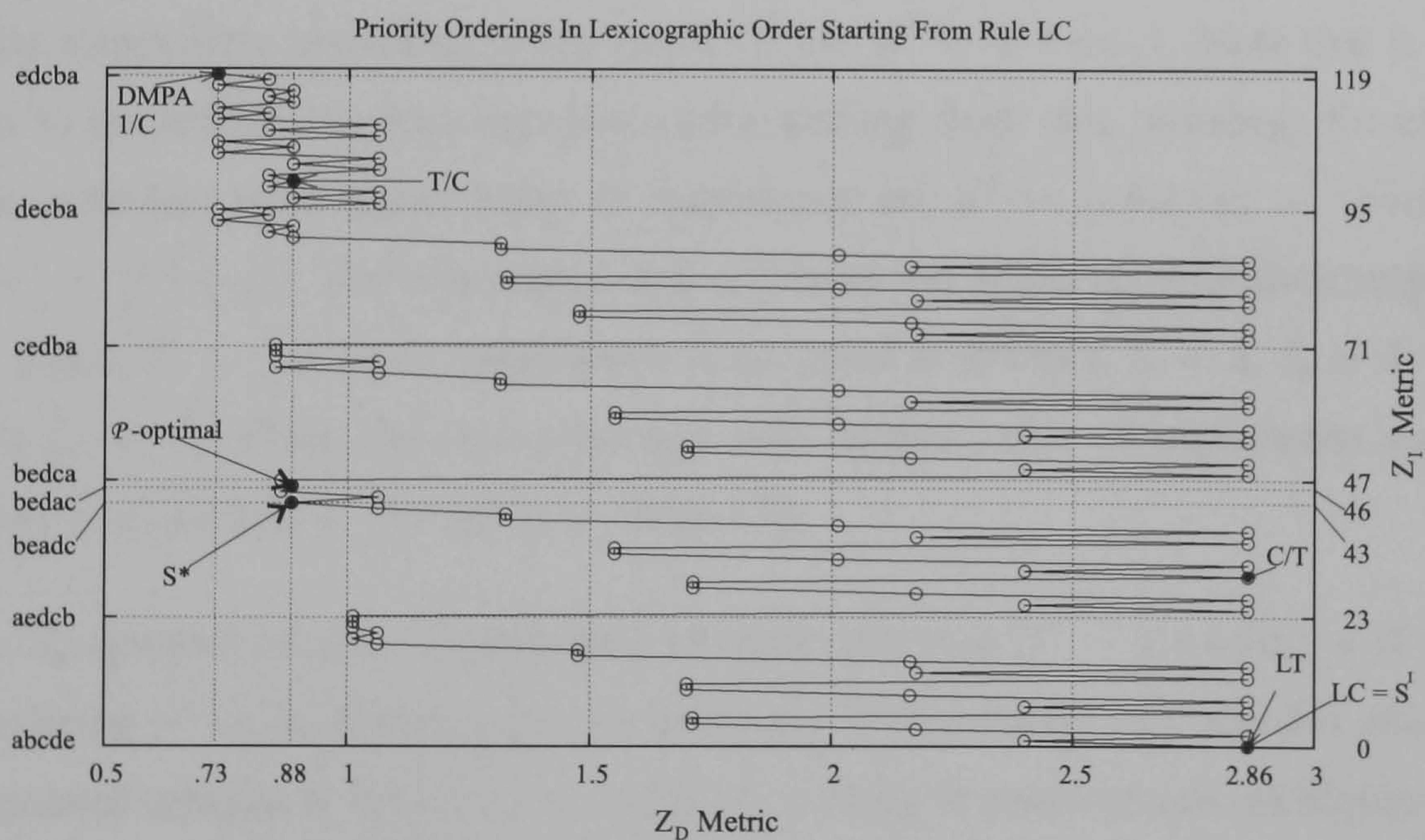


Figure 4.5: All priority assignments of set S_5 ordered lexicographically starting from the ordering $\langle a\,b\,c\,d\,e \rangle$ (i.e. the LC rule). Lines illustrate that following the lexicographic order the first feasible ordering found is the optimal S^*.

Observe that the lines connect the orderings following the lexicographic order; if we start from the point S^I and follow the lines, the first feasible ordering is $S^* = \langle b\,e\,a\,d\,c \rangle$. By simulating S^* on its hyperperiod, we compute 1189 preemptions. On the other hand,

the $\mathcal{P}$*-optimal* solution is computed by simulating all the priority orderings of S_5 on its hyperperiod. The $\mathcal{P}$*-optimal* solution corresponds to the ordering $\langle b\,e\,d\,a\,c\rangle$ with 1178 preemptions.

Thus, the Z_I*-optimal* solution S^* is the lexicographically closest solution to S^I but it is not the $\mathcal{P}$*-optimal* solution. However it is very close and therefore we can conjecture that solving the deadlines and importance scheduling problem with importance assigned by the rule LC is a good heuristic for the deadlines and preemptions scheduling problem.

Finally, note that Figure 4.5 and Figure 4.2 contain the same points. This is because the ordering S^I is the same in both examples. In effect, the task set S_5 ordered by the rule LC is $\langle a\,b\,c\,d\,e\rangle$, which coincides with the assignment of importance in Table 4.2. This permits to enumerate the lexicographic order easily by using the tasks identifiers (i.e. a, b, c, ..., etc). However, in the following experiment the ordering S^I is rather different and then we will identify the orderings by labeling tasks with the importance values.

Rule C/T as the assignment of importance. We repeat the above experiment but assigning importance according to the rule C/T; i.e. $S^I = \langle c\,b\,d\,a\,e\rangle$. Note that it is more difficult to enumerate the lexicographic order starting from this ordering; for example, the four orderings with higher index of importance are $S^I = \langle c\,b\,d\,a\,e\rangle \succ \langle c\,b\,d\,e\,a\rangle \succ \langle c\,b\,a\,d\,e\rangle \succ \langle c\,b\,a\,e\,d\rangle$. For this reason, we will label the orderings with their importance values. Since $S^I = \langle c\,b\,d\,a\,e\rangle$, importance is assigned as follows: $I_c = 5$, $I_b = 4$, $I_d = 3$, $I_a = 2$, $I_e = 1$. Thus, the four orderings with higher index of importance are $S^I = \langle 5\,4\,3\,2\,1\rangle \succ \langle 5\,4\,3\,1\,2\rangle \succ \langle 5\,4\,2\,3\,1\rangle \succ \langle 5\,4\,2\,1\,3\rangle$.

The Z_I*-optimal* solution is computed as above and then $S^* = \langle c\,e\,b\,d\,a\rangle = \langle 5\,1\,4\,3\,2\rangle$. By simulating S^* on its hyperperiod, we compute 1278 preemptions. On the other hand, the $\mathcal{P}$*-optimal* solution is $\langle b\,e\,d\,a\,c\rangle = \langle 4\,1\,3\,2\,5\rangle$ with 1178 preemptions. In Figure 4.6, we can see that S^* has moved away from $\mathcal{P}$*-optimal*. It indicates that the heuristic LC (1189 preemptions) was better than C/T in this particular scenario.

Thus, we conjecture that a heuristic for solving the deadlines and preemption problem may be obtained by assigning importance according to the LC rule and solving the scheduling problem with deadlines and importance. In chapter 7, we perform experiments with a number of task sets that confirm this observation.

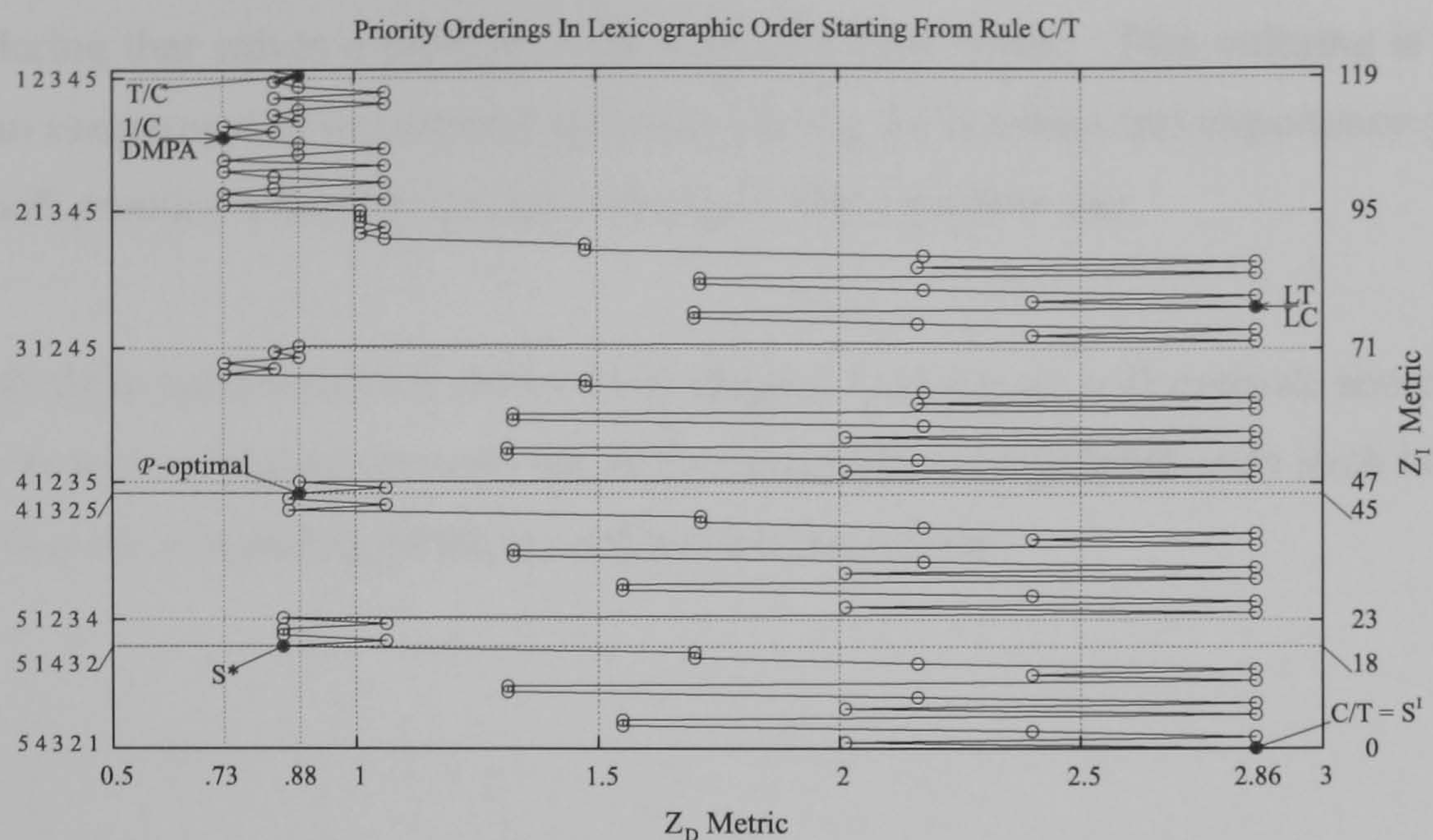


Figure 4.6: All priority assignments of set S_5 ordered lexicographically starting from the ordering $\langle c\,b\,d\,a\,e\rangle$ labelled as $\langle 5\,4\,3\,2\,1\rangle$ (i.e. the C/T rule). $S^* = \langle c\,e\,b\,d\,a\rangle = \langle 5\,1\,4\,3\,2\rangle$ and the $\mathcal{P}$*-optimal* is $\langle b\,e\,d\,a\,c\rangle = \langle 4\,1\,3\,2\,5\rangle$

4.8.3 Summary

In these examples, we show that there exist feasible priority orderings that are Z*-optimal* at task level as well as at application level. For small tasks sets, these solutions can be found by simulating the $N!$ priority orderings. Naturally, it is not a viable solution for many practical problems. Finding $\mathcal{P}$*-optimal* or $\mathcal{J}^{rel}$*-optimal* solutions seem to be NP-hard scheduling problems.

The examples also show that the concept of importance can be correlated with these two QoS metrics such that solving the problem with importance produce similar results as solving the problem with the QoS metrics. While in the above examples, the deadlines and importance problem was solved by exhaustive search, in chapter 5 we will show that it can be solved in pseudo-polynomial time by the DI algorithm. By this way, we propose a tractable approach for some NP-hard problems.

- At the task level, we propose to raise the priorities of tasks with a specific QoS metric by assigning higher importance to such tasks and solving the deadline and importance problem.
- At the application level we propose to use heuristics for determining a priority or-

dering that solves a problem with a specific QoS metric. This ordering is used as an assignment of importance and then solving the deadline and importance problem will produce a feasible solution similar to the infeasible one.

Both these approaches are explored in chapter 7, where we will evaluate some heuristics for assigning importance and we will evaluate the DI algorithm with such heuristics for solving the scheduling problems defined in this chapter.

Chapter 5

Fixed-Priority Scheduling with Deadlines and Importance: The DI Algorithm

5.1 Introduction

In this chapter we present the DI algorithm that solves the scheduling problem defined in section 4.6.1 (page 68): *Given a task set S and an ordering S^I, to find S^* which is an ordering of fixed-priorities that is feasible and is the closest one to S^I.*

Roughly speaking, the idea of the DI algorithm consists on starting from the ordering by importance S^I and then checking its schedulability. As the system may not be feasible, go to the next importance ordering (maintaining the minimal lexicographic distance) by swapping tasks and check for schedulability again. Keep on this process until the system is made schedulable or at least the deadline monotonic ordering is reached.

Note that this process is similar to the swapping algorithm. However, assuming that an ordering is represented as a vector where the leftmost task is the most important and the rightmost is the least important one, the swapping algorithm swaps tasks on right-to-left order stopping when a feasible ordering is found. On the other hand, the DI algorithm swaps tasks on left-to-right order following the lexicographic order and keeping the minimal distance to S^I.

5.2 Process Model

We use the model described in section 4.2 with the following additional notations:

- Any ordering $\langle a\,b\,c\ \ldots j\,k\ \ldots\ x\,y\,z\rangle$ can be represented as $\langle \phi\ \omega\rangle$ where the prefix ϕ is $\langle a\,b\,c\ \ldots j\rangle$ and the suffix ω is $\langle k\ \ldots x\,y\,z\rangle$. The meta-ordering $\langle \phi\ *\rangle$ denotes all the orderings in $\hat{S}$ starting with prefix ϕ.
- For any $A \subseteq S$, $\delta(A)$ is an ordering with priorities assigned according to the Deadline Monotonic Priority Assignment where the function δ orders A according to the relation *"has a shorter deadline than"*. S^D is the ordering by DMPA; i.e. $S^D = \delta(S)$.
- For $\alpha \in \hat{S}$, the function $\mathcal{F}(\alpha)$ returns *true* when α is feasible, i.e. the ordering α passes the FPS schedulability test; otherwise returns *false*.

5.3 Problem Description

Consider the example presented in section 4.8.1 (page 75) where a hypothetical system with several QoS requirements must be implemented. The space of all possible fixed-priority orderings for the task set S_5 is plotted in Figure 5.1. It shows all $5! = 120$ priority orderings of S_5 plotted with Z_D and Z_I metrics.

Observe that S^D is feasible but it has low index of importance. S^I is infeasible since task e has a response time of 229 but a deadline of 80 and therefore it is not schedulable. The Z_I*-optimal* solution $S^* = \langle b\,e\,a\,d\,c\rangle$ has an index of importance of 44. Figure 5.2 shows the response time for these priority assignments.

We observe that an approach for solving this problem consists on utilising the Swapping algorithm. The Swapping algorithm works receiving an infeasible priority ordering and finding a feasible one (if it exists) by swapping pairs of tasks on right-to-left order and stopping when a feasible ordering is found. In our example, the swapping algorithm will receive the infeasible ordering by importance $S^I = \langle a\,b\,c\,d\,e\rangle$.

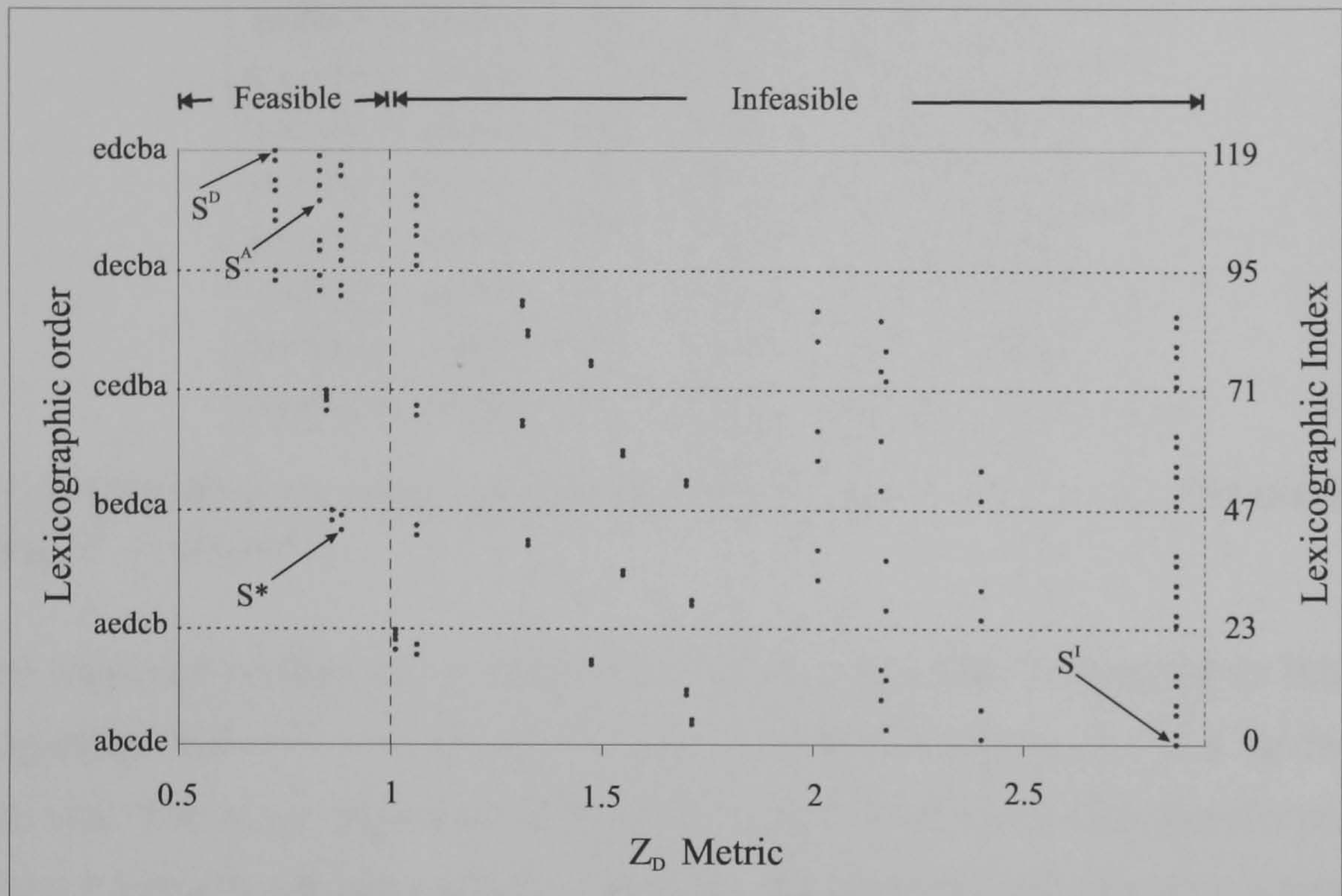


Figure 5.1: Plotting all priority orderings of S_5. S^I is not feasible. S^D is feasible but has low importance metric. S^A is the Swapping algorithm. The optimal bicriteria S^* is $\langle b\, e\, a\, d\, c \rangle$

Task set S_5				
τ	T	D	C	I
e	100	80	13	1
d	240	240	37	2
c	330	330	55	3
b	350	350	56	4
a	480	400	68	5

S^D	
τ	R
e	13
d	50
c	118
b	174
a	292

S^I	
τ	R
a	68
b	124
c	179
d	216
e	**229**

S^A	
τ	R
e	13
a	81
b	150
d	187
c	292

Optimal S^*	
τ	R
b	56
e	69
a	150
d	187
c	292

Figure 5.2: Task set S_5 and priority orderings S^D (DMPA), S^I (Importance), S^A (swapping algorithm with S^I as input), and the optimal S^*. The lexicographic order is $S^I \succ S^* \succ S^A \succ S^D$

Sequence	*R*	*Feasible*	*next*	*swap*
testing e: abcde	229	NO	$\Rightarrow$	$d \leftrightarrow e$
testing d: abced	255	NO	$\Rightarrow$	$c \leftrightarrow d$
testing c: abdec	292	YES	$\Rightarrow$	fix c
testing e: abdec	174	NO	$\Rightarrow$	$e \leftrightarrow d$
testing d: abedc	187	YES	$\Rightarrow$	fix d
testing e: abedc	137	NO	$\Rightarrow$	$b \leftrightarrow e$
testing b: aebdc	150	YES	$\Rightarrow$	fix b
testing e: aebdc	81	NO	$\Rightarrow$	$a \leftrightarrow e$
testing a: eabdc	81	YES	$\Rightarrow$	fix a
testing e: **eabdc**	13	YES	$\Rightarrow$	fix e

Table 5.1: Executing the swapping algorithm with input $S^I = \langle a\, b\, c\, d\, e \rangle$. The result is the ordering $S^A = \langle e\, a\, b\, d\, c \rangle$

The sequence of steps executed by the swapping algorithm is depicted in Table 5.1. The algorithm finds $S^A = \langle e\, a\, b\, d\, c \rangle$, which is a feasible solution but it is far from the optimal one. This is no surprising because it was not developed for this specific problem. However, it could be the case that the swapping algorithm could find a good approximation to the optimal and therefore it may be considered as a heuristic for the deadline and importance problem.

We can figure out a solution for finding S^* looking carefully at Figure 5.3; it contains the same data as Figure 5.1 but the points are connected successively following the lexicographic order. Note that there are groups of either feasible or near feasible orderings (enclosed by a circle) distributed around the indices 23, 47, 71, 95 and 119. In addition note on the left side, which orderings correspond to such points. The index 23 corresponds to $\langle a\, e\, d\, c\, b \rangle$ where $\langle a \rangle$ is the most important task plus a suffix $\langle e\, d\, c\, b \rangle$ ordered by DMPA. The index 47 is $\langle b\, e\, d\, c\, a \rangle$ where $\langle b \rangle$ is the second most important task plus a suffix ordered by DMPA. The rest have the same pattern: a prefix ϕ plus a suffix ω ordered by DMPA. Our algorithm uses this pattern to reduce the search space.

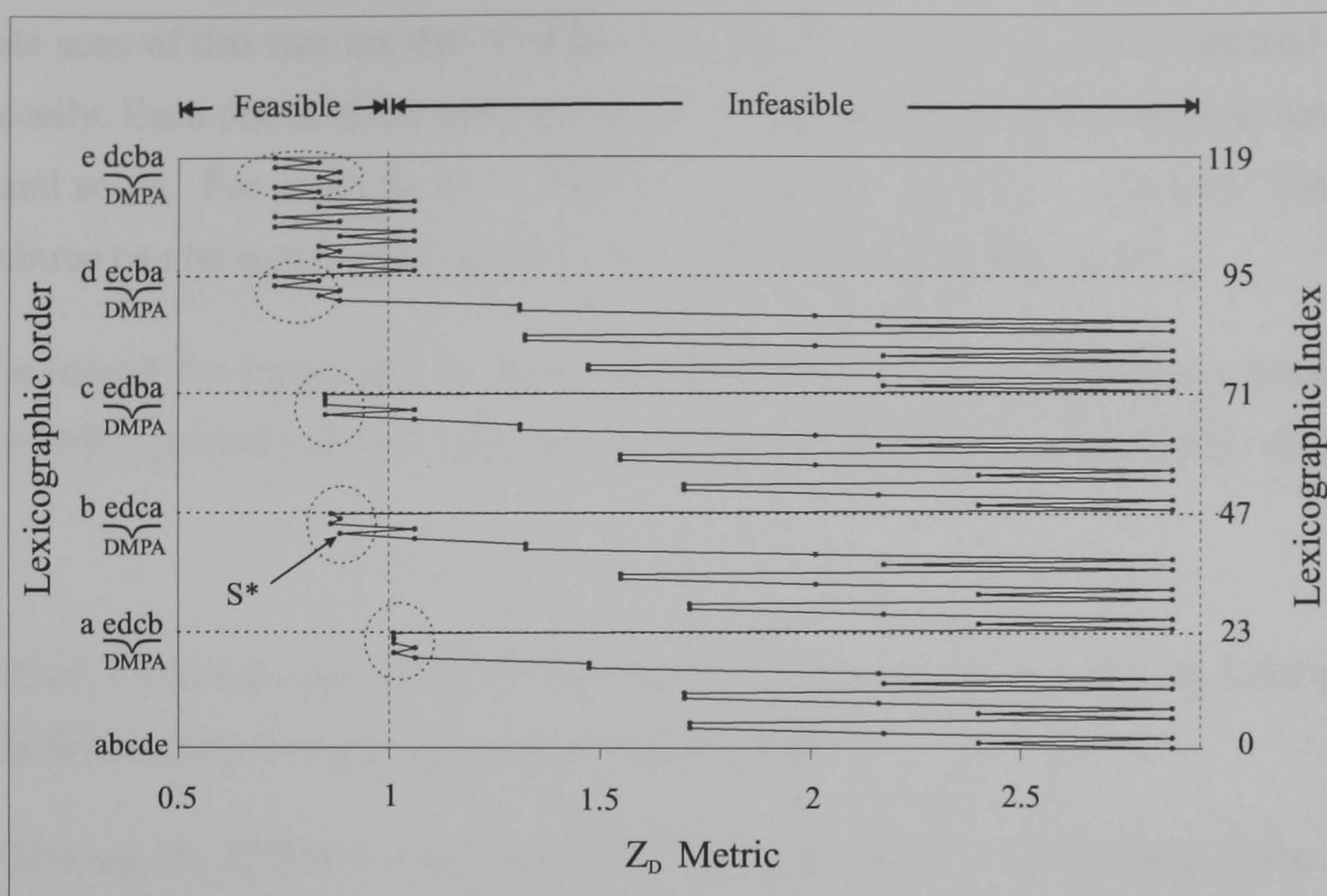


Figure 5.3: Similar to Figure 5.1 but the priority orderings are connected according to their lexicographic order. Circles enclose groups of either feasible or near feasible orderings distributed around the indices 23, 47, 71, 95 and 119. The orderings corresponding to these indices have a similar pattern: a task plus a suffix ordered by DMPA.

5.4 Solving the Deadlines and Importance Problem

This section is divided in two parts:

1. In the first one we will show how the set $\hat{S}$ of all possible priority assignments can be organized into a tree, and how whole subtrees can be skipped by identifying some local minimums.
2. In the second one, we present the algorithm DI, which performs a branch and bound search into the tree defined. In the worst-case it performs $\frac{N^2+N}{2}$ steps to find the solution. The algorithm is described and its optimality is proved.

5.4.1 Organizing the Search Space

The search space consists of the set of all possible priority orderings $\hat{S}$ ordered lexicographically and organized as a tree. The root of the tree is $S^I = \langle a\, b\, c \ldots z \rangle$. The im-

mediate sons of this tree are the N subtrees $\langle a\,*\rangle$, $\langle b\,*\rangle$, $\langle c\,*\rangle, \ldots, \langle z\,*\rangle$ ordered lexicographically. Each one of these subtrees has $N-1$ sub-subtrees where each one has $N-2$ ones and so on. For example, $\langle a\,*\rangle$ has $\langle a\,b\,*\rangle$, $\langle a\,c\,*\rangle$, $\langle a\,d\,*\rangle, \ldots, \langle a\,z\,*\rangle$. Therefore, any subtree can be represented as $\langle \phi\,*\rangle$. A *vertex* is represented as $\langle \phi\,\omega\rangle$.

The reason for organizing the space into subtrees with this configuration comes from an interesting property related with the optimality of DMPA and the generality of the FPS test:

- First, DMPA is optimal in the sense that if a set is schedulable by any fixed-priority ordering then it also is schedulable under DMPA.
- Second, the FPS test is necessary and sufficient and it is independent of the assignment of priorities.

Consider an ordering $S^I = \langle a\,b\,c\,d\rangle$ and an ordering $S^D = \langle d\,c\,b\,a\rangle$. Suppose that we apply the FPS test to S^I and we find that S^I is feasible. The optimality of DMPA indicates that it is not necessary to apply the test to S^D because if S^I is feasible then S^D is also feasible; but the contrary is not true, if S^D is feasible we cannot assure anything about the feasibility of S^I or any other combination. However we can affirm that if S^D is infeasible then any ordering of these four tasks will also be infeasible.

Now, consider that we extend S^D with an arbitrary task such that we have an ordering $\langle x\,d\,c\,b\,a\rangle$ (remember that $\langle d\,c\,b\,a\rangle$ is ordered by DMPA) and we apply the FPS test. If $\langle x\,d\,c\,b\,a\rangle$ is feasible, then there are some orderings in $\langle x\,*\rangle$ that are feasible. However, if $\langle x\,d\,c\,b\,a\rangle$ is infeasible we can affirm that no ordering in $\langle x\,*\rangle$ is feasible.

Note that $\langle x\,d\,c\,b\,a\rangle$ consists of a prefix $\langle x\rangle$ and a suffix $\langle d\,c\,b\,a\rangle$ which is ordered by DMPA. This can be denoted as a vertex $\langle \phi\,\delta(\omega)\rangle$ where ϕ and ω are arbitrary sub-orderings and δ is a function that re-order ω according to DMPA. Using this notation, the next theorem resumes the above observations.

Theorem 5.4.1 (Prefix Theorem). *If the ordering $\langle \phi\,\delta(\omega)\rangle$ is infeasible then any ordering in $\langle \phi\,*\rangle$ is also infeasible.*

Proof. By contradiction, suppose that there exist a feasible ordering S' in $\langle \phi \ast \rangle$, which is not $\langle \phi\, \delta(\omega) \rangle$ because it is infeasible.

Thus, $S' = \langle \phi\ \omega \rangle$ is feasible and $S^\delta = \langle \phi\ \delta(\omega) \rangle$ is infeasible. Both orderings share the same prefix ϕ and differ only by the suffix.

Note that if ω is empty or has only one task then $\omega = \delta(\omega)$ and hence $S' = S^\delta$. Consequently, S^δ *is feasible* and this contradicts the hypothesis. Otherwise, the feasibility of S' and the infeasibility S^δ depend only on the order of tasks in ω; all tasks in ϕ are not affected by any change in ω.

The proof of optimality of DMPA and the proof of optimality of DMPA in the presence of blocking with resources managed under the priority ceiling protocol shows us that, because S' is feasible, exchanging any pair of tasks (in ω) non-ordered by shorter deadline give us a feasible ordering S''. Following the same process we obtain orderings $S''', S'''', \ldots, S^n$ which are also feasible. The last one is S^n with its ω ordered by DMPA. S^n is identical to S^δ and therefore S^δ *is feasible.* This contradicts the hypothesis and concludes the proof. □

Thus, we can know if all the orderings in $\langle \phi \ast \rangle$ are infeasible by testing just one of them, the one with ω ordered by DMPA, i.e. $\langle \phi\, \delta(\omega) \rangle$. In addition, this theorem produces the next corollary that will be used to prove the optimality of the algorithm proposed.

Corollary 5.4.2. *If an ordering $\langle \phi\ \omega \rangle$ is feasible then $\langle \phi\ \delta(\omega) \rangle$ is also feasible.*

The Prefix theorem 5.4.1 affirms that in a tree search, we can discard a whole subtree if its root is infeasible; otherwise we need to look inside it. This process is applied recursively to each subtree performing, by this way, a branch and bound search but only on specific nodes.

5.4.2 The Algorithm DI

The algorithm DI (Deadline & Importance) examines the root of subtrees in lexicographic order from the closest to S^I to the remotest one and from the top to the bottom.There are two preconditions that must be fulfilled:

- The ordering S^D must be feasible; otherwise no feasible solution exists.
- The ordering S^I must be infeasible; otherwise we do not need to perform a search because S^I is the solution.

The next variables are used:

- ϕ is a vector of tasks representing the prefix of the actual subtree indexed.
- ω is a vector of tasks which is the complement of ϕ.
- k indexes the kth task of ω such that $\omega[k]$ is a task acquired orderly from left to right; i.e. from the highest importance important task to the least important one in the lexicographic order.
- τ is $\omega[k]$. τ is used to build a new prefix $\langle\phi\,\tau\rangle$ for indexing the subtree $\langle\phi\,\tau\,*\rangle$.
- $delete(\alpha, \tau)$ is a function that delete a task τ from the ordering α.
- ω_δ is $\delta(\omega)$; i.e. ω ordered by DMPA.
- $\omega_{\delta-\tau}$ is ω_δ without τ. When a new prefix $\langle\phi\,\tau\rangle$ is created, its corresponding suffix is $\omega_{\delta-\tau}$.
- S^*_{test} is the root to be tested.
- $\mathcal{F}$ is a function that returns true, if an ordering passes the FPS schedulability test.

Let S be a task set with orderings S^D and S^I, feasible and infeasible respectively. The algorithm builds keys to index each subtree in lexicographic order. A key is built by appending to ϕ a task τ (step 3) from ω. The lexicographical order is achieved following the sequence in ω which is S^I. The key indices the subtree $\langle\phi\,\tau\,*\rangle$ and hence we build its root $S^*_{test} = \langle\phi\,\tau\;\delta(\omega_{\delta-\tau})\rangle$ (steps 4-5). S^*_{test} is tested (*step 6*) and if it is feasible, it is saved as a partial solution S^*, τ is deleted from both ω and ω_δ, ϕ is updated and the index k is reset (*steps 7-11*); otherwise, k is advanced to the next τ. DI stops when there are not more subtrees to visit. Note that the worst-case is when the only solution is S^D and its order is contrary to S^I. In this case, the root of the first N subtrees will be tested and the

Algorithm 2 DI (*Deadline and Importance*)

Require: S^D feasible, S^I infeasible
1: Set $\phi \Leftarrow \emptyset, \omega \Leftarrow S^I, \omega_\delta \Leftarrow S^D, k = 0$
2: **while** SizeOf(ω) > 1 **do**
3: let $\tau \Leftarrow \omega[k]$
4: let $\omega_{\delta-\tau} \Leftarrow delete(\omega_\delta, \tau)$
5: build $S^*_{test} \Leftarrow \langle \phi \; \tau \; \omega_{\delta-\tau} \rangle$
6: **if** $\mathcal{F}(S^*_{test})$ **then**
7: $S^* \Leftarrow S^*_{test}$
8: $\phi \Leftarrow \langle \phi \; \tau \rangle$
9: $\omega \Leftarrow delete(\omega, \tau)$
10: $\omega_\delta \Leftarrow \omega_{\delta-\tau}$
11: $k \Leftarrow 0$
12: **else**
13: $k \Leftarrow k + 1$
14: **end if**
15: **end while**

N^{th} will be $S^*_{test} = S^D$; afterwards, the next level of $N - 1$ subtrees will be tested and the $(N-1)^{th}$ will be again $S^*_{test} = S^D$ and so on. Thus, in the worst-case, the solution will always be S^D. DI will always stop due to the fact that in the worst-case an ordering by DMPA is built and therefore ω is always reduced.

Complexity. The worst-case occurs when the only solution is S^D and its order is contrary to S^I. In such case the loop is executed N times and ω is reduced to $N - 1$ elements; afterward the loop is executed $N - 1$ times and ω is reduced to $N - 2$ elements and so on. Therefore the loop executes $N + (N-1) + (N-2) + \ldots + 1 = \frac{N^2+N}{2}$ times. At each iteration, a feasibility test of complexity E is performed and therefore the complexity of the algorithm DI is $O(E \times \frac{N^2+N}{2})$. Thus, the algorithm finds the solution in a polynomial number of steps but the total complexity depends on the complexity E of the response-time test, which is pseudo-polynomial.

DI proof

Theorem 5.4.3. *The algorithm* DI *yields an optimal solution for the problem 1 of guaranteeing all deadlines and minimizing the distance to* S^I.

Proof. By contradiction. Suppose that there exists an ordering S^o which is the optimal

solution; i.e. it is both feasible and is lexicographically closest to S^I. Therefore, the solution S^* found by DI is not the optimal solution one and then:

- 1) $S^I \succ S^* \succ S^o$, if S^* is infeasible. This is, the solution S^* is lexicographically closest to S^I but it is infeasible.
- 2) $S^I \succ S^o \succ S^*$, if S^* is feasible. This is, the solution S^* is feasible but S^o is both feasible and lexicographically closest to S^I.

Clearly, the first case is not true because the DI algorithm starts with a feasible ordering and in all their iterations the algorithm accepts only feasible solutions. Consequently, S^* is always feasible. Thus, we need to consider only the second case.

Note the following facts about S^o and S^*:

- They are lexicographically different but before a task j they are identical; i.e. they share the same prefix $\phi = \langle a\, b\, c\, \ldots\, j \rangle$ and differ on their suffix.

$$S^o = \langle \phi\ \omega^o \rangle \ \textit{where}\ \omega^o = \langle k \ldots l \ldots \rangle$$

$$S^* = \langle \phi\ \omega^* \rangle \ \textit{where}\ \omega^* = \ \langle l \ldots k \ldots \rangle$$

  If ϕ is empty then they are different from the first element.

- Note that $k \succ l$ because $S^o \succ S^*$, we
- S^o and S^* share the same vertex $\langle \phi\ \omega_\delta \rangle$ where $\omega_\delta = \delta(\omega^o) = \delta(\omega^*)$.
- S^o is feasible and therefore $\langle \phi\ \omega_\delta \rangle$ is also feasible (corollary 5.4.2). Thus, in the nth iteration when DI finds $\langle \phi\ \omega_\delta \rangle$, it is accepted and stored as the nth partial solution $S^*_n = \langle \phi\ \omega_\delta \rangle$. Consequently, the partial solution S^*_n and the optimal solution S^o share the same prefix and differ only by the suffix (ω_δ and ω^o respectively).

DI builds and tests new orderings by joining the actual prefix ϕ with a task τ and ordering the rest of the tasks by DMPA (steps 3-6). Once that DI builds the partial solution S^*_n, the next task to be tested is either k or l. Since $k \succ l$, the next task to be

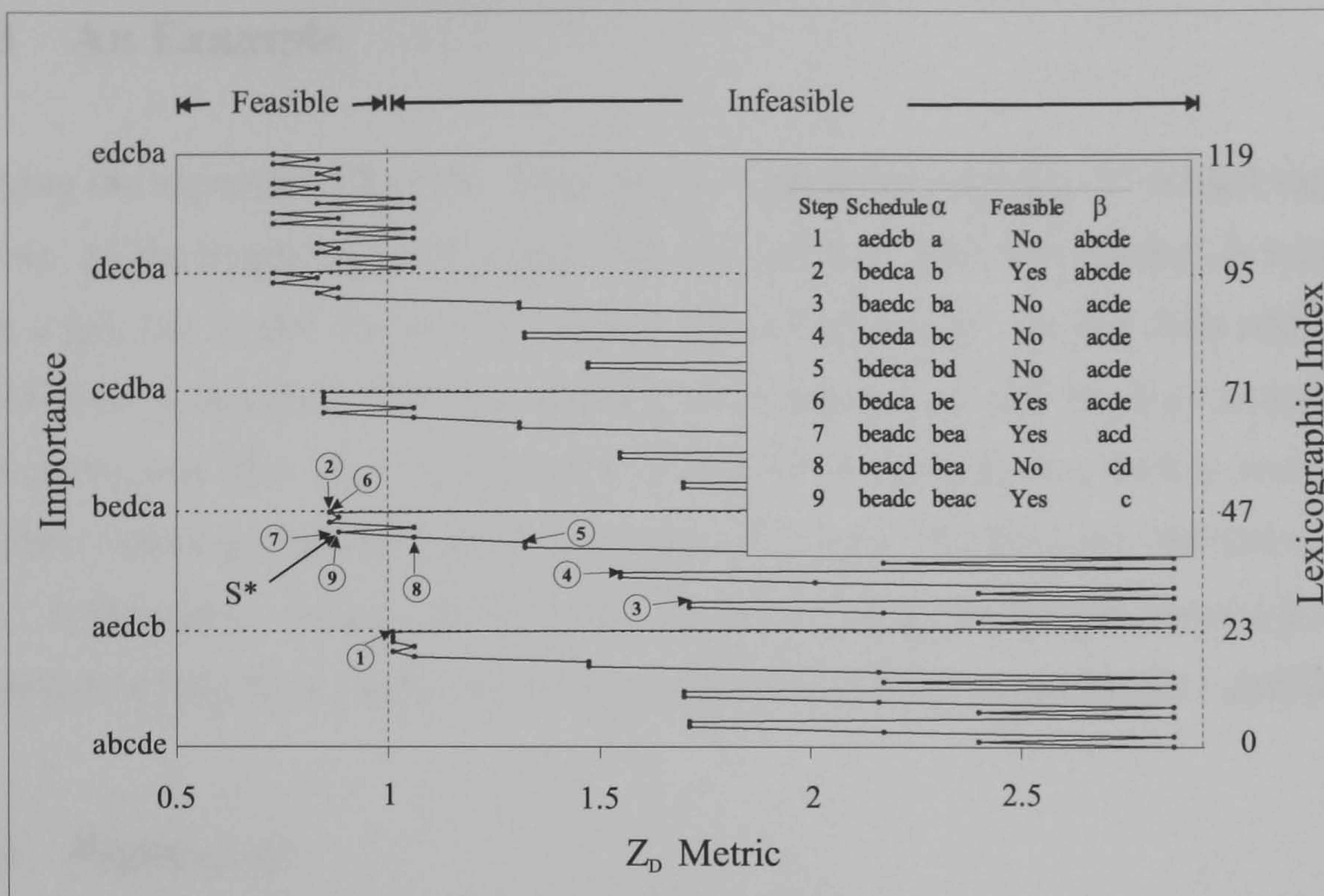


Figure 5.4: DI applied to S_5 showing the sequence of operations.

appended to the prefix ϕ is k, and therefore the ordering $\langle \phi\ k\ \delta(\ldots l \ldots)\rangle$ is tested before that $\langle \phi\ l\ \delta(\ldots k \ldots)\rangle$, which is in the path of the solution S^*.

The ordering tested is

$$S^*_{test} = \langle \phi\ k\ \delta(\ldots l \ldots)\rangle$$

Two cases may occur:

- If S^*_{test} is feasible, DI will accept it and never will find S^* because they both are in different paths. This is not possible because S^* is found by DI.
- If S^*_{test} test is infeasible then S^o must be infeasible since both share the same prefix (Theorem 5.4.1). Therefore, S^o is lexicographically closest to S^I but it is infeasible. This contradicts the hypothesis.

This concludes the proof.

□

5.4.3 An Example

Applying the algorithm DI to the tasks set S_5, it finds the ordering S^* in nine steps (Figure 5.4). At the beginning, ϕ is empty and ω is $\langle abcde \rangle$. The first element is acquired to create a key $\langle a \rangle$ to test the vertex $\langle a\, edcb \rangle$ (① in the graph); the test fails and then the subtree $\langle a\, * \rangle$ is skipped. The next element in ω is obtained and $\langle b\, edca \rangle$ is tested (②); it is feasible and then $\langle b \rangle$ is appended to ϕ and $\langle b \rangle$ is deleted from both ω and ω_δ. The next three orderings tested (③,④,⑤) are infeasible, and therefore only the index k is updated. Afterwards, $\langle b\, edca \rangle$ and $\langle b\, eadc \rangle$ are tested successively and saved (⑥,⑦); the next ordering fails (⑧) and the next one passes the test (⑨). ω has length 1 and DI stops.

5.4.4 Summary

In the context of fixed priority scheduling, we present the DI algorithm which finds the optimal solution to the deadlines ($D \leq T$) and importance problem defined in section 4.6.1. This problem consists on finding a feasible priority ordering that minimises the lexicographic distance to the ordering S^I. Our approach is optimal in the sense that the priority ordering found by DI is feasible and no other feasible priority ordering exists closest to S^I.

The DI algorithm performs a branch and bound search into the space formed by the $N!$ possible orderings. This space of solutions is ordered lexicographically by importance. Due to some properties of the optimality of the deadline monotonic priority assignment, and some particular characteristics of the lexicographic order, we identify patterns into the search space that permits to solve the problem in $O((N^2 + N)/2)$ steps; however the complexity of DI depends on the complexity of the fixed-priority feasibility test. In our case, the complexity of DI is pseudo-polynomial because it uses the response time analysis test. In chapter 7, the DI algorithm will be evaluated.

Chapter 6

Fixed-Priority Scheduling with Deadlines and Conditional Importance: The DI+ Algorithm

6.1 Introduction

In this chapter, importance is extended to include conditional relative importance, which is a natural generalization of simple relative importance. Conditional relative importance expresses preferences for executing a task with regard to other ones in the system subject to conditional clauses. This concept is based on preferential statements of the form "it is more important to execute α than β if some conditions are fulfilled". Observe that the statement is formed by a relative importance statement plus a conditional statement. Thus, conditional relative importance will be represented as a number I plus one or more conditional clauses. This concept allows implementing priority relationships constraints such as precedence constraints and allows including importance as a hard requirement.

We formulate the scheduling problem with deadlines and conditional importance and introduce the DI+ algorithm for solving it. DI+ is DI with some modifications. In particular, the feasibility test and the priority constraint test are grouped into a single function that is called as a subroutine. This does not affect the logic of the original DI algorithm and allows including the feasibility tests for tasks with arbitrary deadlines or weakly-hard constraints, as well as including the swapping algorithm for finding feasible priority orderings when DMPA is not optimal.

6.2 Conditional Relative Importance

When specifying a system, there are situations where a particular QoS requirement can be mandatory (e.g. a safety requirement) or can be conditioned to fulfill other requirements (e.g. a performance requirement conditioned to a reliability one). During the design, these mandatory and/or conditioned QoS requirements must be mapped to the task set. In this case, there exist dependence relationships among tasks that are compulsory for assuring the system correctness.

In order to cope with situations where dependence relationships exist, we include conditional relative importance statements in our model. A *Conditional Relative Importance statement* has the form

$$x \textit{ is more important than } y \textit{ only if } \lambda \textit{ holds}$$

which means that the relative importance of x and y is *conditioned* to a set of λ conditions. Note that when λ is empty, a simple relative importance statement is obtained.

The statement "it is more important to execute x before y only if both z precedes x and z precedes y", is an example of conditional relative importance statement expressing precedence constraints. Note that the set of constraints λ is the union of a conditional clause associated to x (*"z precedes x"*) and other associated to y (*"z precedes y"*).

Definition (Conditional Relative Importance). *Conditional relative importance is a relation of relative importance $\succ$ in S and a set of constraints λ such that for any pair of tasks x and y, with sets of constraints λ_x and λ_y respectively*

"$x \succ y$, λ_{xy}" is interpreted as *"x is more important than y only if λ_{xy} holds".*

where $\lambda_{xy} = \lambda_x \cup \lambda_y$. In other words, x is more important than y only if the constraints λ_x and λ_y are fulfilled. The set of constraints of a task can have zero or more clauses.

Similar to relative importance (see section 4.5.1, page 63), conditional relative importance have to maintain the strict total order properties; i.e. for all x, y and z in S:

- It is not the case that $x \succ x, \lambda_x$ (irreflexive).
- If $x \succ y, \lambda_{xy}$ then it is not the case that $y \succ x, \lambda_{yx}$ (asymmetric).
- If $x \succ y, \lambda_{xy}$ and $y \succ z, \lambda_{yz}$ then $x \succ z, \lambda_{xz}$ (transitive).
- Exactly one of $x \succ y, \lambda_{xy}$ or $y \succ x, \lambda_{yx}$ is true.

Thus, conditional relative importance permits to compare any pair of tasks and to assign a different natural number I to each task such that the relation is preserved.

6.2.1 Task Conditional Importance

Definition 6.2.1 (Task Conditional Importance). *Task conditional importance is a pair $\bar{I} = (I, \lambda)$ constituted by a natural number I and a set of constraints λ associated to the task. Thus, for any pair of tasks x and y with constraints λ_x and λ_y, there exist task conditional importance $\bar{I}_x = (I_x, \lambda_x)$ and $\bar{I}_y = (I_y, \lambda_y)$ respectively (with $I_x \neq I_y$) such that*

$$x \succ y \,,\, \lambda_{xy} \Rightarrow \bar{I}_x > \bar{I}_y$$

where $\lambda_{xy} = \lambda_x \cup \lambda_y$ and $\bar{I}_x > \bar{I}_y$ if and only if $I_x > I_y$

Task conditional importance expresses a preference for executing a task with respect to the other tasks in the system only if the constraints λ are fulfilled. Observe that the constraints λ must be selected to maintain the uniqueness of all I such that $\forall x, y \in S,\ x \neq y \Rightarrow I_x \neq I_y$. Thus, tasks can be compared and ordered totally according to their conditional relative importance.

With respect to the constraints, we note the following:

- In a set of two tasks $S_2 = \{a, b\}$ the ordering $\langle a\ b \rangle$ is formed when $\bar{I}_a > \bar{I}_b$ and then the set of constraints is λ_{ab}. The ordering $\langle b\ a \rangle$ is formed when $\bar{I}_b > \bar{I}_a$ and then the set of constraints is λ_{ba}. Since $\lambda_{ab} = \lambda_{ba}$, set of constraints for any ordering of S_2 is the union of the constraints of each task.

- In a set of three tasks $S_3 = \{a, b, c\}$, the ordering $\langle a\ b\ c \rangle$ is formed when $\bar{I}_a > \bar{I}_b$ and $\bar{I}_a > \bar{I}_c$ and $\bar{I}_b > \bar{I}_c$, and therefore, the set of constraints is $\{\lambda_{ab} \cup \lambda_{ac} \cup \lambda_{bc}\}$, which is $\{\lambda_a \cup \lambda_b \cup \lambda_a \cup \lambda_c \cup \lambda_b \cup \lambda_c\} = \{\lambda_a \cup \lambda_b \cup \lambda_c\}$. Clearly, for other orderings of S_3, the set of constraints is also the union of the constraints of each task; i.e. $\{\lambda_a \cup \lambda_b \cup \lambda_c\}$.

Thus, by generalizing the above observations: given a task set $S = \{a, b, c, \dots\}$ with constraints $\{\lambda_a, \lambda_b, \lambda_c, \dots\}$, the set of constraints of any ordering α in $\hat{S}$ is the union of the constraints of each task, this is, $\Lambda = \{\lambda_a \cup \lambda_b \cup \lambda_c \cup \dots\}$.

6.2.2 Index of Conditional Importance $Z_{\bar{I}}$

Similar to the index of relative importance, the set of $N!$ possible permutations of S form a set $\hat{S}$, which can be ordered lexicographically and indexed by conditional importance. The ordering with the highest conditional importance is $S^{\bar{I}}$ such that all orderings in $\hat{S}$ can be compared and ordered lexicographically with respect to $S^{\bar{I}}$ using the lexicographic index as defined in section 4.5.2:

$$\alpha \succ \beta \Leftrightarrow Z_{\bar{I}}(\alpha) < Z_{\bar{I}}(\beta)$$

The lexicographic index ranks all orderings with respect to $S^{\bar{I}}$ and therefore:

Definition 6.2.2 (Index of Conditional Importance $Z_{\bar{I}}$). *The lexicographic index defines an index of conditional importance $Z_{\bar{I}}$ among all orderings in $\hat{S}$ such that for any orderings $\alpha, \beta \in \hat{S}$ and Λ:*

$$\text{"}\alpha \textit{ is more important than } \beta \textit{ only if } \Lambda \textit{ holds"} \Leftrightarrow Z_{\bar{I}}(\alpha) < Z_{\bar{I}}(\beta)$$

Thus, the index of importance expresses a preference for executing a particular ordering with respect to the other orderings in the set of possibilities $\hat{S}$ subject to Λ constraints.

Since it is assumed that all conditional importances $\bar{I}$ are different, the index of importance is unique; i.e. $\forall \alpha, \beta \in \hat{S}, \alpha \neq \beta \Rightarrow Z_I(\alpha) \neq Z_I(\beta)$. Thus, orderings can be compared and organized in unique levels of importance that reflects a level of QoS where index 0 is the best and the quality decreases as the index increases.

6.2.3 Problem: Deadlines and Conditional Importance

In addition to meeting deadlines ($D \leq T$), in some hard real-time systems fulfilling both hard and soft QoS requirements non-related with the deadlines is an issue. It is our thesis that in some cases, hard and soft QoS requirements can be expressed in terms of conditional importance. A scheduling problem with deadlines and conditional importance can be formulated as follows:

Scheduling Problem 8. *Given a task set S, a set of constraints Λ and an ordering $S^{\bar{I}}$, to find S^* which is an ordering of fixed-priorities that is feasible, meets Λ, and is the closest one to $S^{\bar{I}}$.*

Note that if the constraints Λ are removed, the problem is reduced to the deadlines and importance problem.

6.3 Conditional Importance in Scheduling Problems

Conditional relative importance is a natural generalization of simple relative importance statements but they can be rather difficult to express in the context of priority-based scheduling. Therefore, Λ will be limited to a set of simple restrictions for assigning priorities. These restrictions will be called Λ-constraints . For instance, given tasks x and y and z with priorities P_x, P_y and P_z respectively, a statement with $\Lambda = \{P_y > P_z\}$ could be "x is more important than y only if $P_y > P_z$".

6.3.1 Λ-constraints

Λ-constraints is a set of hard priority constraints λ denoting statements of the form

$$\text{“}P_a > P_b\text{” or “}P_a > K\text{” or “}K > P_b\text{”}$$

for any two tasks a and b, and a constant $K \in \mathbb{N}$; more formally:

Definition 6.3.1 (Λ-constraints). *Let S be a task set, Λ-constraints is a set of priority constraints on S such that any priority constraint λ in Λ-constraints can be*

- $\lambda_{ab} = \{a, b \in S : P_a > P_b\}$
- $\lambda_{aK} = \{a \in S, K \in \mathbb{N} : P_a > K\}$
- $\lambda_{Kb} = \{b \in S, K \in \mathbb{N} : K > P_b\}$

For instance, consider the task set $\{a, b, c\}$ and priorities $3, 2, 1$ where 3 is the highest priority and 1 the lowest one; the 3! orderings are

$$\langle a\,b\,c\rangle \;\langle a\,c\,b\rangle \;\langle b\,a\,c\rangle \;\langle b\,c\,a\rangle \;\langle c\,a\,b\rangle \;\langle c\,b\,a\rangle$$

Consider the constraint $\lambda_{c,1} = \{P_c > 1\}$, which means that the priority of *c* cannot be the lowest one; that is, all orderings where $P_c = 1$ will not meet the constraint and therefore $\langle a\,b\,c\rangle$ and $\langle b\,a\,c\rangle$ are not viable solutions. Now consider the set of Λ-constraints $\Lambda = \{\lambda_{c,1}, \lambda_{a,c}\} = \{P_c > 1, P_a > P_c\}$; only the ordering $\langle a\,c\,b\rangle$ meets all the constraints.

The following section shows how Λ-constraints is used for implementing precedence relationships between tasks.

6.3.2 Precedence Relationships

Precedence relationships are important hard constraints in many systems where tasks involved in an activity depends on other ones. Simple precedence relationships can be included in our model of importance by conditional relative importance statements of the form "x *is more important than* y *only if* z *precedes* x". For the sake of simplicity, we will denote "*z* precedes *x*" as "$z \rightarrow x$".

In priority scheduling, "*x* is more important than *y*" is expressed by assigning priorities such that $P_x > P_y$. In addition, in order to ensure precedence of *z* over *x*, it is necessary to consider the task release time r such that [16]:

"if $\mathsf{r}_z \leq \mathsf{r}_x$ then $P_z > P_x$" or "if $\mathsf{r}_z < \mathsf{r}_x$ then $P_z \geq P_x$"

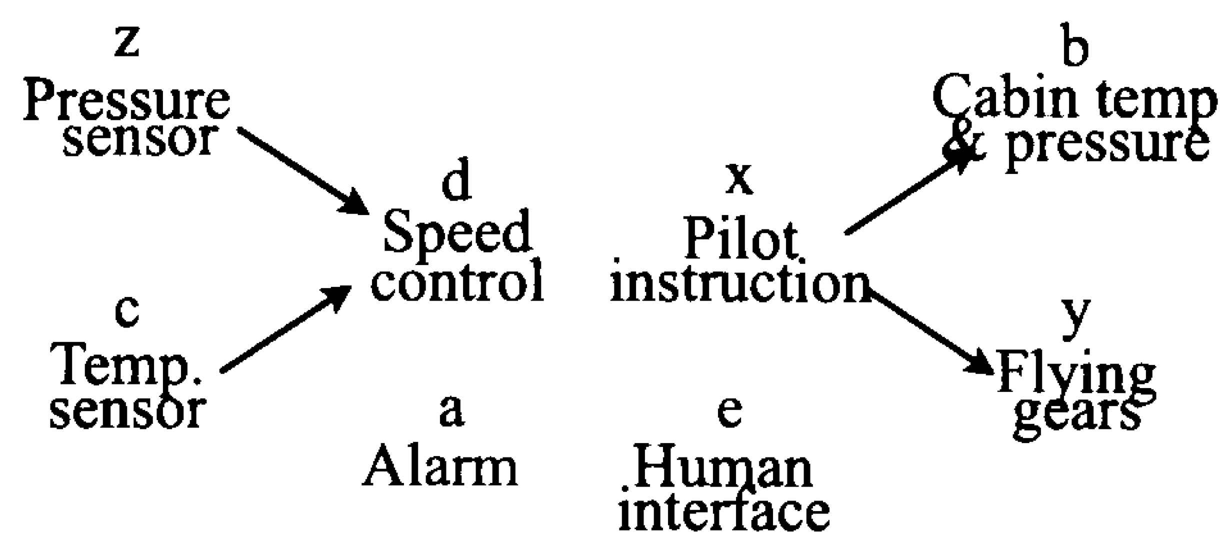


Figure 6.1: Precedence relationships

task	*C*	*T=D*	*R*
a	2	10	2
x	1	16	3
y	2	16	5
b	1	16	6
z	3	32	9
c	2	32	13
d	1	32	14
e	3	56	23

Precedence	Λ-constraints
$z \rightarrow d$	$P_z > P_d$
$c \rightarrow d$	$P_c > P_d$
$x \rightarrow b$	$P_x > P_b$
$x \rightarrow y$	$P_x > P_y$

Table 6.1: Task set S_8 with their precedence constraints. It is ordered by DMPA.

Because in our model tasks can be released simultaneously ($\mathsf{r}_z = \mathsf{r}_x$) we need to assure the strict priority assignment $P_z > P_x$. This is a Λ-constraint λ_{zx} (definition 6.3.1) and then by doing $\bar{I}_x = (I_x, \lambda_{zx})$ and $\bar{I}_y = (I_y, \varnothing)$

$$\text{“}\bar{I}_x > \bar{I}_y\text{”} \Leftrightarrow \text{“}P_x > P_y \text{ only if } P_z > P_x\text{”}.$$

6.3.3 Example

Getting back to the example in chapter 1, the following precedence relationships exist (see Figure 6.1 and Table 6.1): $\{z \rightarrow d, c \rightarrow d, x \rightarrow b, x \rightarrow y\}$, and therefore the Λ-constraints is $\Lambda = \{P_z > P_d, P_c > P_d, P_x > P_b, P_x > P_y\}$.

$S^D = \langle a\, x\, y\, b\, z\, c\, d\, e\rangle$ is the priority ordering given by DMPA which is feasible. Note that the constraints in Λ-constraints are fulfilled but in terms of importance (output jitter in the example) improvements are required.

The task subset with jitter requirements is $\{x, y, z\}$; tasks related with this subset and therefore relatively important are $\{b, c, d\}$; finally $\{a, e\}$ are the least important ones. We

			$\omega[k]$	S^*_{test}	Feasible?
step	ω	k	τ	$\langle\phi\ \tau\ \omega_{\delta-\tau}\rangle$	$\mathcal{F}\left(S^*_{test}\right)$
1	**8** 7 6 5 4 3 2 1	0	8	**8** 2 7 5 6 4 3 1	yes
2	**7** 6 5 4 3 2 1	0	7	8 **7** 2 5 4 3 2 1	yes
3	**6** 5 4 3 2 1	0	6	8 7 **6** 2 5 4 3 1	yes
4	**5** 4 3 2 1	0	5	8 7 6 **5** 2 4 3 1	yes
5	**4** 3 2 1	0	4	8 7 6 5 **4** 2 3 1	**no**
6	4 **3** 2 1	1	3	8 7 6 5 **3** 2 4 1	yes
7	**4** 2 1	0	4	8 7 6 5 3 **4** 2 1	**no**
8	4 **2** 1	1	2	8 7 6 5 3 **2** 4 1	yes
9	**4** 1	0	4	8 7 6 5 3 2 **4** 1	yes
10	**1**	0	1	8 7 6 5 3 2 4 **1**	yes

Table 6.2: DI algorithm with input $S_8^{\bar{I}}$ labelled by importance $\langle 8\,7\,6\,5\,4\,3\,2\,1\rangle$. The solution $\langle 8\,7\,6\,5\,3\,2\,4\,1\rangle$ is found in 10 steps from a universe of 8!

assign the importance $I \in \bar{I}$ to the tasks of Table 6.1 as follows: $\{I_x > I_y > I_z > I_b > I_c > I_d > I_a > I_e\}$. Therefore, the highest importance ordering is $S_8^{\bar{I}} = \langle x\,y\,z\,b\,c\,d\,a\,e\rangle$, which is infeasible since the response time of task a is 12.

Lets apply the DI algorithm to this example. In order to show the sequence of steps followed by DI, let us label by importance the ordering $S_8^{\bar{I}}$ as follows:

$$S_8^{\bar{I}} = \langle x\,y\,z\,b\,c\,d\,a\,e\rangle = \langle 8\,7\,6\,5\,4\,3\,2\,1\rangle$$

Applying DI to $S_8^{\bar{I}}$ produce the sequence of steps depicted in Table 6.2 resulting in the ordering S1 $= \langle x\,y\,z\,b\,d\,a\,c\,e\rangle$. Observe that S1 is lexicographically closer to $S_8^{\bar{I}}$ and therefore it is better in terms of importance than DMPA but it does not fulfill the precedence constraint $c \rightarrow d$ (i.e. $P_c \not> P_d$). However, given the same input $S_8^{\bar{I}}$, a solution is found by the DI+ algorithm introduced in the following section.

6.4 The DI+ Algorithm

The Deadlines and Conditional Importance scheduling problem is solved by the DI+ algorithm, which is an extension of the DI algorithm. DI+ takes as input the ordering $S^{\bar{I}}$ and a set of Λ-constraints, and finds the solution S^* performing a specialized branch and bound

search in $\frac{N^2+N}{2}$ steps in the worst case.

Similar to the DI algorithm, DI+ also tests vertices $\langle \phi\, \delta(\omega) \rangle$ in the search space following the lexicographic pattern specified by its input $S^{\bar{I}}$. However, in this case testing a vertex $\langle \phi\, \delta(\omega) \rangle$ for feasibility also implies testing their Λ-constraints. Therefore, the main difference between DI and DI+ is the feasibility test. While in DI the feasibility test is with respect to the deadlines, in DI+ it is with respect to both deadlines and Λ-constraints. Consequently the proof of optimality of DI also works for DI+; if the test is sufficient and necessary the algorithm is optimal. In addition, the complexity of DI+ increases due to the test of the Λ-constraints.

In both the DI and DI+ algorithms, when a vertex $\langle \phi\, \delta(\omega) \rangle$ is feasible, ϕ is fixed and tasks in ϕ cannot be withdrawn in the future (i.e. there is not backtracking). Consequently, in DI+ the Λ-constraints of all tasks in ϕ have to be verified. Thus, ϕ is accepted in any iteration of DI+ only when $\langle \phi\, \delta(\omega) \rangle$ is feasible and the Λ-constraints of all of all tasks in the prefix are fulfilled. Therefore, an algorithm for testing the Λ-constraints has to be provided to DI+.

Definition 6.4.1. *Given an ordering α and a set of Λ-constraints, let $\mathcal{T}$ be a function which tests the constraints such that*

$$\mathcal{T}(\alpha, \Lambda) = \begin{cases} \textit{true} & \textit{all tasks in } \alpha \textit{ meet } \Lambda\textit{-constraints} \\ \textit{false} & \textit{otherwise} \end{cases}$$

6.4.1 Modifications to DI

The DI+ algorithm (algorithm 4 in page 110) examines the vertices of subtrees in lexicographic order from the closest one to $S^{\bar{I}}$ to the remotest one, and from the top to the bottom in the same form that the DI algorithm but with some minor modifications: the stop condition, the feasibility test, and a final check to verify whether a solution was found at the end. More specifically:

1. DI+ accepts a set of Λ-constraints which can be empty; in such a case DI+ reduces to DI.

2. In DI+ there is not guarantee that the suffix ω is always decreasing (in DI does). This takes place, for example, when at each iteration of DI+ the feasibility test (steps 5-6) fails; in this case, tasks from ω will never be moved to the prefix ϕ and then the length of ω will remain constant while the index k is increased. Consequently, the algorithm has to verify that the index k does not overrun ω (step 2).

3. In DI a feasible solution is always returned (e.g. DMPA in the worst case) but in DI+ such feasible solution could not exist (e.g. DMPA is feasible but constraints are not fulfilled). A feasible solution has been found when ω is empty; otherwise it does not exist because the Λ-constraints are not been satisfied.

In addition, steps 5 and 6 of the DI algorithm are modified. The following lines indicate that an ordering S^*_{test} is built by joining the prefix $\langle\phi\ \tau\rangle$ with the suffix $\langle\omega_{\delta-\tau}\rangle$, which is then tested for feasibility.

5: build "$S^*_{test} \Leftarrow \langle\phi\ \tau\ \omega_{\delta-\tau}\rangle$"

6: **if** $\mathcal{F}(S^*_{test})$ **then**

These two steps are re-written as follows:

5: build "$S^*_{test} \Leftarrow \mathcal{F}_\Lambda(\langle\phi\ \tau\rangle, \langle\omega_{\delta-\tau}\rangle, \Lambda)$"

6: **if** $(S^*_{test} \neq null)$ **then**

where $\mathcal{F}_\Lambda$ is a function that receives a prefix $\langle\phi\ \tau\rangle$, a suffix $\langle\omega_{\delta-\tau}\rangle$ and the Λ-constraints. $\mathcal{F}_\Lambda$ builds the ordering $\langle\phi\ \tau\ \omega_{\delta-\tau}\rangle$ and test its feasibility. If it is feasible, $\mathcal{F}_\Lambda$ tests all the tasks in the prefix $\langle\phi\ \tau\rangle$ with respect to their constraints in Λ; constraints no related to tasks in the prefix are ignored. If the prefix passes the test then the ordering $\langle\phi\ \tau\ \omega_{\delta-\tau}\rangle$ is returned; otherwise $\mathcal{F}_\Lambda$ returns a *null* ordering. The algorithm 3 implements $\mathcal{F}_\Lambda$.

Let S be a task set with orderings S^D and $S^{\bar{I}}$; S^D must be feasible. The algorithm builds keys to index each subtree in lexicographic order. A key is built by appending to ϕ a task τ (step 3) from ω. The lexicographical order is achieved following the sequence in ω which is $S^{\bar{I}}$. The key indexes the subtree $\langle\phi\,\tau\ *\rangle$ and hence $\mathcal{F}_\Lambda$ builds and test $\langle\phi\ \tau, \omega_{\delta-\tau}\rangle$

Algorithm 3 $\mathcal{F}_\Lambda(\phi, \omega, \Lambda)$

1: built $\alpha \Leftarrow \langle\phi, \omega\rangle$
2: **if** $\mathcal{F}(\alpha)$ and $\mathcal{T}(\phi, \Lambda)$ **then**
3: return α
4: **else**
5: return *null*
6: **end if**

Algorithm 4 DI+ (Deadline and Importance Extended)

Require: S^D feasible, $S^{\bar{I}}$, Λ =Λ-constraints
1: Set $\phi \Leftarrow \emptyset$, $\omega \Leftarrow S^{\bar{I}}$, $\omega_\delta \Leftarrow S^D$,$k = 0$
2: **while** SizeOf(ω) > k **do**
3: let $\tau \Leftarrow \omega[k]$
4: let $\omega_{\delta-\tau} \Leftarrow delete(\omega_\delta, \tau)$
5: build $S^*_{test} \Leftarrow \mathcal{F}_\Lambda(\langle\phi\ \tau\rangle, \langle\omega_{\delta-\tau}\rangle, \Lambda)$
6: **if** ($S^*_{test} \neq null$) **then**
7: $S^* \Leftarrow S^*_{test}$
8: $\phi \Leftarrow \langle\phi\ \tau\rangle$
9: $\omega \Leftarrow delete(\omega, \tau)$
10: $\omega_\delta \Leftarrow \omega_{\delta-\tau}$
11: $k \Leftarrow 0$
12: **else**
13: $k \Leftarrow k + 1$
14: **end if**
15: **end while**
Ensure: ω is empty

(steps 4-5). If $\mathcal{F}_\Lambda$ returns $S^*_{test} \neq null$, then S^*_{test} is saved as a partial solution S^*, τ is deleted from both ω and ω_δ, ϕ is updated and the index k is reset (*steps 7-11*); otherwise, k is advanced to the next τ. DI+ stops when there are no more subtrees to visit. A feasible solution has been found when ω is empty; otherwise the Λ-constraints were not fulfilled.

Complexity

Two possible worst-case scenarios exist: (1) when the only solution is S^D and its order is contrary to $S^{\bar{I}}$ and (2) when the Λ-constraints are not met. In both cases the loop is executed N times and ω is reduced to $N - 1$ elements; afterward the loop is executed $N - 1$ times and ω is reduced to $N - 2$ elements and so on. Therefore the loop executes $N + (N - 1) + (N - 2) + \ldots + 1 = \frac{N^2+N}{2}$ times. At each iteration, $\mathcal{F}_\Lambda$ is executed. Its

Algorithm 5 $V(\phi, \Lambda)$

Require: tasks $\tau \in \phi$ and constraints $\Lambda = \{\lambda_1, \lambda_2, \ldots\}$

```
1: for all τj ∈ φ do
2:    for all λi ∈ Λ do
3:       if NOT(τj meets λi) then
4:          return FALSE
5:       end if
6:    end for
7: end for
8: return TRUE
```

complexity is the complexity $E1$ of the feasibility test $\mathcal{F}$ plus the complexity $E2$ of the function $\mathcal{T}$. Therefore the complexity of the algorithm DI+ is $O((E1+E2)\times\frac{N^2+N}{2})$. Thus, the algorithm finds the solution in a polynomial number of steps but the total complexity is at least pseudo-polynomial due to the FPS test.

6.4.2 Example

Lets continue with the example of chapter 1 (see Table 6.1), where some tasks have jitter requirements. First at all, we need to provide an algorithm for checking the Λ-constraints.

Algorithm 5 is a function V that receives as input the tasks in prefix ϕ and a set of Λ-constraints; i.e. it implements $\mathcal{T}(\alpha, \Lambda)$ (see definition 6.4.1). $V(\phi, \Lambda)$ returns true when all tasks in ϕ meet the constraints; otherwise it returns false. The complexity of this algorithm varies depending on the size of ϕ and Λ tested but in the worst-case is $O(N \times M)$ where N and M are the number of tasks and the number of constraints respectively. This algorithm assumes that the ordering given by DMPA meets the Λ-constraints; i.e. if there exists a constraint $P_a > P_b$ then $D_a \leq D_b$. Although simplistic it is sufficient for this example since the tasks in Table 6.1 meet this assumption. This simplifies the complexity of algorithm 5.

We define in section 6.3.3 the highest importance ordering as $S_8^{\bar{I}} = \langle x\,y\,z\,b\,c\,d\,a\,e\rangle$ and show that the solution found by DI was S1. Applying DI+ to $S_8^{\bar{I}}$ (see Table 6.3) give us S2:

- S1: DI($S_8^{\bar{I}}$) produces $\langle x\,y\,z\,b\,d\,a\,c\,e\rangle = \langle 8\,7\,6\,5\,3\,2\,4\,1\rangle$

			$\omega[k]$	S^*_{test}	$\mathcal{F}_\Lambda$	
step	ω	k	τ	$\langle\phi\ \tau\ \omega_{\delta-\tau}\rangle$	$\mathcal{F}\ (S^*_{test})$	$V(\phi,\Lambda)$
1	**8 7 6 5 4 3 2 1**	0	8	**8** 2 7 5 6 4 3 1	yes	yes:$P_8 > P_7, P_8 > P_5$
2	**7 6 5 4 3 2 1**	0	7	8 **7** 2 5 6 4 3 1	yes	yes:$P_8 > P_7$
3	**6 5 4 3 2 1**	0	6	8 7 **6** 2 5 4 3 1	yes	yes:$P_6 > P_3$
4	**5 4 3 2 1**	0	5	8 7 6 **5** 2 4 3 1	yes	yes:$P_8 > P_5$
5	4 3 2 1	0	4	8 7 6 5 **4** 2 3 1	**no**	yes:$P_4 > P_3$
6	4 **3** 2 1	1	3	8 7 6 5 **3** 2 4 1	yes	**no**:$P_4 \not> P_3, P_6 > P_3$
7	4 3 **2** 1	2	2	8 7 6 5 **2** 4 3 1	yes	yes:$P_2 \notin \Lambda$
8	**4** 3 1	0	4	8 7 6 5 2 **4** 3 1	yes	yes:$P_4 > P_3$
9	**3** 1	0	3	8 7 6 5 2 4 **3** 1	yes	yes:$P_6 > P_3, P_4 > P_3$
10	**1**	0	1	8 7 6 5 2 4 3 **1**	yes	yes:$P_2 \notin \Lambda$

Table 6.3: DI+ algorithm with input $S_8^{\bar{I}}$ labelled by importance $\langle 8\,7\,6\,5\,4\,3\,2\,1\rangle$ with constraints $\Lambda = \{P_6 > P_3, P_4 > P_3, P_8 > P_5, P_8 > P_7\}$

- S2: DI+(S_8^I, Λ) produces $\langle x\,y\,z\,b\,a\,c\,d\,e\rangle = \langle 8\,7\,6\,5\,2\,4\,3\,1\rangle$

Observe that S1 is lexicographically closer to S_8^I than S2 but S1 does not meet the Λ-constraints. Note that both solutions are identical until the element 4. This is because the first four tasks of the input S_8^I were found feasible until *step 5* (see Tables 6.2 and 6.3). In the next iteration, task labelled as 3 ($I_d = 3$) is selected and a prefix $\langle 8\,7\,6\,5\ 3\rangle$ is built. The ordering tested for feasibility is $\langle(8\,7\,6\,5\ 3)\,2\,4\,1\rangle$, which is feasible and accepted in S1; at this point S2 needs to verify the Λ-constraints and hence it calls to $V(\phi, \Lambda)$, and then notes that $P_4 \not> P_3$. Thus, it is rejected and both solutions will follow different paths.

In summary, the DI+ algorithm provides support for some hard priority constraints that we call Λ-constraints. It works for tasks sets where the deadlines are less than or equal to the periods. It can be easily integrated into the software development cycle to play with different importance assignments and different priority constraints until finding fixed-priority orderings that meet particular QoS necessities. Note that DI+ can be easily adapted to store the partial solutions shown in Tables 6.2 and 6.3 in order to provide additional information to decision makers.

Naturally, there exist systems where the assumptions of strict hard deadlines or deadlines less than or equal to the periods do not hold. In the following section we sketch how the DI+ algorithm can be extended to cope with problems where tasks can have arbitrary deadlines or weakly-hard constraints.

6.5 Extending the DI+ Algorithm

The DI+ algorithm depends mainly on both the feasibility test and the optimality of the DMPA algorithm. Nevertheless, in the presence of tasks with arbitrary deadlines or tasks with weakly-hard deadlines, such FPS algorithms are no longer valid. However, by substituting the feasibility test with the extended version of the response time analysis test or with the feasibility test for weakly-hard systems respectively, DI+ can cope with such kind of problems. In addition, since DMPA is not optimal the swapping algorithm (on substitution of DMPA) has to be included in the solution for the problem. In this sense, we define the following functions:

Definition 6.5.1 (Feasibility test with arbitrary deadlines). *For $\alpha \in \hat{S}$, let $\mathfrak{F}_E(\alpha)$ be a function that implements the feasibility test using the extended response time equation (section 2.3.5), such that $\mathfrak{F}_E(\alpha)$ returns* true *when α is feasible and* false *otherwise.*

Definition 6.5.2 (Feasibility test with weakly-hard deadlines). *For $\alpha \in \hat{S}$, let $\mathfrak{F}_W(\alpha)$ be a function that implements the feasibility test using the algorithms for weakly-hard systems (section 2.3.7), such that $\mathfrak{F}_W(\alpha)$ returns* true *when α is feasible and* false *otherwise.*

Definition 6.5.3 (Swapping algorithm with $\mathfrak{F}_E$ or $\mathfrak{F}_W$). *For any priority ordering $\alpha = \langle\phi\,\omega\rangle$ and a feasibility test $\mathfrak{F}_E$ (or $\mathfrak{F}_W$), let $\delta_\phi(\omega)$ be a function that implements the swapping algorithm (with $\mathfrak{F}_E$ or $\mathfrak{F}_W$) maintaining ϕ fixed and only swapping tasks in ω such that*

$$\delta_\phi(\omega) = \begin{cases} \omega^* & \textit{if } \omega^* \textit{ was found such that } \langle\phi\ \omega^*\rangle \textit{ is feasible} \\ null & \textit{otherwise} \end{cases}$$

With respect to the feasibility test, by using $\mathfrak{F}_E$ or $\mathfrak{F}_W$ for testing priority orderings we solve part of the problem of including tasks with arbitrary deadlines or weakly-hard constraints in DI+.

With respect to the swapping algorithm, observe that in each iteration the DI+ algorithm tests vertices $\langle\phi\,\delta(\omega)\rangle$ relaying on the fact that if this vertex is infeasible, then there are not any feasible orderings in the subtree $\langle\phi\,*\rangle$ (i.e. the Prefix theorem). The suffix of this vertex is ordered by δ according to the DMPA rule since it is an optimal ordering.

Algorithm 6 $\mathcal{F}^*_\Lambda(\phi, \omega, \Lambda)$

1: $\omega^* \Leftarrow \delta_\phi(\omega)$
2: **if** $\omega^* = null$ **then**
3: return **false**
4: **end if**
5: built $\alpha \Leftarrow \langle \phi, \omega^* \rangle$
6: **if** $\mathcal{T}(\phi, \Lambda)$ **then**
7: return α
8: **else**
9: return *null*
10: **end if**

However, in this case it does not hold but we can substitute δ with δ_ϕ that implements the swapping algorithm which is optimal. Observe that the result is equivalent, if $\langle \phi\, \delta_\phi(\omega) \rangle$ is infeasible, then non-feasible orderings exist in the subtree $\langle \phi\, * \rangle$.

The function $\mathcal{F}^*_\Lambda$ in algorithm 6 implements these changes. By substituting $\mathcal{F}_\Lambda$ by $\mathcal{F}^*_\Lambda$ in the DI+ algorithm we include arbitrary deadlines or weakly-hard constraints in the scheduling problem with deadlines and importance.

6.5.1 An Example

Consider a tasks set $S_W = \{a, b, c, d, e\}$ with simple relative importances $I_a > I_b > I_c > I_d > I_e$ such that $S^I = \langle a\, b\, c\, d\, e \rangle$ is an infeasible ordering under the response time feasibility test. Moreover, the task set ordered by DMPA (S^D) is also infeasible since task a misses its deadline ($R_a = 511$) (see Table 6.4).

task	C	*D=T*	*R*	*Any (n,m)*	*I*
e	80	80	17	(7,20)	1
d	110	110	41	(1,1)	2
c	270	270	142	(4,7)	3
b	280	280	263	(1,1)	4
a	420	420	511	(2,4)	5

Table 6.4: Task set S_W with Weakly-Hard Constraints

Let assume us that some tasks tolerate missing several deadlines if it occurs in a determined pattern such as that defined by the weakly-hard constraints model (see definition 2.3.11). In particular, tasks a, c and e have weakly-hard constraints of the type (n, m)

(a task meets any *n* in *m* deadlines), and tasks b and d have strong weakly-hard constraints (i.e. they have hard deadlines).

Applying the weakly-hard feasibility test to S^D (i.e $\mathcal{F}_W(S^D)$), it can be shown that the task a meets its weakly hard constraint (any 2 in 4 deadlines) and the rest of the tasks meet all the deadlines. Thus, the set is weakly-hard feasible but it has the lowest index of importance because S^D is exactly the contrary to S^I (see Figure 6.2).

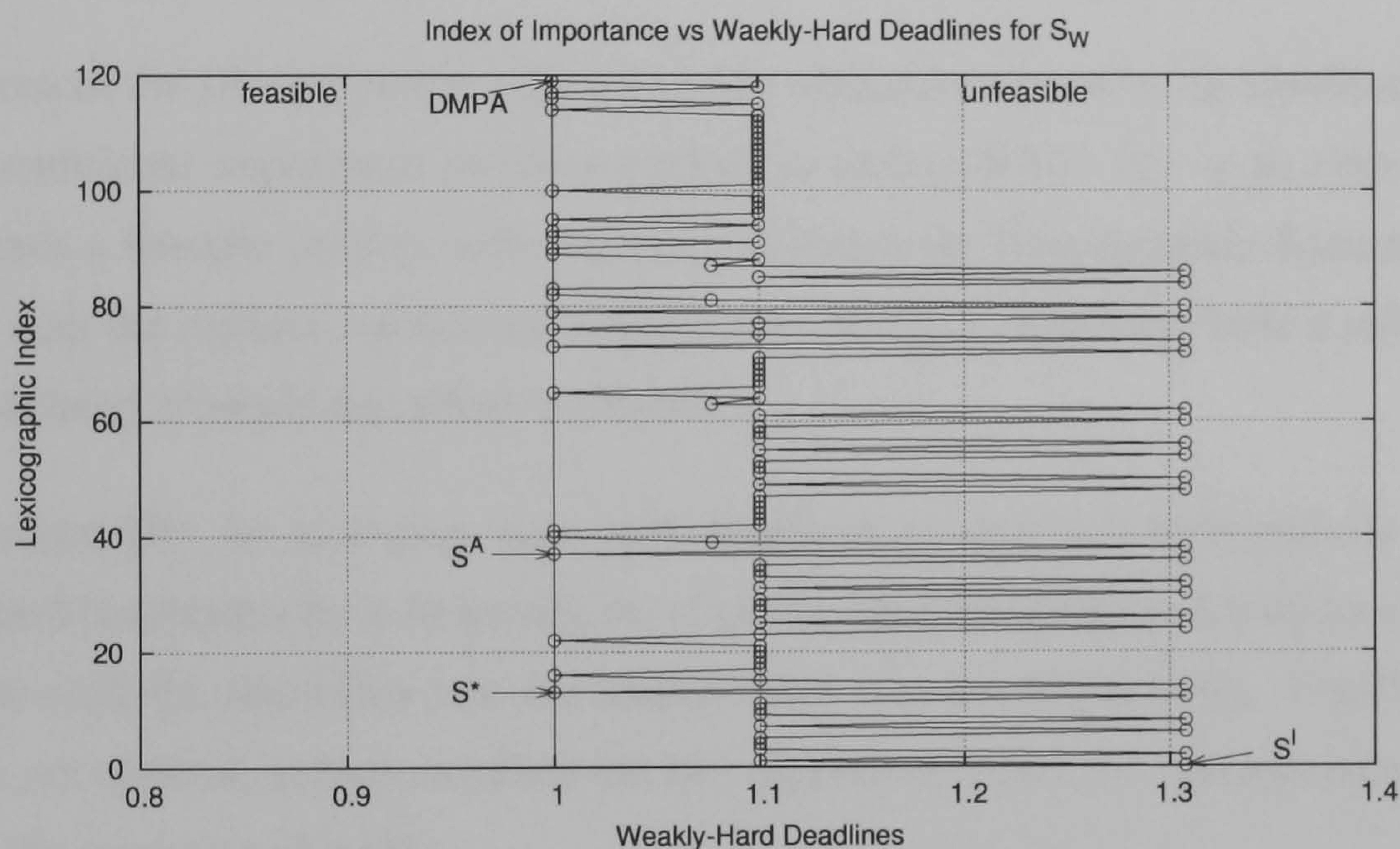


Figure 6.2: 5! Priority orderings of set S_W ordered lexicographically. There are 32 weakly-hard feasible priority orderings

Executing the swapping algorithm with input S^I produces an ordering $S^A = \langle b\,d\,a\,e\,c\rangle$ with index of importance 37. In this solution, tasks $\{b, d, a\}$ do not miss any deadline and tasks $\{e, c\}$ meet their weakly-hard constraints. Thus, this solution is better than the DMPA solution since the two tasks with higher importance do not miss any deadline.

The optimal solution $S^* = \langle a\,d\,b\,e\,c\rangle$ found by DI+ has index 13. This is even better since the task with the highest importance has the highest priority. Also tasks $\{a, d, b\}$ do not miss any deadline and tasks $\{e, c\}$ meet their weakly-hard constraints.

6.6 Summary

Conditional relative importance is a generalization of simple relative importance, which adds hardness to the importance concept and captures some dependence relationships among tasks. In our model, dependence relationships are limited to a set of simple restrictions for assigning higher priority to a higher importance task only if a set of constraints called Λ-constraints are fulfilled.

We present the DI+ algorithm which finds the optimal solution to the deadlines ($D \leq T$) and conditional importance problem defined in section 6.2.3. DI+ is an extension of DI that finds a feasible priority ordering that minimises the lexicographic distance to the ordering with the highest conditional importance. The DI+ algorithm finds a solution to the motivational example described in chapter 1.

We extend DI+ for including tasks with deadlines greater than their periods or with weakly-hard constraints by substituting the response time feasibility test with its extended version or with the feasibility test for weakly-hard systems respectively. Finally, since DMPA is not optimal, at each iteration the DI+ algorithm orders the corresponding suffix by using the swapping algorithm.

Chapter 7

Evaluation

7.1 Introduction

In this chapter, we propose solutions to the multicriteria scheduling problems defined in chapter 4. A solution is an assignment of fixed-priorities found by combining the DI algorithm with a heuristic for assigning importance.

The chapter is divided into two main sections:

- In the first one, we want to discover heuristics for assigning importance. By testing the simple rules for assigning fixed-priorities depicted in Table 4.4 (page 81), we determine which are the best ones with respect to a QoS metric without considering the deadlines. Afterwards, we propose them as heuristics for assigning importance.
- In the second one, we are interested in judging the DI algorithm jointly with the heuristics discovered. In general, such heuristics produce infeasible solutions. However, feasible solutions are found when the heuristic is used as input for the DI algorithm. If any combination shows good performance with respect to a QoS metric, then we propose such combination as a technique for solving the scheduling problem with deadlines and QoS metric.

Since in [17] it is compared RMPA against EDF under several QoS metrics, and those results may be refuted by our approach, in our experiments we will include these two policies. In effect, experiments carried out by Buttazzo [17] have shown that EDF outperforms

RMPA for a number of metrics. Our experiments confirm the results of [17] in the sense that EDF exhibits better performance than RMPA. However, we find that our approach outperforms EDF in most of the scenarios tested.

As the swapping algorithm can be considered as a good heuristic for the scheduling problem where the DI algorithm is optimal, in section 7.5 we compare both algorithms. In addition, a simple evaluation of the DI+ algorithm for task sets with priority constraints is presented in section 7.6.

7.2 Experimental Setup

The experiments carried out in this thesis require the simulation of schedules of task sets during a fixed window of time as follows: firstly, we generate a number of synthetic task sets; afterwards, each task set is simulated under a scheduling policy, and several performance metrics are recorded to be compared. Three attributes that affect the accuracy of the results of the simulation can be identified:

- *Task set*: how a task set is created.
- *Window of time*: for a given task set, how long to run each simulation.
- *Number of task sets*: how many task sets must be simulated to get representative results.

The criteria for selecting these attributes are described in the next sections. In summary, the parameters defined consist of 1000 synthetic sets of N tasks randomly generated with periods uniformly distributed in [100,1000] and computation times adjusted to fix any utilisation U. Deadlines are equal to the periods. We vary $U = 0.5, 0.55, 0.60, 0.65, 0.70,$ $0.75, 0.80,$ $0.85, 0.90, 0.95$ and vary $N = 5, \ldots 14$. All task sets generated are feasible under RMPA. Simulations were performed in parallel on two desktop PC computers. The discrete event simulator used in our experiments was specifically built for this research using algorithms described in [84].

7.2.1 Task Sets

An important factor that affects the result of simulating scheduling algorithms is the method for generating random task sets. In [85], it is noted that the problem is the probability density function of the random variables used to generate the task set parameters. It is shown how the typical random generation routine bias the simulation of some specific scheduling algorithm. In this sense, they propose the UUniFast algorithm [85] for generating task sets with uniform distribution. The UUniFast algorithm receives an utilisation U and the number of tasks N for producing utilisations U_j uniformly distributed in $(0, U]$ subject to $U = \sum_{j=1}^{N} U_j$. We use this algorithm for generating our task sets.

A set of N tasks for a utilisation U is generated as follows: The UUniFast algorithm is executed with parameters N and U; it produces $U_j,\ j = 1, \ldots, N$. Posteriorly, we chose periods T_j uniformly distributed in [100,1000] and compute worst-case computation times as $C_j = U_j T_j$. Doing $D_j = T_j$ and assigning priorities according to RMPA, we perform a schedulability test using the response time analysis. If the task set is feasible the task set is stored; otherwise it is discarded. Note that because all task sets generated are feasible under RMPA, they are also feasible under EDF.

7.2.2 Window of Time

In order to determine the window of time to be used in the simulations, we perform some experiments simulating 10 task sets for different time windows using RMPA and EDF policies. We take measures of different metrics to observe the limits beyond which the performance does not improve; i.e. major increase in time ticks simulated results in none or minor variation of the corresponding metric. Results are plotted in Figures 7.1 and 7.2. Each point corresponds to 10 tasks sets of 10 tasks each task set with $U = 0.75$.

Figure 7.1(page 121) shows the variation on the number of preemptions, absolute and relative output jitter with respect to the window of time simulated. With respect to the total number of preemptions, the variation is linear and therefore almost any window is representative. With respect to output jitter, the data is steady after 100,000 ticks.

Figure 7.2 (page 122) shows the variation on maximum latency, relative maximum

latency, and relative average response time with respect to the window of time simulated. With respect to maximum latency, the data is steady after 100,000 ticks. With respect to relative maximum latency, the data is not completely steady but after 70,000 ticks the variation is minimal. With respect to relative average response time, the data is almost steady after 100,000 ticks.

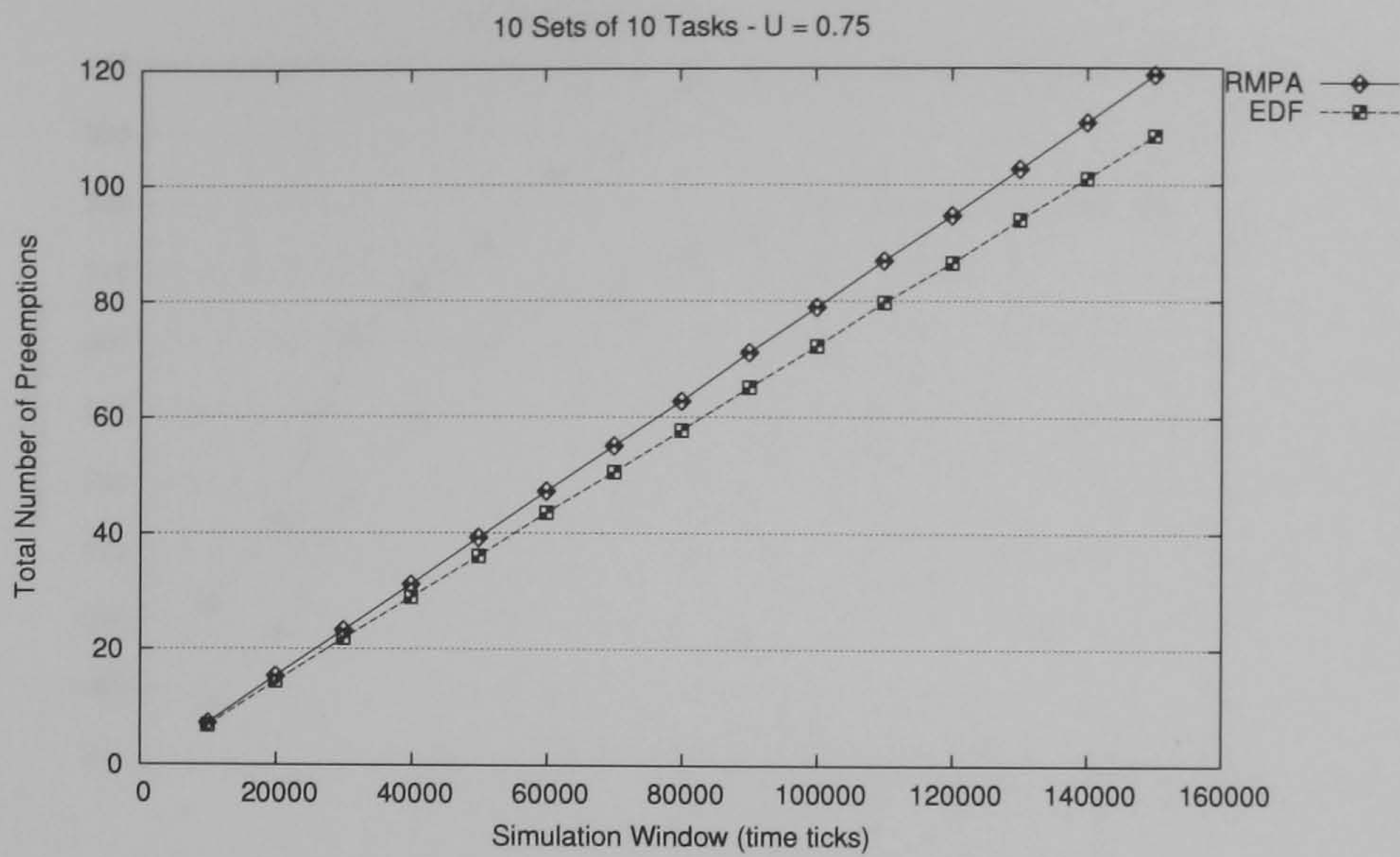
These results suggest that for our task sets, it is sufficient a window of 100,000 ticks for performing the simulations; beyond this size the performance does not improve substantially.

7.2.3 Number of Task Sets

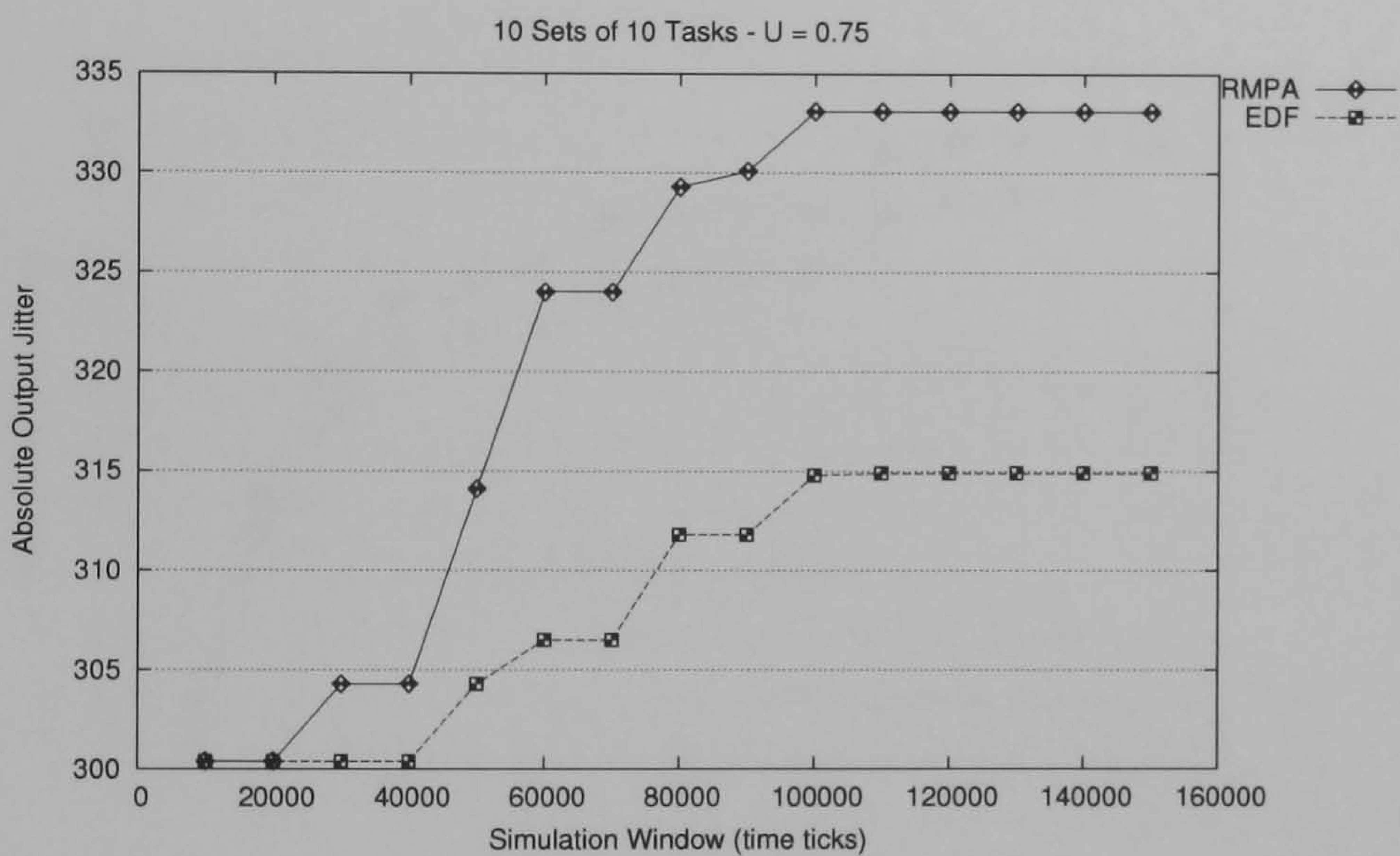
In order to determine the number of task sets to be used in the simulations we perform some experiments simulating groups of 10, 100, 800 and 1000 task sets during 100,000 ticks under RMPA and at different utilisation. We take measures of different metrics to observe the limits beyond which the performance does not improve; i.e. major increase in the number of task sets simulated results in none or minor variation of the corresponding metric. Results are plotted in Figure 7.3 (page 123).

We perform the same experiment for the six metrics as above but the results are similar and therefore we only show the ones where higher variation is observed. In fact, except for the relative maximum latency metric, the rest of the metrics show lower variability after simulating 100 task sets. Figures (a) and (b) show that it could be sufficient to perform simulations with 100 task sets. In these two plots, we omit data for 800 task sets because its line is overlapped with the line of 1000 task sets. Figure (c) shows that for the relative maximum latency metric, the variability is high but it reduces when more than 800 task sets are included.

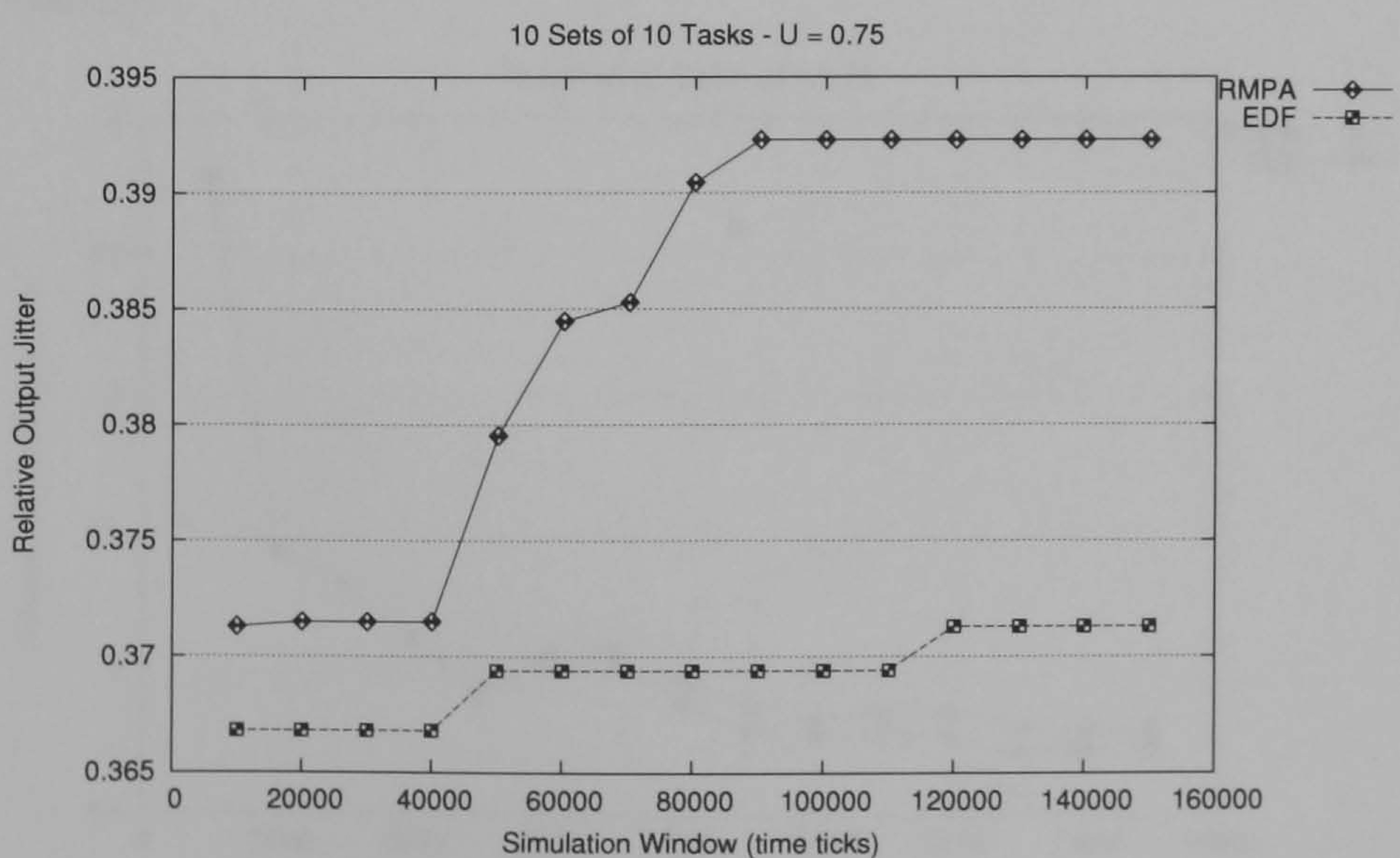
These results suggest that for our task sets, it should be sufficient to simulate (at least) 100 tasks sets excepting when the relative maximum latency is the metric; in this last case it is necessary to simulate between 800 and 1000 task sets. We select 1000 task sets to get representative results.



(a) The variation is linear. Thus, almost any window is representative

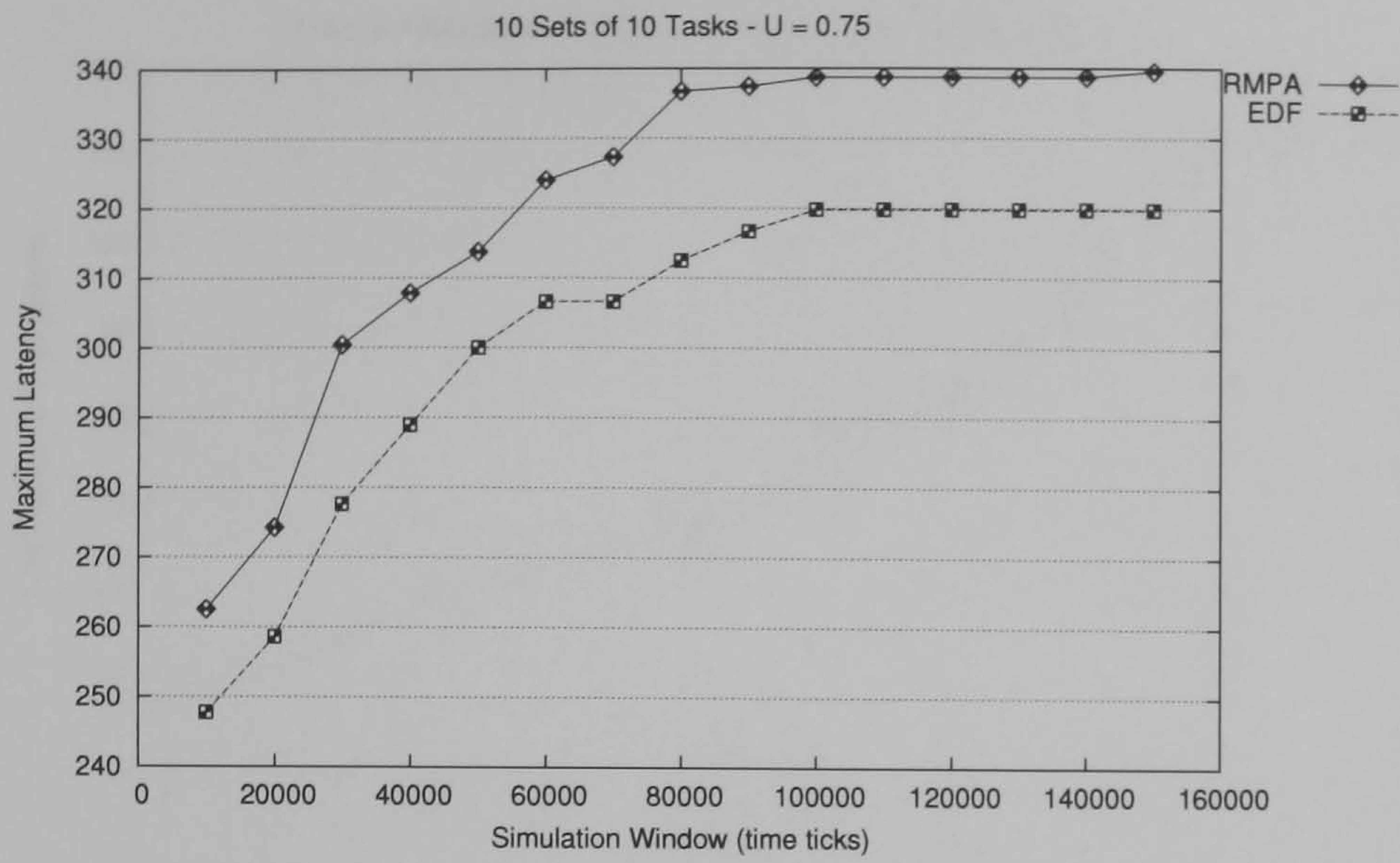


(b) The data is steady after 100,000 ticks

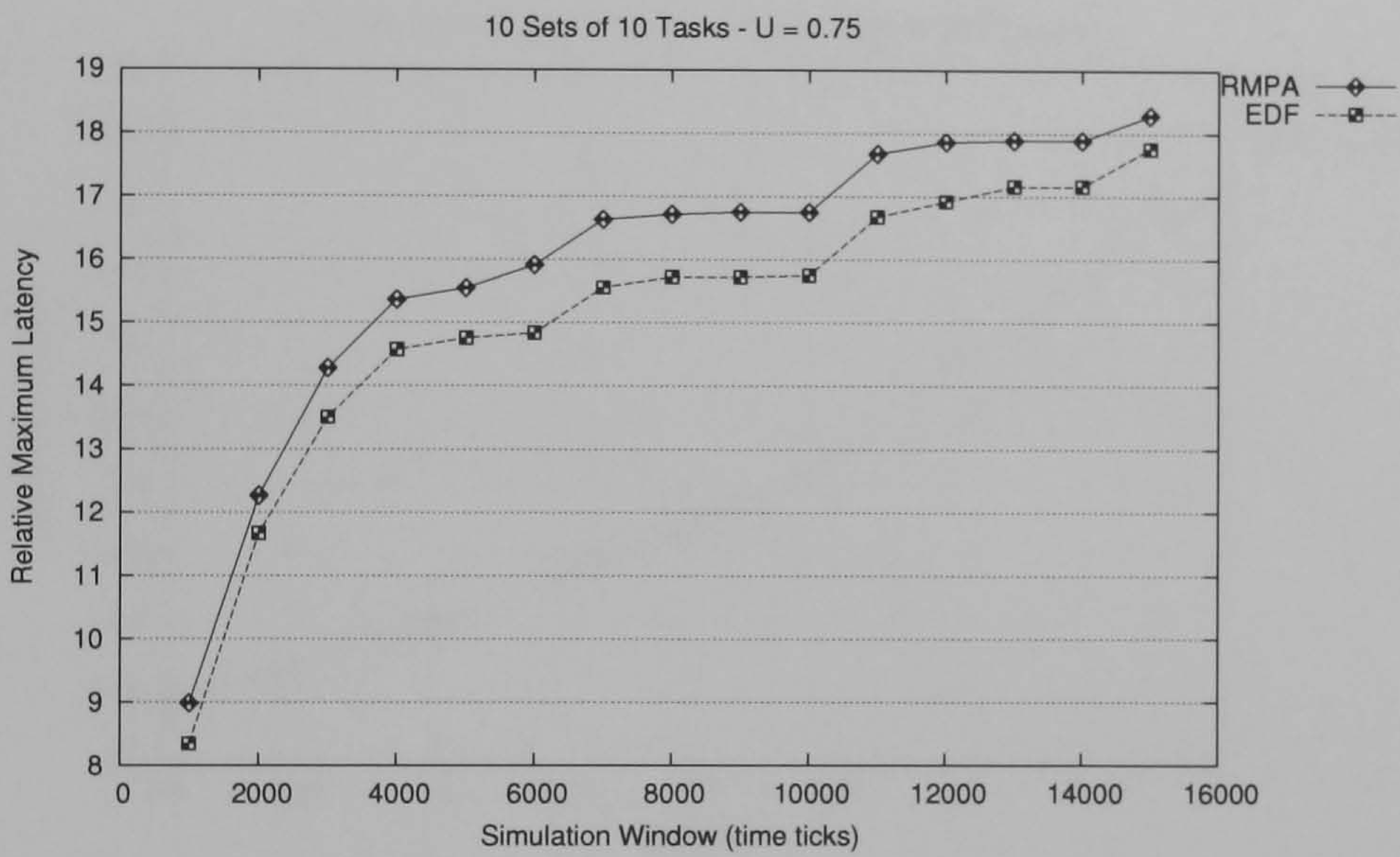


(c) The data is almost steady after 90,000 ticks

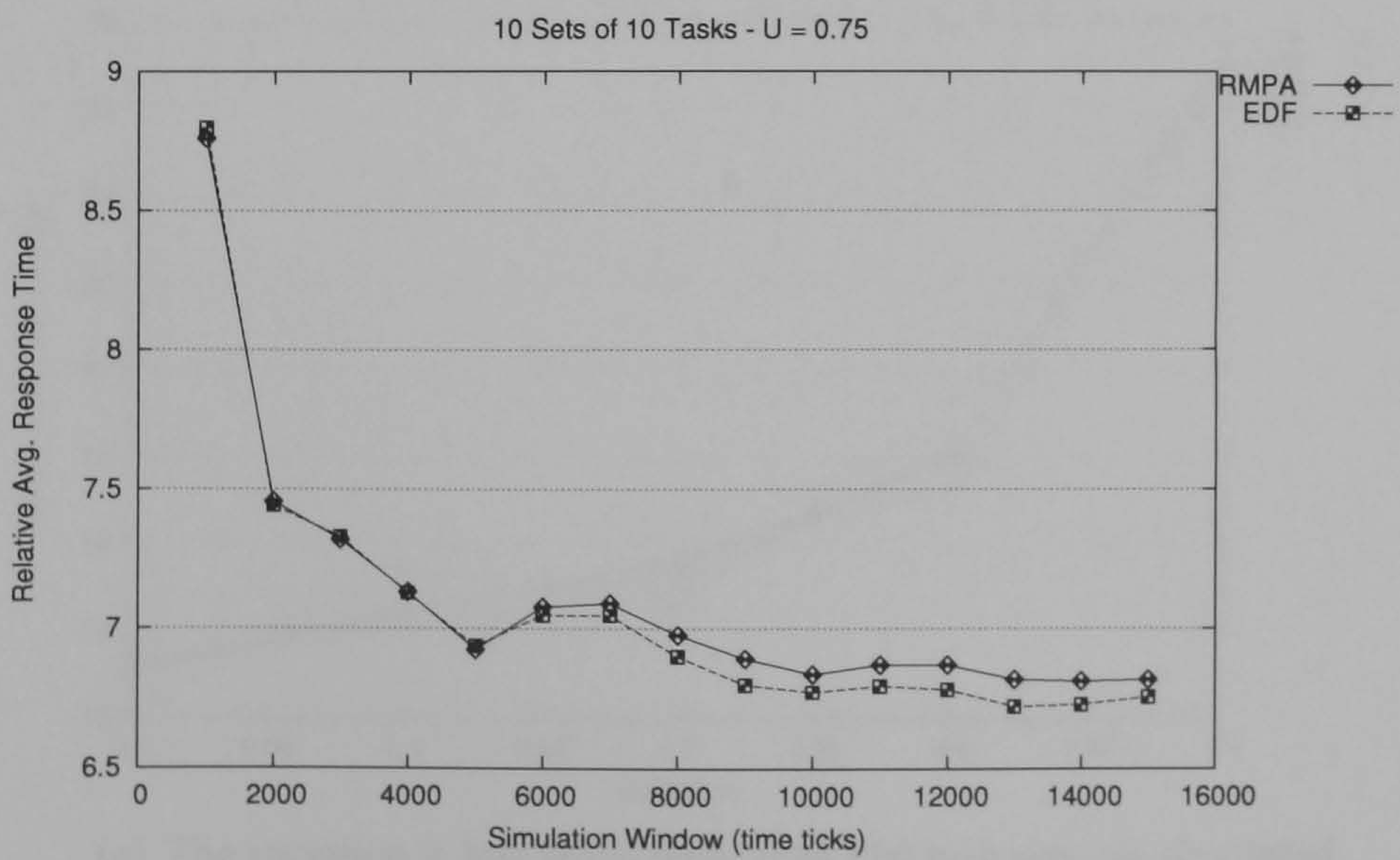
Figure 7.1: Variation on number of preemptions, absolute and relative output jitter with respect to the time window simulated



(a) The data is steady after 100,000 ticks

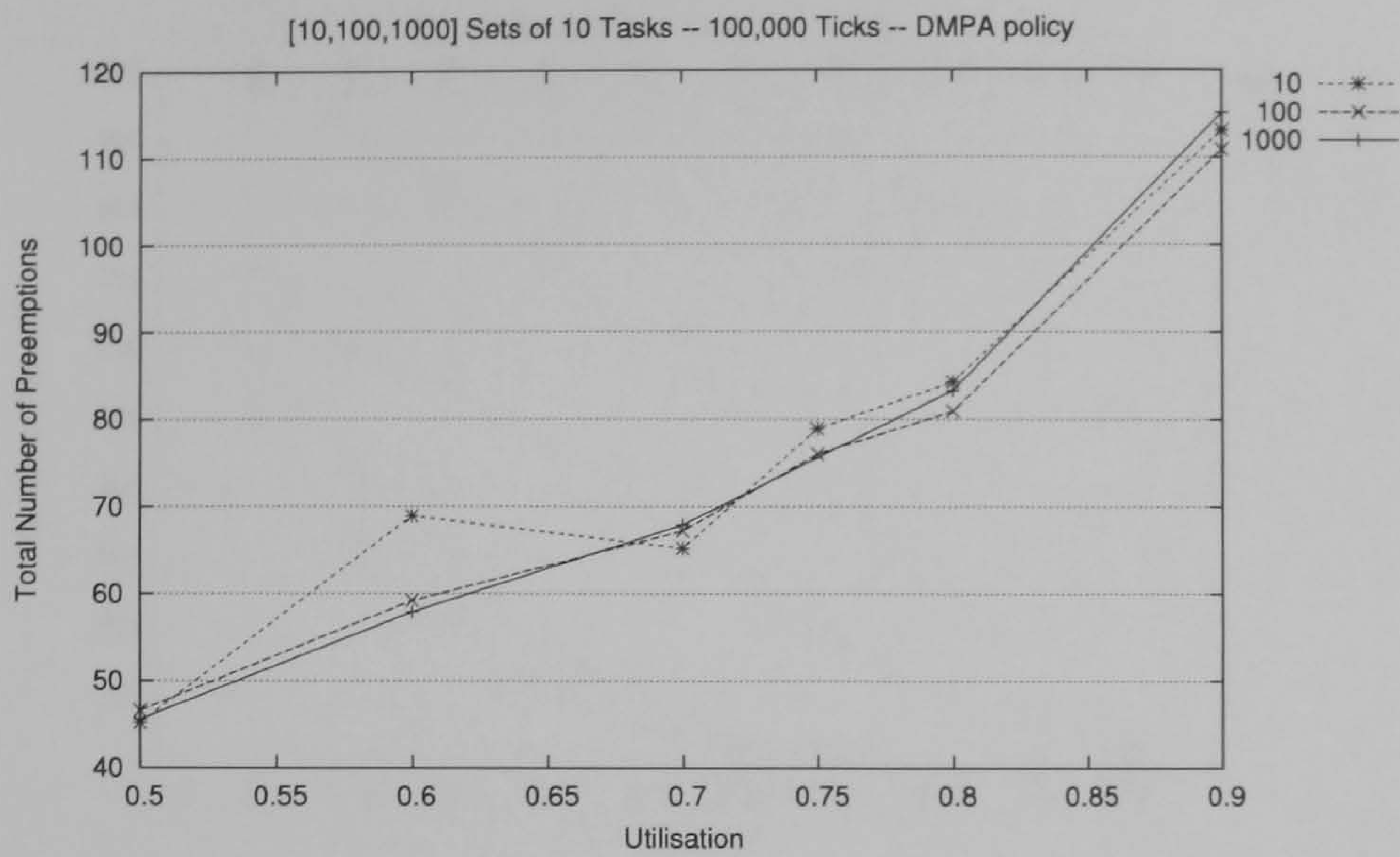


(b) The data is not completely steady but after 70,000 ticks the variation is minimal

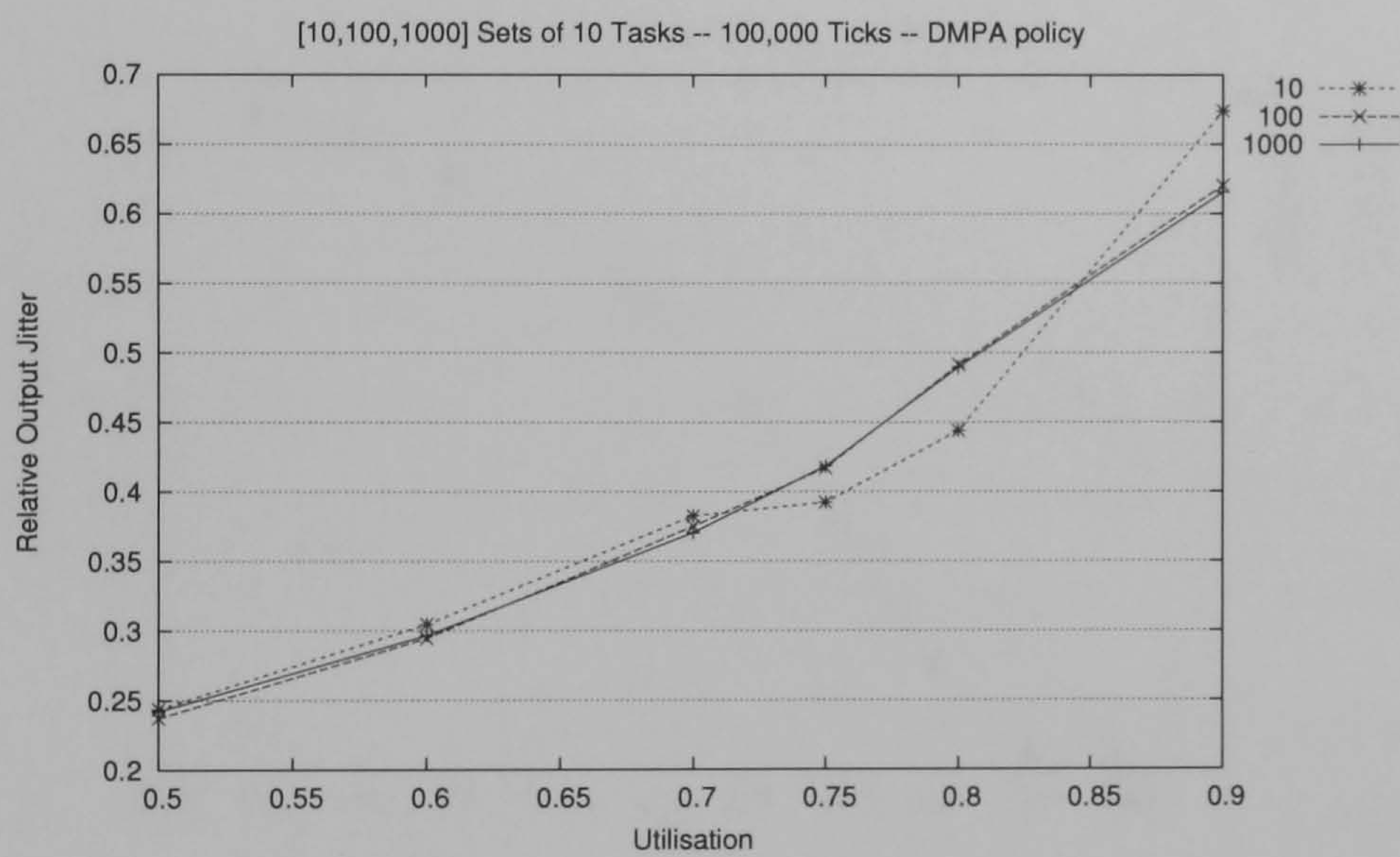


(c) The data is practically steady after 100,000 ticks

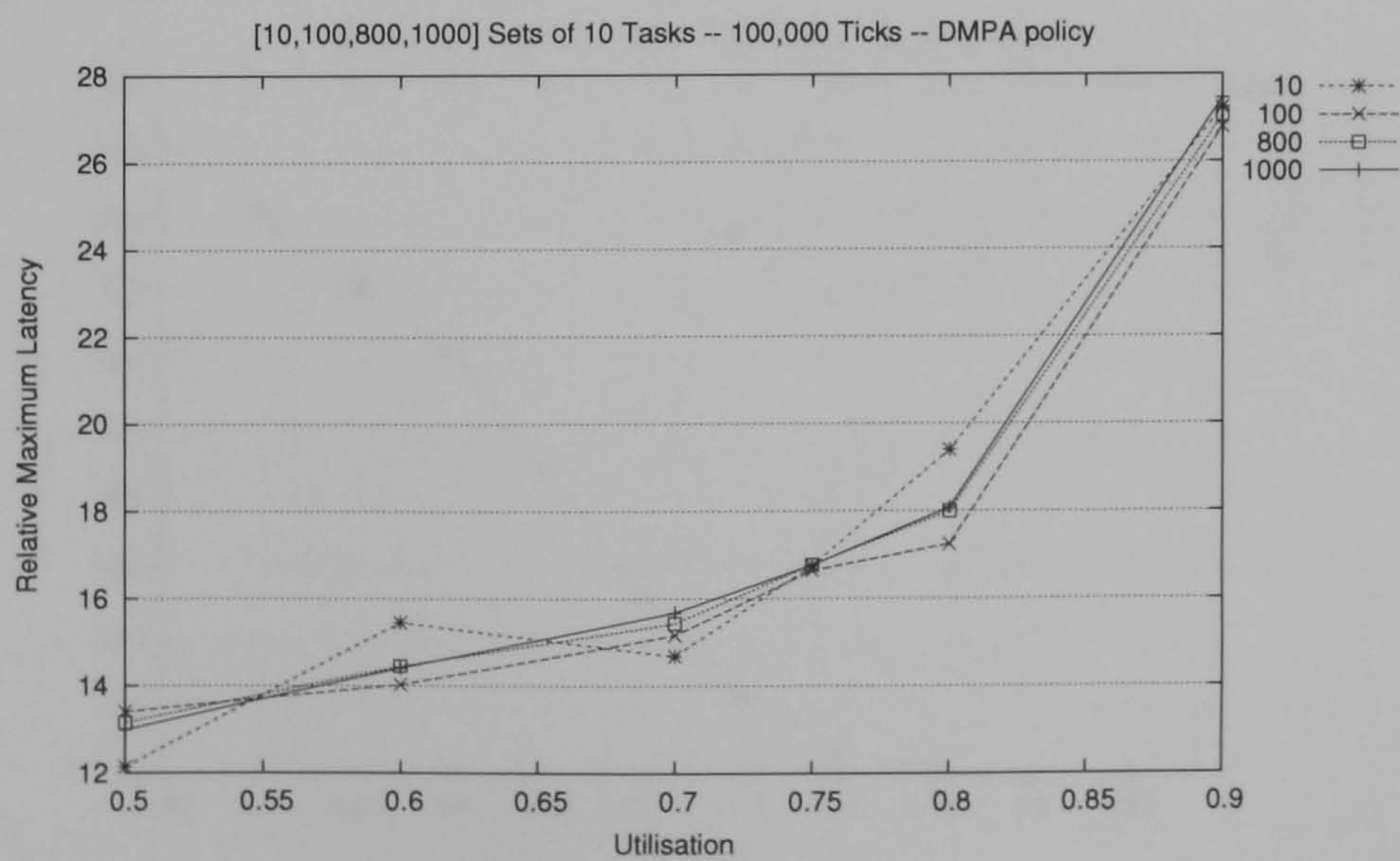
Figure 7.2: Variation on maximum latency, relative maximum latency, and relative average response time with respect to the time window simulated



(a) The variation is low when more than 100 task sets are simulated

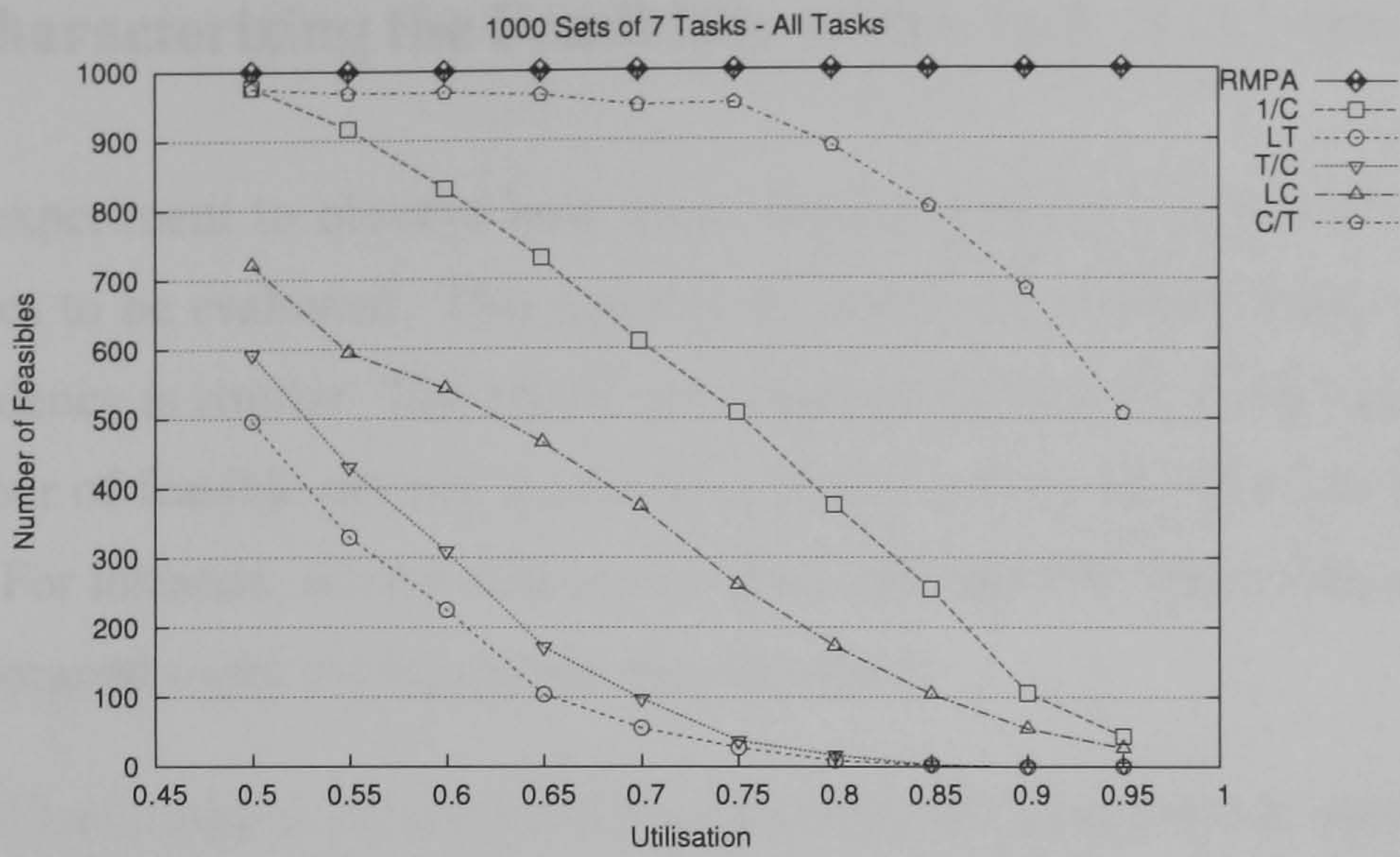


(b) The variation is low when more than 100 task sets are simulated

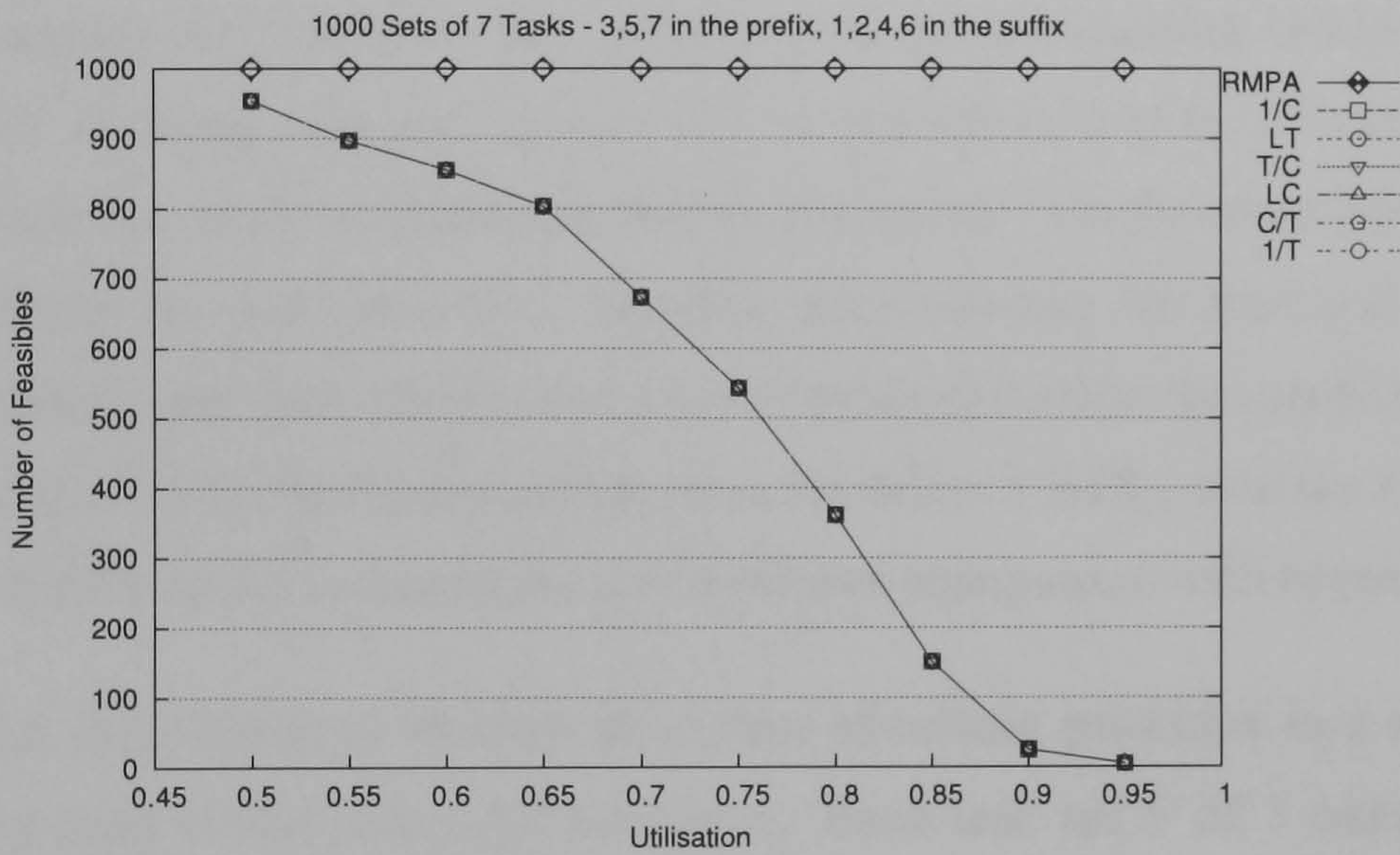


(c) The variation is low when more than 800 task sets are simulated

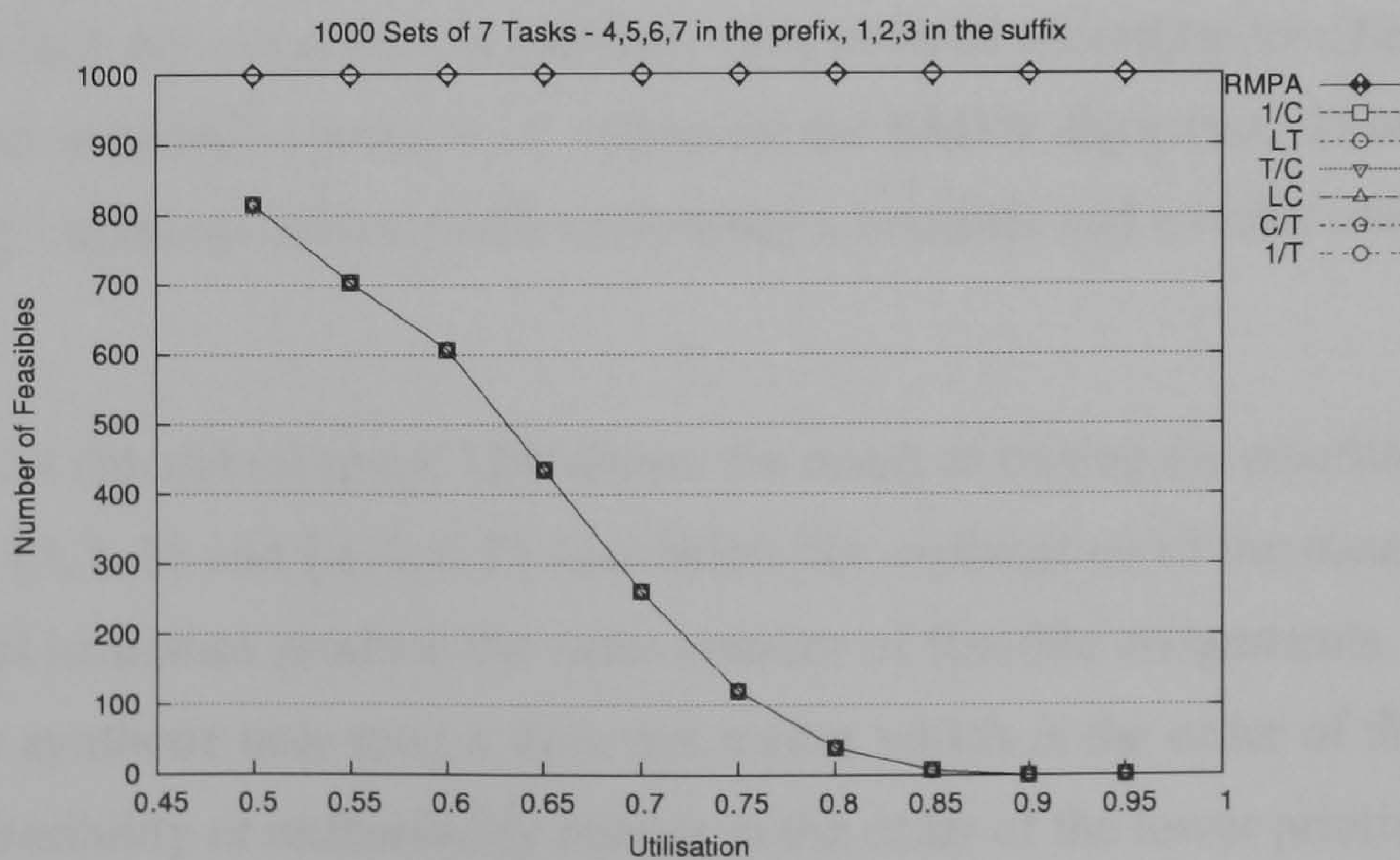
Figure 7.3: Variation of results of different metrics with respect to the number of task sets simulated. When more than 100 task sets are simulated, low variation is observed



(a) Among all heuristics, C/T produces the greater number of feasible ones



(b) All heuristics produce the same number of feasible assignments



(c) All heuristics produce the same number of feasible assignments

Figure 7.4: Number of feasible priority assignments. Under RMPA all tasks sets are feasible. The heuristics get worse as the utilisation increase

7.2.4 Characterizing the Feasibility of the Task Sets Generated

We do an experiment to observe how many feasible priority assignments are found by the heuristics to be evaluated. This experiment is for sets of seven tasks. For other task sets the tendency is similar. The results are shown in Figure 7.4. Each point corresponds to the number of feasible priority assignments computed for all tasks sets under different utilisation. For instance, all task sets are feasible under RMPA, while task sets with fixed-priorities assigned under the heuristics miss deadlines.

Figure 7.4 (a) (page 124) shows the results when all tasks are included in the assignment. Observe that the number of feasible priority assignments decreases as the utilisation increases. Among all heuristics, C/T produces the greater number of feasible ones followed by 1/C. It seems coherent because C/T can be considered as 1/T (i.e. RMPA) with weights C and then many assignments will be equivalent. On the other hand, it is known that 1/C reduces the makespan (i.e. the difference between the start and finish time of a sequence tasks) and then shorter tasks have higher priorities for completing soon; this reduces the interference and increases the schedulability. Finally, note the LT rule; it is the contrary to RMPA and it is clearly the worst priority assignment with respect to deadlines.

We do an experiment to observe the effect of raising priorities in a subset of tasks and ordering such subset using the heuristics. Each task set S of 7 tasks is split up in two subsets A and A'. Tasks in A are more important than the ones in A' and therefore higher priorities are assigned. In addition, A is ordered according to a heuristic. Lower priorities are assigned to tasks in A' following the RMPA algorithm. In other words, we have priority orderings with a prefix ordered by a heuristic and a suffix ordered according to RMPA.

Figure 7.4 (b) and (c) (page 124) shows the result of raising the priorities of subsets A labelled as $\{3, 5, 7\}$ and $\{4, 5, 6, 7\}$ (see below for explanation of the meaning of labels). Note that all heuristics produce the same number of feasible assignments. This indicates that (in our synthetic task sets) it does not matter which is the order of the high priority tasks, the feasibility or unfeasibility resides in the order of the lower priority ones.

This last result is significant for our evaluation. This means that when assigning im-

PA	Priority Assignment Rule
1/T	the shorter the period, the higher the priority
1/C	the shorter the computation time, the higher the priority
LT	the larger the period, the higher the priority
LC	the larger the computation time, the higher the priority
T/C	the larger the ratio T/C, the higher the priority
C/T	the larger the ratio C/T, the higher the priority

Table 7.1: Simple rules for assigning priorities

portance with the heuristics, higher importance tasks will remain practically fixed in their positions while only the lower ones will be changed. Thus, algorithms that exchange priorities starting from the lower importance tasks will achieve good solutions in terms of the lexicographic order. In addition, note that some heuristics produce a significant number of feasible priority assignments. As a consequence of this scenario, the DI algorithm and the swapping algorithm produce almost the same results. For example, a peculiar result is that in all our experiments, when importance is assigned with the rule "the shorter computation, the higher the importance" (1/C rule), the DI algorithm and the swapping algorithm always produce the same priority ordering.

7.3 Finding Heuristics for Assigning Importance

A performance evaluation of the fixed-priority assignment rules depicted in Table 7.1 (this is the same table from page 81) is carried on to determine how good (or bad) they are with regard to a QoS metric. If at least one of such priority assignments shows good performance, it is considered a good heuristic and is selected as a rule for assigning importance.

In order to measure effectively the performance of the simple rules, the maximum and minimum values achievable (with respect to a particular metric) for any fixed-priority assignment are determined by exhaustive search in the $N!$ priority orderings of a task set. Thus, all results are between the maximum and minimum achievable and therefore, they can be normalized. The experiments were conducted for tasks sets of 5, 6 and 7 tasks. The results obtained are similar and therefore, only results for 7 tasks are shown.

In order to identify orderings similar to RMPA, we label tasks as $\{1, 2, \ldots . 7\}$ where

1 corresponds to the task with the shortest period and 7 to the one with the largest period; thus, the ordering by RMPA is $\langle 1\,2\,3\,4\,5\,6\,7\rangle$. The experiments are as follows:

- *All tasks included.* Priorities are assigned to tasks according to the rules in Table 7.1. The orderings are simulated and the QoS metrics are recorded for all seven tasks.
- *Tasks 3,5,7 in the prefix.* Orderings are built with a prefix ordered by the rules in Table 7.1 and a suffix ordered by RMPA. Tasks in the prefix are $A = \{3, 5, 7\}$. The orderings are simulated and the QoS metric is measured only for tasks in the prefix. Note that $\langle 1\,2\,4\,6\rangle$ is the suffix for all the orderings in this experiment.
- *Tasks 4,5,6,7 in the prefix.* As above, but the tasks in the prefix are $A = \{4, 5, 6, 7\}$. Note that $\langle 1\,2\,3\rangle$ is the suffix for all the orderings in this experiment.

We noted in our experiments that, in general, it is difficult to distinguish the differences among the results when plotting absolute values. For this reason, in some plots we normalize all the data with respect to the maximum and minimum achievable. This problem is illustrated in the first experiment.

The results are illustrated by drawing plots of different priority assignments with respect to a QoS metric plotted as a function of the total system utilisation. Each point corresponds to the average results of 1000 task sets. The results are displayed such that minimum values of the QoS metric are better.

For instance, let assume us that the metric is the total number of preemptions. According to section 4.6.2 (page 68), for a set S of N tasks and with respect to the schedule under a particular priority ordering α, lets $\mathcal{P}_S(\alpha)$ be the total number preemptions in a window of time (i.e. 100,000 ticks) of all instances of all tasks in S (note that $\mathcal{P}_S(\alpha)$ corresponds to equation 4.6.2 in section 4.6.2). By simulating the $N!$ priority assignments of S, we have a set $\mathcal{P}_S = \{\mathcal{P}_S(\alpha_k) : k = 1, 2, \dots, N!\}$ which has the total number of preemptions of all orderings of S. The maximum and minimum total number of preemptions of a task set S under any priority assignment are $\mathcal{P}_S^{\max} = \max\{\mathcal{P}_S\}$ and $\mathcal{P}_S^{\min} = \min\{\mathcal{P}_S\}$ respectively. Thus, the total number of preemptions normalized for a particular priority assignment α is $\bar{\mathcal{P}}_S(\alpha) = (\mathcal{P}_S(\alpha) - \mathcal{P}_S^{\min})/(\mathcal{P}_S^{\max} - \mathcal{P}_S^{\min})$.

Since we have task sets $S_1, S_2, \ldots, S_{1000}$ with identical utilisation, for a particular priority assignment α of S_i with total number of preemptions $\mathcal{P}_{S_i}(\alpha)$, its average total number of preemptions is computed by the function

$$F_{\mathcal{P}}(\alpha) = \frac{\sum_{i=1}^{1000} \mathcal{P}_{S_i}(\alpha)}{1000}$$

and the average total number of preemptions normalized is computed by the function

$$\bar{F}_{\mathcal{P}}(\alpha) = \frac{\sum_{i=1}^{1000} \bar{\mathcal{P}}_{S_i}(\alpha)}{1000} \tag{7.3.1}$$

In addition, the maximum and minimum average total number of preemptions are

$$\text{MAX} = \frac{\sum_{i=1}^{1000} \mathcal{P}_{S_i}^{\max}}{1000} \quad \text{and} \quad \text{MIN} = \frac{\sum_{i=1}^{1000} \mathcal{P}_{S_i}^{\min}}{1000}$$

respectively.

For the other metrics, $\mathcal{P}$ can be substituted by the corresponding metric $\mathcal{J}$, $\mathcal{J}^{rel}$, $\mathcal{L}$, $\mathcal{L}^{rel}$ or $\mathcal{R}$.

7.3.1 Total Number of Preemptions

The first experiment compares how good are the heuristics with regard to the total number of preemptions. The results are shown in Figures 7.5 and 7.6.

In Figure 7.5(a) (page 132), each point corresponds to the average total number of preemptions of 1000 task sets for different utilisation. By using the above equations, $F_{\mathcal{P}}(\alpha)$ is computed for α in { RMPA, 1/C, LT, T/C, LC, C/T }; for instance, $F_{\mathcal{P}}(\text{RMPA}) = 45$, $F_{\mathcal{P}}(1/\text{C}) = 56$ and $F_{\mathcal{P}}(\text{LC}) = 31$ when the utilisation is 0.5. In addition, the results for the EDF schedule, and the maximum (MAX) and minimum (MIN) average total number of preemptions are also plotted. Note that it is difficult to distinguish the differences among the results of the different priority assignments. For this reason, we draw the plot of the normalized data in Figure 7.6.

In Figure 7.6(a) (page 133), the total number of preemptions normalized is depicted. The equation 7.3.1 is used; for instance, $\bar{F}_{\mathcal{P}}(\text{RMPA}) = 0.52$, $\bar{F}_{\mathcal{P}}(1/\text{C}) = 0.94$ and

$\bar{F}_{\mathcal{P}}(\text{LC}) = 0.06$ when the utilisation is 0.5. Observe that the differences are now evident. None of the orderings achieve the minimum total number of preemptions possible.

When all tasks are included (Figures 7.5(a) and 7.6(a)), the best heuristic is LC followed by C/T. Note that EDF shows great performance. When only a subset is included (Figures 7.5(b, c) and 7.6(b, c)), the best heuristics are LC and C/T, which are almost overlapped. In fact, all heuristics produce good results for a task subset for all QoS metrics of this study. This is coherent because all tasks in the subset have higher priorities and therefore suffer less interference. Naturally, RMPA and EDF perform poorly because they both are unaware of local requirements; it also applies for all metrics in this study. *We can conclude that the rules LC and C/T are good heuristics for assigning importance when the total number of preemptions is the QoS metric.*

7.3.2 Output Jitter

This experiment compares how good are the heuristics with regard to the absolute output jitter and the relative output jitter. In Figure 7.7 (page 134), each point corresponds to the average absolute output jitter of 1000 task sets for different utilisation. By observing Figure 7.7(a), it seems that the best heuristics are 1/C, T/C and LT when all tasks are included. Therefore, we will assume that these are the best heuristics. However, as depicted in Figure 7.4, T/C and LT are the rules that produce less feasible priority assignments and hence, it may be difficult to find feasible priority orderings similar to them. When only a task subset is included, the best are 1/C and T/C (Figure 7.7(b, c)).

In Figure 7.8 (page 134), the results of the average relative output jitter are shown. When all tasks are included, all heuristics show bad performance (Figure 7.8(a)). Note how EDF outperforms all fixed-priority assignments. RMPA performs better than all the heuristics and it is close to the minimal relative output jitter achievable. As a local metric (Figure 7.8(b, c)), 1/T is the best heuristic followed by 1/C.

As in the preemptions experiment, the plots suggest that increasing the priority is sufficient to reduce the output jitter effects with respect to a task subset. We can conclude that with respect to output jitter:

- *for absolute output jitter, the rules T/C, LT and 1/C could be suitable to be used as importance assignments.*
- *for relative output jitter, ordering by RMPA produce the best results followed by C/T. As a local metric, the best ones are 1/T and 1/C.*

7.3.3 Latency

The results of the experiments with latency are depicted in Figures 7.9 and 7.10. In Figure 7.9 (page 136), the maximum latency results are shown. As a global metric (Figure 7.9(a)), the best heuristics are LT and T/C. Unfortunately, when these priority assignments were used as inputs for the DI algorithm, the results were worse than the results produced by RMPA. It is because, as depicted in Figure 7.4 (page 124), these two rules are the worst with respect to feasibility and hence, it may be difficult to find feasible priority orderings rather close to them to be useful. As a local metric (Figure 7.9(b, c)), the best is LC even so as in the above experiments, the plot suggests that increasing the priority is sufficient to improve the performance with respect to the metric.

In Figure 7.10 (page 137), the relative maximum latency results are shown. In general, The best heuristic is 1/C followed by T/C. We can conclude that with respect to latency:

- *for maximum latency, the rules LT, T/C and 1/C could be suitable to be used as importance assignments.*
- *for relative maximum latency, 1/C and T/C are the best ones.*

7.3.4 Maximum Relative Average Response-Time

The results of the experiments with respect to the maximum relative average response-time are depicted in Figure 7.11 (page 138). In all cases, the best heuristic is T/C which shows excellent performance, followed by 1/C and LT.

We can conclude that in this case the rule T/C is excellent and the rules 1/C and LT are good heuristics.

In the next section we will use the heuristics with the DI algorithm for finding solutions to multicriteria scheduling problems. (*Continue in page 131*).

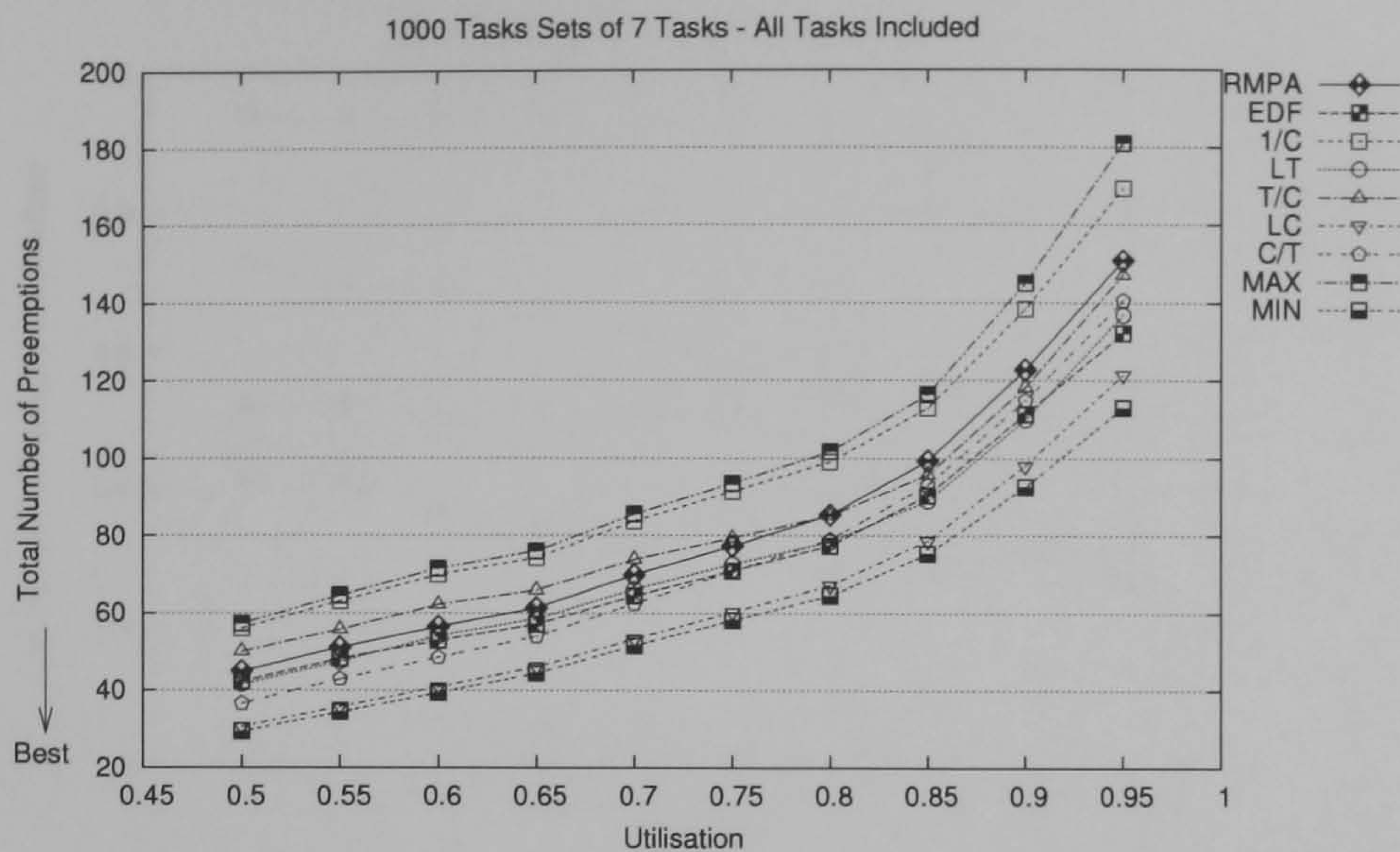
7.4 Evaluating the DI Algorithm with Heuristics

The best heuristics discovered in the above sections are used for assigning importance such that when used as input for the DI algorithm, solutions to the scheduling problems defined in chapter 4 are proposed. The results are plotted in groups as follows:

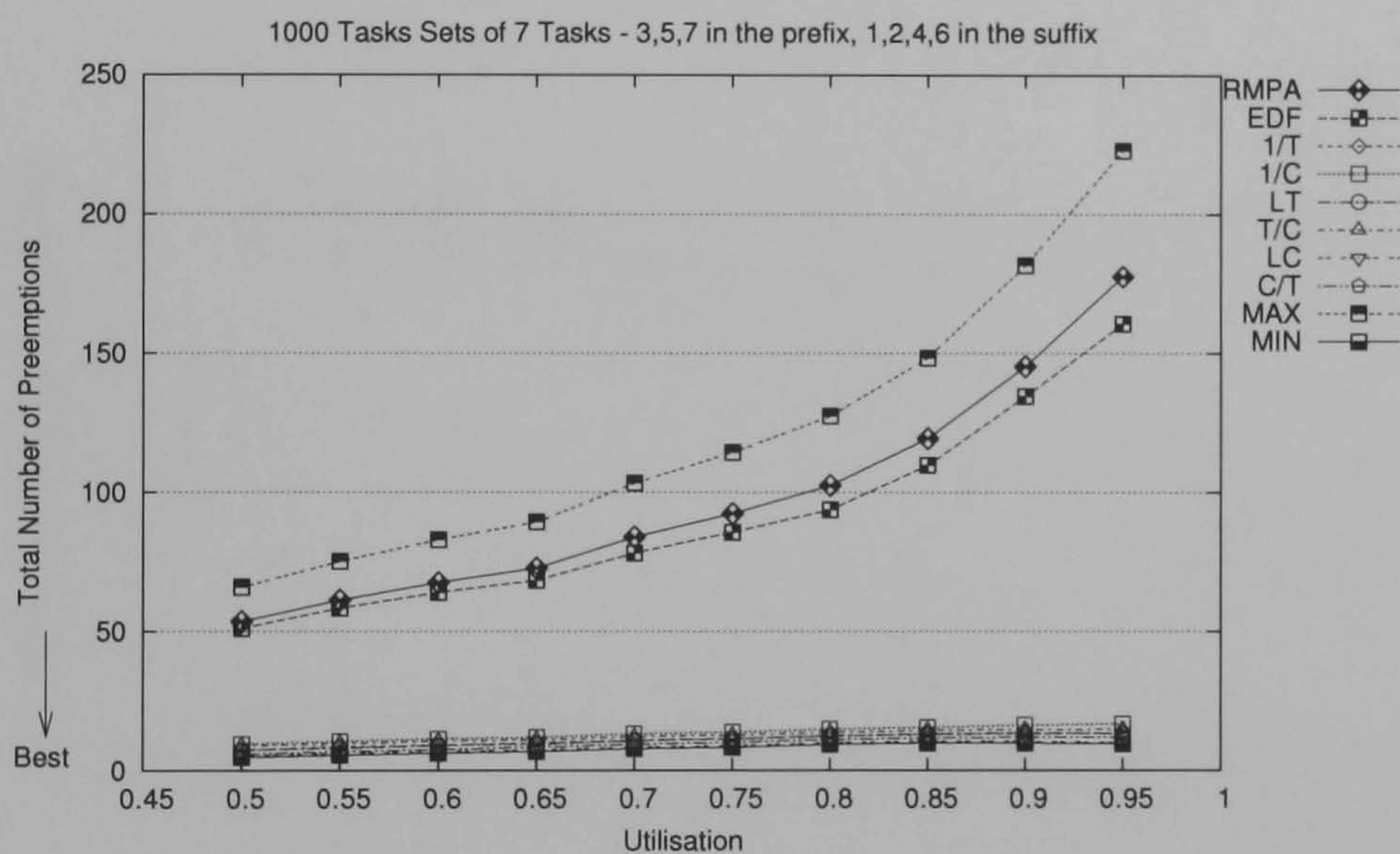
PLOTS A. The solutions obtained by combining the DI algorithm with at least two heuristics for assigning importance are evaluated. In order to effectively compare their performance, the maximum and minimum values achievable and the Z-*optimal* solution ($\mathsf{FP}_{\mathsf{opt}}$ in the plots) are determined by exhaustive search. Results of the experiments for sets of 5, 6 and 7 tasks are similar and therefore, only results for 7 tasks are shown.

PLOTS B. The combination DI with heuristic is computed for sets of N tasks (where $N = 5, 6, \ldots, 14$) with utilisations $U = \{0.5, 0.7, 0.9\}$. The results are not normalized since we do not compute the $N!$ possible solutions.

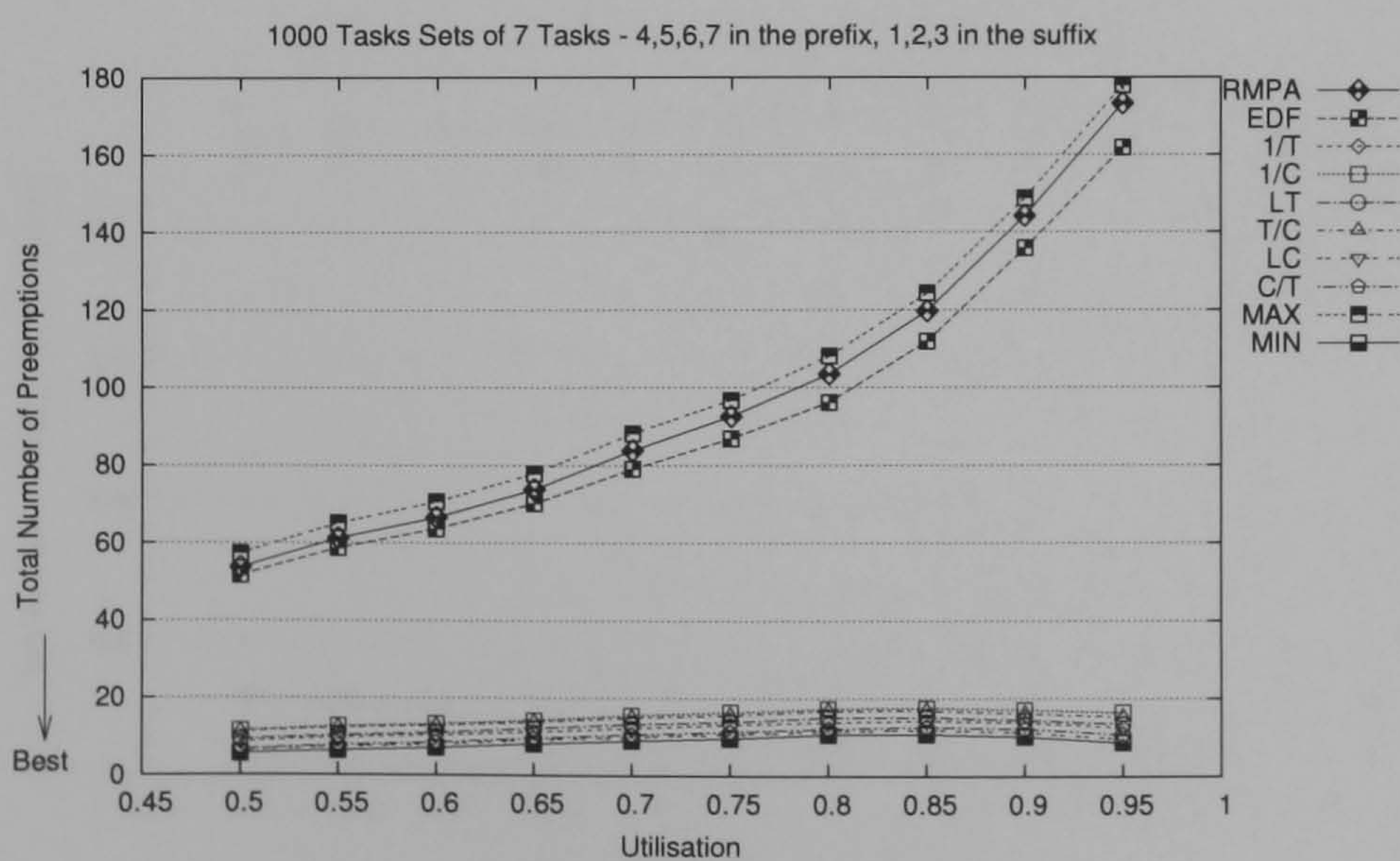
PLOTS C. Since that each point in any plot of type A corresponds to the average result of 1000 task sets, there exists information that is hidden and is really interesting to show. This information is shown in Plots C where we compare the results of the best combination DI algorithm with heuristics against EDF for only three of these points corresponding to $U = \{0.5, 0.7, 0.9\}$. Since each point corresponds to 1000 task sets, there are 1000 points corresponding to the EDF results and 1000 points corresponding to the DI-heuristic results. We order the EDF data such that a baseline is formed when plotted. The DI data is also ordered with respect to the EDF data such that if both results were exactly the same, a single line will be showed. Since this is not true, the DI data appear scattered around the EDF baseline. Points above the baseline are worse and points below are better than the EDF results. These plots correspond to sets of 7 tasks.



(a) The best is *LC*; note how close it is to MIN (the best achievable)

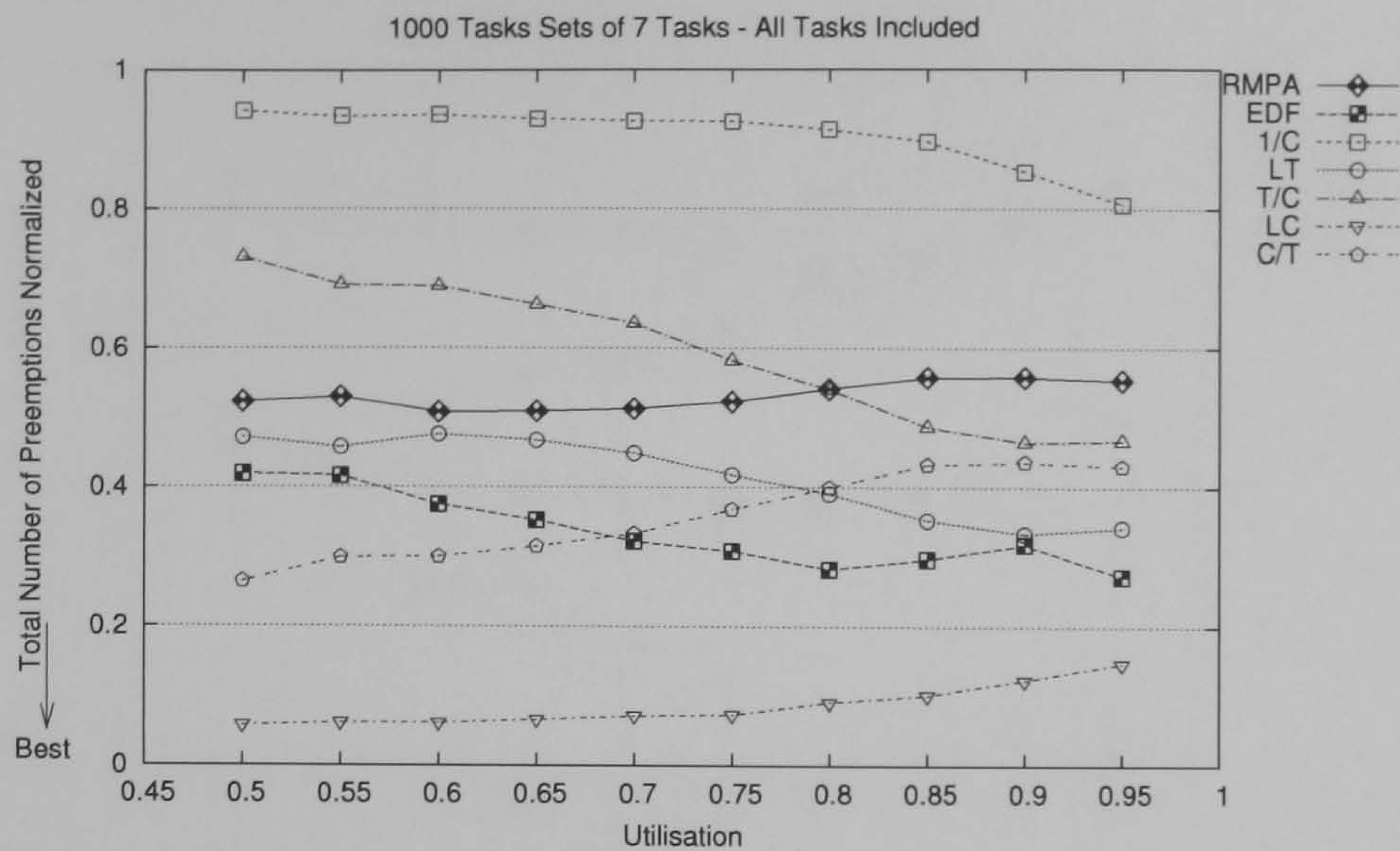


(b) The best is *LC* although the rest show good performance

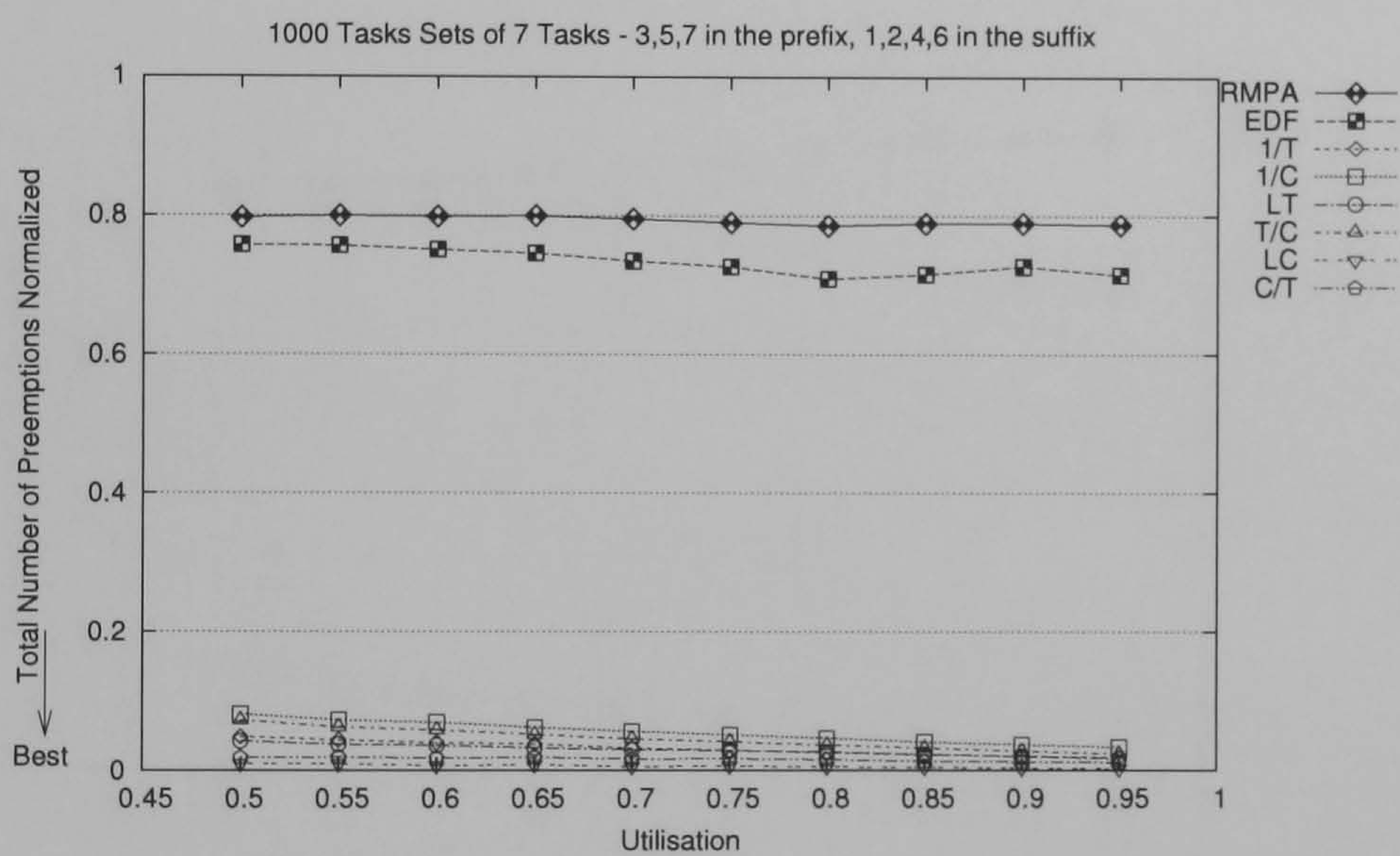


(c) The best is *LC*

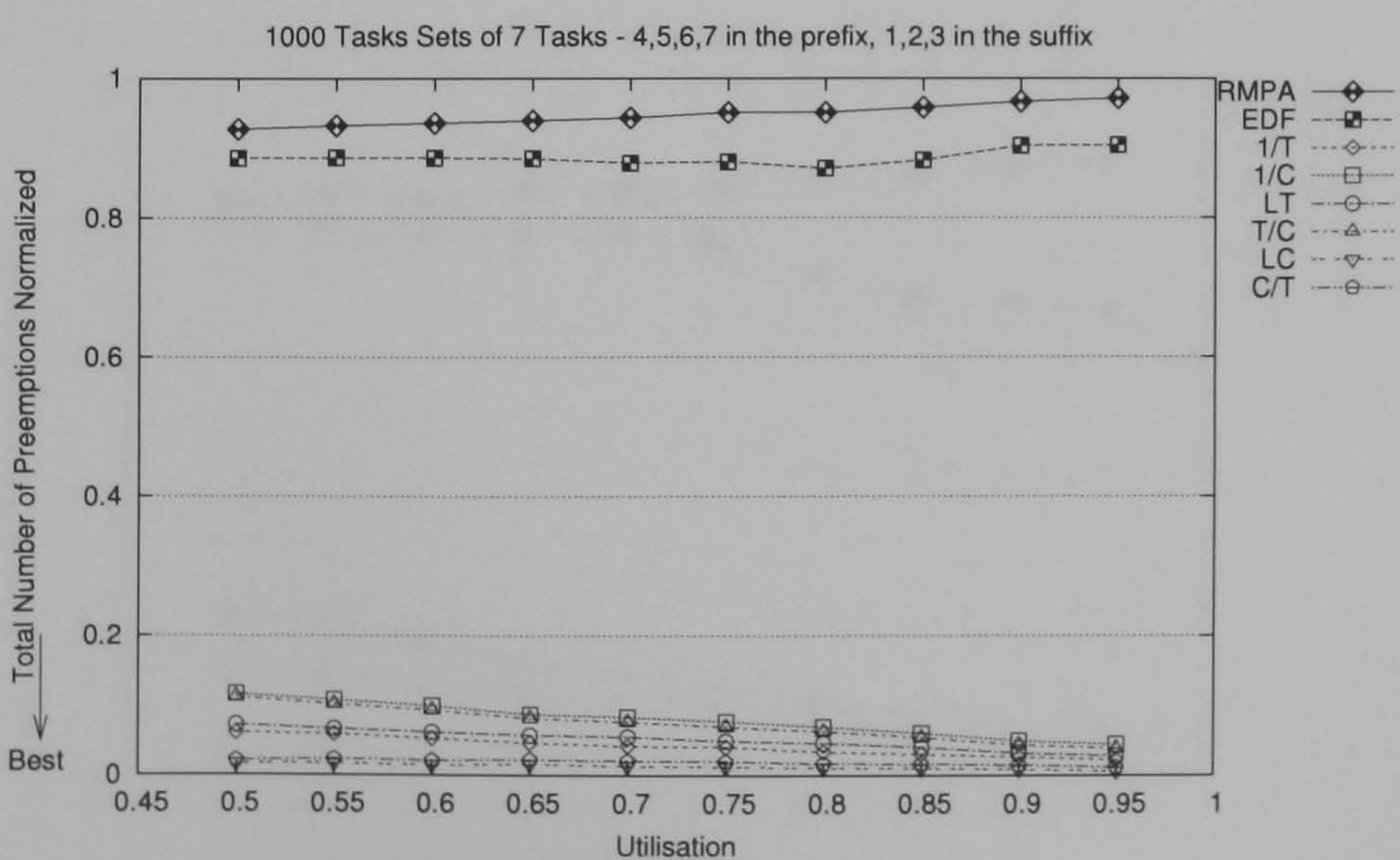
Figure 7.5: Total Number of Preemptions under different priority assignments. Only RMPA and EDF guarantee deadlines. The best heuristic is LC. Note that in (c) RMPA is close to the worst possible value (i.e. MAX)



(a) The best is LC; note how close it is to MIN (the best achievable)

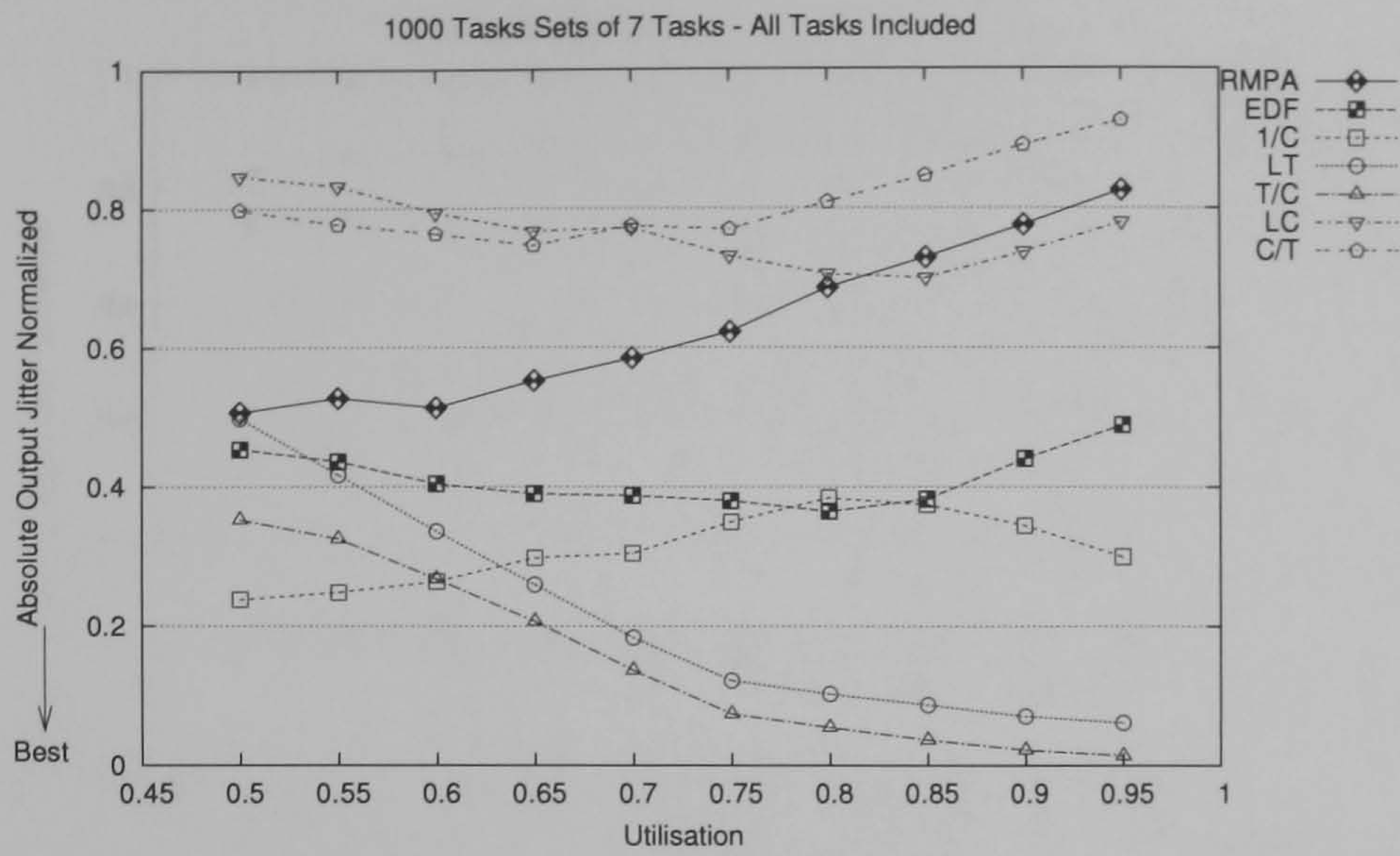


(b) The best is LC although the rest show good performance

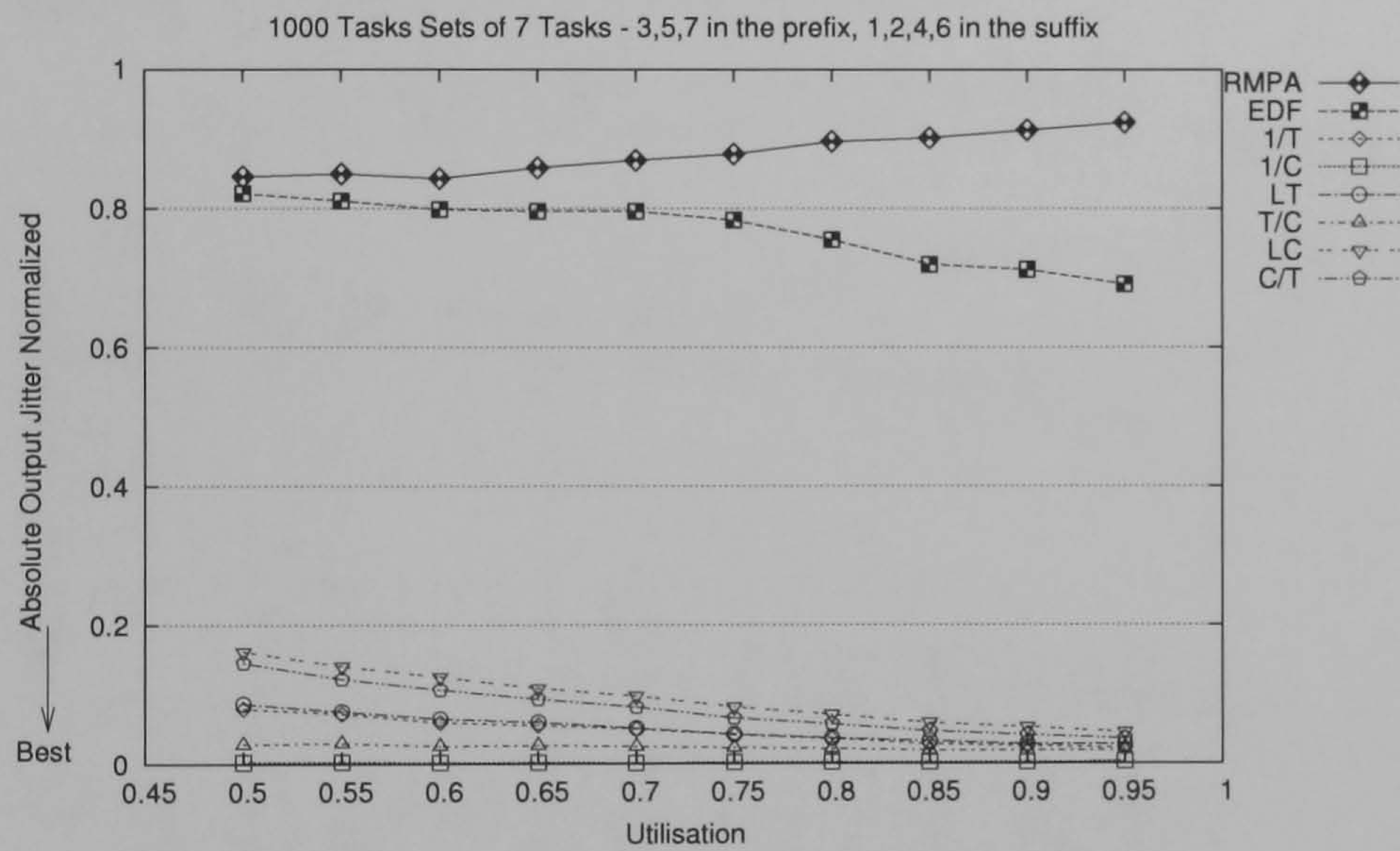


(c) The best is LC

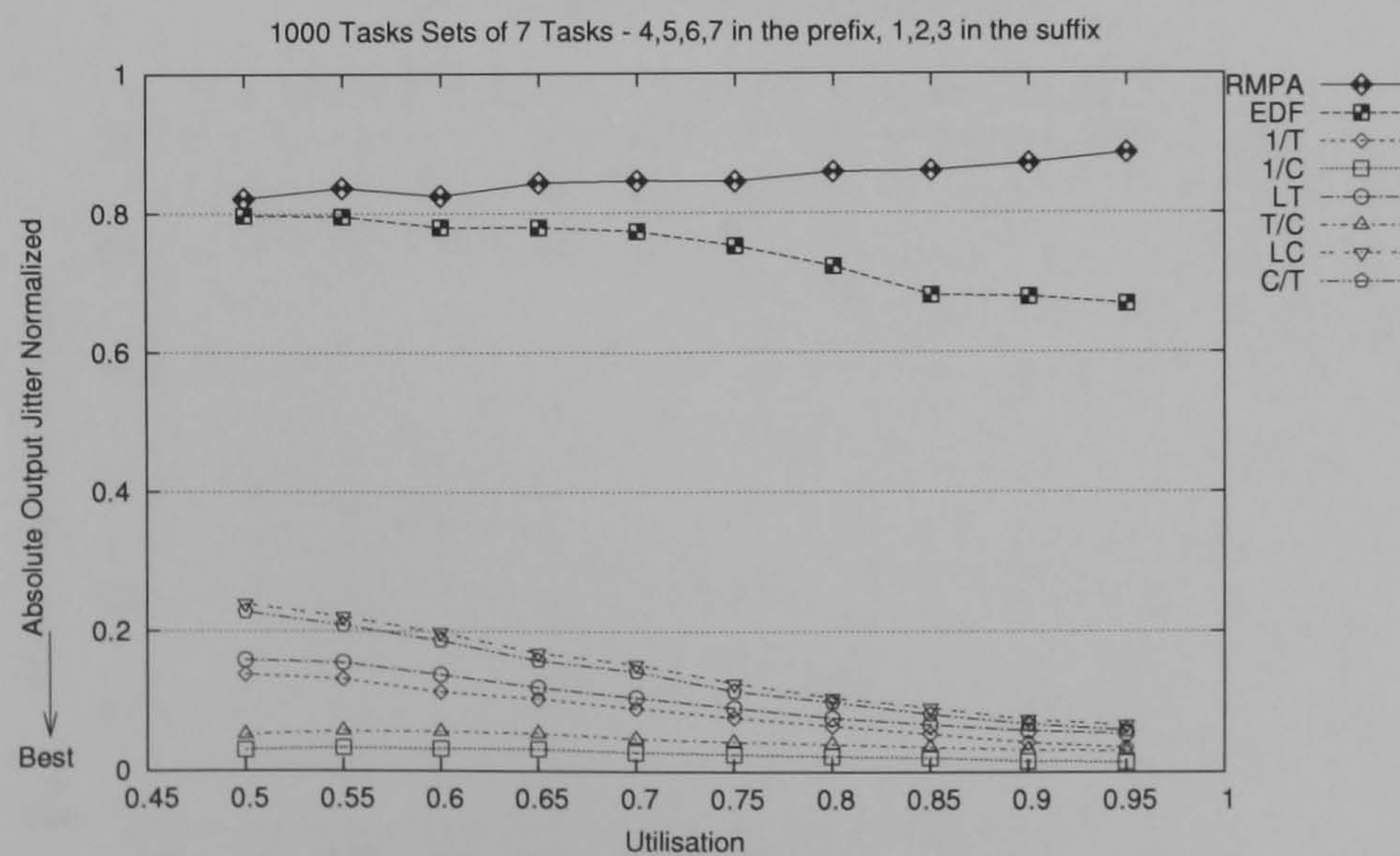
Figure 7.6: Total Number of Preemptions Normalized under different priority assignments. Only RMPA and EDF guarantee deadlines. The best heuristic is LC. Note that in (c) RMPA is close to the worst possible value



(a) The best heuristic is T/C followed by LT and 1/C

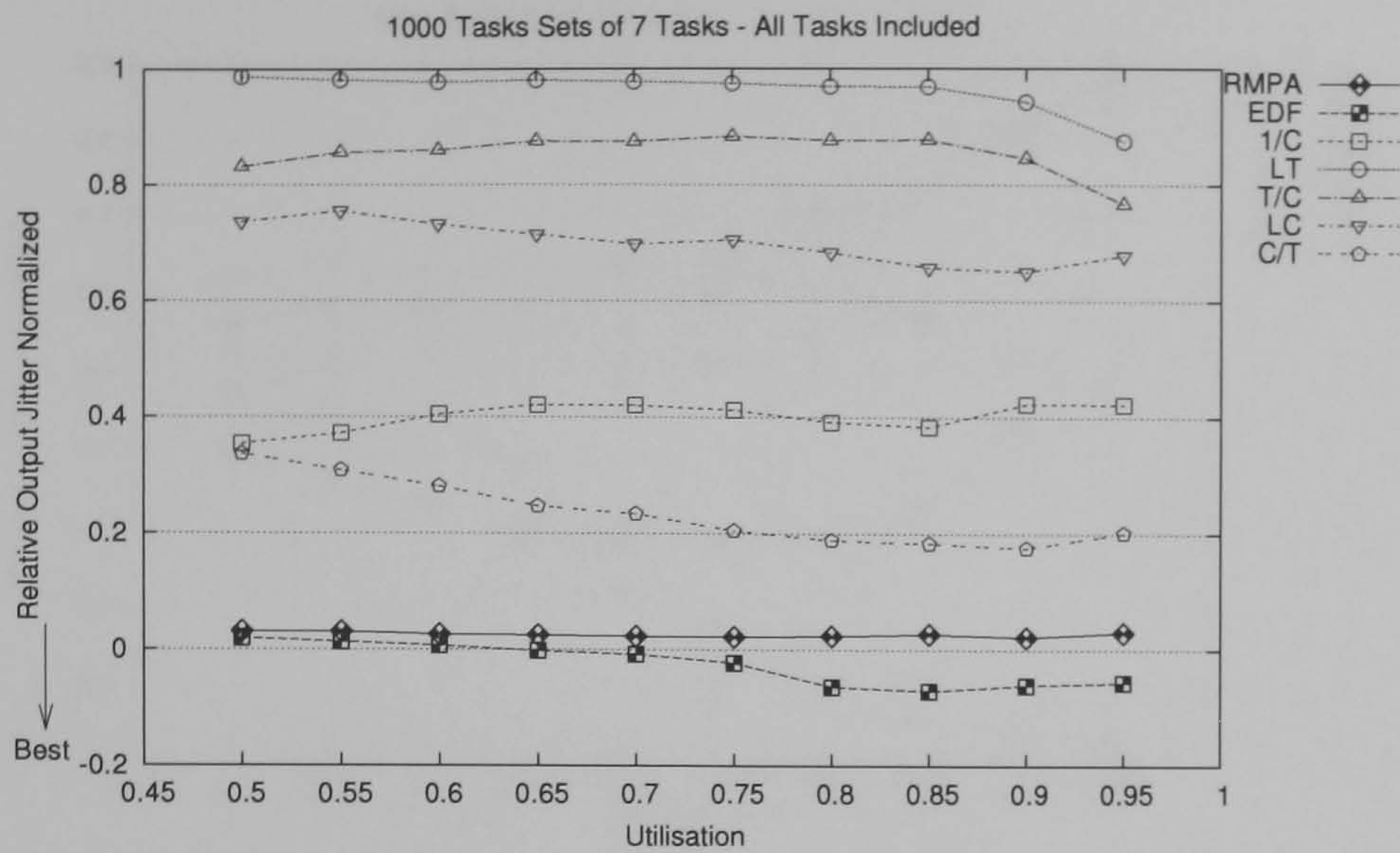


(b) The best heuristic is 1/C followed by T/C

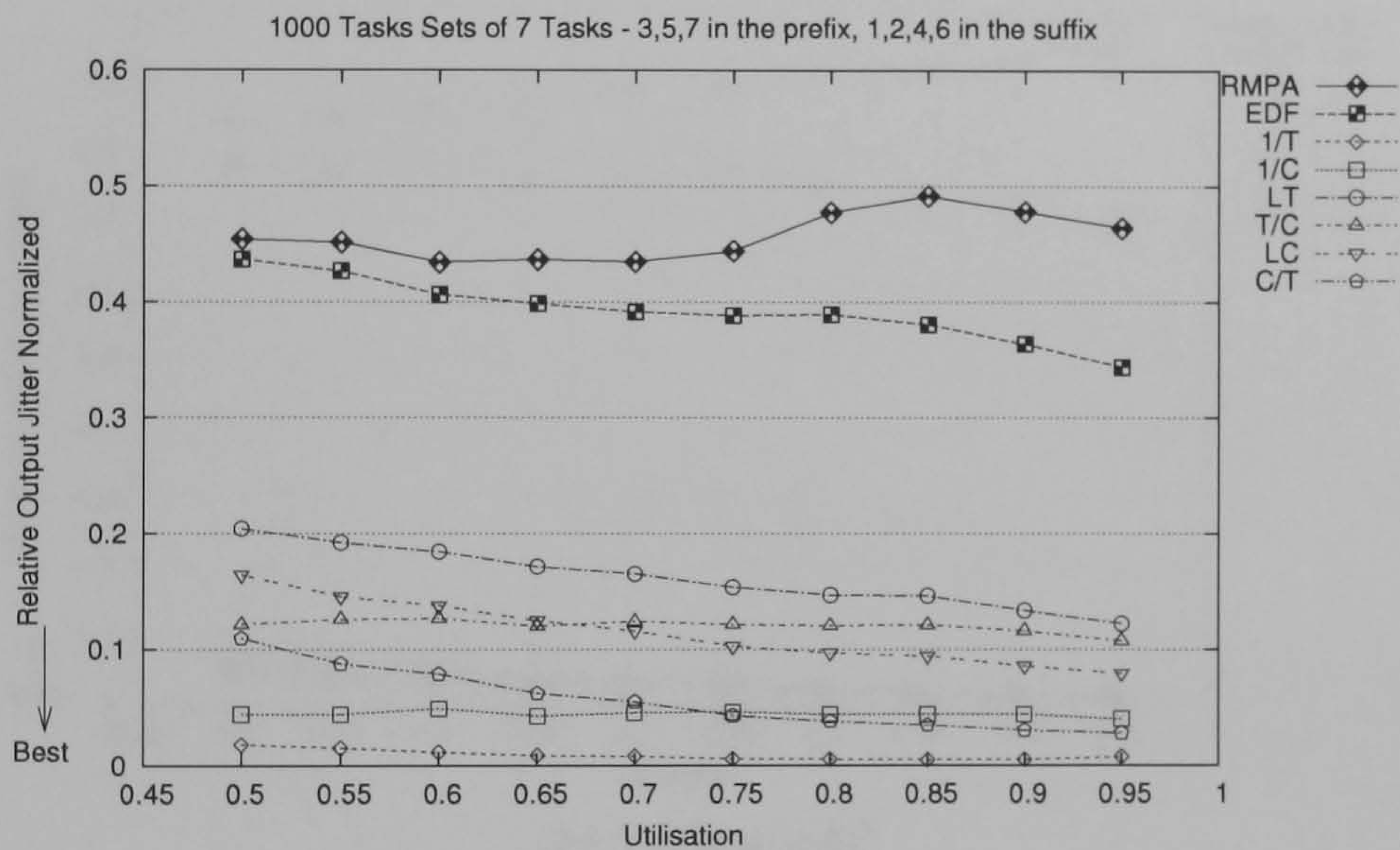


(c) The best heuristic is 1/C followed by T/C

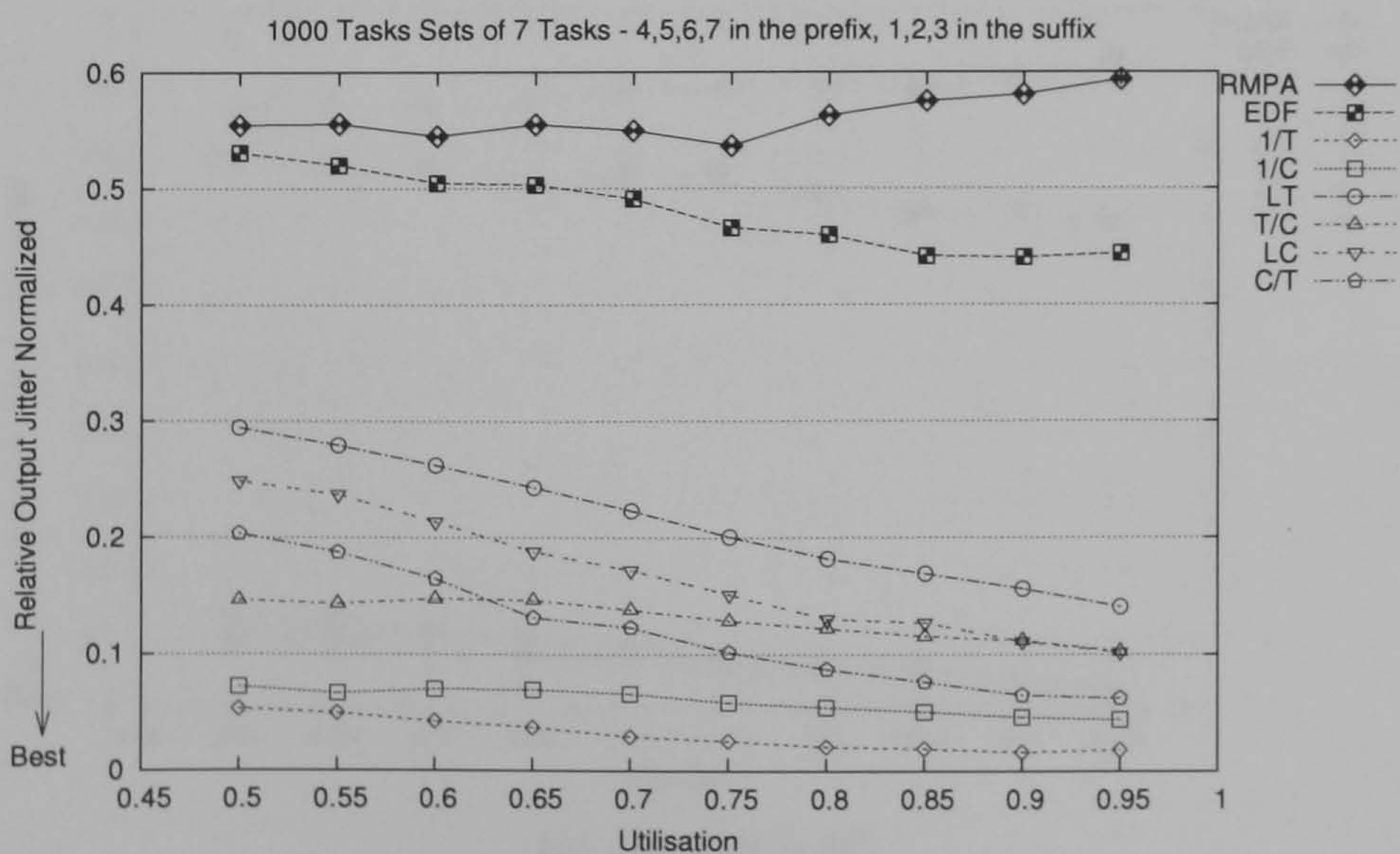
Figure 7.7: Absolute Output Jitter under different heuristics. The best ones are T/C and 1/C



(a) All heuristics show bad performance. RMPA is the best fixed-priority assignment. EDF is by far the best solution achievable

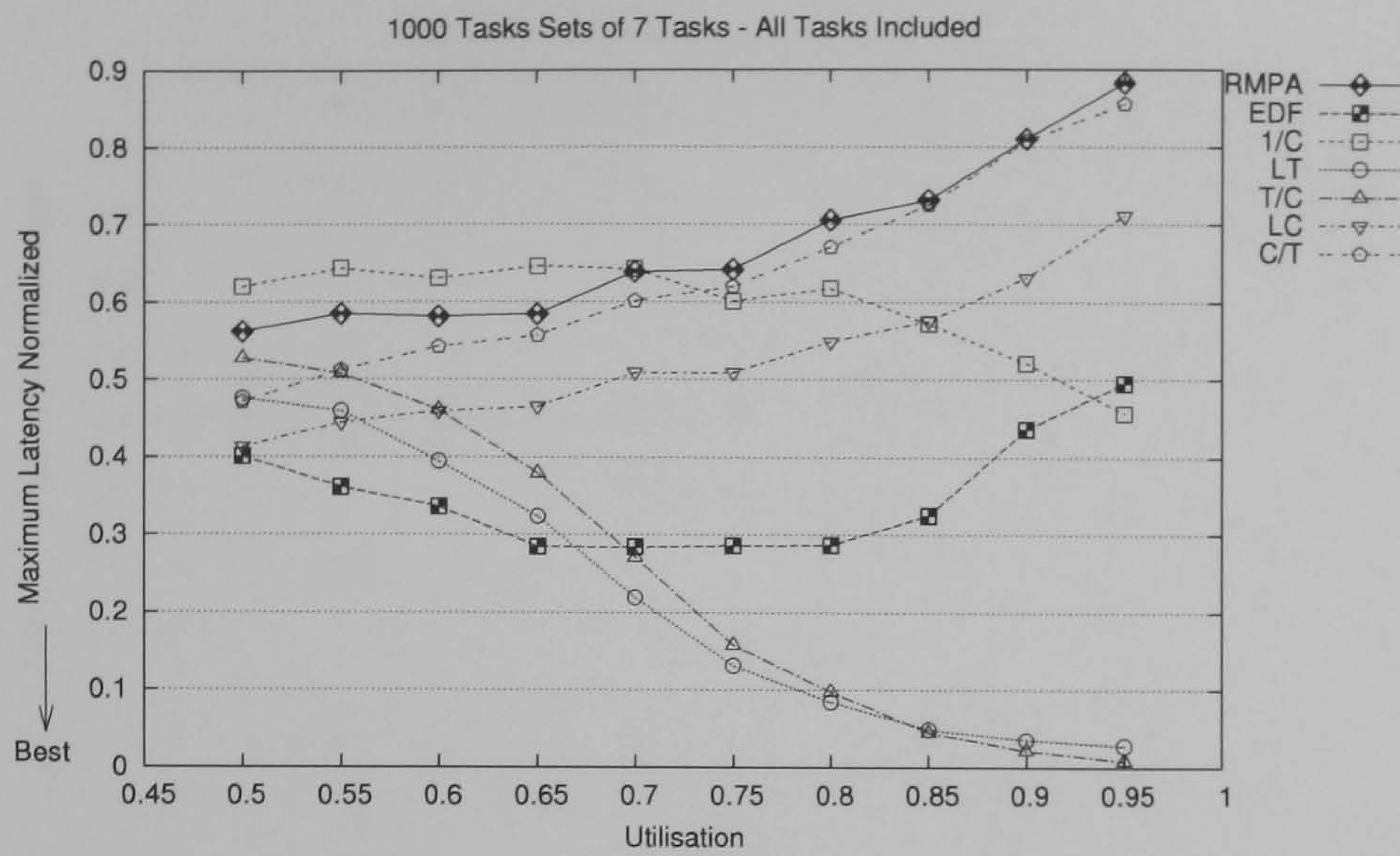


(b) The best is 1/T followed by 1/C

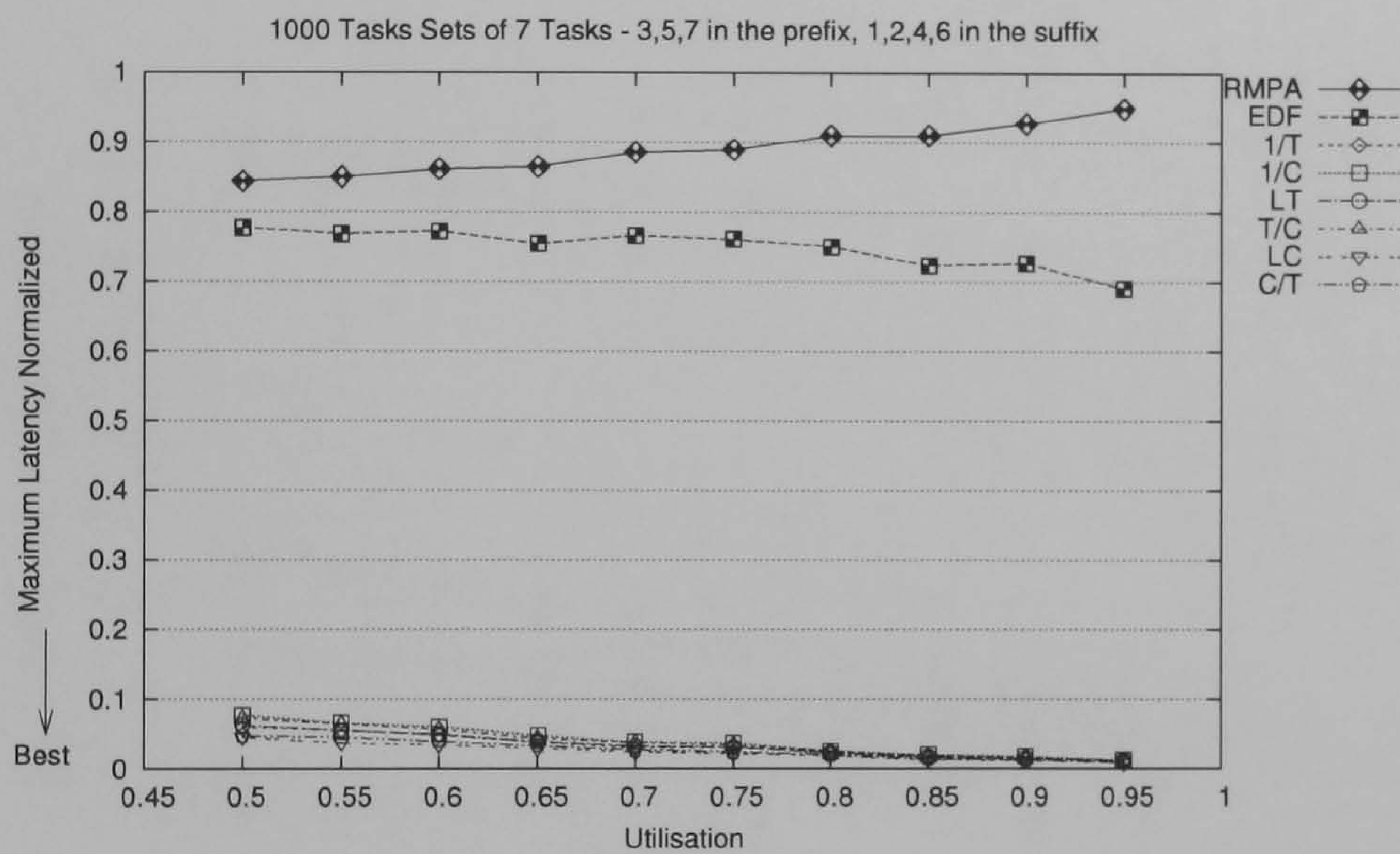


(c) The best is 1/T followed by 1/C

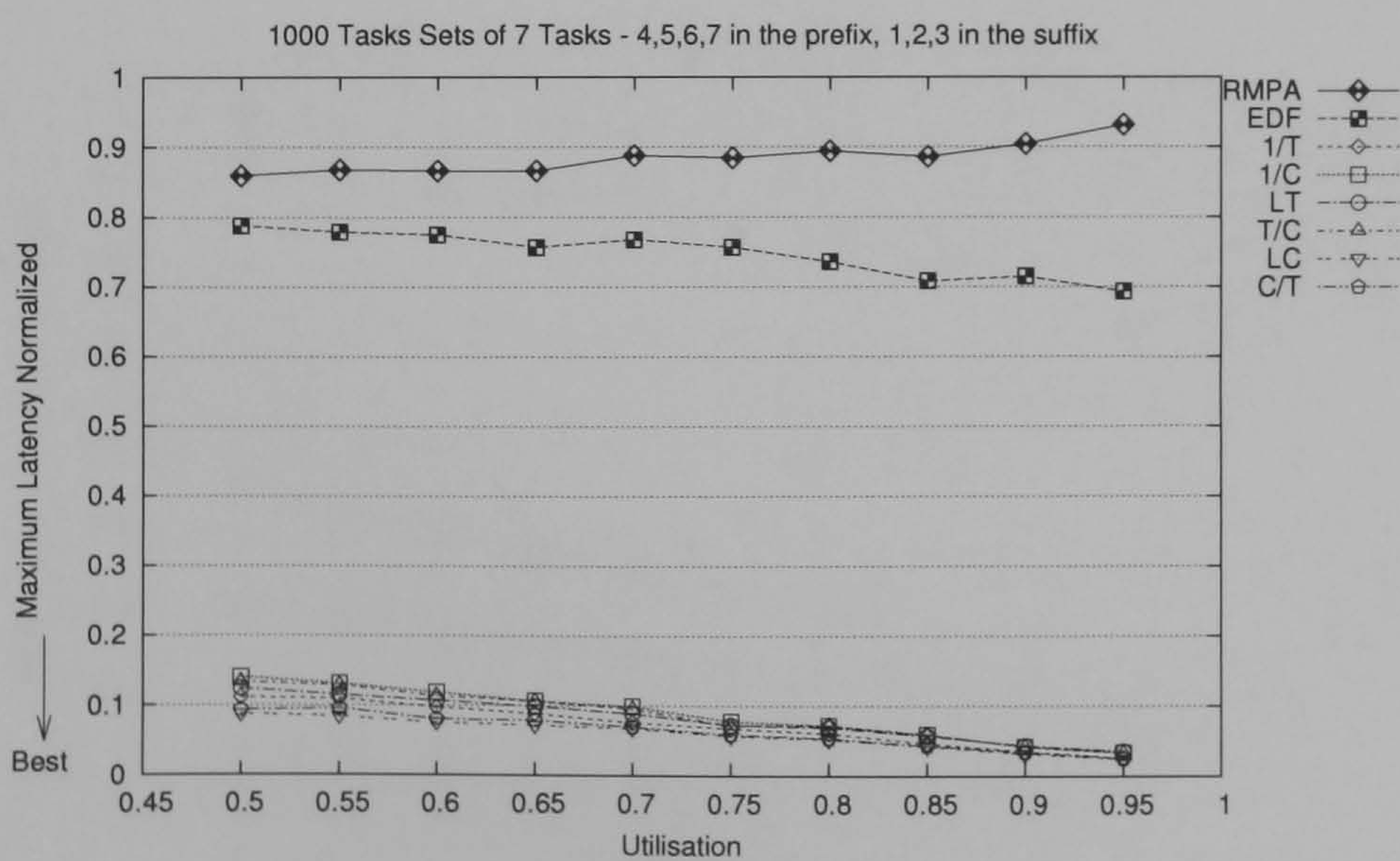
Figure 7.8: Relative Output Jitter under different heuristics. Assigning priorities by shorter deadlines is by far the best option.



(a) Among the heuristics, LT and T/C show the best performance.

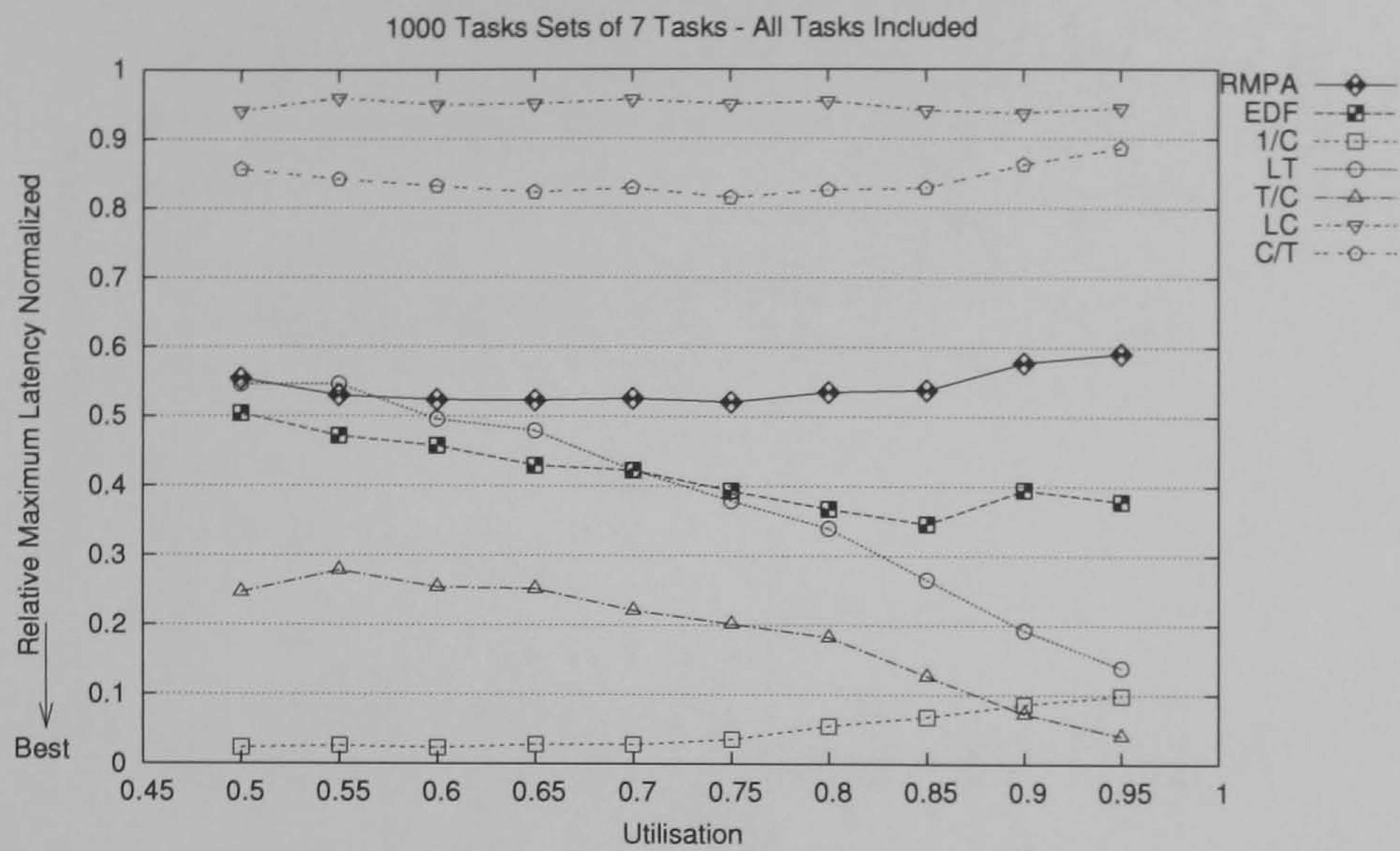


(b) The best is LC

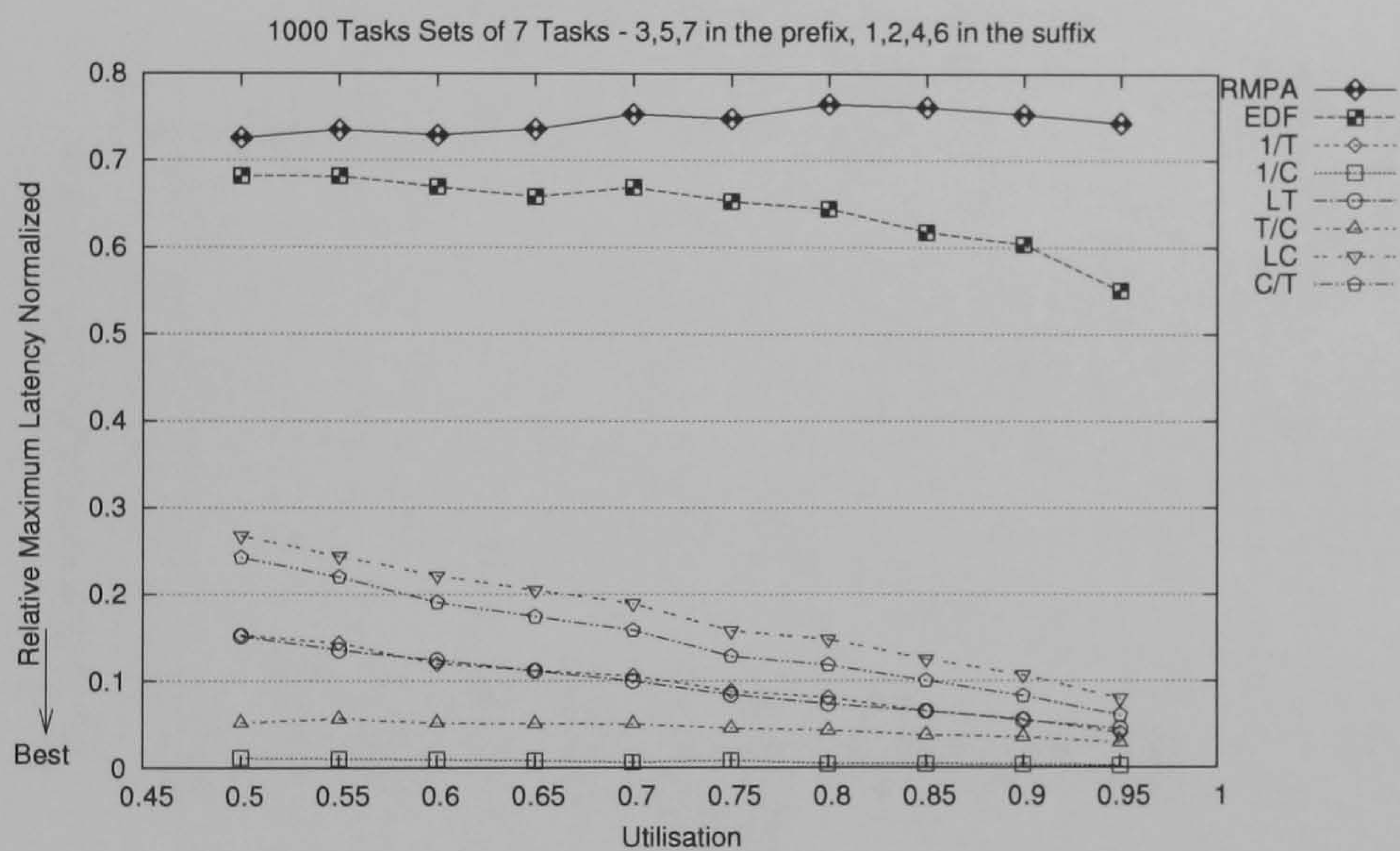


(c) The best is LC

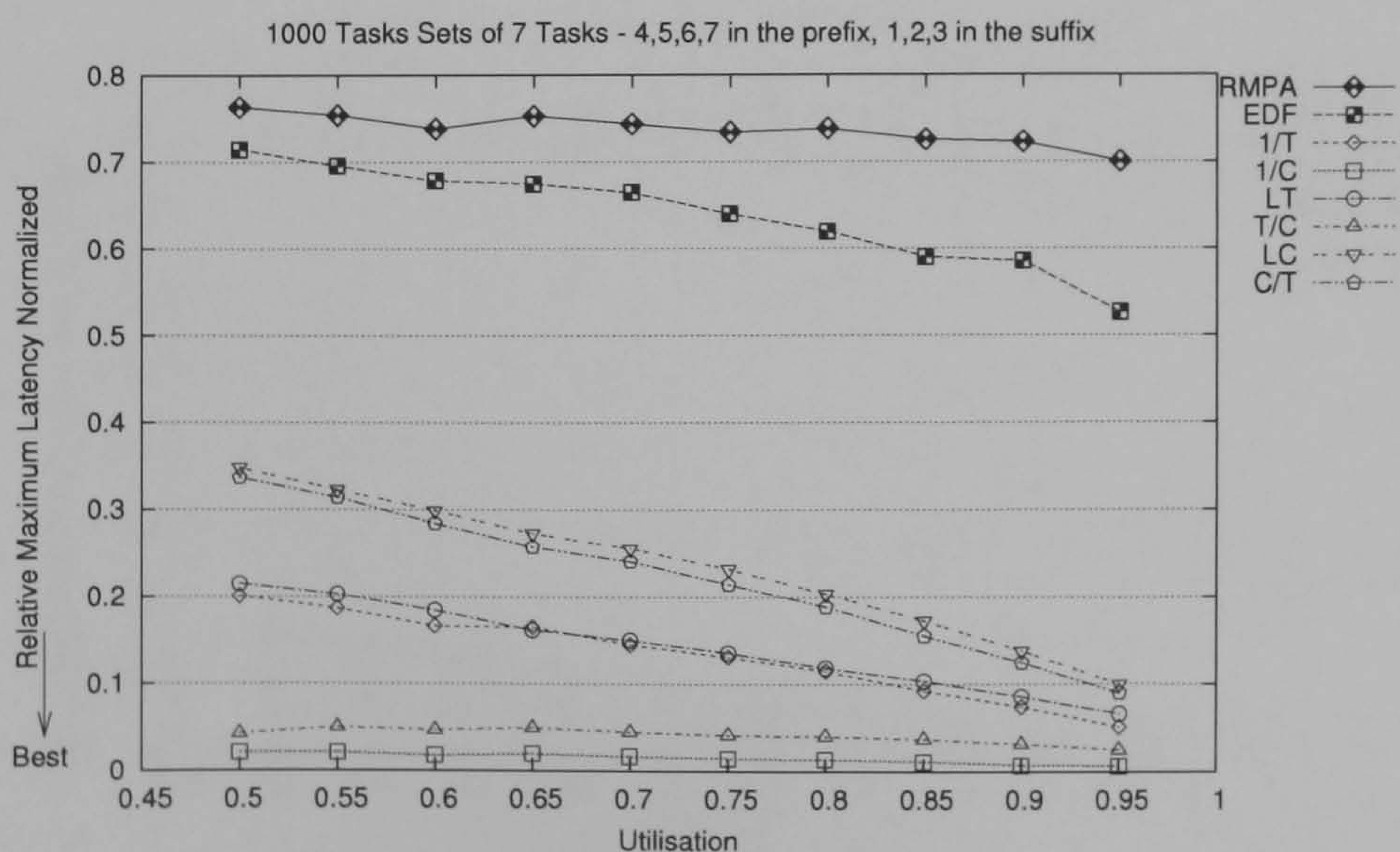
Figure 7.9: Maximum Latency under different priority assignments. It seems that LT, T/C and LC are the best heuristics



(a) In general, $1/C$ and T/C show the best performance

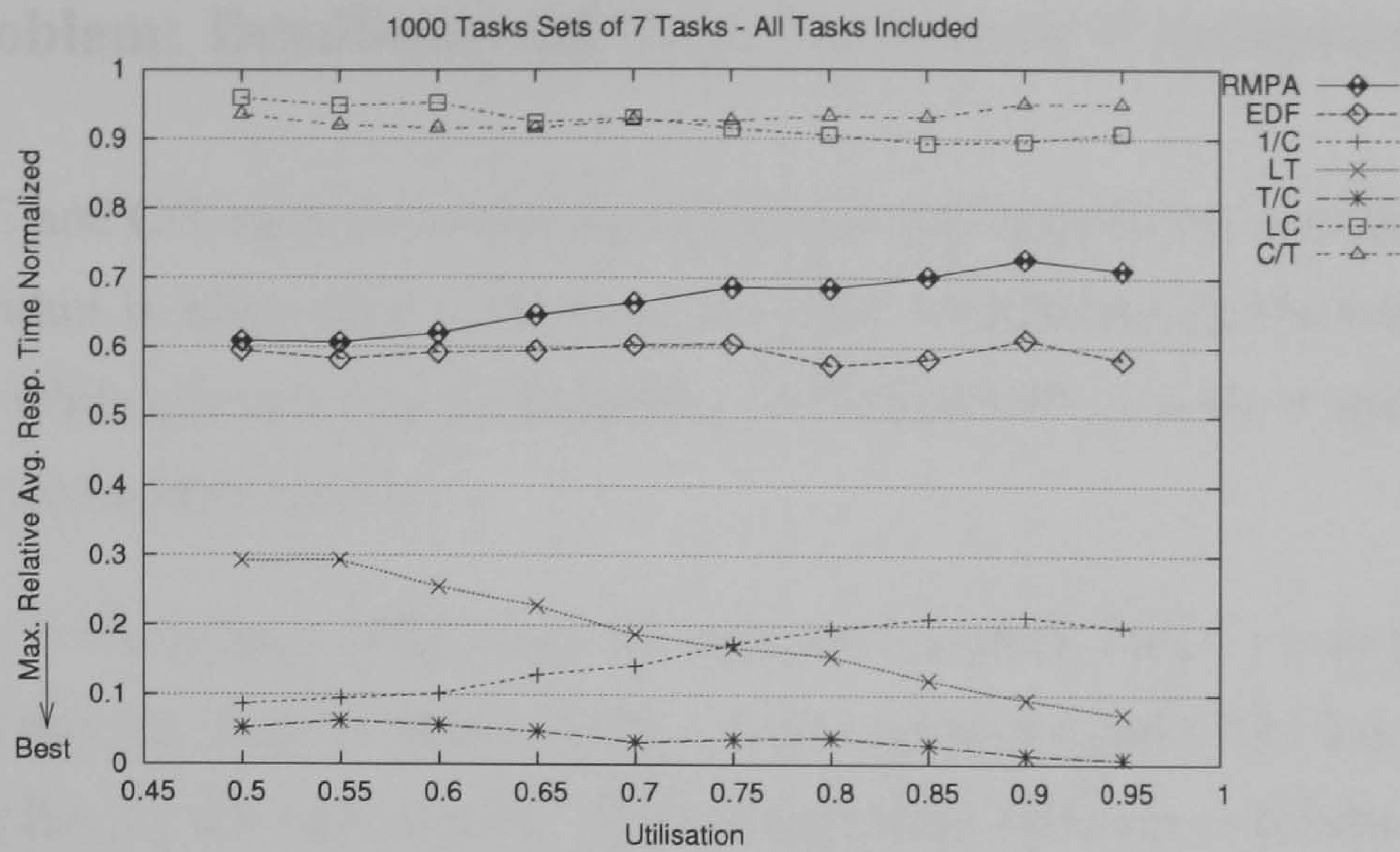


(b) The best is $1/C$

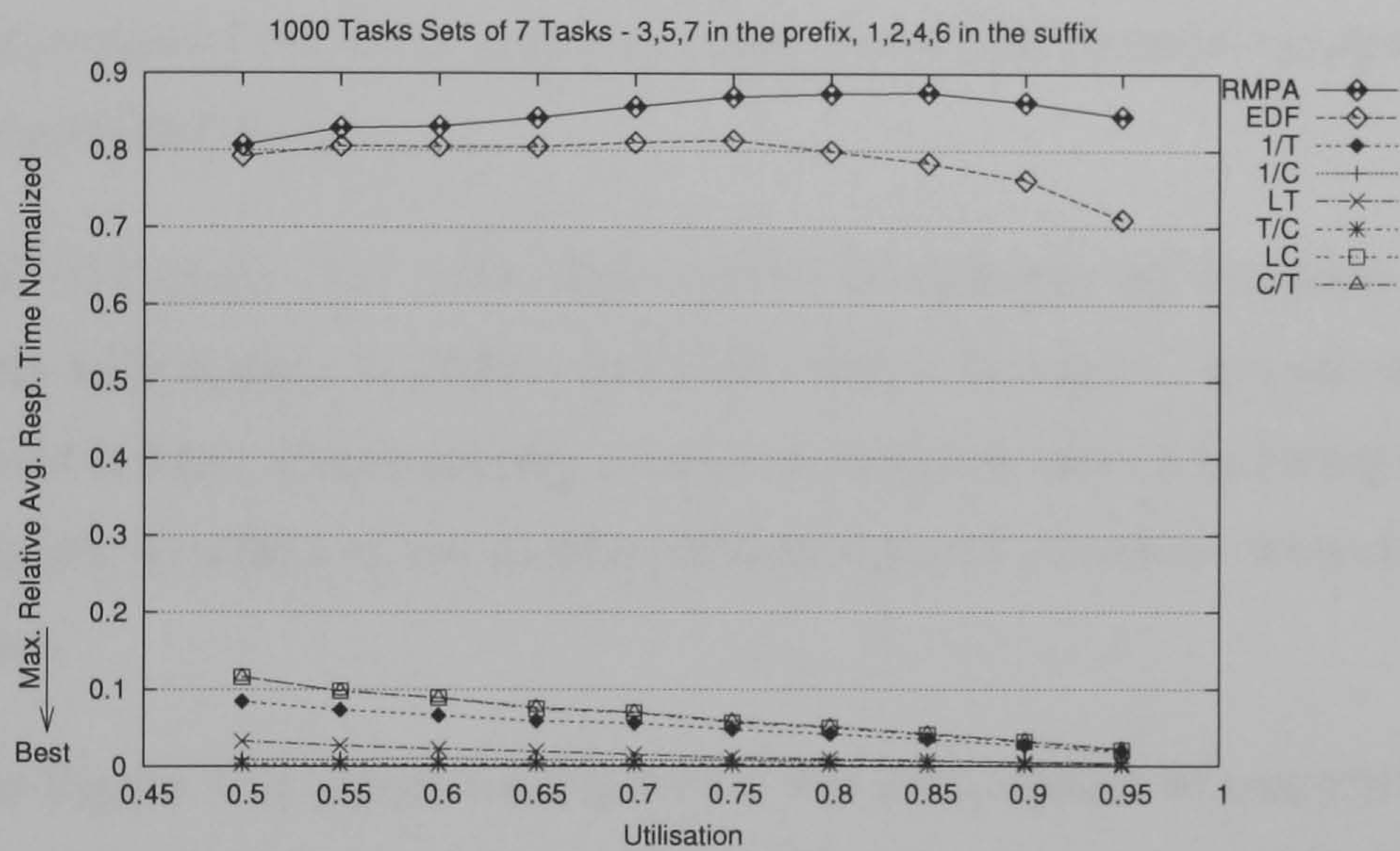


(c) The best is $1/C$

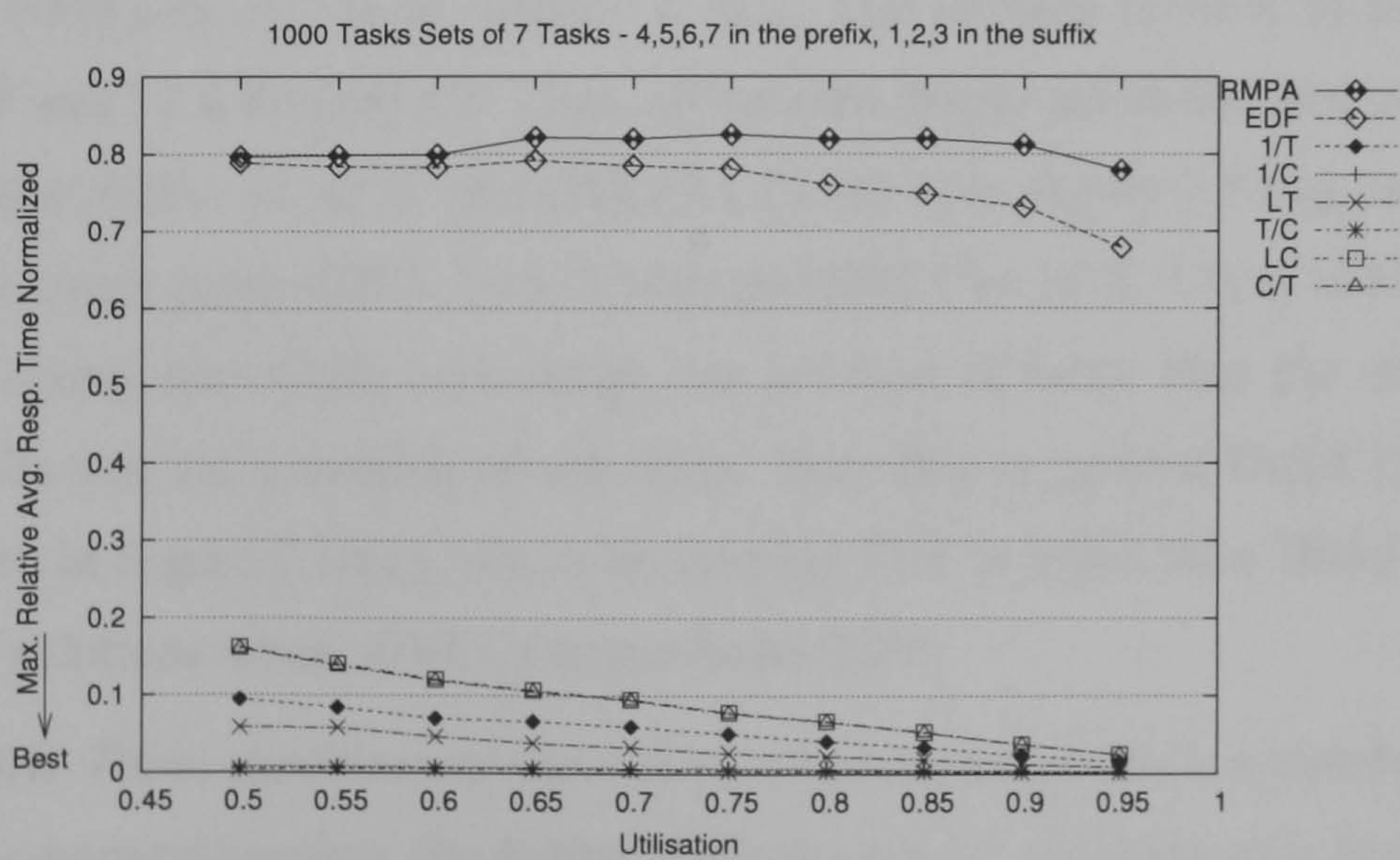
Figure 7.10: Relative Maximum Latency under different priority assignments. In general, among the heuristic, the best is $1/C$.



(a) T/C shows excellent followed by $1/C$ and LT



(b) The best are T/C and $1/C$



(c) The best are T/C and $1/C$

Figure 7.11: Maximum Relative Average Response-Time under different priority assignments. Among the heuristics, the best ones are T/C and $1/C$.

7.4.1 Problem: Deadlines and Total Number of Preemptions

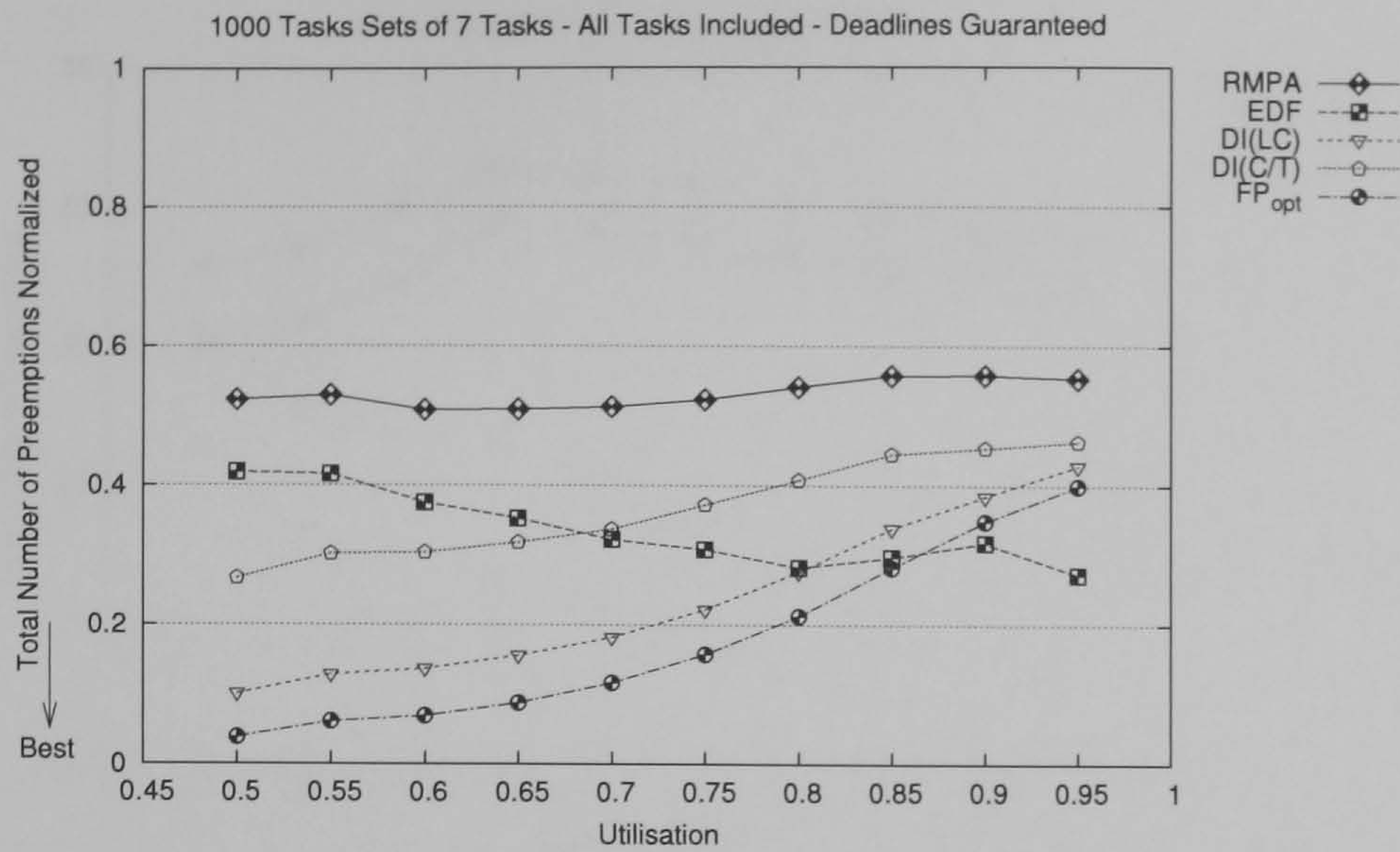
Using the LC and C/T rules for assigning importance our experiment compares how good the DI algorithm is when used with these two rules for minimising the total number of preemptions while guaranteeing the deadlines. In the plots, FP_{opt} is the $\mathcal{P}$*-optimal* solution computed by exhaustive search.

In Figure 7.12(a) (page 140), when all tasks are included, DI(LC) is in general better than EDF. In fact, the distance between the $\mathcal{P}$*-optimal* solution and DI(LC) is less than 7%. In Figure 7.12(b, c), for task subsets, DI with both rules achieves excellent results. This is because DI tries to maintain intact the priorities of higher importance tasks, affecting mainly the priorities of the lower important ones. Note how close of the optimal solution both DI(LC) and DI(C/T) are.

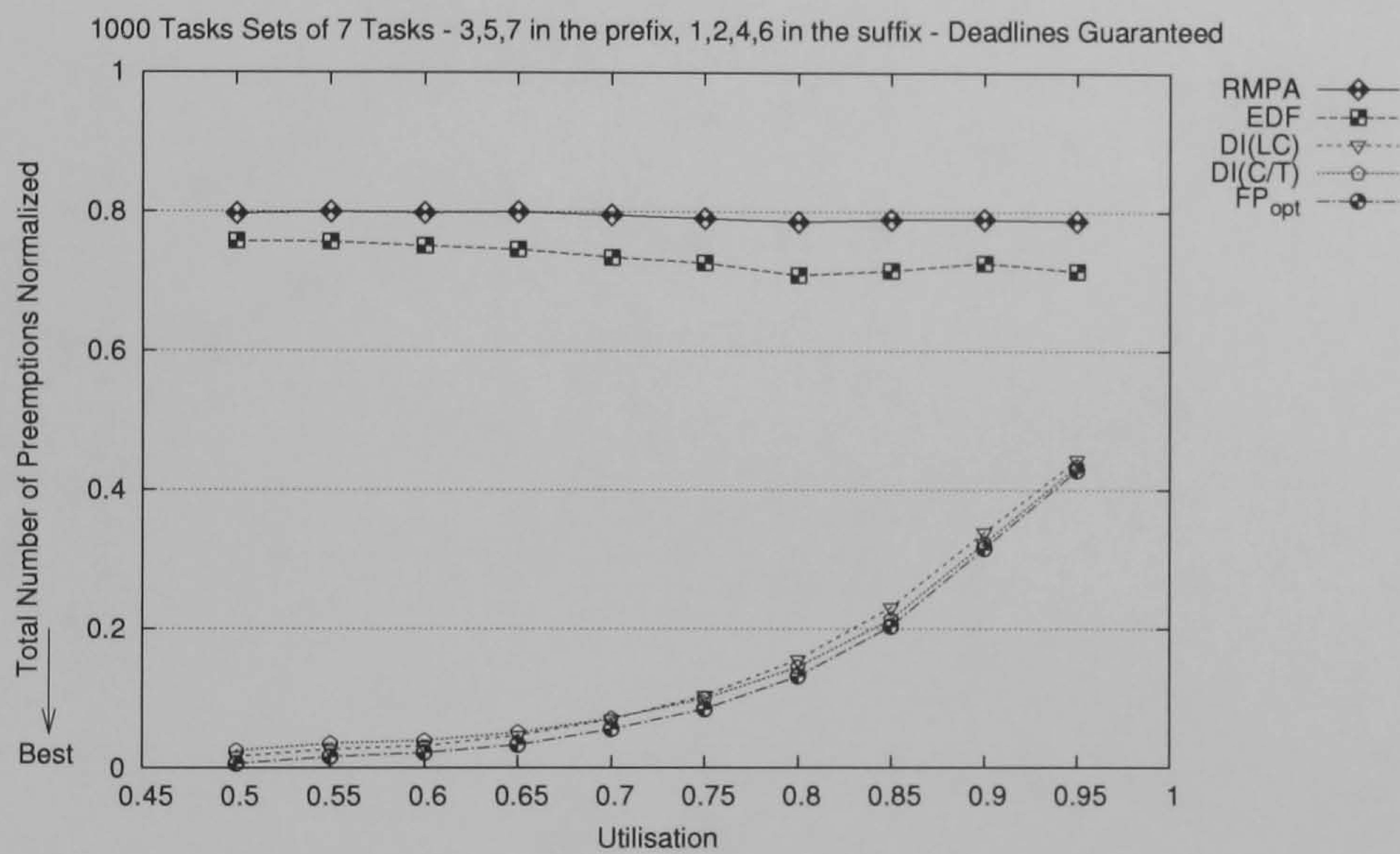
In Figure 7.13 (page 141), when the number of tasks per set increases, DI(LC) improves notably with respect to RMPA and EDF. This is because C in each task decreases since the period is fixed. Consequently, a task can complete sooner reducing the chance of being preempted. In addition, the number of task releases decreases when the number of tasks increases.

Finally, in Figure 7.14 points corresponds to the total number of preemptions obtained when a task set is scheduled by either EDF or DI(LC). For instance, Figure 7.14(a) corresponds to 1000 sets of 7 tasks with $U = 0.5$. The average number of preemptions is 42.3 for EDF and 32.4 for DI(LC). Thus, all the data displayed in this plot corresponds to the single point EDF= $(7, 42.3)$ and DI(LC)= $(7, 32.4)$ in Figure 7.13(a), and also corresponds to the single point EDF= $(0.5, 0.418)$ and DI(LC)= $(0.5, 0.101)$ in Figure 7.12(a). Figure 7.14 shows that while on average one solution is better than the other one, it is not necessarily true for a number of solutions. Note that in general DI(LC) is better than EDF and even in Figure 7.14(c), where on average EDF is better than DI(LC), there exist a number of solutions where DI(LC) outperforms EDF.

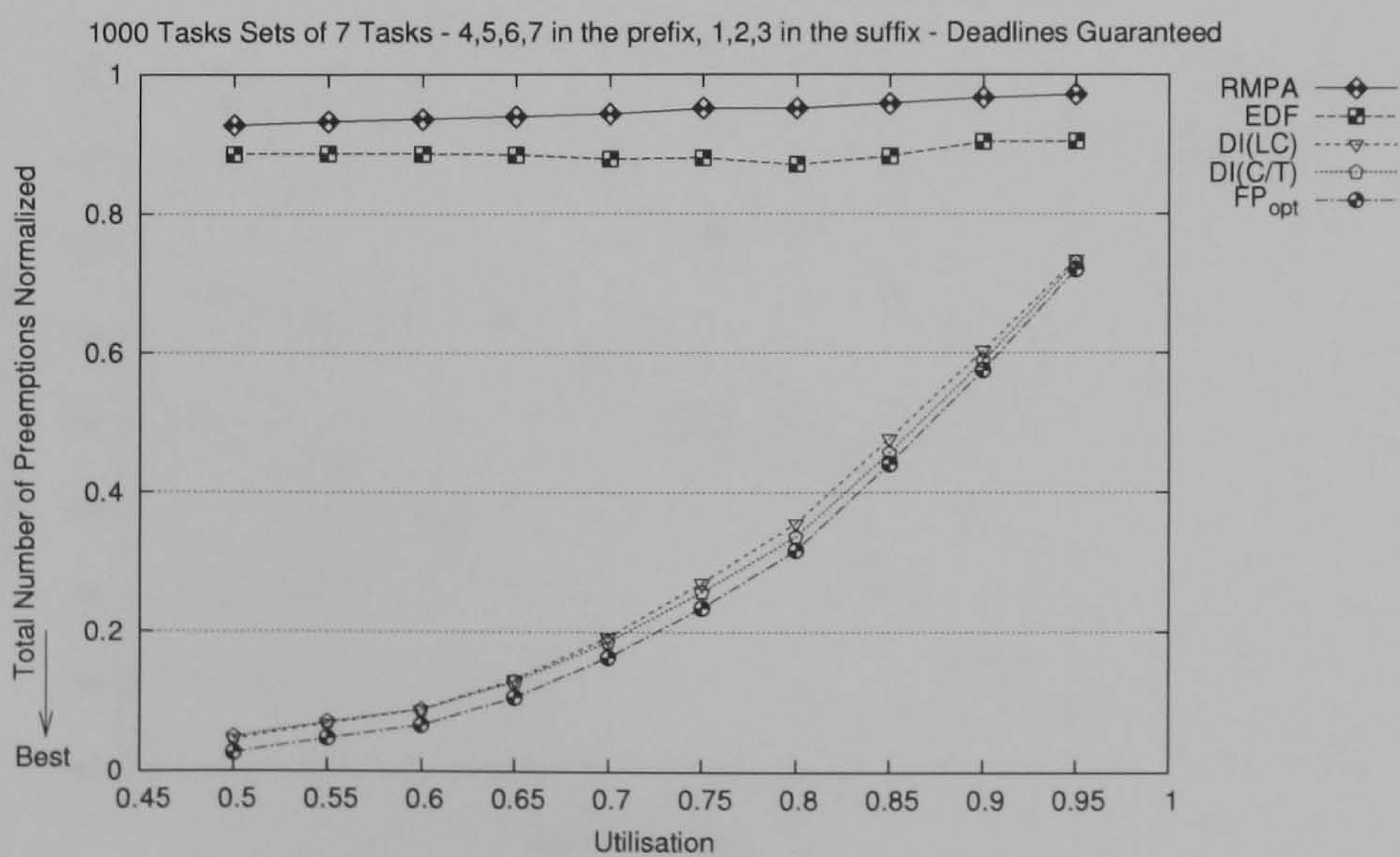
Remark 7.4.1. *These experiments show that assigning importance according to the rule "the largest computation time, the higher the importance" and finding a feasible solution with the DI algorithm provides a near-optimal solution for the scheduling problem of finding a feasible fixed-priority assignment that minimises the total number of preemptions.*



(a) By far, DI(LC) outperforms RMPA. It also is better than EDF for $U \leq 0.8$

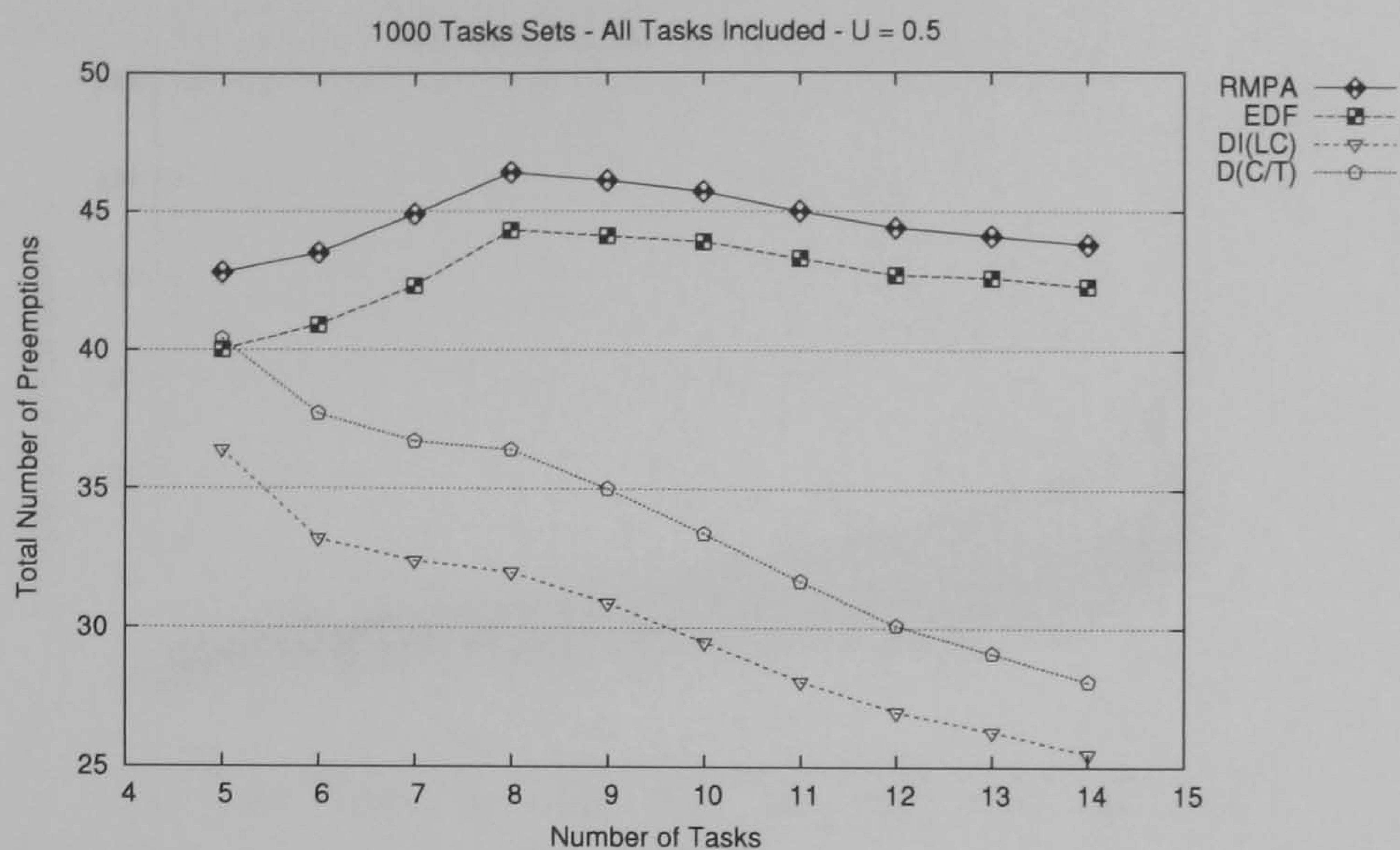


(b) Both DI(LC) and DI(C/T) are very close of the optimal solution

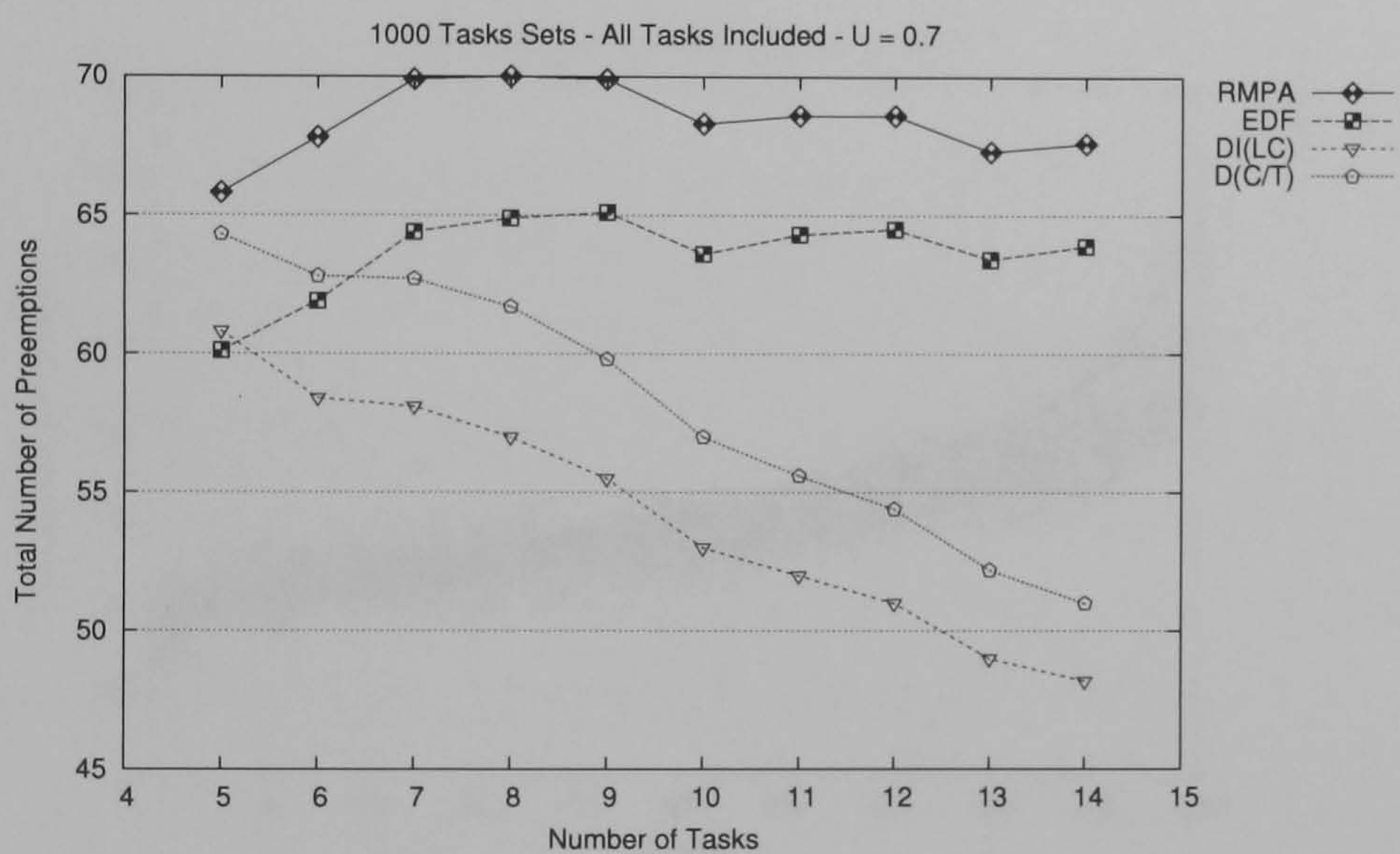


(c) Both DI(LC) and DI(C/T) are very close of the optimal solution

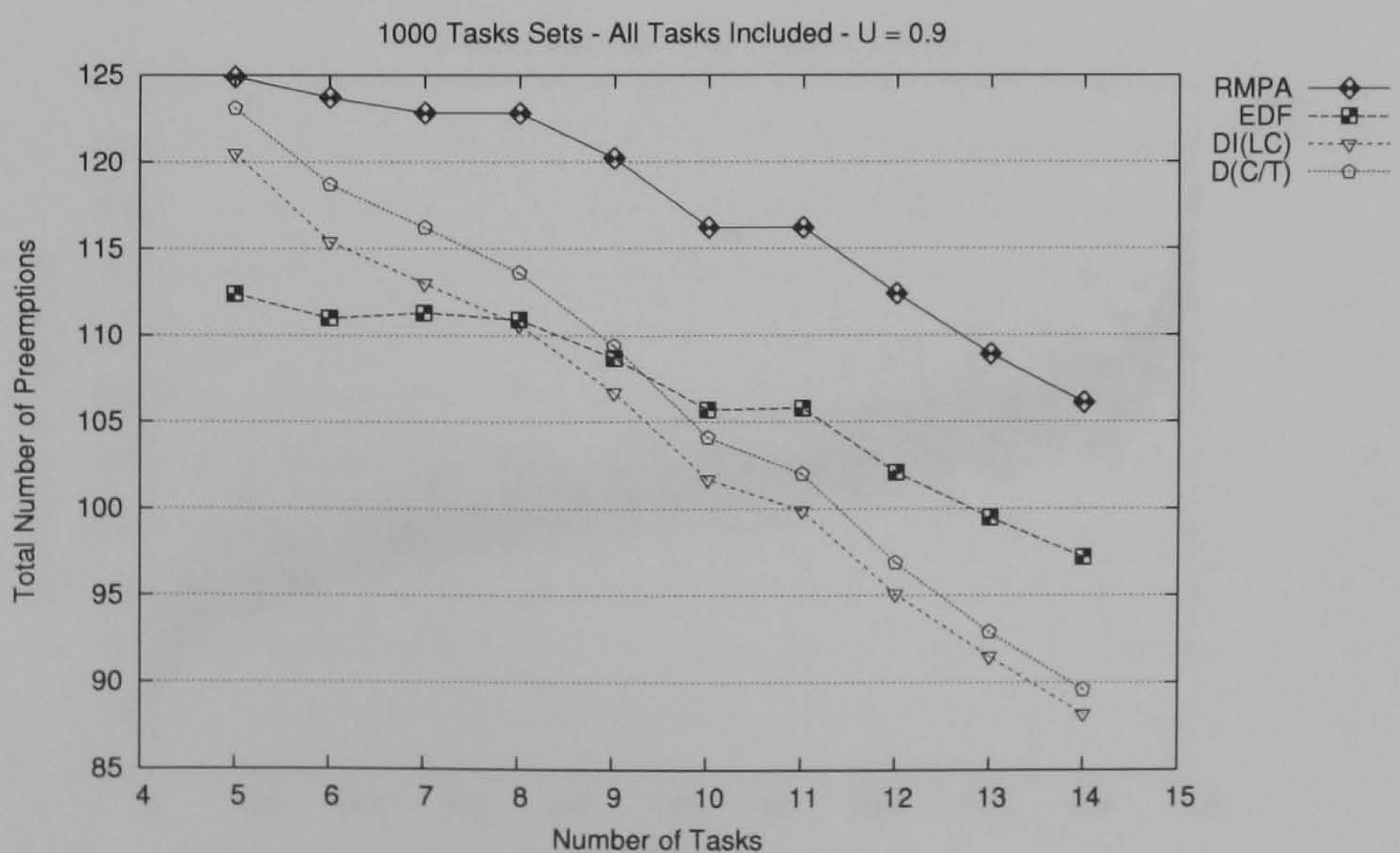
Figure 7.12: PLOTS A TYPE. Guaranteeing Deadlines and Minimizing the Total Number of Preemptions. $\mathsf{FP}_{\mathsf{opt}}$ computed by exhaustive search is the best. In general, *DI(LC)* is the best tractable solution excepting when $U > 0.8$ and the metric includes all tasks. In this case EDF is better



(a) DI(LC) outperforms both RMPA and EDF

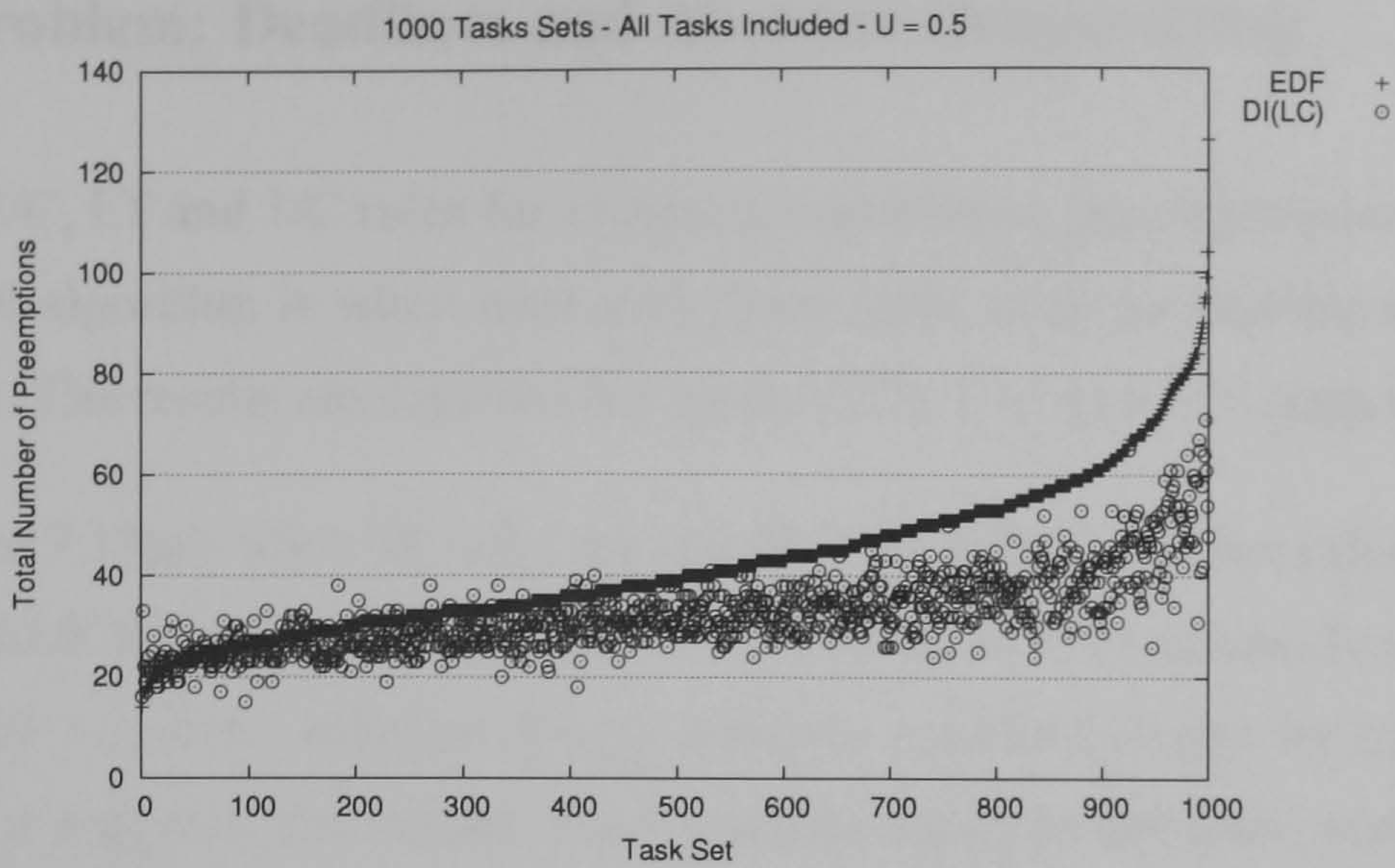


(b) DI(LC) outperforms both RMPA and EDF

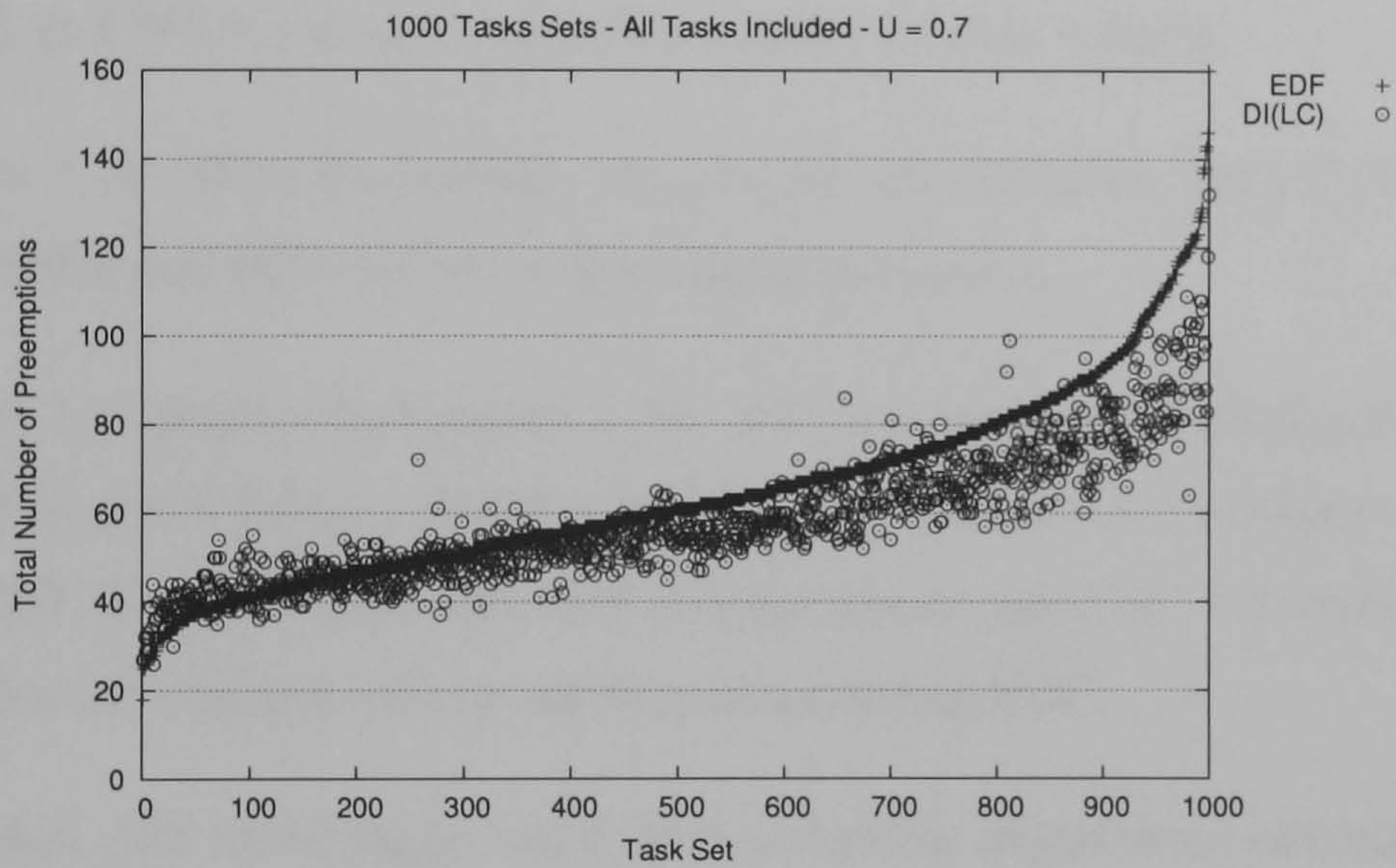


(c) DI(LC) outperforms both RMPA and EDF

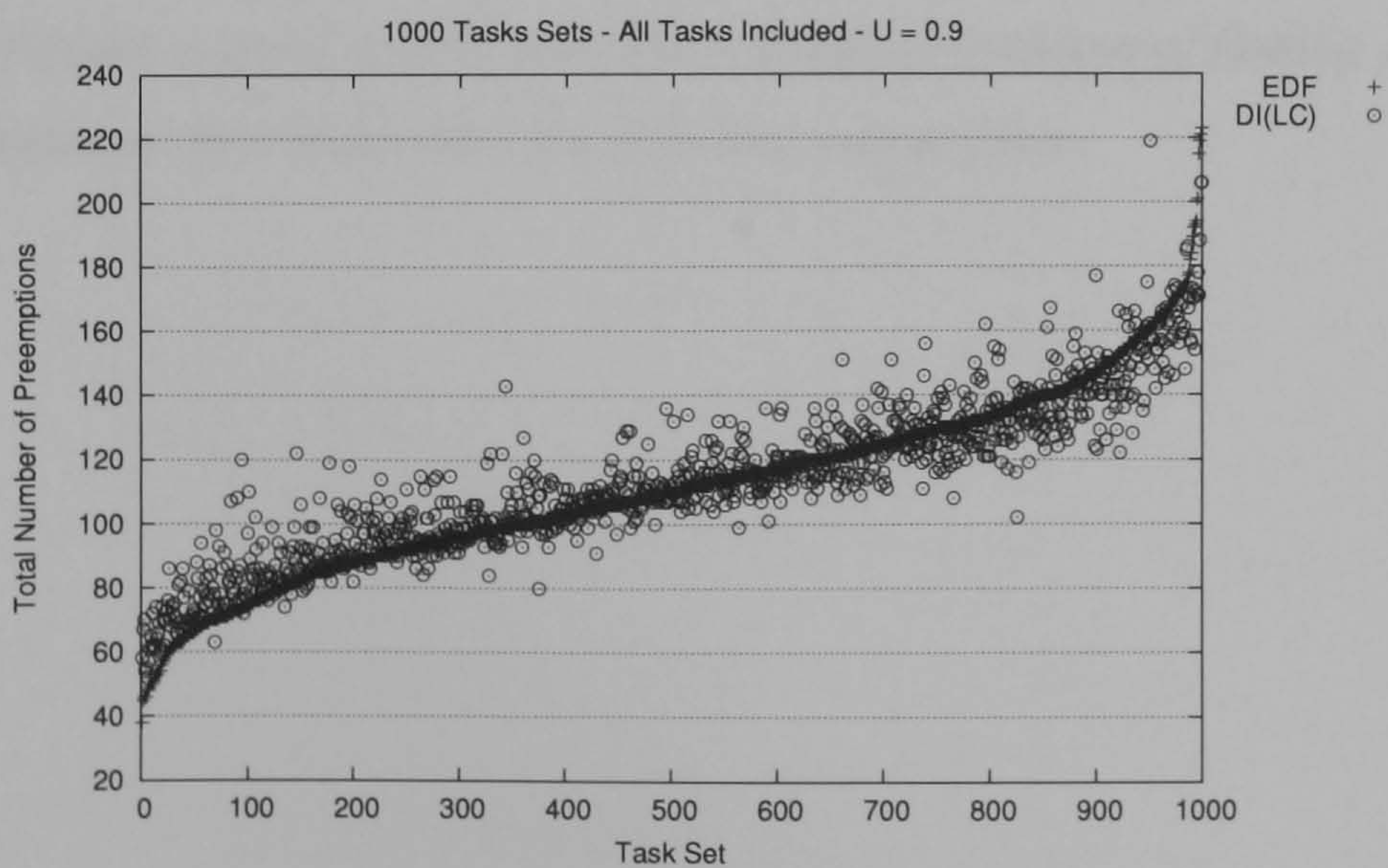
Figure 7.13: PLOTS B TYPE. Guaranteeing Deadlines and Minimizing the Total Number of Preemptions for different task sets



(a) On average the number of preemptions is 42.3 for EDF and 32.4 for DI(LC)



(b) On average the number of preemptions is 64.4 for EDF and 58.1 for DI(LC)



(c) On average the number of preemptions is 111.3 for EDF and 113 for DI(LC)

Figure 7.14: PLOTS C TYPE. Comparing DI(LC) against EDF for the Deadlines and Total Number of Preemptions problem. While on average one outcome is better than the other one, it is not necessarily true for a number of solutions

7.4.2 Problem: Deadlines and Absolute Output Jitter

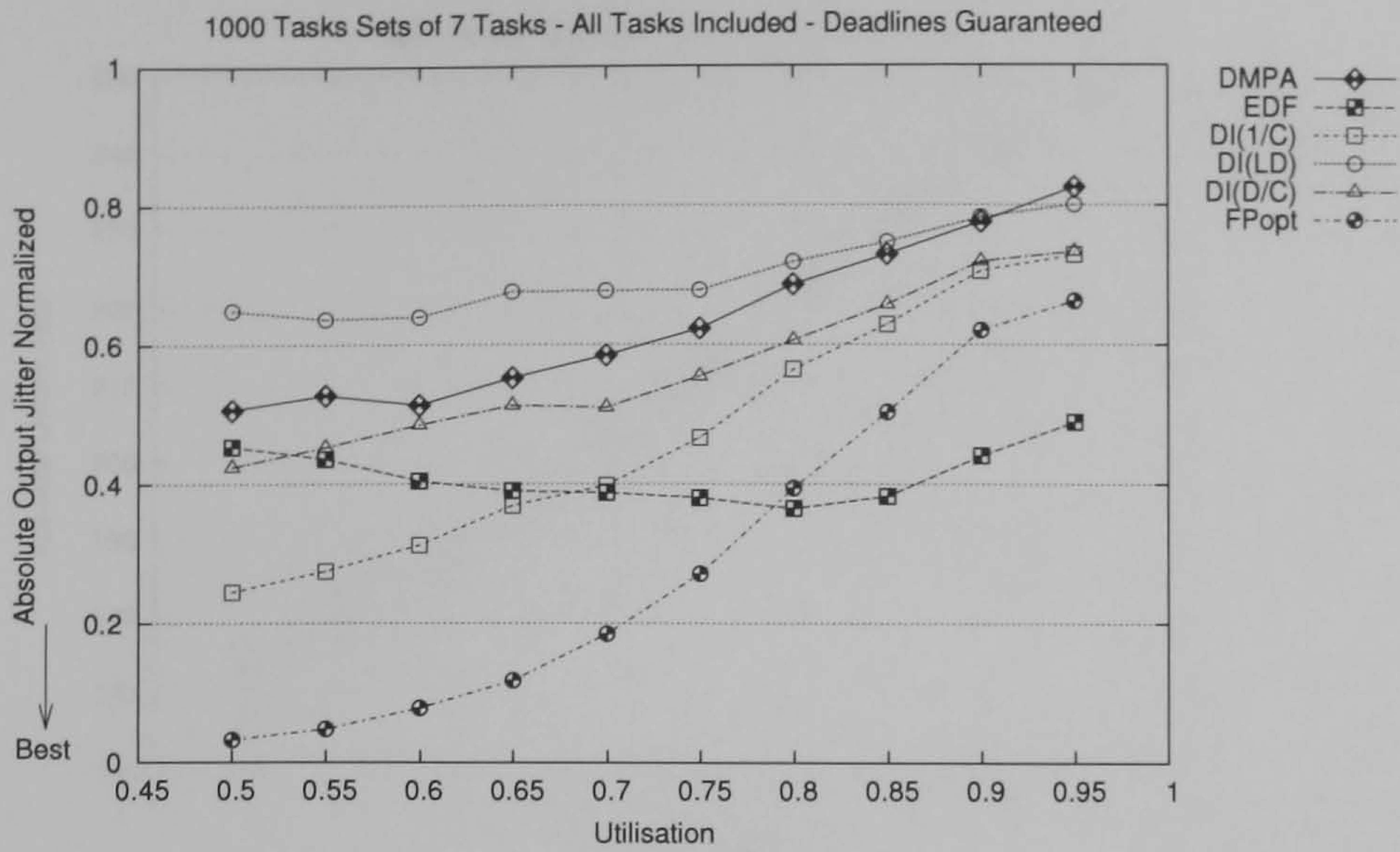
Using the T/C, LT and 1/C rules for assigning importance, our experiment compares how good the DI algorithm is when used with these three rules for minimising the absolute output jitter. The results are depicted in Figures 7.15, 7.16 and 7.17 (pages 144- 146).

In Figure 7.15(a), when all tasks are included DI(1/C) is our best solution performing better than EDF for low utilisation. Both DI(1/C) and DI(T/C) are also better than RMPA. Note that the $\jmath$*-optimal* solution ($\mathsf{FP}_{\mathsf{opt}}$) achieves excellent results for low and medium utilisation; it suggests that further improvements could be obtained with additional research in algorithms for assigning importance. On the other hand, Figure 7.15(b, c) shows that DI(1/C) and D(T/C) produce excellent results for task subsets.

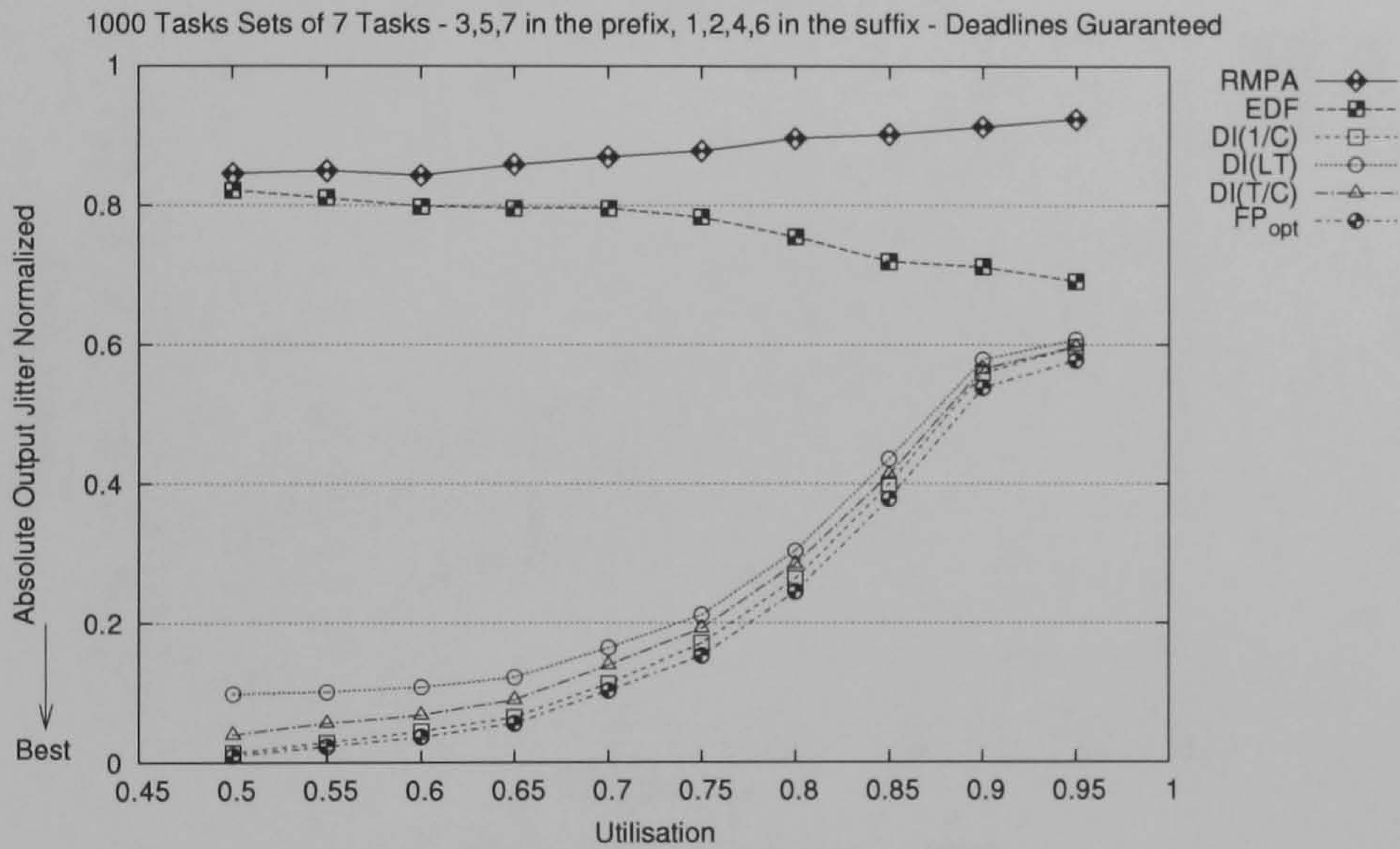
In Figure 7.16, when the number of tasks per set increases, DI(1/C) is notably better than both RMPA and EDF for low and medium utilisation.

In Figure 7.17 points corresponds to the absolute output jitter obtained when a task set is scheduled by either EDF or DI(1/C). It shows how the DI(1/C) solution is scattered with respect to EDF. Note that although EDF is much better than DI(1/C) when $U = 0.9$, there exist a number of solutions where DI(1/C) outperforms EDF.

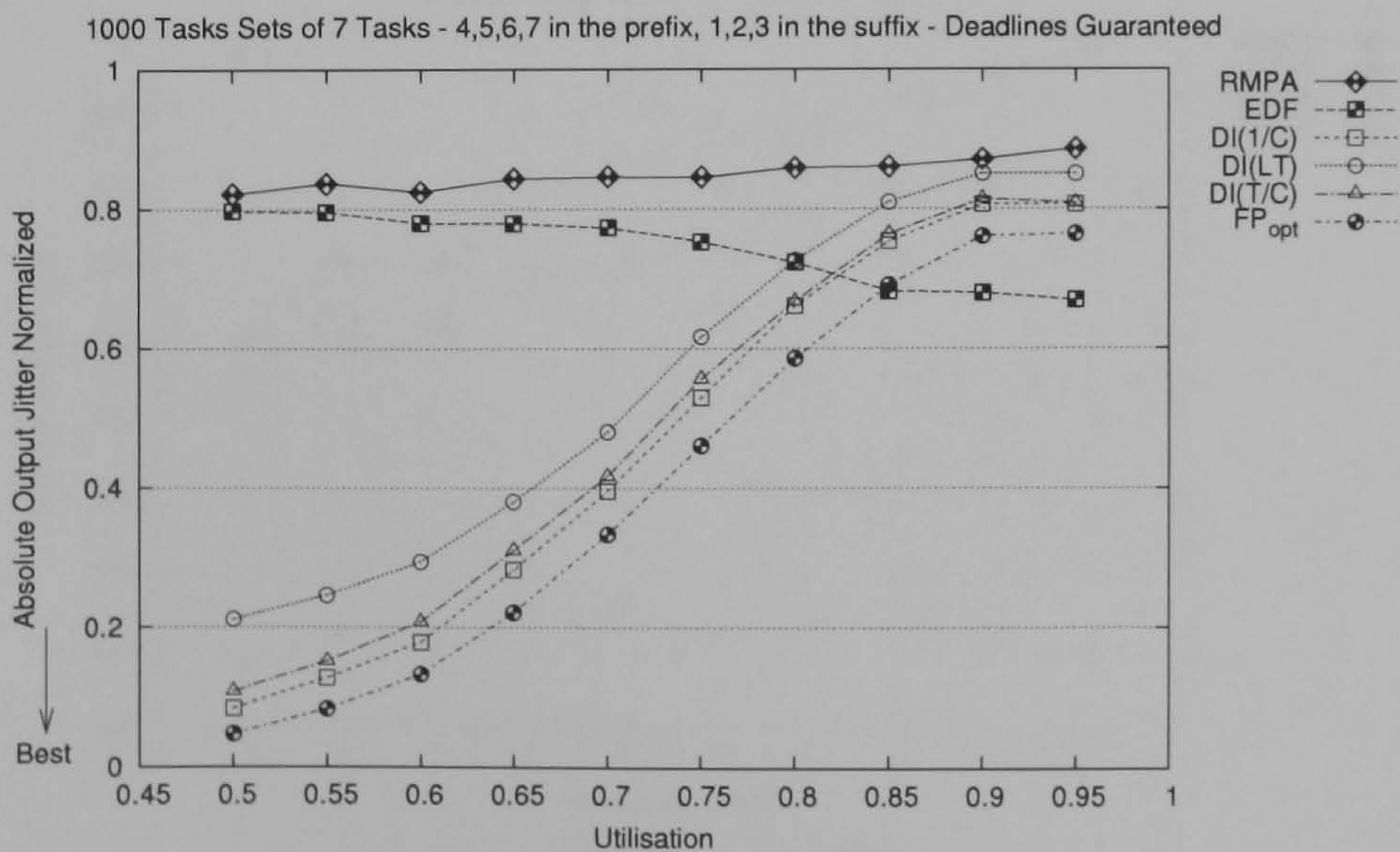
Remark 7.4.2. *The experiments show that assigning importance according to the rule "the larger* $1/C$*, the higher the importance" and finding a feasible solution with the DI algorithm provides a good solution for the scheduling problem of finding a feasible fixed-priority assignment that minimises the absolute output jitter.*



(a) DI(1/C) is our best solution even so it is far from the optimal one. DI(T/C) is also better than RMPA

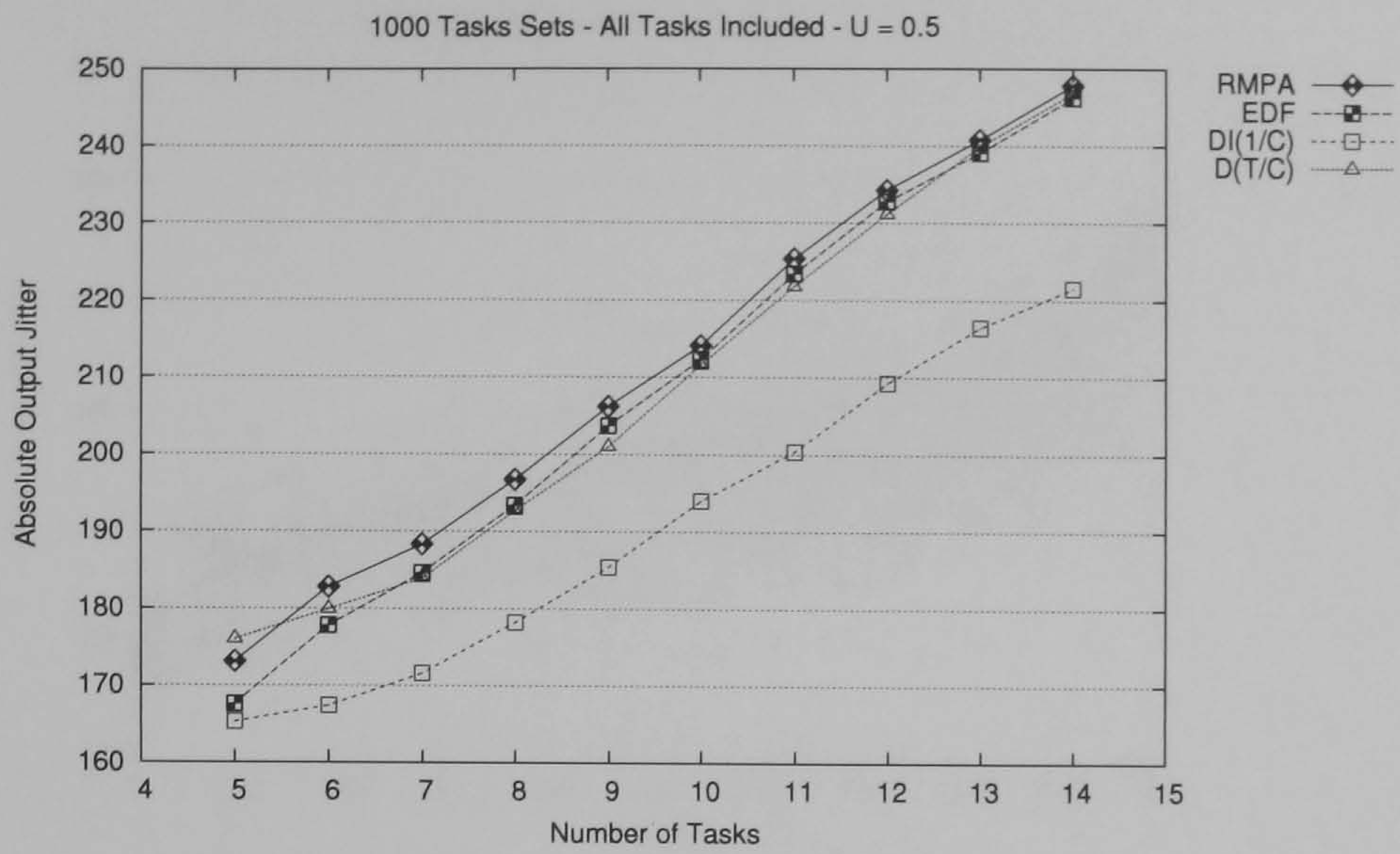


(b) DI(1/C) and D(T/C) produce excellent results

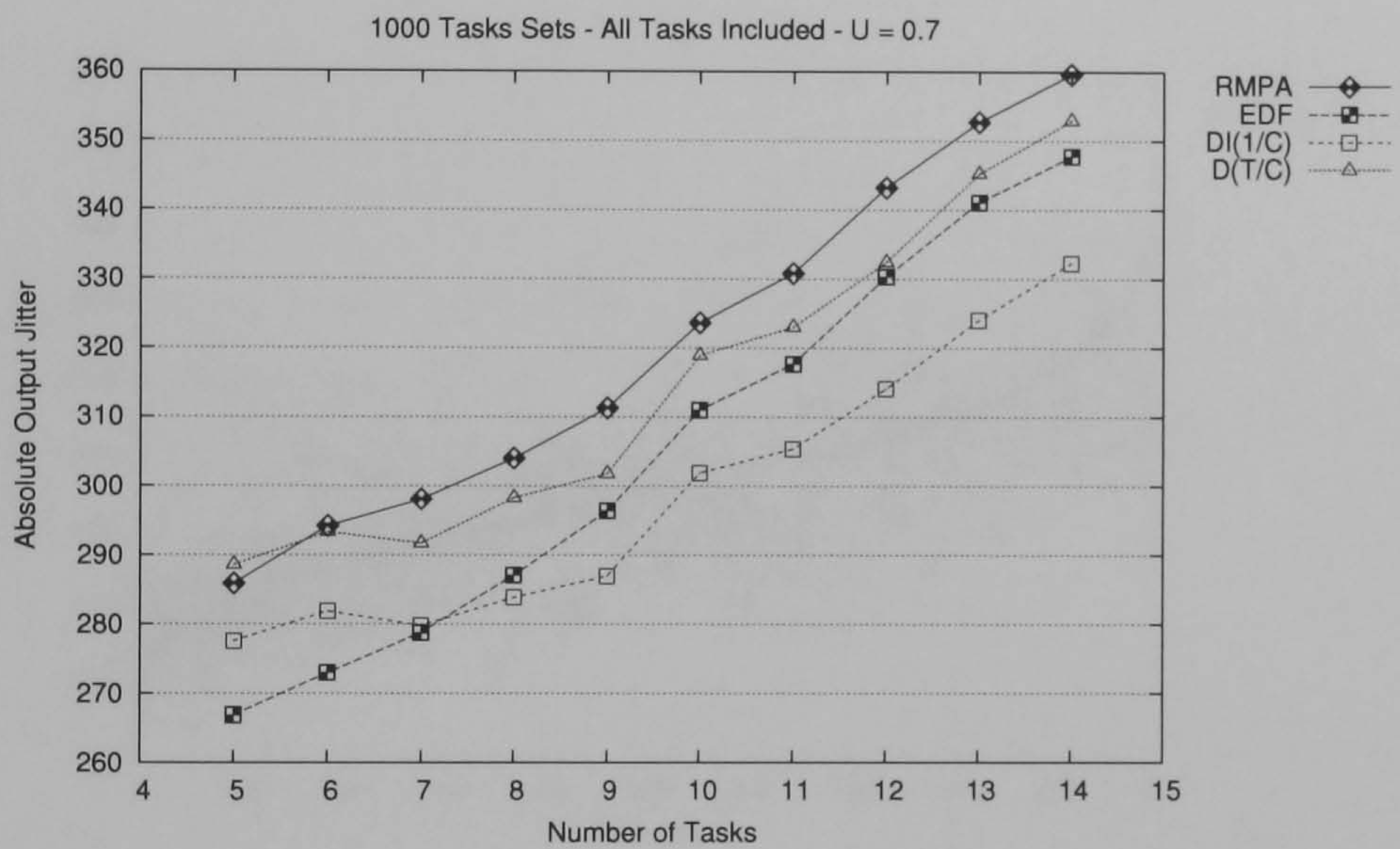


(c) DI(1/C) and D(T/C) produce good results even so they outperforms EDF only for $U < 0.85$

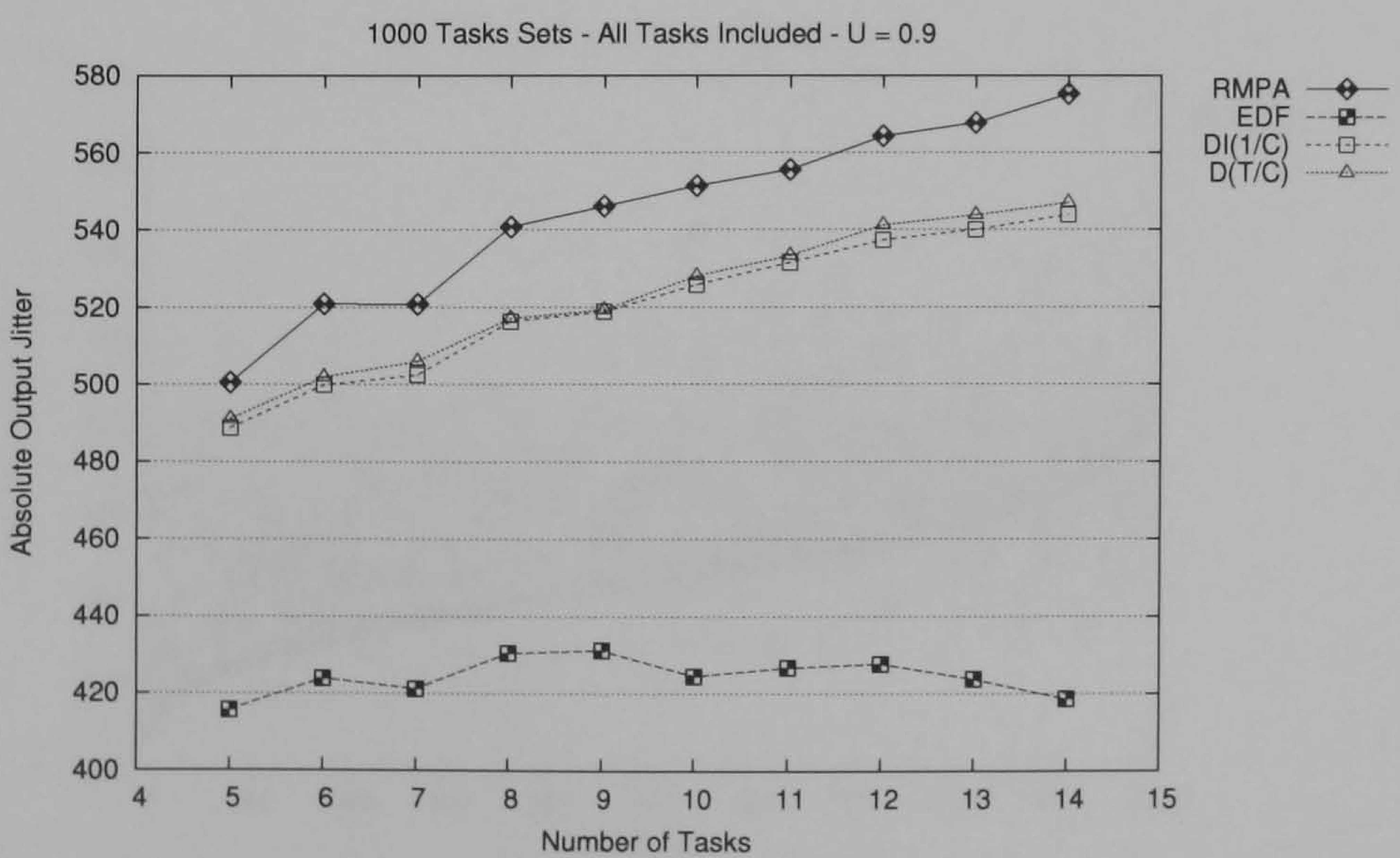
Figure 7.15: PLOTS A TYPE. Guaranteeing Deadlines and Minimising the Absolute Output Jitter. DI(1/C) and D(T/C) produce good results.



(a) DI(1/C) is the best

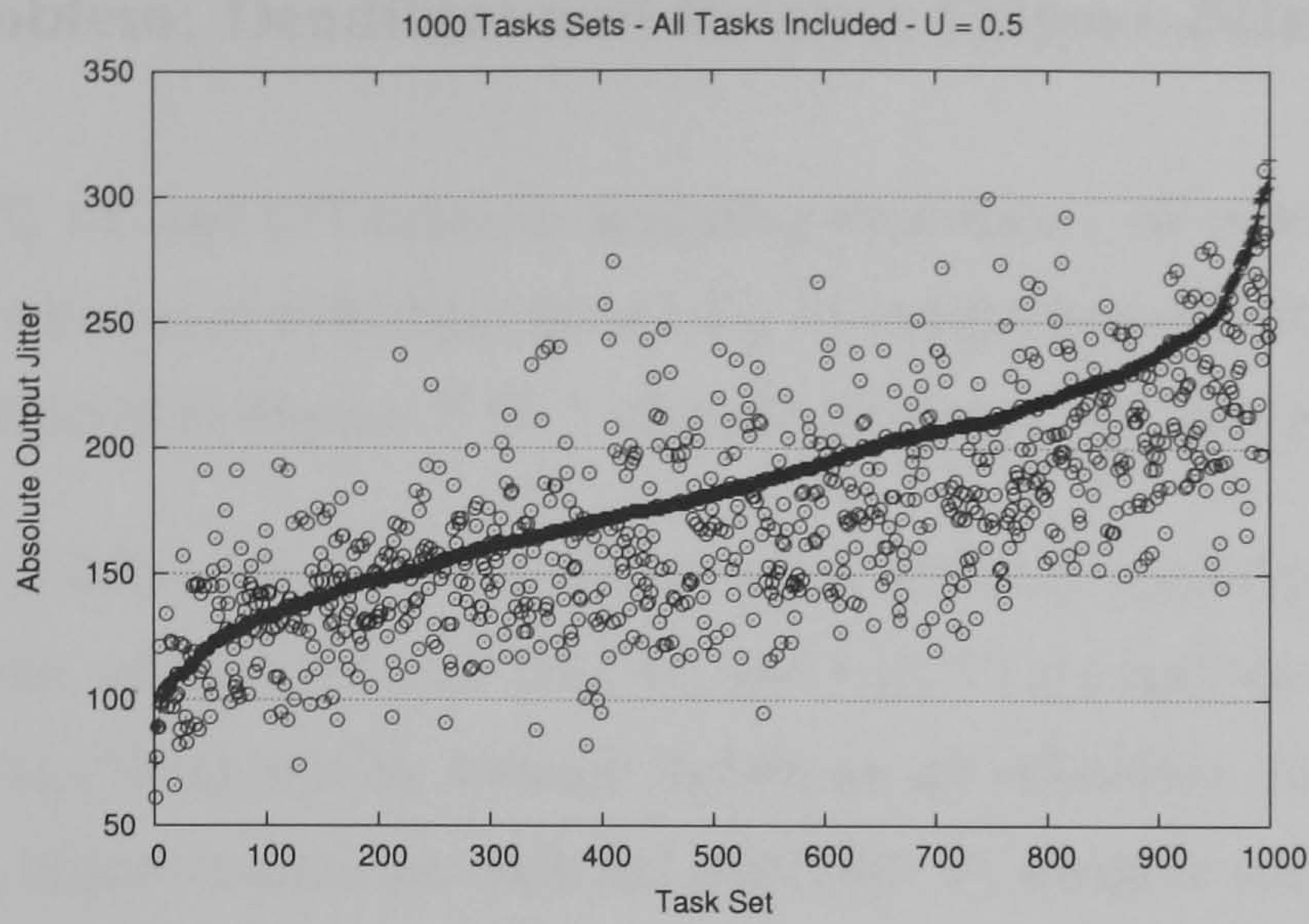


(b) DI(1/C) is better than EDF

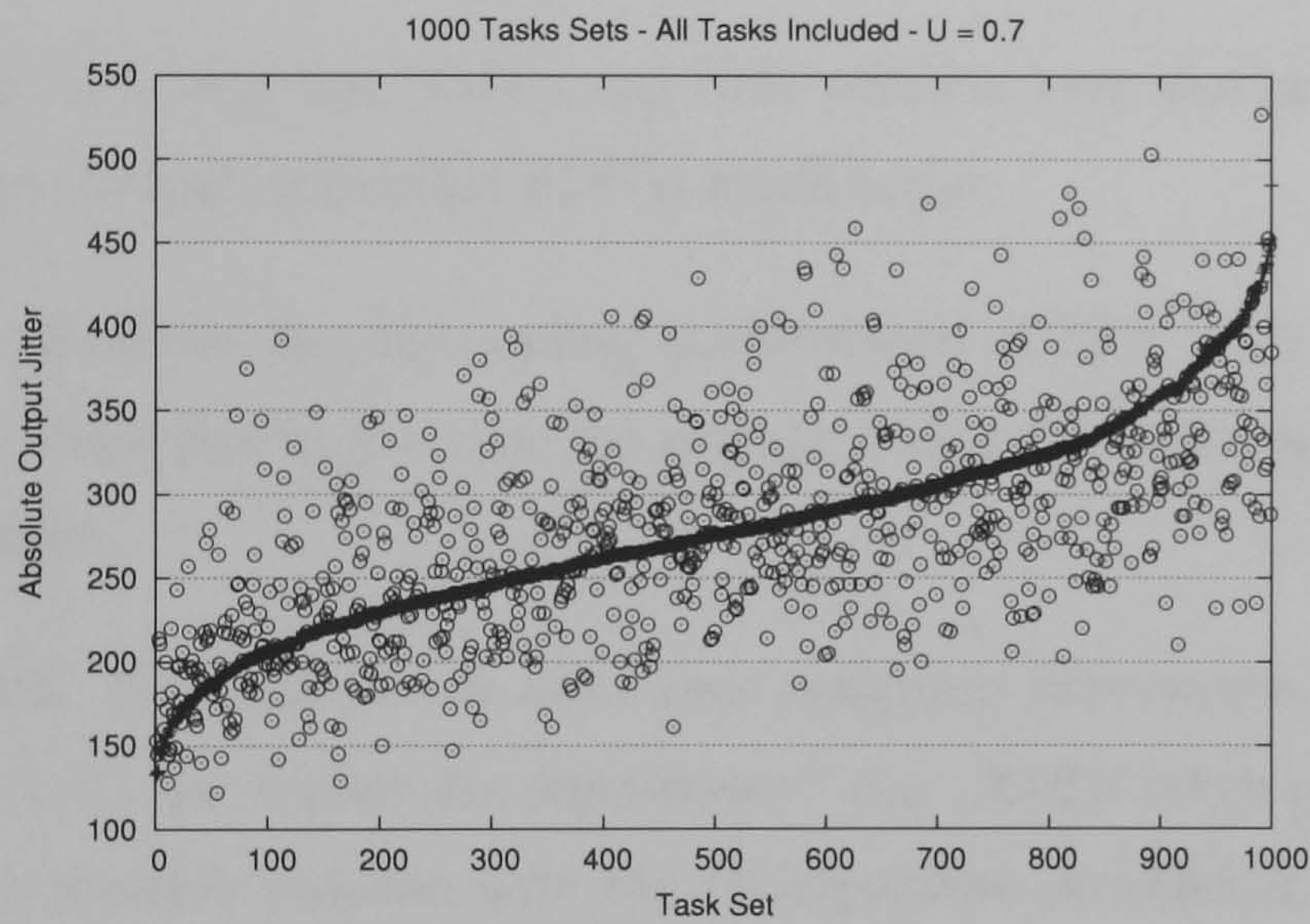


(c) The best is EDF

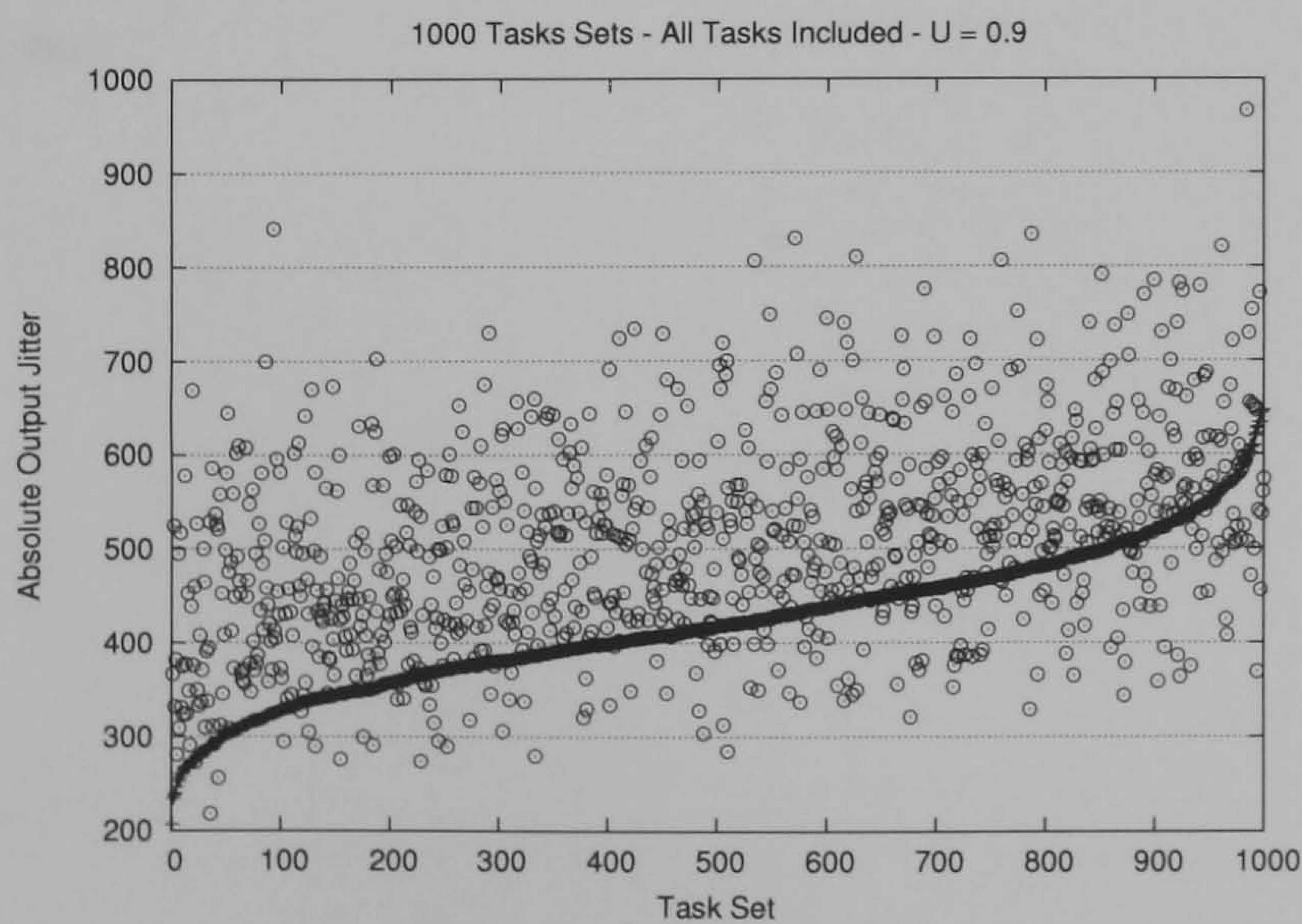
Figure 7.16: PLOTS B TYPE. Guaranteeing Deadlines and Minimising the Absolute Output Jitter



(a) On average the absolute output jitter is 184.6 for EDF and 171.6 for DI(1/C)



(b) On average the absolute output jitter is 278.8 for EDF and 279.8 for DI(1/C)



(c) On average the absolute output jitter is 421.3 for EDF and 502.5 for DI(1/C)

Figure 7.17: PLOTS C TYPE.Comparing DI(1/C) against EDF for the Deadlines and Absolute Output Jitter problem. While on average one outcome is better than the other one, it is not necessarily true for a number of solutions

7.4.3 Problem: Deadlines and Relative Output Jitter

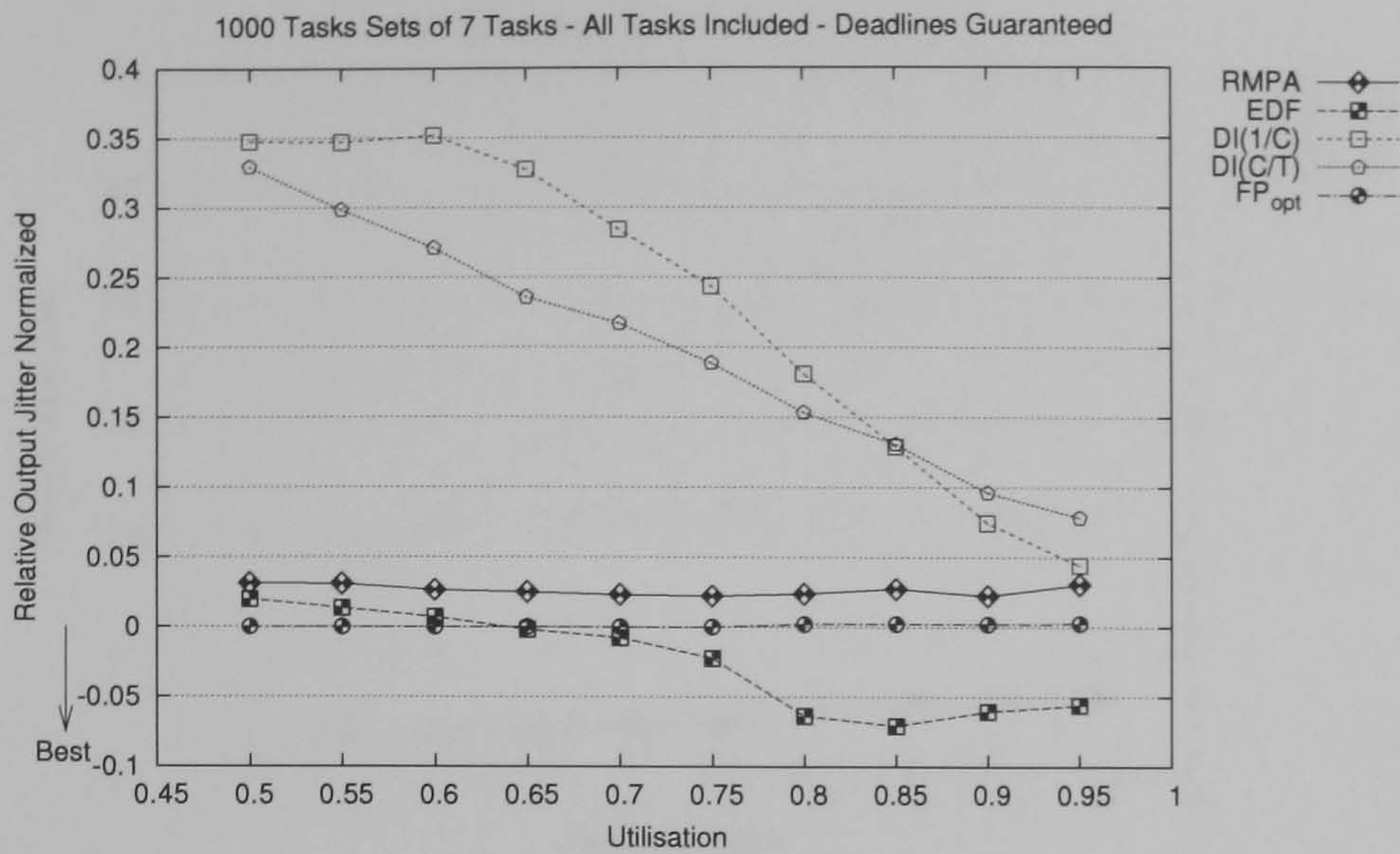
Using the 1/T, 1/C and C/T rules for assigning importance, we compare how good the DI algorithm is when used with these three rules for minimising the relative output jitter. The results are depicted in Figures 7.18, 7.19 and7.20 (pages 148, 149 and 150).

In Figure 7.18(a), when all task are included, RMPA is a good solution and is close to the $\mathcal{J}^{rel}$*-optimal* solution ($\mathsf{FP}_{\mathsf{opt}}$). DI(C/T) and DI(1/C) are bad solutions. Note that EDF outperforms any fixed-priority solution for almost all utilisation. In Figure 7.18(b, c), for task subsets, improvements are obtained with DI(1/T) which is very close to $\mathcal{J}^{rel}$*-optimal* solution.

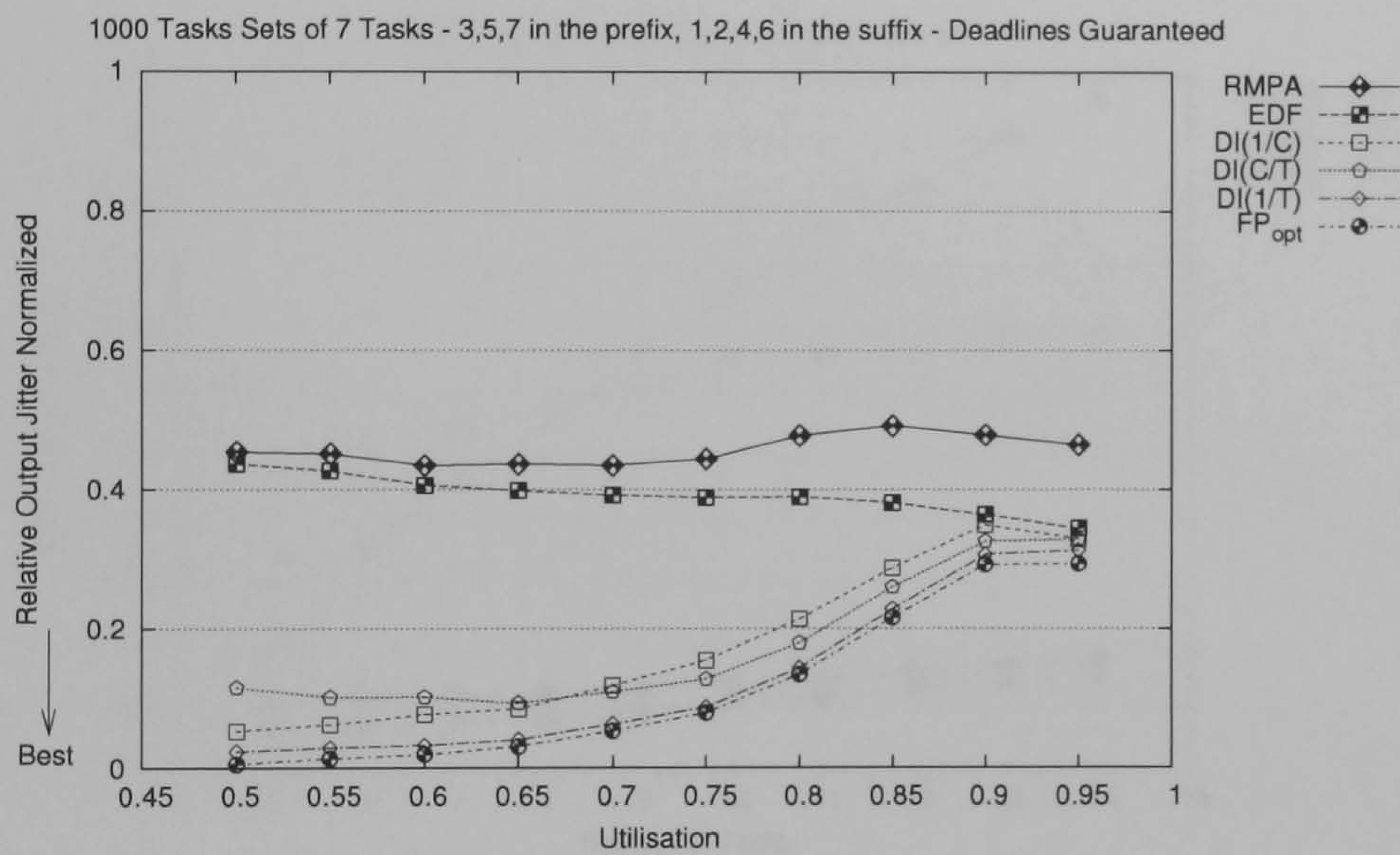
In Figure 7.19, note that RMPA and EDF perform very similar for low and medium utilisation, but for high utilisation EDF is much better.

Figure 7.20 shows the outstanding performance of EDF with respect to RMPA for this problem. Note that in this case the average truly reflects the superiority of EDF with respect to RMPA.

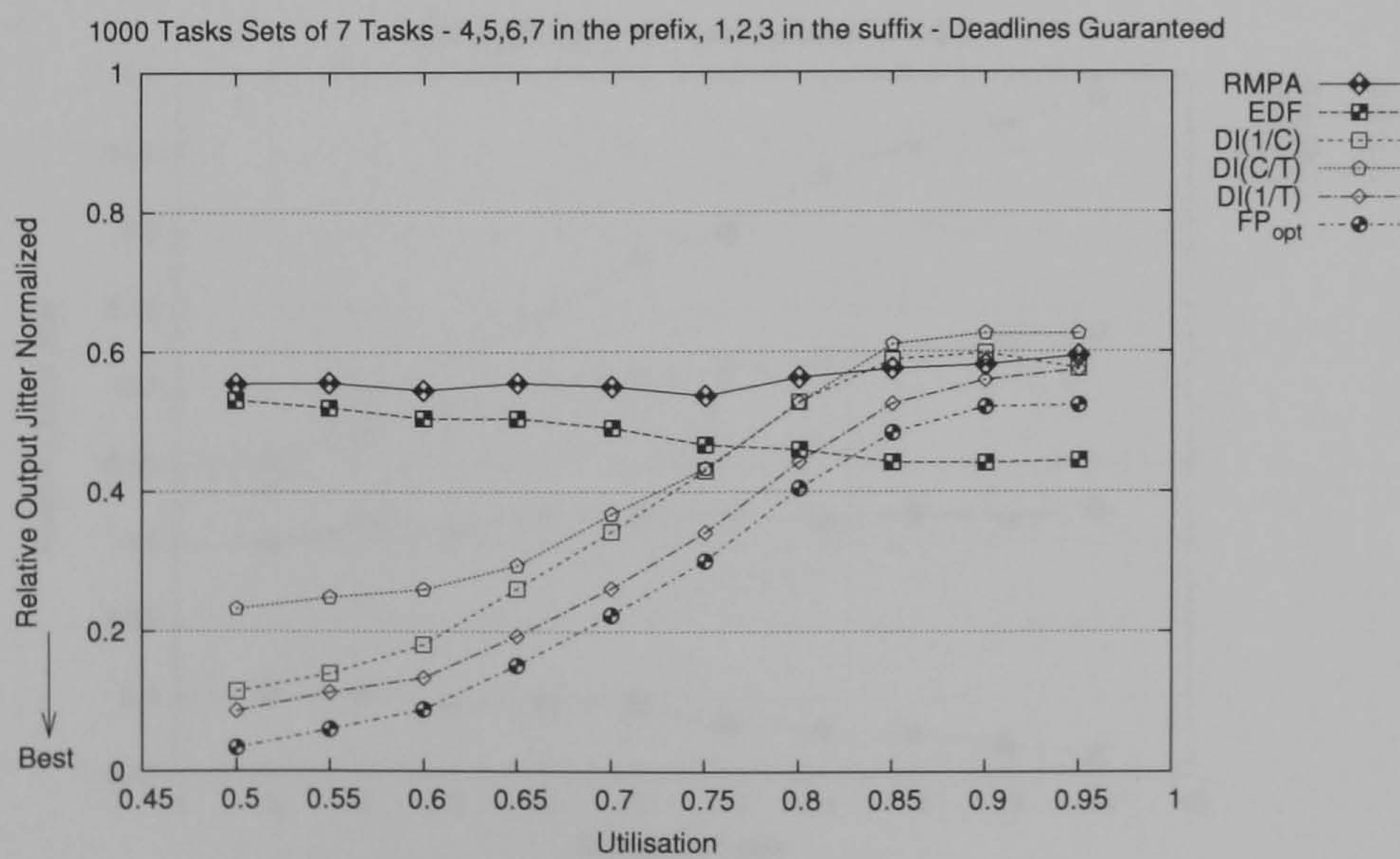
Remark 7.4.3. *The experiments show that assigning importance according to the rule "the shorter* $1/T$*, the higher the importance" (i.e. RMPA when all tasks are included) and finding a feasible solution with the DI algorithm provides a good solution for the scheduling problem of finding a feasible fixed-priority assignment that minimises the relative output jitter.*



(a) RMPA is the best FPS solution

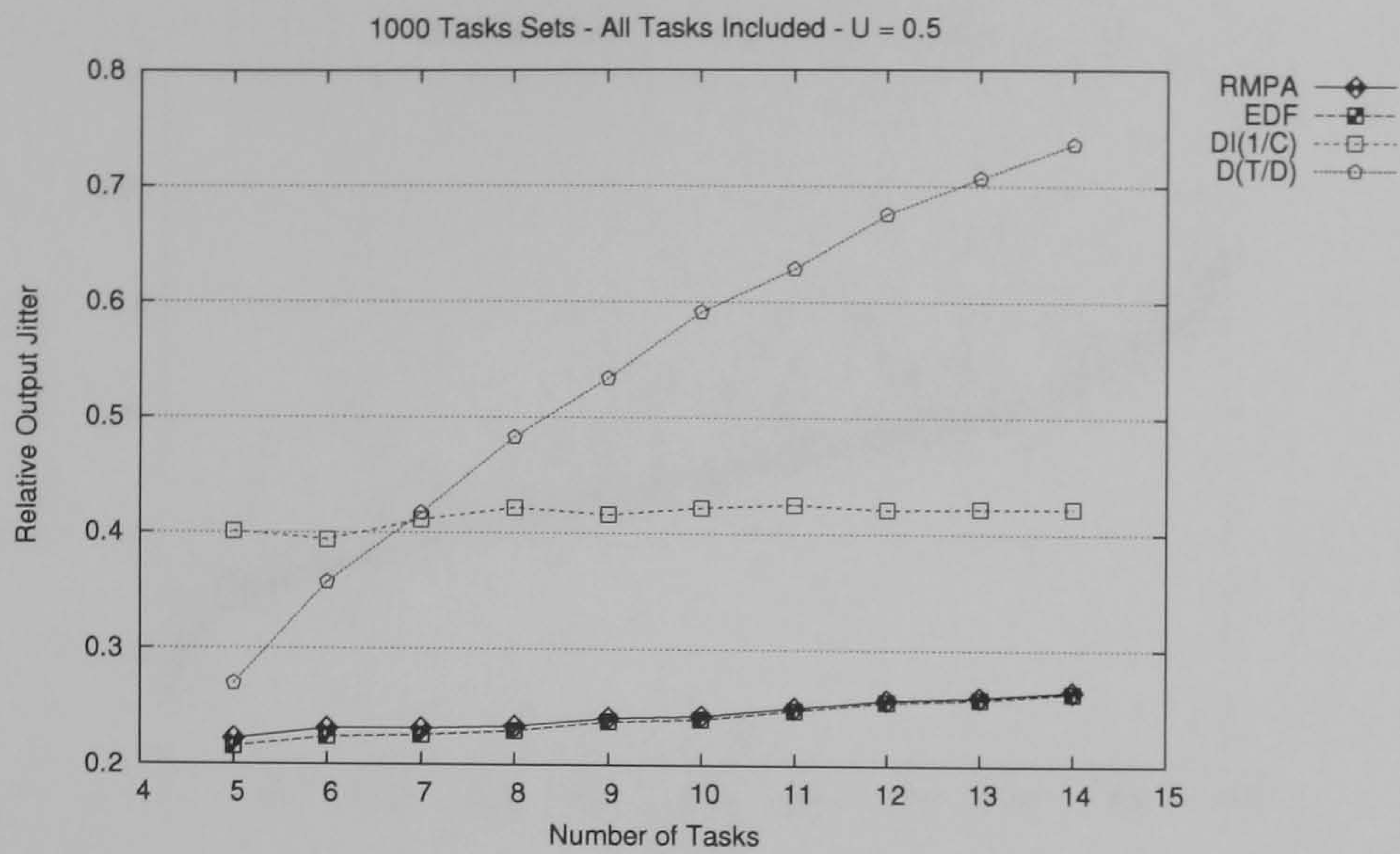


(b) DI(1/T) is the best FPS solution

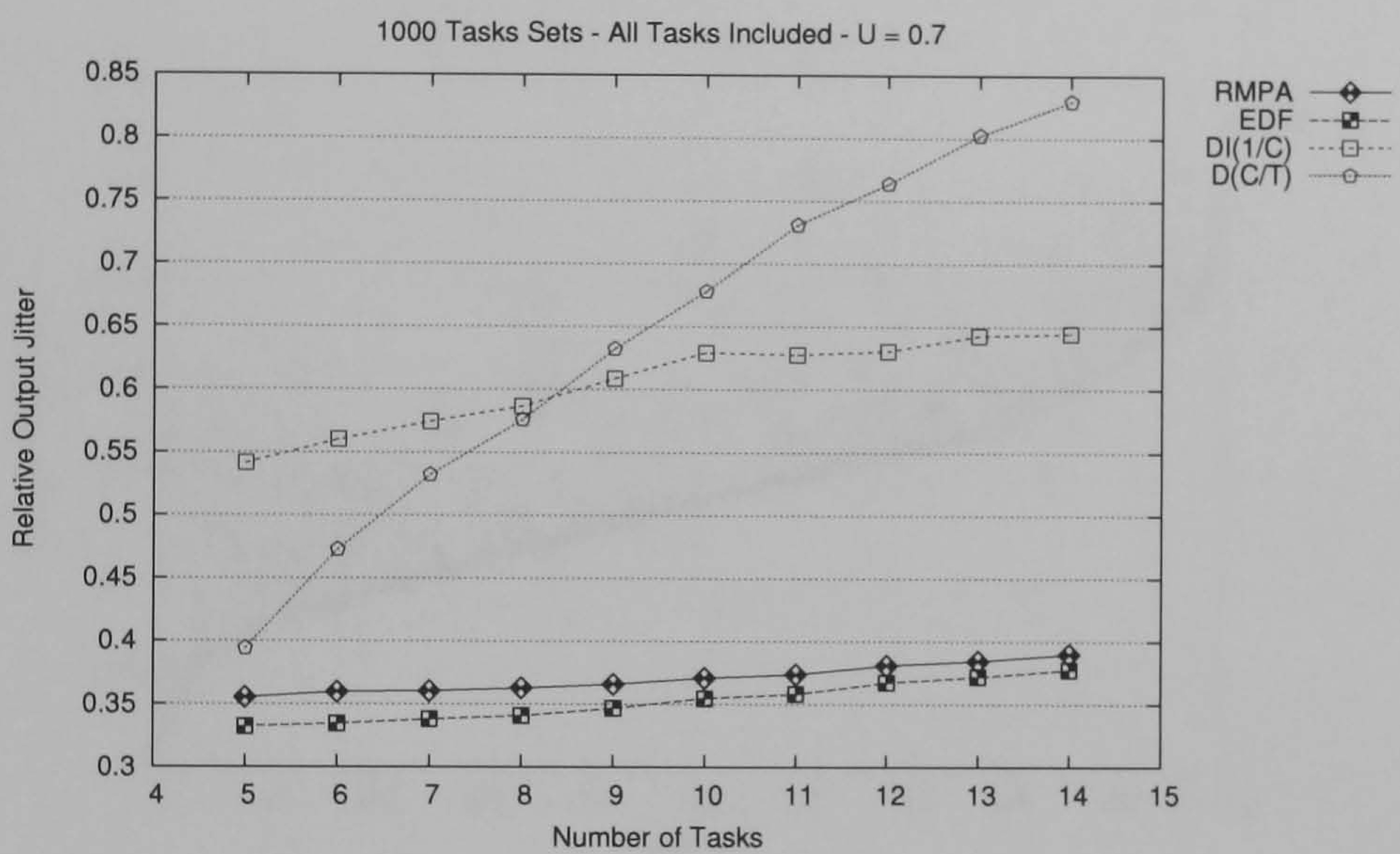


(c) DI(1/T) is the best FPS solution

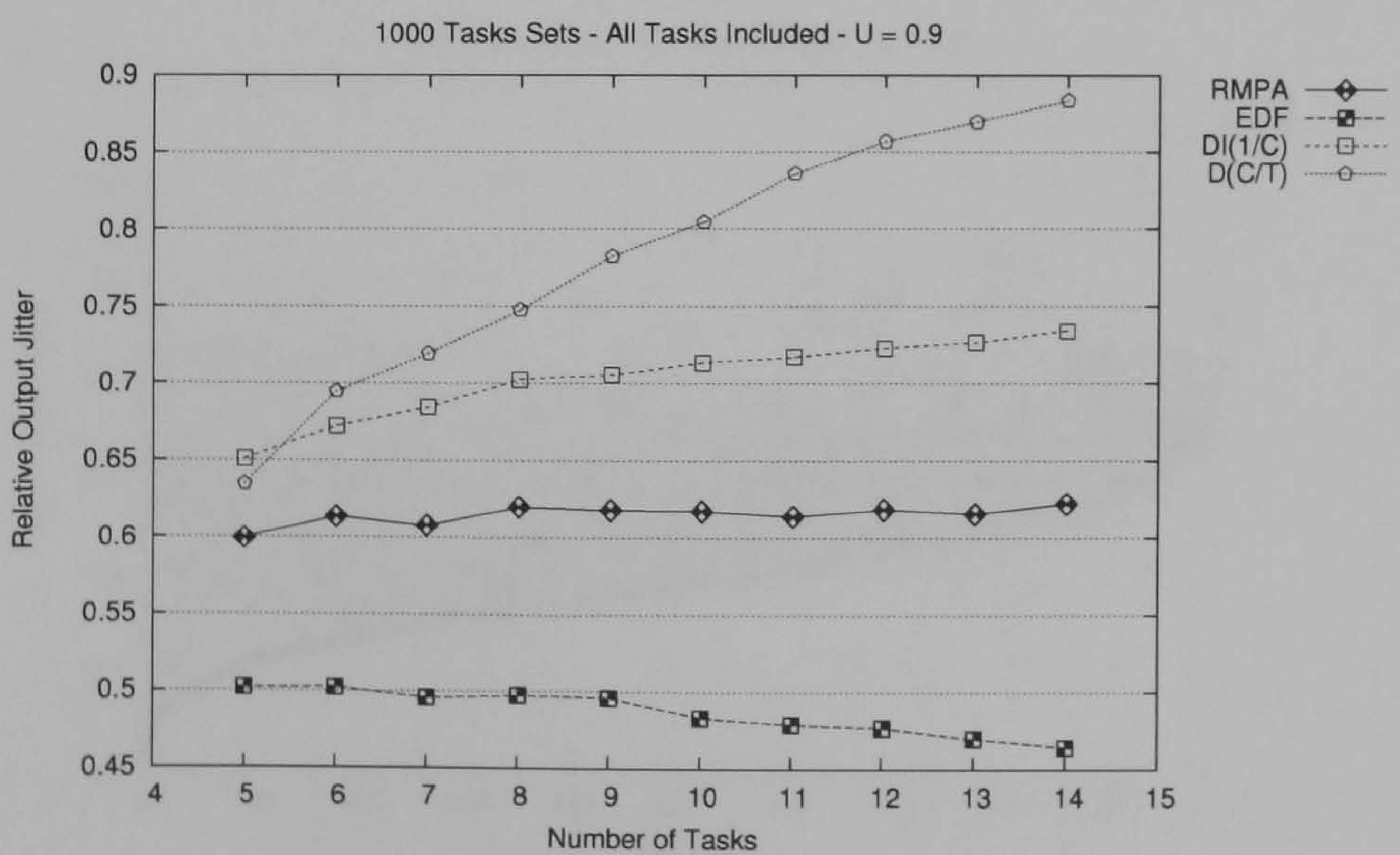
Figure 7.18: PLOTS A TYPE. Guaranteeing Deadlines and Minimising the Relative Output Jitter. Assigning priorities by shorter period is the best solution in FPS



(a) RMPA is the best FPS solution

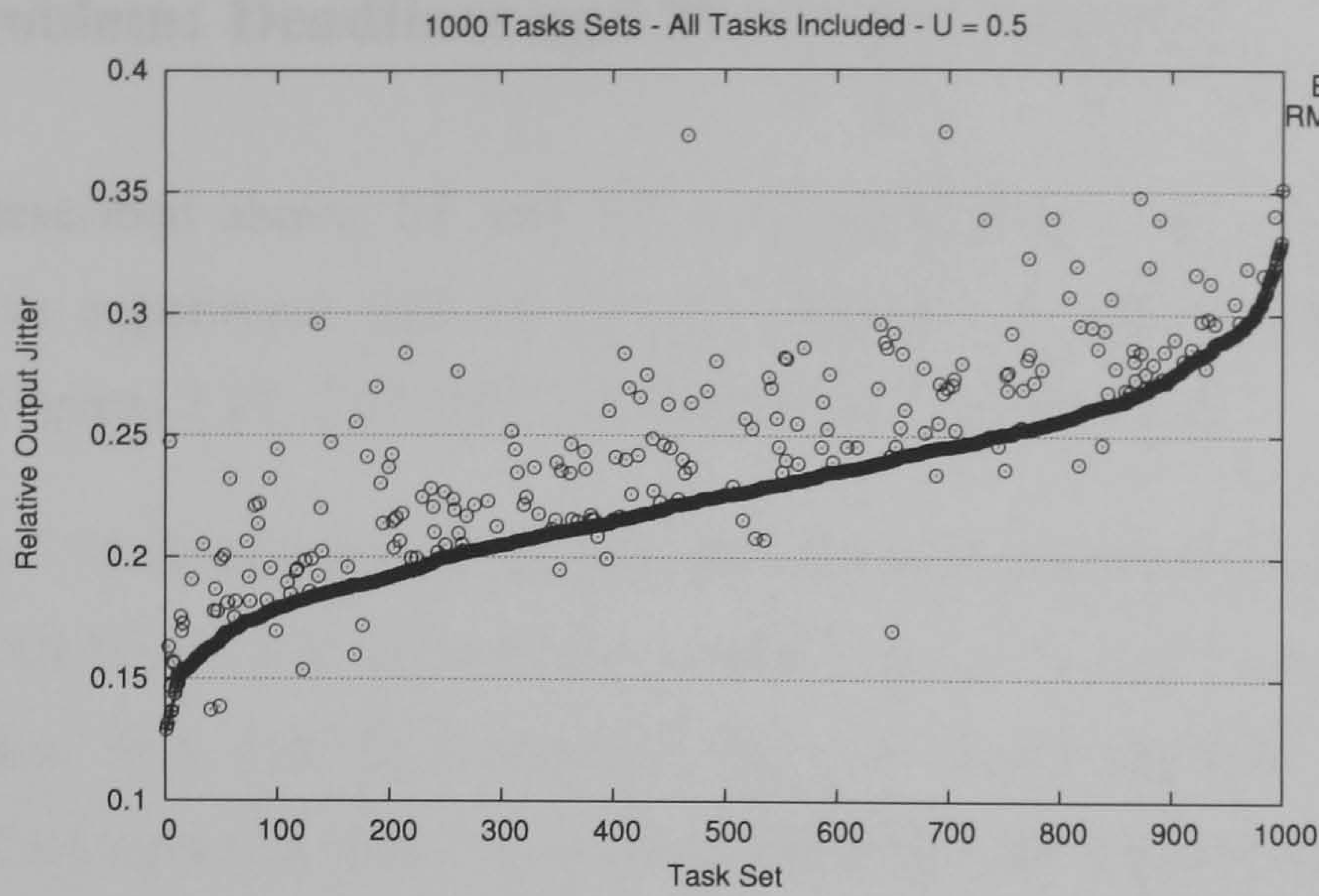


(b) DI(1/T) is the best FPS solution

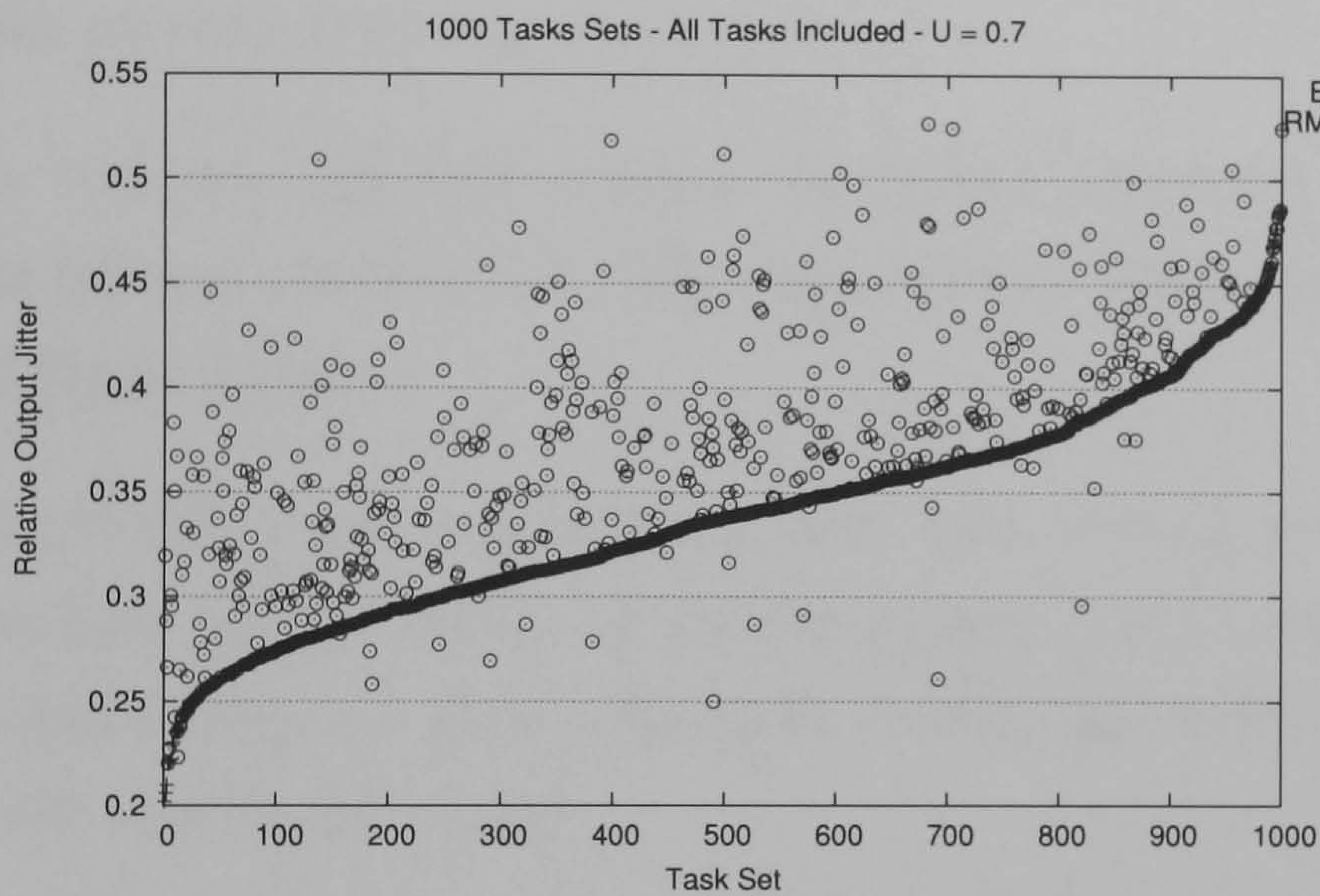


(c) DI(1/T) is the best FPS solution

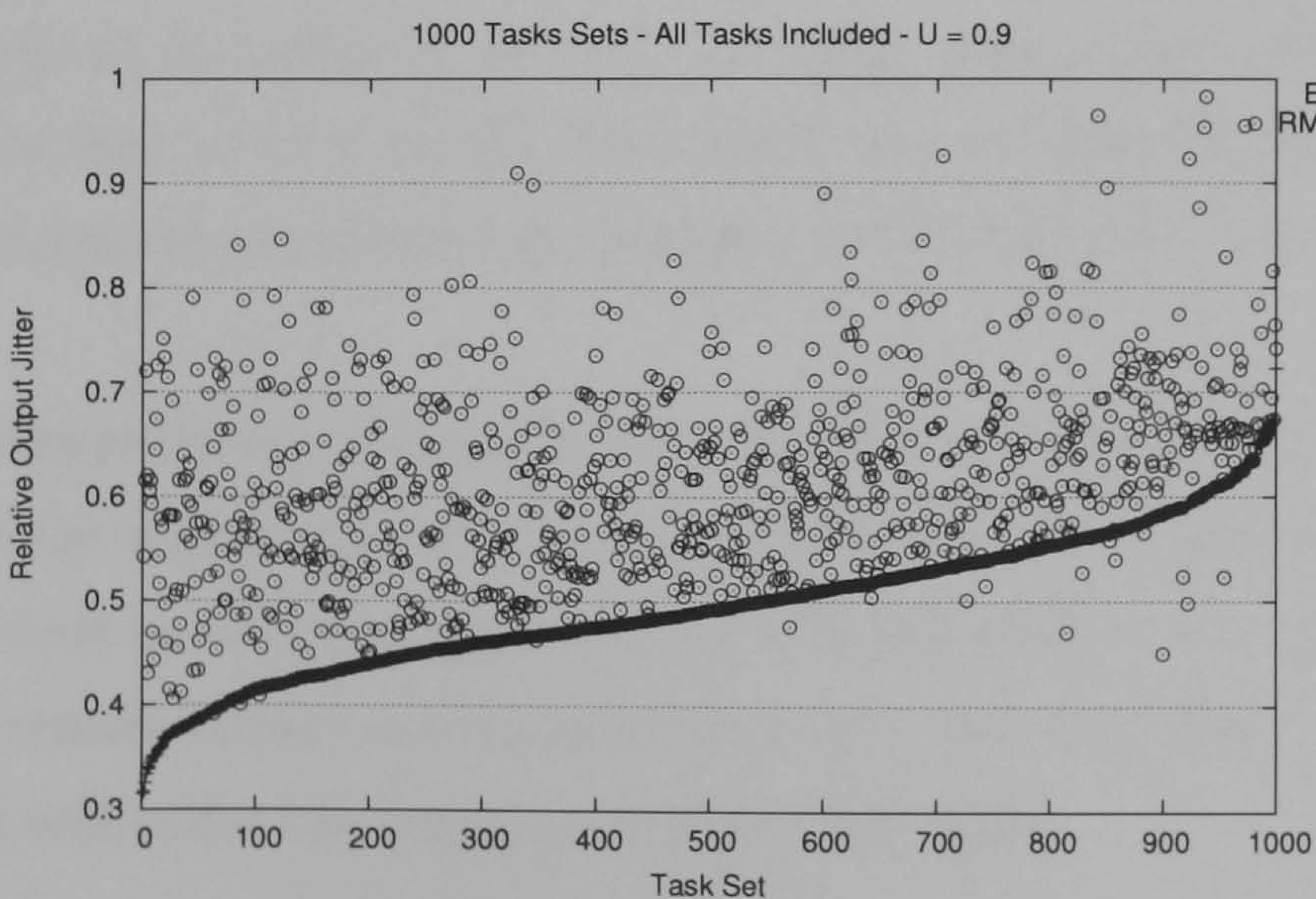
Figure 7.19: PLOTS B TYPE. Guaranteeing Deadlines and Minimising the Relative Output Jitter



(a) On average the relative output jitter is 0.225 for EDF and 0.231 for RMPA



(b) On average the relative output jitter is 0.338 for EDF and 0.360 for RMPA



(c) On average the relative output jitter is 0.495 for EDF and 0.607 for RMPA

Figure 7.20: PLOTS C TYPE. RMPA against EDF for the Deadlines and Relative Output Jitter problem. While on average one outcome is better than the other one, it is not necessarily true for a number of solutions

7.4.4 Problem: Deadlines and Maximum Latency

As it was described above, LT and T/C rules performed badly with the DI algorithm. Therefore, we experiment with all the rules looking for the best one. The results are depicted in Figures 7.21, 7.22 and 7.23 (pages 152, 153 and 154).

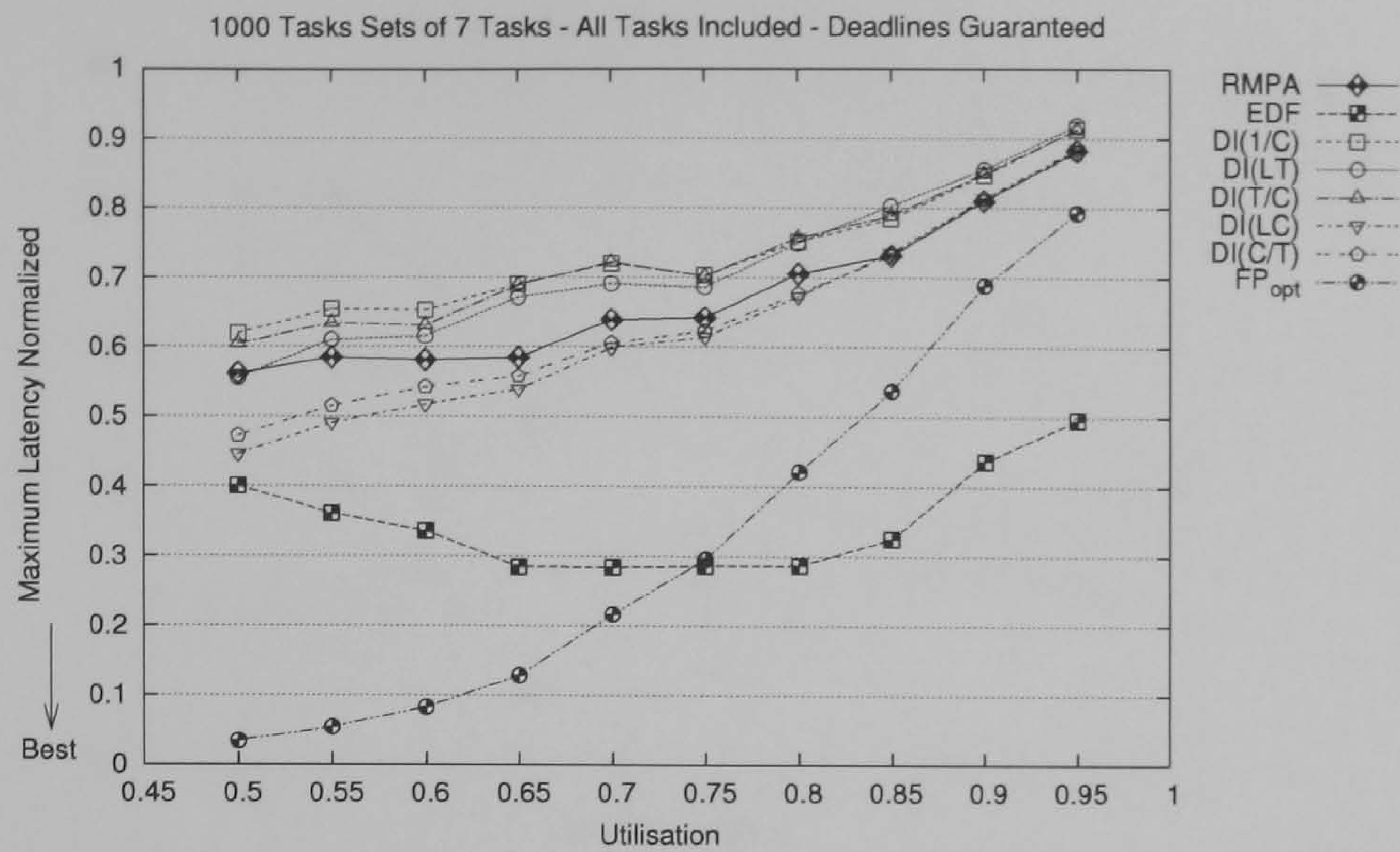
In Figure 7.21, RMPA and all DI solutions perform badly with respect to EDF. Our best solution is DI(LC) but it is far from the optimal one and is near to the RMPA solution for high utilisation. Note that the $\mathcal{L}$*-optimal* solution is better than EDF for low and medium utilisation. Consequently, there exists room for additional improvements in fixed-priority assignment algorithms. For task subsets, improvements are obtained with DI(LC) and DI(C/T) which are close to the $\mathcal{L}$*-optimal* solution.

In Figure 7.22, note that EDF is better than RMPA and all DI solutions. Observe that when the utilisation increases, the difference between RMPA and the DI solutions is minimal (see Figure 7.22(c)).

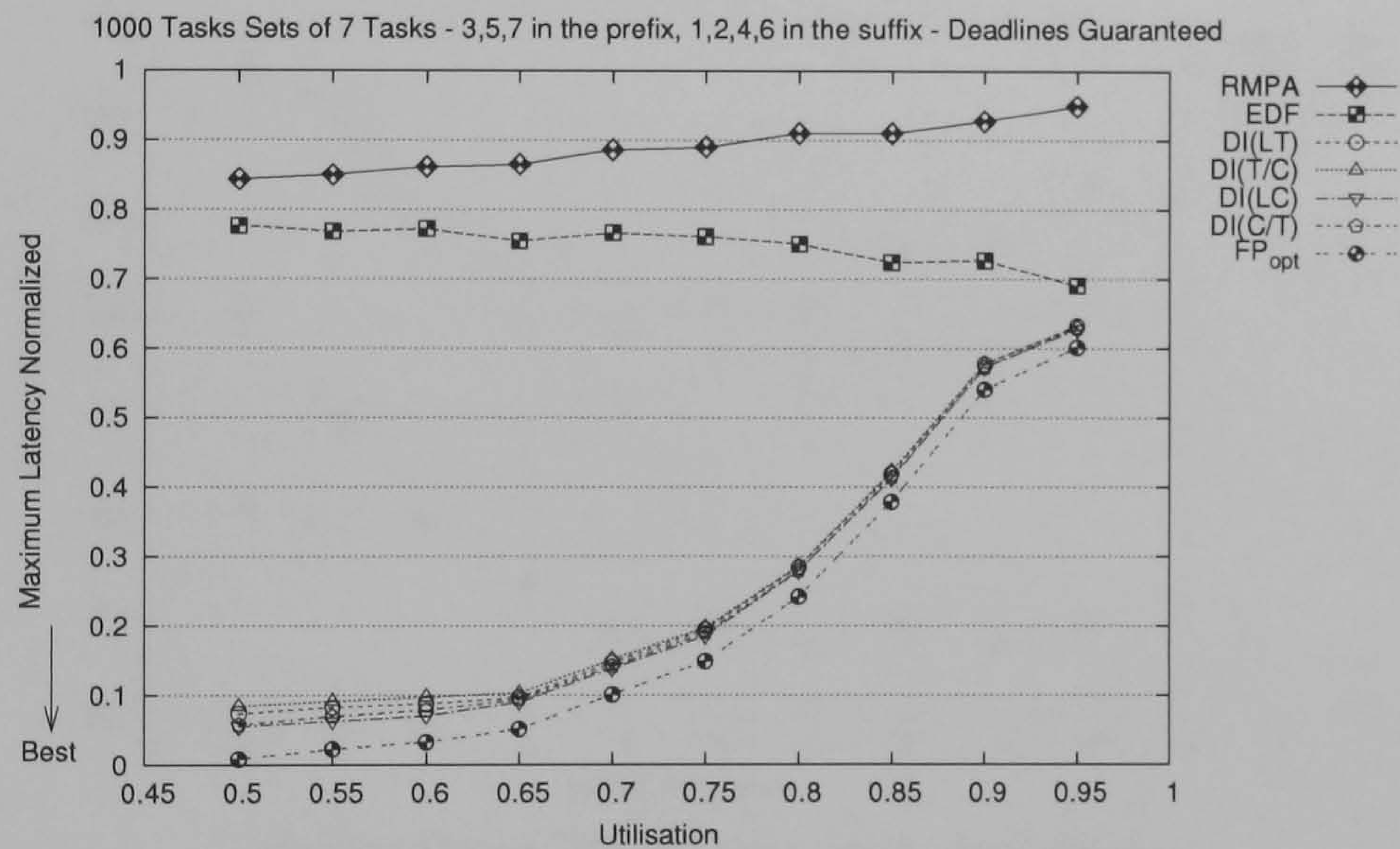
In despite of EDF is on average much better than DI(LC), in Figure 7.23 can be observed that there exists a number of solutions where DI(LC) outperforms EDF. For instance, all data in Figure 7.23(b) corresponds to the single point EDF=$(7, 299.6)$ and DI(LC)=$(7, 362.2)$ in Figure 7.22(b).

Remark 7.4.4. *The results of these experiments shows that the DI algorithm with importance assigned according to the rule "the larger computation time, the higher the importance" provides a better solution than RMPA for the scheduling problem of finding a feasible fixed-priority assignment that minimises the maximum latency.*

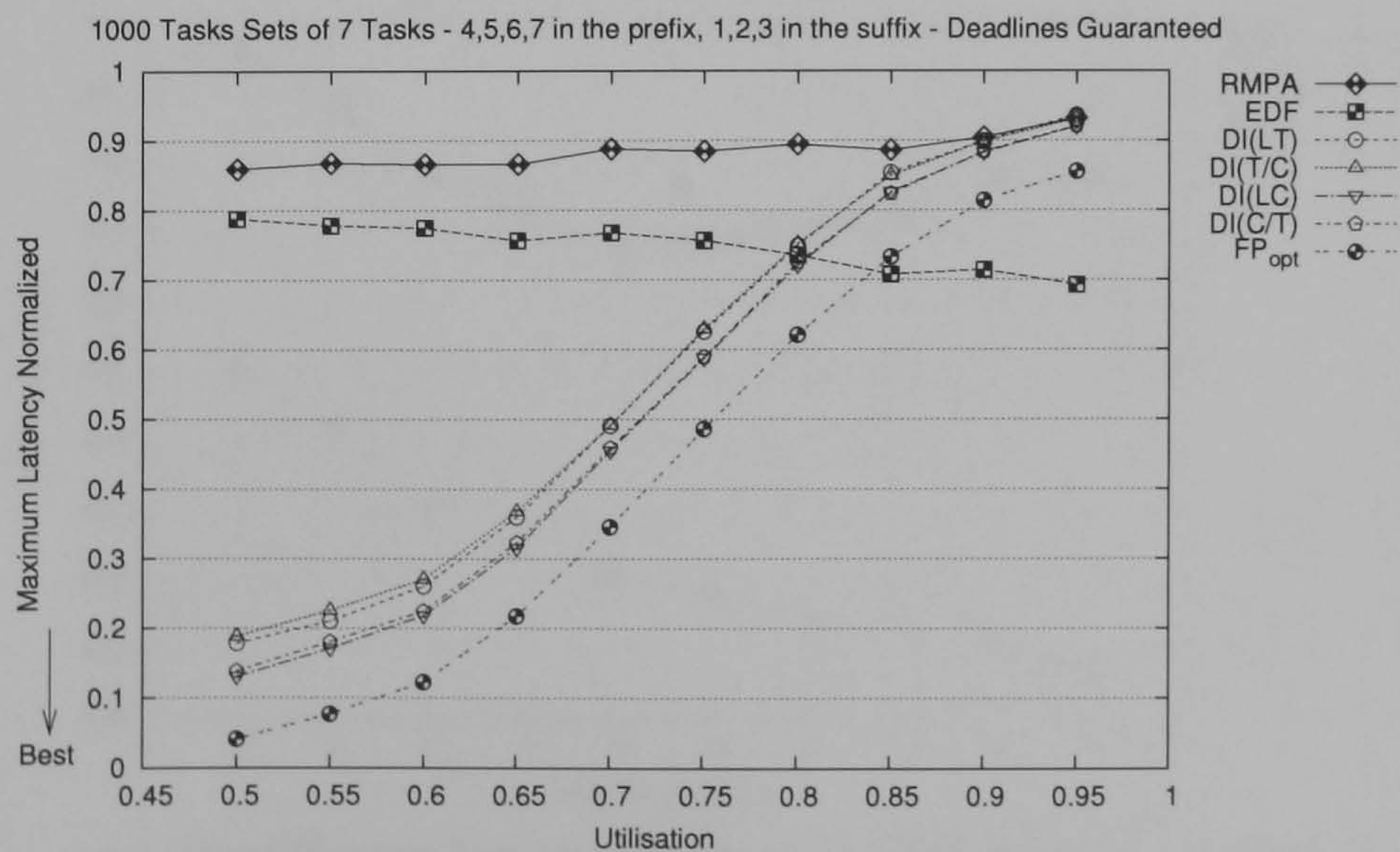
The maximum latency of a task (i.e. the maximum elapsed time between the completion time and the beginning time) is greater in tasks with long C than in tasks with short C. In addition, it increases due to the interference produced by lower priority tasks. Thus, intuitively the remark is true because under DI(LC), a task with long C has a higher priority and hence, it will suffer less interference from other tasks.



(a) Our best solution is DI(LC), however, note how far it is from the optimal one

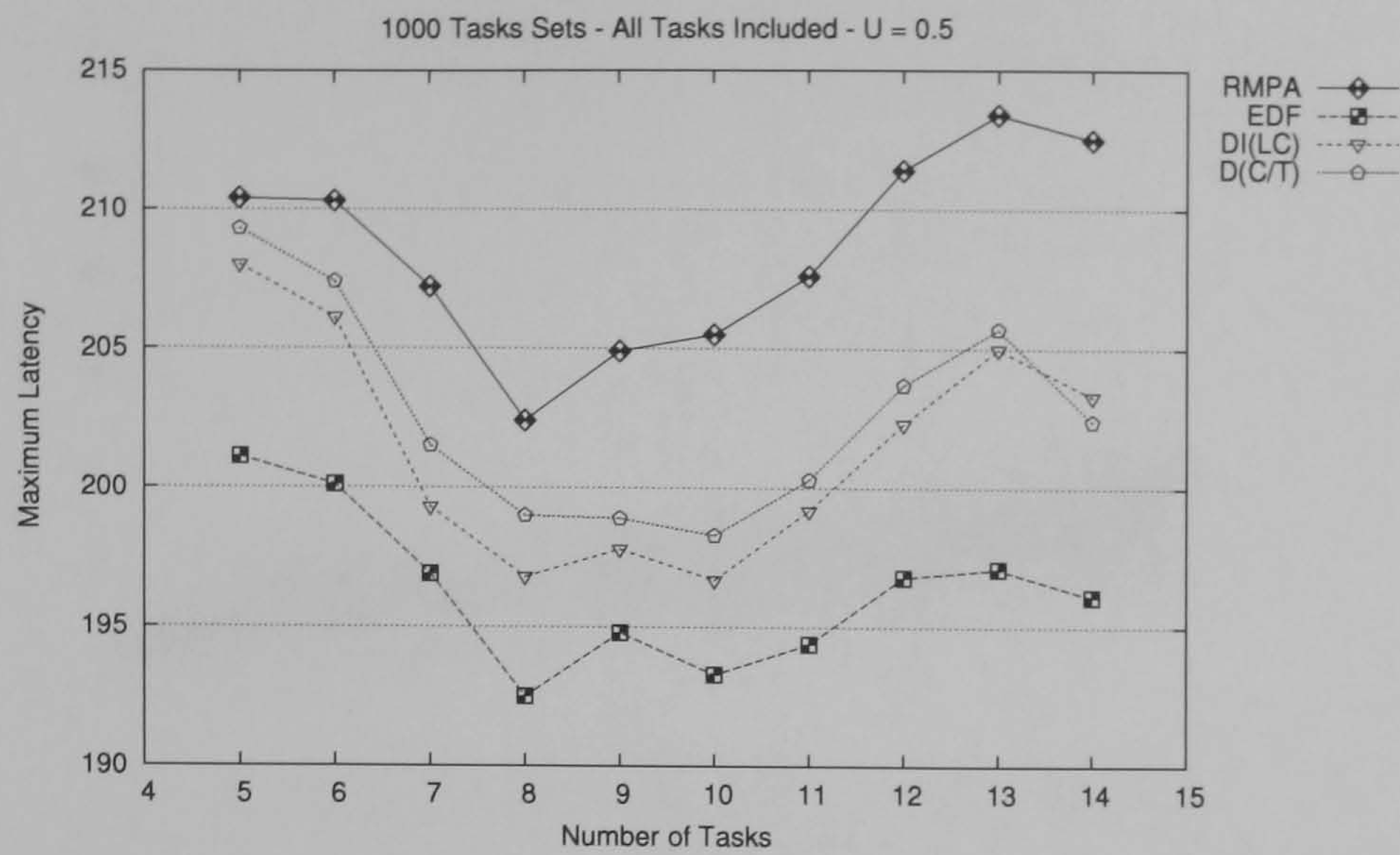


(b) The best is DI(LC) even so all DI solutions perform good

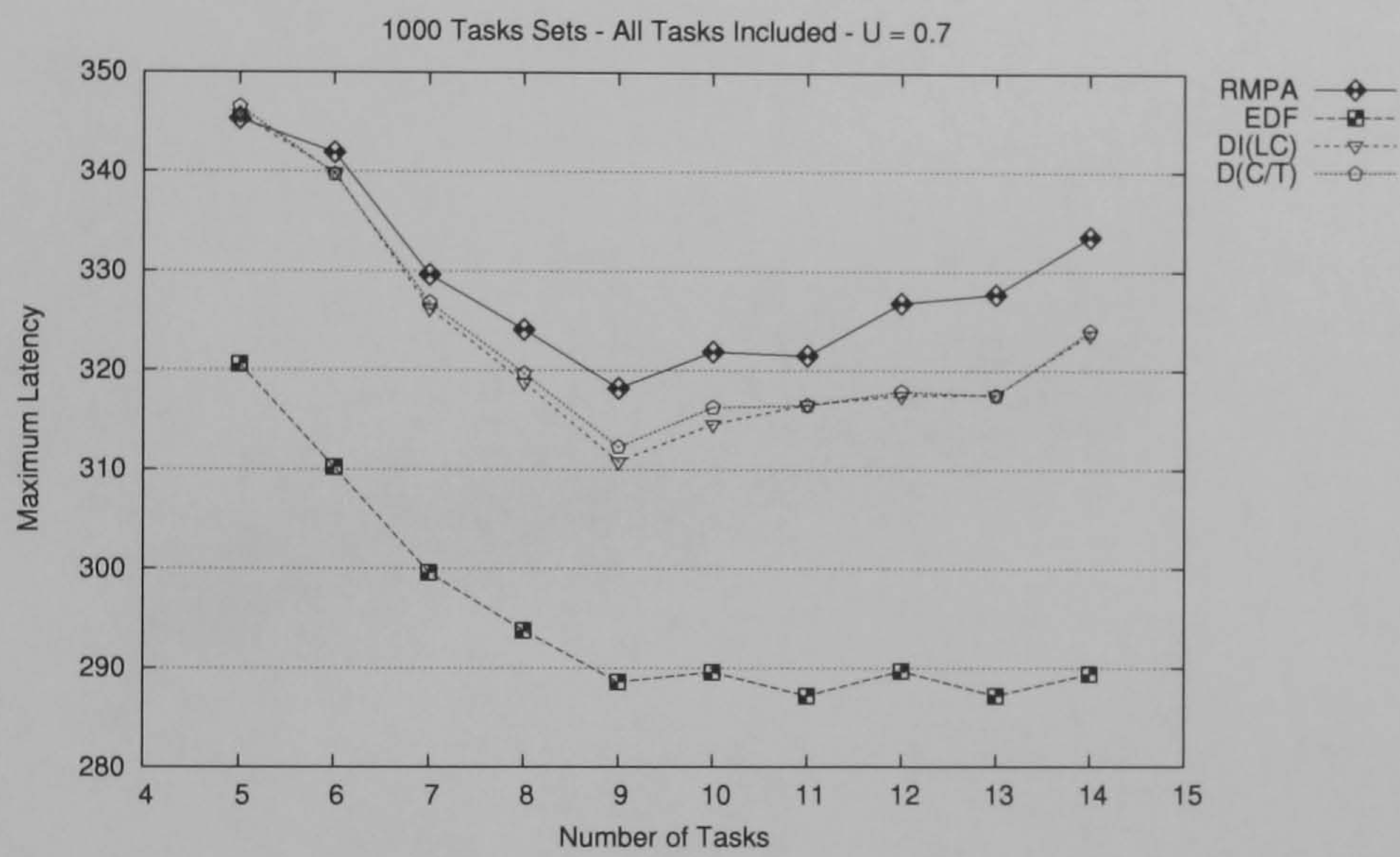


(c) The best are DI(LC) and DI(C/T)

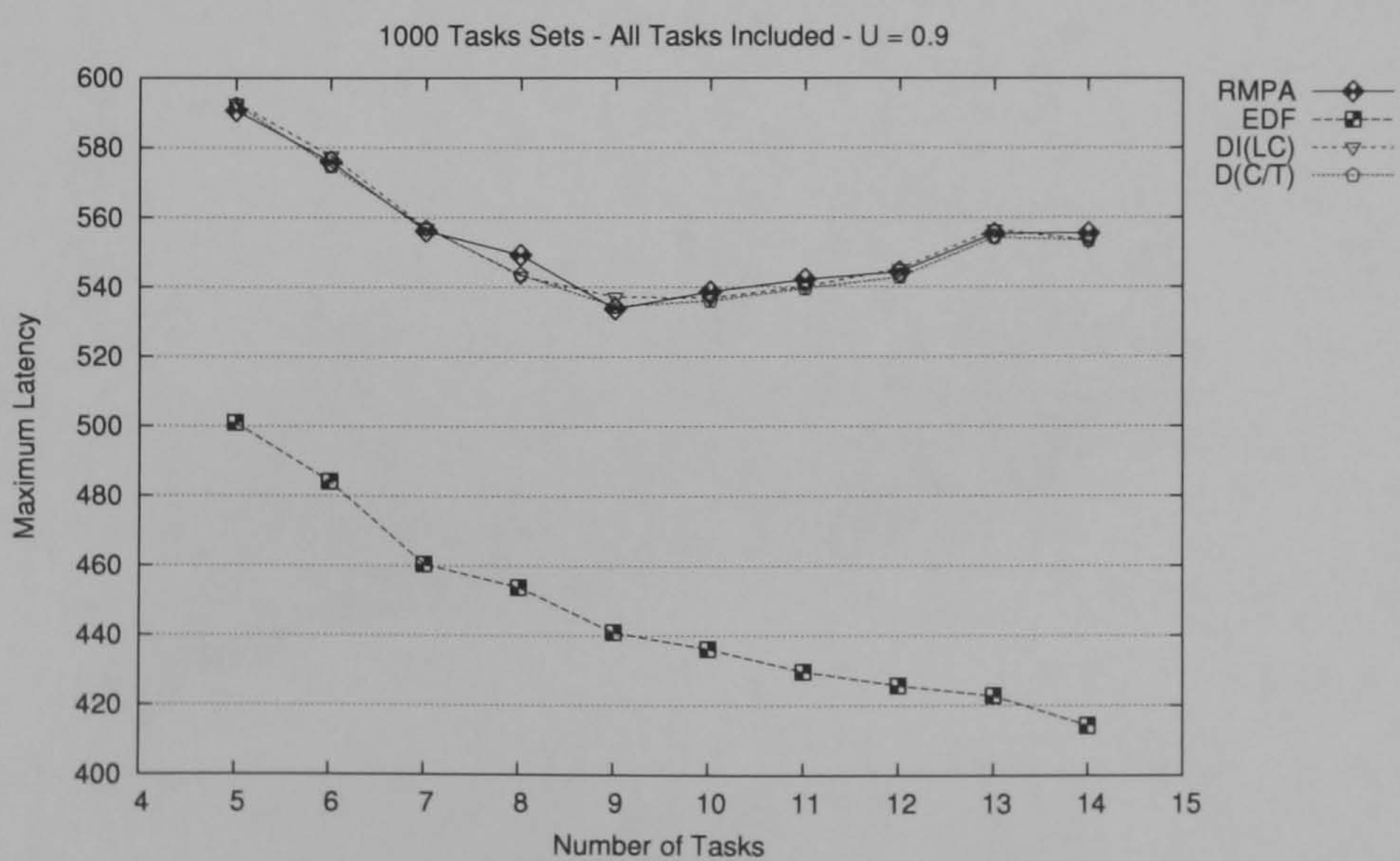
Figure 7.21: PLOTS A TYPE. Guaranteeing Deadlines and Minimising the Maximum Latency under different priority assignments. Our best solution is DI(LC)



(a) DI(LC) is our best solution

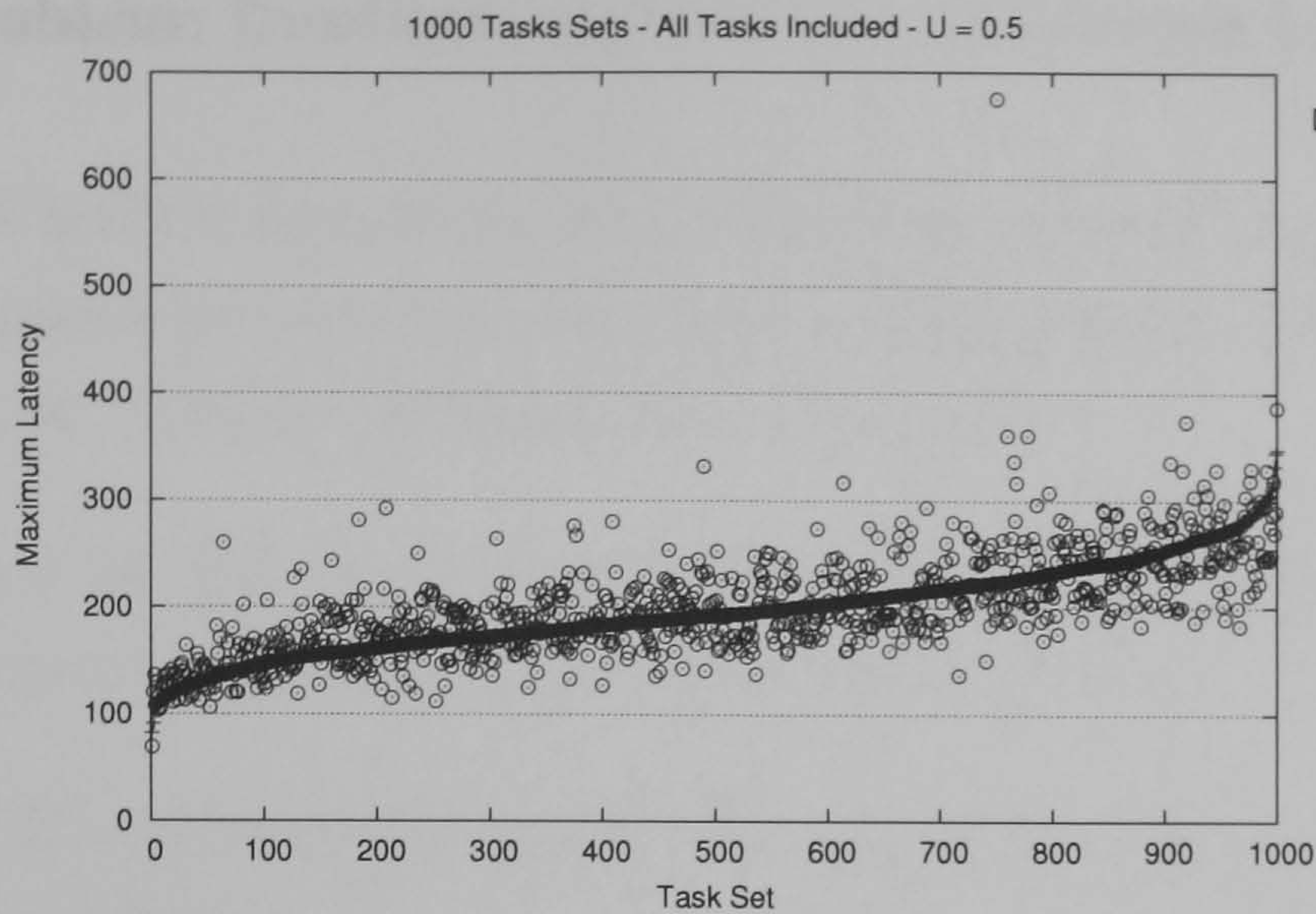


(b) DI(LC) and DI(C/T) show similar performance

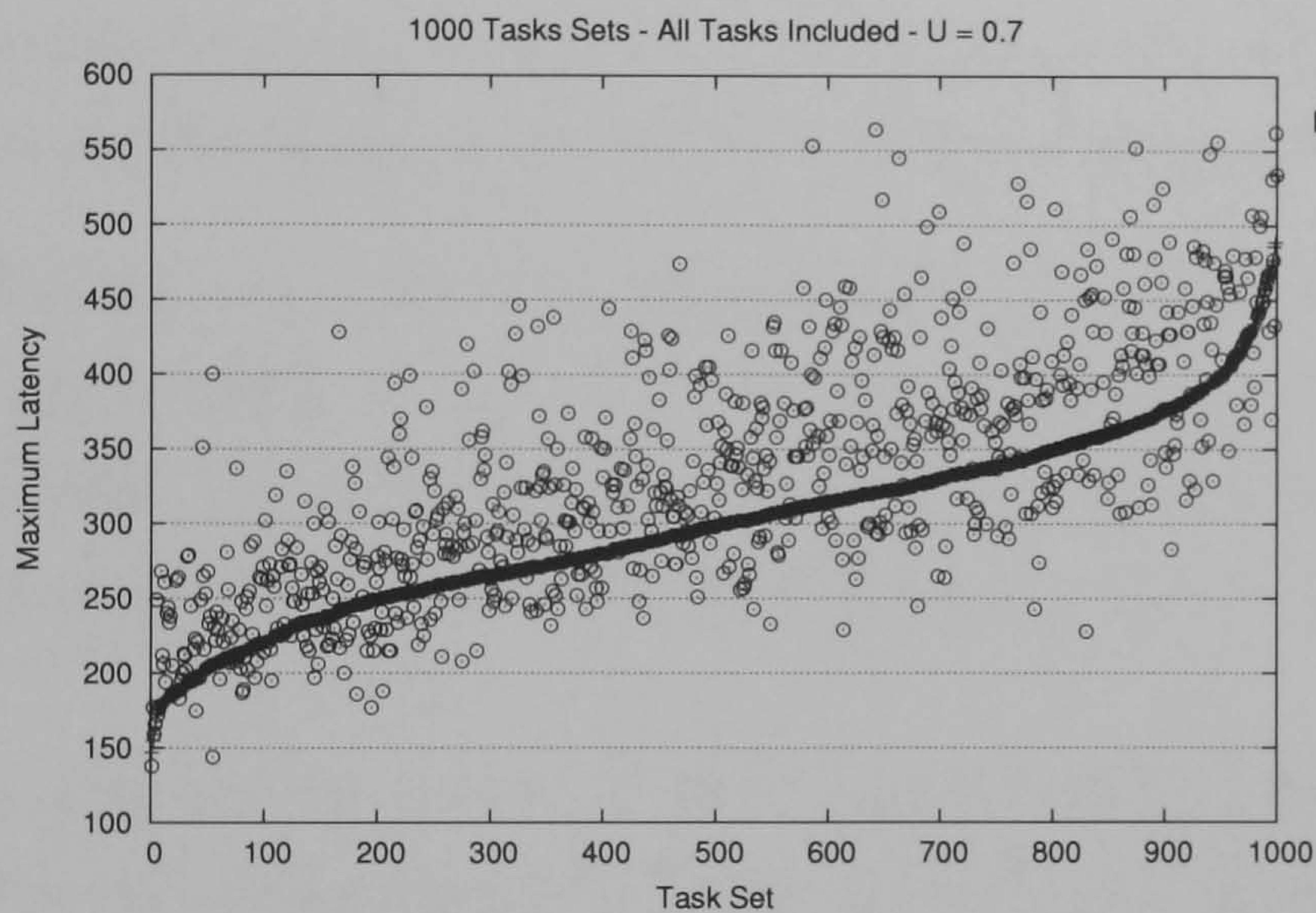


(c) The difference between RMPA and the DI solutions is minimal

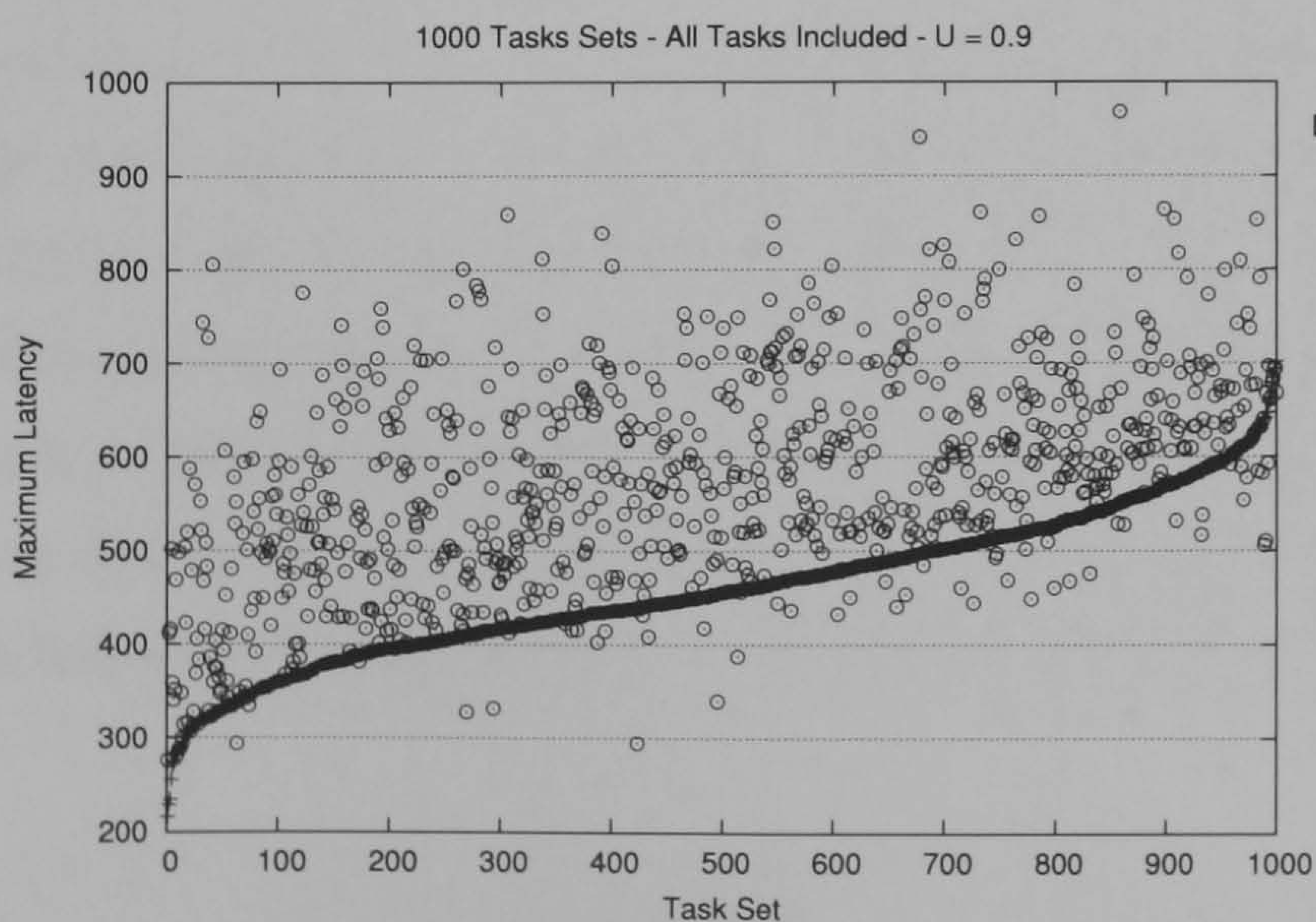
Figure 7.22: PLOTS B TYPE. Guaranteeing Deadlines and Minimising the Maximum Latency



(a) On average the maximum latency is 196.9 for EDF and 199.8 for DI(LC)



(b) On average the maximum latency is 299.6 for EDF and 326.2 for DI(LC)



(c) On average the maximum latency is 460.6 for EDF and 557.0 for DI(LC)

Figure 7.23: PLOTS C TYPE. Comparing DI(LC) against EDF for the Deadlines and the Maximum Latency problem. While on average one outcome is better than the other one, it is not necessarily true for a number of solutions

7.4.5 Problem: Deadlines and Relative Maximum Latency

Using the 1/C and T/C rules for assigning importance, our experiment compares how good the DI algorithm is for minimising the relative maximum latency. The results are depicted in Figures 7.24, 7.25 and 7.26 (pages 156, 157 and 158).

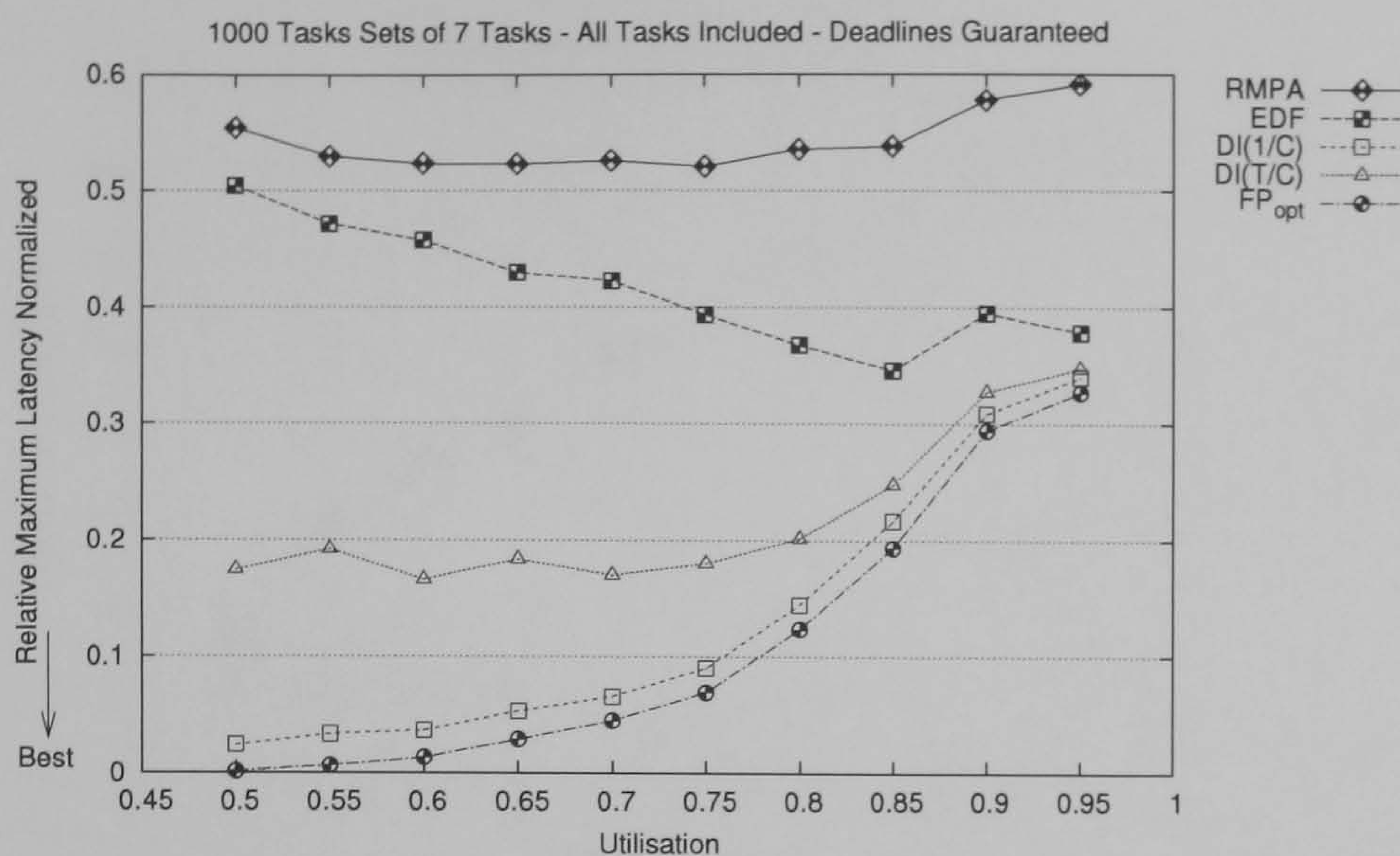
In Figure 7.24, both DI(1/C) and DI(T/C) outperform EDF in the three plots for all utilisation values. Note that DI(1/C) is a near-optimal solution.

Figure 7.25 confirms that both DI(1/C) and DI(T/C) provide excellent solutions for this particular problem.

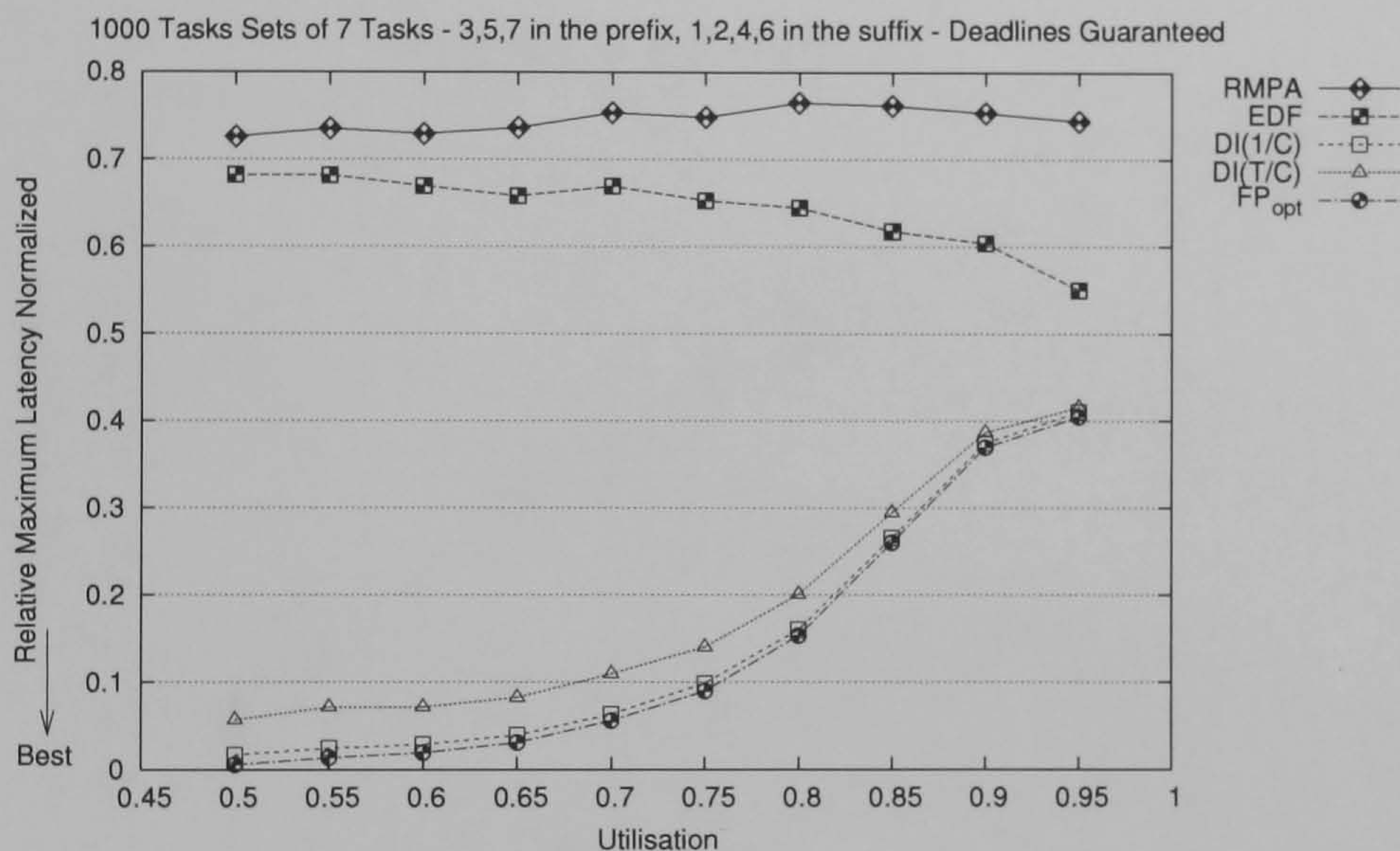
The superiority of DI(1/C) with respect to EDF is confirmed in Figure 7.26. In the three plots, observe how the stability of the DI(1/C) solution contrasts with the EDF solution.

Remark 7.4.5. *The results of these experiments shows that the DI algorithm with importance assigned according to the rule "the shorter computation time, the higher the importance" provide near-optimal solutions for the scheduling problem of finding a feasible fixed-priority assignment that minimises the relative maximum latency.*

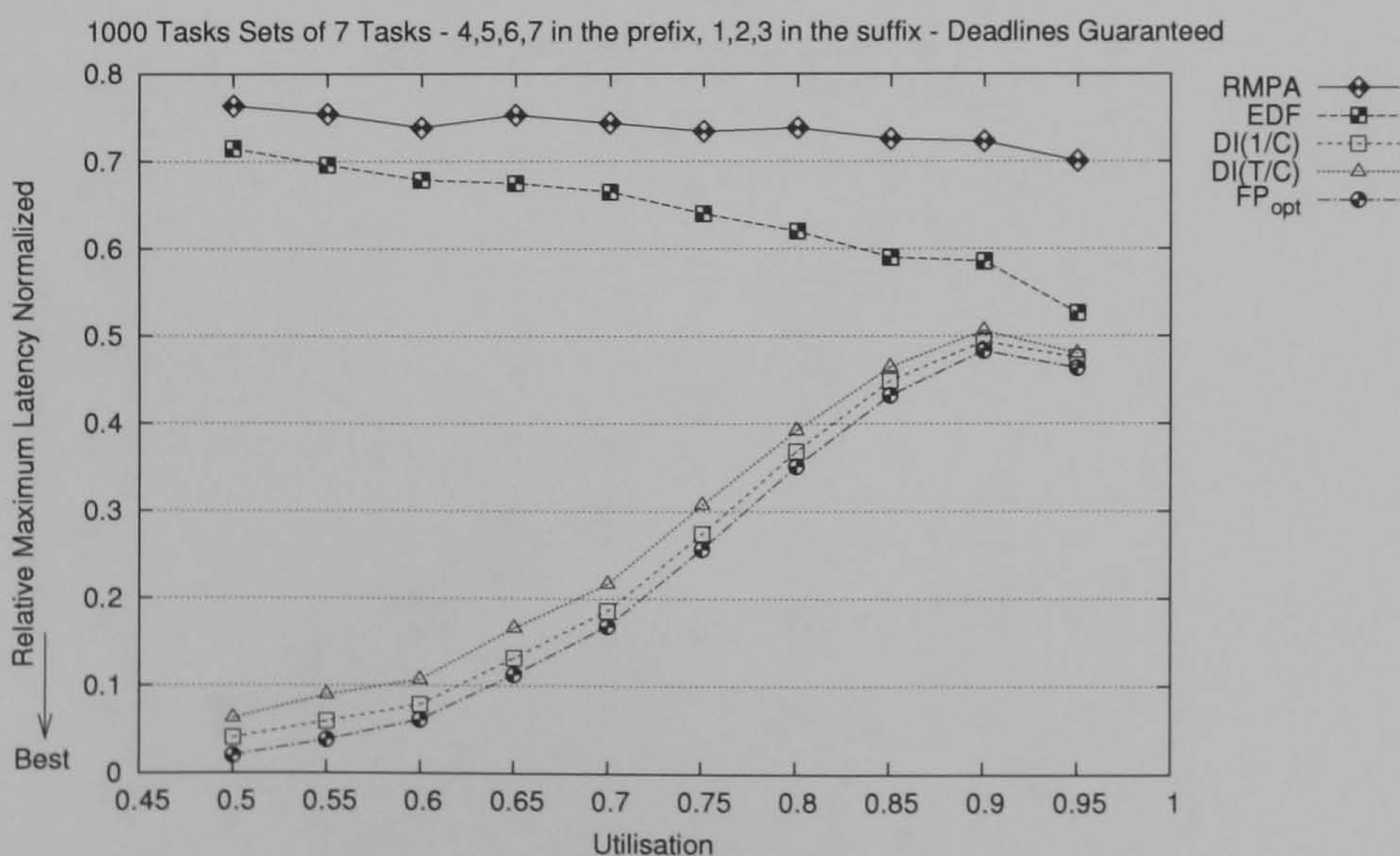
Note that in the previous section we got an opposite result to this one. This is because while the maximum latency varies with respect to C, the relative maximum latency is insensitive to the variation of C. For instance, consider a system of two tasks a and b with C_a and C_b respectively, and $T_a = T_b$, and released at the critical instant. Independently of the priority assignment (i.e. $\langle a\, b\rangle$ or $\langle b\, a\rangle$) the tasks have maximum latencies C_a and C_b respectively and then the maximum latency is $\max\{C_a, C_b\}$; on the other hand, they have relative maximum latencies $\frac{C_a}{C_a} = 1$ and $\frac{C_b}{C_b} = 1$ and therefore, the relative maximum latency is 1. Thus, intuitively the above remark is true because under DI(1/C), tasks complete soon and hence, their relative maximum latencies are shorter; i.e the maximum elapsed time between the completion time and the beginning time divided by C will be shorter.



(a) DI(1/C) is a near-optimal solution. Both DI(1/C) and DI(T/C) outperform EDF

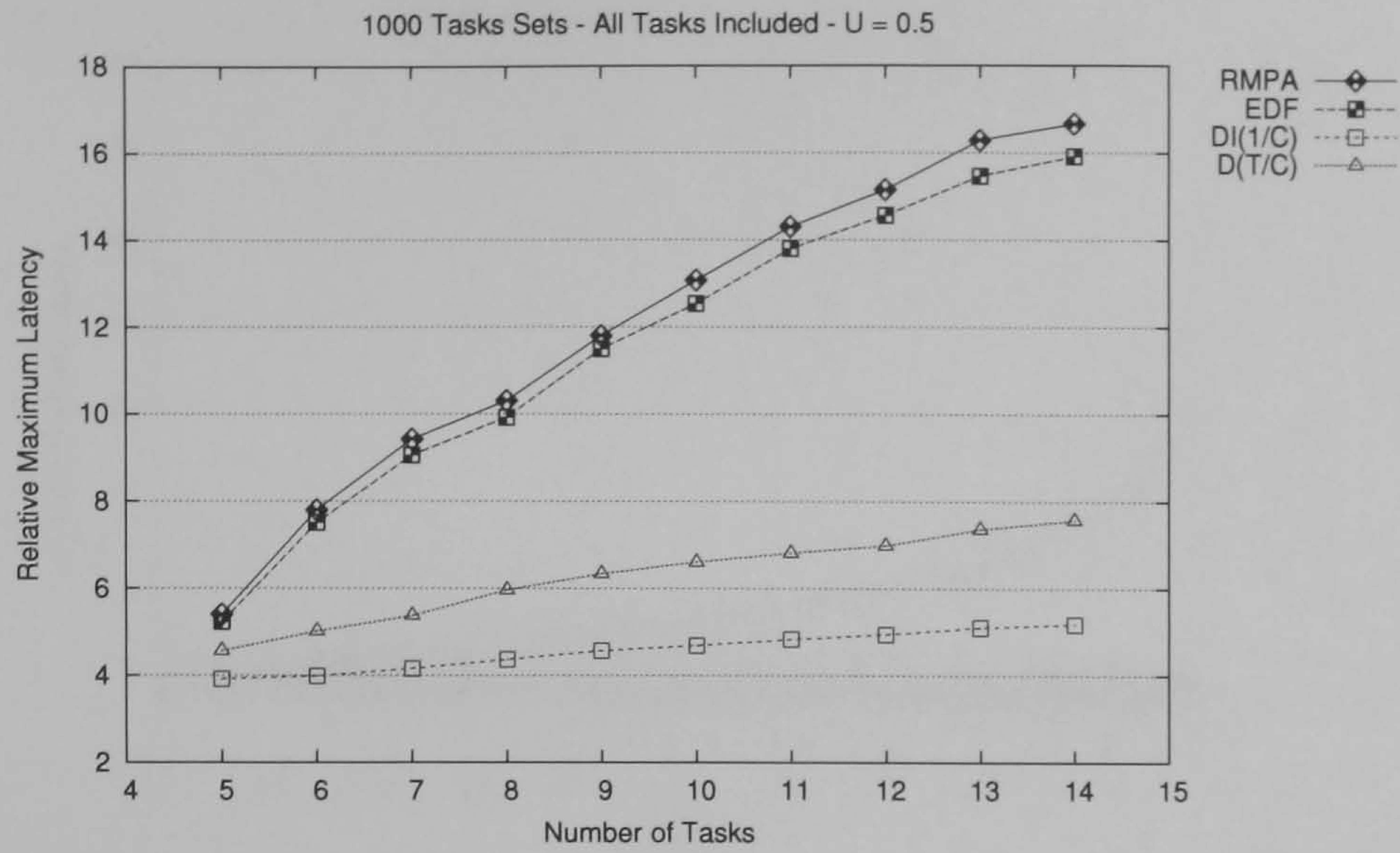


(b) Both DI(1/C) and DI(T/C) are excellent solutions

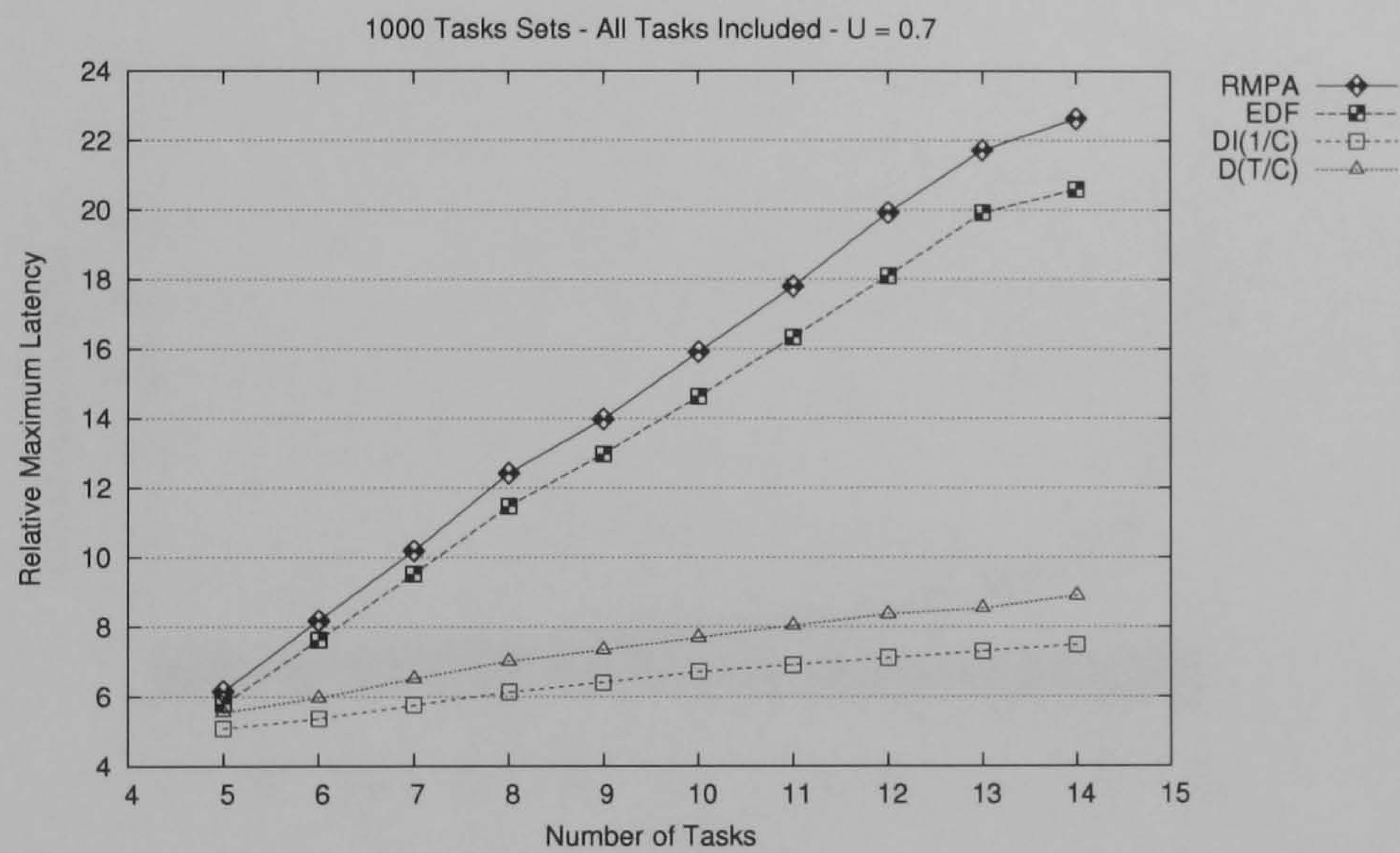


(c) Both DI(1/C) and DI(T/C) are excellent solutions

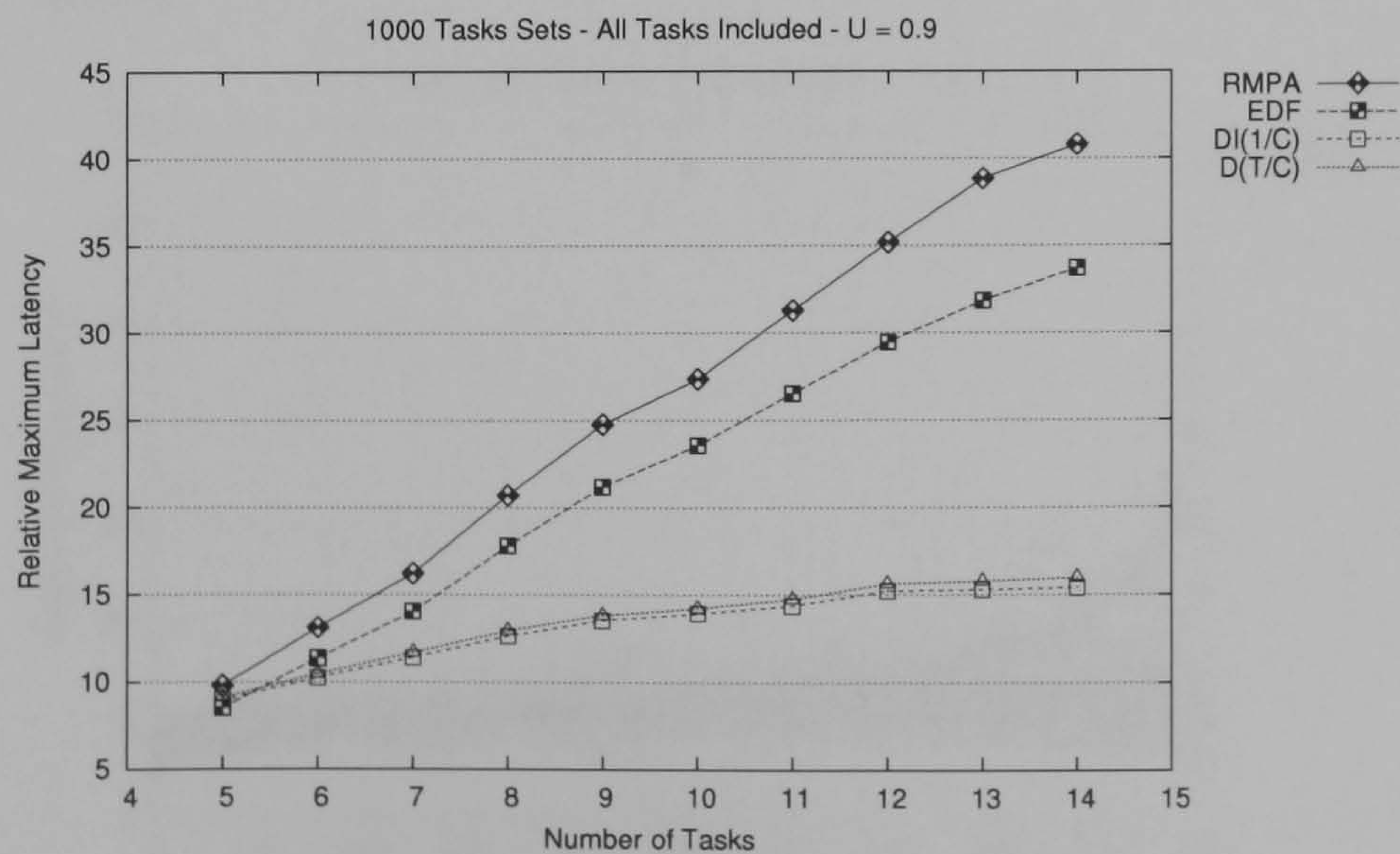
Figure 7.24: PLOTS A TYPE. Guaranteeing Deadlines and Minimising the Relative Maximum Latency. In all cases, both DI(1/C) and DI(T/C) outperform EDF. Note that DI(1/C) is a near-optimal solution.



(a) Both DI(1/C) and DI(T/C) are excellent solutions. Note that EDF and RMPA perform similar

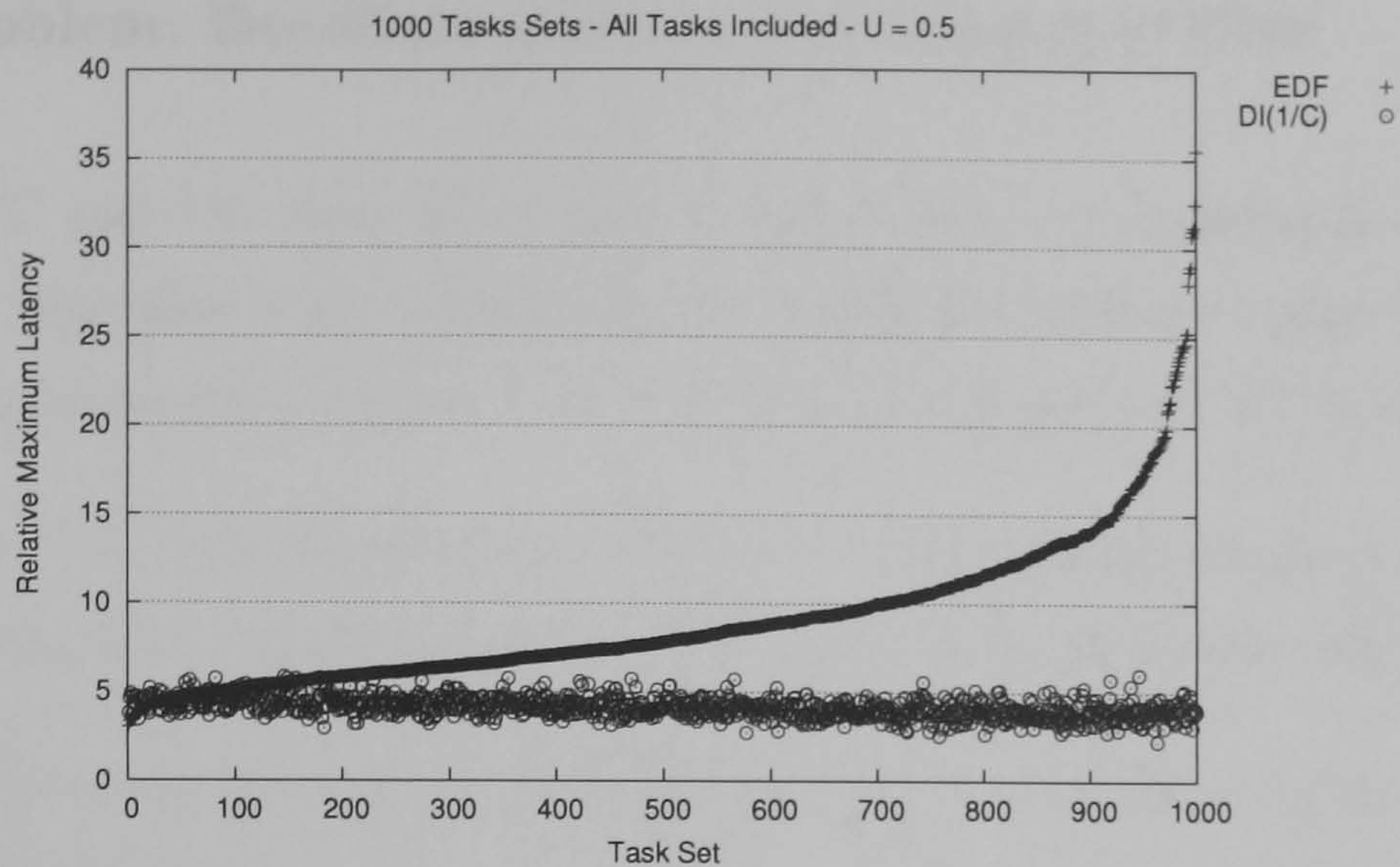


(b) Both DI(1/C) and DI(T/C) are excellent solutions

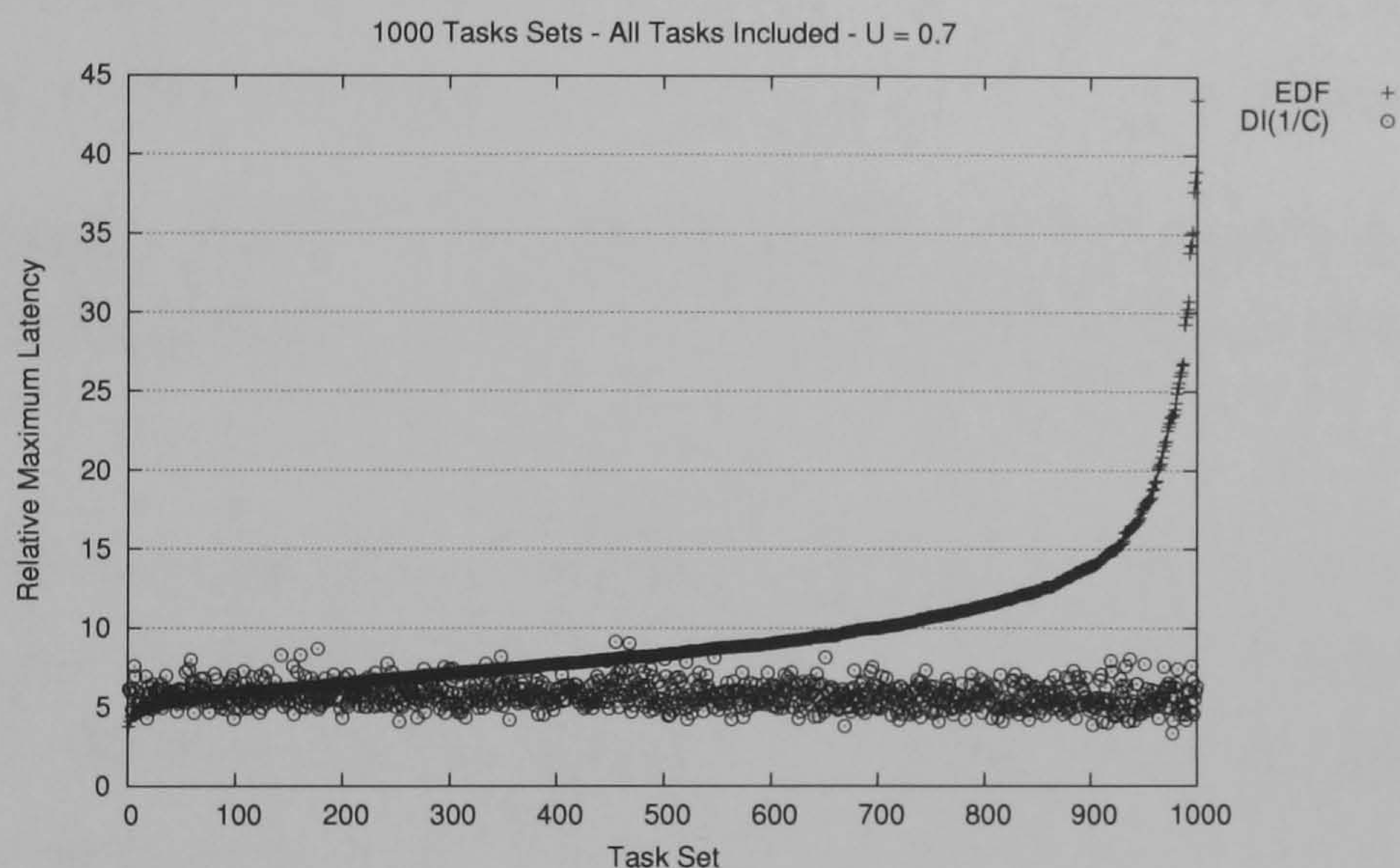


(c) Both DI(1/C) and DI(T/C) are excellent solutions

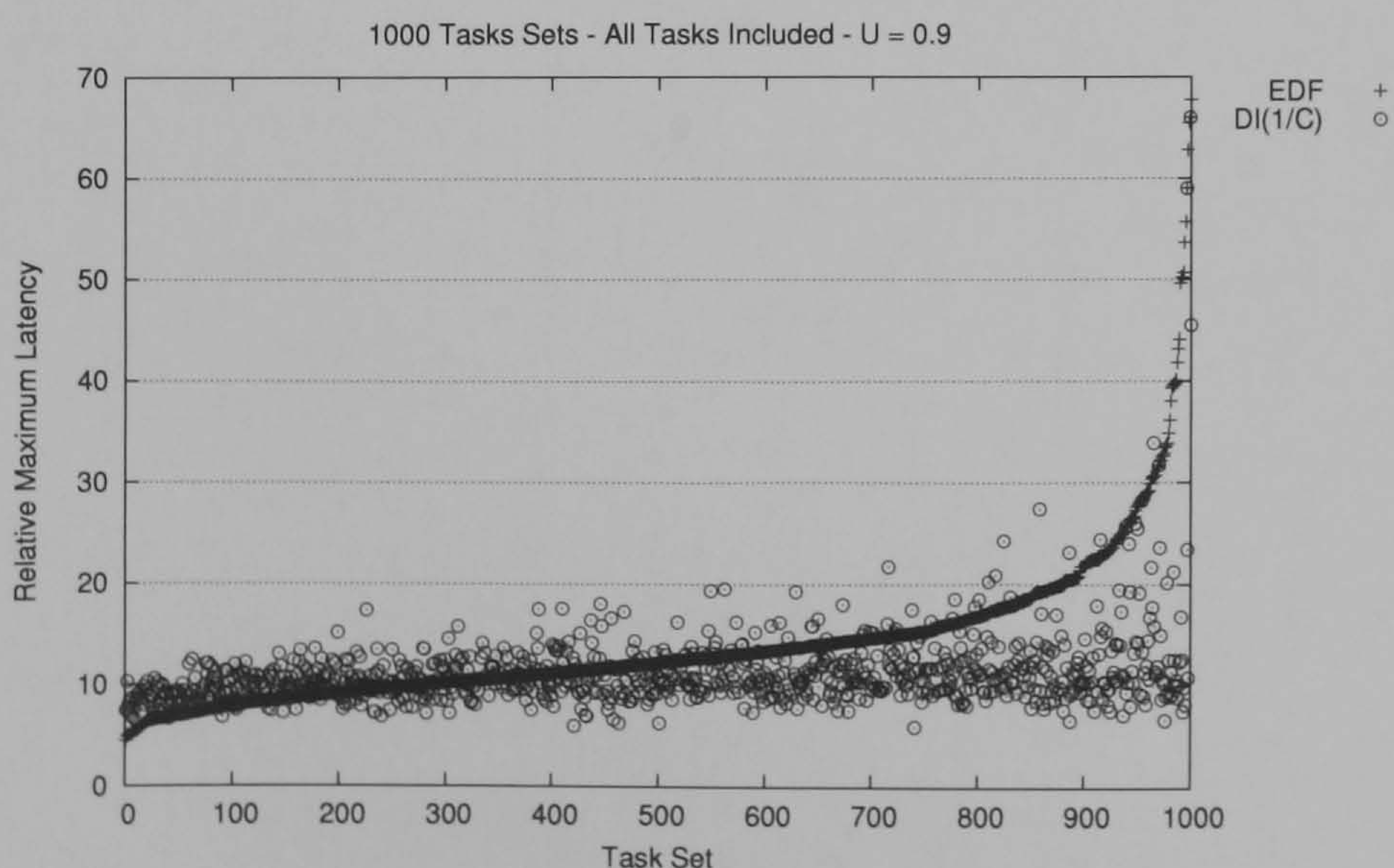
Figure 7.25: PLOTS B TYPE. Guaranteeing Deadlines and Minimising the Relative Maximum Latency



(a) On average the relative maximum latency is 9.06 for EDF and 4.16 for DI(1/C)



(b) On average the relative maximum latency is 9.52 for EDF and 5.74 for DI(1/C)



(c) On average the relative maximum latency is 14.03 for EDF and 11.42 for DI(1/C)

Figure 7.26: PLOTS C TYPE. Comparing DI(1/C) against EDF for the Deadlines and the Relative Maximum Latency problem. Observe the excellent performance of DI(1/C)

7.4.6 Problem: Deadlines and Average Response-Time

Using the 1/C and T/C rules for assigning importance, our experiment compares how good the DI algorithm is for minimising the maximum relative average response-time. The results are depicted in Figures 7.28, 7.28 and 7.29 (pages 160, 161 and 162).

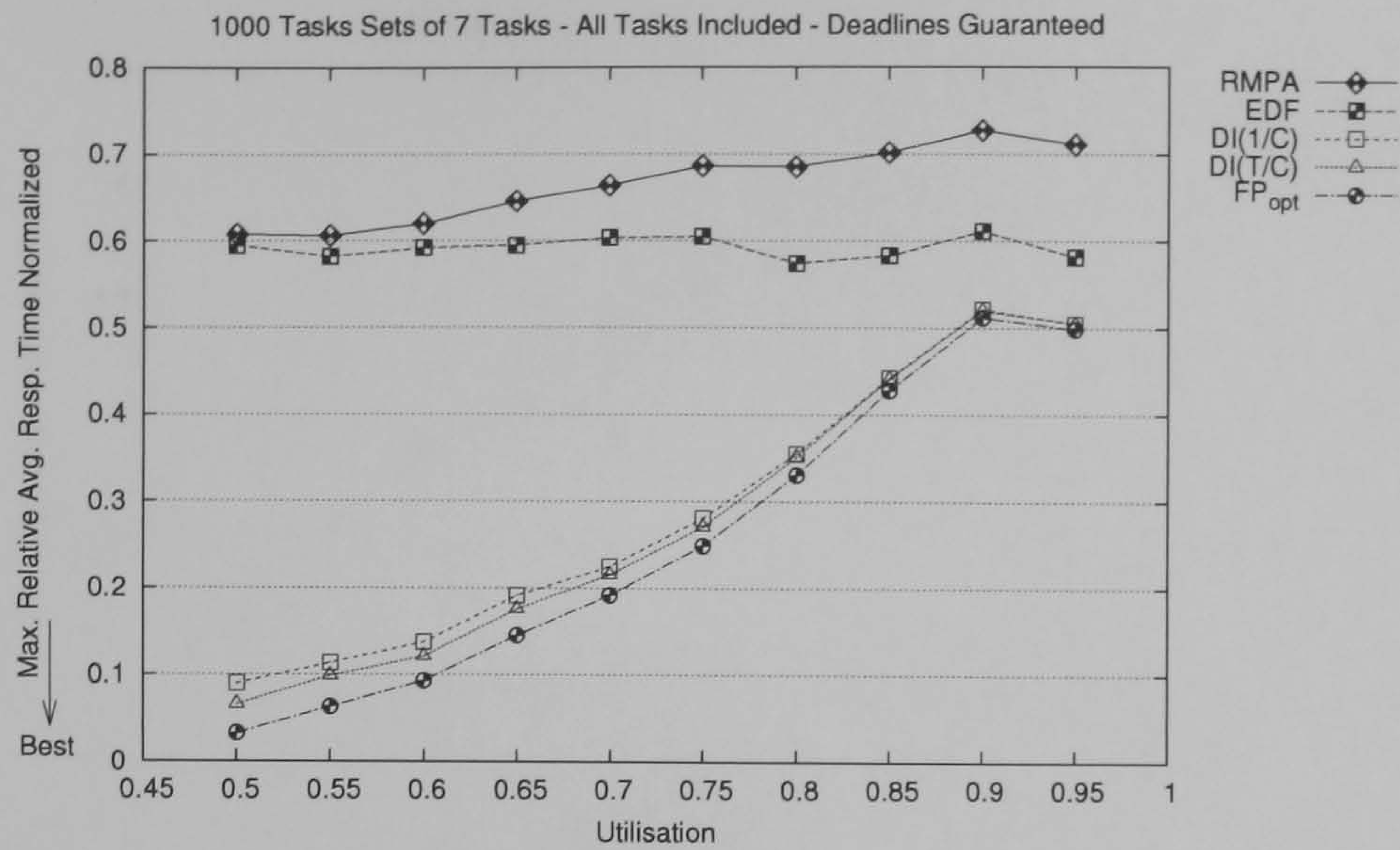
In Figure 7.27, both DI(1/C) and DI(T/C) are near-optimal solutions outperforming EDF in the three plots for all utilisation. DI(T/C) is slightly better than DI(1/C).

Figure 7.25 confirms that both DI(1/C) and DI(T/C) provide excellent solutions for this particular problem.

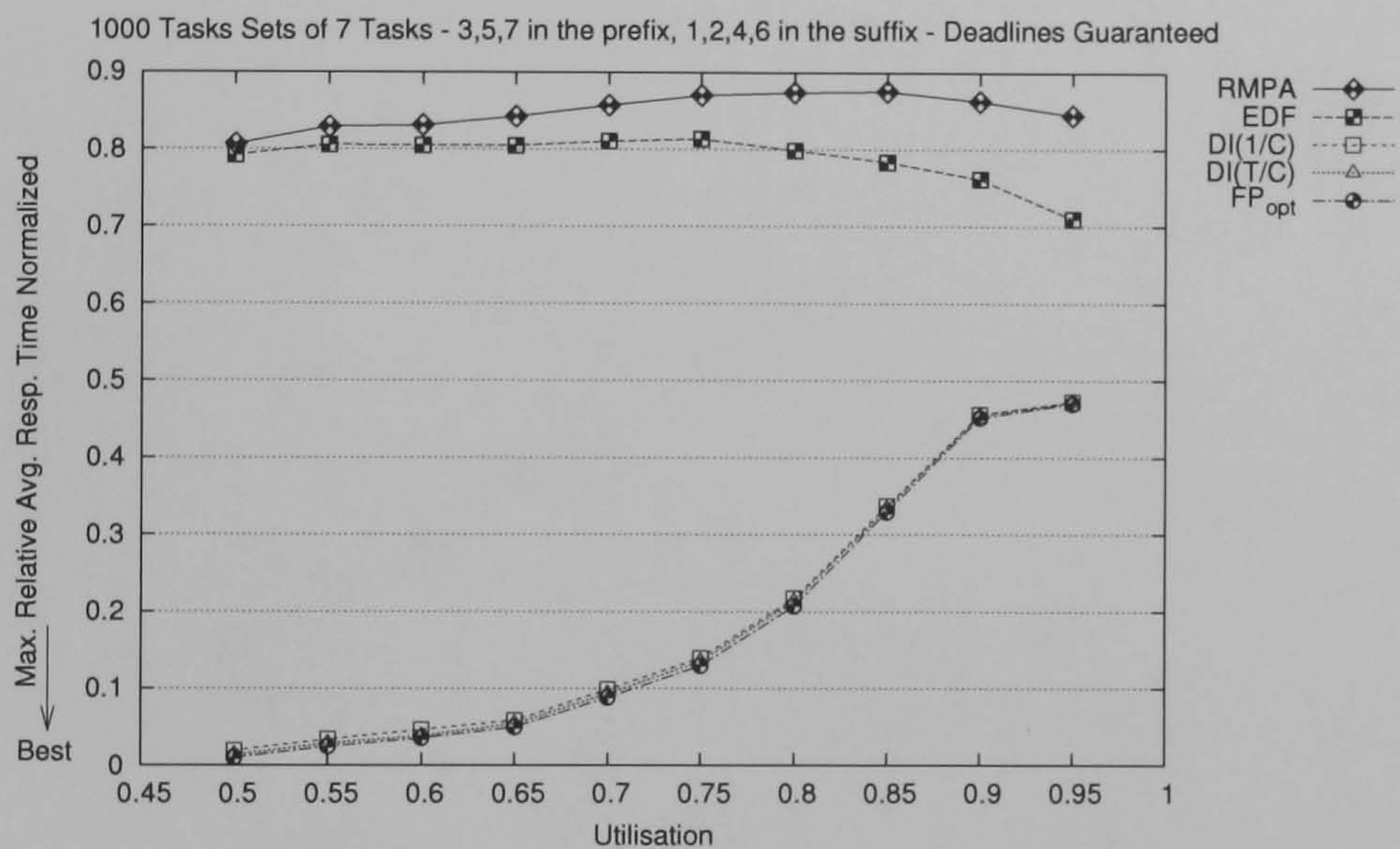
Figure 7.28, confirms that both DI(1/C) and DI(T/C) are provide excellent solutions for this particular problem. Although almost overlapped, DI(T/C) is still better than DI(1/C).

The superiority of DI(T/C) with respect to EDF is confirmed in Figure 7.29. In the three plots, observe how the stability of the DI(T/C) solution contrasts with the EDF solution.

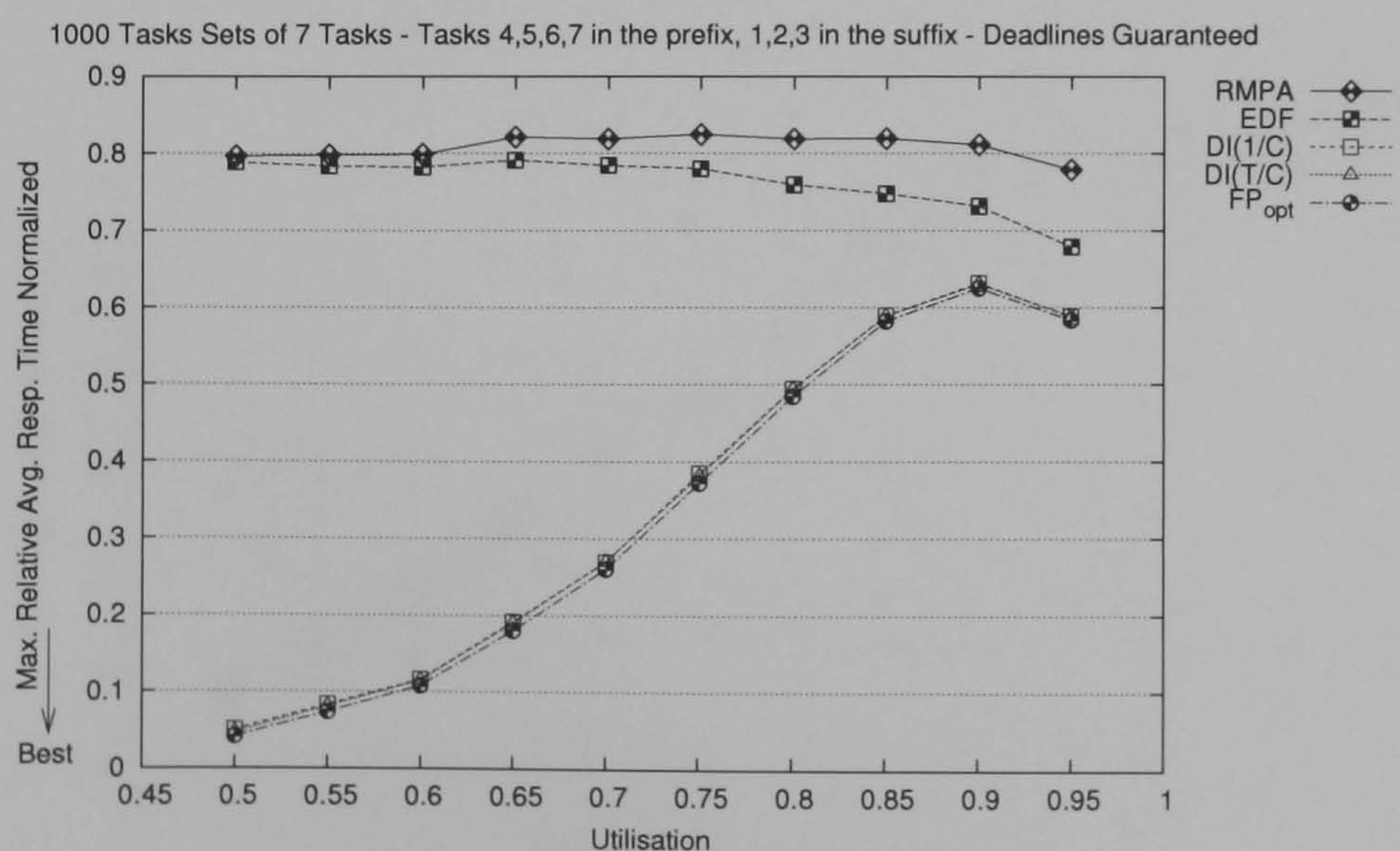
Remark 7.4.6. *The experiments show that the DI algorithm with importance assigned according to the rule "the shorter computation time, the higher the importance" provide near-optimal solutions for the scheduling problem of finding a feasible fixed-priority assignment that minimises the relative maximum latency.*



(a) Both DI(1/C) and DI(T/C) are near-optimal solutions

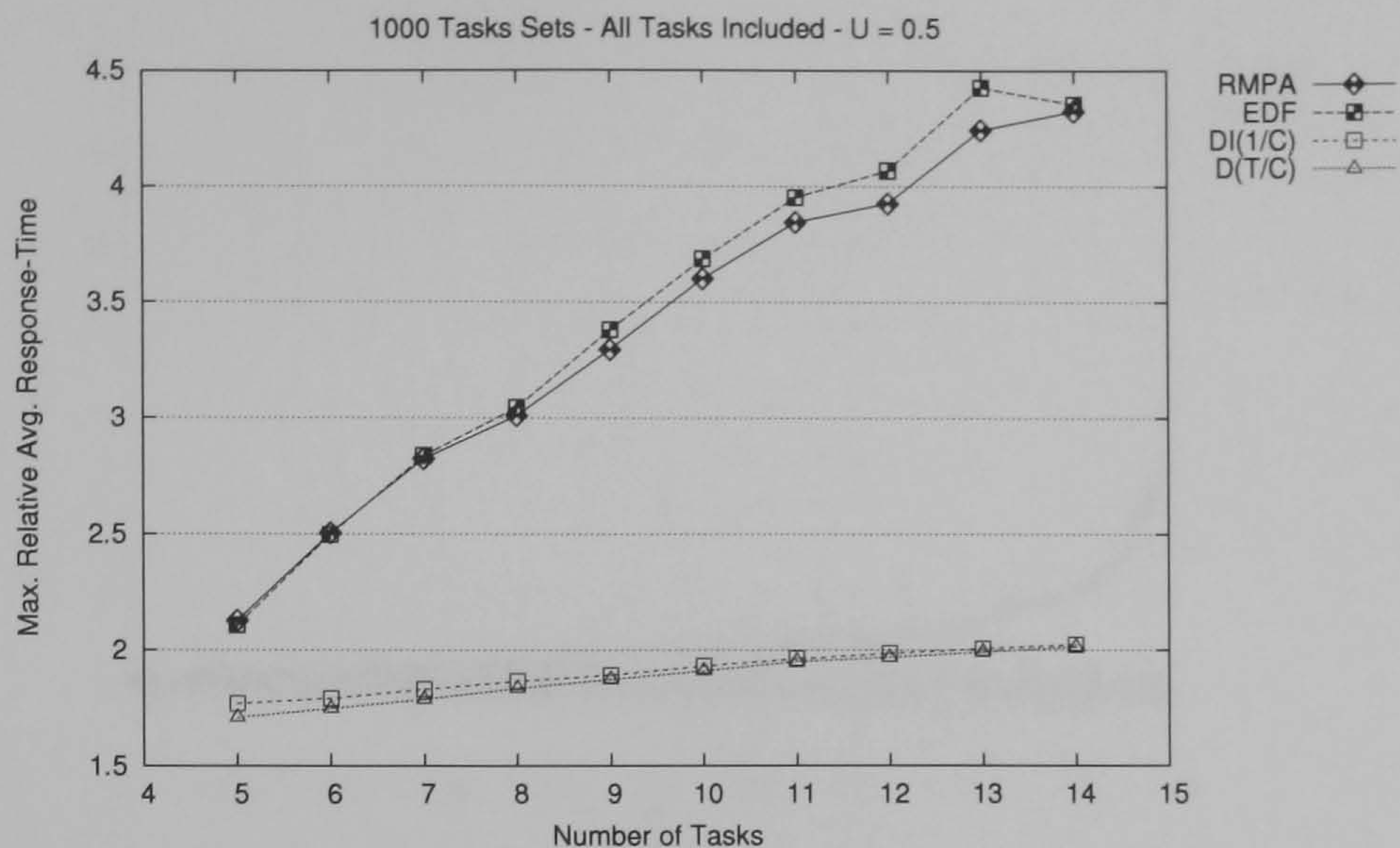


(b) Both DI(1/C) and DI(T/C) are near-optimal solutions

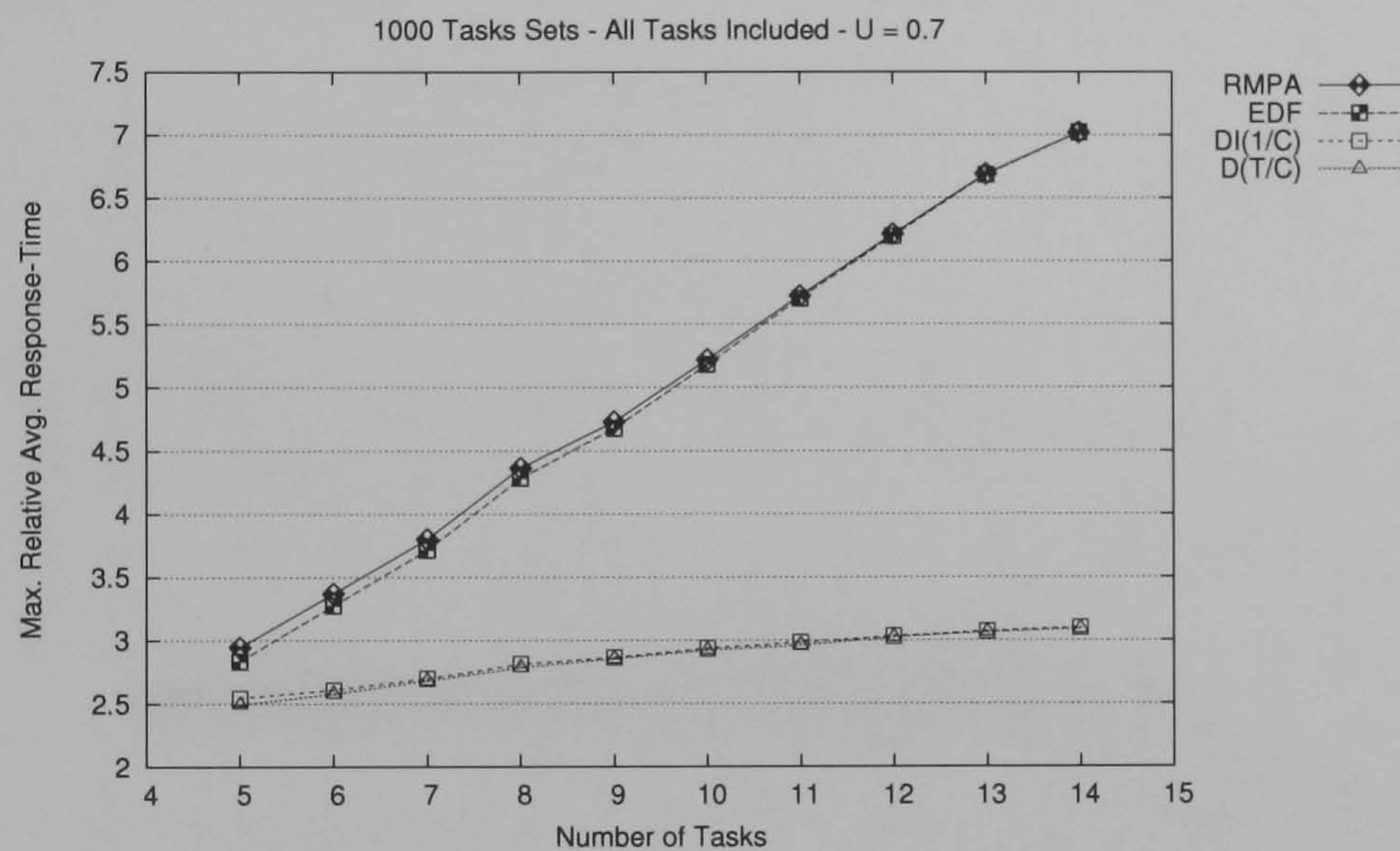


(c) Both DI(1/C) and DI(T/C) are near-optimal solutions

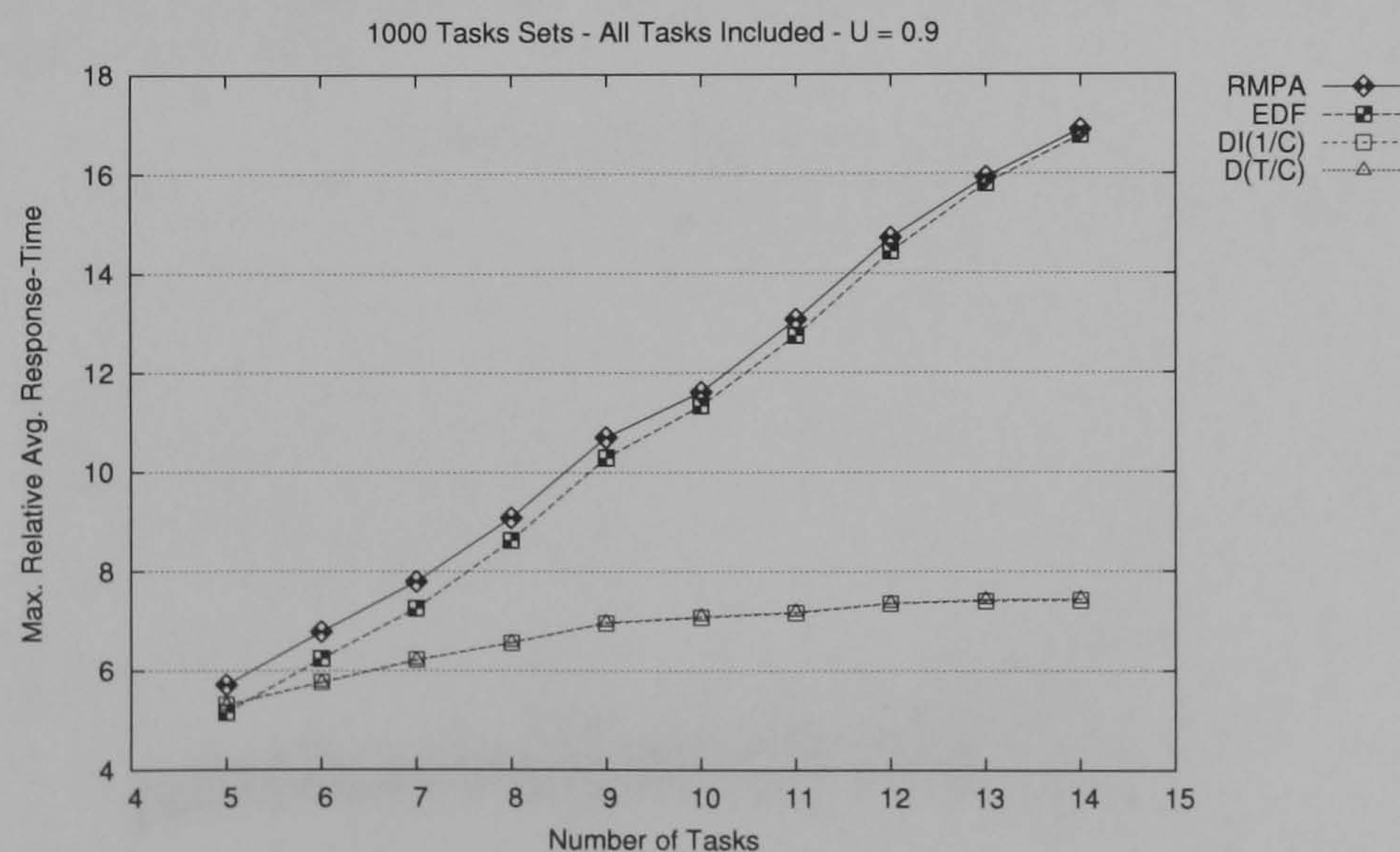
Figure 7.27: PLOTS A TYPE. Guaranteeing Deadlines and Minimising the Maximum Relative Average Response-Time. In all cases, both DI(1/C) and DI(T/C) are near-optimal solutions and outperform EDF



(a) Both DI(1/C) and DI(T/C) are excellent solutions

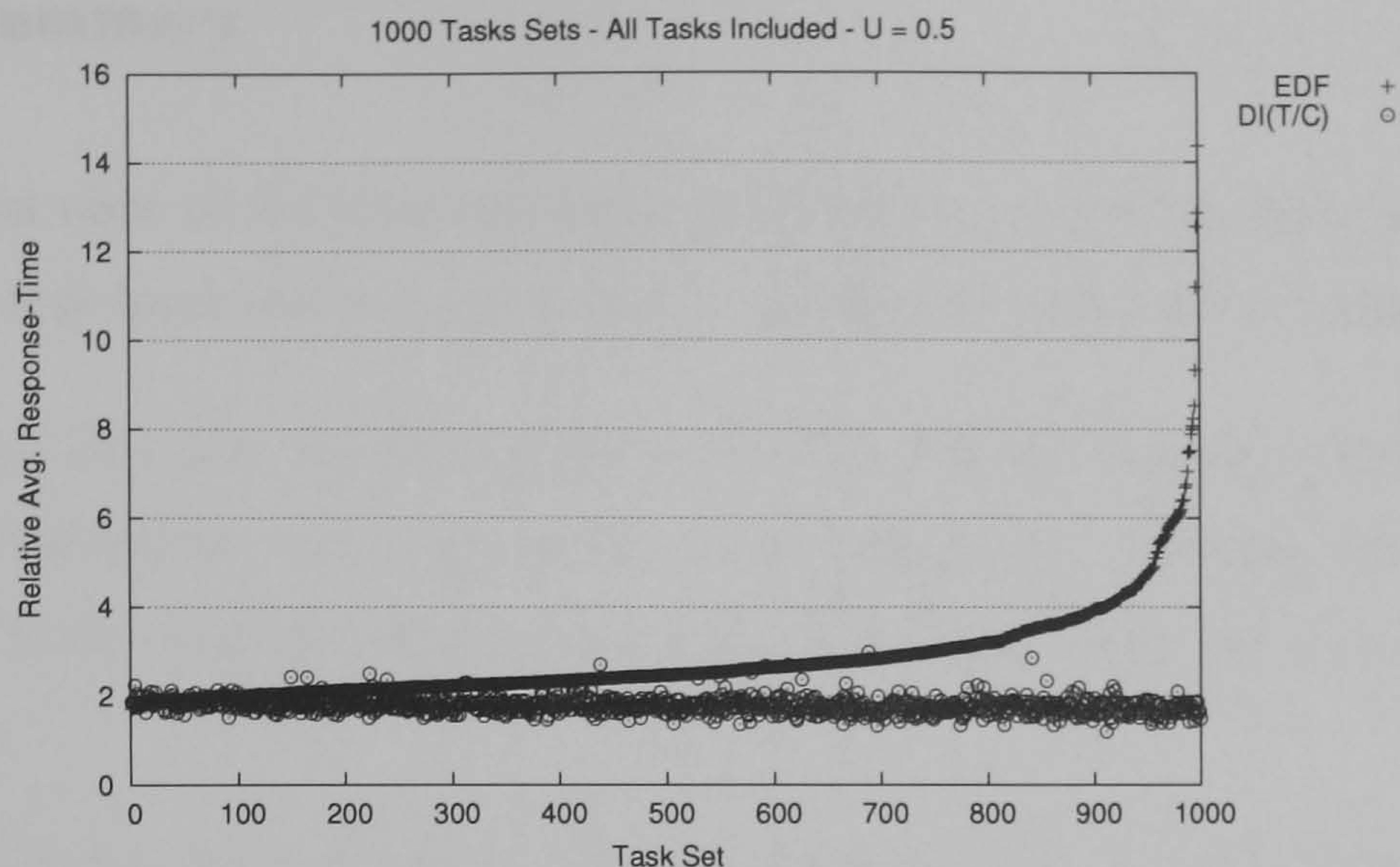


(b) Both DI(1/C) and DI(T/C) are excellent solutions

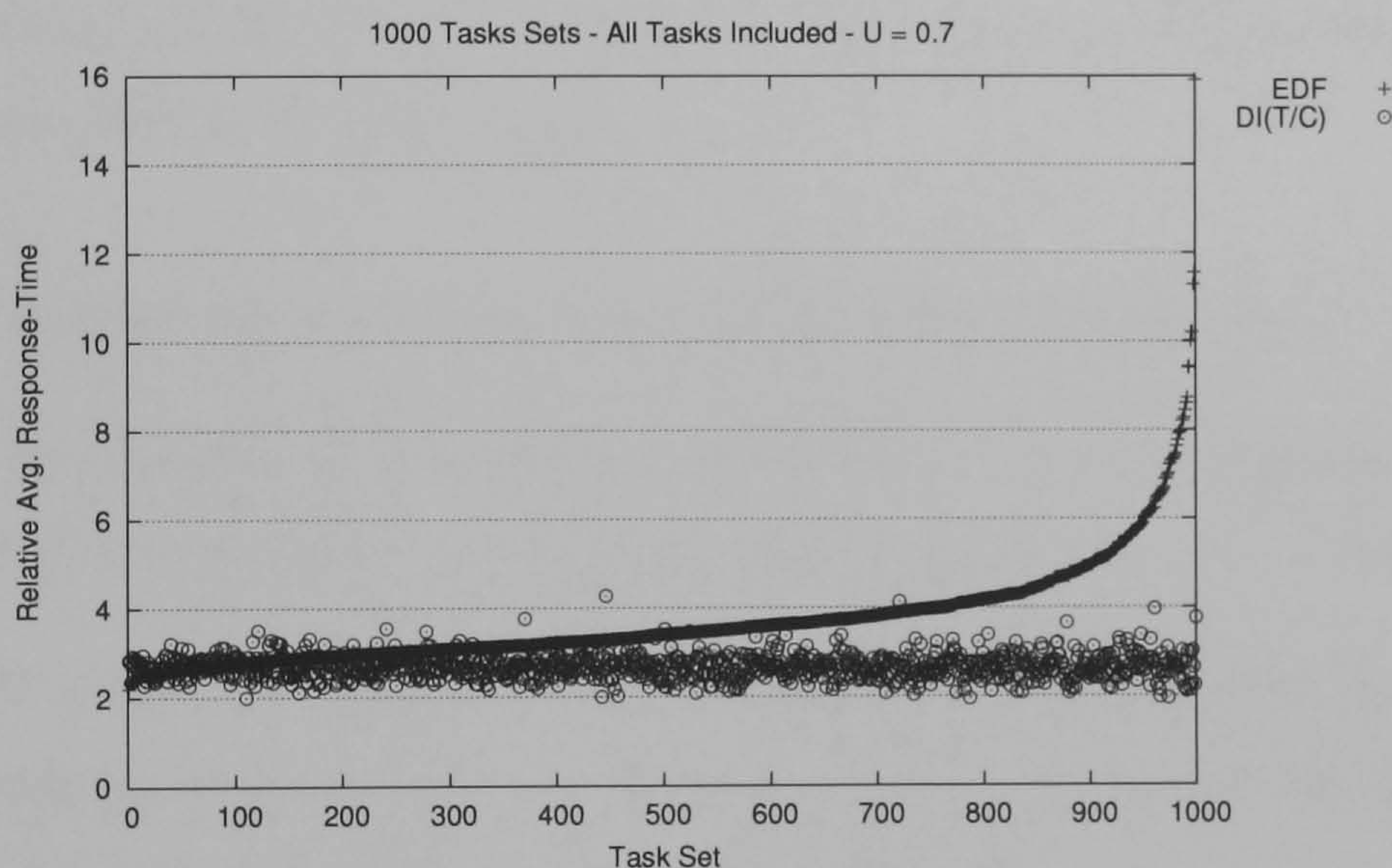


(c) Both DI(1/C) and DI(T/C) are excellent solutions

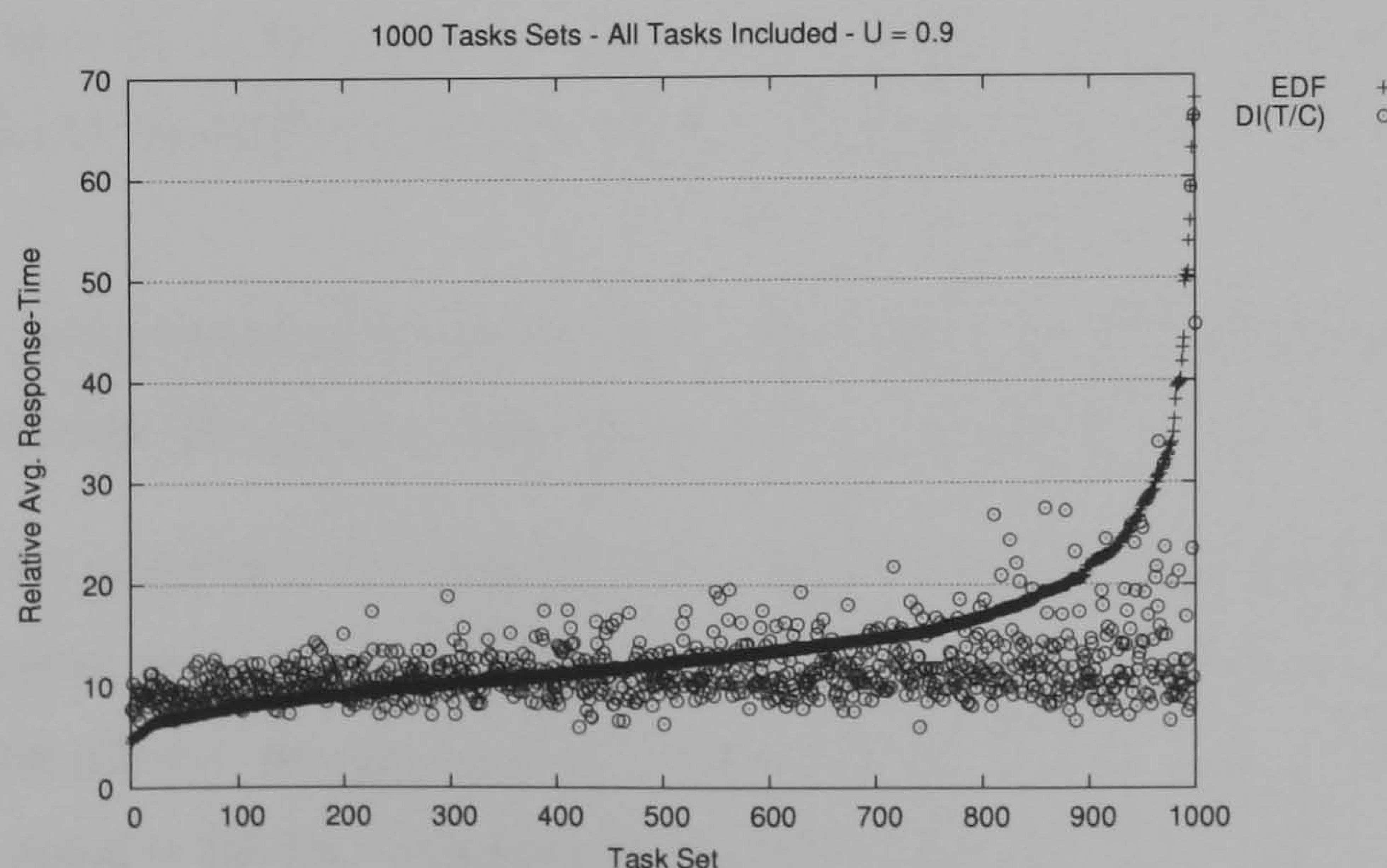
Figure 7.28: PLOTS B TYPE. Guaranteeing Deadlines and Minimising the Maximum Relative Average Response-Time



(a) On average the maximum relative average response-time is 2.82 for EDF and 1.78 for DI(T/C)



(b) On average the maximum relative average response-time is 3.72 for EDF and 2.68 for DI(T/C)



(c) On average the maximum relative average response-time is 7.28 for EDF and 6.21 for DI(T/C)

Figure 7.29: PLOTS C TYPE. Comparing DI(T/C) against EDF for the Deadlines and the Maximum Relative Average Response-Time problem. Observe the excellent performance of DI(T/C)

7.4.7 Summary

Observe that none of the solutions evaluated achieves the best possible performance with either global or local metrics; i.e. none hits the optimal computed by exhaustive search.

In almost all cases, the DI algorithm with any rule for assigning importance exhibits excellent performance as a local metric. This is because, in general, the impact is more related to the fact that the priorities are raised rather than the rules used for assigning the importance.

Table 7.2 presents a summary of the results of the evaluation (as a global metric) of the different FPS solutions to bicriteria problems using the DI algorithm compared with the solutions obtained with RMPA and EDF. The best results are indicated as 1s in the table and the worst ones as 4s. Observe that:

- EDF outperforms RMPA in all scheduling problems evaluated.
- EDF outperforms our DI solutions for problems where the secondary objectives are either to minimise the relative output jitter or to minimise the maximum latency.
- The DI solutions outperform EDF for problems where the secondary objectives are to minimise the total number of preemptions, absolute output jitter, the relative maximum latency, or the relative average response-time.

Note that some results are good with respect to more than one QoS criterion and therefore they can be considered as multicriteria scheduling solutions. In this sense:

- EDF outperforms our DI solutions for problems where the secondary objectives are to minimise the relative output jitter and the maximum latency.
- DI(1/C) is our best FPS solution when deadlines have to be met and the secondary objectives are to minimise the absolute output jitter, the relative maximum latency and the relative average response-time (note that in this last problem DI(1/C) is almost equal to DI(T/C) which is rated as number 1). In other words, DI(1/C) provides an excellent solution for a scheduling problem with four objectives.

	The DI Algorithm with Heuristics						
	$1/C$	*LC*	*LT*	T/C	C/T	*RMPA*	*EDF*
Preemptions		1			2	4	3
Absolute Output Jitter	1			3		4	2
Relative Output Jitter	3				4	2	1
Maximum Latency		2			3	4	1
Rel. Maximum Latency	1			2		4	3
Rel. Avg. Response-Time	2			1		4	3

Table 7.2: Classification according to the performance. 1 is the best and 4 the worst scheduling solution.

- DI(LC) is our best solution when deadlines have to be met and the secondary objective is to minimise the total number of preemptions. This solution is also better than RMPA when the maximum latency is included into the objectives. Consequently, DI(LC) provides an excellent solution for a scheduling problem with three objectives.

- DI(T/C) is an excellent solution when deadlines have to be met and the secondary objectives are to minimise the relative average response-time and the relative maximum latency.

7.5 DI Algorithm vs Swapping Algorithm

In this section the DI algorithm is compared against the swapping algorithm in two scenarios:

- when the scheduling problem consists in finding a feasible priority assignment that maximises the importance;

- when the scheduling problem consists in finding a feasible priority assignment that maximises a QoS metric.

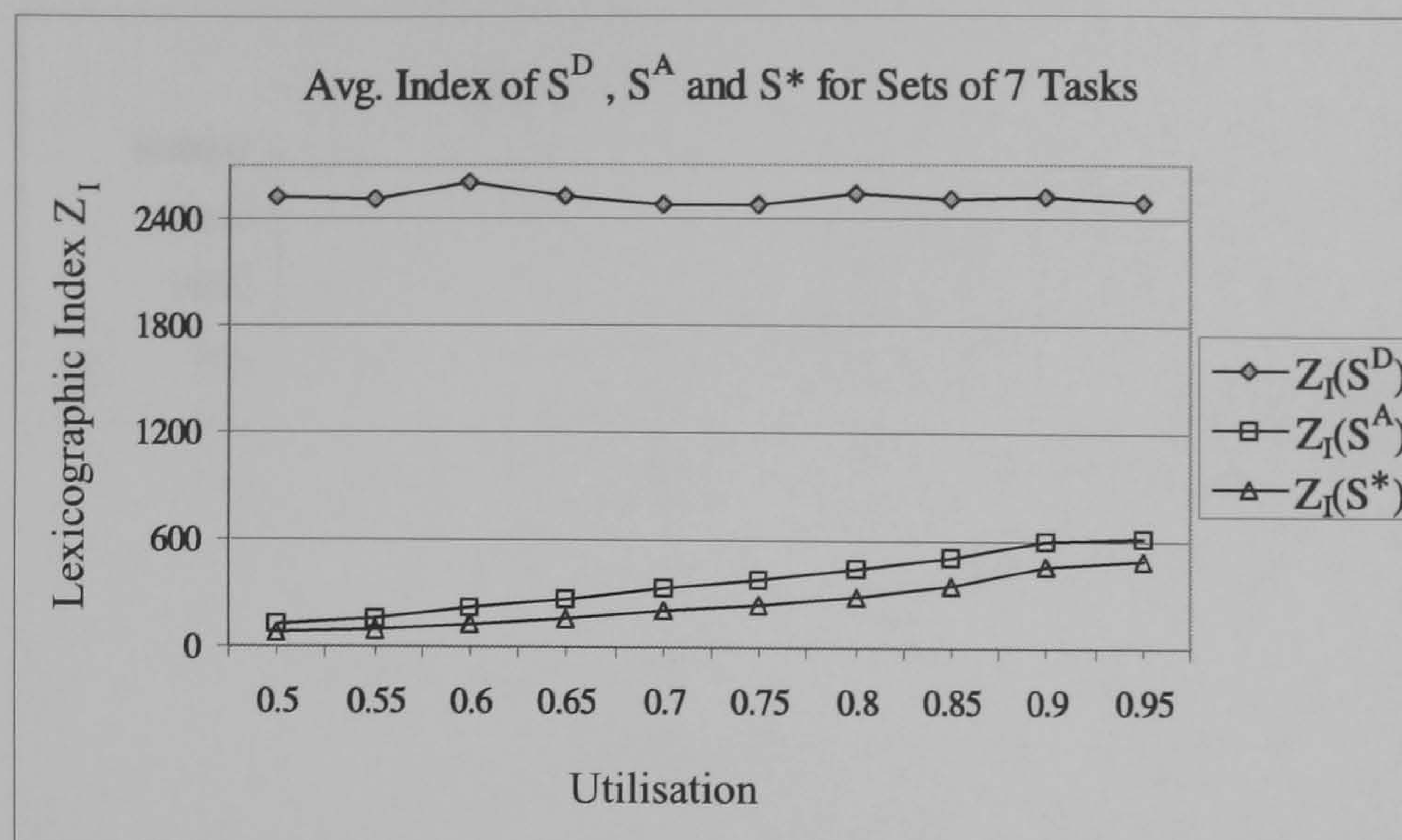


Figure 7.30: For high utilisations, the number of feasible tasks sets decreases and therefore, the distance between S^D and S^* gets smaller. The distance between S^A and S^* increases slightly.

7.5.1 Maximising the Importance Metric

This section compares the DI algorithm against the swapping algorithm when the problem consists in finding a feasible priority assignment that maximises the importance. The swapping algorithm receives an infeasible ordering and finds a feasible one (if it exists) by swapping pairs of task priorities. In our context, the swapping algorithm will receive an infeasible ordering by importance and will produce a feasible one.

We have generated random task sets of N tasks with utilisation from 0.5 to 0.95 and $N = 4, 5, 6, 7, 8$ tasks. Each task set is created by randomly choosing task's computation times between 2 and 50 time units, and then by randomly choosing periods to approximate the utilisation desired. The importance is assigned according to the rule "the larger the deadline, the higher the importance"; note that this is an extreme case where the importance of tasks is contrary to the RMPA order. In addition, it is guaranteed that all task sets are feasible under rate monotonic scheduling.

Let S be a task set with randomly chosen importances and with priority orderings S^I (by importance), S^D (by RMPA), S^A (by swapping algorithm) and S^* (by DI algorithm); in terms of the lexicographic order, the goodness of the orderings is given by the index metric Z_I. For each set, we compute both their orderings and their indices. A point in Figure 7.30 represents the average index of 1000 sets of 7 tasks for a level of utilisation.

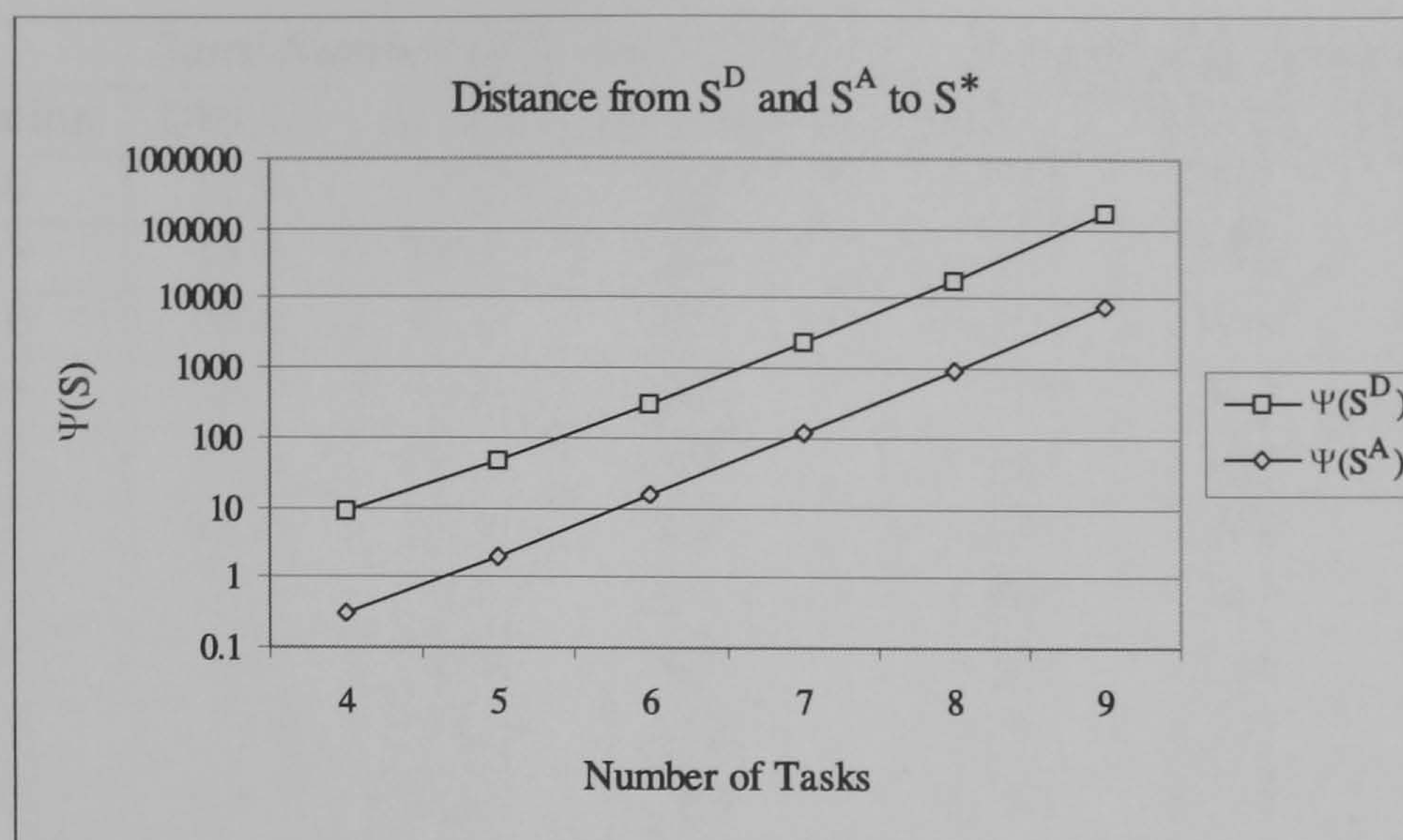


Figure 7.31: The solutions S^D and S^A move dramatically away from S^* conforming the number of tasks increases.

For high utilisation the number of feasible task sets decreases, and hence the difference between RMPA and DI is reduced. On the other hand, the difference against S^A grows slightly. Note that by the optimality of DI, it is never the case that the index of S^D or S^A can be lower than S^*.

The plots for task sets with N equal to 4, 5, 6 are not depicted because they are similar to Figure 7.30. More interesting is to show how fast the solutions S^D or S^A move away from the optimal S^*. Consider the data in Figure 7.30; computing the distances $Z_I(S^D) - Z_I(S^*)$ and $Z_I(S^A) - Z_I(S^*)$ for their respective utilisation and calculating its average give us the average distance $\Psi(S^D)$ and $\Psi(S^A)$ respectively. Computing these distances for different N give us Figure 7.31; this is the variation of the distance to S^* with respect the number of tasks per set. Note how fast they move away from S^*.

7.5.2 Maximising/Minimising a QoS Metric

The swapping algorithm receives an infeasible ordering by importance and produces a feasible one; for instance, assuming that the rule 1/C produces an infeasible ordering, the swapping algorithm produces a feasible one denoted as A(1/C). In the experiments presented in section 7.4, the swapping algorithm was also included but the results were not shown since they were almost the same as those obtained with the DI algorithm and hence the lines are overlapped. It has an explanation.

	Total Number of Preemptions				*Relative Avg. Resp-Time*		
Utilisation	DI(LC)	A(LC)	Identical		DI(T/C)	S(T/C)	Identical
0.50	32.4	32.5	924		1.787	1.791	953
0.55	38.4	38.4	904		1.954	1.960	944
0.60	44.0	44.1	892		2.134	2.139	925
0.65	49.7	49.8	891		2.399	2.403	912
0.70	58.1	58.2	855		2.682	2.689	896
0.75	66.5	66.5	842		3.075	3.084	890
0.80	75.3	75.4	838		3.662	3.668	913
0.85	89.9	90.0	799		4.543	4.545	920
0.90	113.0	113.1	759		6.211	6.213	926
0.95	141.5	141.6	725		9.245	9.244	953

Table 7.3: Comparing solutions obtained with DI and swapping algorithms for different utilisation with heuristics LC and T/C. Note how similar the results are. From 1000 priority orderings, the column "Identical" shows the number of identical ones.

- Firstly, observing Figure 7.4(a) (page 124) note that when assigning priorities with the heuristics there already exist a number of feasible assignments such that DI and swapping algorithms results are necessarily the same.
- Secondly, we note that when assigning importance with the heuristics, higher importance tasks are feasible and therefore they remain practically fixed in their positions; only the lower priority ones are changed.
- Thirdly, there exist a number of priority orderings that produce identical solutions, and many of such orderings have minor differences.

Consequently, the results of the DI and the swapping algorithms will be either identical or different in some lower priority tasks. For such reasons both algorithms achieve similar solutions. So similar that when importance is assigned with the rule 1/C the DI algorithm and the swapping algorithm always produce the same priority ordering; i.e. DI(1/C) is identical to A(1/C). Consequently, A(1/C) is also an excellent multicriteria scheduling solution.

Table 7.3 shows the differences between DI(LC) and A(LC) as well as DI(T/C) and A(T/C) solutions. For instance, for utilisation 0.5 the average total number of preemptions for DI(LC) and A(LC) are 32.4 and 32.5 respectively. Thus DI(LC) is slightly better than A(LC). Observe that there are 924 identical priority orderings and therefore only 76

contributes to this difference. Figure 7.32 shows a better comparison between DI(LC) and A(LC). The baseline is formed by DI(LC); points above the baseline are worst and points below are best than DI(LC) results. Note that A(LC) is better in some occasions.

7.6 DI+ Algorithm: QoS and Priority Constraints

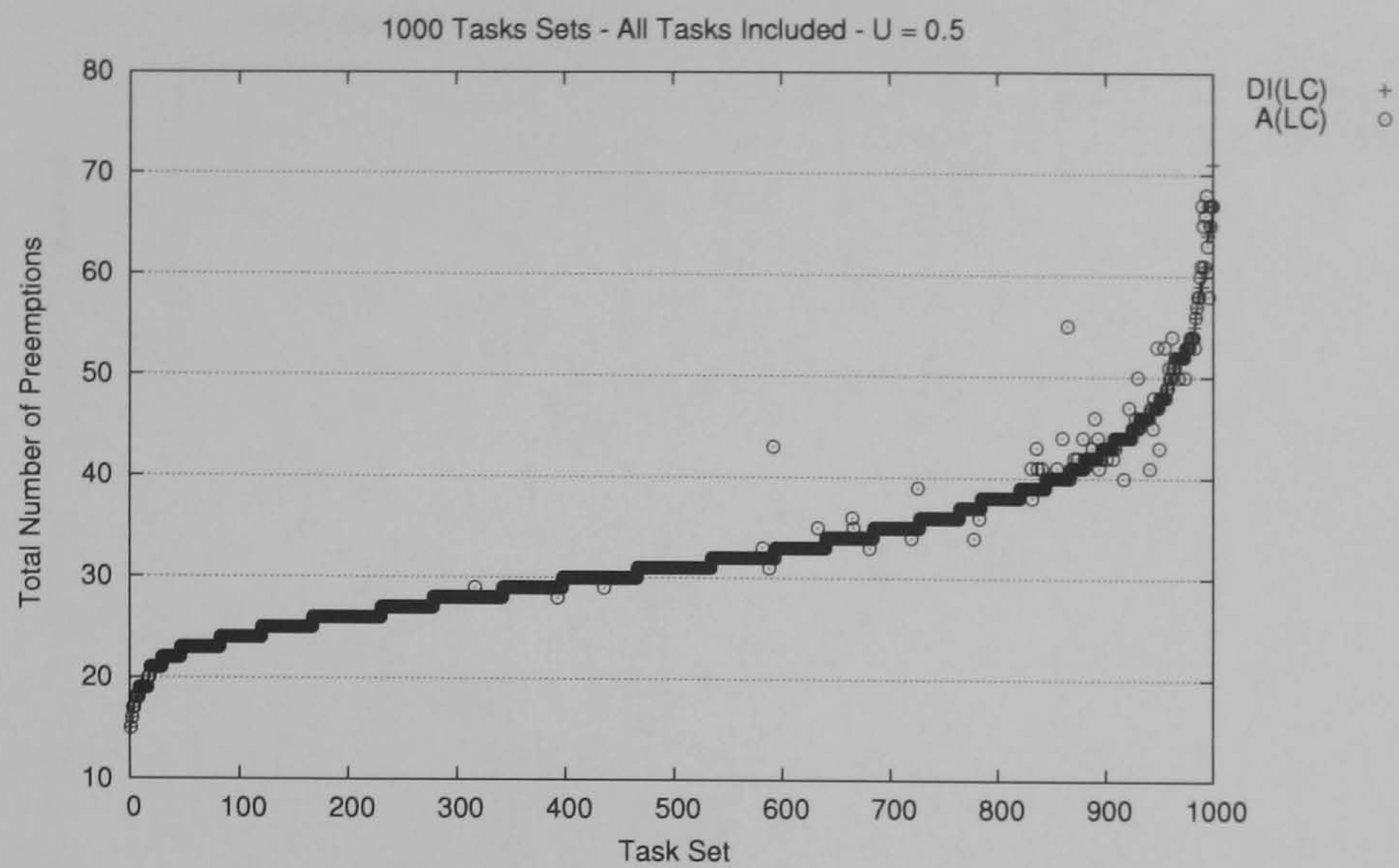
In this section we present a simple evaluation of the DI+ algorithm using the same task sets generated to evaluate DI. In order to compare DI+ against RMPA, the priority orderings must be both feasible and fulfill some priority constraints. By ordering tasks by RMPA, we label tasks as $1, 2, \ldots, 7$, and define the next priority constraints for all task sets: $P_7 > P_1$ and $P_6 > P_5$. *Thus, under RMPA all task sets fulfill both the deadlines and the priority constraints.* We selected these constraints arbitrarily and without a special reason.

Of the six different scheduling problems with QoS requirements, we only present plots for five of them since no DI solution outperforms RMPA for the problem with relative output jitter requirements. The results are shown in Figures 7.33 and 7.34 (pages 170-171). Only RMPA and DI+ solutions meet the priority constraints.

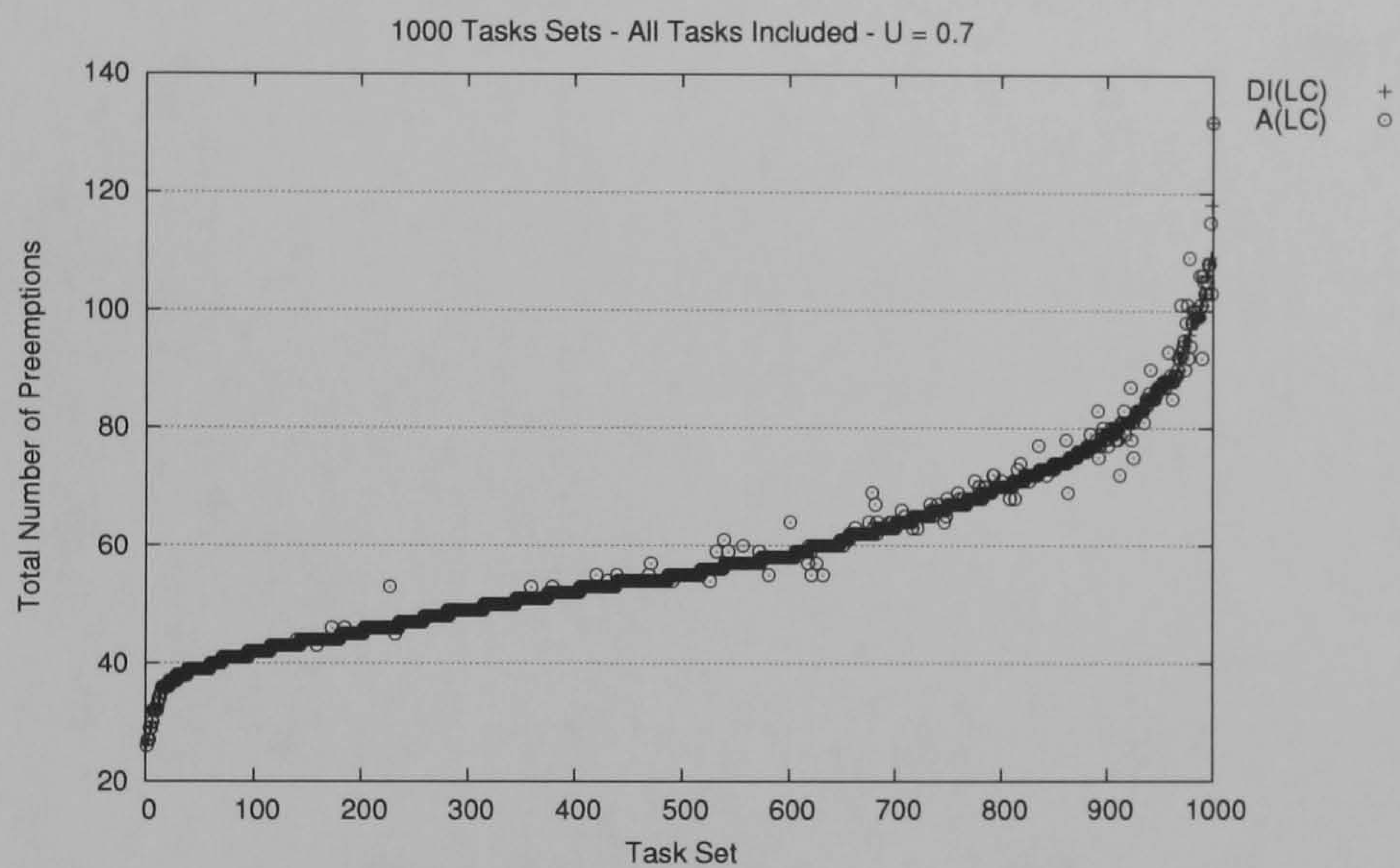
Observe that in all cases, constraining the freedom for assigning priorities reduces the chance for improving the QoS metric. Moreover, some results can be worse as shown in Figure 7.33(c), where for high utilisation the DI+(LC) solution is worse than RMPA for the deadlines and maximum latency problem. This is because for this particular problem, the difference between RMPA and the DI solutions is minimal for high utilisation (see Figure 7.21(a) in page 152). Consequently, by adding the constraints the DI+ solutions tend to be worse.

Note that while in some cases the differences are minimal (e.g. total number of preemptions problem), in other cases the differences are significant (e.g. relative maximum latency problem).

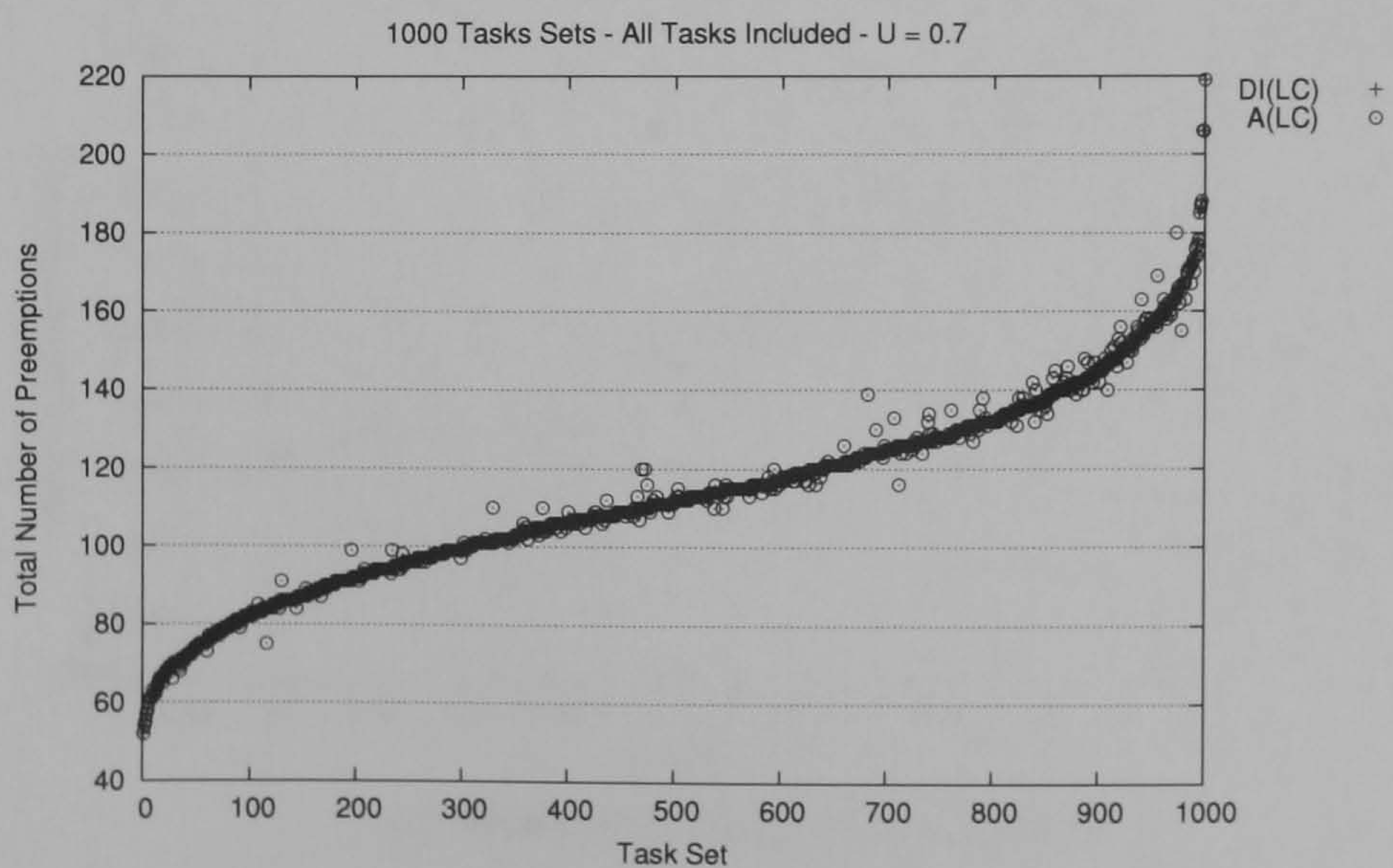
In summary, in this experiment we can observe that priority constraints simply reduce the space of search where the DI+ algorithm can search for a solution, but we can still obtain improvements in the QoS.



(a) The average are DI(LC)=32.4 and S(LC)=32.5. There are 924 identical solutions

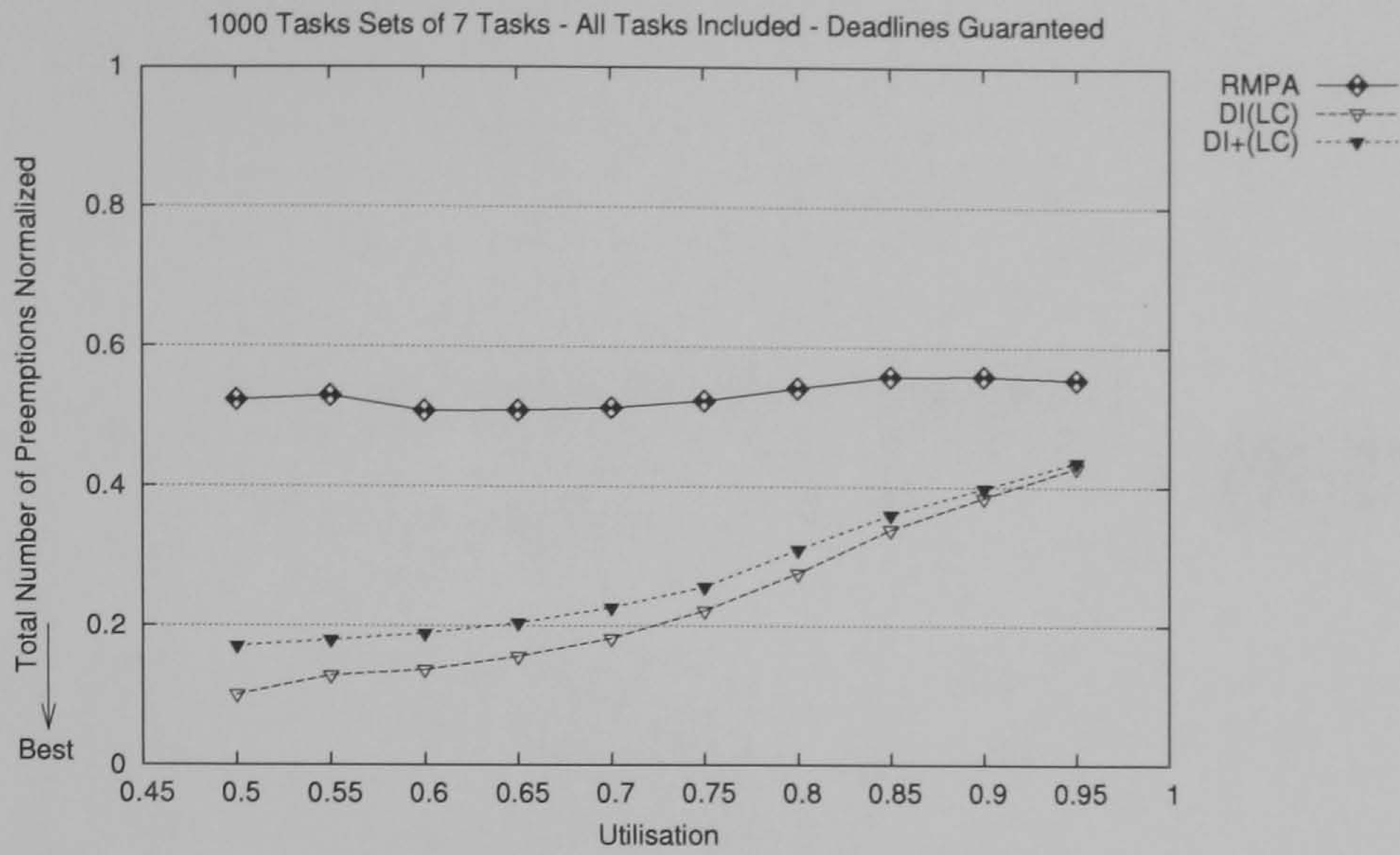


(b) The average are DI(LC)=58.1 and S(LC)=58.2. There are 855 identical solutions

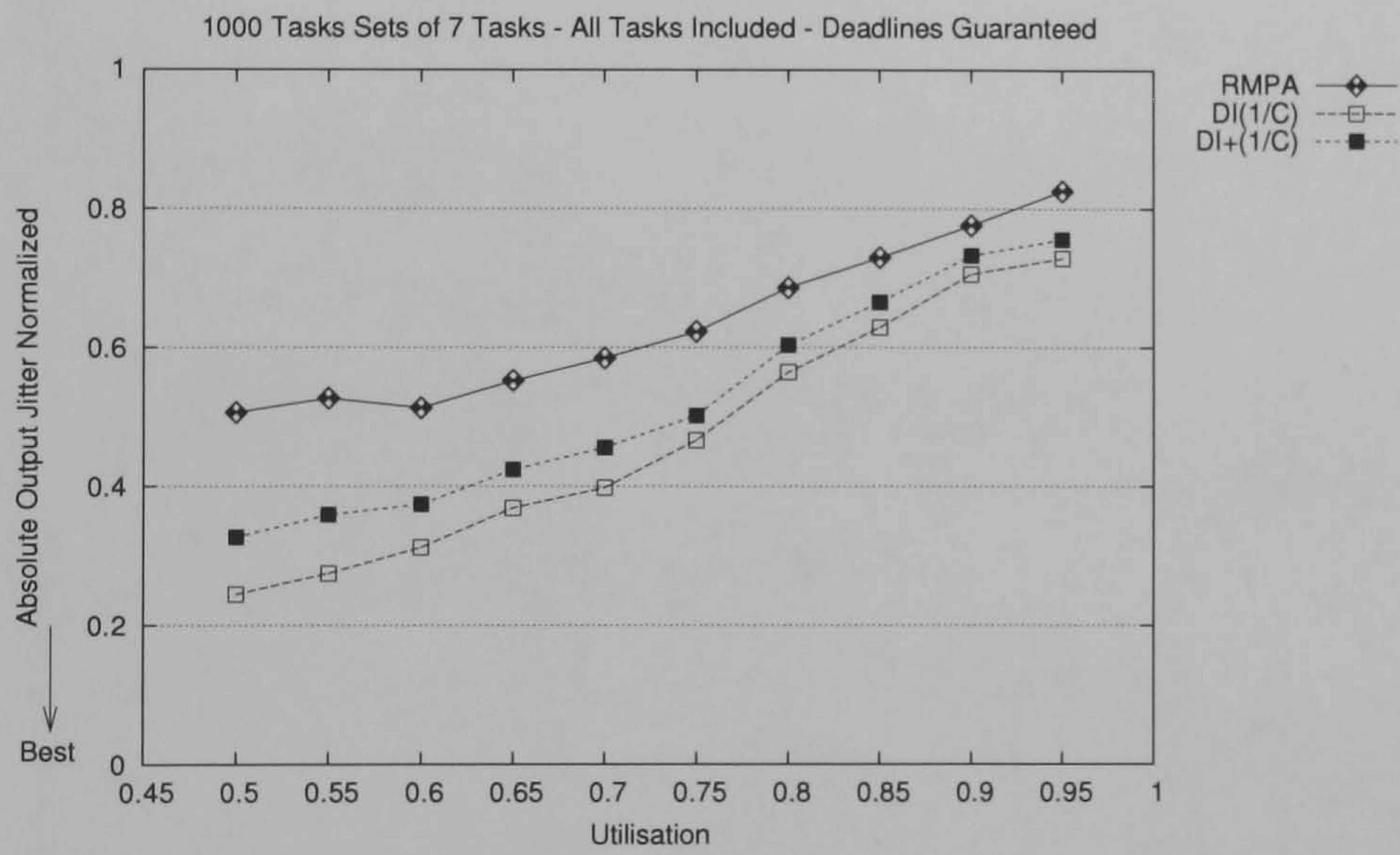


(c) The average are DI(LC)=113 and S(LC)=113.1. There are 759 identical solutions

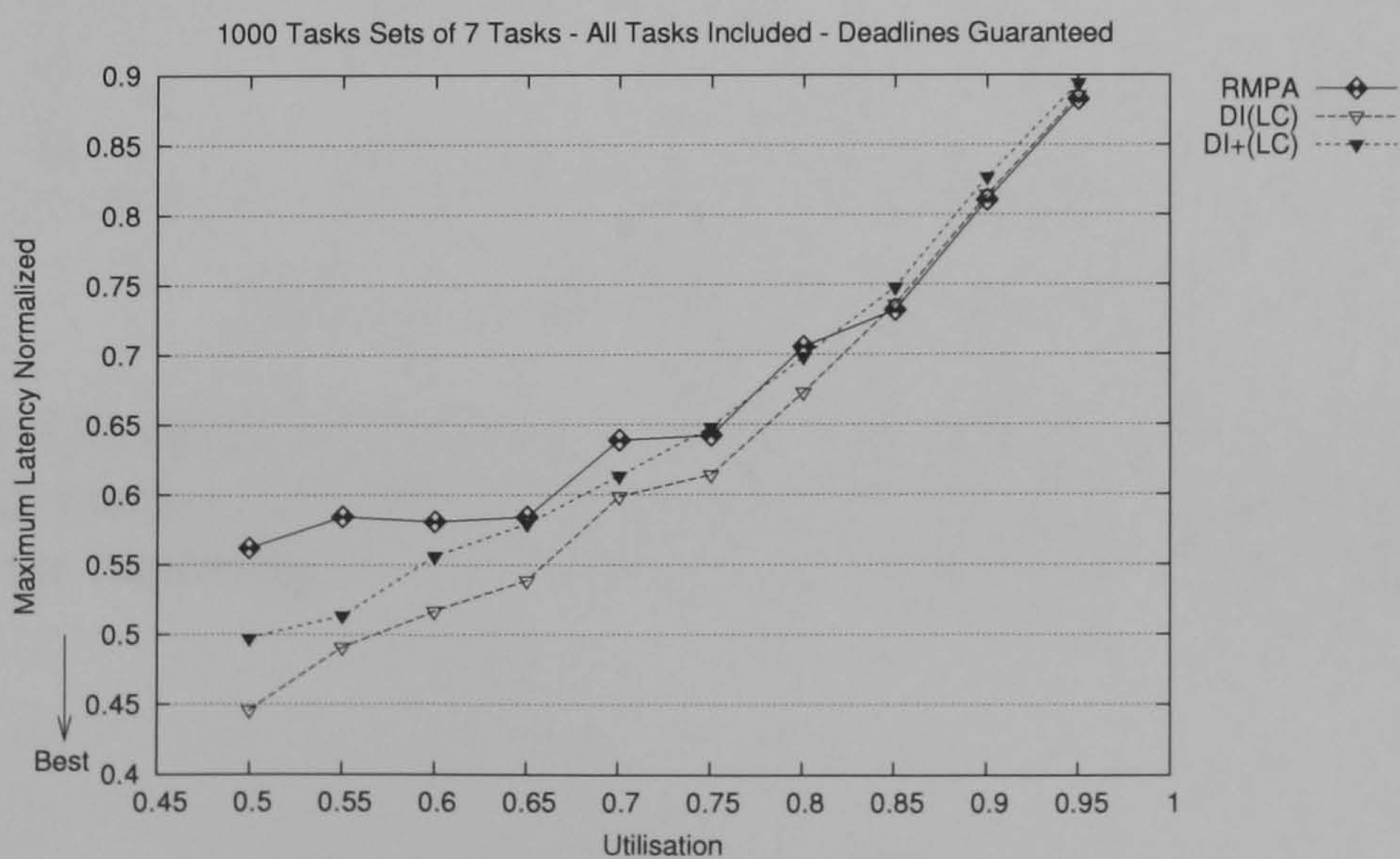
Figure 7.32: Comparing DI(LC) against A(LC) results for the problem of minimising the total number of preemptions



(a) Minimising the Total Number of Preemptions

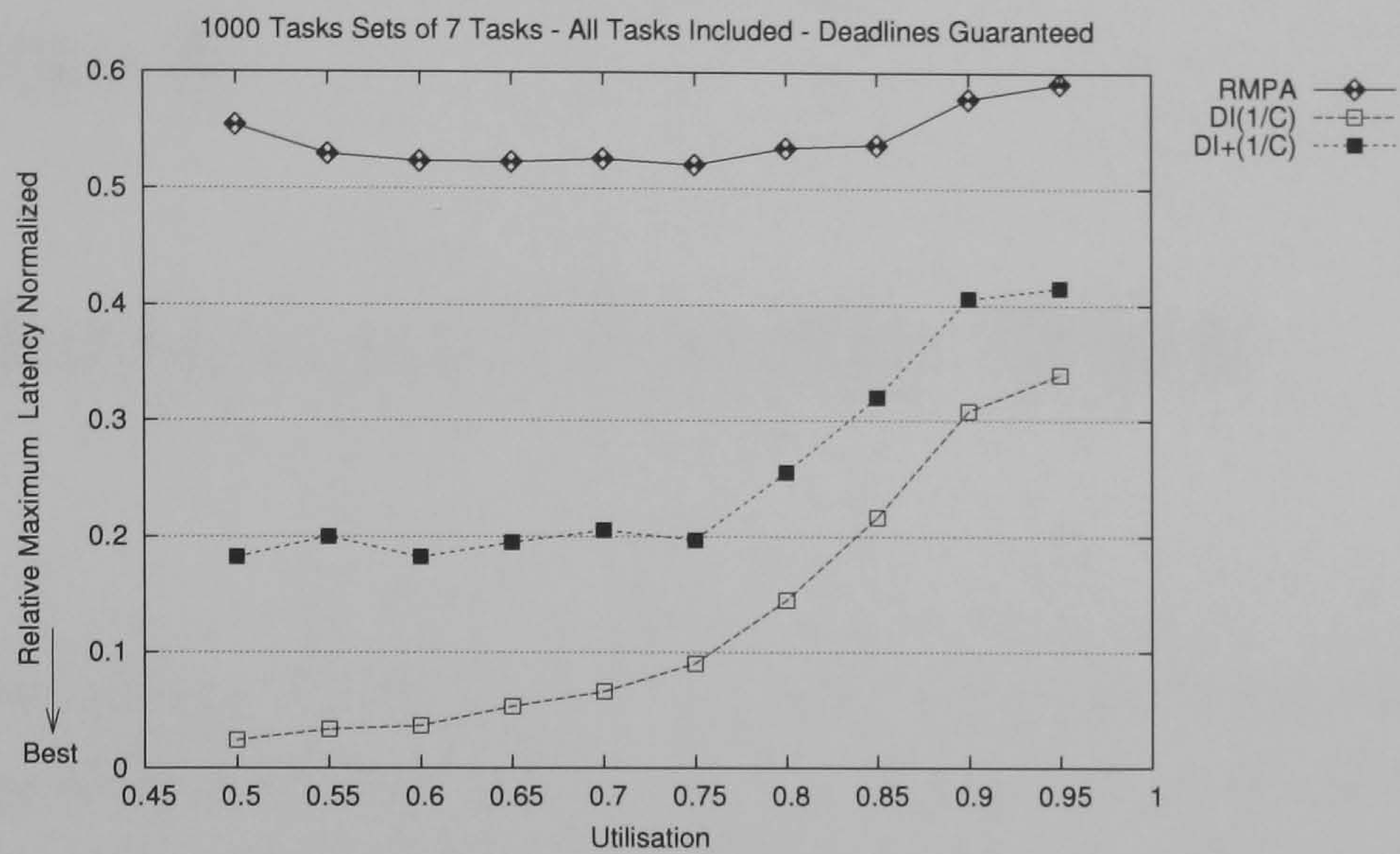


(b) Minimising the Absolute Output Jitter

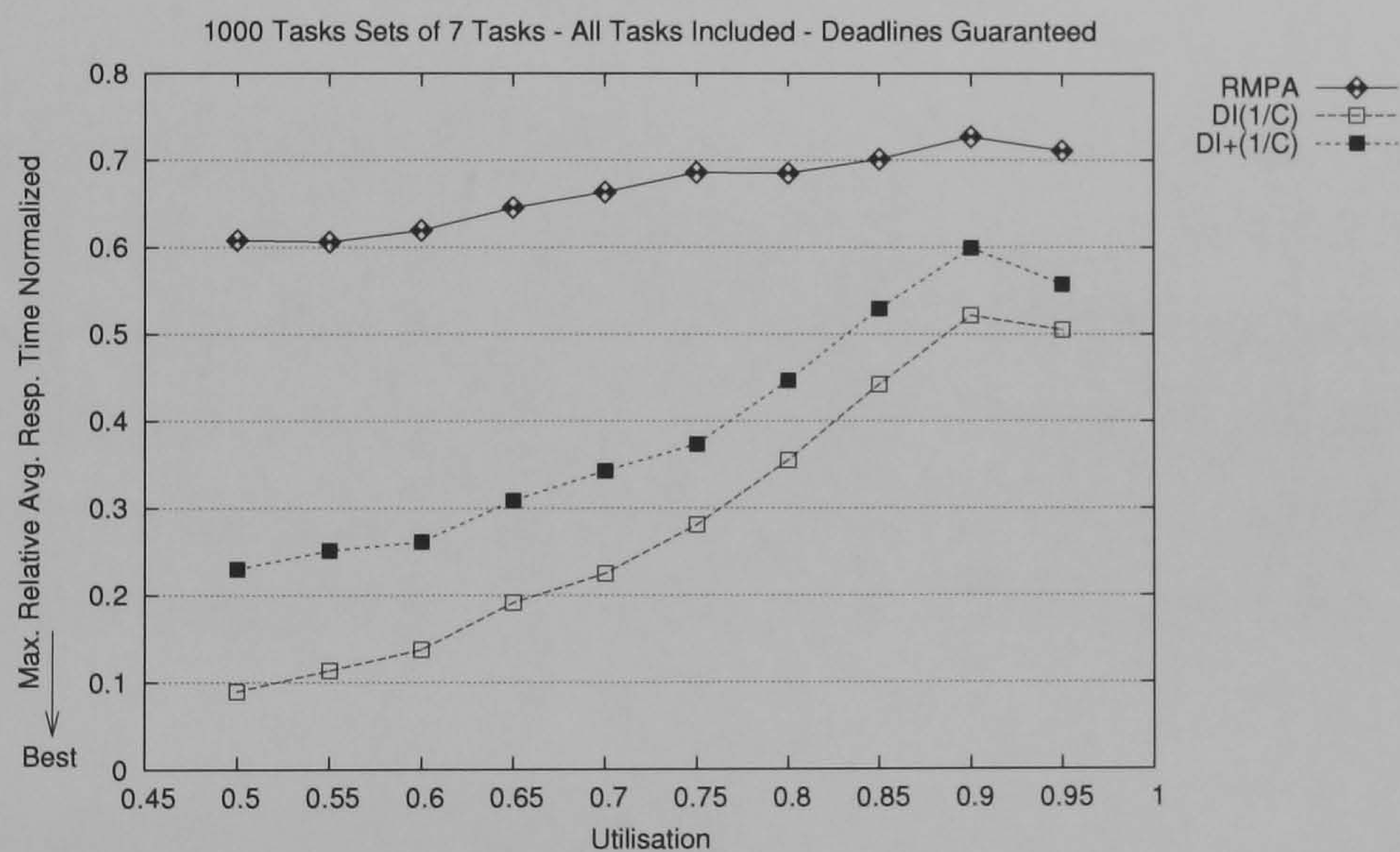


(c) Minimising the Maximum Latency

Figure 7.33: Guaranteeing Deadlines and Priority Constraints $\{P_7 > P_1, P_6 > P_5\}$ for different QoS metrics. Note that constraining the freedom for assigning priorities reduces the chance for improving the QoS metric. The DI solution does not meet the priority constraints



(a) Minimising the Relative Maximum Latency



(b) Minimising the Relative Average Response-Time

Figure 7.34: Guaranteeing Deadlines and Priority Constraints $\{P_7 > P_1, P_6 > P_5\}$ for different QoS metrics. Note that constraining the freedom for assigning priorities reduces the chance for improving the QoS metric. The DI solution does not meet the priority constraints

Chapter 8

Conclusions and Further Work

This thesis has shown that fixed-priority assignments that meet simultaneously deadlines and QoS requirements can be obtained without performing exhaustive search in the set of $N!$ possible assignments. Naturally, exhaustive search can be utilised but only for small task sets.

In the context of fixed-priority scheduling in hard real-time systems, problems where simultaneously deadlines and QoS requirements have to be fulfilled have been studied. While most approaches rely on complex algorithms that artificially modify the attributes or structure of tasks, and/or require non-standard run-time support, we propose an alternative mechanism based on pure assignment of fixed priorities. In this chapter we review the contributions of our research and present some directions for further work.

8.1 Review of Contributions

It is our thesis that optimal or near-optimal solutions to scheduling problems with multiple objectives can be obtained by finding assignments of fixed priorities. A solution consists of a feasible priority assignment that minimises a metric for measuring importance. The solution proposed is generic and can be applied to a wide variety of scheduling problems in hard real-time systems.

The main contributions of this thesis are:

1. the development of a model for expressing QoS in terms of relative importance between tasks;

2. the development of algorithms that optimally solve the scheduling problem of meeting deadlines and minimizing (as a secondary criterion) the importance criterion;

3. the application of the algorithms and the importance criterion to solve practical multicriteria scheduling problems.

8.1.1 Importance

The first main contribution of this thesis is the development of a framework for expressing the QoS in terms of importance relationships. We argue that the traditional approaches based on value have a number of deficiencies due to the intrinsic complexities involved in the design and utilisation of utility functions. Instead, we propose a notion of QoS based on relative importance and conditional relative importance preferences; the former expresses that in a schedule it is desirable to run a task in preference to another one, and the latter expresses that in a schedule it is desirable to run a task in preference to another one subject to conditional clauses. We affirm that executing tasks in the order of importance with respect to some requirements improves the QoS of a system.

While relative importance is expressed as a number, conditional relative importance requires a set of conditional clauses expressed as priority assignment conditions attached to each task. This simplified model captures even complex QoS metrics between tasks including specific restrictions such as precedence constraints.

We propose the model of importance for improving QoS in two specific levels of abstraction:

1. at task level by assigning higher priorities to higher importance tasks while the deadlines are guaranteed;

2. at application level by finding priority assignments that are both feasible and as close as possible to a specific assignment.

In this context, we propose the lexicographic rule as a measure for expressing different levels of importance between tasks (i.e. task importance) as well as for expressing different levels of importance between priority orderings (i.e. index of importance). Thus, the metric of importance is defined in terms of the lexicographic distance. With this metric, we formulate the scheduling problem as finding a feasible priority ordering that meets the deadlines and minimises the lexicographic distance; minimising such distance maximises the importance. Afterwards, we solve the scheduling problem optimally using the DI and DI+ algorithms.

8.1.2 DI and DI+ Algorithms

The second main contribution of this thesis is the DI algorithm introduced in chapter 5. It solves optimally the scheduling problem of finding a feasible priority assignment that maximises the importance.

The DI algorithm performs a branch and bound search into the space formed by the $N!$ possible orderings, ordered lexicographically by importance. Taking advantage of some interesting properties of the optimality of the deadline monotonic priority assignment and some particular characteristics of the lexicographic order, we identify some patterns in the search space. This permits us to solve the problem in $O((N^2 + N)/2)$ steps; however its complexity depends on the feasibility test; for instance, DI is pseudo-polynomial when the response-time test is used. Our approach is optimal in the sense that the priority ordering found is feasible and no other feasible priority assignment exists with higher importance.

The DI+ algorithm was introduced in chapter 6 with the objective of solving optimally the scheduling problem of finding a feasible priority assignment that maximises the conditional relative importance. DI+ is DI with some modifications that allow including the feasibility tests for tasks with arbitrary deadlines or weakly-hard constraints, as well as including the swapping algorithm for finding feasible priority orderings when DMPA is not optimal. DI+ permits the solution of the problem in $O((N^2 + N)/2)$ steps; however its complexity depends on the feasibility test and the priority assignment algorithm. While in problems with arbitrary deadlines the complexity is still pseudo-polynomial, problems with weakly-hard constraints could require $(N^2 + N)/2$ simulations in the worst-case,

which is computationally costly.

With the DI and DI+ algorithms, a decision maker can specify a priority assignment based on objective or subjective considerations about the system and, if such assignment is infeasible, the algorithms can find a feasible one as close as possible of the assignment desired. Although algorithms such as the swapping algorithm can be considered as good heuristics for this specific problem, such solutions are not optimal and are unpredictable in the sense that we do not know how close they are from the optimal one (see Figure 5.1).

The scheduling problems solved by DI and DI+ are not only theoretically interesting but they can also be applied to practical problems. In this sense, we propose to use them jointly with specific importance assignments with the objective of solving multicriteria scheduling problems.

8.1.3 Solutions to Multicriteria Problems

When used with specific importance assignments, the DI and DI+ algorithms allow finding near-optimal or good priority assignment for problems where the QoS can be correlated with the concept of importance.

By extensive simulation, we propose heuristics for assigning importance and we evaluate its effectiveness when used jointly with the algorithms. These heuristics allow proposing simple solutions for practical multicriteria problems in real-time systems at zero cost for the FPS base mechanism.

We evaluate the six different DI solutions formed with the rules from Table 7.1 with respect to six different scheduling problems defined in chapter 4. We also compare the DI solutions against the solutions obtained with RMPA and EDF. The results show that our approach improves the RMPA solution and outperforms EDF in some scenarios.

Two types of scheduling problems were evaluated: (A) when all tasks are included in the problem and (B) when only a subset of tasks is included in the problem. The following results apply to both types.

- *DI(1/C)* is the best solution for the *Deadlines and Absolute Output Jitter*, and the

Deadlines and Relative Maximum Latency problems.

- *DI(LC)* is a near-optimal solution for the *Deadlines and Total Number of Preemptions* problem. It is also a better solution than RMPA for the *Deadlines and Maximum Latency* problem.

- *DI(T/C)* is an excellent solution for the *Deadlines and Relative Average Response-Time* problem.

For the *Deadlines and Relative Output Jitter* problem, RMPA is better than the DI solutions for a problem of type A. On the other hand, for a problem of type B, the solution DI(1/T) provides a good solution.

For scheduling problems with three or more objectives, in Table 7.2 we can observe the following results, which are valid for both types of problems:

- *DI(1/C)* is our best DI solution. In effect, DI(1/C) shows excellent performance when the deadlines have to be met and the secondary objectives are to minimise the absolute output jitter and the relative maximum latency and the relative average response-time.

- *DI(LC)* provides an excellent solution for a scheduling problem where the deadlines have to be met and the secondary objectives are to minimise both the total number of preemptions and the maximum latency.

- *DI(T/C)* is an excellent solution when deadlines have to be met and the secondary objectives are to minimise both the relative average response-time and the relative maximum latency.

Thus, our approach is very attractive as it only relies on a fixed priority mechanism (widely supported by commercial RTOS and standards) and does not require changes to the attributes or structure of tasks. The priority assignment is performed off-line and standard schedulability tests are utilised. Moreover, the pseudo-polynomial computational complexity of the DI algorithm makes the approach tractable even for large tasks sets giving results close to the (usually non computable) optimal priority assignment.

In addition, our experiments show that the swapping algorithm is as effective as the DI algorithm when used jointly with the importance assignments for the scheduling problems tested. This result makes clear that the concept of importance can be used with other scheduling algorithms. Observe that while it is true that both algorithms perform similarly, the DI algorithm is more predictable than the swapping algorithm since we know the DI solution is the lexicographic closer one to the importance assignment.

8.2 Suggestions for Further Work

The scheduling algorithms for solving multicriteria problems evaluated are not *Z-optimal*; for instance, DI(1/C) is not so close to the the *ȷ-optimal* solution. Thus, there exists room for improvements in two components of our approach:

- Algorithms for assigning importance. Instead of using simple rules, more effective algorithms for assigning importance could be developed considering more complex functions (e.g. $I = T/C^2$). Moreover, weights can be added to simple rules for expressing preferences in a particular solution. For instance, the rule 1/C could becomes $\frac{weight}{C}$.
- Improvements to the basic DI algorithm. The DI algorithm picks up tasks in the predetermined lexicographic order; i.e. the lexicographic order established is static. However, it could be the case that depending on how some of the tasks are accommodated during the operation of the DI algorithm, the original lexicographic order can be dynamically modified. For instance, if one of the two higher priorities is not assigned to a specific task, then it could be preferred to assign a lower priority.

The scheduling approaches proposed in this thesis impact on all the FPS techniques that use mainly RMPA or DMPA as fixed-priority assignment algorithms. In this sense we suggest further investigation in topics such as:

- Real-time database systems. A real-time database is composed of real-time objects which are updated by periodic sensor transactions. A real-time object has associated

a temporal validity interval; i.e. it is useful before its temporal validity interval expires. Thus, one of the important design goals is to guarantee that temporal data remain fresh. Therefore, efficient scheduling approaches are desired to guarantee the freshness of temporal data of the most important objects while minimizing the processor workload resulting from periodic sensor transactions. Finding feasible priority assignments that promote higher importance transactions and reduce the number of preemptions will improve the performance in those systems.

- Energy issues. In scheduling approaches related with energy management, power reductions are obtained by using hardware and/or software technologies. In effect, special hardware such as dynamic voltage scalable processors and I/O-devices with multiple power modes offer alternatives to save energy. With respect to software, the development of algorithms to make power-management decisions taking advantage of variations in system workload and resources is an active research area. In this context, FPS approaches with energy management try to guarantee deadlines and to minimise energy consumption without considering the importance of individual tasks. Adding the concept of importance will improve the quality in those systems.

8.3 Final Remark

Citing Burns [19] who states fixed-priority scheduling is a mature engineering approach for the construction of hard real-time systems, we consider that some day in a near future we will be able to state

fixed-priority scheduling is a sufficiently mature engineering approach for the construction of hard real-time systems that meet both timing and QoS requirements.

This research strives to achieve this objective.

Appendix A. Notation

$a, b, \ldots$ Roman small letters denote tasks . . . page 60

$\alpha, \beta, \ldots$ Greek small letters denote orderings or sub-orderings . . . page 60

$\{a, b, \ldots\}$ denote a task set . . . page 60

$\langle a, b, \ldots \rangle$ denote an ordering . . . page 60

$\succ$ operator used for comparing tasks or orderings by importance . . . page 63

b beginning time of a task . . . page 17

B maximum blocking . . . page 59

c completion time of a task . . . page 17

C worst-case execution time of a task . . . page 17

D deadline of a task . . . page 18

$F(k)$ worst-case finalization time at release k . . . page 37

I importance of a task . . . page 63

$\bar{I}$ conditional importance of a task . . . page 102

J release jitter of a task . . . page 19

$\mathcal{J}$ Absolute Output Jitter metric . . . page 69

$\mathcal{J}^{rel}$ Relative Output Jitter metric . . . page 70

$\mathcal{L}$ Maximum Latency metric . . . page 71

$\mathcal{L}^{rel}$ Relative Maximum Latency metric . . . page 71

μ μ-pattern of a task with weakly-hard constraints . . . page 36

O offset of a task . . . page 18

P priority of a task . . . page 59

$\mathcal{P}$ Total Number of Preemptions metric . . . page 69

r release time of a task . . . page 17

r_0 initial release of a task . . . page 17

R response time of a task . . . page 18

$\mathcal{R}$ Maximum Relative Average Response Time metric .... page 72

S a set of tasks .... page 17

$\hat{S}$ the set of all possible $N!$ orderings of S .... page 60

$\hat{S}_F$ the set of all feasible orderings of S .... page 60

S^A task set S ordered by the swapping algorithm .... page 91

S^D task set S ordered by the deadline monotonic priority assignment . page 60

S^I task set S ordered by task importance .... page 61

T period or minimal inter-arrival time of a task .... page 18

U utilisation of a task set (i.e $\sum_{j=1}^{N} C_j/T_j$) .... page 20

V relative value (or weight) of a task .... page 20

Z_D Deadlines metric .... page 74

Z_I Index of Importance metric .... page 65

$Z_{\bar{I}}$ Index of Conditional Importance metric .... page 103

Bibliography

[1] The American Heritage. *Dictionary of the English Language*. Houghton Mifflin, fourth edition, 2000.

[2] M. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Prentice Hall, second edition, August 2001. ISBN: 0-13-028138-7.

[3] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, 3rd edition, 2001.

[4] H. Wu, B. Ravindran, E. D. Jensen, and P. Li. Time/Utility Function Decomposition Techniques for Utility Accrual Scheduling Algorithms in Real-Time Distributed Systems. *IEEE Trans. Comput.*, 54(9):1138–1153, 2005.

[5] G. Bernat, A. Burns, and A. Llamosi. Weakly Hard Real-Time Systems. *IEEE Transactions on Computers*, 50(4):308–321, 2001.

[6] IEEE Standard. *IEEE Recommended Practice for Software Requirements Specifications*, 1998. Standard 830-1998.

[7] E. D. Jensen, C. D. Locke, and H. Tokuda. A Time-Driven Scheduling Model for Real-Time Operating Systems. In *IEEE Real-Time Systems Symposium*, pages 112–122, 1985.

[8] P. Li, B. Ravindran, and E. D. Jensen. Adaptive Time-Critical Resource Management Using Time/Utility Functions: Past, Present, and Future. In *Proc. of the 28th Annual Int. COMPSAC'04 - Workshops and Fast Abstracts*, Washington, DC, USA, 2004. IEEE Computer Society.

[9] S. Aldarmi and A. Burns. Dynamic Value-Density for Scheduling Real-Time Systems. In *11th Euromicro Conference on Real-Time Systems*, Jun 1999.

[10] J. Xu and D. L. Parnas. Priority Scheduling Versus Pre-Run-Time Scheduling. *Real-Time Systems*, 18(1):7–23, 2000.

[11] J. A. Stankovic and R. Rajkumar. Real-Time Operating Systems. *Real-Time Systems*, 28(2-3):237–253, 2004.

[12] V. T'kindt and J. Ch. Billaut. *Multicriteria Scheduling Theory, Models and Algorithms (Chapters 1-4)*. Springer-Verlag, 1st edition, August 2002. ISBN-10: 3540436170.

[13] S. Baruah, G. C. Buttazzo, S. Gorinsky, and G. Lipari. Scheduling Periodic Task Systems to Minimize Output Jitter. In *Proc of the 6th IEEE Int. Conf. on Real-Time Computing Systems and Applications*, pages 62–69, Hong Kong, December 1999.

[14] P. Marti, J. M. Fuertes, K. Ramamritham, and G. Fohler. Jitter Compensation for Real-Time Control Systems. In *RTSS '01: Proc. of the 22nd IEEE Real-Time Systems Symposium*, page 39, Washington, DC, USA, 2001. IEEE Computer Society.

[15] A. Cervin. Improved Scheduling of Control Tasks. *Euromicro Conference on Real-Time Systems*, 00:0004, 1999.

[16] L. David, F. Cottet, and N. Nissanke. Jitter Control in On-Line Scheduling of Dependent Real-Time Tasks. In *RTSS '01: Proc. of the 22nd IEEE Real-Time Systems Symposium*, page 49, Washington, DC, USA, 2001. IEEE Computer Society.

[17] G. C. Buttazzo. Rate Monotonic vs. EDF: Judgment Day. *Real-Time Systems*, 29(1):5–26, 2005.

[18] G. Bernat. *Specification and Analysis of Weakly Hard Real-Time Systems*. PhD thesis, Dep. Ciences Matematiques i Informatica, Universitat de les Illes Balears, 1998.

[19] A. Burns. Preemptive Priority-Based Scheduling: An Appropriate Engineering Approach. In *Advances in Real-Time Systems*, pages 225–248. Prentice-Hall, Inc., 1995.

[20] A. Burns and D. Prasad. Value-Based Scheduling of Flexible Real-Time Systems for Intelligent Autonomous Vehicle Control. In Editors M.A. Salichs and A. Halme, editors, *Proc. of the 3rd IFAC Symposium on Intelligent Autonomous Vehicles*, May 1998.

[21] A. Burns, D. Prasad, A. Bondavalli, F. D. Giandomenico, K. Ramamritham, J. A. Stankovic, and L. Stringini. The Meaning and Role of Value in Scheduling Flexible Real-Time Systems. *Journal of Systems Architecture*, vol 46(4):305–325, 2000.

[22] K. W. Tindell, A. Burns, and A. J. Wellings. Mode Changes in Priority Pre-Emptively Scheduled Systems. In *IEEE Real-Time Systems Symposium*, pages 100–109, 1992.

[23] J. Real and A. Crespo. Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal. *Real-Time Systems*, 26(2):161–197, 2004.

[24] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[25] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997.

[26] TimeSys Corp. The Concise Handbook of Real-Time Systems, v 1.3, 2002.

[27] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *J. ACM*, 20(1):46–61, 1973.

[28] J. Y. T. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks. *Performance Evaluation*, vol 2:237–250, 1982.

[29] N. C. Audsley. Optimal Priority Assignment and Feasibility of Static Priority Tasks with Arbitrary Start Times. Technical report, Dept. Computer Science, University of York, 1991.

[30] A. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. PhD thesis, MIT, Department of EE and CS, MIT/LCS/TR-297, May 1983.

[31] E. D. Jensen, C. D. Locke, and H. Tokuda. A Time Driven Scheduling Model for Real-Time Operating Systems, 1985.

[32] D. Stewart and P. Khosla. Real-Time Scheduling of Sensor Based Control Systems. IEEE Workshop on Real-Time Operating Systems and Software, May 1991.

[33] C. D. Locke. Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives. *Real-Time Systems*, 4(1):37–53, 1992.

[34] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proc. 8th Workshop on Real-Time Operating Systems and Software*. IEEE, Atalanta, 1991.

[35] K. W. Tindell, A. Burns, and A. J. Wellings. An Extendible Approach for Analyzing Fixed Priority Hard Real-Time Tasks. *Real-Time Systems*, 6(2):133–151, 1994.

[36] K. W. Tindell. Adding Time-Offsets to Schedulability Analysis. Internal Report YCS-94-221, University of York, Computer Science Dept., 1994.

[37] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.

[38] E. Bini, G. C. Buttazzo, and G. M. Buttazzo. A Hyperbolic Bound for the Rate Monotonic Algorithm. In *Proc. of the 13th Euromicro Conference on Real-Time Systems*, page 59, Washington, DC, USA, 2001. IEEE Computer Society.

[39] M. Joshep and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 1(29):390–395, 1986.

[40] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic Task Scheduling for Hard Real-Time Systems. *Real-Time Systems*, 1(1):27–60, 1989.

[41] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. *IEEE Transactions on Computers*, 44(1):73–91, Jan 1995.

[42] R. Davis. Dual Priority Scheduling: A Means of Providing Flexibility in Hard Real-Time Systems. Report YCS230, University of York, UK, May 1994.

[43] IEEE. *Standard Glossary of Software Engineering Terminology*. New York, September 1990.

[44] M. Glinz. Rethinking the Notion of Non-Functional Requirements. In *Proceedings of the Third World Congress for Software Quality (3WCSQ 2005)*, volume II, pages 55–64, 2005.

[45] T. Gilb. Towards the Engineering of Requirements. Requirements Engineering 2, 165-169., 1997.

[46] A. Dardenne, A.V. Lamsweerde, and S. Fickas. Goal-Directed Requirements Acquisition. *Science of Computer Programming*, 20(1-2):3–50, 1993.

[47] A. van Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. In *RE '01: Proc. of the 5th IEEE International Symposium on Requirements Engineering*, page 249, Washington, DC, USA, 2001. IEEE Computer Society.

[48] R. Kazman, M. Klein, and P. Clements. Evaluating Software Architectures for Real-Time Systems. In *Annals of Software Engineering*, volume 7 of *71-93*. Springer Netherlands, 1999.

[49] L. Thiele and R. Wilhelm. Design for Timing Predictability. *Real-Time Systems*, 28(2-3):157–177, 2004.

[50] L. R. Welch and S. Brandt. Toward a Realization of the Value of Benefit in Real-Time Systems. In *Proc. of the 15th International Parallel & Distributed Processing Symposium*, page 93, Washington, DC, USA, 2001. IEEE Computer Society.

[51] R. Rajkumar, C. Lee, J. P. Lehoczky, and D. Siewiorek. A Resource Allocation Model for QoS Management. In *Proc. of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, page 298, Washington, DC, USA, 1997. IEEE Computer Society.

[52] C. Lee, J. P. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hansen. A Scalable Solution to the Multi-Resource QoS Problem. In *Proc. of the 20th IEEE Real-Time Systems Symposium*, page 315, Washington, DC, USA, 1999. IEEE Computer Society.

[53] S. Ghosh, J. Hansen, R. Rajkumar, and J. P. Lehoczky. Integrated Resource Management and Scheduling with Multi-Resource Constraints. In *Proc. of the 25th IEEE International Real-Time Systems Symposium (RTSS'04)*, pages 12–22, Washington, DC, USA, 2004. IEEE Computer Society.

[54] K. T. Kornegay, G. Qu, and M. Potkonjak. Quality of Service and System Design. In *WVLSI '99: Proc. of the IEEE Computer Society Workshop on VLSI'99*, page 112, Washington, DC, USA, 1999. IEEE Computer Society.

[55] B. Ravindran, E. D. Jensen, and P. Li. On Recent Advances in Time/Utility Function Real-Time Scheduling and Resource Management. *ISORC*, 00:55–60, 2005.

[56] H. Wu, B. Ravindran, E. D. Jensen, and P. Li. Time/Utility Function Decomposition in Soft Real-Time Distributed Systems. Technical report, The MITRE Corp., 2004.

[57] C. D. Locke. *Best-Effort Decision-Making for Real-Time Scheduling.* PhD thesis, Carnegie Mellon University, 1986.

[58] K. Ramamritham and J. A. Stankovic. Scheduling Algorithms and Operating Systems Support for Real-Time Systems. In *Proc. of the IEEE*, volume 82, Jan. 1994.

[59] J. W. Wendorf. Implementation and Evaluation of a Time-Driven Scheduling Processor. In *Real-Time Systems Symposium*, pages 172 – 180, December 1988.

[60] G. C. Buttazzo, M. Spuri, and F. Sensini. Value vs. Deadline Scheduling in Overload Conditions. In *Proc. of the 16th IEEE Real-Time Systems Symposium (RTSS '95)*, page 90, Washington, DC, USA, 1995. IEEE Computer Society.

[61] J. R. Haritsa, M. J. Carey, and M. Livny. Value-Based Scheduling in Real-Time Database Systems. *The VLDB Journal*, 2(2):117–152, 1993.

[62] R. K. Clark. *Scheduling Dependent Real-Time Activities.* PhD thesis, Carnegie Mellon University, 1990.

[63] P. Li and B. Ravindran. Fast, Best-Effort Real-Time Scheduling Algorithms. *IEEE Trans. Comput.*, 53(9):1159–1175, 2004.

[64] T. F. Abdelzaher, E. M. Atkins, and K. G. Shin. QoS Negotiation in Real-Time Systems and Its Application to Automated Flight Control. *IEEE Trans. Comput.*, 49(11):1170–1183, 2000.

[65] W. Lee and B. Sabata. Admission Control and QoS Negotiations for Soft Real-Time Applications. *IEEE Int. Conf. on Multimedia Computing and Systems*, 01, 1999.

[66] H. Wu, B. Ravindran, E. D. Jensen, and P. Li. Energy-Efficient, Utility Accrual Scheduling Under Resource Constraints for Mobile Embedded Systems. In *Proc. of the 4th ACM International Conference on Embedded Software*, pages 64–73, New York, NY, USA, 2004. ACM Press.

[67] A. Mauthe and G. Coulson. Scheduling and Admission Testing for Jitter-Constrained Periodic Threads. *Multimedia Systems*, 5(5):337–346, 1997.

[68] D. Prasad, A. Burns, and M. Atkins. The Valid Use of Utility in Adaptive Real-Time Systems. *Real-Time Systems*, 25(2-3):277–296, 2003.

[69] G. Lima and A. Burns. An Optimal Fixed-Priority Assignment Algorithm for Supporting Fault-Tolerant Hard Real-Time Systems. *IEEE Transactions on Computers*, 52(10):1332–1346, Oct 2003.

[70] P. Richard. A Tool for Controlling Response Time in Real-Time Systems. In *Computer Performance Evaluation / TOOLS*, pages 339–348, 2002.

[71] Y. Wang and M. Saksena. Scheduling Fixed Priority Tasks with Preemption Threshold. Proc. of the Sixth International Conference on Real-Time Computing Systems and Applications, 1999. IEEE Computer Society.

[72] R. Dobrin and G. Fohler. Reducing the Number of Preemptions in Fixed Priority Scheduling. In *Proc. of the 16th Euromicro Conference on Real-Time Systems*, volume 00, pages 144–152, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[73] R. Melhem, N. AbouGhazaleh, H. Aydin, and D.l Mosse. Power Management Points in Power-Aware Real-Time Systems. In *Power Aware Computing*, pages 127–152. Kluwer Academic Publishers, 2002.

[74] R. Krishnapura and S. Goddard. Dynamic Real-Time Scheduling for Energy Conservation in I/O Devices. In *Workshop on Constraint-Aware Embedded Software*, volume 2. 24th IEEE Real-Time Systems Symposium, Dec 2003.

[75] Y. Shin and K.Choi. Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems. In *Proc. of the 36th ACM/IEEE Conference on Design Automation*, pages 134–139. ACM Press, 1999.

[76] H. Aydin, R. Melhem, D. Mosse, and P.M. Alvarez. Determining Optimal Processor Speeds for Periodic Real-Time Tasks with Different Power Characteristics. In *Proc. of the 13th EuroMicro Conference on Real-Time Systems*, Delft, Netherlands, Jun 2001.

[77] X. Li and L. K. Soh. Applications of Decision and Utility Theory in Multi-Agent Systems. Technical Report TR-UNL-CSE-2004-0014, Computer Science,University of Nebraska-Lincoln, 2004.

[78] M. N. Rothbard. Toward a Reconstruction of Utility and Welfare Economics, 1956.

[79] R. P. Murphy. Utility Functions and Divided Preferences: A Note On Hlsmann. New York University, July 2000.

[80] Jr. J. B. Cruz and M. A. Simaan. Ordinal Games and Generalized Nash and Stackelberg Solutions. *Journal of Optimization Theory and Applications*, 107(2):205–222, November 2000.

[81] R. Brafman and C. Domshlak. Introducing Variable Importance Tradeoffs into CP-Nets. In *Workshop on Planning and Scheduling with Multiple Criteria*, April 2002.

[82] C. Boutilier, R. Brafman, C. Geib, and D. Poole. A Constraint-Based Approach to Preference Elicitation and Decision Making. In *AAAI Spring Symposium on Qualitative Decision Theory*, pages 19–28, Menlo Park, California, 1997. American Association for Artificial Intelligence.

[83] H.l Andréka, M. Ryan, and P. Schobbens. Operators and Laws for Combining Preference Relations. *J. Log. Comput.*, 12(1), 2002.

[84] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw Hill, 3rd edition, 2000.

[85] E. Bini and G. C. Buttazzo. Biasing Effects in Schedulability Measures. *16th Euromicro Conference on Real-Time Systems*, pages 196–203, 2004.