INTERACTIVE DESIGN SPACE EXPLORATION OF REAL-TIME EMBEDDED SYSTEMS

UNMESH DUTTA BORDOLOI

NATIONAL UNIVERSITY OF SINGAPORE

2008

INTERACTIVE DESIGN SPACE EXPLORATION OF REAL-TIME EMBEDDED SYSTEMS

UNMESH DUTTA BORDOLOI

(B.Tech., Computer Science Engineering,

National Institute of Technology, Rourkela, India)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2008

List of Publications

---

1. U. D. Bordoloi and S. Chakraborty. Accelerating System-Level Design Tasks using Commodity Graphics Hardware: A Case Study. *Accepted to International Conference on VLSI Design (8th International Conference on Embedded Systems)*, January 2009.

2. U. D. Bordoloi. Interactive Performance Debugging of Real-Time Embedded Systems, SIGDA PhD Forum, *Design Automation Conference (DAC)*, June 2008.

3. U. D. Bordoloi and S. Chakraborty. Interactive Schedulability Analysis. *ACM Transactions on Embedded Computing Systems (TECS)*, pages 1-27, Volume 7, Issue 1, December 2007.

4. U. D. Bordoloi, S. Chakraborty, and A. Hagiescu. Performance Debugging of Heterogeneous Real-Time Systems. Book Chapter in *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pages 285-300, Springer Netherlands, 2007.

5. J. Feng, S. Chakraborty, B. Schmidt, W. Liu, and U. D. Bordoloi. Fast Schedulability Analysis Using Commodity Graphics Hardware. In *Proc. 13th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 400-408, IEEE Computer Society, 2007.

6. A. Hagiescu, U. D. Bordoloi, S. Chakraborty, P. Sampath, P. V. V. Ganesan, and S. Ramesh. Performance Analysis of FlexRay-based ECU Networks. *In Proc. 44th Design Automation Conference (DAC)*, pages 284 - 289, ACM, 2007.

7. U. D. Bordoloi and S. Chakraborty. Performance Debugging of Real-Time Systems using Multicriteria Schedulability Analysis. In *Proc. 13th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 193-202, IEEE Computer Society, 2007.

8. U. D. Bordoloi and Samarjit Chakraborty. Interactive Schedulability Analysis. In *Proc. 12th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 147-156, IEEE Computer Society, 2006. (Invited to a special issue of ACM Transactions on Embedded Computing Systems, on selected best papers from RTAS'06).

Acknowledgments

These past few years as a doctoral researcher have been one of the most memorable and enjoyable times of my life. I would like to acknowledge the wonderful people without whom this experience would not have been possible.

Throughout my PhD candidature, I have received valuable guidance and stimulating suggestions from Dr. Samarjit Chakraborty and I am grateful to him for this. His positive outlook and zeal for research has inspired me on countless occasions. I also appreciate his patience for thoroughly revising my written manuscripts and providing insightful feedback. Dr. Samarjit Chakraborty has also been a friend and I have immensely benefited from his help and advice. Indeed, it is rare to meet personalities with such unassuming nature.

I am grateful to all the members of my dissertation committee for writing the reports in such short time inspite of their busy schedules. I would like to thank Dr. P. S. Thiagarajan and Dr. Weng Fei Wong for suggesting significant improvements. Thanks are also due to Dr. Marco Platzner for being my external reviewer and for his valuable remarks and corrections.

This thesis would be incomplete without the contributions of my colleagues Jimin Feng and Andrei Hagiescu, colleagues at Embedded Systems Lab. Discussions with researchers at Nanyang Technical University and at General Motors, India Science Lab have lead to fruitful projects, and I gratefully acknowledge their help. I also

thank Dr. S. Ramesh at General Motors, India Science Lab, for useful advice and encouragement during my research work.

It was my good fortune to have amazing lab-mates in the Embedded Systems Lab. I have fully exploited the privilege of being a part of this truly enjoyable environment to ask anyone for all kinds of help, without thinking twice. Indeed, without all the help that you guys offered, I would have been overwhelmed with my numerous issues with latex, code, and what not! I also appreciate all the enlightening discussions, technical and non-technical, with all of you that were so much a part of my graduate life.

Thanks to the responsive and capable workforce at Technical Helpdesk, there were hardly any issues with any technical equipment that I had to use. I also appreciate the efficient administrative work of the Graduate Office, School of Computing, especially Ms. Loo Line Fong.

I sincerely thank the National University of Singapore for supporting me financially, and encouraging me with generous Fellowships.

Unlimited love has been showered on me from all my relatives, uncles, aunts, and cousins, and I have been blessed with an incredible family. I have a terrific Kokaideo (elder brother), one with a PhD in computer science. His wisdom has benefited me all my life, and because of his wise words, I knew from day one what to expect in a PhD. I have a spirited and smart sister, Xuwodi, and her cheerfulness always keeps my spirits up.

Finally, there is no means by which I may repay all the sacrifices that my parents made for me. Without their far-sightedness, and broad-mindedness, this journey would have been never possible.

Contents

Abstract

A typical design of a real-time embedded system involves an iterative design space exploration process. In general, the design space exploration strategy needs to address two separate concerns.

1. How to cover the entire design space during the exploration process? Typically, the designer is confronted with a prohibitively large design space, where the design points are associated with conflicting tradeoffs with respect to various performance metrics like real-time response, costs etc.

2. How to quantitatively evaluate a single design point with respect to the various performance metrics? The designer needs to run a *performance analysis* to evaluate each design point, and for most realistic system models such *performance analysis* is time consuming.

The above issues lead to tedious iterations during design space exploration of real-time embedded systems. A system designer would choose the values of the system parameters and define an initial design point. The designer would then invoke a *performance analysis* tool to evaluate the performance metrics corresponding to the design point. If the designer is not satisfied with the resulting performance numbers, then he/she would modify some of the parameters and invoke the *performance analysis* once again. This iterative design space exploration is repeated

until a satisfactory design is found. Unfortunately, as discussed above, each time the *performance analysis* tool is invoked it takes a long time to run — which might be in the tune of several hours – and this critically impacts the usability of the tool in the interactive design space exploration sessions.

Current approaches rely mostly on ad-hoc techniques like genetic algorithms to handle the high running times associated with such iterative design space exploration processes. In this thesis we present systematic/formal approaches which provide provable performance guarantees. We propose (i) novel algorithmic techniques (both exact and approximate), as well as (ii) hardware-based techniques to accelerate the computationally expensive *performance analysis* in each iteration. We also introduce (i) a scheme to approximate the potentially exponential sized design space with only a polynomial number of points and (ii) techniques to provide insightful feedback to the designer regarding the design parameters he may choose to modify in each iteration. In particular, this thesis makes the following contributions.

- We introduce the novel concept of "interactive" design space exploration to accelerate each iteration in an interactive design session. We demonstrate our idea with respect to a schedulability analysis problem. Our algorithm is based on the observation that if only a small number of system parameters are changed in each iteration, then it is not necessary to re-run the full schedulability analysis algorithm, thereby making the iterative design process considerably faster. We demonstrate that using our scheme can lead to more than 20× speedup for each invocation of the schedulability analysis algorithm, compared to the case where the full algorithm is run. Such fast iterations also allow the designer to evaluate the schedulability for much larger design space within a short time. We also outline some techniques for

providing feedback on the potential system parameters that can be changed to obtain a schedulable system when a task set is not schedulable.

- Design space exploration for hardware/software co-design involves identifying all possible implementations to expose the different possible performance tradeoffs associated with each of them. Unfortunately, the problem of optimally computing even one feasible solution in most common setups is computationally intractable (NP-hard). In this thesis we derive a polynomial-time approximation algorithm for solving it. Furthermore, our scheme also approximates the potentially exponential sized solution set with only a polynomial number of points. This is more meaningful from a practical perspective, as the designer is presented with a reasonably few well-distinguishable tradeoffs, rather than an exponentially large number of solutions, many of which are similar to each other.

- We introduce the new technique of employing graphics processing units (GPUs) to lower the high running times associated with heavy duty kernels of design space exploration problems. To demonstrate our idea, we present GPU-based engines to diminish the long running times associated with an expensive hardware/software design space exploration problem and a schedulability analysis problem. Our experiments on the GPU demonstrate tremendous speed up (upto 100×) of the expensive kernel of our problems.

- Apart from the above, we have also been concerned real-life design issues, specially in the automotive domain. In this regard, we have developed novel analytical methods which facilitate fast design space exploration of system parameters for safety-critical applications in the automotive domain. In contrast to traditional simulation methods which take hours to run, our analytical model returns results in a matter of few seconds, and is ideal for interactive design sessions.

To summarize, this thesis is concerned with issues arising in design space exploration of real-time embedded systems. Interactive design cycles associated with design space exploration techniques are known to be tedious, and this thesis proposes novel algorithmic, analytic and hardware-based techniques to ease the tedious design cycles.

List of Figures

List of Tables

Chapter 1

Introduction

An embedded system is an electronic device which contains a special-purpose computing system embedded within it. Typically, such a device is a combination of hardware and software designed to meet the special functionality of the system. These systems are found in numerous application domains ranging from brake controllers in automobiles and controllers in industrial plants, to mobile health monitoring devices.

Most of the embedded systems, such as those mentioned above, need to continuously interact with their physical environment through sensors and actuators. Once the embedded system receives an input on the sensors, it needs to do some computation and if required, send an output signal on the actuators. As most of these applications are safety-critical, failure of the system to reply within the expected time interval might lead to a catastrophic accident, possibly loss of human-life. For instance, a delayed response of an automated brake-controller in a moving car might result in a fatal crash. Thus, apart from guaranteeing correct computation, many embedded systems must also meet real-time constraints, i.e. they must finish the computation and react to stimuli within a definite time interval.

Furthermore, due to considerations such as limited space and costs, the amount of memory available is scarce in most of these real-time embedded devices. Also, these devices are often mobile and have to run on batteries, which means that the power consumption should be limited as much as possible for longer life of the devices.

System-Level Performance Analysis

From the above discussion, we note that apart from being functionally correct, a real-time embedded system must conform to certain non-functional or *performance metrics* like timing constraints, memory size restrictions, power limitations, etc. To check whether all such performance metrics of a system are satisfied, the design of real-time embedded system typically starts with a *system-level performance analysis*.

Thus, in a design cycle, the designer would typically invoke a system-level performance analysis to seek answers to questions related to performance metrics like: Given a set of jobs chosen to run on a processor, does there exist an execution order or schedule which satisfies the timing constraints (*Schedulability Analysis*)? Which functions should be implemented in hardware and which in software to maximize performance and minimize the hardware costs (*Partitioning*)? Do the system-level timing properties meet the design requirements (*Timing Analysis*)? What would be the total response time or the *end-to-end delay* of the system once the system receives an input on the sensors, till it sends an output signal on the actuators?

In the next section, we introduce the problem of design space exploration of real-time embedded systems, and discuss the role of system-level performance analysis in design space exploration cycles.

1.1 Design Space Exploration

Because of the many alternatives for mapping and partitioning, application optimization, and architecture selection during the system design process, a designer of a complex embedded system is confronted with a large design space. Each point in the design space is associated with conflicting tradeoffs with respect to various performance metrics like real-time response, costs etc. For instance, response time (performance) of a system may be improved by implementing larger portions of task for a given application in the hardware (providing that the application offers enough "hardware realizable" functionalities) at the expense of an silicon area overhead. By extensively playing around with system parameters, designers can generate the trade-off curves in the design space defined by performance and area costs. Such a process of systematically altering design parameters has been recognized as an exploration of the design space.

Broadly, the design space exploration process consists of two orthogonal issues [36].

1. Firstly, the designer has to identify all the design points. Typically, the designer is confronted with a large design space, where a large number of implementation choices have to be investigated in order to determine design trade-offs between various possibly conflicting performance metrics.

2. The designer also needs to run a performance analysis to quantitatively evaluate each design point in order to compare their relative merits with respect to various performance metrics. For most realistic system models the performance analysis is time consuming and involves running one or more computationally expensive cores. We discuss this role of performance analysis in design space exploration elaborately in the following section.

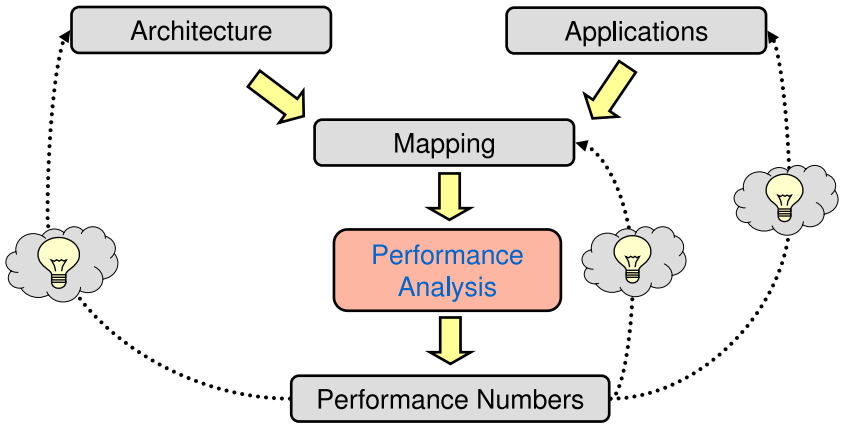


Figure 1.1: Role of Performance Analysis in Interactive Design Space Exploration.

1.1.1 Role of Performance Analysis in Design Space Exploration

Design space exploration of a real-time embedded system is not a one-step procedure, but rather an iterative procedure (see Figure 1.1). This process is well-known as the Y-chart methodology [42, 50, 86], and involves the following steps. The process starts with a specification of a set of representative target *applications*, which must be implemented on an *architecture* such that predefined performance constraints with respect to cost, real-time response, etc. are satisfied. In an explicit *mapping* step, the target *application* is mapped onto the candidate *architecture*. The designer then invokes a *performance analysis* tool to evaluate the performance metrics corresponding to the design point. If the designer is not satisfied with the resulting performance numbers, then he/she would modify some of the parameters and invoke the *performance analysis* once again. The designers might interpret the performance numbers manually, or might be inspired by feedback provided by the performance analysis tools to propose the new parameter values (this interpretation process is indicated in Figure 1.1 by the lightbulb). The designer may modify (i) the application parameters (worst-case *execution* times, *deadlines* and *periods*), (ii) the selection of architecture building blocks (number of processors, processor frequencies, hardware costs (in terms of ASIC/FPGA area)), or (iii) the

mapping strategy itself. This iterative design space exploration is repeated until a satisfactory design is found. Thus, a real-life design session of a embedded system for a system-level designer is *interactive*; they repeatedly invoke system-level performance analysis tools during the design exploration cycles.

Unfortunately, it turns out that interactive design space exploration is quite tedious. The prime reason for this being the fact that for most realistic system models the system-level performance analysis involves running one or more computationally expensive cores. Hence, each time the tool is invoked, the system designer has to wait for a long time (which might be in the tune of several hours) to let the analysis run to completion and this critically impacts the usability of the tool in the interactive design sessions.

1.1.2 Challenges

In the above we discussed the two major concerns in design space exploration: (i) a prohibitively large design space that must be covered during the exploration process, and (ii) a heavy-duty performance analysis to evaluate each design point. In this section, we shall discuss the particular reasons behind long and exhausting interactive design space exploration sessions associated with some common computationally expensive system-level performance analysis problems.

- **Schedulability Analysis**

  Schedulability analysis is used to determine if the temporal properties of a real-time system are satisfied. If the analysis returns a negative answer, the designer repeatedly changes system parameters and re-runs the analysis. However, for most realistic task models, schedulability analysis algorithms often involves running one or more computationally expensive cores [47, 11,

9]. Hence, each time the schedulability analysis tool is invoked, it takes a long time to run and this hampers the productivity of the designer in the iterative design sessions.

Apart from making the iterative design sessions faster, there are additional challenges involved with interactive schedulability analysis. For example, in each iteration of the design, if the designer randomly chooses a system parameter and makes a change, this change might not lead to a feasible system. The challenge is to develop a mechanism such that the tool provides the designer with some concrete feedback regarding what system parameter should be changed that would likely yield a feasible solution.

- **Hardware/Software Partitioning**

  Design space exploration plays an integral part in hardware/software partitioning; it involves evaluating the possible performance versus area trade-offs associated with all possible design points. Unfortunately, optimally computing even one feasible design point in most common setups is computationally expensive [36, 60]. Moreover, typically, there might be infinitely many points in the design space. Thus, the straightforward approach to determine the design points by an exhaustive search is intractable and not practical enough to be used in an interactive design cycle.

  Traditionally, researchers have been using different techniques to get around the high running times associated with such problems. The most notable amongst these are heuristics like genetic and evolutionary algorithms [37, 48]. However, these algorithms do not yield exact solutions and neither do they offer any kind of performance guarantee. Therefore, new techniques are necessary which are efficient as well as provide formal guarantees on the optimality of the design points that are returned.

- **Timing Analysis of Distributed Real-Time Applications**

  Over the past decade, embedded systems have increasingly become distributed in nature with different scheduling and arbitration schemes being used on the different processors and buses. One foremost example of such distributed real-time systems may be found in today's automobiles where electronic systems have gradually replaced mechanical ones in cars and trucks. Such distributed systems are rapidly increasing in size, communication complexity and software content. For example, today's vehicles can have more than 70 control units or processors, connected by multiple communication buses and running millions of lines of software [5]. Analysing such heterogeneous systems to verify timing and other system-level properties pose a major challenge. Traditional traditional design processes do not handle such complexity; system-level design methodology is required [65, 70]. Important system-level design decisions here involve identifying optimal scheduling policies, parameters of the bus protocol, end-to-end timing delays, buffer sizes, etc. Commercially available design tools for automotive electronics like Decomsys [27] and Dspace [28] rely on simulation techniques to provide such answers. Such simulation tools take long running times and coupled with naive design space exploration techniques, the total design cycle becomes very long.

1.2 Thesis Contributions

In the above discussion, we have identified two broad issues. Firstly, despite high running times associated with computationally expensive kernels of the performance analysis machinery (which lead to tedious interactive design cycles), current high-level design methodologies and tools have no support to address the problem.

Moreover, so far only ad-hoc solutions like evolutionary algorithms and exhaustive search techniques have been used in order to cope the prohibitively large design space to cope with multi-objective optimization design problems. In this thesis we present systematic/formal approaches which provide provable performance guarantees. We propose (i) novel algorithmic techniques, both exact and approximate, as well as (ii) hardware-based techniques to accelerate the computationally expensive *performance analysis* in each iteration. We also introduce (i) a scheme to approximate the potentially exponential sized design space with only a polynomial number of points and (ii) techniques to provide with insightful feedback to the designer regarding the design parameters he may choose to modify in each iteration. In particular, this thesis proposes novel techniques for interactive design space exploration by addressing the challenges associated with common system-level performance analysis problems discussed in Section 1.1.2.

- **Interactive Schedulability Analysis**

  We propose a novel approach to bring down the high running times associated with schedulability analysis algorithms, especially in the context of an iterative design process. It is based on the observation that if only a small number of design parameters are changed, then it is not required to invoke the full schedulability analysis machinery. Rather, certain data structures can be created when the algorithm is run for the first time, and on subsequent invocations of the algorithm it is possible to exploit these data structures and run only a small subset of the regular schedulability analysis algorithm. We refer to this as *interactive* schedulability analysis because it would typically be used in an interactive mode—a designer would keep on modifying the values of a small number of system parameters and use this algorithm to test whether the system becomes schedulable.

  This concept of interactive schedulability analysis is fairly general and can

be applied to a number of well-known task models. In this thesis, we have chosen the recently proposed recurring real-time task model [9] to illustrate this scheme. It has been shown in [9] that this model generalizes a number of task models. Further, it can be used to model realistic applications with conditional branches and fine-grained deadline constraints. Our experimental results show that using our scheme can lead to more than 20× speedup for each invocation of the schedulability analysis algorithm, compared to the case where the full algorithm is run.

Note that the designer repeatedly changes system parameters so that the schedulability analysis may yield a feasible solution. If the designer randomly chooses a system parameter and makes a change it might not lead to a feasible system. In our work, we also devise a technique using which a system designer can be provided some feedback regarding which system parameter(s) should be changed that would likely yield a feasible solution.

- **Hardware/Software Partitioning**

  We develop an efficient scheme for design space exploration in the context of hardware/software co-design of real-time systems. Such systems nowadays consist of a heterogeneous mix of fully-programmable processors, fixed-function components or hardware accelerators, and partially-programmable engines. Hence, system designers are faced with an array of implementation possibilities for an application at hand. Such possibilities typically come with different tradeoffs involving cost, power consumption and packaging constraints. As a result, a designer is no longer interested in *one* implementation that meets the specified real-time constraints (i.e. is schedulable), but would rather like to identify *all* schedulable implementations that expose the different possible performance tradeoffs formally known as the Pareto front. In this thesis we formally define this multicriteria schedulability analysis

problem and derive a polynomial-time approximation algorithm for solving it. This result is interesting because the problem of optimally computing even one schedulable solution in our setup (and in most common setups) is computationally intractable (NP-hard).

The second reason which makes our work interesting is that there can be an exponentially large number of points in the Pareto front, which makes it impossible to compute this entire set in polynomial time. Hence, our polynomial-time approximation algorithm by default also implies approximating the (potentially exponential size) set with only a polynomial number of points. In a typical design cycle, a system designer inspects all the tradeoffs in the set and then selects one, or at most a few implementations. Hence, from a practical perspective, it is more meaningful if the designer is presented with a reasonably few well-distinguishable tradeoffs in the set, rather than an exponentially large number of solutions, many of which are very similar to each other. Our approximation algorithm is therefore not only attractive in terms of time-complexity, but also returns more meaningful solutions.

- **Accelerating Performance Analysis Using GPUs**

  We introduce the novel idea of using commodity graphics hardware (more specifically, graphics processing units or GPUs) to accelerate the expensive cores associated with heavy-duty kernels of design space exploration problems. The two foremost reasons why GPUs are an attractive platform for such non-graphics computations are—(i) modern GPUs are extremely powerful (e.g. high-end GPUs such as nVIDIA GeForce 8800 GTX have a FLOPS rating of around 330 GigaFLOPS, whereas high-end general-purpose processors are only capable of around 25 GigaFLOPS) (ii) GPUs are now commodity items as their costs have dramatically reduced over the last few years. Thus, the attractive price-performance ratios of GPUs gives us an

enormous opportunity to change the way system-level performance analysis tools perform, with almost no additional cost. In fact, recent years have seen the increasing use of graphics processing units (GPUs) for a wide variety of general-purpose computing tasks. Examples of these include scientific computing [35, 45], computational geometry [2], database processing [3], image processing [56, 58], astrophysics [67] and bioinformatics [53].

In this thesis, we use the schedulability analysis of the recurring real-time task model problem and the hardware/software co-design problem to establish the utility of the GPUs in accelerating system-level performance analysis algorithms. Our experiments on the GPU demonstrate tremendous speed up (upto 16×) of the schedulability analysis algorithm and (upto 100×) speed-up of the hardware/software co-design problem.

- **Performance Analysis of Applications in Automotive Electronics**

  We have also been concerned with practical cases of embedded system design, and in this regard, we have specifically worked in the automotive domain. Our contributions in this direction are discussed below.

  We propose an analytical framework for compositional performance analysis of a network of processors that communicate via a FlexRay bus. FlexRay is fast emerging as the predominant protocol for in-vehicle automotive communication systems. Given a specification of the applications running on the system, their partitioning and mapping on the different processors, their activation rates or periods and the message priorities, our framework can be used to answer various performance analysis related questions. These include the maximum end-to-end delay experienced by the different message types, the amount of buffer space required within a communication controller associated with a processor and the utilizations of the different processors and the FlexRay bus.

In contrast to traditional simulation methods which takes hours to run, our analytical model returns results in a matter of few seconds, and is ideal for fast analysis in interactive design cycles. The framework allows the designer to extensively play around with the FlexRay protocol parameters in order to identify the suitable performance metric. Also, it can help in resource dimensioning (e.g. designing the various processors) and determining optimal scheduling policies for multitasking processors.

1.3 Organization of this Thesis

In the following we give a brief overview of the contents of this thesis. Chapter 2 presents our scheme for "interactive" schedulability analysis. We also describe a technique using which a system designer can be provided some feedback on potential modifications that may be done when a task set is not schedulable.

Our work on design space exploration using approximation techniques is presented in Chapter 3. We formally define the single criteria version of the problem, prove that it is NP-hard and derive a polynomial-time approximation scheme for solving it. This is followed by our solution to the multicriteria problem.

Chapter 4 deals with our idea of accelerating performance analysis problems using commodity graphics processor units (GPUs). Towards this, we propose two GPU-based engines — (i) for a hardware/software co-design and (ii) for a schedulability analysis algorithm.

Chapter 5 contains the results related to performance analysis of FlexRay based automotive networks. Finally, we summarize this thesis in Chapter 6 with directions for future work.

Chapter 2

Interactive Schedulability Analysis

Schedulability analysis plays an integral role in the system-level design of real-time embedded systems. Once a designer chooses the values of the relevant system parameters, schedulability analysis is used to determine whether it is possible to assign to each job a processor time equal to its worst-case execution requirement, between its ready time and its deadline. If such an analysis returns a negative result (i.e. there exist legal scenarios where certain jobs might miss their deadlines), then some of the system parameters are relaxed and the analysis is invoked once again. On the other hand, if such an analysis returns a positive result (i.e. all jobs definitely meet their deadlines), the designer might want to constrain some of the system parameters and re-invoke the analysis to find a tighter set of design parameters where the system is schedulable. Thus, in a typical system design process, this iteration is repeated a number of times where the designer evaluates the schedulability for a extensive set of design parameters.

Unfortunately, the schedulability analysis problem for most task models is intractable (usually co-NP hard). Therefore, known algorithms for these models

have an exponential time complexity and at best run in pseudo-polynomial time. As a result, the above-mentioned iterative design process can become overly tedious for even reasonably-sized problems. To get around this, recent research in the real-time systems area has focused on either obtaining efficient pseudo-polynomial time algorithms or on *approximately* solving the schedulability analysis problem [4, 21, 32].

In this chapter, we propose another possible approach to bring down the high running times associated with schedulability analysis algorithms, especially in the context of an iterative design process. It is based on the observation that if only a small number of design parameters are changed, then it is not required to invoke the full schedulability analysis machinery. Rather, certain data structures can be created when the algorithm is run for the first time, and on subsequent invocations of the algorithm it is possible to exploit these data structures and run only a small subset of the regular schedulability analysis algorithm. We refer to this as *interactive* schedulability analysis because it would typically be used in an interactive mode—a designer would keep on modifying the values of a small number of system parameters and use this algorithm to test whether the system becomes schedulable.

This concept of interactive schedulability analysis is fairly general and can be applied to a number of well-known task models. In this thesis, we have chosen the recently proposed recurring real-time task model [9] to illustrate this scheme. It has been shown in [9] that this model generalizes a number of task models. Further, it can be used to model realistic applications with conditional branches and fine-grained deadline constraints.

Before proceeding further, we would like to clarify what we mean by "modifying the values of system parameters" in the context of scheduling a set of task graphs.

The relevant system parameters are determined by the underlying task model. For example, in the recurring real-time task model, vertices of task graphs are annotated with worst-case *execution times* and *deadlines*. The edges are annotated with *minimum intertriggering separation times* and each task graph is associated with a *period*, which specifies the minimum time interval between two consecutive triggerings of the graph. When the schedulability analysis of a task set returns a negative answer (i.e. *not* schedulable), a designer would typically relax a few deadline constraints associated with some of the vertices of the task graphs and run the algorithm once again. Other possible modifications might consist of increasing the values of some intertriggering separations, or increasing the period associated with a task graph, or decreasing the execution times associated with some of the vertices (possibly by rewriting/optimizing the code corresponding to those vertices). It might even be possible to split a vertex into two or more vertices, i.e. change the structure of a task graph.

Note that once a task set becomes schedulable, it is possible that a designer might now want to *constrain* (or reduce) the values of some of the above-mentioned parameters like deadlines, intertriggering separations, or task periods. This is in order to test whether the task set still remains schedulable with a tighter deadline, intertriggering separation, or period constraint. Often such an iterative process is used to obtain the tightest set of constraints under which a task set remains schedulable.

Overview of the Proposed Scheme

In this thesis, we discuss our proposed interactive scheme in the context of dynamic priority feasibility analysis in a preemptive uniprocessor environment. A standard methodology based on the *processor demand criteria* (see [10] and [17]) has emerged

for the feasibility analysis of such systems. Towards this, the worst-case workload that can possibly be generated by a task (graph) is represented by a function called the *demand-bound function*. The demand-bound function of a task T, denoted by $T.dbf(t)$, takes as an argument a positive real number t and returns the maximum possible cumulative execution requirement of jobs that can be legally generated by T and which have their ready-times and deadlines both within a time interval of length t. A set of concurrently executing tasks $\mathcal{T}$ is then schedulable under a fully preemptive uniprocessor model if and only if for all $0 < t \leq t_{\max}$, $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$, where $t_{\max}$ is a function of the execution requirements of the tasks in $\mathcal{T}$ and their periods. This scheme therefore involves two stages:

(i) Computing $T.dbf(t)$ for all $t \leq t_{\max}$ and $T \in \mathcal{T}$, and

(ii) Checking that $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t, \ \forall \ 0 < t \leq t_{\max}$.

For the recurring real-time task model, it turns out that for an arbitrary task graph T, computing $T.dbf(t)$ for any t is NP-hard (see [20]). Further, $t_{\max}$ is pseudo-polynomial in the size of problem. Hence, a pseudo-polynomial number of checks have to be performed in stage (ii).

While computing $T.dbf(t)$ for different values of t in stage (i), we construct a table for each task graph $T \in \mathcal{T}$ (the details of which are described later in this chapter). In an iterative design cycle, once the deadline $d(v)$ of a vertex $v \in T$ is changed and the schedulability analysis algorithm is invoked, the table corresponding to T need not be recomputed from scratch. Rather, only parts of it are updated—which is significantly faster than recomputing the entire table. For any t, $T.dbf(t)$ (where T is the task graph with the changed $d(v)$) can now be computed from this updated table.

Similarly, we also avoid checking the condition $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$ for *all* $0 < t \leq t_{\max}$. When the deadline $d(v)$ of a vertex $v \in T$ is changed, we compute the values of t at which the condition for schedulability i.e. $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$ can possibly change due to $d(v)$. We then check the schedulability condition only for these values of t, which again can be considerably faster than checking this condition for all $t \leq t_{\max}$.

Related Work

To the best of our knowledge, the concept of interactive schedulability analysis—in the form that we present in this thesis—has not been investigated before. The need for appropriate tool sets for interactive timing analysis has been emphasized in [79] and several other papers. [79] introduced an interactive tool, which helps to debug timing errors in real time programs. However, no formal or algorithmic results were presented. Neither did [79] present any result on how to speedup *interactive* timing analysis.

Most of the previous research on obtaining efficient algorithms for schedulability analysis for different real-time task models focused on designing either efficient pseudo-polynomial algorithms, or polynomial time solutions for restricted versions of task models. More recently, the concept of *approximate schedulability analysis* has been investigated in a number of papers (see, for example, [21], [4], and [32]). Unlike exact schedulability analysis, approximate schedulability analysis might return false positives or false negatives. Here, the basic idea is that if the schedulability analysis algorithm is occasionally allowed to return a false answer, then such an algorithm can be designed to run in polynomial time. For example, if the algorithm is allowed to return false positives then in some cases although a task set is *not* schedulable, the algorithm incorrectly returns *schedulable*. However,

it can be guaranteed that even in such cases no task will miss its deadline by more than a prespecified time interval. Further, for *most* task sets the algorithm will return the correct answer. A similar algorithm that only returns false negatives can also be designed.

None of the above research directions however exploit the fact that often the schedulability analysis algorithm is repeatedly invoked, with minor modifications in the task graphs. This is the scenario we address in this thesis. Although not directly related to the problem we address in this thesis, recently there has been some work on computing the *space* of task periods and worst-case execution times that lead to schedulable systems (this is often referred to as computing the *schedulable region*) [14]. The problem we address here, on the other hand, is an online or an interactive debugging scenario, where the designer is concerned with identifying *one* set of system parameters that lead to a schedulable design.

Organization of this Chapter

The rest of this chapter is organized as follows. In the next section we give some necessary background and an overeview of our scheme. This is followed by the related work in this domain. In Section 2.1, we describe the recurring real-time task model and its schedulability analysis. Towards this, we present a dynamic programming algorithm for computing the *demand-bound function* for this model in Sections 2.1.2 and 2.1.3. In Section 2.2 we then present our scheme for interactive schedulability analysis, which partly makes use of the dynamic programming algorithm. Our experimental results are described in Section 2.3. When a task set is not schedulable, it is often helpful if the system designer can be provided feedback on the potential system parameters that can be changed to obtain a schedulable system. In Section 2.4 we outline some techniques for providing such feedback,

and finally, we conclude this chapter in Section 2.5.

2.1 The Recurring Real-Time Task Model and its Schedulability Analysis

The recurring real-time task model was recently proposed by Baruah in [8, 9]. It is especially suited for accurately modeling conditional real-time code with recurring behavior, i.e. where code blocks have conditional branches and run in an infinite loop, as is the case in many embedded applications. Further, this model also generalizes a number of well-known task models such as the multiframe model [55], the generalized multiframe model [10] and the recurring branching task model [7].

A recurring real-time task T is represented by a task graph which is a directed acyclic graph with a unique source (a vertex with no incoming edges) and a unique sink (a vertex with no outgoing edges) vertex. Associated with each vertex v of this graph is its execution requirement $e(v)$, and deadline $d(v)$. Whenever the vertex v is *triggered*, it generates a job which has to be executed for $e(v)$ amount of time within $d(v)$ time units from the triggering-time. Each directed edge (u, v) in the graph is associated with a minimum intertriggering separation $p(u, v)$, denoting the minimum amount of time that must elapse before the vertex v can be triggered after the triggering of the vertex u.

The semantics of the execution of such a task graph state that the source vertex can be triggered at any time, and if some vertex u is triggered then the next vertex v can be triggered only if there exists a directed edge (u, v) and at least $p(u, v)$ amount of time has passed since the triggering of the vertex u. If there are directed edges (u, v_1) and (u, v_2) from the vertex u (representing a conditional branch) then

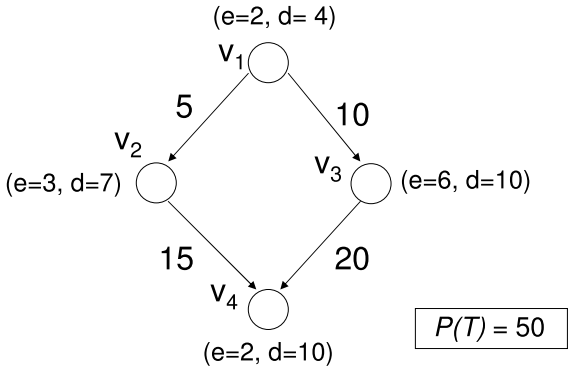


Figure 2.1: An example recurring real time task.

only one among v_1 and v_2 can be triggered, after the triggering of u. The triggering of the sink vertex can be followed by the source vertex getting triggered again but any two consecutive triggerings of the source vertex should be separated by at least $P(T)$ units of time, called the *period* of the task graph.

Therefore, a sequence of vertices $v_1, v_2, \ldots, v_k$ getting triggered at time instants $t_1, t_2, \ldots, t_k$, is legal if and only if there are directed edges (v_i, v_{i+1}), and $t_{i+1} - t_i \geq p(v_i, v_{i+1})$ for $i = 1, \ldots, k-1$. The only exception is that v_{i+1} can also be the source and v_i the sink vertex, and in that case if there exists some vertex $v_j, j < i$, in the sequence such that v_j is also the source vertex then $t_{i+1} - t_j \geq P(T)$ must be additionally satisfied. The real-time constraints require that the job generated by triggering vertex $v_i, i = 1, \ldots, k$, be assigned the processor for $e(v_i)$ amount of time within the time interval $(t_i, t_i + d(v_i)]$.

Once jobs are generated, they execute independently of each other (and therefore a restriction like first-come-first-served can not hold). Therefore, to ascertain that a job generated by a vertex u completes execution before a job generated by a vertex v, when u and v belong to the same task graph and there is a directed edge from u to v, then either of the following conditions must hold: $p(u, v) \geq d(u)$, which guarantees that the vertex v can be triggered only after the job generated by vertex u has completed execution, or that $d(u) \leq p(u, v) + d(v)$, which guarantees that the absolute deadline of the job generated by vertex v is larger than or equal

to the absolute deadline of the job generated by vertex u. In the real-time systems literature the first requirement is referred to as the *frame separation property* [74] and the second as the *localized Monotonic Absolute Deadlines property* (*l-MAD*) [10]. In this thesis, we assume either one of these two properties to hold.

Two points may be noted here. First, the original recurring real-time task model and its schedulability analysis, as proposed by Baruah in [9], is based on the *frame separation property* assumption. Second, our assumption that the *l-MAD* property leads to a job generated by a vertex u completing its execution before a job generated by a vertex v (when there is a directed edge from u to v) is based on the implicit assumption of the underlying scheduler uses the earliest deadline first (EDF) policy. We believe that this is a realistic assumption because EDF is known to be the optimal preemptive scheduling policy (i.e. if a task set is schedulable then EDF results in a feasible schedule) and it is widely used in real-life systems. Clearly, if the scheduling policy is not EDF then the *l-MAD* property along with the *processor demand criteria* for schedulability does not guarantee that a job generated by a vertex u will complete its execution before a job generated by v whenever there is a directed edge from u to v. Hence, we will from now on assume that the scheduling policy being used is EDF whenever the *l-MAD* property is assumed to hold true.

Figure 2.1 illustrates an example recurring real-time task. In this task, vertex v_3, for instance, has an execution requirement $e(v_3) = 6$, which must be met within 10 time units (its deadline) from its triggering time. The edge (v_1, v_3) has been labeled 10, which implies that the vertex v_3 can be triggered only after a minimum of 10 time units from the triggering of v_1 (i.e. the minimum intertriggering separation time). Edges (v_1, v_2) and (v_1, v_3) from vertex v_1 imply that either v_2 or v_3 can be triggered after v_1. The period of the task (the minimum time interval between two consecutive triggerings of the source vertex) is 50.

2.1.1 Task Sets and Schedulability Analysis

A task set $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ consists of a collection of task graphs, the vertices of which can get triggered independently of each other. A triggering sequence for such a task set $\mathcal{T}$ is legal if and only if for every task graph T_i, the subset of vertices of the sequence belonging to T_i constitute a legal triggering sequence for T_i. In other words, a legal triggering sequence for $\mathcal{T}$ is obtained by merging together (ordered by triggering times, with ties broken arbitrarily) legal triggering sequences of the constituting tasks.

The schedulability analysis of a task set $\mathcal{T}$ is concerned with determining whether the jobs generated by all possible legal triggering sequences of $\mathcal{T}$ can be scheduled such that their associated deadlines are met. Algorithms for the schedulability analysis of such task sets, in a preemptive uniprocessor setup, are based on certain *task independence assumptions*. These are: (i) The runtime behavior of a task is independent of any other tasks in the system. (ii) The constraints according to which legal job sequences are generated can be specified without any references to *absolute time*. Assumption (i) states that each task generates jobs independently of the jobs generated by other tasks in the system. Therefore, it is not permissible, for example, to require a task to generate a job in response to a job generated by another task. Assumption (ii) states that all temporal specifications defining the rules according to which jobs are generated by a task can only be relative to the time at which the task begins execution, or can be relative to the ready-time of another job of the same task. Therefore, a constraint like the ready-times of two consecutive jobs of a task must be separated by at least p time units, conforms to this requirement. Lastly, the time at which a task begins execution (i.e. the first job is generated) is not *a priori* known. For example, a task can begin execution in response to some external event.

Note that although the task independence assumptions restrict the job generation process of a task (for example, by specifying the minimum separation between the generation of two jobs), they make no assumptions about the interactions between the jobs once they are generated. Once a job is generated, it executes independently of any other job in the system, including those generated by the same task.

Given a sequence of jobs generated by a task set $[(T_i, a_i, e_i, d_i), (T_j, a_j, e_j, d_j), \ldots]$ (T_i refers to a task, a_i is the ready time of a job, e_i is its execution requirement, and d_i is its absolute deadline), the task independence assumptions imply that the sequence is legal if and only if all subsequences formed by jobs from the individual tasks are also legal (follows from Assumption (i)). Assumption (ii) implies that if $[(a_1, e_1, d_1), (a_2, e_2, d_2), \ldots]$ is a legal sequence of jobs generated by a task, then the sequence $[(a_1 - t, e_1, d_1 - t), (a_2 - t), e_2, d_2 - t), \ldots]$ is also legal, where t is any real number.

It directly follows from the description of the recurring real-time task model in Section 2.1 that the model indeed satisfies the above task independence assumptions (and so does a wide variety of other task models such as the sporadic, multiframe, generalized multiframe, and the recurring branching models). The recurring real-time task model therefore lends itself to schedulability analysis based on the *processor demand criteria*, that we outlined in Section 2.

2.1.2 The *demand-bound function*

Recall from Section 2 that a task set $\mathcal{T}$ is schedulable if and only if $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$ for all $0 < t \leq t_{\max}$. It can be proved that

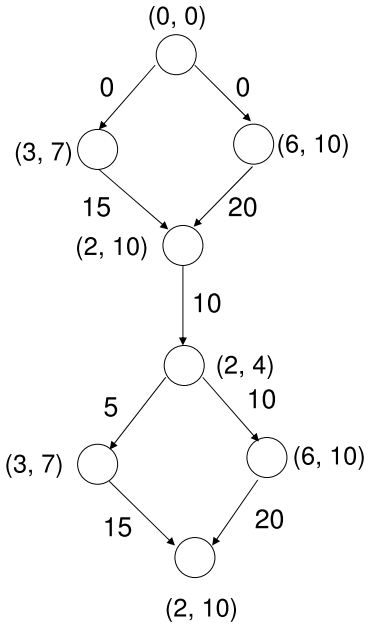
$$t_{\max} = \frac{\sum_{T \in \mathcal{T}} 2E(T)}{1 - \sum_{T \in \mathcal{T}} \frac{E(T)}{P(T)}}$$

where $E(T)$ is the maximum cumulative execution requirement arising from a sequence of vertices on any path from the source to the sink vertex of the task graph T (see [9] for details).

For any task graph T, computing the value of $T.dbf(t)$ for some (large) value of $t \leq t_{\max}$ might involve multiple traversals (loops) through the task graph. It was shown in [9] that if for a task graph T, $T.dbf(t)$ is known for all "small values" of t then it is possible to calculate from these, the value of $T.dbf(t)$ for *any* t. "Small values" of t for a task graph T are those for which the sequence of vertices that contribute towards computing $T.dbf(t)$ contain the source vertex at most once. The value of $T.dbf(t)$ for larger values of t is made up of some multiple of $E(T)$ plus $T.dbf(t')$ where t' is "small" in the sense described above. $T.dbf(t)$ for any t can hence be computed as follows (for a more detailed description, refer to [9]).

$$\begin{aligned} T.dbf(t) = \max\{ &\lfloor t/P(T) \rfloor E(T) + T.dbf(t \bmod P(T)), \\ &(\lfloor t/P(T) \rfloor - 1)E(T) + T.dbf(P(T) + t \bmod P(T))\} \end{aligned} \tag{2.1}$$

To compute $T.dbf(t)$ for "small" values of t, [9] constructs a new task graph by taking two copies of the task graph of T and adding an edge from the sink vertex of the first graph to the source vertex of the second and finally replacing the source vertex of the first with a "dummy" vertex with execution requirement and deadline equal to zero. The intertriggering separations on all edges outgoing from this source vertex is also made equal to zero. (Two copies of the task graph in Figure 2.1 are joined in the fashion described above, and the resulting task graph is shown in Figure 2.2). $T.dbf(t)$ for all values of t are then calculated by enumerating all possible paths in this new graph. For arbitrary task graphs, this incurs a computation time which is exponential in the number of vertices in the task graph. The list alongside the task graph in Figure 2.2 gives us few values of $T.dbf(t)$ corresponding to some selected "small" values of t for this task graph.



t	*T.dbf(t)*
2	0
4	2
7	3
10	6
20	8
30	10

Figure 2.2: Finding $T.dbf(t)$ for "small" values of t.

For instance, when $t = 4$, the $T.dbf(t)$ is 2, implying that within any time interval of 4 units the total execution requirement of jobs which have both their ready times and deadlines within this interval is 2. This means that there is no other permissible sequence of jobs which will have a demand greater than 2 within an time interval of 4. Similar explanation applies to other pairs of values listed in the table.

2.1.3 Computing the *demand-bound function*

In this section we present a dynamic programming algorithm for computing the demand-bound function $T.dbf(t)$ for any task graph T. It was shown in [20] that computing $T.dbf(t)$ for any t is NP-hard for an arbitrary task graph T and a dynamic programming algorithm for computing it was given. The algorithm that we present here includes a minor extension of the algorithm in [20], so that it may be used by our interactive framework. The algorithm runs in pseudo-polynomial

time and constructs a table, which is then used by our interactive schedulability analysis framework that we describe in Section 2.2.

The algorithm given below (Algorithm 1) constitutes stage (i) of the two stages that we listed in Section 2. We first give an algorithm for computing the demand-bound function of a task graph for "small values" of t. Using this, we then compute the demand-bound function for any value of t as explained in Section 2.1.2.

Given a task graph T, let T' denote the graph formed by joining two copies of T by adding an edge from the sink vertex of the first graph to the source vertex of the second, and replacing the source vertex of the first copy by a "dummy" vertex. If the frame separation property is followed then the newly added edge is labeled with an intertriggering separation of $p = d(v_{sink})$, and if the l-MAD property is followed then it is labeled with $p = \max\{0, d(v_{sink}) - d(v_{source})\}$, where v_{source} and v_{sink} denotes the source and the sink vertices of T. Now we give a pseudo-polynomial time algorithm based on dynamic programming, for computing $T'.dbf(t)$ for values of t that do not involve any looping through T', i.e. we consider only "one-shot" executions of T'.

Let there be n vertices in T' denoted by $v_1, \ldots, v_n$, and without any loss of generality we assume that there can be a directed edge from v_i to v_j only if $i < j$. Following our notation described in Section 2.1, associated with each vertex v_i is its execution requirement $e(v_i)$ which here is assumed to be integral (a pseudo-polynomial algorithm is meaningful only under this assumption), and its deadline $d(v_i)$. Associated with each edge (v_i, v_j) is the minimum intertriggering separation $p(v_i, v_j)$.

Let $t_{i,e}$ be the minimum time interval within which the task T' can have an execution requirement of exactly e time units due to some legal triggering sequence, considering only a subset of vertices from the set $\{v_1, \ldots, v_i\}$, if all the triggered

vertices are to meet their respective deadlines. Let $t^i_{i,e}$ be the minimum time interval within which a sequence of vertices from the set $\{v_1, \ldots, v_i\}$, and ending with the vertex v_i, can have an execution requirement of exactly e time units, if all the vertices have to meet their respective deadlines. Lastly, let $E = \max_{i=1,\ldots,n} e(v_i)$. Clearly, nE is an upper bound on $T'.dbf(t)$ for any $t \geq 0$ for one-shot executions of T'.

It can be shown by induction that Algorithm 1 correctly computes $T'.dbf(t)$, and has a running time of $O(n^3E)$. This algorithm, in addition, computes the values of a set of boolean variables which are referred to as $flag_{i,e}$. For any given value of i and e, $flag_{i,e}$ is set to $PREVIOUS$ if $t_{i-1,e} < t^i_{i,e}$ else it is set to $SELF$. The use of this variable will be explained in Section 2.2 when we describe our interactive schedulability analysis framework.

2.2 Interactive Schedulability Analysis for the Recurring Real-Time Task Model

Having introduced all the necessary background, we are now in a position to describe our framework for interactive schedulability analysis. Recall from Section 2 that this framework is composed of two steps: (i) Computing $T.dbf(t)$ for all $t \leq t_{\max}$ and $T \in \mathcal{T}$, and (ii) Checking that $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$, $\forall\ 0 < t \leq t_{\max}$.

When the schedulability analysis algorithm is invoked for the first time, for each task graph $T \in \mathcal{T}$, Algorithm 1 is used to compute the values of $t^i_{i,e}$, $t_{i,e}$, and $flag_{i,e}$, which constitutes step (i). These are then stored in a table, which we will refer to as the *dbf-table*. For any task graph T, its *dbf-table* consists of rows which correspond to the vertices of T (ranging from 1 to n, assuming that T consists of

Algorithm 1 Computing $T'.dbf(t)$

Require: Task graph T', and a real number $t \geq 0$

for $e \leftarrow 1$ to nE **do**

$t_{1,e} \leftarrow \begin{cases} d(v_1) & \text{if } e(v_1) = e \\ \infty & \text{otherwise} \end{cases}$

$flag_{1,e} \leftarrow \begin{cases} \texttt{SELF} & \text{if } e(v_1) = e \\ \texttt{PREVIOUS} & \text{otherwise} \end{cases}$

$t^1_{1,e} \leftarrow t_{1,e}$

end for

for $i \leftarrow 1$ to $n-1$ **do**

  for $e \leftarrow 1$ to nE **do**

    Let there be directed edges from the vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ to v_{i+1}

$t^{i+1}_{i+1,e} \leftarrow \begin{cases} \min\{t^{i_j}_{i_j,e-e(v_{i+1})} - d(v_{i_j}) + p(v_{i_j}, v_{i+1}) + \\ d(v_{i+1}) \,|\, j = 1, \ldots, k\} \text{ if } e(v_{i+1}) < e, \\ d(v_{i+1}) \text{ if } e(v_{i+1}) = e, \text{ and } \infty \text{ otherwise} \end{cases}$

    $t_{i+1,e} \leftarrow \min\{t_{i,e}, t^{i+1}_{i+1,e}\}$

    if $t_{i+1,e} = t^{i+1}_{i+1,e}$ **then**

      $flag_{i+1,e} \leftarrow$ `SELF`

    else

      $flag_{i+1,e} \leftarrow$ `PREVIOUS`

    end if

  end for

end for

$T'.dbf(t) \leftarrow \max\{e \,|\, t_{n,e} \leq t\}$

n vertices) and columns which correspond to the different execution requirements that may be demanded by T due to a triggering of these vertices (ranging from 1 to nE). A cell (i, e) in this table contains three different values: $t_{i,e}$, $t^i_{i,e}$ and $flag_{i,e}$.

Now suppose that the schedulability analysis algorithm fails in step (ii), i.e. there exists some $\hat{t} \leq t_{\max}$ such that $\sum_{T \in \mathcal{T}} T.dbf(\hat{t}) > \hat{t}$. Then the system designer might choose to modify certain system parameters and run the schedulability analysis algorithm once again. Typically, this would involve rerunning steps (i) and (ii) from scratch. However, using our scheme for interactive schedulability analysis, we would instead only *update* the existing *dbf-tables* and recompute the appropriate

$T.dbf(t)$ values from the updated tables. In most cases, this would be considerably faster than recomputing all the $T.dbf(t)$ values from scratch. Clearly, only the *dbf-tables* of task graphs that have been modified will have to be updated. Once the appropriate $T.dbf(t)$s have been recomputed, depending on the nature of the modifications made (e.g. deadlines have only been *relaxed*), the checking involved in step (ii) can be resumed from $\hat{t}$ onwards. There is no need to check the condition $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$ for values of $t < \hat{t}$ since the task set already passed the schedulability test for these values of t.

The second possible scenario is when the task set $\mathcal{T}$ satisfies the schedulability test in step (ii) for all $t \leq t_{\max}$ (i.e. $\mathcal{T}$ is schedulable). In this case, the designer might still want to modify certain system parameters (e.g. constrain the deadlines associated with some of the vertices) and run the schedulability analysis algorithm once again. This might be to test if the task set remains schedulable under a tighter set of constraints. In this case, we would again update the *dbf-tables* and recompute the appropriate $T.dbf(t)$ values from the updated tables, as before. However, step (ii) will now become more involved—rather than checking the condition $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$ for *all* $t \leq t_{\max}$, we check this condition only for those values of t at which the sum $\sum_{T \in \mathcal{T}} T.dbf(t)$ might have changed.

In the following two subsections we discuss the details of the two above-mentioned scenarios. Recall that in this thesis we shall only be concerned with deadlines associated with vertices of task graphs being modified.

2.2.1 Relaxing the Deadline of a Vertex

Given a task graph T, let us assume that T' is obtained by joining two copies of T, followed by adding an edge from the sink vertex of the first copy to the source vertex of the second and replacing the source vertex of the first copy by a

Algorithm 2 *dbf-table* update: Deadline relaxed case

Require: Task graph T', a real number $t \geq 0$, and a vertex number *node* such that deadline associated with vertex v_{node} in T' has been relaxed.

for $e \leftarrow 1$ to nE **do**
  for $i \leftarrow node - 1$ to $n - 1$ **do**
    Let there be directed edges from the vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ to v_{i+1}
    $$t_{i+1,e}^{i+1} \leftarrow \begin{cases} \min\{t_{i_j,e-e(v_{i+1})}^{i_j} - d(v_{i_j}) + p(v_{i_j}, v_{i+1}) + \\ d(v_{i+1}) \,|\, j = 1, \ldots, k\} \text{ if } e(v_{i+1}) < e, \\ d(v_{i+1}) \text{ if } e(v_{i+1}) = e, \text{ and } \infty \text{ otherwise} \end{cases}$$
    $t_{i+1,e} \leftarrow \min\{t_{i,e}, t_{i+1,e}^{i+1}\}$
    if $t_{i+1,e} = t_{i+1,e}^{i+1}$ **then**
      $flag_{i+1,e} \leftarrow$`SELF`
    else
      $flag_{i+1,e} \leftarrow$`PREVIOUS`
    end if
    if $i + 1 = node$ **then**
      if $flag_{i+1,e} =$ `PREVIOUS then`
        `break;`
      `else if` $flag_{i+2,e} =$ `SELF then`
        `break;`
      `end if`
    else if $flag_{i+2,e} =$ `SELF then`
      `break;`
    `end if`
  end for
end for
$T'.dbf(t) \leftarrow \max\{e \,|\, t_{n,e} \leq t\}$

"dummy" vertex (as described in Section 2.1.3). We also assume that the *dbf-table* of T' has been computed. Now let us suppose that the deadline $d(v)$ associated with a vertex $v \in T$ has been relaxed. Unless v is the source vertex of T, this results in the deadlines of two vertices in T' (both of which correspond to the same vertex v in T) getting changed. Algorithm 2 then correctly updates *dbf-table* to reflect this change. Note that it has to be invoked either once or twice depending on whether v is a source vertex of T or not.

To understand how Algorithm 2 works, let us assume that the deadline associated with the vertex v_{node} in T' has been relaxed, where the vertices of T' are $v_1, \ldots, v_n$, with a directed edge from v_i to v_j only if $i < j$. The algorithm starts traversing the rows of the *dbf-table* starting from the row *node* and ending at row n (lines 1 and 2). Hence, it does not recompute the values in the cells in rows 1 to $(node - 1)$. This is because the values in these cells do not depend on the deadline of node v_{node} (i.e. $d(v_{node})$), and therefore remain unchanged even after $d(v_{node})$ has been relaxed. Note that this immediately follows from the fact that *dbf-table* is computed using a dynamic programming algorithm, where the computation of the ith row depends only on the parameters associated with the subset of vertices $\{v_1, v_2, \ldots, v_i\}$.

Lines 3 to 10 of this algorithm are the same as lines 8 to 15 of Algorithm 1. They compute the values of the $(i + 1, e)$th cell of the *dbf-table* using the values in the cells which have been previously computed or updated. During the first iteration of the loop spanning across lines 2 to 20, $i + 1 = node$. From lines 11 to 19 of Algorithm 2 it may be seen that cells corresponding to vertices numbered higher than *node* are selectively recomputed based on the values of the $flag$ variables. In what follows, we first explain how the value of the $flag$ variable is exploited for this selective update and then we work through an example.

The main principle behind the selective update relies on two observations:

1. Let k and e be such that $node < k \leq n$ and $1 \leq e \leq nE$. Therefore, (k, e) is a cell in the *dbf-table* that is above the row *node*. Our observation is that although the variable $t_{k,e}$ depends on both $t^k_{k,e}$ and $t_{k-1,e}$ (see line 5), upon relaxation of $d(v_{node})$, $t_{k,e}$ would change if and only if $t_{k-1,e}$ has changed. In other words, the values in the cell (k, e) will change only if the cell in the

previous row and the *same* column has changed. This corresponds to the case that $t^k_{k,e}$ does not depend on the deadline associated with v_{node} (see line 4 in Algorithm 2 for how the value of $t^k_{k,e}$ is determined).

2. The variable $flag_{k,e}$ for any cell in the *dbf-table* is assigned to $PREVIOUS$ if $t_{k,e}$ depends on $t_{k-1,e}$, i.e. we say that it depends on the *previous* cell in the column e. Similarly, $flag_{k,e}$ is assigned to $SELF$ if $t_{k,e}$ depends on $t^k_{k,e}$, i.e. we say that it depends on the same cell or on *self*.

These two observations should be used to reason about the behavior of Algorithm 2. In the row *node*, the algorithm traverses all the cells for $e = \{1, \ldots, nE\}$ and updates the values $t_{node,e}$, $t^{node}_{node,e}$ and $flag_{node,e}$ in each cell. For each cell $(node, e)$, the algorithm also updates the cell higher up on the column (i.e. cell $(node+1, e)$) depending on the updated $flag$ value on the cell $(node, e)$ and the existing $flag$ value in the higher cell $(node + 1, e)$. This is explained below in further detail.

- If $flag_{node,e} = PREVIOUS$, it implies that $t_{node,e} = t_{node-1,e}$ (follows from observation (2)) and since $t_{node-1,e}$ remains unchanged with any change in $d(v_{node})$, $t_{node,e}$ need not be modified as well. Hence, we need not update any cell in the column e (follows from observation (1)).

- On the other hand, if $flag_{node,e} = SELF$, it implies that $t_{node,e}$ will change with the relaxation of $d(v_{node})$. This follows from the facts that $t_{node,e} = t^{node}_{node,e}$ (observation (2)), and that $t^{node}_{node,e}$ has now been updated. Now, there might be two different scenarios:

  1. If $flag_{node+1,e} = SELF$, we need not update any cell on the column e. $flag_{node+1,e} = SELF$ in the cell $(node + 1, e)$ implies $t^{node+1}_{node+1,e} < t_{node,e}$ before the change. After we have relaxed $d(v_{node})$, $t_{node,e}$ must have increased or remained unchanged. Hence, $t^{node+1}_{node+1,e} < t_{node,e}$ still holds,

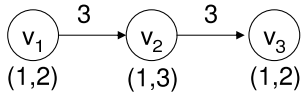


Figure 2.3: The task graph T.

and there is no need to update the cell $(node + 1, e)$ or the cells higher up in the column.

2. If $flag_{node+1,e} = PREVIOUS$, $t_{node+1,e}$ would change as well (observation (1)). This change might then similarly propagate along the higher cells of the column e, if the value of their respective flags equals $PREVIOUS$.

This selective updation of the *dbf-table* is what is taken care of in the lines 11 to 19 of Algorithm 2. Clearly, this avoids recomputing the table from stratch which often saves a large chunk of computation.

Illustrative Example

To appreciate why Algorithm 2 will often be computationally less expensive compared to recomputing the entire *dbf-table*, let us consider a small example. Let T be a task graph with 3 vertices, v_1, v_2, v_3, such that an edge from v_i to v_j exists if and only if $j = i + 1$. Let $e(v_i) = 1$ for all $1 \leq i \leq 3$ in T. The deadlines of the vertices are $d(v_1) = 2$, $d(v_2) = 3$, and $d(v_3) = 2$. The minimum intertriggering separation times associated with the edges are $p(v_1, v_2) = 3$, and $p(v_2, v_3) = 3$ (see Figure 2.3). Let T' be the graph that is formed by joining two copies of this task graph T in the fashion described in Section 2.1.3. T' is shown in Figure 2.4.

The *dbf-table* of T' is shown in Table 2.1. For any $1 \leq i \leq 6$ and $1 \leq e \leq 6$, the (i, e)th cell of this table contains the values of $t_{i,e}$, $t^i_{i,e}$, and $flag_{i,e}$ (in this order), where P and S denotes the $PREVIOUS$ and $SELF$ values of $flag_{i,e}$ respectively.

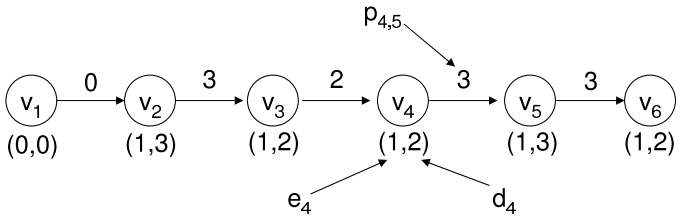


Figure 2.4: The task graph T'.

$i\uparrow$	$e \rightarrow$					
	1	**2**	**3**	**4**	**5**	**6**
6	2, 2, S	4, 5, P	7, 8, P	10, 10, S	13, 13, S,	∞, ∞, S
5	2, 3, P	4, 6, P	7, 8, P	11, 11, S	∞, ∞, S,	∞, ∞, S
4	2, 2, S	4, 4, S	7, 7, S	∞, ∞, S	∞, ∞, S,	∞, ∞, S
3	2, 2, S	5, 5, S	∞, ∞, S	∞, ∞, S	∞, ∞, S,	∞, ∞, S
2	3, 3, S	∞, ∞, S	∞, ∞, S	∞, ∞, S	∞, ∞, S,	∞, ∞, S
1	∞, ∞, S	∞, ∞, S	∞, ∞, S	∞, ∞, S	∞, ∞, S,	∞, ∞, S

Table 2.1: *dbf-table* of T'.

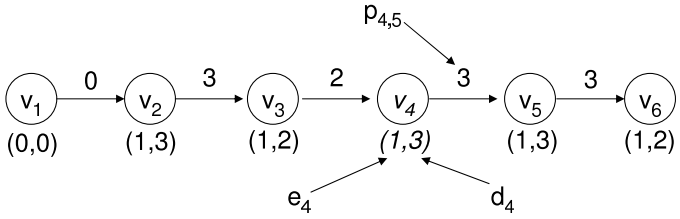


Figure 2.5: Graph T' after relaxing the deadline associated with the vertex v_4 from 2 to 3.

Assume that the deadline of the source vertex of T has been changed from 2 to 3. This implies that the deadline of v_4 in T' is relaxed from 2 to 3. The task graph with its new deadlines is illustrated in Figure 2.5. We then update the *dbf-table* using Algorithm 2. The new *dbf-table* is shown in Table 2.2. Only the cells of Table 2.1 which were updated using Algorithm 2 are shown using a bold-italic font in Table 2.2.

Since only the deadline of v_4 was relaxed, the execution demand arising from any vertex numbered less than 4 remains unchanged. Hence, the only potential cells of Table 2.1 which might be effected are on or above row 4. Algorithm 2 first traverses row 4 of this table and recomputes the values of its cells. However, it does not

$i\uparrow$	$e \rightarrow$					
	1	**2**	**3**	**4**	**5**	**6**
6	2, 2, S	***5, 5, S***	***8, 8, S***	10, 10, S	13, 13, S,	∞, ∞, S
5	2, 3, P	***5, 6, P***	***8, 8, S***	11, 11, S	∞, ∞, S,	∞, ∞, S
4	***2, 3, P***	***5, 5, S***	***8, 8, S***	***∞, ∞, S***	***∞, ∞, S***	***∞, ∞, S***
3	2, 2, S	5, 5, S	∞, ∞, S	∞, ∞, S	∞, ∞,S,	∞, ∞, S
2	3, 3, S	∞, ∞, S	∞, ∞, S	∞, ∞, S	∞, ∞, S,	∞, ∞, S
1	∞, ∞, S	∞, ∞, S	∞, ∞, S	∞, ∞, S	∞, ∞, S,	∞, ∞, S

Table 2.2: The updated *dbf-table* after relaxing the deadline associated with the vertex v_4 from 2 to 3.

"propagate" a change upwards, along the column of a cell, if the $flag$ in the cell is now equal to $PREVIOUS$. This is because if the value of the $flag$ equals to $PREVIOUS$, then it implies that the value of $t_{4,e}$ is equal to $t_{3,e}$ which remains unaltered. Further, any $t_{j,e}$, where $j > 4$, need not be changed as a result of relaxing $d(v_4)$. Recall that this follows from observation (1). To verify observation (1) note that by definition (Section 2.1.3) $t^5_{5,1} = d(v_5) = 3$, $t^5_{5,2} = p_{4,5} + d(v_5) = 6$, etc. These remain unaltered even after the deadline of v_4 is changed and thus are same in both tables – Table 2.1 and Table 2.2. For example, we can verify from Table 2.2 that when $e = 1$, this is indeed the case. Hence, this clearly saves a significant amount computation compared to the case where the full *dbf-table* is recomputed.

The second scenario is when one of the cells has its $flag$ set to $SELF$. In our example, cells $(4, 2)$, and $(4, 3)$ illustrate this scenario. Let us consider cell $(4, 3)$, where $flag_{4,3} = SELF$ implies that $t_{4,3} = t^4_{4,3}$. $t^4_{4,3}$ being in row 4 was updated and hence the value of $t_{4,3}$ has changed as well and might in turn lead to changes in the cells higher up along this column. Therefore, we need to check whether any higher numbered vertices might also be effected. The cell $(5, 3)$ had $flag = PREVIOUS$ (see Table 2.1) and hence $t_{5,3}$ needs to be recomputed. Similarly, cell $(6, 3)$ is also recomputed. Note that cell $(4, 4)$, has its $flag$ set to $SELF$; however, since $flag_{5,4} = SELF$ we need not propagate the change along the higher numbered columns. This again saves a significant amount of computation time.

If the schedulability test for a task set $\mathcal{T}$ fails at $t = \hat{t}$ then in this case (i.e. when deadlines associated with vertices are only being relaxed) after the deadlines associated with one or more vertices are relaxed, the check in step (ii) of our scheme can be resumed at $t = \hat{t}$.

2.2.2 Constraining the Deadline of a Vertex

Let us now consider the case where the deadline of a vertex $v \in T$ is *constrained*. As in the previous case, depending on whether v is a source vertex in T or not, this would result in two vertices in T' getting affected (where T' is obtained by joining two copies of T). Again, let v_{node} be a vertex in T' whose deadline is constrained. Then Algorithm 3 updates the *dbf-table* corresponding to T'. Algorithm 3 is similar to Algorithm 2, except for a pair of extra conditions in lines 15 and 20. The use of these two conditions will be clarified in the following discussion.

The two observations listed in Section 2.2.1 hold true even in the case when the deadline of a vertex is constrained. Hence, based on the values of the $flag$ variables we can once again find out the appropriate conditions for updating the *dbf-table*.

- If $flag_{node,e} = PREVIOUS$ then this case is exactly similar to the corresponding case where a deadline is relaxed.
- On the other hand, if $flag_{node,e} = SELF$ then we know that $t_{node,e} = t^{node}_{node,e}$ and this implies that the value of $t_{node,e}$ has decreased as a result of constraining the deadline of v_{node}. In such a case, if $flag_{node+1,e} = PREVIOUS$ then the scenario is again similar to the corresponding case where the deadline of v_{node} was relaxed. Hence, the value $t_{node+1,e}$ will have to be updated. The change might then "propagate" along the higher cells of the column e, depending on the value of their flags.

Algorithm 3 *dbf-table* update: Deadline constrained case

Require: Task graph T', a real number $t \geq 0$, and a vertex number *node* such that deadline associated with vertex v_{node} in T' has been constrained.

1: **for** $e \leftarrow 1$ to nE **do**
2: **for** $i \leftarrow node - 1$ to $n - 1$ **do**
3: Let there be directed edges from the vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ to v_{i+1}
4: $t^{i+1}_{i+1,e} \leftarrow \begin{cases} \min\{t^{i_j}_{i_j,e-e(v_{i+1})} - d(v_{i_j}) + p(v_{i_j}, v_{i+1}) + \\ d(v_{i+1}) \,|\, j = 1, \ldots, k\} \text{ if } e(v_{i+1}) < e, \\ d(v_{i+1}) \text{ if } e(v_{i+1}) = e, \text{ and } \infty \text{ otherwise} \end{cases}$
5: $t_{i+1,e} \leftarrow \min\{t_{i,e}, t^{i+1}_{i+1,e}\}$
6: **if** $t_{i+1,e} = t^{i+1}_{i+1,e}$ **then**
7: $flag_{i+1,e} \leftarrow$`SELF`
8: **else**
9: $flag_{i+1,e} \leftarrow$`PREVIOUS`
10: **end if**
11: **if** $i + 1 = node$ **then**
12: **if** $flag_{i+1,e} =$ `PREVIOUS then`
13: `break;`
14: `else if` $flag_{i+2,e} =$ `SELF then`
15: `if` $t_{i+1,e} \geq t^{i+2}_{i+2,e}$ `then`
16: `break;`
17: `end if`
18: `end if`
19: **else if** $flag_{i+2,e} =$ `SELF then`
20: `if` $t_{i+1,e} \geq t^{i+2}_{i+2,e}$ `then`
21: `break;`
22: `end if`
23: `end if`
24: **end for**
25: **end for**
26: $T'.dbf(t) \leftarrow \max\{e \,|\, t_{n,e} \leq t\}$

- However, if $flag_{node,e} = SELF$ and if $flag_{node+1,e} = SELF$ as well (which implies that $t_{node+1,e} = t^{node+1}_{node+1,e}$), the scenario is different from when the deadline of v_{node} was relaxed. The reason for this being, after the deadline was constrained, it might now be that $t_{node,e}$ has decreased. Thus, $t^{node+1}_{node+1,e} < t_{node,e}$, which was true before the change, might no longer hold. Hence,

$t_{node+1,e}$ might be assigned to the new value $t_{node,e}$, instead of the existing value $t^{node+1}_{node+1,e}$ and we need to update the cell $(node+1, e)$. Similar reasoning also holds true when we select any cell (i, e) for updating where $i > node$. This explains the need for the extra pair of conditions.

Efficiently Performing Step (ii)

As discussed, here we would like to avoid performing the check $\sum_{T\in\mathcal{T}} T.dbf(t) \leq t$ for all values of $t \leq t_{\max}$. Let us assume that the deadline associated with a certain vertex of T has been constrained. We also assume that T belongs to a task set $\mathcal{T}$, which was originally schedulable. Algorithm 3 is then used to update the *dbf-table* associated with T. Now our goal is to identify those values of t at which the sum $\sum_{T\in\mathcal{T}} T.dbf(t)$ was modified; we would like to check the condition $\sum_{T\in\mathcal{T}} T.dbf(t) \leq t$ only at these values of t. Towards this, we first scan the updated *dbf-table* and identify those values of t for which $t < P(T)$ and either $T.dbf(t)$ or $T.dbf(t + P(T))$ have been updated. Let t_{change} be the first such value of t in this table. Let t_{check} be a possible value of t that we are interested in identifying. It then follows from Eqn. 2.1 in Section 2.1.2 that for each value of t_{change}, there will be multiple t_{check}s. These t_{check}s are given by:

$$t_{check} = t_{change} + kP(T)$$

where $k = 0, \ldots, N$ and N is the largest integer satisfying the inequality $t_{change} + NP(t) \leq t_{max}$.

The above procedure has to be repeated for all possible values of t_{change} in the updated *dbf-table* and the corresponding t_{check}s are identified. The schedulability test $\sum_{T\in\mathcal{T}} T.dbf(t) \leq t$ is then performed at these t_{check}s.

2.2.3 Running Times

Note that both the algorithms for updating the *dbf-table* (i.e. Algorithms 2 and 3), have a worst-case running time of $O(n^3E)$. Hence, in the worst-case, updating the *dbf-table* involves the same computational cost as that involved in computing this table from scratch. Clearly, at least from a theoretical standpoint, our scheme would have been more attractive had this been otherwise. However, as we have pointed out in Section 2.2.1, for most problems the actual running time incurred by our algorithms would be significantly less than what would be involved in recomputing the entire *dbf-table*. As an example, let us consider Algorithm 2. We saw that when the deadline of a vertex v_{node} was relaxed, then the cells $1, 2, \ldots, nE$ of row *node* were unconditionally recomputed. However, any cell on a row numbered higher than *node* will have to be updated depending on the conditions in lines 11 to 19 of the algorithm. Hence, updating a single column of the *dbf-table* will incur the worst-case cost only when the value of $t_{node,e}$ is less than $t_{i,e}$ for all $i > node$. Further, for the worst-case (in terms of updating the *dbf-table*) to occur, the worst-case update scenario of a column must happen for *all* columns $1, 2, \ldots, nE$. For most problem instances, such corner cases are unlikely to happen and as our experimental results show in Section 2.3, our scheme results in a significant speedup compared to recomputing the *dbf-table* for each change.

Similarly, in the worst-case, stage (ii) might also require that the schedulability condition $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$ to be checked for all $t \leq t_{\max}$. But once again, for most problem instances, this is unlikely to happen.

Finally, note that the space complexity of storing a *dbf-table* with n vertices is $O(n^2E)$. For each vertex i we store $t_{i,e}$, $t^i_{i,e}$, and $flag_{i,e}$, where e ranges from 1 to nE.

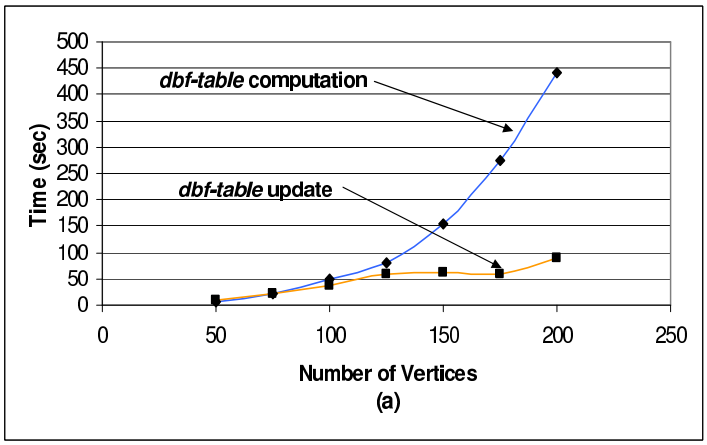
2.3 Experimental Results

We conducted two broad categories of experiments. In Section 2.3.1 we report some experimental results that were obtained by running the dynamic programming algorithm (Algorithm 1) and our proposed algorithms for interactive schedulability analysis (Algorithms 2 and 3) on a set of synthetic task graphs. In Section 2.3.2 we illustrate the benefits of efficiently performing *Step*(ii) of the schedulability analysis (which we described in Section 2.2.2).

2.3.1 Experiments with Step (i)

For our experiments we randomly generated synthetic task graphs using two parameters. The first is the maximum execution requirement, E, associated with any vertex of a graph. The second parameter is called the *connectivity factor*. If $v_1, \ldots, v_n$ are the vertices of a task graph such that there is an edge from v_i to v_j only if $j > i$, then while generating the graph, for each vertex v_j we construct an edge from v_i to v_j with a probability equal to the connectivity factor of the graph, for all $i = 1, \ldots, j-1$.

The parameters (i.e. E and the connectivity factor) used to generate our synthetic graphs were chosen such that the graphs represent realistic network packet processing applications. The details of this application may be found in [19]. A connectivity factor equal to 0.4 was used to generate all the task graphs since this results in graphs which are similar to those arising in practice. It may be noted here that a higher connectivity factor would clearly result in more paths in any graph. Hence, this would lead to higher savings from our scheme compared to when all the paths in a graph are exhaustively enumerated to compute the *demand-bound function*. E was set equal to either 200 or 600, representing two possible cases



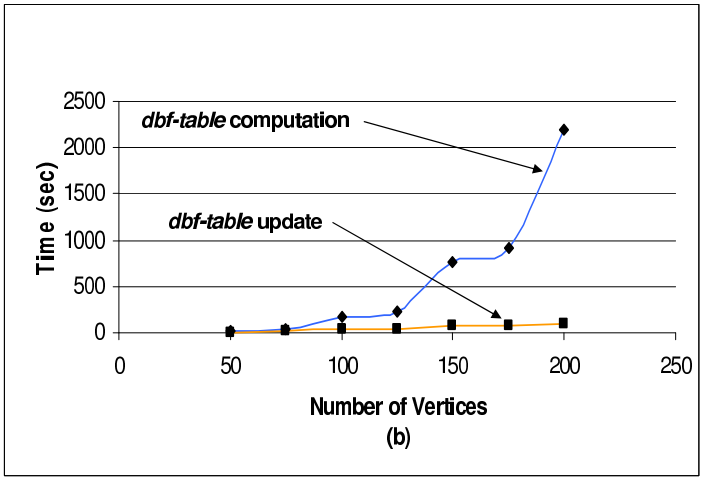


Figure 2.6: Running times for updating the *dbf-table* when the deadline of a vertex was relaxed (a) $E = 200$ and (b) $E = 600$.

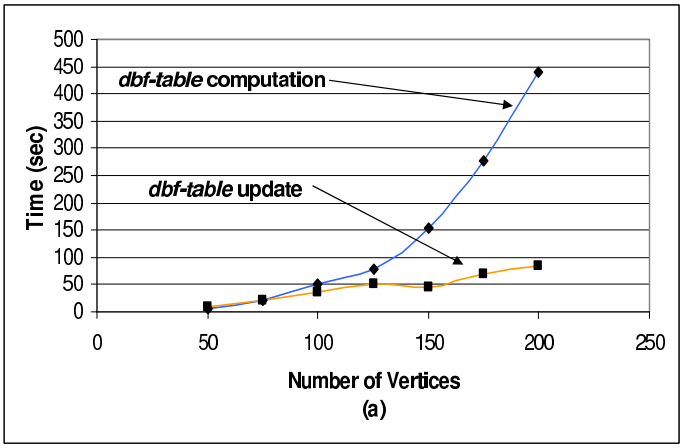
in the above-mentioned application.The inter-triggering time for each edge was generated such that it satisfies the (*l-MAD*) property (see 2.1). The experiments for *Step*(*ii*) (Section 2.3.2) also involve the periods, and for our study we have assumed this to be set between 800 and 2000 for each task.

Figure 2.6 shows the running times involved in computing the *dbf-table* of a single task graph. Once the deadline associated with a vertex of this task graph was relaxed, we have (i) recomputed the entire *dbf-table* using Algorithm 1, and (ii) updated the *dbf-table* using Algorithm 2. Figures 2.6(a) and 2.6(b) show the running times incurred for task graphs with their number of vertices ranging from

50 to 200, which were generated by setting $E = 200$ and $E = 600$ respectively. The task graphs formed by joining together two copies of our original task graphs had 100 to 400 vertices (as explained in Section 2.1.3), and the computation of the *dbf-table* used these graphs.

For each randomly generated task graph, we randomly selected a vertex of this graph and relaxed its deadline by a certain amount. The *dbf-table* associated with this task was then (i) entirely *recomputed*, and (ii) *updated* using our proposed scheme. For each task graph, this process was repeated for five randomly selected vertices. The results in Figures 2.6(a) and 2.6(b) report the maximum *dbf-table* update time incurred among these five vertices, along with the time required to recompute the entire *dbf-table*. These results illustrate the savings achieved by our proposed scheme. With $E = 600$, we obtain a speedup of more than 20×, which translates into the schedulability analysis running in approximately 2 minutes instead of 40 minutes. In an interactive design environment, the former waiting time is clearly more tolerable than the latter. It should also be noted that with larger values of E, even higher speedups will be obtained. Figures 2.7(a) and 2.7(b) show similar results for the case where the deadline of a vertex was constrained.

We also conducted another set of experiments with relatively smaller task graphs (containing 50 vertices), while varying the value of E from 1000 to 10000. Here, it may be noted that the execution requirement associated with any vertex of a graph is expressed in terms of *time units*. Such time units depend on the application at hand and might denote milliseconds, microseconds, or even the number of clock cycles of the processor on which the task graphs are required to execute. Hence, experiments with large values of E are completely realistic. Our motivation behind experimenting with small task graphs is that most realistic applications are likely to be represented by task graphs containing relatively few vertices. The steps involved in this set of experiments are exactly similar to those of the earlier experiments.



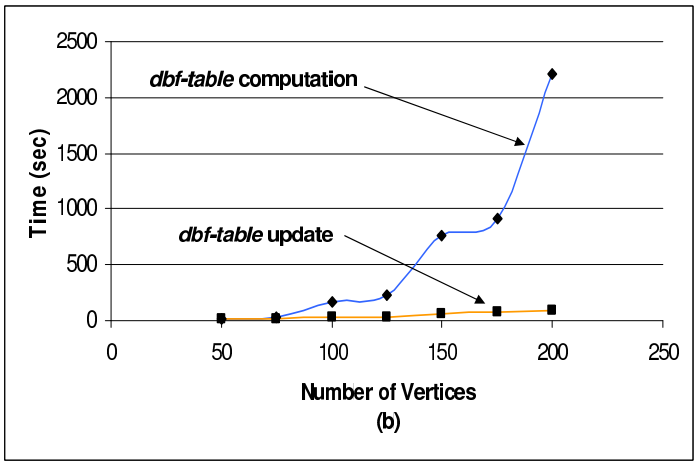


Figure 2.7: Running times for updating the *dbf-table* when the deadline of a vertex was constrained (a) $E = 200$ and (b) $E = 600$.

Figure 2.8(a) shows how the *dbf-table* update time and computation time changes with increasing E (the maximum execution time associated with a vertex), when the deadline associated with a randomly chosen vertex of a task graph is relaxed. Figure 2.8(b) shows the corresponding results when the deadline associated with a vertex is constrained. Note that in both the cases we obtain speedups of around $5\times$, which are significant if a design tool is to be used in an interactive fashion.

All the CPU times reported above were measured on a Linux machine with Fedora Core 3, running on a 3.0 GHz CPU with a 2 GB RAM.

It may be noted that all our implementations were done in C++, did not make

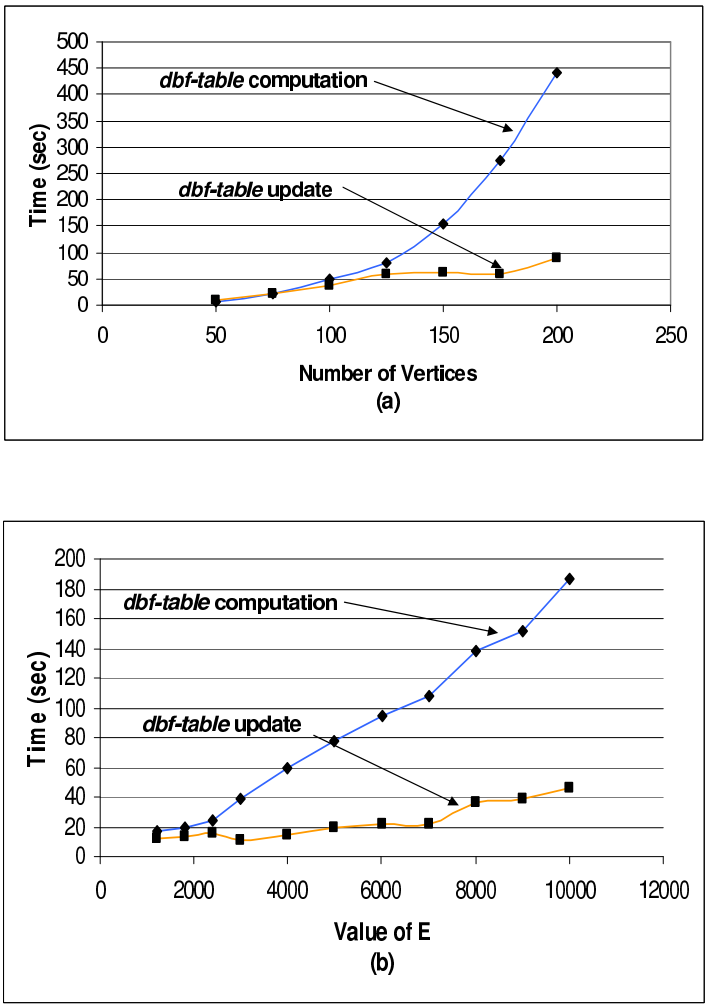


Figure 2.8: Running times for updating the *dbf-table* for a task graph with 50 vertices, as the maximum execution requirement associated with a vertex (E) is increased. (a) Deadline of a randomly chosen vertex is relaxed, and (b) Deadline of a randomly chosen vertex is constrained.

use of any graphical interfaces for specifying the task graphs, and the code was specifically optimized for running the schedulability analysis. In practice, a design tool supporting schedulability analysis would be more involved. More specifically, the task graphs might be integrated with other application-specific data structures that are not be optimized for the schedulability analysis algorithm. In such cases, the speedups obtained by our interactive schedulability analysis might be considerably higher compared to the results reported here. This is because it involves fewer traversals through these task graphs in subsequent invocations of the

Task Sets	Task Graphs	
Set 1 #vertices/task graph = 10 max. exec. req. of a vertex (E) = 200 $t_{\max} = 54.6 \times 10^3$	T_1 T_2 T_3	1.647×10^3 1.799×10^3 4.474×10^3
Set 2 #vertices/task graph = 20 max. exec. req. of a vertex (E) = 200 $t_{\max} = 368.353 \times 10^3$	T_1 T_2 T_3	3.759×10^3 2.662×10^3 84.634×10^3
Set 3 #vertices/task graph = 30 max. exec. req. of a vertex (E) = 200 $t_{\max} = 823.834 \times 10^3$	T_1 T_2 T_3	8.657×10^3 4.975×10^3 104.517×10^3
Set 4 #vertices/task graph = 40 max. exec. req. of a vertex (E) = 200 $t_{\max} = 806.714 \times 10^3$	T_1 T_2 T_3	7.017×10^3 13.906×10^3 55.96×10^3
Set 5 #vertices/task graph = 50 max. exec. req. of a vertex (E) = 200 $t_{\max} = 1431 \times 10^3$	T_1 T_2 T_3	6.861×10^3 13.005×10^3 8.945×10^3

Table 2.3: Number of checks required in *Step (ii)* of the proposed interactive schedulability analysis, versus $t_{\max}$, which is equal to the number of checks that a regular schedulability analysis algorithm would perform.

analysis, thereby saving the overheads associated with these traversals due to the potentially complicated data structures. This observation stems from our attempt to integrate this schedulability analysis algorithm inside a tool-suite [30] where the task graphs were specified using a graphical user interface and were embedded inside other data structures that were a part of this tool-suite. In this implementation we observed 20× speedups using our algorithm for task graphs with less than 40 vertices. However, with the optimized C++ implementation of our algorithm, such speedups could only be seen for task graphs with around 200 vertices.

2.3.2 Experiments with Step (ii)

In Section 2.2.2, we had outlined an efficient method to perform *Step*(*ii*) of our proposed interactive schedulability analysis. This section illustrates the savings obtained by using that method. For our experiments, we generated five task sets with each set consisting of three task graphs. The number of vertices in these task graphs ranged over 10 to 50, with the first task set consisting of task graphs with 10 vertices, the second task set consisting of task graphs with 20 vertices, and so on. The value of E for all the task graphs was set to 200.

We randomly chose a vertex of a task graph and constrained its deadline. We then computed the number of checks that were needed to perform *Step*(*ii*), following the description in Section 2.2.2. The results obtained are shown in Table 2.3. This experiment was repeated for each task graph in the five task sets. The table shows the results for five task sets, with each set containing three task graphs. The numbers in the rightmost column are the number of checks in *Step (ii)* when the deadline associated with a randomly chosen vertex of the task graph in the same row is constrained. Note from Table 2.3 that there are cases where the number of checks of the schedulability condition reduce to almost 0.5% of the total number of checks that would be performed by a regular schedulability analysis algorithm. This again illustrates the potential savings that our interactive schedulability analysis can achieve.

2.4 Providing Feedback to the System Designer

In what we have seen so far, if a task set fails the schedulability test for a certain $\hat{t}$, a system designer is allowed to randomly select some of the vertices of certain task graphs, relax their deadlines and rerun the analysis. However, relaxing the deadline

of some randomly selected vertex might not make the task set schedulable. Hence, it would be meaningful to provide some feedback to the designer about potential vertices, whose deadlines might be changed to make the task set schedulable. Other types of feedback like changing the periods of certain task graphs or increasing the intertriggering separation times associated with some of the edges of a task graph might also be meaningful. Such feedback can be provided using the scheme we have presented in this chapter.

Towards this, the algorithm used for computing the *dbf-table* (i.e. Algorithm 1) needs to be changed, so that some additional data structures are computed. These data structures, $Q_{i,e}$ and $Q^e_{i,e}$, are computed by Algorithm 4.

Recall that each cell in our *dbf-table* contains three different values: $t^i_{i,e}$, $t_{i,e}$, and $flag_{i,e}$. In addition to these, we now store two lists $Q_{i,e}$ and $Q^e_{i,e}$ in each cell. $Q_{i,e}$ records the subset of vertices from the set $\{v_1, \ldots, v_i\}$, whose triggering demands an execution time of e, within any time interval of length $t_{i,e}$. Similarly, $Q^e_{i,e}$ lists the subset of vertices from $\{v_1, \ldots, v_i\}$, which ends with the vertex v_i and has an execution requirement of e within any time interval of length $t^i_{i,e}$. Algorithm 4 not only returns $T'.dbf(t)$, but also the list of vertices $Q(t)$ whose triggering results in the execution demand of $T'dbf(t)$.

We now explain how $Q(t)$ can be used to provide useful feedback to a system designer. Recall from Section 2.1.2 that we create a list of $T.dbf(t)$ for all "small" values of t. To this list, we now add the data structure $Q(t)$ containing the vertices that contribute to $T.dbf(t)$. During the schedulability test in step (ii), suppose the test fails at $\hat{t}$. If $\hat{t}$ is "small", then we can find the desired list of vertices $Q(\hat{t})$ directly from the table. If $\hat{t}$ is "large", we check whether $T.dbf(\hat{t})$ is equal to $\lfloor \hat{t}/P(T) \rfloor E(T) + T.dbf(\hat{t} \bmod P(T))$ or $(\lfloor \hat{t}/P(T) \rfloor - 1)E(T) + T.dbf(P(T) + \hat{t} \bmod P(T))$(see Eqn. 2.1) ($T.dbf(\hat{t})$ has to be equal to either of these two values).

Algorithm 4 Computing of $T'.dbf(t)$ with data structures for providing feedback

Require: Task graph T', and a real number $t \geq 0$

1: **for** $e \leftarrow 1$ to nE **do**
2: **if** $e(v_1) = e$ **then**
3: $t_{1,e} \leftarrow d(v_1)$
4: $flag_{1,e} \leftarrow$ `SELF`
5: $enqeue(Q_{1,e}, v_1)$
6: **else**
7: $t_{1,e} \leftarrow \infty$
8: $flag_{1,e} \leftarrow$ `PREVIOUS`
9: **end if**
10: $t^1_{1,e} \leftarrow t_{1,e}$
11: **end for**
12: **for** $i \leftarrow 1$ to $n-1$ **do**
13: **for** $e \leftarrow 1$ to nE **do**
14: Let there be directed edges from the vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ to v_{i+1}
15: $t^{i+1}_{i+1,e} \leftarrow \begin{cases} \min\{t^{i_j}_{i_j, e-e(v_{i+1})} - d(v_{i_j}) + p(v_{i_j}, v_{i+1}) + \\ d(v_{i+1}) \,|\, j = 1, \ldots, k\} \text{ if } e(v_{i+1}) < e, \\ d(v_{i+1}) \text{ if } e(v_{i+1}) = e, \text{ and } \infty \text{ otherwise} \end{cases}$
16: Let v_{min} be the vertex from amongst the set of vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$, which gave us the minimum value for the expression evaluated in line number 15
17: **if** $e(v_{i+1}) < e$ **then**
18: $Q^{i+1}_{i+1,e} \leftarrow Q_{min, e-e(v_{i+1})}$
19: $enqueue(Q^{i+1}_{i+1,e}, v_{i+1})$
20: **else if** $e(v_{i+1}) = e$ **then**
21: $enqueue(Q^{i+1}_{i+1,e}, v_{i+1})$
22: **end if**
23: $t_{i+1,e} \leftarrow \min\{t_{i,e}, t^{i+1}_{i+1,e}\}$
24: **if** $t_{i+1,e} = t^{i+1}_{i+1,e}$ **then**
25: $Q_{i+1,e} \leftarrow Q^{i+1}_{i+1,e}$
26: $flag_{i+1,e} \leftarrow$`SELF`
27: **else**
28: $Q_{i+1,e} \leftarrow Q_{i,e}$
29: $flag_{i+1,e} \leftarrow$`PREVIOUS`
30: **end if**
31: **end for**
32: **end for**
33: $T'.dbf(t) \leftarrow \max\{e \,|\, t_{n,e} \leq t\}$
34: $Q(t) \leftarrow Q_{n,e}$

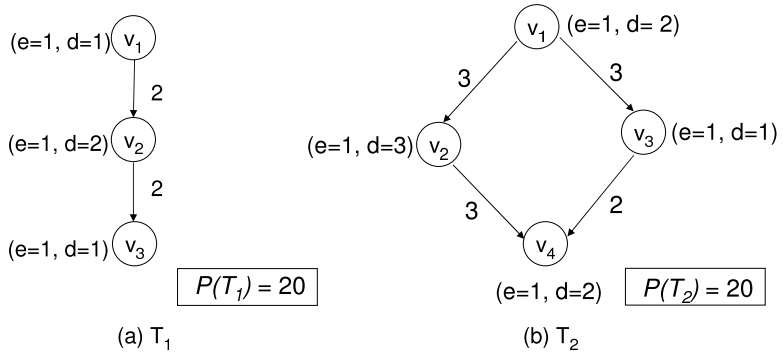


Figure 2.9: Task graphs (a) T_1 and (b) T_2 of our example task set τ.

If $T.dbf(\hat{t})$ is equal to the former expression then we select the vertices listed as $Q(\hat{t} \bmod P(T))$ from our table, otherwise we select the vertices corresponding to $Q(P(T) + \hat{t} \bmod P(T))$.

Hence, given any $\hat{t}$ for which the schedulability test failed, for any task graph T we can identify the legal sequence of vertices whose triggering contributed to $T.dbf(\hat{t})$. This sequence of vertices can now be used by the system designer to modify their associated deadlines or the intertriggering separations associated with their edges. In what follows, we refer to this sequence of vertices as the *critical path* of a task graph that is responsible for its (non-) schedulability.

2.4.1 Illustration of the Feedback Provided for an Example Task Set

Consider a task set τ, consisting of two task graphs T_1 and T_2, shown in Figure 2.9. Now assume that we would like to verify whether τ is schedulable, and in case it is not, we would like to change the deadlines of the *appropriate* vertices in order to make it schedulable. Here we illustrate how the scheme that we presented above can be used to effectively identify such appropriate vertices.

T_1' and T_2' (shown in Figure 2.10) were obtained by joining two copies of T_1 and T_2

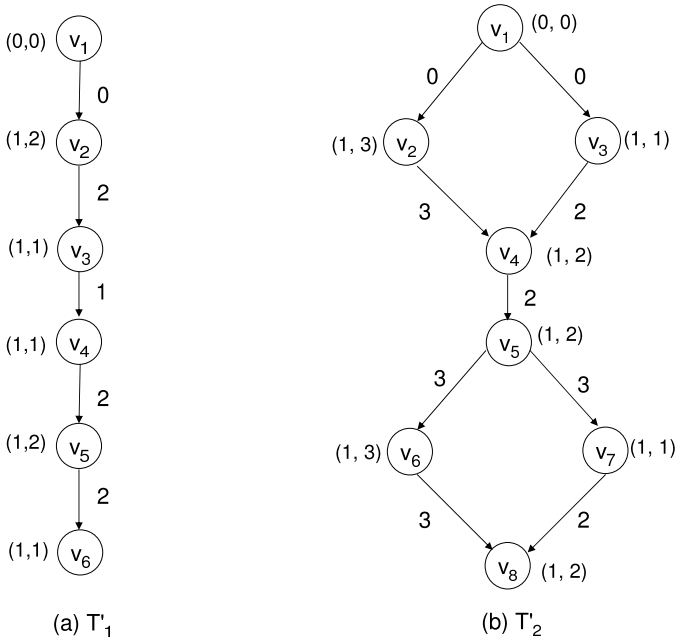


Figure 2.10: Task graphs (a) T_1' and (b) T_2' obtained from T_1 and T_2 respectively.

respectively, and will be used to compute $dbf(t)$ for "small" values of t.

Clearly, the schedulability analysis returns a negative answer for the task set τ. Further, Algorithm 4 provides the following feedback concerning the potential vertices whose deadlines may be relaxed:

- Critical Path for Task Graph T_1': v_6
- Critical Path for Task Graph T_2': v_7

Indeed from Figure 2.10, we see that v_6 of T_1' and v_7 of T_2', both demand 1 unit of execution time within a time interval of 1 unit. Thus, $\sum_{T \in \mathcal{T}} T.dbf(1) = 2$, implying that the condition $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$ is not satisfied at $t = 1$. Now, one might choose to relax the deadlines associated with v_3 and v_7 of T_2' from 1 to 2. It may be noted here that in practice, the task graphs T_1' and T_2' will not be visible to a designer and he or she will only work with the original graphs T_1 and T_2. Any changes made in these two task graphs can easily be translated to appropriate changes in T_1' and T_2'.

Now we re-run the analysis and find that the task set is still not schedulable, along with the following feedback:

- Critical Path for Task Graph T_1': v_3, v_4
- Critical Path for Task Graph T_2': v_8

To see that these paths are indeed critical to schedulability, note that from the path v_3, v_4 we get $T_1'.dbf(2) = 2$. Similarly, in task graph T_2', v_8 leads to $T_2'.dbf(2) = 1$. Thus, $\sum_{T \in \mathcal{T}} T.dbf(t) > t$, at $t = 2$. Again, to move towards a schedulable system, we now relax the deadline of v_4 of T_1' from 1 to 2, and rerun the analysis.

However, the task set is still not schedulable, and the feedback provided is as follows:

- Critical Path for Task Graph T_1': v_3, v_4, v_5, v_6
- Critical Path for Task Graph T_2': v_3, v_4, v_5

One can verify that the above sequence of paths lead to $\sum_{T \in \mathcal{T}} T.dbf(6) = 7$, thereby failing the schedulability test. This time we select v_5 of T_2', and relax its deadline from 2 to 3, thereby obtaining a schedulable system.

In the above example, we have seen the benefits of the feedback mechanism on a small task set. In larger systems where many more task graphs and more vertices would be involved, this mechanism would certainly be of immense benefit.

2.5 Summary

In this chapter we presented a scheme for efficient schedulability analysis of recurring real-time task sets, where the schedulability analysis is repeatedly invoked

with small modifications in the task set. Since this scheme is used in an interactive fashion, we referred to it as *interactive schedulability analysis*.

As discussed in Chapter 1, many system-level design tools in the electronic design automation domain are being used by the designers in a interactive fashion during design space exploration. Since, our method exploits this repeated invocation of the algorithm to achieve speed-ups as well to provide feedback, it has the potential to be applied to all such problems.

Chapter 3

Efficiently Computing Performance Tradeoffs using Multicriteria Schedulability Analysis

---

As mentioned in Chapter 1, performance analysis of real-time embedded systems occupy a major chunk of their overall design time in iterative design space exploration. In this context, we also discussed some of the reasons that lead to the tedious design sessions for schedulability analysis and mulitcriteria hardware/software co-design. In Chapter 2, we introduced our *interactive schedulability analysis* framework to ease the tedious design cycles, and in this chapter, we shall be concerned with computing tradeoffs in a standard mulitcriteria hardware/software co-design problem.

Real-time embedded systems are increasingly becoming heterogeneous and consist of a mix of fully- and partially-programmable processors, fixed-function hardware

accelerators and different kinds of buses and memory modules. Applications to be implemented on such systems are partitioned and mapped onto these different processors and hardware components. This results in a large number of implementation possibilities with different performance tradeoffs. As a result, a designer is no longer interested in *one* implementation that meets the real-time constraints associated with a given application (i.e. is schedulable), but would rather like to identify *all* schedulable implementations that expose the different possible performance tradeoffs. In this chapter, we shall introduce an efficient and formal methodology towards this.

As a simple example where identifying multiple performance trade-offs is crucial, consider two applications (or tasks) T_1 and T_2 which are required to run concurrently and have predefined deadline constraints. Both T_1 and T_2 can be partially implemented in hardware, with their remaining parts implemented as software running on the same programmable processor P. Such a scheme is in line with two possible realistic realizations. First, there is the fine-grained approach of customizable processors where the system designer may choose to implement frequently occurring computation patterns in hardware. For example, Xtensa [34] from Tensilica is a configurable processor core. The XPRES compiler provided by Tensilica generates the custom instructions from the C code corresponding to a task, and the designer may choose to map them directly to the hardware. Secondly, such a scheme is also in line with CPU/FPGA architectures (e.g. Virtex-II PRO from Xilinx), which consist of one or more programmable processors embedded within the FPGA's logic fabric. Various techniques have been proposed to partition a given application for such hardware-software architectures [49].

In such scenarios, the portions (or even fractions) of two tasks, T_1 and T_2 to be implemented in hardware constitute the different implementation options. The two objectives to be optimized are the total hardware cost and the minimum clock fre-

quency of P (which, for example, might influence its power consumption). Clearly, there can be different implementation options which satisfy T_1 and T_2's deadline constraints. If larger fractions of T_1 and T_2 are implemented in hardware, then the hardware cost increases and the required clock frequency of P decreases, and vice versa. For any schedulable implementation, if (c, f) denotes the corresponding hardware *cost* and clock *frequency*, then a designer will be interested in identifying all possible tuples $(c_1, f_1), \ldots, (c_n, f_n)$ which capture the different performance tradeoffs. In the multicriteria optimization parlance, the set $\{(c_1, f_1), \ldots, (c_n, f_n)\}$ is referred to as the *Pareto curve* and each point (c_i, f_i) in this set is called a *Pareto-optimal solution* [26] (see Figure 3.1). Each (c_i, f_i) in this set has the property that there does not exist any schedulable implementation of T_1 and T_2 with a performance vector (c, f) such that $c \leq c_i$ and $f \leq f_i$, with at least one of the inequalities being strict. Further, let $\mathcal{S}$ be the set of performance vectors corresponding to all schedulable implementations. Let $\mathcal{P}$ be the set of performance vectors $\{(c_1, f_1), \ldots, (c_n, f_n)\}$ corresponding to all the Pareto-optimal solutions. Then for any $(c, f) \in \mathcal{S} - \mathcal{P}$ there exists a $(c_i, f_i) \in \mathcal{P}$ such that $c_i \leq c$ and $f_i \leq f$, with at least one of these inequalities being strict (i.e. the set $\mathcal{P}$ contains *all* performance tradeoffs). The vectors $(c, f) \in \mathcal{S} - \mathcal{P}$ are referred to as *dominated solutions*, since they are "dominated" by one or more Pareto-optimal solutions as shown in Figure 3.1.

In this chapter we present a polynomial-time approximation algorithm for computing the Pareto curve $\mathcal{P} = \{(c_1, f_1), \ldots, (c_n, f_n)\}$. This result is interesting because even the single-criteria version of the problem in very simple settings turns out to be intractable (NP-hard). Given a set of tasks and a processor P running at a predefined clock frequency, the single-criteria version of this problem is to come up with a schedulable (on P) implementation of these tasks with the minimum hardware cost. In other words, the processor has a predefined clock frequency which is

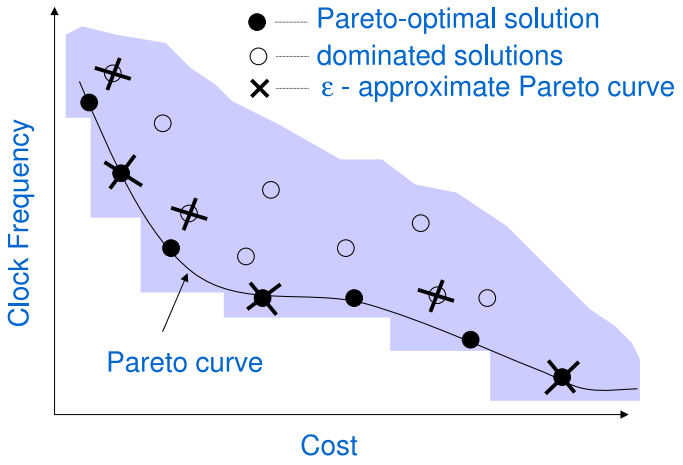


Figure 3.1: Pareto-optimal solutions.

provided as an input. Note that the well-studied schedulability analysis problem [11, 17] — where the goal is to decide whether the task set is entirely schedulable on P — is a special case of the single-criteria version of our problem.

The second reason which makes our work interesting is that there can be an exponentially large number of performance vectors (c_i, f_i) in the Pareto curve $\mathcal{P}$, which makes it impossible to compute this entire set in polynomial time. Hence, our polynomial-time approximation algorithm by default also implies approximating the (potentially exponential size) set $\mathcal{P}$ with only a polynomial number of points. In a typical design or performance debugging scenario, a system designer inspects all the tradeoffs in the set $\mathcal{P}$ and then selects one, or at most a few implementations. Hence, from a practical perspective, it is more meaningful if the designer is presented with a reasonably few well-distinguishable tradeoffs in the set $\mathcal{P}$, rather than an exponentially large number of solutions, many of which are very similar to each other. Our approximation algorithm is therefore not only attractive in terms of time-complexity, but also returns more meaningful solutions, as we show later in this chapter.

Overview of the proposed scheme

Our proposed scheme takes as an input an error parameter ϵ and returns an ϵ-*approximate Pareto curve* which we denote as ϵ-Pareto curve (or $\mathcal{P}_\epsilon$) in the rest of this thesis. Given a Pareto curve $\mathcal{P} = \{(c_1, f_1), \ldots, (c_n, f_n)\}$, an ϵ-approximate Pareto curve is defined as *any* set $\mathcal{P}_\epsilon = \{(c'_1, f'_1), \ldots, (c'_m, f'_m)\}$ such that for any $(c_i, f_i) \in \mathcal{P}$, there exists a $(c'_j, f'_j) \in \mathcal{P}_\epsilon$ for which $c'_j \leq (1+\epsilon)c_i$ and $f'_j \leq (1+\epsilon)f_i$. In other words, corresponding to any point on the Pareto curve $\mathcal{P}$, there exists a point on $\mathcal{P}_\epsilon$, each of whose coordinates are at most ϵ distance away from the corresponding coordinates of the point on $\mathcal{P}$. Hence, each "tradeoff" in $\mathcal{P}$ has an "ϵ-approximation" in $\mathcal{P}_\epsilon$, where the semantics of ϵ-approximation are as defined above. In Figure 3.1, each point on the Pareto curve (denoted by $\bullet$) is approximated by some point (denoted by $\times$) which may be a Pareto-optimal solution or a dominated solution. The set of $\times$ points depend on the value of ϵ, and constitute the set $\mathcal{P}_\epsilon$. Since ϵ is an input provided by the system designer, the error between the approximate and the optimal Pareto curves can be made as small as desired. The running time of our approximation algorithm, as we show later, is polynomial in the size of the problem instance and polynomial in $\frac{1}{\epsilon}$, but exponential in the number of objectives/criteria. However, since the number of objectives is typically small for most real-life problems, this should not pose any problem. Finally, as one might expect, the running time of the algorithm increases as the error parameter ϵ is made smaller.

Our algorithm is made up of the following two parts.

(i) The first part is a polynomial-time approximation algorithm for solving the single-criteria version of the problem. Recall that here we are given a processor P with a predefined clock frequency (or alternatively, a target processor utilization). The goal is to compute a partition of each task such that the

portions mapped onto P are schedulable and the total hardware cost is minimized. As we describe later, we assume that each task comes with a specified number of hardware implementation possibilities, i.e. certain subtasks or portions which might be implemented in hardware or in software, and the remaining can only be implemented in software. We believe that this is more realistic than assuming that a task can be arbitrarily partitioned into hardware and software. The approximation algorithm for the single-criteria version takes as an input an error parameter ϵ. It returns a hardware cost which is guaranteed to be no more than $(1 + \epsilon)$ times the minimum cost incurred to schedule the tasks on P with the predefined clock frequency or processor utilization. Alternatively, it says that there does not exist any schedulable implementation of the task set under the possible hardware-implementation options.

(ii) The second part of our algorithm involves imposing a k-dimensional grid on the objective space, where k is the number of objectives being considered. In the case of our bicriteria example (where $k = 2$), this boils down to a rectangular grid. We then (approximately) solve a single-criteria version of our problem for each grid point by using our approximation algorithm outlined in part (i) and retain only the Pareto-optimal solutions (or rather the "Pareto-optimal grid points"). The crux of this step is in the choice of the grid dimensions, which are also functions of the error parameter ϵ that was used in part (i). By appropriately choosing the grid dimensions, we can guarantee that the approximate Pareto curve is within ϵ distance from the optimal Pareto curve. Further, the number of calls to the approximation algorithm in part (i) is restricted to a polynomial in the problem size and in $\frac{1}{\epsilon}$, but exponential in the number of objectives k.

In summary, both parts (i) and (ii) incur an error in the computation of the Pareto

curve. However, the cumulative error is bounded such that the resulting points in the objective space still cover the entire Pareto curve and approximate it with a maximum error of ϵ in all the objectives.

Related Work

There exists a large body of work on multiobjective optimization [26] and also on multicriteria scheduling and decision making [78]. However, a significant portion of these approaches address the problems from an engineering perspective and relies on heuristics and randomized search techniques such as evolutionary algorithms (e.g. see [23]). Our work in this thesis differs from these approaches by taking a classical approximation algorithms standpoint, where the goal is to provide formal guarantees on the quality of the results obtained.

Further, we are also not aware of any work on multicriteria *schedulability analysis* of the form that we present in this thesis. Flexible scheduling with multiple concerns is considered to be an important problem in the real-time systems domain (e.g. see [13]). However, to the best of our knowledge, no formal algorithmic solution to this problem is known so far. Our work is also tangentially related to a number of recent papers on performance debugging of real-time and embedded systems from a timing/schedulability analysis perspective. For example, [15, 66, 68, 69, 83] address the problem of *sensitivity analysis* of real-time systems where the goal is to compute permissible changes in certain system parameters that do not result in the required timing/schedulability constraints to be violated. Such changes, especially in a multidimensional setting [68], might be viewed as possible schedulable implementations which are associated with different performance tradeoffs. Similarly, [14] addressed the problem of computing the "schedulable region" or *space* of task periods and worst-case execution times that lead to schedulable systems. The

main difference between this line of work and our results is that both sensitivity analysis and schedulable region computation do not explicitly consider schedulable implementation *tradeoffs*, which is our main focus.

The algorithmic techniques presented in this chapter have been motivated by [46] and [61]. More specifically, [46] used a partitioning technique to divide a multidimensional objective space into hyper-rectangles – as we do in part (ii) of our scheme – but used it for improving the search quality of a randomized search algorithm. The result that *any* Pareto curve can be ϵ-approximated by a polynomial-size approximate Pareto curve was first proved in [61]. However, for many problems, efficiently (i.e. in polynomial time) computing such approximate Pareto curves might not be possible. Our work in this thesis shows that for the multicriteria schedulability analysis problem, such approximate Pareto curves can also be *computed* in polynomial time.

Organization of this Chapter

The rest of the chapter is organized as follows. In the next section we introduce our task model and some necessary notations. In Section 3.2 we then formally define the single-criteria version of the problem, prove that it is NP-hard and derive a polynomial-time approximation scheme for solving it. This is followed by our solution to the multicriteria problem. Some of the experimental results we obtained are described in Section 3.4. Finally, we conclude in Section 3.5.

For ease of exposition, all the algorithms presented in this chapter are for a bicriteria schedulability analysis problem; more specifically, the one we described as an example at the beginning of this chapter. However, all our results trivially extend to higher dimensional settings. Similarly, we also considered a simple sporadic

task model [11, 52]. Again, it is possible to extend our algorithms to more general task models such as multiframe [55], generalized multiframe [10], and recurring real-time [9] models.

3.1 Task Model

In this work, we use the sporadic task model in a preemptive uniprocessor environment to illustrate our approximation scheme. Thus, we are interested in the schedulability analysis of a task set $\tau = \{T_1, T_2, \ldots, T_m\}$ consisting of m hard real-time tasks. Any task T_i can get triggered independently of other tasks in τ. Each task T_i generates a sequence of jobs; each job is characterized by the following parameters:

- *Release Time*: the release time of two successive jobs of the task T_i is separated by a minimum time interval of P_i time units.
- *Deadline*: each job generated by T_i must complete by D_i time units since its release time.
- *Workload*: the worst case execution requirement of any job generated by T_i is denoted by E_i.

Throughout this thesis, we assume the underlying scheduling policy to be the earliest deadline first (EDF). Again, our algorithm can be suitably modified to handle other scheduling policies as well. Assuming that for all tasks T_i, $D_i \geq P_i$, the schedulability of the task set τ can be given by the following condition.

Theorem 3.1.1 *A set of sporadic tasks τ is schedulable under EDF if and only if*

Tasks in the Task Set	Workload	Cost
# choices for task $T_1 = 3$ $E_1 = 12$, $P_1 = 40$	10	15
	8	45
	4	90
# choices for task $T_2 = 2$ $E_2 = 6$, $P_2 = 16$	5	24
	2	42
# choices for task $T_3 = 3$ $E_2 = 6$, $P_3 = 25$	8	11
	6	26
	5	82

Table 3.1: Implementation choices for three different tasks in a task set. Each row of this table shows the new execution requirement (on a programmable processor) because of a part of the task being implemented in hardware, along with the incurred hardware cost.

$$(U = \sum_{i=1}^{m} \frac{E_i}{P_i}) \leq 1$$

where U is the processor utilization due to τ [11, 52].

3.2 The Single-Criteria Problem

In this section, we formally state the single-criteria version of the problem along with an illustrative example. We then show that this problem is intractable even for the simple sporadic task model described in Section 3.1. Finally, we derive a fully polynomial-time approximation scheme (FPTAS) [40] for solving it.

Recall that we are given a processor P with a predefined clock frequency, and a specified number of subtasks of each task T_i which can be implemented in hardware. Our goal is to identify the implementation choices that lead to the minimum hardware cost, provided the portions of the tasks mapped onto P are schedulable.

If the task set is entirely schedulable on the processor P (i.e. $U \leq 1$), then the problem is trivial and we need not incur any hardware costs. However, if the

entire task set is not schedulable on P, then certain portions of some of the tasks in the set will have to be implemented in hardware to reduce the load on P. The problem is then that of identifying which portions or subtasks of each task should be mapped onto hardware such that the minimum hardware cost is incurred.

For each task T_i, let there be n_i hardware implementation choices. Each of these n_i choices is associated with a certain hardware cost. Choosing the jth implementation choice for the task T_i lowers its execution requirement on P from E_i to $e_{i,j}$. Equivalently, the amount by which the execution requirement of T_i gets lowered on P is $\delta_{i,j} = E_i - e_{i,j}$. Hence, for each task T_i we have a set of choices $S_i = \{(\delta_{i,1}, c_{i,1}), \ldots, (\delta_{i,n_1}, c_{i,n_1})\}$, where $c_{i,j}$ is the hardware cost associated with the jth implementation choice. The goal is to identify one choice for each task, which would lower the processor utilization to less than or equal to one, and minimize the total hardware cost. In what follows, we shall refer to this as the *minimum cost schedulability analysis* problem.

We now illustrate this problem with the help of an example. A task set τ has three tasks $\{T_1, T_2, T_3\}$ with $\{E_1 = 12, P_1 = 40\}$, $\{E_2 = 6, P_2 = 16\}$, and $\{E_3 = 11, P_3 = 25\}$. Clearly the processor utilization $U > 1$ and hence this task set is not schedulable, without some of the subtasks being mapped onto hardware. The different possible hardware implementation choices for each task in this set is shown in Table 3.1. Each row of this table shows the new execution requirement of a task on P after a part of this task is implemented in hardware, and the associated hardware cost. Note that as the execution requirement or workload of a task decreases, its associated hardware cost increases.

Following the notation we introduced above, for T_1 we have $e_{1,1} = 10$, $e_{1,2} = 8$ and $e_{1,3} = 4$. The corresponding hardware costs are $c_{1,1} = 15$, $c_{1,2} = 45$ and $c_{1,3} = 90$. Hence, the implementation choices for T_1 are given by the set $S_1 =$

$\{(2, 15), (4, 45), (8, 90)\}$. The choices for T_2 and T_3 can be similarly computed from this table. Note that while T_1 and T_3 have three choices each, T_2 has only two choices. Thus, $n_1 = n_3 = 3$ and $n_2 = 2$. The goal is to select one choice from each set S_1, S_2 and S_3, such that we obtain a minimum-cost schedulable system.

3.2.1 NP-hardness

We show that the minimum cost schedulability analysis problem is NP-hard using a polynomial-time transformation from the 0-1 knapsack problem [40].

Theorem 3.2.1 *The minimum cost schedulability analysis problem is NP-hard.*

Proof: The decision version of the minimum cost schedulability analysis problem asks whether there is a set of choices of the execution requirements such that the condition $U \leq 1$ is satisfied, and the total cost is $\leq C$.

The knapsack problem specifies m items with integral weights w_i and profits p_i, $i = 1, 2, \ldots, m$, an integral weight constraint W and a profit goal G. Let the m binary variables $x_i \in \{0, 1\}$ correspond to the selection of the ith item. The knapsack decision problem asks if there exists a subset of items, the sum of whose profits $\sum_{i=1}^{m} p_i x_i \geq G$ and the sum their weights is $\sum_{i=1}^{m} w_i x_i \leq W$.

We transform the knapsack problem into a special instance of our problem which is obtained by setting $n_i = 1$, for $i = 1, 2, \ldots, m$. Towards this, let $\delta_{i,1} = p_i$ and $c_{i,1} = w_i$. Hence, corresponding to each item i in the knapsack problem with weight w_i and profit p_i, there is a task T_i with $\delta_{i,1} = p_i$ and cost $c_{i,1} = w_i$. For this problem instance, let all the m tasks in the task set τ have the same deadline D. Further, let all the periods be equal to their deadlines, i.e. $P_i = D$ for all $\{i = 1, 2, \ldots, m\}$.

The values D and E_i are chosen such that the $\sum E_i - D = G$. Our claim is that the minimum cost schedulability analysis decision problem returns a *Yes* answer if and only if the knapsack problem returns a *Yes*. To verify this, let us first consider the *if* direction. This immediately implies that $\sum_{i=1}^{m} c_i x_i \leq C$ for the problem instance we constructed. For our special instance, where $n_i = 1$, the binary variable $x_i \in \{0, 1\}$ corresponds to the selection of the $\{i, 1\}$th choice. Again, a solution to the knapsack problem also implies that:

$$\begin{aligned}
&\sum_{i=1}^{m} p_i x_i \geq G \\
\Rightarrow &\sum \delta_{i,1} x_i \geq \sum E_i - D \\
\Rightarrow &\sum E_i - \sum \delta_{i,1} x_i \leq D \\
\Rightarrow &(E_1 - \delta_{1,1} x_1) + (E_2 - \delta_{2,1} x_2) + \cdots + (E_m - \delta_{m,1} x_m) \leq D \\
\Rightarrow &U \leq 1
\end{aligned}$$

The claim can be similarly verified in the other direction. Thus, the special case of the minimum cost schedulability analysis problem is NP-hard and the theorem follows. □

3.2.2 Approximating the Minimum Cost Schedulable Solution

In this section we first present a dynamic programming algorithm (Algorithm 5) to compute the minimum cost that must be incurred to obtain a schedulable task set. This algorithm runs in pseudo-polynomial time. We then use this algorithm to derive a *fully polynomial-time approximation scheme* (FPTAS) for the same problem.

Let $U_{i,j}$ be the minimum utilization that might be achieved by considering only a

subset of tasks from $\{1, 2, \ldots, i\}$ when the cost is exactly j. If no such subset exists we set $U_{i,j} = \infty$. Let the maximum cost be C i.e. $C = max_{(i=1,2,\ldots,n;j=1,2,\ldots,n_i)} c_{i,j}$. Clearly, mC is an upper bound on the total cost that might be incurred. All other notations used are as introduced in Section 3.2.

Lines 1 to 5 of Algorithm 5 initialize $U_{0,0}$ to $\sum_{i=1}^{m} E_i/P_i$, and $U_{0,j}$ to ∞ for $j = \{1, 2, \ldots, mC\}$. The values $U_{i,j}$ for $i = 1$ to $i = m$ are computed using the iterative procedure in lines 7 to 26. For an iteration where $(i = i')$ and $(j = j')$, we say that $U_{i',j'+c_{i,k}}$ is *updated* using the recursive computation in lines 16 to 23 where $U_{i',j'+c_{i,k}}$ is assigned the value $\{U_{i'-1,j} - \delta_{i',k}/P_{i'}\}$. Thus, $U_{i',j'+c_{i,k}} \neq U_{i'-1,j'+c_{i,k}}$, i.e. $U_{i',j'+c_{i,k}}$ does not carry the value from the previous iteration but is updated with a new value. When such an updated entry is accessed after a few iterations (i.e. when $j = j' + c_{i,k}$), this updated value should not get re-initialized to its previous value (line 10). This is taken care of in lines 9 to 12 with the help of the *if-else* conditional statements on the variable tag_j . Towards this, the value tag_j is set to 0 for updated entries in lines 18 to 19. It can be easily verified that the running time of Algorithm 5 is $O(nmC)$, where $n = \sum_{i=1}^{m} n_i$, and its space complexity is $O(m^2C)$.

Next, we present an FPTAS for the minimum cost schedulability analysis problem. Towards this, we divide the cost space between 1 and mC into $O(n \log_{1+\epsilon} mC)$ intervals as $(1, (1+\epsilon)^{1/n}], ((1+\epsilon)^{1/n}, (1+\epsilon)^{2/n}], \ldots$.

Our FPTAS is based on Algorithm 5. But instead of running it for all possible cost values, from 0 to mC, we only consider the value 0 and the upper end points of the partitioned intervals we described above. Let $\widetilde{U}_{i,\widetilde{j}}$, represent the utilization value with the cost *at most* $\widetilde{j}$, where $\widetilde{j}$ always takes the value 0 or the value of one of the upper endpoints of the above mentioned intervals. During the iteration for the current entry $\widetilde{U}_{i,\widetilde{j}}$, the following procedure is executed for the recursive

Algorithm 5 Minimum-cost schedulability analysis

Require: The task set τ, and a set S_i for each task T_i.

1: $U_{0,0} \leftarrow \sum_{i=1}^{m} E_i/P_i$
2: $tag_j \leftarrow 1$
3: **for** $j \leftarrow 1$ to mC **do**
4: $\quad U_{0,j} \leftarrow \infty$
5: $\quad tag_j \leftarrow 1$
6: **end for**
7: **for** $i \leftarrow 1$ to m **do**
8: $\quad$ **for** $j \leftarrow 0$ to mC **do**
9: $\quad\quad$ **if** $tag_j = 1$ **then**
10: $\quad\quad\quad U_{i,j} = U_{i-1,j}$
11: $\quad\quad$ **else**
12: $\quad\quad\quad tag_j = 1$
13: $\quad\quad$ **end if**
14: $\quad\quad$ For each pair $(\delta_{i,k}, c_{i,k})$ that belongs to the set S_i
15: $\quad\quad$ **if** $(j + c_{i,k}) \leq mC$ **then**
16: $\quad\quad\quad$ **if** $tag_{j+c_{i,k}} = 1$ **then**
17: $\quad\quad\quad\quad U_{i,j+c_{i,k}} \leftarrow \min\{U_{i-1,j+c_{i,k}}, U_{i-1,j} - \delta_{i,k}/P_i\}$
18: $\quad\quad\quad\quad$ **if** $U_{i,j+c_{i,k}} = U_{i-1,j} - \delta_{i,k}/P_i$ **then**
19: $\quad\quad\quad\quad\quad tag_{j+c_{i,k}} = 0$
20: $\quad\quad\quad\quad$ **end if**
21: $\quad\quad\quad$ **else**
22: $\quad\quad\quad\quad U_{i,j+c_{i,k}} \leftarrow \min\{U_{i,j+c_{i,k}}, U_{i-1,j} - \delta_{i,k}/P_i\}$
23: $\quad\quad\quad$ **end if**
24: $\quad\quad$ **end if**
25: $\quad$ **end for**
26: **end for**
27: $MinCost \leftarrow \min\{j \mid U_{n,j} \leq 1\}$

equations in lines 17 and 22 of Algorithm 5. The cost of $[\widetilde{j} + c_{i,j}]$ is rounded up to the next upper endpoint $\widetilde{u}$. The value in this entry i.e. $\widetilde{U}_{i-1,\widetilde{u}}$ is compared with $\widetilde{U}_{i,\widetilde{j}} - \delta_{i,j}/P_i$, and the minimum of the two is stored in $\widetilde{U}_{i,\widetilde{u}}$. This explains how to update the main recursive equation in lines 15 and 17. The value for tag_j can be updated in a similar way (lines 19 and 20). The running time of the resulting algorithm is $O(n^2 \log_{1+\epsilon} mC)$ (which is bounded by a polynomial in the problem size and in $\frac{1}{\epsilon}$).

Theorem 3.2.2 *Our approximation algorithm for the minimum cost schedulability analysis problem is a $(1+\epsilon)$-approximation scheme.*

Proof: The algorithm can be proved to be a $(1+\epsilon)$-approximation scheme if we can show that $\widetilde{j} \leq (1+\epsilon)j$. This is achieved by proving the following two properties for all values of i and j.

Property 1: $\widetilde{U}_{i,\widetilde{j}} \leq U_{i,j}$

Property 2: $\widetilde{j} \leq (1+\epsilon)^{i/m} j$

We will prove these two properties using induction on i. First, consider the items only from the task T_1, i.e. $i = 1$. This implies that in the exact algorithm, there will be an update for $U_{i,p_{1,k}}$ corresponding to all $k = \{1, 2, \ldots, n_1\}$. Property 1 holds by equality. We can easily verify that $\widetilde{j}/(1+\epsilon)^{1/n} \leq c_{i,k}$ (Property 2) follows from the fact that $\widetilde{j}$ is the upper bound in the same interval as j, and hence $\widetilde{j}/(1+\epsilon)$ will definitely be in the preceding interval.

Let us now consider the induction step for any $i > 1$, assuming that both the properties hold true for $i-1$, i.e. we have dealt with the tasks T_1 to T_{i-1}. This step considers the pairs $(\delta_{i,k}, c_{i,k})$ in the set S_i. The entries in the array $U_{i,j}$ which are not updated definitely satisfy both the properties.

Now consider entries which are updated i.e. an item $(\delta_{i,k}, c_{i,k})$ was added to $U_{i,j}$ such that $U_{i,j+c_{i,k}}$ was updated. Since the claim is true for $i-1$, there exists $\widetilde{U}_{i-1,\widetilde{j}}$ such that $\widetilde{j} \leq (1+\epsilon)^{i-1/n} j$. Now, we consider $c_{i,k}$ which will be added to $\widetilde{j}$, and is rounded up to $\widetilde{u}$. Given the manner in which we constructed our intervals, we have

$$\widetilde{u}/(1+\epsilon)^{1/n} \leq \widetilde{j} + c_{i,k} \leq \widetilde{j}(1+\epsilon)^{i-1/n} + c_{i,k}$$

$$\widetilde{u}/(1+\epsilon)^{1/n} \leq (\widetilde{j} + c_{i,k})(1+\epsilon)^{i-1/n}$$

$$\Rightarrow \widetilde{u} \leq (\widetilde{j} + c_{i,k})(1+\epsilon)^{i-1/n}$$

Property 1 can be verified using the following steps, where the inequality holds by induction.

$$\begin{aligned}\widetilde{U}_{i,\widetilde{j}} &= min\{\widetilde{U}_{i,\widetilde{j}}, U_{i-1,\widetilde{j}} - \delta_{i,\widetilde{j}}/P_i\} \\ &\leq \{U_{i-1,j} - \delta_{i,j}/P_i\} = U_{i-1,j+c_{i,k}}\end{aligned}$$

□

3.3 Multicriteria Schedulability Analysis

In the previous section we addressed the single-criteria version of the problem, namely we assumed the processor's clock frequency to be prespecified. In this section, we relax this assumption and present a scheme to compute the *Pareto curve* containing the *Pareto-optimal* set of performance vectors $\{(c_1, f_1), \ldots, (c_n, f_n)\}$, where (c_i, f_i) denotes the hardware *cost* and the clock *frequency* for a particular schedulable implementation.

For simplicity of exposition, we will henceforth assume that the processor P's clock frequency is constant and all the execution times of the tasks are specified with respect to this clock frequency. Our objective will be to minimize P's utilization (by mapping certain subtasks onto hardware) and at the same time also minimize the total hardware cost. In other words, our goal is to compute the *cost-utilization* Pareto curve $\{(c_1, u_1), \ldots, (c_n, u_n)\}$ for a prespecified clock frequency of P. It is straightforward to see that such a Pareto curve can be easily transformed into a *cost-frequency* Pareto curve with P's utilization being ≤ 1 for the different frequency values.

Unfortunately, computing the exact *cost-utilization* Pareto curve is computation-

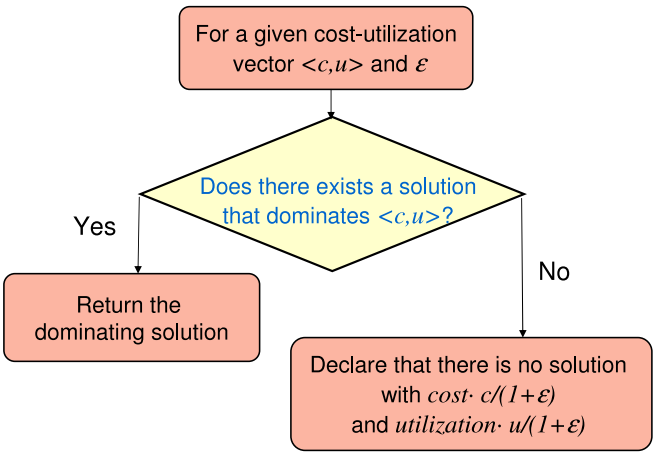


Figure 3.2: The GAP problem corresponding to our *cost-utilization* tradeoff problem.

ally intractable. This can be easily verified from the following two facts. First, the Pareto curve would typically contain an exponential number of points (which obviously cannot be computed in polynomial time). Second, computing any one point on the Pareto curve is NP-hard, as we showed in Section 3.2. Hence, our goal is to *approximately* compute this curve in polynomial time.

Recent work by Papadimitriou and Yannakakis [61] has shown that for any multiobjective optimization problem, there exists a polynomial-sized ϵ-approximate Pareto curve $\mathcal{P}_\epsilon$ for any given ϵ. Further, [61] showed that a necessary and sufficient condition for computing such a $\mathcal{P}_\epsilon$ in polynomial time is the existence of a polynomial-time algorithm for solving, what was referred to as the *GAP problem*. In what follows, we state the version of the GAP problem that arises in our setting and show that it can be solved in polynomial time.

3.3.1 The GAP Problem

For a two-dimensional multiobjective optimization problem, the GAP problem can be stated as follows: Given a vector $b = (b_1, b_2)$, either return a solution whose vector dominates b, or report that there is no solution whose vector is better than

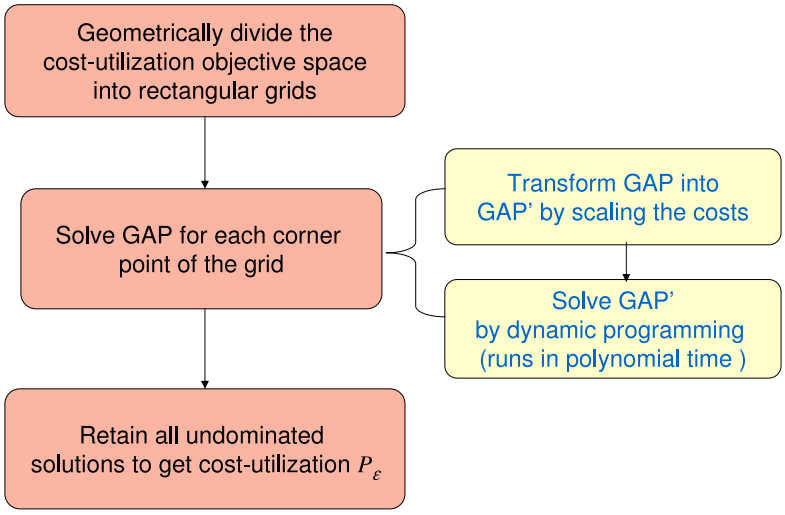


Figure 3.3: An FPTAS for computing $\mathcal{P}_\epsilon$ using an algorithm for solving GAP.

b by at least a factor of $1+\epsilon$ in both dimensions. In our setup, the objective is to minimize the utilization $U(S) = \sum_{i=1}^{m} \frac{E_i - x_{i,j}\delta_{i,j}}{P_i}$ and the cost $C(S) = \sum_{i=1}^{m} c_{i,j}x_{i,j}$, where S is the chosen implementation among the various available options (see Table 3.1). Hence, the corresponding GAP problem can be stated as: Given a cost c, a utilization u and an $\epsilon \geq 0$, either return a solution S such that $C(S) \leq c$ and $U(S) \leq u$, or else declare that there is no solution S such that $C(S) \leq \frac{c}{1+\epsilon}$ and $U(S) \leq \frac{u}{1+\epsilon}$ (see Figure 3.2). In this section, we will show that there exists a polynomial-time algorithm to solve this GAP problem.

Note that a polynomial-time algorithm to solve the GAP problem implies an FPTAS for computing P_ϵ. This is because the following FPTAS can be devised using the algorithm for solving GAP (shown schematically in Figure 3.3). First, geometrically partition the objective space along all dimensions with a ratio $1+\epsilon'$, where $\epsilon' = (1+\epsilon)^{1/2} - 1$. For each corner point of this grid, call the GAP routine (i.e. the algorithm for solving GAP) with the parameter ϵ', and keep all the undominated solutions (see Figure 3.4 for an illustration of this procedure). This implies that for each rectangle which contains a solution in the exact Pareto curve, there will also be a solution within the same rectangle which belongs to $\mathcal{P}_\epsilon$. The *distance* between

these two solutions can be bounded using the dimensions of the rectangle. Hence, for every solution s in the Pareto curve, there exists a solution q in $\mathcal{P}_\epsilon$ such that $\frac{q}{(1+\epsilon)} \leq s$. Moreover, because the number of rectangles is polynomially bounded, it follows that the number of points in $\mathcal{P}_\epsilon$ will also be a polynomial.

Theorem 3.3.1 *There exists an algorithm for constructing the cost-utilization ϵ-Pareto curve, which runs in time polynomial in the size of the input and in $\frac{1}{\epsilon}$.*

Proof: As discussed above, a necessary and sufficient condition for the existence of an FPTAS for computing the approximate cost-utilization Pareto curve $\mathcal{P}_\epsilon$ is that the following GAP problem should be solvable in time, which is polynomial in the input size and in $1/\epsilon$.

Problem Statement: Given a cost c, utilization u and an $\epsilon \geq 0$ either return a solution S such that $C(S) \leq c$ and $U(S) \leq u$, or else declare that there is no solution S such that $C(S) \leq \frac{c}{1+\epsilon}$ and $U(S) \leq \frac{u}{1+\epsilon}$.

Solution to the GAP Problem: We now present a polynomial-time algorithm to solve this GAP problem. It involves the following two steps:

- *Transforming Costs*

  Let $r = \lceil \frac{m}{\epsilon} \rceil$. Modify each cost $c_{i,j}$ to $c'_{i,j}$ such that $c'_{i,j} = \lceil \frac{c_{i,j} r}{c} \rceil$. This leads to the following properties:

  (a) If a solution with the transformed costs satisfies $C'(S) \leq r$, then $C(S) \leq c$.

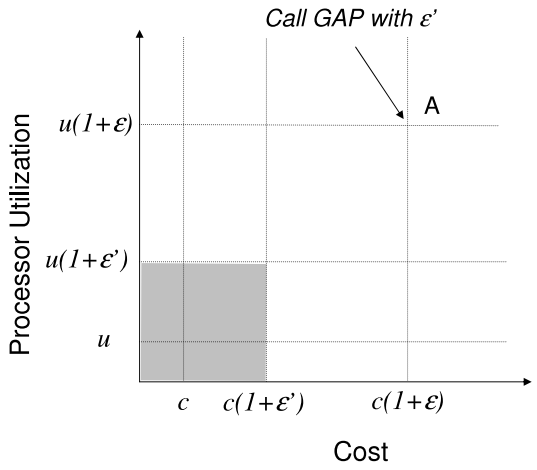


Figure 3.4: Solving the GAP problem for the corner point A will either return a dominating solution or declare that there is no solution in the shaded area.

Proof of Property (a):

$$\sum c'_{i,j}x_{i,j} = \sum \left\lceil \frac{c_{i,j}x_{i,j}r}{c} \right\rceil \geq \frac{r}{c}\sum c_{i,j}x_{i,j}$$

Hence, $C'(S) \leq r \Rightarrow \frac{r}{c}\sum c_{i,j}x_{i,j} \leq r$

This implies that $C(S) \leq c$.

(b) If the solution satisfies $C(S) \leq \frac{c}{1+\epsilon}$, then $C'(S) \leq r$.

Proof of Property (b):

$$C(S) \leq \frac{c}{1+\epsilon} \Rightarrow \sum c_{i,j}x_{i,j} \leq \frac{c}{1+\epsilon}$$
$$\Rightarrow \sum \frac{c_{i,j}x_{i,j}}{c}\frac{m}{\epsilon} \leq \frac{m}{\epsilon(1+\epsilon)}$$
$$\Rightarrow \sum \left\lceil \frac{c_{i,j}x_{i,j}r}{c} \right\rceil \leq \left\lceil \frac{m}{\epsilon(1+\epsilon)} \right\rceil$$
$$\Rightarrow C'(S) \leq \left\lceil \frac{m}{\epsilon} \right\rceil = r \Rightarrow C'(S) \leq r$$

Consider the problem of determining if there exists a solution with the modified costs such that $C'(S) \leq r$. Let us call this problem GAP$'$. From property (a), we know that if this problem returns an affirmative answer then the GAP problem would also return a dominating solution. On the

other hand, if GAP$'$ returns a negative answer then property (b) leads to the conclusion that there is no solution with cost $\leq c/(1+\epsilon)$. Hence, from the above properties we can infer that solving GAP$'$ is equivalent to solving the original GAP problem.

- *Solving GAP$'$*

  We present a dynamic programming algorithm to solve the GAP$'$ problem. This algorithm can be constructed with the following adjustments to Algorithm 5.

  1. Run Algorithm 5 with the modified costs $c'_{i,j}$.
  2. Instead of iterating over all the cost values up to mC, iterate only up to a cost value of at most r.
  3. Finally, if the minimum value in the final array $U_{n,\{1,\ldots,r\}}$ is such that it is $\leq u$, then return the solution otherwise declare that there is no solution.

  Computing each row of the table built by this dynamic programming algorithm requires $O(n_i r)$ running time. Hence, this algorithm runs in time $O(nm/\epsilon)$, where $n = \sum n_i$.

Hence, a polynomial-time algorithm exists for solving the GAP problem, which in turn proves our theorem. □

Now that we have presented the GAP subroutine for our problem, we can present the full algorithmic details for computing the *cost-utilization* Pareto curve. Recall that we already outlined this scheme in Figure 3.3. Algorithm 6 specifies the steps to compute the ϵ-approximate *cost-utilization* Pareto curve in some more detail. Note, that in step 1 of Algorithm 6 we partition only the cost space (and not both utilization and cost space). This is because if a point (c, u) dominates the corner

Algorithm 6 Approximating the Pareto curve.

1: Partition the range of costs from 1 to mC geometrically with a ratio $1 + \epsilon' = (1 + \epsilon)^{1/2}$, thus dividing the cost space into $O(log_{1+\epsilon} mC)$ coordinates.
2: For each coordinate b, call *Algorithm 1* with transformed costs $c'_{i,j} = \lceil \frac{c_{i,j} r}{b} \rceil$, where $r = \lceil \frac{m}{\epsilon'} \rceil$.
3: For each run of Step 2, find the solution with the minimum utilization.
4: Retain all the undominated solutions from the solutions found in Step 3. This will represent a ϵ-Pareto curve.

(c_1, u_1) and $u_1 < u_2$, then (c, u) definitely dominates (c_1, u_2) . In steps 2 and 3, we scale the costs, run Algorithm 1 for every co-ordinate in the partitioned cost space and retain the minimum utilization at each co-ordinate. The runtime complexity of this algorithm is $O(\frac{nm}{\epsilon} log_{1+\epsilon} mC)$.

3.4 Experimental Results

In this section we report some of the experimental results that we obtained by running our approximation algorithm on a set of synthetic task sets. (Note that to compute the exact Pareto curve, we need to run the Algorithm 5 and then retain all the undominated solutions.) We also compared these results with those obtained by running the optimal algorithm. In Section 3.4.1 we show the running times of the optimal and the approximation algorithms. In Section 3.4.2 we illustrate the difference in the sizes of $\mathcal{P}_\epsilon$ and the exact Pareto curve.

For our experiments we randomly generated tasks with execution requirements between 200 and 600 time units; the periods were between 600 and 20000 time units. The number of hardware implementation choices associated with any task was varied between 1 and 10, i.e. $1 \leq n_i \leq 10$. For each choice, the maximum value associated with any $\delta_{i,j}$ was set to E_i.

All the CPU times reported below were measured on a machine with Windows

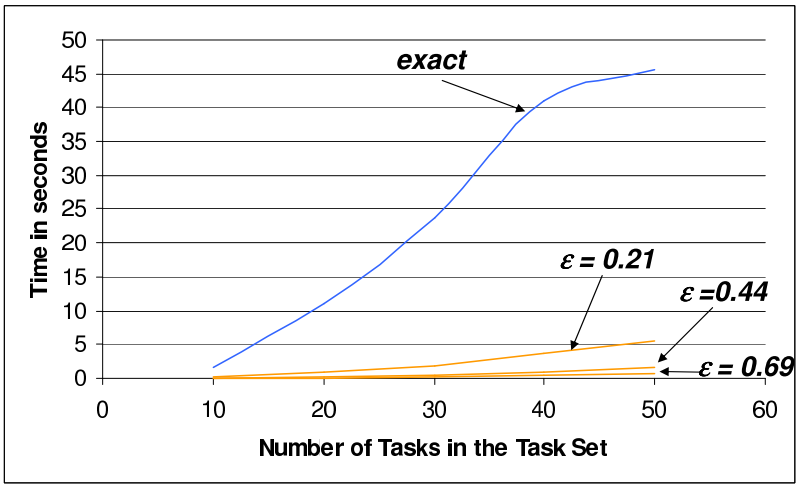


Figure 3.5: Graph comparing the running times of the exact and the approximate algorithms for various task sets with $C = 10000$.

XP, running on a 3.0 GHz CPU with 1 GB RAM. All the implementations were done in C++.

3.4.1 Running Times

Figure 3.5 shows the running times involved in computing the exact Pareto curve and the FPTAS for three different values of ϵ when the number of tasks in the task set is progressively increased from 10 to 50. These task sets were generated with the parameter $C = 10000$. It can be seen that even for small values of ϵ (e.g. when $\epsilon = 0.69$) the approximate algorithm runs about 40 times faster than the exact algorithm. For larger values of ϵ (e.g. $\epsilon = 3$), the speedups are even more significant (note that ϵ need not be ≤ 1).

The reason behind choosing the values 0.21, 0.44, and 0.69 for ϵ is as follows. Our approximation algorithm involves the computation of the value $(1 + \epsilon)^{1/2}$. This value might turn out to be an irrational number if ϵ is not carefully chosen. Hence, to avoid any possible rounding-off errors in our implementation, the above values

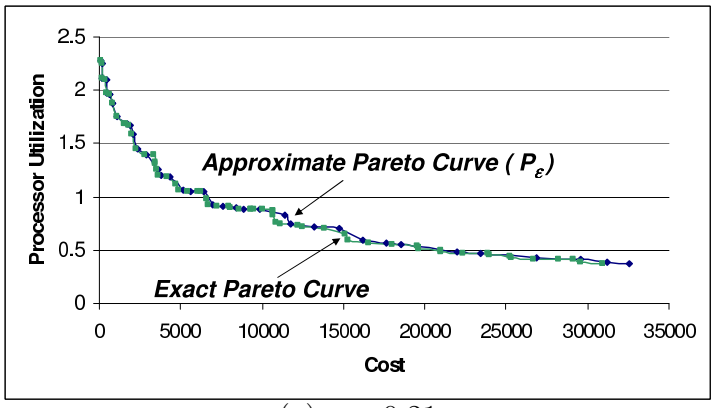
were chosen for ϵ.

3.4.2 Size of the Pareto Curves

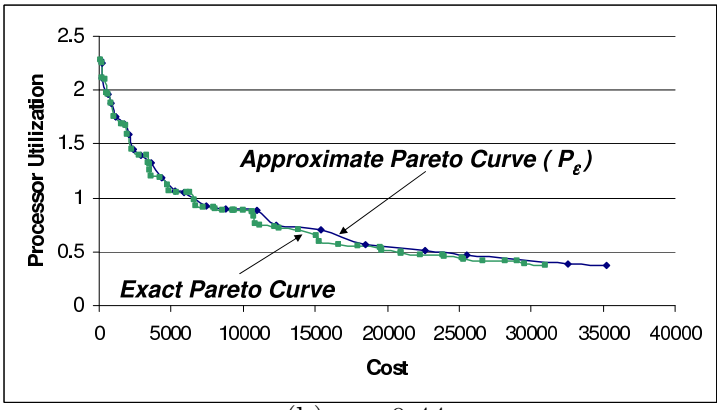
As discussed in Section 3.3, the *cost-utilization* Pareto curve typically contains an exponential number of points. The approximation algorithm generates a polynomial-sized ϵ-approximate Pareto curve. In this section, we compare the number of points in the exact Pareto curve and in $\mathcal{P}_\epsilon$. To help visualize the difference in their sizes, we choose a relatively smaller problem instance for our algorithm with a task sets of 10 tasks in each set and $C = 5000$. Figure 3.6 shows the exact Pareto curve and the $\mathcal{P}_\epsilon$ generated by our algorithm.

The following two observations can be easily visualized from these graphs: (i) the number of points in $\mathcal{P}_\epsilon$ decrease with a corresponding increase in the value of ϵ, and (ii) the gap between the exact and approximate curves widens with larger values of ϵ, implying that the relative error indeed increases.

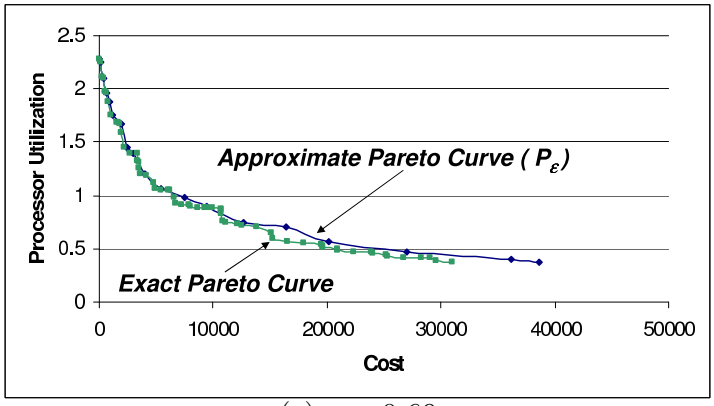
These graphs show the Pareto curves for a task set with 10 tasks. Table 3.2 lists the number of points in the exact Pareto curve and in $\mathcal{P}_\epsilon$ for task sets with 10, 20, 30, 40 and 50 tasks. The numbers in the rightmost column in this table are the number of points in $\mathcal{P}_\epsilon$ when the value of ϵ is set to 0.21, 0.44, 0.69, and 3. From this table it can once again be seen that as the relative error is allowed to increase, the size of the approximate Pareto curve decreases. Note from Table 3.2 that for small values of ϵ (e.g. $\epsilon = 0.21$), the size of the $\mathcal{P}_\epsilon$ contains up to 96% less points compared to the optimal Pareto curve.



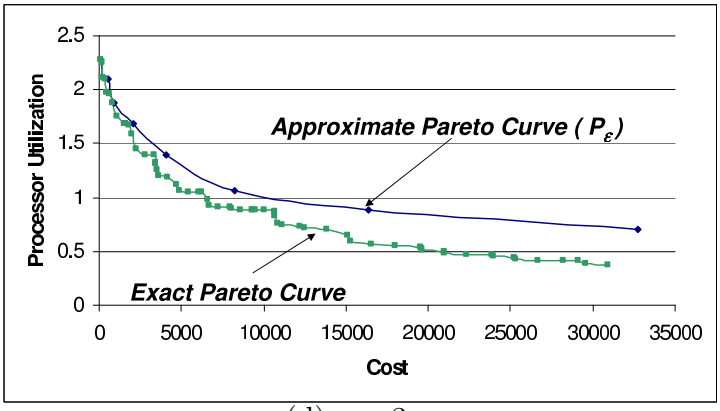
(a) $\epsilon = 0.21$



(b) $\epsilon = 0.44$



(c) $\epsilon = 0.69$



(d) $\epsilon = 3$

Figure 3.6: The exact and approximate Pareto curves for a task set with 10 tasks.

Task Sets	ϵ	**# Points on P_ϵ**
# tasks in the task set $\tau_1 = 10$ # points in the exact Pareto curve $= 62$	0.21 0.44 0.69 3	40 26 22 9
# tasks in the task set $\tau_2 = 20$ # points in the exact Pareto curve $= 239$	0.21 0.44 0.69 3	60 38 26 11
# tasks in the task set $\tau_3 = 30$ # points in the exact Pareto curve $= 828$	0.21 0.44 0.69 3	63 37 27 12
# tasks in the task set $\tau_4 = 40$ # points in the exact Pareto curve $= 1061$	0.21 0.44 0.69 3	76 44 31 12
# tasks in the task set $\tau_5 = 50$ # points in the exact Pareto curve $= 2033$	0.21 0.44 0.69 3	72 42 30 12

Table 3.2: Number of points in $\mathcal{P}_\epsilon$ generated by our proposed approximation algorithm, versus the number of points in the optimal Pareto curve.

3.5 Summary

In this chapter we introduced a multicriteria version of the classical schedulability analysis problem, which arises in hardware/software co-design setting. We showed that this problem is NP-hard even for simple task models and presented an approximation algorithm for solving it. The experimental results show that our approximation algorithm is not only computationally efficient, but also returns more meaningful results from a practical perspective (as discussed in Section 3).

There are a number of directions in which this work can be extended. The most notable among these being a possible extension of our scheme to account for communication costs and dependencies between parts of a task, some of which are

implemented in hardware and the remaining in software. Such details were abstracted away in this work for the sake of a clean theoretical formulation. For example, if the sub-tasks are realized on a reconfigurable device like the FPGA, it would be a challenging problem to extend the framework to incorporate the multitasking hardware tasks.

Chapter 4

GPU-Based Acceleration of System-Level Analysis Tools

In Chapter 2 and Chapter 3, we have discussed novel algorithmic techniques to speed-up computationally expensive cores in system-level performance analysis problems, namely, schedulability analysis and hardware/software co-design. In this chapter, we explore the possibility of using commodity graphics processing units (GPUs) to accelerate such computational kernels, and thereby improve the running time and usability of the design space exploration tools that use them.

There are two main reasons behind exploiting GPUs for such non-graphics related applications (in contrast to using, say, FPGA-based accelerators): (i) modern GPUs are extremely powerful (e.g. high-end GPUs such as nVIDIA GeForce 8800 GTX have a FLOPS rating of around 330 GigaFLOPS, whereas high-end general-purpose processors are only capable of around 25 GigaFLOPS) (ii) GPUs are now commodity items as their costs have dramatically reduced over the last few years. Hence, the attractive price-performance ratios of GPUs gives us an enormous opportunity to change the way design automation tools perform, with

almost no additional cost. In fact, recent years have seen the increasing use of graphics processing units (GPUs) for different general-purpose computing tasks. These span across numerical algorithms [35, 45], computational geometry [2], database processing [3], image processing [56, 58], astrophysics [67] and bioinformatics [53]. On the other hand, in spite of a wide variety computationally expensive system-level design and analysis problems that need to be regularly solved by design tools running on desktops and laptops equipped with high-end GPUs, the use of GPUs for accelerating such problems has not been sufficiently explored so far.

From a computer architecture standpoint, GPUs naturally support what are referred to as streaming algorithms [82]. In this chapter we reformulate a schedulability analysis problem and a multicriteria design space exploration problem related to hardware/software partitioning as a streaming algorithm which can be efficiently implemented on a GPU. Our results in this chapter show that using GPUs it may result in more than 16× speedup of the schedulability analysis problem and upto 100× speedup of the core of the design space exploration algorithm. These speedups will certainly improve the usability of a tool for system-level analysis, especially when used in an interactive fashion (i.e. where the designer repeatedly makes small changes to the problem and invokes the tool until a satisfactory solution is obtained). Our contributions are also significant because one of the core problems that we solve is a general knapsack problem (viz. the *multiple-choice* knapsack problem). Given the generality of this combinatorial optimization problem we believe that our results might initiate an interest to explore the use of GPUs for accelerating other problems as well.

Related Work

Over the last two decades, numerous approaches have been proposed to accelerate computationally expensive algorithms arising in the EDA domain. Many of these approaches are similar to our work in the sense that they also exploit some form of parallelism in the application. The most notable approaches have used multiprocessors and reconfigurable hardware like FPGAs. However, none of them have explored the possibility of employing GPUs, which in contrast to FPGAs involve no extra hardware cost since most computing platforms today are equipped with GPUs. Further, high-level APIs and programming languages such OpenGL [71] and CUDA [25] have greatly simplified the task of programming GPUs.

Results reported in [73, 87] represent early efforts towards using multiprocessors to reduce computation time of EDA algorithms like VLSI routing. In [29] parallel algorithms for design space exploration to be run on a multiprocessor system have been described. More recently, [43] has proposed techniques for reducing simulation time by building simulation models for execution on multiprocessor systems. Further, the use of reconfigurable computing to accelerate problems from the EDA domain has been proposed in [1, 72, 88]. All of these proposals are for accelerating the Boolean SAT problem which also lies at the core of several EDA applications. Other efforts in this direction include hardware-based acceleration for fast simulation [12, 39, 44]. Towards this, hardware acceleration is used to offload compute-intensive tasks from the software simulator.

In contrast to the above threads of work, the main advantage of our approach stems from the low cost associated with GPU-based acceleration since all desktop and notebook computers are now invariably equipped with GPUs. Hence, no extra hardware investment is necessary and EDA design tools can seamlessly incorporate our technique in a manner that is completely transparent to the end-user or the

design engineer using the tool. Further, as mentioned above, with the development of high-level APIs and programming languages for graphics programming, it is now easy to exploit GPUs to accelerate the back-end of any EDA design tool with relatively low additional programming effort and graphics-specific knowledge.

Organization of this Chapter

The rest of this chapter is organized as follows. In the next section we discuss the GPU architectures. In Section 4.2 we present our GPU-based schedulability analysis algorithm, followed by the description of the GPU-based engine for design space exploration algorithm in Section 4.3.

4.1 GPU Architectures

Before introducing our GPU based engine, we give a brief overview of the GPU architecture in this Section — we highlight the GPU pipeline, the features that make GPUs attractive *stream processors* and the challenges in programming the GPUs.

The GPU Pipeline: All of today's commodity GPUs structure their graphics computation in a fixed order of processing stages called the graphics pipeline. Figure 4.1 shows the pipeline stages in a modern GPU. The input to the pipeline is a list of geometry, expressed as vertices in object (3D) co-ordinates and the output is an image in a framebuffer (framebuffer is the portion of graphics card memory that holds the information necessary to display a screen image). The first stage of the pipeline (on vertex processors), performs geometric transformations on each vertex and transforms each vertex from object space (3D) into screen space (2D) and

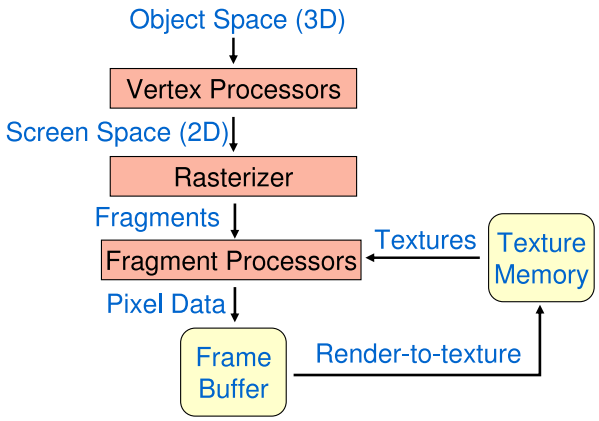


Figure 4.1: The GPU graphics pipeline.

assembles the vertices into triangles. Thus, the output of the first stage or geometry stage is triangles in screen space. The next stage, rasterization, determines the screen positions covered by each triangle. The result of the rasterization stage is a data stream of elements or fragments for each pixel location covered by a triangle. Each incoming data element has a set of texture co-ordinates that reference a texture memory (see Figure 4.1). The third stage or the fragment stage, consists of multiple fragment processors. They generate the addresses into the texture memory referred by the fragments and fetch their associated texture values. This data is used by a user defined program executing on the processors to compute the fragment color (i.e. the color for each pixel). The output is finally written to the frame-buffer memory.

In this work, we will concentrate only on the fragment processors. In fact, a vast majority of general-purpose GPU applications use only fragment programs for their computation. This is because — (i) they are last in the graphics pipeline and the output may be read directly (ii) they are highly parallel (they are more in number than vertex processors) (iii) they have a better memory read performance compared to the vertex processors.

GPUs as Streaming Processors: In order to meet the ever increasing performance requirements set by the gaming industry, modern GPUs use two types of

parallelism. First, multiple processors work on the vertex and fragment processing stage, i.e. they operate on different vertices and fragments in parallel. For example, a typical graphics card such as the nVidia GeForce 7900 GT has 8 vertex processors and 24 fragment processors. Second, each fragment processor can perform four concurrent vector operations such as instructions on the texture coordinates or on the color components of the incoming data stream.

Such explicit parallelism make GPUs an excellent platform for *stream processing* applications. Streaming processors read an input *stream* (which is a collection of records requiring similar computation), and apply the *kernel* (or operations to be performed on each element) to the stream and write the results into an output stream. Since there are no dependencies between the various elements of the stream, they provide immense data parallelism for the multiple processors running the kernels. Another feature of stream processing applications is that several kernels often operate successively on the streams, and the output stream of the leading kernel is the input stream for the following kernel (see Figure 4.2).

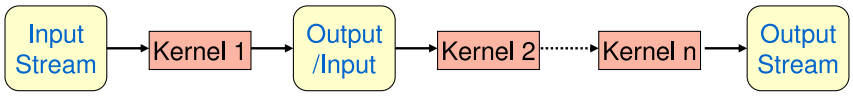


Figure 4.2: Streaming model that applies kernels to an input stream and writes to an output stream.

The Challenge in Programming GPUs: Programming the GPU is not as straightforward as implementing an application on the CPU. This is because, the GPU follows a highly parallel stream processing computational paradigm. The kernels on all of the fragment processors run in parallel and hence, can not have data dependency on each other. Thus, the challenge is to correctly identify the data parallel segments so that dependency constraints are not violated. Hence, given any application, it must be first appropriately recast as an streaming application for an efficient implementation on the GPU.

4.2 Case Study 1: GPU-based Acceleration of Schedulability Analysis Problem

In this section, we shall reformulate a standard demand bound criteria-based schedulability analysis algorithm [10, 17] as a streaming algorithm which can be efficiently implemented on a GPU. To illustrate this approach, we have chosen the recently proposed *recurring real-time task model* [9]. As discussed in Section 2.1, this model generalizes a number of well-known task models. Further, it can be used to model realistic applications with conditional branches and fine-grained deadline constraints. Hence, it forms a good starting point to explore the possibility of using GPUs for accelerating system-level timing and schedulability analysis problems. We have already discussed the schedulability analysis of the recurring real-time task model [9] in Section 2.1.1, and introduced the various notations associated with it. In the following Section, we shall briefly recall the scheme, and list the algorithm which forms the computationally intensive core of the scheme.

4.2.1 Schedulability Analysis of Recurring Real-Time Task Sets

Recall from Section 2.1.1 that the schedulability analysis of the recurring real-time task set is based on the *processor demand criteria* methodology. Towards this, the worst-case workload that can possibly be generated by a task (graph) is represented by a function called the *demand-bound function*. The demand-bound function of a task T, denoted by $T.dbf(t)$, takes as an argument a positive real number t and returns the maximum possible cumulative execution requirement of jobs that can be legally generated by T and which have their ready-times and deadlines both within a time interval of length t. A set of concurrently executing

Algorithm 7 Computing $T.dbf(t)$ using dynamic programming

Require: Task graph T, and a real number $t \geq 0$

1: **for** $e \leftarrow 1$ to nE **do**
2: $\quad t_{1,e} \leftarrow \begin{cases} d(v_1) & \text{if } e(v_1) = e \\ \infty & \text{otherwise} \end{cases}$
3: $\quad t^1_{1,e} \leftarrow t_{1,e}$
4: **end for**
5: **for** $i \leftarrow 1$ to $n-1$ **do**
6: $\quad$ **for** $e \leftarrow 1$ to nE **do**
7: $\quad\quad$ Let there be directed edges from the vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ to v_{i+1}
8: $\quad\quad t^{i+1}_{i+1,e} \leftarrow \begin{cases} \min\{t^{i_j}_{i_j, e-e(v_{i+1})} - d(v_{i_j}) + p(v_{i_j}, v_{i+1}) \\ \qquad +d(v_{i+1}) \,|\, j = 1, \ldots, k\} \text{ if } e(v_{i+1}) < e, \\ d(v_{i+1}) \text{ if } e(v_{i+1}) = e, \text{ and } \infty \text{ otherwise} \end{cases}$
9: $\quad\quad t_{i+1,e} \leftarrow \min\{t_{i,e}, t^{i+1}_{i+1,e}\}$
10: $\quad$ **end for**
11: **end for**
12: $T.dbf(t) \leftarrow \max\{e \,|\, t_{n,e} \leq t\}$

tasks $\mathcal{T}$ is then schedulable under a fully preemptive uniprocessor model if and only if for all $0 < t \leq t_{\max}$, $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$, where $t_{\max}$ is a function of the execution requirements of the tasks in $\mathcal{T}$ and their periods. This scheme therefore involves two stages:

(i) Computing $T.dbf(t)$ for all $t \leq t_{\max}$ and $T \in \mathcal{T}$, and

(ii) Checking that $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t, \ \forall\, 0 < t \leq t_{\max}$.

For the recurring real-time task model, it turns out that for an arbitrary task graph T, computing $T.dbf(t)$ for any t is NP-hard (see [20]) and therefore forms the computationally intensive kernel of the schedulability analysis algorithm. We had presented a dynamic programming (DP) based algorithm (Algorithm 1) for computing $T.dbf(t)$ for any task graph T and time interval length t, which was used in the context of the interactive schedulability in Chapter 2. In Algorithm 7 we re-list this algorithm (for computing $T.dbf(t)$), but without the statements (lines 11 to 16 of Algorithm 1) that were specially included for the creation of data structures used in the *interactive* framework (Chapter 2). This algorithm was already explained in detail in 2.1.3. In the following Section 4.2.2, we reformulate this computationally expensive algorithm as a streaming algorithm for implementing it on a GPU.

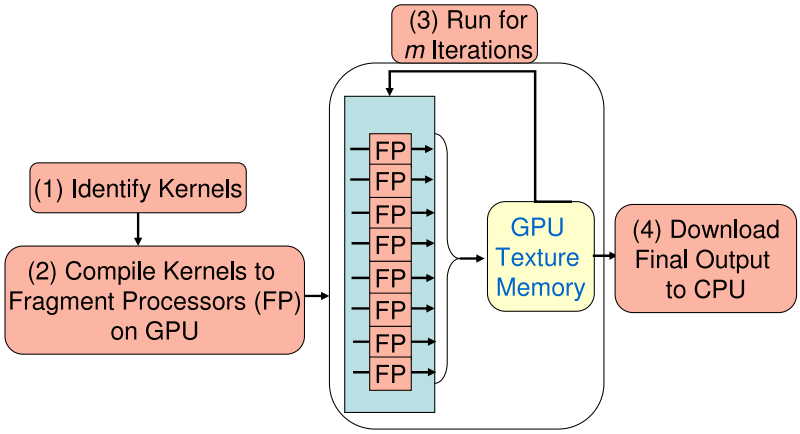


Figure 4.3: The overall scheme to design and implement a GPU based algorithm.

4.2.2 Schedulability Analysis on GPUs

Before discussing the details of our GPU-based engines, we outline the overall scheme to reformulate any algorithm as a stream processing application to run it on the GPU (Figure 4.3). The first and most important step is to identify the data parallel kernels. Next, we need to compile the kernels to the fragment processors and properly set up the GPU data structures. Finally, depending on the application, we determine the number of iterations on the fragment processors, and on completion, download the output to the CPU.

Following the above discussion, to take advantage of a GPU's parallel computation capability, we first identified portions of Algorithm 7 where computation on data elements can be done independently from each other and there is a significant amount of computational intensity relative to the time spent in transferring data. In Algorithm 7 the computation of the matrix cells (storing $t_{i,e}$ and $t^i_{i,e}$) in the inner loop of the dynamic programming (DP) algorithm (i.e. lines 6 - 10) can be done independently of each other. Therefore, the basic idea is to compute the DP-based matrix in a row-by-row fashion.

This matrix is stored as a texture in the texture memory of the GPU. Kernels (as explained in Section 4.1) are then used to implement the arithmetic operations

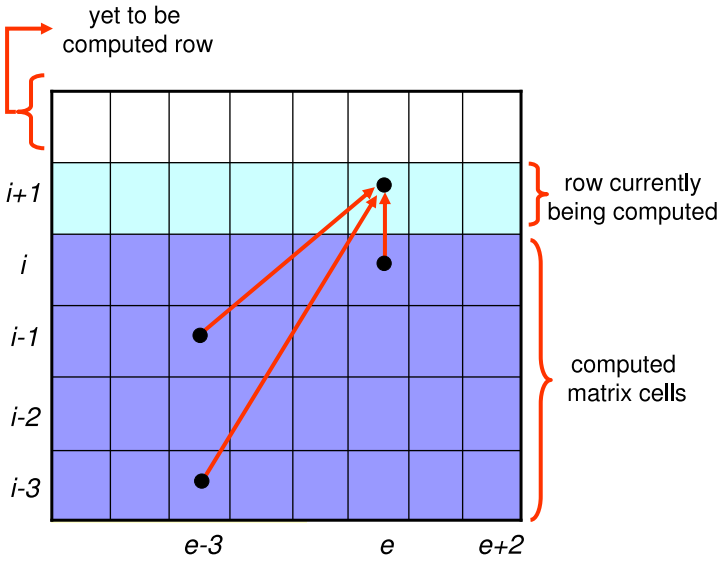


Figure 4.4: Data dependency graph for Algorithm 7. Computation of a cell in the DP matrix is dependent on texture fetching from already computed cells.

specified by the recurrence relations in lines 8 and 9 of Algorithm 7. A complete row of the matrix is computed in parallel by the fragment processors in the GPU. The newly-computed row is then stored in the texture memory. Finally, the subsequent kernel reads previously-computed rows form this memory and this process is repeated until the full matrix is computed.

Hence, our GPU-based implementation of Algorithm 7 requires n passes through the rendering pipeline, where n is the number of vertices in a task graph T' as explained in Section 2.1.3. In each pass, dependent texture lookups (lookup texture from computed addresses, i.e. lines 8 and 9 of Algorithm 7) must be performed. Figure 4.4 shows an example of this dependency for the $t_{i+1,e}$th cell – the computation of the $t_{i+1,e}$th cell depends on the values of the $t_{i,e}$th, $t^{i-1}_{i-1,e-3}$th and the $t^{i-3}_{i-3,e-3}$th cells. Clearly, such dependencies for any vertex i depends on the vertices from which there are incoming edges to i. It may be noted that since the GPU internal memory bandwidth is much slower compared to its compute capacity, dependent texture fetching does hamper the performance. However, the speedup achieved in the computation offsets this loss in performance — the result-

Algorithm 8 The Streaming Formulation of the DP

Require: Task graph T, and a real number $t \geq 0$

1: **for** $i \leftarrow 1$ to $n-1$ **do**

2: Let there be directed edges from the vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ to v_{i+1}

3: $tmp = f() - e(v_{i+1})$

4: $t^i_{i,new} \leftarrow \begin{cases} \min\{t^{i_j}_{i_j,tmp} - d(v_{i_j}) + p(v_{i_j}, v_{i+1}) \\ \qquad +d(v_{i+1}) \,|\, j = 1, \ldots, k\} \text{ if } e(v_{i+1}) < e, \\ d(v_{i+1}) \text{ if } e(v_{i+1}) = e, \text{ and } \infty \text{ otherwise} \end{cases}$

5: $t_{i,new} \leftarrow \min\{t_{i,prev}, t^i_{i,new}\}$

6: **end for**

ing overall speedup still being very attractive compared to a purely CPU-based implementation.

Algorithm 8 shows the pseudo-code of recursive algorithm for a kernel. $t_{i,prev}$ and $t^i_{i,prev}$ are the old values of a cell in the texture memory, and $t_{i,new}$ and $t^i_{i,new}$ are the new values computed by the kernel. $f()$ is a function which returns column value of the cell which is being computed by this kernel i.e. the corresponding value j (see Line 3). Lines 4 corresponds to the recursive equation in line 8 of the Algorithm 9.

Data Structures

This section discusses the data structure created on the GPU memory for *streams* of our GPU-based computation. The matrix computed by the DP algorithm (which stores $t^{i+1}_{i+1,e}$ and $t_{i+1,e}$ values) is of size $n \times nE$. As mentioned before, the GPU-based implementation of our algorithm has n passes through the rendering pipeline. Following the memory organization supported by GPU architectures, we used two buffers to compute the matrix - one of which serves as the source buffer (containing the already computed rows of the matrix) and the other serves as the destination buffer (containing the row being computed in a certain pass). During each pass through the rendering filter, the destination buffer of the previous pass serves as

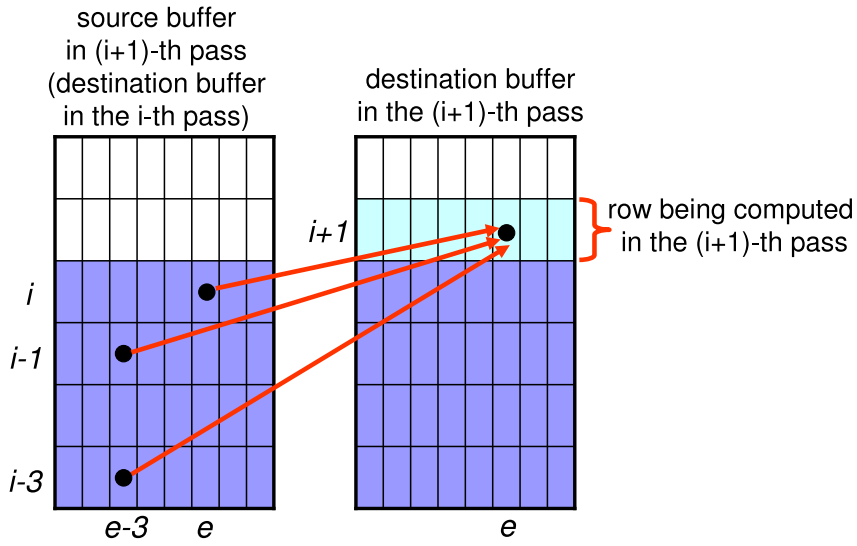


Figure 4.5: Data buffers in the GPU memory during the $(i+1)$-th pass through the rendering pipeline. Filling the destination buffer requires rendering a $(i+1) \times nE$ quadrilateral.

the current source buffer and their roles are interchanged from one pass to the next. Corresponding to the dependency relation shown in Figure 4.4, Figure 4.5 illustrates the use of the source and destination buffers during the $(i+1)$-th pass.

We represented a task graph using two RGB32 format textures. One texture was used to store vertex-related information (i.e. execution requirements, deadlines and the number of parent vertices). The other texture stored parental edge information for each vertex (i.e. ID of the parent vertices and the minimum intertriggering separation times).

The above matrix computation procedure was implemented using OpenGL's [71] *Render-to-Texture* support – `Pbuffer` extension or the newer lighter-weighted frame buffer object extension (FBO). For the former, a double-buffered `Pbuffer` is defined to avoid context switching. For FBO, two texture objects are attached to the frame buffer object bound for rendering, with one texture for writing and the other one for reading. These are swapped during each new pass as explained above. In each pass, the previous render target buffer binds as texture for reading and the previous buffer for reading becomes the render target.

Current GPU architectures and memory models impose a limit to the size of the dynamic programming problem that can be run on the GPU. For example, the size of the 2D render target is limited to 4096 in each dimension. Therefore, the largest possible matrix that it can hold is of size 16,777,216 cells. For a task graph with 150 vertices and maximum requirement of 600 per vertex, the required matrix size is 13,500,000. Hence, this is almost an upper bound on the problem size that can be supported.

Fortunately, GPUs offer a mechanism to write out multiple output targets in a single pass of the rendering pipeline, using what are called multiple render targets (MRTs). With MRTs, the fragment program can output up to four sets of color values; each set associated with Red/Green/Blue/Alpha (RGBA) components. This means we can output up to 16 floating point values per pixel. Hence, using MRTs it is possible to overcome the restriction on the problem size to a large extent by aggregating four dynamic programming matrix entries in a single pixel. The computations on each pixel can now handle four matrix entries instead of one. Correspondingly, the fragment program has to be rewritten to perform computation on four consecutive table entries. With the MRT implementation, it is easily possible to analyze task graphs of up to 250 vertices and execution requirements of up to 600 per vertex, which might be sufficient for many practical systems.

4.2.3 Results and Discussion

Our experiments were performed on a 3.0 GHz Pentium 4 CPU with 1 GB of RAM. It had a PCI express board equipped with an nVIDIA GeForce 8800 GTX GPU with 768 MB RAM. We used OpenGL with Shader Model 3.0 support. For render to texture support we have tried both the old `Pbuffer` extension and new frame buffer objects (FBO) extension.

n, E	Running Time (sec) CPU only	Running Time (sec) GPU (with FBO)	Upload time (sec)	Total Time (sec) GPU	Speedup
50, 600	0.330	0.155	0.035	0.190	1.74
100, 600	2.281	0.395	0.128	0.523	4.36
150, 600	17.924	1.123	0.367	1.490	12.03
200, 600	30.656	1.723	0.483	2.206	13.90
250, 600	48.735	2.372	0.657	3.029	16.09

Table 4.1: Comparing the running times of a purely CPU-based schedulability analysis versus a GPU-accelerated analysis.

We calculated the total processing time on the GPU to be the sum of data structure uploading time and the computation time on the GPU. The downloading time is negligible because we only need to download the last row of the matrix for each task graph (see Algorithm 7).

We observed that there is not much difference between the double `Pbuffer` and FBO extension based implementations. This is because they are both shortcuts to high-level memory management operations that do not make significant differences in performance. The main results of our study are tabulated in Table 4.1. As GPU implementations with multiple render targets (MRTs) are more scalable (i.e. can handle larger task graphs), we mostly show the results for this implementation.

The download and upload times are approximately linear in the size of the task graph instances. As the overheads involved in the stream-oriented reformulation of the schedulability analysis algorithm gets amortized by large task graphs, the GPU-based analysis shows its competitive advantage. For task graph with 250 vertices and maximum execution requirement of 600 per vertex, the GPU-based implementation shows more than 16× speedup.

For a set of synthetic task graphs, Figure 4.6 shows how the running times of the schedulability analysis algorithm scales with increasing sizes of the graphs (the maximum execution requirement associated with any vertex was always set to

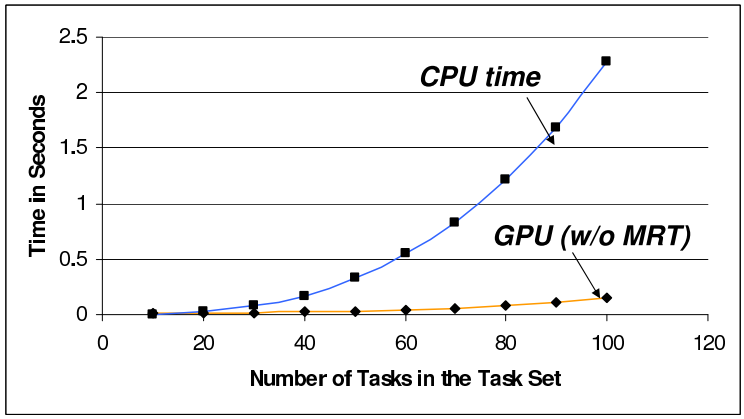


Figure 4.6: Running times of the schedulability analysis algorithm for a purely CPU-based implementation, versus a GPU-based implementation with a single render target.

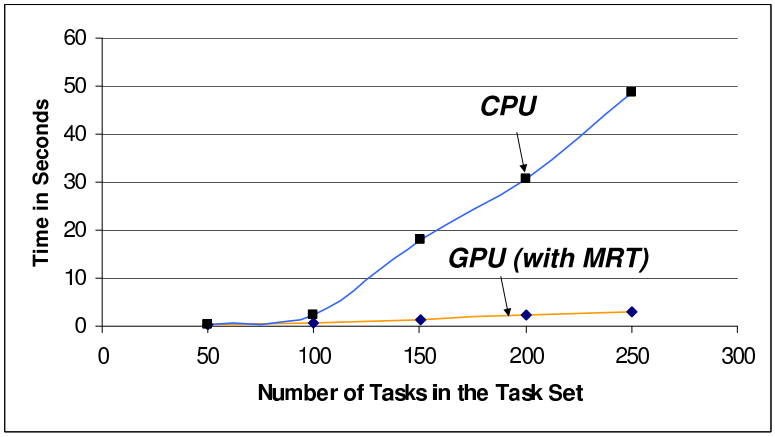


Figure 4.7: Running times of the schedulability analysis algorithm for a purely CPU-based implementation, versus a GPU-based implementation with multiple render targets.

600). Using a single render target, the largest task graphs that may be analyzed in our setup contain around 100 vertices. Compared to a purely CPU-based implementation, the GPU-based analysis results in more than 4× speedups, with the analysis times reducing from 2.2 sec to 0.5 sec. Much more attractive speedups may be obtained with larger task graphs, which can be handled when multiple render targets are used. Figure 4.7 shows this comparison. From this figure, it may be noted that for task graphs with around 250 vertices, the analysis time reduces from approximately 49 secs to 2.3 seconds. When built into a design tool, such speedups greatly improve the usability of the tool since feedback to any changes

in a design can now be obtained instantaneously.

4.3 Case Study 2: GPU-based Acceleration of Design Space Exploration Problem

In this section, we study our second case study where we use a common multi-criteria design space exploration problem (Chapter 3) to establish the utility of the GPUs in accelerating system-level design tasks. Recall that the multicriteria optimization problem was to identify the set $\{(c_1, f_1), \ldots, (c_n, f_n)\}$ which is referred to as the *Pareto curve*. Each point (c_i, f_i) in this set (called a *Pareto-optimal solution*) has the property that there does not exist any schedulable implementation of T_1 and T_2 with a performance vector (c, f) such that $c \leq c_i$ and $f \leq f_i$, with at least one of the inequalities being strict. Further, let $\mathcal{S}$ be the set of performance vectors corresponding to all schedulable implementations. Let $\mathcal{P}$ be the set of performance vectors $\{(c_1, f_1), \ldots, (c_n, f_n)\}$ corresponding to all the Pareto-optimal solutions. Then for any $(c, f) \in \mathcal{S} - \mathcal{P}$ there exists a $(c_i, f_i) \in \mathcal{P}$ such that $c_i \leq c$ and $f_i \leq f$, with at least one of these inequalities being strict (i.e. the set $\mathcal{P}$ contains *all* performance tradeoffs). The vectors $(c, f) \in \mathcal{S} - \mathcal{P}$ are referred to as *dominated solutions*, since they are "dominated" by one or more Pareto-optimal solutions as shown in Figure 3.6.

In this chapter, we present a GPU based engine, GPUPareto, for high speed computation of the Pareto curve $\mathcal{P} = \{(c_1, f_1), \ldots, (c_n, f_n)\}$. Our algorithm consists of the following two parts:

- The first part involves running a pseudo-polynomial time dynamic programming algorithm to find all the design points. This algorithm is the computationally expensive core of the design space exploration and in this chapter

we re-formulate this algorithm as a streaming algorithm to accelerate it on GPUs.

- The second part involves retaining the non-dominated solutions from the set of solutions found in the previous step. Since, this part is not amenable to GPU based acceleration, we run it on the CPU for optimized performance.

In essence, GPUPareto involves both GPU and CPU to achieve optimal performance improvement. In the following we shall briefly recall the task model and the formal problem statement, before introducing the design of GPUPareto.

4.3.1 Task Model

In this chapter, we use the sporadic task model in a preemptive uniprocessor environment to illustrate our GPU based design space exploration scheme. Thus, we are interested in the schedulability analysis of a task set $\tau = \{T_1, T_2, \ldots, T_m\}$ consisting of m hard real-time tasks. Any task T_i can get triggered independently of other tasks in τ. Each task T_i generates a sequence of jobs; each job is characterized by the following parameters:

- *Release Time*: the release time of two successive jobs of the task T_i is separated by a minimum time interval of P_i time units.
- *Deadline*: each job generated by T_i must complete by D_i time units since its release time.
- *Workload*: the worst case execution requirement of any job generated by T_i is denoted by E_i.

Throughout this chapter, we assume the underlying scheduling policy to be the earliest deadline first (EDF). Again, our algorithm can be suitably modified to handle other scheduling policies as well. Assuming that for all tasks T_i, $D_i \geq P_i$, the schedulability of the task set τ can be given by the following condition.

Theorem 4.3.1 *A set of sporadic tasks τ is schedulable under EDF if and only if*

$$(U = \sum_{i=1}^{m} \frac{E_i}{P_i}) \leq 1$$

where U is the processor utilization due to τ [11, 52].

4.3.2 The Problem Statement

In this section, we briefly recall the the multi-objective problem, and the pseudo-polynomial time algorithm for solving it, which were discussed in Section 3.2.

Recall that we are given a processor P, and a specified number of subtasks of each task T_i which can be implemented in hardware. For simplicity of exposition, we will henceforth assume that the processor P's clock frequency is constant and all the execution times of the tasks are specified with respect to this clock frequency. Our objective will be to minimize P's utilization (by mapping certain subtasks onto hardware) and at the same time also minimize the total hardware cost. In other words, our goal is to compute the *cost-utilization* Pareto curve $\{(c_1, u_1), \ldots, (c_n, u_n)\}$ for a prespecified clock frequency of P. It is straightforward to see that such a Pareto curve can be easily transformed into a *cost-frequency* Pareto curve with P's utilization being ≤ 1 for the different frequency values.

For each task T_i, let there be n_i hardware implementation choices. Each of these n_i choices is associated with a certain hardware cost. Choosing the jth implementation choice for the task T_i lowers its execution requirement on P from E_i

to $e_{i,j}$. Equivalently, the amount by which the execution requirement of T_i gets lowered on P is $\delta_{i,j} = E_i - e_{i,j}$. Hence, for each task T_i we have a set of choices $S_i = \{(\delta_{i,1}, c_{i,1}), \ldots, (\delta_{i,n_1}, c_{i,n_1})\}$, where $c_{i,j}$ is the hardware cost associated with the jth implementation choice. In this setup, the objective is to minimize the utilization $U(S) = \sum_{i=1}^{m} \frac{E_i - x_{i,j}\delta_{i,j}}{P_i}$ and the cost $C(S) = \sum_{i=1}^{m} c_{i,j}x_{i,j}$, where S is the chosen implementation among the various available options. In Chapter 3 we had illustrated this problem with the help of an example (see Section 3.2).

4.3.3 A Pseudo-polynomial Time Algorithm

Unfortunately, computing the exact *cost-utilization* Pareto curve is computationally intractable. This can be easily verified from the following two facts. First, the Pareto curve would typically contain an exponential number of points (which obviously cannot be computed in polynomial time). Second, computing any one point on the Pareto curve is NP-hard. This result on complexity was shown in 3.2.1 by a polynomial transformation of the knapsack problem.

Now, we present our algorithm to compute the Pareto curve. It consists of two parts. First, a dynamic programming algorithm (Algorithm 9) computes the minimum utilization that might be achieved for each possible cost. This algorithm runs in pseudo-polynomial time, and hence, turns out to be the expensive kernel of our scheme. In Section 4.3.4, we reformulate this algorithm to derive an accelerated GPU based scheme. The second part involves finding out all undominated solutions (*cost-utilization* Pareto curve) from the entire solution set found by the dynamic programming algorithm. This is a straightforward implementation, and is not a subject of discussion in this thesis.

Algorithm 9 description: Let $U_{i,j}$ be the minimum utilization that might be achieved by considering only a subset of tasks from $\{1, 2, \ldots, i\}$ when the cost is

Algorithm 9 Minimum-cost schedulability analysis

Require: The task set τ, and a set S_i for each task T_i.

1: $U_{0,0} \leftarrow \sum_{i=1}^{m} E_i/P_i$
2: **for** $j \leftarrow 1$ to mC **do**
3: $\quad U_{0,j} \leftarrow \infty$
4: **end for**
5: **for** $i \leftarrow 1$ to m **do**
6: $\quad$ **for** $j \leftarrow 0$ to mC **do**
7: $\quad\quad$ For each pair $(\delta_{i,k}, c_{i,k})$ that belongs to the set S_i
8: $\quad\quad$ $U_{i,j} \leftarrow \min\{U_{i-1,j}, U_{i-1,j-c_{i,k}} - \delta_{i,k}/P_i\}$
9: $\quad$ **end for**
10: **end for**

exactly j. If no such subset exists we set $U_{i,j} = \infty$. Let the maximum cost be C i.e. $C = max_{(i=1,2,\ldots,n;j=1,2,\ldots,n_i)} c_{i,j}$. Clearly, mC is an upper bound on the total cost that might be incurred. All other notations used are as introduced in Section 4.3.1 and Section 4.3.2. Lines 1 to 4 of Algorithm 9 initialize $U_{0,0}$ to $\sum_{i=1}^{m} E_i/P_i$, and $U_{0,j}$ to ∞ for $j = \{1, 2, \ldots, mC\}$. The values $U_{i,j}$ for $i = 1$ to $i = m$ are computed using the iterative procedure in lines 5 to 10. Thus, any non-infinity value $U_{n,j}$ for $j = \{1, 2, \ldots, mC\}$ implies a feasible design choice of the task set with utilization $U_{n,j}$ and cost j. It can be easily verified that the running time of Algorithm 9 is $O(nmC)$, where $n = \sum_{i=1}^{m} n_i$, and its space complexity is $O(m^2C)$.

We illustrate the working of the Algorithm 9 with the help of a toy task set. Consider a task set with 2 tasks — T_1 and T_2. The characteristics of jobs of task T_1 and T_2 are $\{P_1 = D_1 = 5, E_1 = 4\}$, and $\{P_2 = D_2 = 5, E_2 = 2\}$. The set of implementation choices for T_1 and T_2 are respectively $S_1 = \{(\delta_{1,1} = 1, c_{1,1} = 2), (\delta_{1,2} = 2, c_{1,5} = 5)\}$, and $S_2 = \{(\delta_{2,1} = 1, c_{1,3} = 2)\}$. The dynamic programming table for the utilization values $U_{i,j}$ built by the Algorithm 9 for this toy task set is shown in Table 4.2. We have chosen a small task set with only 2 tasks and a small value of $C = 5$, so that the table is small enough to be fit within the space restrictions of this paper. Row 0 in the table is filled according to the initialization in lines 1 to 4 of the Algorithm 9. For example,

$i\uparrow$	$j \rightarrow$										
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
2	1.2	∞	1	1	∞	0.8	∞	∞	0.6	∞	∞
1	1.2	∞	1	∞	∞	0.8	∞	∞	∞	∞	∞
0	1.2	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

Table 4.2: Illustration of the table built by Algorithm 9.

$U_{0,0} = \sum_{i=1}^{m} E_i/P_i = (2/5) + (4/5) = 1.2$. Row 1 and row 2 are constructed using the recursive procedure (lines 5 to 10 in Algorithm 9). Consider the computation of $U_{1,2}$ at row 1 and $j = 2$ (lines 7 and 8 in Algorithm 9). Row 1 corresponds to T_1, and hence the pairs of the set S_1 are being considered — $\{\delta_{1,1} = 1, c_{1,1} = 2\}$ and $\{\delta_{1,1} = 1, c_{1,1} = 2\}$. With the first choice $\{\delta_{1,1} = 1, c_{1,1} = 2\}$, we get $U_{1,2} = \infty$, and with $\{\delta_{1,2} = 2, c_{1,5} = 5\}$, $U_{1,2}$ evaluates to 1. Thus, the final value of $U_{1,2} = 1$, which is the minimum of the two. The values of other cells in this table may be worked out similarly.

4.3.4 The Design of GPUPareto

We outlined the overall scheme to reformulate any algorithm as a stream processing application to run it on the GPU (Figure 4.3). Recall that, we first need to appropriately identify the data parallel computation of the dynamic programming (DP) algorithm, Algorithm 9 to be mapped to the GPU. This is crucial because in Algorithm 9, the computation of the recurrence relation (line 7 to 8) involves non-trivial data dependencies. Towards this, we constructed the data dependency graph — Figure 4.8 shows the dependency for the $U_{i,j}$th cell. Recall that $U_{i,j}$ be the minimum utilization that might be achieved by considering only a subset of tasks from $\{1, 2, \ldots, i\}$ when the cost is exactly j. The computation of $U_{i,j}$ depends on the $U_{i-1,j}$th cell and on the values of $U_{i-1,j-c_{i,k}}$ i.e the $U_{i-1,j-c_{i,1}}$th cell, the $U_{i-1,j-c_{i,2}}$th cell, and so on. (Recall that $c_{i,j}$ is the hardware cost associated with the jth implementation choice of task T_i.) Thus, the figure depicts the fact

that the computation of $U_{i,j}$ depends *only* on previously computed cells i.e cells in $i-1$th row and *not* on cells in the ith row.

The observation here is that computation of U_{i,j_1} in the ith iteration is independent of U_{i,j_2}, where j_1 and j_2 may be any values between 1 and mC. In other words, given any iteration i computing any two values of $U_{i,j}$ for different j are independent of each other. Hence, in Algorithm 9, the computation of the cells in the inner loop of the dynamic programming algorithm (i.e. lines 6 to 9) can be done independently of each other. Therefore, the basic idea is to compute the DP-based matrix in a row-by-row fashion.

Now we are ready to describe our formulation of this DP as a streaming application — the cells in a previously computed row are the *streams*, and the arithmetic operations specified by the recurrence relations in lines 7 and 8 of Algorithm 9 are implemented as *kernels*. Each row (streams) of the DP-based matrix is stored as a texture in the texture memory of the GPU, and the recurrence relations (kernels) are compiled to the fragment processors (as explained in Section 4.1). A complete row of the matrix is computed in parallel by the fragment processors in the GPU. Note that since we have correctly mapped the data parallel sections to the fragment processors, there are no incorrect data fetches and we can achieve correct results. The newly-computed row is then stored in the texture memory. Finally, the subsequent kernel (i.e the next iteration of the DP) reads this computed row form this memory and this process is repeated for m passes, where m is the number of tasks in a task set τ as explained in Section 4.3.1. Of course, at the start of our streaming application, we have to set the initial value of the cells of the first row in the texture memory according to the initialization in lines 1 to 4 of Algorithm 9. Algorithm 10 shows the pseudo-code of recursive algorithm for a kernel. U_{prev} is the old value of a cell in the texture memory, and U_{new} is the new value computed by the kernel. $f()$ is a function which returns column value of the

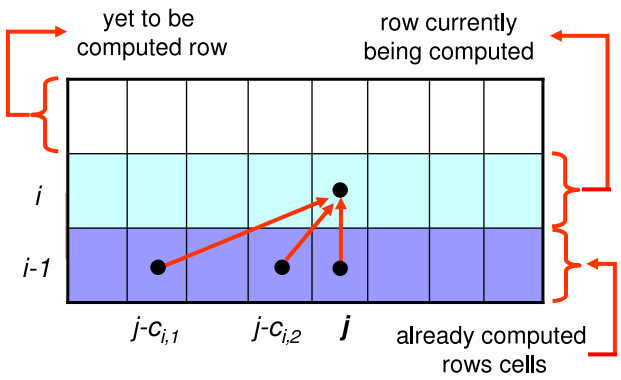


Figure 4.8: Data dependency graph for Algorithm 9.

Algorithm 10 The Streaming Formulation of the DP

Require: The task set τ, and a set S_i for each task T_i.

1: **for** $i \leftarrow 1$ to m **do**
2: For each pair $(\delta_{i,k}, c_{i,k})$ that belongs to the set S_i
3: $tmp = f() - c_{i,k}$
4: $U_{new} \leftarrow \min\{U_{prev}, U_{tmp} - \delta_{i,k}/P_i\}$
5: **end for**

cell which is being computed by this kernel i.e. the corresponding value j (see Line 3). Thus, $U_{tmp} - \delta_{i,k}/P_i$ (line 4) corresponds to the recursive equation in line 8, Algorithm 9.

Data Structures

This section discusses the data structure created on the GPU memory for *streams* of our GPU-based computation. We need to store two rows (which stores $U_{i,j}$ values) each of size $m \times C$ – one previously computed row is being read and one row is being currently computed by the DP algorithm. Following the memory organization supported by GPU architectures, we used two texture buffers to store the cells of the row - one of which serves as the source buffer (containing the previously computed row of the matrix) and the other serves as the destination buffer (containing the row being computed in a certain pass). During each pass through the GPU pipeline, the destination buffer of the previous pass serves as

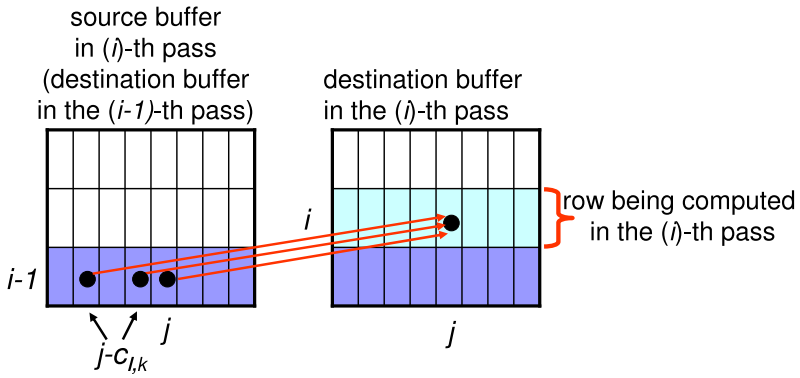
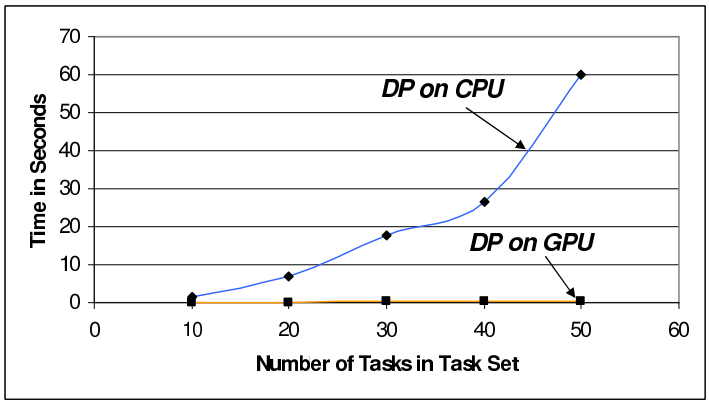


Figure 4.9: Data buffers in the GPU memory during the (i)-th pass through the rendering pipeline.

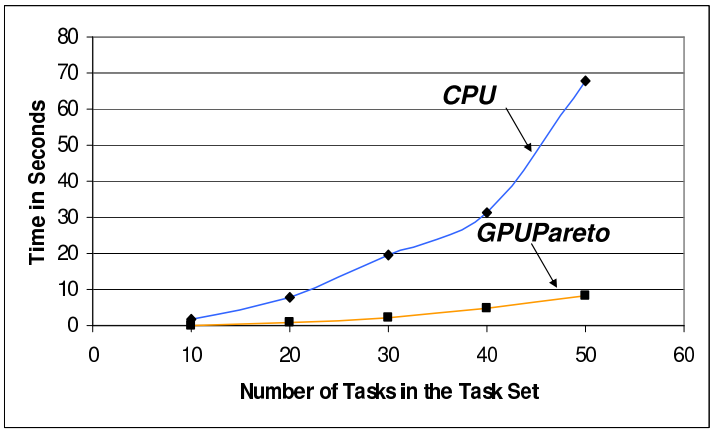
the current source buffer and their roles are interchanged from one pass to the next. Corresponding to the dependency relation shown in Figure 4.8, Figure 4.9 illustrates the use of the source and destination buffers during the (i)-th pass.

The above matrix computation procedure was implemented using OpenGL's [71] *Render-to-Texture* support. The two texture objects are attached to the frame buffer object bound for rendering, with one texture for writing and the other one for reading. These are swapped during each new pass as explained above. In each pass, the previous render target buffer binds as texture for reading and the previous buffer for reading becomes the render target.

In this Section we have discussed the GPU based dynamic programming algorithm, which is the first part our design space exploration algorithm. The second part involves retaining the undominated solutions (*cost-utilization* Pareto curve) from the entire solution set found in the previous step. Since, this is not compute intensive and is not amenable to GPU-based acceleration, we implement it on the CPU. (The algorithm is straightforward and is not elaborated due to space constraints.) Thus, our engine, GPUPareto leverages both CPU and GPU for efficient design space exploration.



(a) Only DP Algorithm Analysis Time



(b) The Overall Analysis Time

Figure 4.10: Running times for a purely CPU-based implementation, versus a GPU-based implementation - GPUPareto.

4.3.5 Experimental Results

In this section we report some of the experimental results that were obtained by running our GPUPareto engine on a set of synthetic task sets. We compared these results with those obtained by running a pure CPU implementation.

For our experiments we randomly generated tasks with execution requirements between 200 and 600 time units; the periods were between 600 and 20000 time units. The number of hardware implementation choices associated with any task was varied between 1 and 10, i.e. $1 \leq n_i \leq 10$. For each choice, the maximum value associated with any $\delta_{i,j}$ was set to E_i. The parameter C, which is the

n	Upload time (seconds) GPU	Running Time (seconds) GPU	Download Time (sec) GPU	Compute Undominated (sec) CPU	Total Time (seconds) GPUPareto	Total Time (seconds) only CPU	Speedup
10	0.01598	0.03095	0.00865	0.03489	**0.09**	**1.572**	**17.380**
20	0.02838	0.0778	0.01031	0.88549	**1.002**	**7.695**	**7.680**
30	0.03857	0.19547	0.01534	1.99349	**2.243**	**19.542**	**8.713**
40	0.04868	0.30509	0.02056	4.49815	**4.872**	**31.171**	**6.397**
50	0.06009	0.47448	0.02542	7.85111	**8.411**	**67.981**	**8.082**

Table 4.3: Detailed breakdown of time taken by GPUPareto and comparison with a purely CPU-based analysis.

maximum cost associated with any implementation choice was set to 16384 for our experiments. This number was chosen because graphics processors lack integer arithmetic. Using floating point values might lead to wrong address calculations (see line 3, Algorithm 10) due to improper rounding-off. Thus, if C is not a power of 2 for a given task set, one needs to choose the next higher power of 2 as an upper bound. To show that this is not a restriction of our scheme, we choose 16384 (2^{14}) and show that even for such large values our DP algorithm runs within fraction of a second. Furthermore, with $C = 16384$, there are upto 16384 design points, and typically around 6100 points on the pareto curve for task set with around 50 tasks. Such large design space instances are clearly very suitable to test the applicability of a the GPUPareto scheme.

All the CPU times reported below were measured on a machine with Windows XP, running on a 3.0 GHz CPU with 1 GB RAM. Our machine had a PCI express board equipped with an nVIDIA GeForce 8800 GTX GPU with 768 MB RAM, where we conducted our GPU experiments. All the implementations were done in C++. For the GPU implementation, we used OpenGL with Cg as the shader language (for programming the fragment processors).

Figure 4.10(a) shows the time taken to compute the DP on the CPU versus time taken on the GPU, when the number of tasks in the task set is progressively increased from 10 to 50. Our efficient implementation on the GPU achieves tremendous (upto 100×) speedup. Figure 4.10(b) shows the overall running time involved

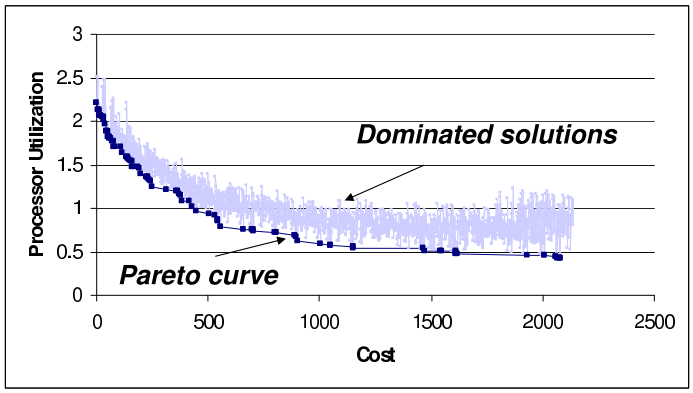


Figure 4.11: The Pareto curve obtained for a task set of 10 tasks.

in computing the exact Pareto curve on the CPU versus time taken by the GPU-Pareto engine. In order to accurately report the total processing time on the GPU, we take into account the sum of data structure uploading time to GPU memory, the computation time on the GPU and the downloading time from GPU memory. The total time taken by the GPUPareto engine adds this sum with the time taken to run the second part of our algorithm, which is run on the CPU. (Recall, that to compute the exact Pareto curve, we need to run (i) the Algorithm 9 (GPU implementation) and (ii) then retain all the undominated solutions (CPU implementation).) For the interested reader, the detailed breakdown of our results for the 5 different task sets is tabulated in Table 4.3. The table also presents the comparison with a CPU based implementation in the last column.

Compared to a purely CPU-based implementation, the GPU-based analysis results in significant speedups, with the analysis times reducing from more than a minute to less than 9 seconds. Such speedups allow a designer to get almost instantaneous feedback during an interactive design session with an automated tool, thereby improving design productivity. Further, this comes at no additional cost, assuming that the desktop/notebook computer running the design tool already has a commodity GPU.

Finally, we show the results that are obtained by GPUPareto for two different task

sets. In Figure 4.11, the Pareto Curve is shown for a task set with 10 tasks. The shaded region in the graph show the dominated solutions and the thick line depicts the Pareto curve.

4.4 Summary

Using two case studies, we showed that modern commodity graphics hardware may be exploited to accelerate computationally expensive kernels in design space exploration tools. In particular, we presented GPU based engines to solve a heavy-duty schedulability analysis problem of a generalized task model and a standard multiobjective hardware/software co-design problem. We showed that our implementations achieve very attractive speedups compared to a standard CPU-based implementation.

It is worth mentioning here that GPUs also have certain disadvantages. Firstly, they consume significant amounts of power. However, we do not envision this to be a problem because any design space exploration tool using GPUs will typically run in a general purpose computation environment where resources are not hard limitations. Secondly, of late the recent increase in precision to 32-bit floating point has enabled a host of new GPGPU applications, but 64-bit double precision arithmetic still remains a distant promise [59]. Although the lack of double precision hampers or prevents GPUs from being applicable to very large-scale computational science problems, as our case studies show a large number of design space exploration tools may still leverage the GPU acceleration power without loss of accuracy. Hence, inspite of the challenges, we believe that the potential benefits are too large to ignore as demonstrated in this work.

Chapter 5

Performance Analysis of FlexRay-based ECU Networks

So far, in this thesis we have dealt with issues in design space exploration that arise in the context of general system-level analysis problems, namely, schedulability analysis and multicriteria hardware/software co-design. In this chapter, we shall be concerned practical issues which arise specifically in the automotive electronics domain. Real-time embedded systems in this domain have been of particular interest since the last two decades as there has been a phenomenal increase in the use of electronic components in automotive systems, resulting in the replacement of purely mechanical or hydraulic-implementations of many functionalities. The main motivation behind this stems from lower cost, reduced weight, new and innovative functionalities and the need for faster design cycles.

In spite of this rapid increase of software content and communication complexity, the system-level analysis and design space exploration methodologies for the automotive domain are still not mature. In particular, newly introduced automotive specific bus protocols have several characteristics which must be taken into account

for such system-level timing analysis. This is will be of focus in the chapter, and towards this, in the following section we present a brief background.

Background and Related Work

In earlier designs of automotive electronics, different functions were implemented as stand-alone electronic control units (ECUs), with each ECU consisting of one or more microcontrollers and a set of sensors and actuators. However, with the rapid increase in the complexity of the different functionalities, it became imperative to have distributed implementations, where different parts of a task are implemented on different ECUs with messages and signals being exchanged between them. For example, an ECU implementing *crash preparation* needs inputs from *wheel rotation sensors*, *radars*, and ECUs implementing tasks such as *object detection*, *data fusion* and *object selection*. Today, in high-end cars, it is common to have around 70 ECUs exchanging upto 2500 signals between them [5]. Hence, it is infeasible to connect the different ECUs with point-to-point links. This has led to the development of bus-based ECU networks, where communications between multiple ECUs are multiplexed over one or more shared buses. Consequently, this also gave rise to the need for different communication protocols specifically targeting automotive communication systems.

Today, the most commonly used protocols [57] include the Controller Area Network (CAN) [18], the Local Interconnection Network (LIN) [51] and the J1850 from the Society for Automotive Engineers (SAE) [41, 54]. The different protocols can be classified into two major groups: (i) time-triggered, and (ii) event-triggered. Communication activities in the latter class are triggered by the occurrence of specific events and the protocol defines a policy for resolving the contention for the shared bus when messages from multiple ECUs or tasks are ready at the same

time. For example, in the case of CAN, data is segmented into *frames* and each frame is labeled with a priority which is used to resolve bus contention. Time-triggered protocols, on the other hand, schedule communication activities or frame transfers at predetermined points in time, which are commonly referred to as *slots*. The sequence of slots and their lengths for different message types are statically defined and the resulting schedule repeats itself infinitely.

Event-triggered protocols are clearly more efficient in terms of communication bandwidth usage and allow incremental system design (i.e. new ECUs or tasks can be added without redesigning the system from scratch). However, they are difficult to analyze because of their dynamic nature. Hence, verifying timing properties and detecting faults often become problematic. This poses a serious hindrance to their deployment when the functions involved are safety-critical and require hard real-time guarantees. On the other hand, time-triggered protocols are highly predictable in terms of their temporal behavior, but suffer from poor communication bandwidth utilization and are inflexible. The addition of new ECUs, or the modification of any tasks require a complete redesign and reevaluation of the entire system.

As a result, recently there has been a lot of emphasis on hybrid protocols, that combine the time-triggered and event-triggered paradigms. Protocols in this class include TTCAN [80], FTT-CAN [31] and FlexRay [33]. FlexRay is currently backed by many major automotive companies and will most likely become the de-facto standard for automotive communication systems very soon. This has led to a lot of recent interest in timing and predictability analysis techniques and tool-support targeting FlexRay-based designs.

Our work is in line with these efforts and proposes an analytical framework for compositional performance analysis of a network of ECUs that communicate via

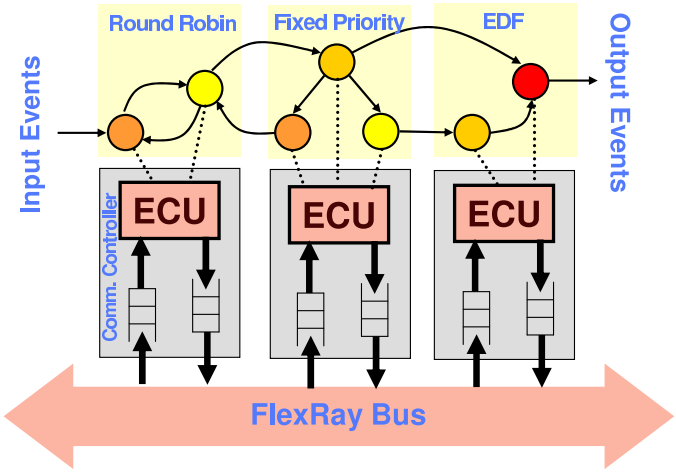


Figure 5.1: A FlexRay-based network of ECUs, with an application partitioned and mapped onto multiple ECUs.

a FlexRay bus. Given a specification of the applications running on the system, their partitioning and mapping on the different ECUs, their activation rates and the mapping of the resulting messages onto the different FlexRay slots along with the message priorities (see Figure 5.1), our framework can be used to answer various performance analysis-related questions. These include the maximum end-to-end delay experienced by the different message types, the amount of buffer space required within a communication controller associated with an ECU and the utilizations of the different ECUs and the FlexRay bus. Our framework can also be used for deriving the parameters of the FlexRay protocol (e.g. lengths of the static and dynamic segments and priorities of the messages mapped onto the dynamic segment). Further, it can help in resource dimensioning (e.g. designing the various ECUs) and determining optimal scheduling policies for multitasking ECUs.

In the FlexRay protocol, a communication cycle consists of a combination of a time-triggered or static (ST) segment and an event-triggered or dynamic (DYN) segment. Such a communication cycle is repeated in a periodic fashion. The ST segment uses a time-division multiple access (TDMA) scheme and the DYN segment uses—what is often referred to as—*Flexible TDMA*. The ST segment has

all the virtues of a time-triggered paradigm, i.e. the timing properties of messages mapped onto this segment are highly predictable. But it is mostly suited for periodic messages and has low communication bandwidth utilization. The DYN segment compensates this drawback, but suffers from the usual shortcomings of an event-triggered paradigm. As a result, most of the current implementations of FlexRay heavily lean towards using only the ST segment, with the DYN segment being unutilized. The only advantage of FlexRay that is being exploited in this process is its high bandwidth. To fully utilize the benefits of this protocol, it is important that suitable analysis techniques be developed that can provide timing and performance guarantees for messages mapped onto the DYN segment as well. This is complicated because of two reasons: (i) the DYN part of the protocol is more complex than the ST part, and (ii) the potential messages targeted for the DYN segment tend to be more irregular (e.g. high-volume multimedia data) than those mapped onto the ST segment (the DYN segment has been specifically designed for such messages).

Commercially available design tools for FlexRay-based systems (e.g. those from dSPACE [28] and DECOMSYS [27]) today mostly rely on simulation. As a result, they are time consuming to use and cannot provide formal performance guarantees, which are important in the automotive domain. Although formal timing analysis techniques have been proposed for protocols such as CAN [75, 77] and TTP [63], none of them seem to extend in a straightforward manner to model the DYN segment FlexRay.

Very recently, the first attempt to formally model the behavior of the DYN segment was reported in [64]. Given the arrival rates of the different message streams mapped onto the DYN segment, [64] computes the worst-case delay experienced by any message due to blocking by the ST segment and contention from higher priority messages. Computing this worst-case delay was shown to be similar to

a bin covering problem [24] and was solved using an integer linear programming (ILP) formulation. Further, computationally efficient (but pessimistic) heuristics were also presented to bound this delay. Although, this certainly represents an important step towards formally analyzing the FlexRay protocol, it suffers from certain drawbacks which might hamper its application to real-life problems. The first, and most important of these being that [64] analyzes the FlexRay bus in isolation, i.e. requires the input rates or periods of the arriving messages and computes the worst-case delay due to transmission over the bus. A system designer, on the other hand is typically interested in computing the worst-case end-to-end delays of messages originating from a sensor, passing over multiple ECUs and the FlexRay bus, and finally activating an actuator (see Figure 5.1 for an illustration). In this process, a message stream arriving at the FlexRay bus need not be purely periodic and might get modified depending on the scheduling policies on the different ECUs.

The framework we present in this chapter addresses this concern. It is fully compositional and models both the ECUs and the FlexRay bus in a seamless manner. Hence, it does not make any a priori assumption on the timing properties of the message streams arriving at the bus. Further, in contrast to [64]—which is only restricted to computing the worst-case response times of messages—our framework can be used to answer a wider variety of performance-related questions and will also be helpful for synthesizing a FlexRay schedule (i.e. determine the slot sizes and message priorities) when maximum end-to-end delays are provided as design constraints. Lastly, our approach does not involve any computationally expensive step like solving an ILP and would hence scale to real-life settings. We have implemented our framework using a combination of Java and Matlab, which can be used as a stand-alone design tool, or can serve as a plugin to standard tool suites (e.g. DECOMSYS Tools [27]). Such a plugin can be used to obtain hard performance

guarantees, which can then be cross-validated using simulation.

Organization of the chapter

The rest of this chapter is organized as follows. In the next section we briefly discuss the FlexRay protocol. In Section 5.2 we give an overview of our basic framework and the challenges in modeling the DYN segment of FlexRay. In Section 5.3, we introduce the working of our scheme with the help of small examples of FlexRay based networks. This is followed by a formal performance model for FlexRay, which is the main result of this work. A case study is presented in Section 5.5.

5.1 Overview of FlexRay

As mentioned in the previous section, each FlexRay communication cycle is partitioned into a ST and a DYN segment. The lengths of these segments need not be equal, but are fixed over the different cycles (hence these lengths are among the parameters that need to be determined when the FlexRay schedule is synthesized). The ST segment is further partitioned into a fixed number of equal-length slots. Each slot is allocated to a specific task and a task is allowed to send a message only during its allocated slot. If a task has no messages to send, then its slot goes empty (i.e. other tasks are not allowed to use it).

The DYN segment is also partitioned into equal-length slots, but each slot size is much smaller and is referred to as a *minislot*. Tasks which send messages on the DYN segment are assigned fixed priorities. At the beginning of each DYN segment, the highest priority task is allowed to send a message. The length of such a message can be arbitrarily long (i.e. can occupy an arbitrary number of

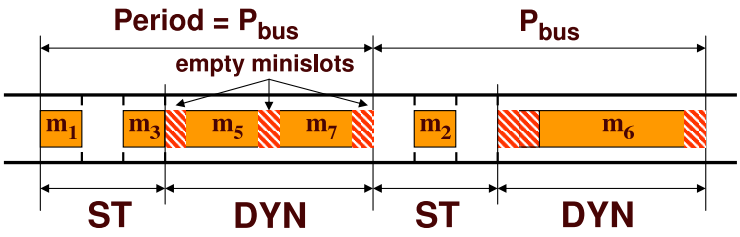


Figure 5.2: Two typical FlexRay communication cycles.

minislots), but has to fit within one DYN segment. However, if the task has no message to send, then only one minislot goes empty. In either case, the bus is then given to the next highest-priority task and the same process is repeated till the end of the DYN segment. Further, when its turn comes, a task is only allowed to send a message if it fits into the remaining portion of the DYN segment. For further details of this protocol, we refer the reader to the excellent description in [64] or to the full specification [33].

As an example, consider eight tasks $T_1, \ldots, T_8$ mapped onto different ECUs, which send messages on the FlexRay bus. Any message sent by a task T_i is labeled as m_i. Tasks T_1, T_2 and T_3 send messages over the ST segment and T_4 to T_8 over the DYN segment. For the DYN segment, the priorities of the tasks decrease from T_4 to T_8. Figure 5.2 shows two consecutive FlexRay communication cycles resulting from this mapping. In the first cycle, task T_2 has no message to send (hence the corresponding slot in the ST segment is empty) and in the second cycle T_1 and T_3 have nothing to send.

Similarly, in the first cycle, tasks T_5, T_6 and T_7 have messages to send, but not T_4 and T_8. Hence, there is one empty minislot corresponding to T_4 in the DYN segment, followed by the message m_5. The size of m_6 is bigger than the remaining length of the DYN segment, hence it is not sent; instead there is one empty minislot in its place. This is followed by m_7 and another empty minislot resulting out of no message from T_8. In the second cycle, T_4 and T_5 have no messages to send,

which results in two empty minislots. These are followed by m_6 which could not be sent in the first cycle. The DYN segment ends with one empty minislot which might either be because T_7 had nothing to send or its message was longer than one minislot.

It may be noted that (i) the ST and DYN segments are independent of each other, and (ii) techniques for analyzing the timing behavior of the ST segment are already known (because it uses a TDMA scheme) [63, 76]. Hence, from now on we will only focus on modeling the behavior of the DYN segment (however, we will of course take into account the blocking effects of the ST segment).

5.2 Basic Framework

In this section we give an overview of our basic modeling framework and the challenges faced in modeling the DYN segment of FlexRay. In the next section we show how these challenges are addressed. Our modeling techniques are motivated by [22], where a mathematical framework was presented for analyzing the timing properties of multiprocessor embedded systems. Our main contribution in this work lies in appropriately modifying this framework to model the FlexRay protocol, which turns out to be a non-trivial task, as we show in this section.

The system architectures we are interested in consist of multiple ECUs communicating via a FlexRay bus. One or more applications are partitioned into tasks, which are then mapped onto different ECUs. ECUs running multiple tasks use a scheduler to share the available processing resources as shown in Figure 5.1. Each task is activated at a certain rate or is triggered by an output from another task. Once activated, it needs to be processed and hence consumes a fixed number of processor cycles from the ECU on which it is running.

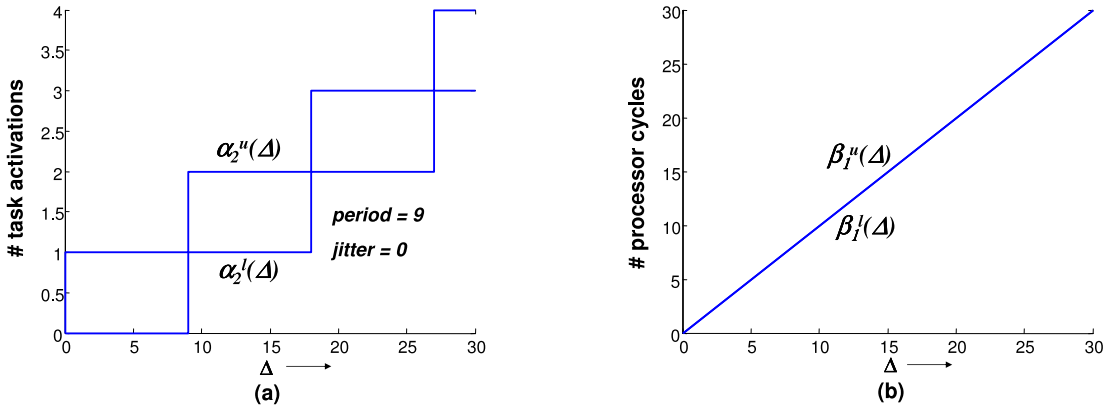


Figure 5.3: (a) α^u and α^l corresponding to a periodic activation. (b) β^u and β^l of an unloaded processor.

At the heart of the framework being discussed lies the modeling of (i) the triggering pattern of tasks (or the event model) which generates an execution demand on a ECU and communication demand on the bus, and (ii) the service offered by a ECU (or the bus) to each task running on it (i.e. the resource model).

Event Model:

The arrival rate of any event stream triggering a task is upper- and lower-bounded by two functions $\alpha^u(\Delta)$ and $\alpha^l(\Delta)$. Let $R(t)$ be the total number of events that arrive during the time interval $[0, t]$. Then $\alpha^l(\Delta) = \min_{t \geq 0}\{R(t + \Delta) - R(t)\}$ for any Δ. Similarly, $\alpha^u(\Delta) = \max_{t \geq 0}\{R(t + \Delta) - R(t)\}$. Hence, $\alpha^u(\Delta)$ and $\alpha^l(\Delta)$ denote the maximum and minimum number of events that might arrive within *any* interval of length Δ. The timing properties of standard event models — like *periodic, periodic with jitter* and *sporadic* — as well as more arbitrary arrival patterns can be represented by an appropriate choice of α^u and α^l. For example, a periodic event stream with period 9 can be represented by an upper and lower bound shown in Figure 5.3(a). It is also possible to determine the values of $\alpha^u(\Delta)$ and $\alpha^l(\Delta)$ corresponding to any given arbitrary event trace (from measurements or from simulation) and a real number Δ by sliding a window of length Δ over the

trace and recording the minimum and maximum number of events lying within the window respectively. The upper and the lower arrival curves corresponding to the trace can be determined by following this procedure for different values of Δ.

Resource Model:

Similarly, let $\beta^u(\Delta)$ and $\beta^l(\Delta)$ denote upper and lower bounds on the *service* available to a task. Let $S(t)$ be the number of activations of this task that were serviced during the time interval $[0, t]$. Then, $\beta^l(\Delta) = \min_{t \geq 0}\{S(t + \Delta) - S(t)\}$ for any Δ, and $\beta^u(\Delta) = \max_{t \geq 0}\{S(t + \Delta) - S(t)\}$. If there are multiple tasks running on an ECU, the service bounds β^u and β^l available to any task will clearly depend on the scheduling policy being used. Further, if $\beta^u(\Delta)$ and $\beta^l(\Delta)$ are expressed in terms of the maximum and minimum number of available *processor cycles*, then they can easily be converted to service expressed as — the number of task activations that can be serviced within any Δ. This is done by scaling $\beta^u(\Delta)$ and $\beta^l(\Delta)$ with the execution requirement incurred by the task due to each activation.

As an example, the upper and lower bounds on the service in the case of an unloaded ECU can represented as two straight lines that coincide with each other (see Figure 5.3(b)). The slope of these lines denotes the clock frequency of the ECU. Communication resources (e.g. buses) can be similarly modeled, the *service curves* in this case typically bound the number of transmittable bits within any given time interval. Such service curves can be derived from a formal model of the resource, or from data sheets, or in some cases by simple measurements.

System Composition and Analysis:

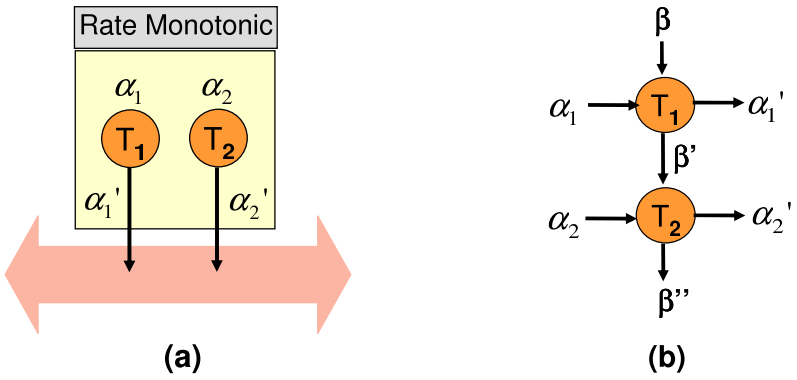


Figure 5.4: (a) Rate monotonic scheduling of two tasks. (b) Corresponding scheduling network.

An event stream entering a resource gets processed, thereby generating an outgoing stream of events/data which can activate other tasks on the same ECU, or might be transferred over the bus to trigger tasks running on other ECUs. Let $\alpha^{u\prime}(\Delta)$ and $\alpha^{l\prime}(\Delta)$ denote upper and lower bounds on the number of such events generated within *any* time interval of length Δ. It can be shown that (see [84]):

$$\alpha^{l\prime}(\Delta) = \min\{\inf_{0\leq\mu\leq\Delta}\{\sup_{\lambda>0}\{\alpha^{l}(\mu+\lambda)-\beta^{u}(\lambda)\}+\beta^{l}(\Delta-\mu)\},\beta^{l}(\Delta)\} \quad (5.1)$$

$$\alpha^{u\prime}(\Delta) = \min\{\sup_{\lambda>0}\{\inf_{0\leq\mu<\lambda+\Delta}\{\alpha^{u}(\mu)+\beta^{u}(\lambda+\Delta-\mu)\}-\beta^{l}(\lambda)\},\beta^{u}(\Delta)\} \quad (5.2)$$

Similarly, the bounds on the *remaining service* after processing the activations of a task are given by:

$$\beta^{l\prime}(\Delta) = \sup_{0\leq\lambda\leq\Delta}\{\beta^{l}(\lambda)-\alpha^{u}(\lambda)\} \quad (5.3)$$

$$\beta^{u\prime}(\Delta) = \max\{\inf_{\lambda>\Delta}\{\beta^{u}(\lambda)-\alpha^{l}(\lambda)\},0\} \quad (5.4)$$

Given α^{u}, α^{l} and β^{u}, β^{l}, it is also possible to compute the maximum *delay* experienced by a task before its activation is serviced and the maximum number of

backlogged activations with the following equations:

$$\text{delay} \leq \sup_{t \geq 0}\{\inf_{\tau \geq 0}\{\alpha^u(t) \leq \beta^l(t+\tau)\}\} \tag{5.5}$$

$$\text{backlog} \leq \sup_{t \geq 0}\{\alpha^u(t) - \beta^l(t)\} \tag{5.6}$$

With the help of an example, we now show how a system architecture may be modeled using the above results. Consider the setup shown in Figure 5.4(a). It consists of two tasks T_1 and T_2 which are being scheduled using a rate monotonic scheduler. Both T_1 and T_2 are activated periodically, with T_1's period being 4 time units and T_2's period being 9 time units. Each activation of T_1 and T_2 requires 1 and 2 processor cycles respectively to process. The upper and lower bounds on the activation of T_2 (i.e. α_2^u and α_2^l) were shown in Figure 5.3(a). They are similar for T_1, except for the difference in the length of the period. The upper and lower bounds on the service offered by the unloaded ECU (in terms of the number of processor cycles available over any time interval) were shown in Figure 5.3(b). Since T_1 has a smaller activation period, it has a higher priority (because of rate monotonic scheduling) and hence the full service offered by the unloaded ECU is available to it.

As discussed above, using α_1^u, α_1^l and β_1^u, β_1^l, we can compute $\beta_1^{u\prime}$ and $\beta_1^{l\prime}$, which are bounds on the *remaining* service (that is left over after processing T_1). This remaining service is now available to the lower-priority task (i.e. T_2). This concept is illustrated in the form of a *scheduling network* for a rate monotonic (or any fixed priority) scheduler in Figure 5.4(b).

$\beta_1{}'$ is used for servicing task T_2 (see Figure 5.5(a)), which along with α_2 can be used to compute upper and lower bounds on the events generated by each serviced activation of T_2 (β and α often refer to the tuples β^u, β^l and α^u, α^l). These bounds

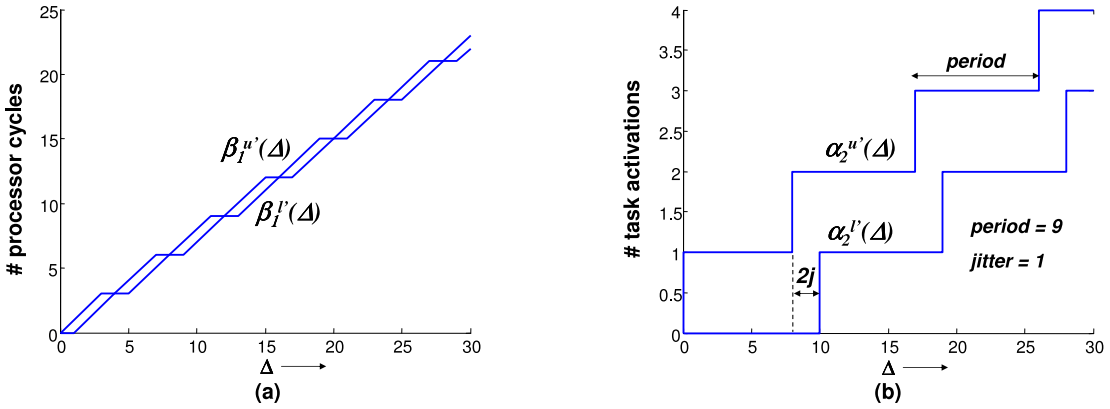


Figure 5.5: (a) Bounds on the *remaining* service after processing task T_1. (b) Bounds on the messages generated by T_2.

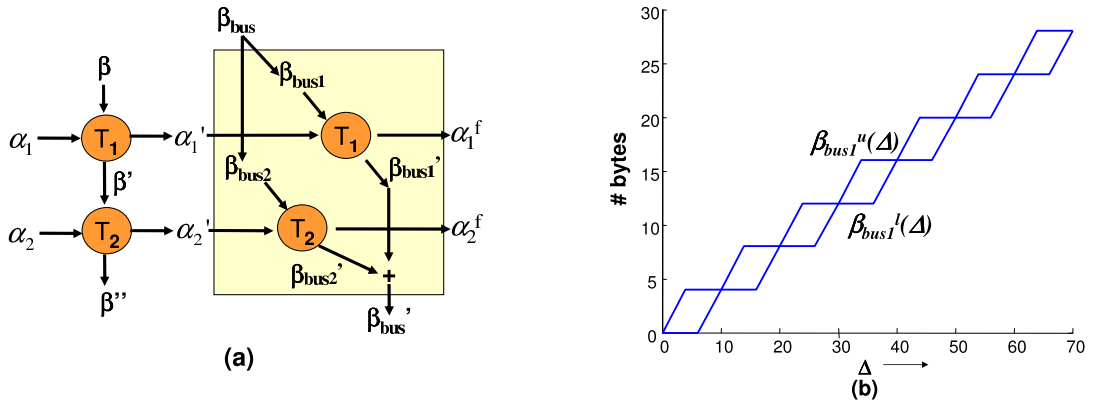


Figure 5.6: (a) Performance model of the complete architecture (b) The bounds on the service available on the *TDMA* bus to messages from T_1.

are shown in Figure 5.5(b). From this figure, note that this event stream is periodic with a period of 9 time units and a jitter of 1 time unit. It is straightforward to see that the distance between $\alpha_2^{u\prime}$, $\alpha_2^{l\prime}$ is equal to twice the jitter of the event stream.

So far we described how to use this framework to analyze a ECU, but the same technique is also applicable to communication resources (e.g. buses). To illustrate this, we now model the complete architecture (along with the communication bus) shown in Figure 5.4(a). Assume that the bus transmits the processed streams $\alpha_1{}'$ and $\alpha_2{}'$ as messages to another ECU (which is not shown in this architecture). The performance model of the complete architecture including the bus is now shown

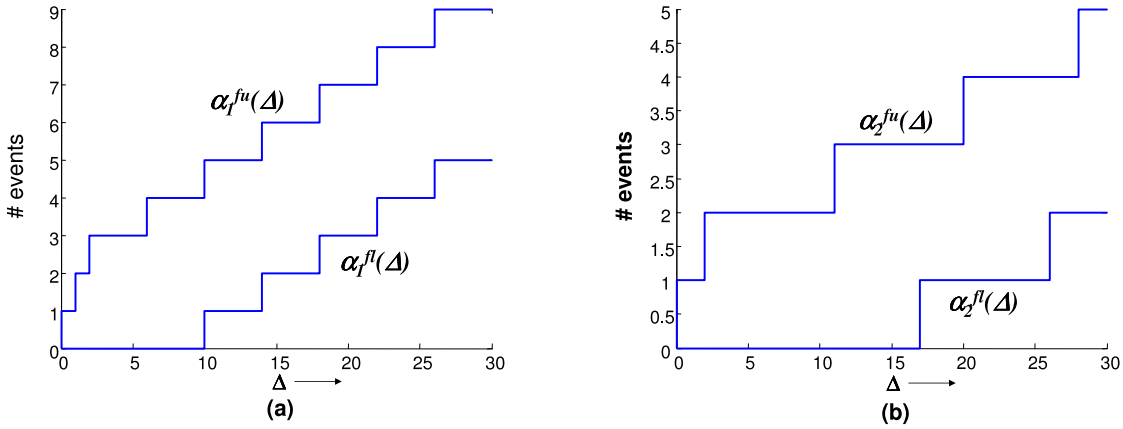


Figure 5.7: (a) Upper and lower bounds on the transmitted messages over the bus arising from T_1. (b) Bounds on the transmitted messages from T_2.

in Figure 5.6(a). Suppose that each serviced activation of T_1 and T_2 generates a message of size 1 byte that is to be transmitted over the bus. The $TDMA$ scheduler running on the bus has a cycle length of 10 time units and provides slot sizes that are suitable for transmitting 4 and 3 bytes of data from T_1 and T_2 respectively during every cycle. The service curves corresponding to this bus availability to T_1 is shown in Figure 5.6(b). Finally, Figure 5.7 shows the timing properties (or bounds on the arrival rate) of the transmitted messages from T_1 and T_2. From the timing properties of the message stream injected by T_2 on the bus (Figure 5.5(b)) and the timing properties of these transmitted messages (Figure 5.7(b)), it may be noted that the jitter increases from 1 to 7.5 time units. These transmitted messages can now trigger tasks running on other ECUs and the same procedure may be applied to analyze them as well.

5.2.1 Difficulties in Modeling FlexRay

Recall from Section 5.1 that in the DYN segment of FlexRay, tasks are given access to the bus in decreasing order of their priorities. In other words, the task with the highest priority is offered access to the bus at the start of the DYN segment.

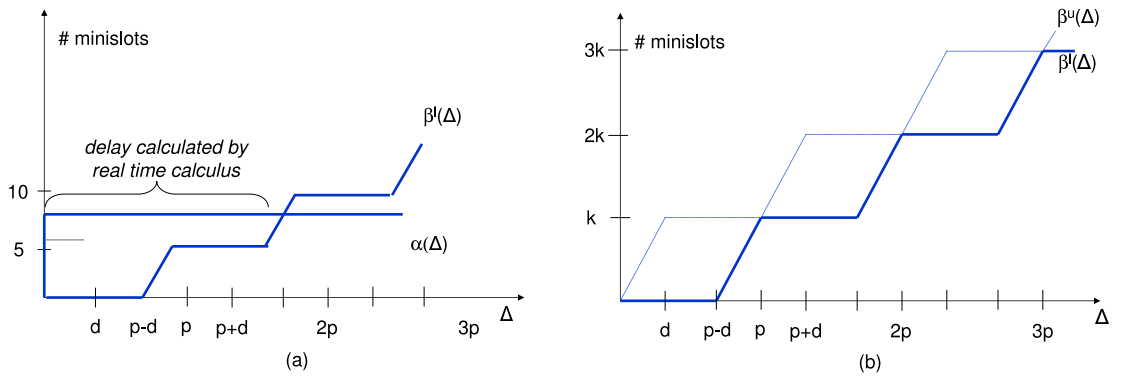


Figure 5.8: (a) Computing maximum delay from α^u and β^l. (b) Total service offered by the DYN segment.

Further, once given access to the bus, a task can occupy it till the end of the current DYN segment. Hence, the most straightforward approach would be to model this protocol as a fixed priority scheduler, as shown in Figure 5.4(b). Here, β would be used to model the total service offered by the DYN segment and successive β's would be computed from the message sizes and message generation rates of the different tasks. However, this approach does not work because of the following properties of FlexRay: (i)A task can send at most one message in each DYN segment (where the maximum length of the message can be equal to the length of the DYN segment). (ii) One minislot is consumed from the available service each time a task is not ready to transfer a message, before the next lower priority task is allowed to send its message on the bus. (iii) If a DYN message is generated by its sender task after the slot has started, the message to wait until the next bus cycle starts in order to contend for the bus. (iv) A task is only allowed to send a message if it fits into the remaining portion of the DYN segment, i.e. a message cannot straddle two communication cycles.

The modeling framework presented above does not incorporate these restrictions when representing the service availability of a resource using the upper and lower bounds $\beta^u(\Delta)$ and $\beta^l(\Delta)$. To see this, consider Figure 5.8(a), which shows α^u corresponding to the arrival of a single message (of length equal to 8 minislots)

that is to be transmitted over the DYN segment (of length 5 minislots). Here, the length of each communication cycle (or period) is assumed to be p time units and the length of the DYN segment is equal to d time units. The lower bound on the service β^l corresponding to the DYN segment is also shown in this figure. Note that over time intervals Δ of length less than or equal to $p - d$, no service might be available from the DYN segment due to the blocking by the ST segment.

Since the length of the message in this case is longer than the length of the DYN segment, this message will never get transmitted. However, the framework we described above models the message to be transmitted over two communication cycles, thereby incurring a delay equal to the maximum horizontal distance between α^u and β^l (see Figure 5.8(a)). In the next Section, we will see how our framework models all the FlexRay properties and thus, correctly analyzes scenario like this.

5.3 Illustrative Examples

In this section, we shall illustrate the working of our scheme with the help of small examples of FlexRay based networks. This will be followed by a more formal description in the next section.

Example 1

For the first example, consider a task T_1 transmitting a single message, m_1 of 2 bytes over the FlexRay DYN segment every 10 milli-seconds (ms). This set-up is shown in Figure 5.9(a). We are considering a FlexRay cycle length of 10 ms and a DYN segment length of 5 ms. Assume that m_1 is the highest priority message. Hence, its transmission has to begin in the first minislot in the DYN segment. For

simplicity of exposition, we assume that number of minislots per milli-second is 1, and that 1 byte is transmitted in 1 minislot. Thus, once m_1 gets access to the bus, it requires 2 minislots to be transmitted completely which turns out to be a time interval of two milli-seconds.

Before describing our framework to evaluate the worst case delay of m_1 over the FlexRay bus, we perform an analysis by hand. Figure 5.9(b) graphically shows this analysis. The worst case scenario occurs when the message is ready just after its minislot starts. Thus, the message m_1 has to wait the one FlexRay cycle (DYN+ST=10ms) for its next turn, and then it gets transmitted over the next 2ms. Thus, the worst case delay is 12 ms.

Lets us now apply our framework and evaluate the delay of m_1 over the FlexRay bus. To compute the delay for m_1 using our scheme, we require arrival curve α_1, and the service curve β_1 for m_1 (see Equation 5.5). Since the period of 10ms is known for m_1, α_1 may be readily constructed (see Section 5.2). However, computing β_1 — the service that is available to message m_1 is not straightforward because of the FlexRay properties discussed in the previous section. In the following, we will illustrate how to construct β_1 in a step-by-step fashion such the FlexRay properties are correctly incorporated.

Step 1: We have seen that the total service available for the entire DYN segment can be modeled as β (shown in Figure 5.9(c)). Here the sloped curve segments represent the service available for each DYN segment. In our first step, we extract 2 minislots of service during each communication cycle from β. This models the property 1 which implies that during any communication cycle at most 2 minislots are available to T_1 (since a task can send at most one message in each cycle). Figure 5.9(c) shows step 1.

Step 2: In our second step, we subtract one minislot from each DYN segment to

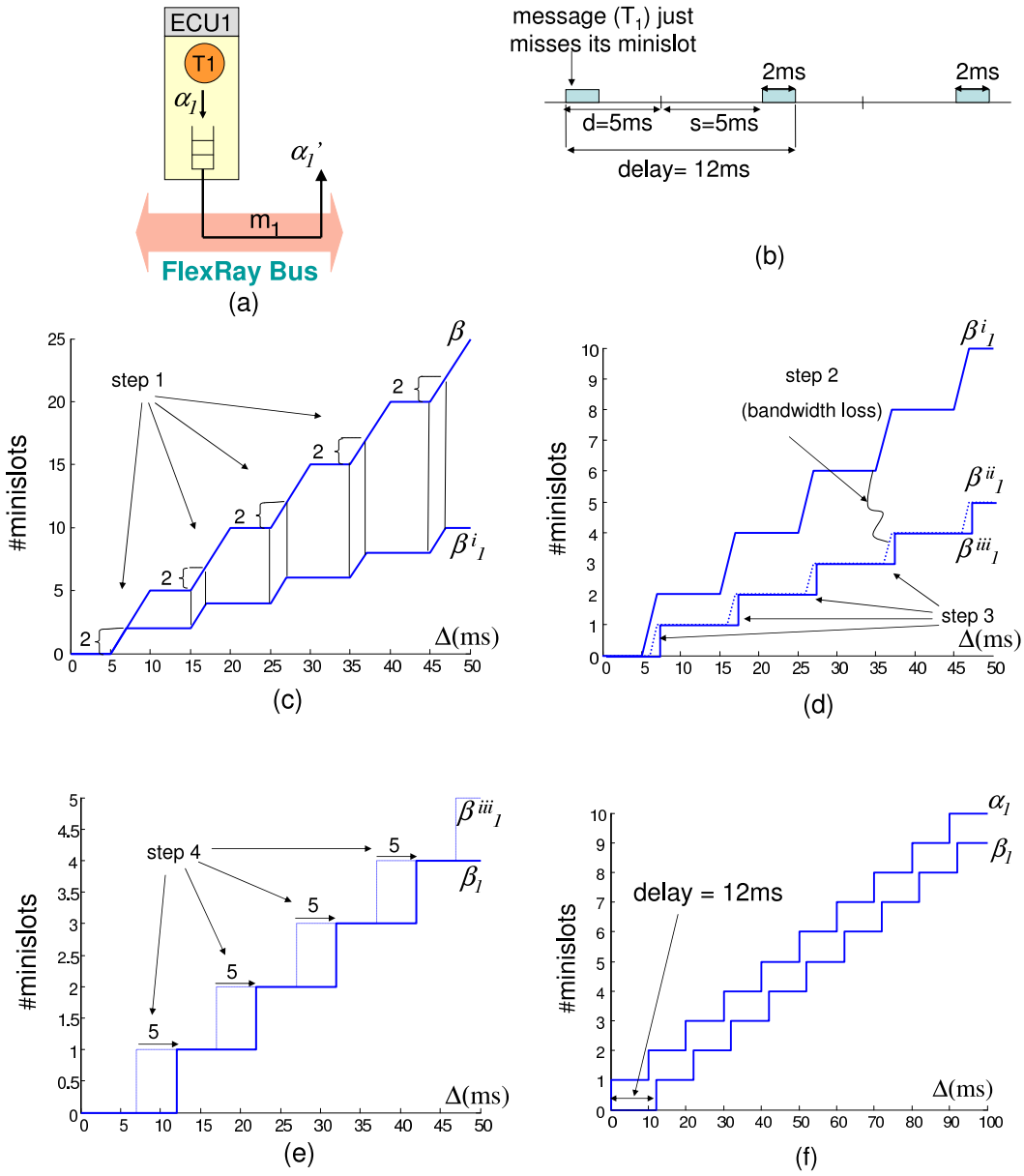


Figure 5.9: Example 1 (a) Architecture. (b) Analyzing actual delay of m_1. (c) Step 1. (d) Steps 2 and 3. (e) Step 4. (f) Delay of m_1 computed by our framework.

model property 2 of bandwidth loss (Section 5.2.1). The resulting service curve is shown in Figure 5.9(d). This now ensures a bandwidth loss of one minislot in each cycle where the service is not consumed. However, this would also lead to pessimistic results because we have decreased available service even in cycles that would be consumed. To avoid this, we adjust message size of $m1$ by subtracting 1 minislot, i.e $2 - 1 = 1$ minislots in the subsequent analysis of service consumption for messages transmitted by task T_1. Thus, we have ensured bandwidth loss as well as consistency in computation of the delay.

Step 3: Property 3 mentions that a message must start at the beginning of the communication slot. If a message is ready just after its minislot has started the message has to wait for the next cycle. Thus, in each cycle either the entire service is available or it is not available at all. To reflect this, in step 3, we discretized the service bound obtained from Step 2, i.e. convert it into a step-function. This is shown in Figure 5.9(d).

Step 4: Property 4 says a message cannot straddle two communication cycles. We observe from Figure 5.9(b) that any interval Δ of length less than 12ms can be positioned to straddle two communication cycles. Hence,the minimum service available from the DYN segment over intervals of length less than 12ms should be equal to 0. However, from the Figure 5.9(d) we can observe that the resulting service guarantees service availability within a time interval 7ms (length of ST + actual transmission time of $m1$). To achieve property 4, the service bound resulting from step 3 is shifted by 5 (the length of DYN segment) time units. Step 4 is reflected in Figure 5.9(e).

Finally, using β_1 and the arrival curve α_1 for m_1, the delay 12ms is correctly computed by our framework (see Figure 5.9(f)).

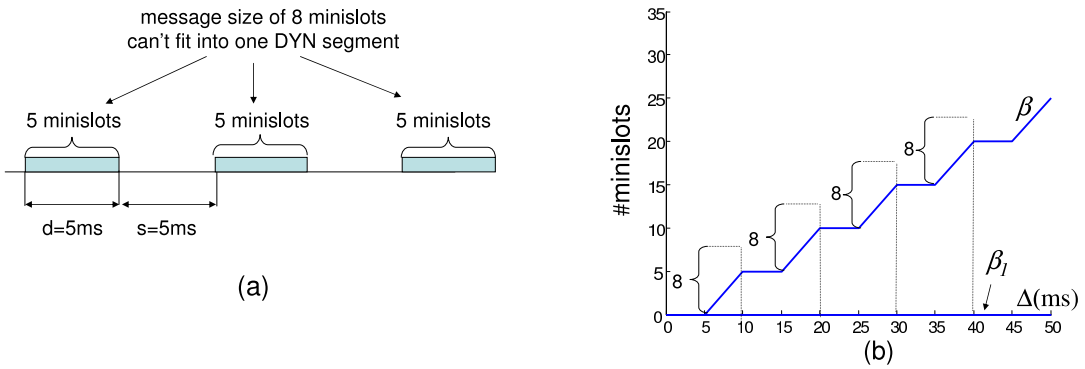


Figure 5.10: Example 2 (a) Message does not fit into one DYN segment. (b) Step 1 results in nullified β_1.

Example 2

Let us consider a second example with similar architecture as Example 1 but where the message size is 8 bytes i.e. 8 minislots. Recall that this is essentially the same scenario we described in the Section 5.2.1. Figure 5.10(a) shows that the message would never be transmitted because it does not fit into a single DYN segment. Figure 5.10(b) shows the step 1 for construction of the service curve β_1 of this message. We need to extract 8 minislots from each segment, which is greater than available resource. In this case, our framework nullifies service available in such communication cycles. The resulting service curve as shown in Figure 5.10(b) correctly reflects that no service is available in any time interval. Now when we compute delay for m_1, our framework would return a infinite delay, as we had analyzed in Section 5.2.1.

Example 3

Our third example is slightly more involved with two messages being transmitted over the FlexRay bus as shown in Figure 5.11(a). This message is transmitted over the DYN segment with priority 2, and the size of the message is 2 bytes. The rest of the architecture is same as Example 1.

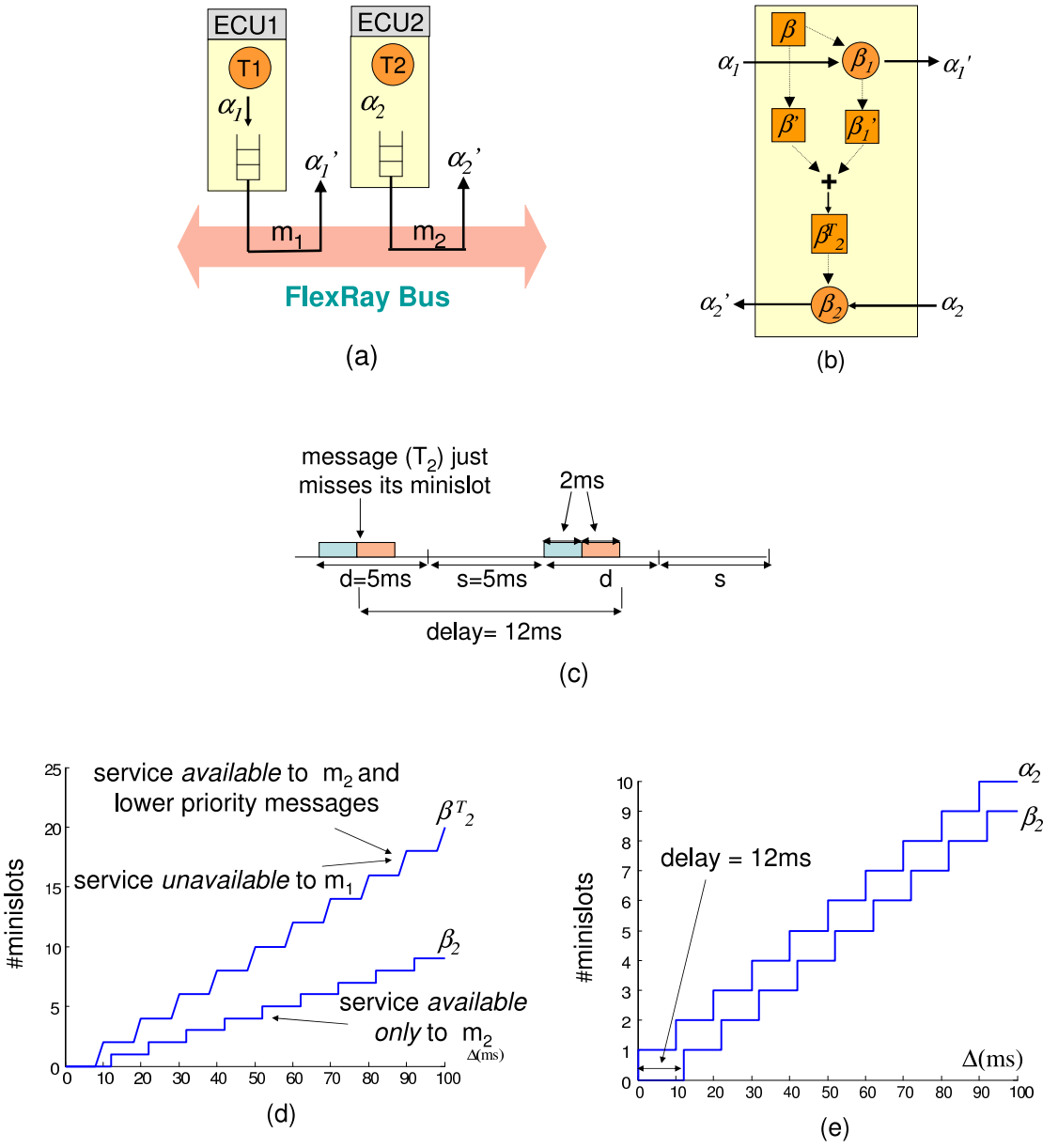


Figure 5.11: Example 3 (a) Architecture. (b) Overview of our scheme. (c) Analyzing actual delay of m_2. (d) Transformation. (e) Delay of m_2 computed by our framework.

Before describing our framework, we analyze the worst case delay for m_2 over the FlexRay bus (Figure 5.11(c)) . As in example 1, the worst case scenario occurs when the message is ready just after its minislot starts. Thus, the message m_2 has to wait the one cycle (DYN+ST=10ms) for its next turn, and then gets transmitted over the next 2ms. Thus, the worst case delay for m_2 in this case is 12ms.

The architecture and parameters are similar to Example 1 for message m_1. Thus,

all the analysis described in Example 1 hold here as well. We are now interested in the analysis of m_2. In Example 1, we have seen how to obtain β_1 from the total service β, the service available to messages m_1 and all lower priority message. Similarly, once we find β_2^T — the total service available to m_2 and lower priority messages — we can apply the same technique to find β_2. In the following, we explain how to compute β_2^T.

The service available to the lower priority tasks, β_2^T (i.e. $T_2, \ldots, T_n$) is made up of two components (i) service that was *unavailable* to T_1, and (ii) service that was *unutilized* by T_1. In the following we describe these components:

(i) In Example 1 we extracted β_1 from β, but the rest of the service i.e. service that was *unavailable* to T_1 will be available to lower to priority messages. This component is given by the following equation

$$\bar{\beta}^l(\Delta) = \sup_{0 \leq \lambda \leq \Delta} \{\beta^l(\lambda) - \beta_1^i(\lambda)\} \tag{5.7}$$

(ii) The service that was *unutilized* by T_1 will also be available to lower priority messages and is denoted by $\beta_1^{l\prime}$. Note that in this example however, m_1 is triggered every 10ms which means one instance of m_1 is ready every FlexRay cycle, and hence the service is consumed each cycle. Thus, entire service is utilized and *unutilized* is effectively zero. This can be verified using Equation 5.3.

Thus, β_2^T, which represents the service available to the lower priority tasks is equal to service that was *unavailable* to T_1. This curve is shown in Figure 5.11(d), which is then transformed in the same way as in Example 1, but using information specific to messages from task T_2 in order to obtain β_2. Figure 5.11(e) shows the delay 12ms computed by our framework using β_2.

Figure 5.11(b) now shows an complete overview of our scheme. Here, α_1 bounds the arrival rate of m_1 at the bus and β is the service offered by the unloaded bus. β_1 is the service available to m_1 which has already been analyzed in Example 1. β' is the service remaining from β (i.e. *unavailable* to m_1). ${\beta_1}'$ is the service that is *unutilized* by m_1 (from what was available to it). The sum of β' and ${\beta_1}'$ is the service available to m_2 and lower priority messages gives us by β_2^T. Finally, the triggering rate of T_2 (which is equal to the arrival rate of m_2 at the bus) is bounded by α_2.

Example 4

In our final example, we consider the an architecture similar to Example 3, but with different parameters for the messages. Assume that m_1 has a size 5 byte, while m_2 is a message with size 4 bytes and both are triggered every 20ms.

Figure 5.12(a) shows the worst case delay for m_2 over the FlexRay bus. We have already seen that the worst case scenario occurs when the message is ready just after its minislot starts. In contrast to Example 2, the message m_2 is now blocked not for one cycle but two cycles (2×(DYN+ST=10ms)=20ms) for its next turn. This is because m_1 may occupy the entire DYN segment in the next cycle. Once m_2 accesses the bus, it gets transmitted over the next 2ms. Thus, the worst case delay for m_2 in this case is 22ms.

Following the discussion for the previous example we need to compute the total service (β_2^T) which is equal to the service (i) *unutilized* by m_1 and (ii) the service *unavailable* to m_1. In this example, the service *unavailable* to the message m_1 is zero over all time intervals. This is because m_1 size is 5 minislots and thus, entire DYN segment is available to the message. This may also be verified from

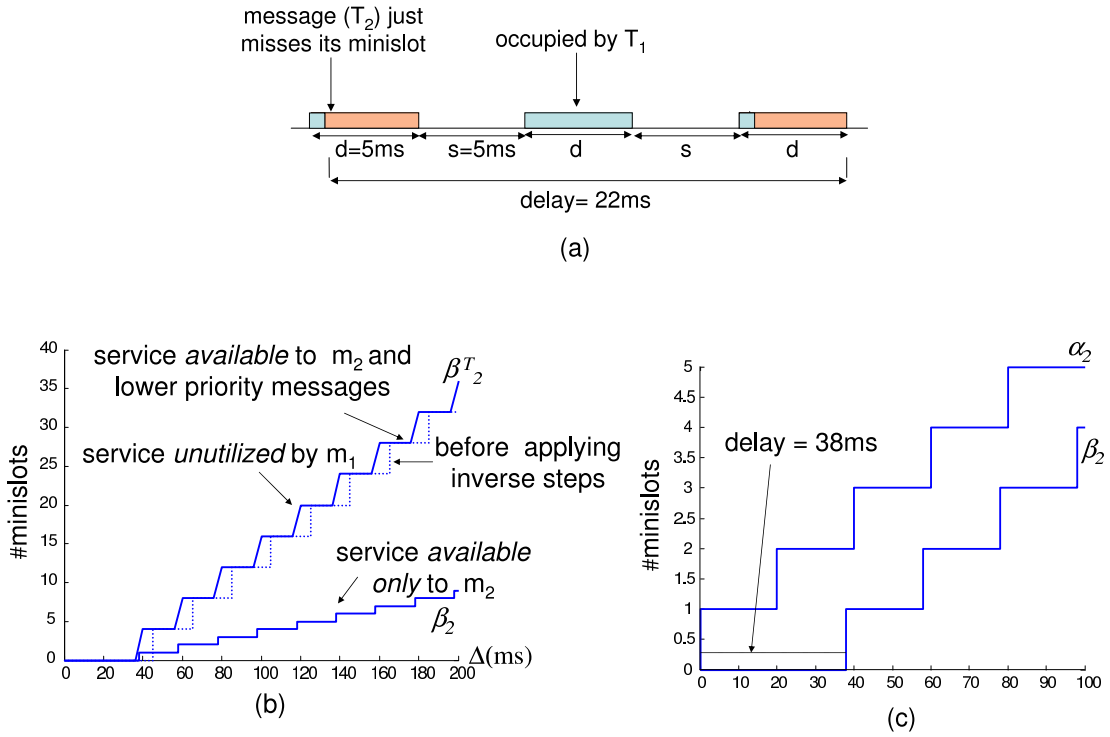


Figure 5.12: Example 4 (a) Analyzing actual delay of m_2. (b) Transformation. (c) Delay of m_2 computed by our framework.

Equation 5.7. Thus, the total service (β_2^T) is equal to the *unutilized* service. The *unutilized* service, β_1', by m_1 is obtained from Equation 5.3. However, recall that this is specific to messages from task T_1 because this incorporates message size dependent adjustments such that the respective service can be consumed just according to the FlexRay restrictions. So it first needs to be transformed by applying the "inverse" of Steps 2 and 3 that were applied to β^l. Figure 5.12(b) shows the relevant curves obtained .

Figure 5.12(e) shows the delay 38ms computed by our framework using β_2. Note the results are pessimistic because the Equation 5.3 returns bounds, which are not tight [16].

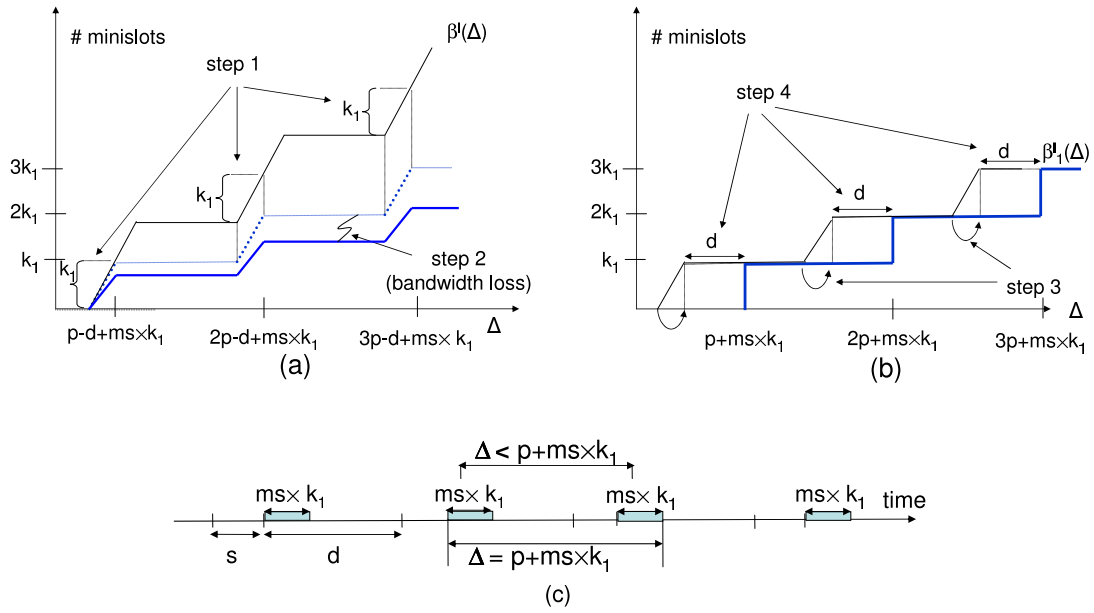


Figure 5.13: (a) Steps 1 and 2 for transforming β^l. (b) Shifting the resulting service bound. (c) Blocking time.

5.4 Modeling FlexRay

Having described how to apply our framework for a series of small examples, we now provide a formal description. FlexRay – as described in Section 5.2.1 – restricts the amount of available service that can actually be used. Hence, while the service bounds $\beta^u(\Delta)$ and $\beta^l(\Delta)$ capture the limits on the total service available to the DYN segment, we need to model how much of this service can actually be used.

Towards this, assume that tasks $T_1, \ldots, T_n$ send messages over the DYN segment with any message from task T_i being denoted by m_i and has a length of k_i minislots. The length of the DYN segment is assumed to be equal to k minislots (or d time units) and the length of a communication cycle, as before, is equal to p time units. Each minislot is assumed to be `MS` time units long.

Let $\beta^l(\Delta)$ be the lower bound on the service (expressed in terms of number of minislots) offered by the unloaded DYN segment to all the tasks. Further, let β_i^l be the service offered by the DYN segment to task T_i. To obtain β_1^l, the function

β^l needs to be algorithmically transformed. As one shall observe, our algorithm essentially transforms each segment in the curve. It can be easily verified that each "increasing" segment of the service curve β^l corresponds to an additional DYN segment (which is guaranteed to be available within the corresponding time interval). The transformations applied to each slope of the service curve capture the minimum guaranteed service that is available *only* to message m_1 during each DYN segment, and thus, produce the curve β_1.

The algorithm to obtain β_1^l from β^l consists of the following steps:

1. Extract k_1 minislots of service during each communication cycle from β^l. This is because during any communication cycle at most k_1 minislots are available to T_1 (since a task can send at most one message - property 1). Nullify the communication cycles containing less then k_1 minislots.

2. A minislot is lost even when a task does not transmit any message (property 2). This is accounted by subtracting one minislot from each communication cycle corroborated with an adjusted message size of $k_1 - 1$ minislots in the subsequent analysis of service consumption for messages transmitted by task T_1. Steps 1 and 2 are shown in Figure 5.13(a).

3. Discretized the service bound obtained from Step 1, i.e. convert it into a step-function. It reflects the property that a message must start at the beginning of the communication slot. If a task just misses its turn in the DYN segment, it has to wait for the next communication cycle (see Figure 5.13(b)).

4. The resulting service bound is shifted by d time units. This is to model that a message has to be completely sent within a single DYN segment (property 4). Note from Figure 5.13(c) that any interval Δ of length less than $p+\texttt{MS}\times k_1$ can be positioned to straddle two communication cycles. Hence, the minimum

service available from the DYN segment over intervals of such length is equal to 0. The shifted service bound in Figure 5.13(b) reflects this.

The resulting service bound, which we denote as β_1^l correctly represents the minimum or guaranteed service from the DYN segment that is available to messages from T_1. This β_1^l can now be plugged into the framework outlined in Section 5.2 to compute the maximum delay suffered by any m_1, the maximum number of backlogged m_1s and the timing properties of the transmitted messages (which might trigger other tasks). Towards this $\alpha_1^u(\Delta)$ is used as an upper bound on the number of messages generated by T_1 within any interval of length Δ.

The service available to the lower priority tasks, β_2^T (i.e. $T_2, \ldots, T_n$) is made up of two components (i) service that was *unavailable* to T_1, and (ii) service that was *unutilized* by T_1. The following steps describe the computation of these components and their addition to obtain β_2^T:

1. The remaining service left after performing transformation 1 (i.e. the service that was *unavailable* to T_1) is given by the following equation

$$\bar{\beta}^l(\Delta) = \sup_{0 \leq \lambda \leq \Delta} \{\beta^l(\lambda) - \beta_1^l(\lambda)\} \tag{5.8}$$

2. The service that was *unutilized* by T_1 can be computed from β_1^l and α_1^u using Equation 5.3and is denoted by $\beta_1^{l\prime}$.

3. However, $\beta_1^{l\prime}$ cannot be directly added to $\bar{\beta}^l$ because it is specific to messages from task T_1 (i.e. incorporates message size dependent adjustments such that the respective service can be consumed just according to the FlexRay restrictions). So it first needs to be transformed by applying the "inverse" of Steps 2 and 3 that were applied to β^l, and the resulting function is added to $\bar{\beta}^l$.

Thus, β_2^T, which represents the service available to the lower priority tasks is computed. This is then transformed in the same way as β^l, but using information specific to messages from task T_2. This procedure is then repeated for all the tasks $T_3, \ldots, T_n$.

5.5 Adaptive Cruise Control Application: A Case Study

We will now show the utility of the framework discussed in Section 5.2 in modeling an adaptive cruise control (ACC) application. This is followed by an illustration of how this model can be used for formal performance analysis and debugging of an architecture consisting of multiple heterogeneous ECUs communicating via a FlexRay bus. When compared to simulation-oriented approaches—which can be time consuming and do not provide any formal guarantees—our framework can be used to quickly evaluate multiple design choices to determine whether they meet the performance constraints at hand. The main challenge here is to determine end-to-end timing properties of event/data streams which pass through multiple ECUs (implementing different scheduling policies) and the FlexRay bus. Each of these processing/communication elements modify the timing properties of the stream as it passes through it.

System Description:

As shown in Figure 5.14, the ACC subsystem consists of five ECUs communicating via a FlexRay bus. The bus has a communication cycle of 16 ms. The length of the DYN segment is of 10 ms and consists of 140 minislots, and the length of ST segment is 6 ms. Each minislot in the DYN segment can accommodate 4 bytes

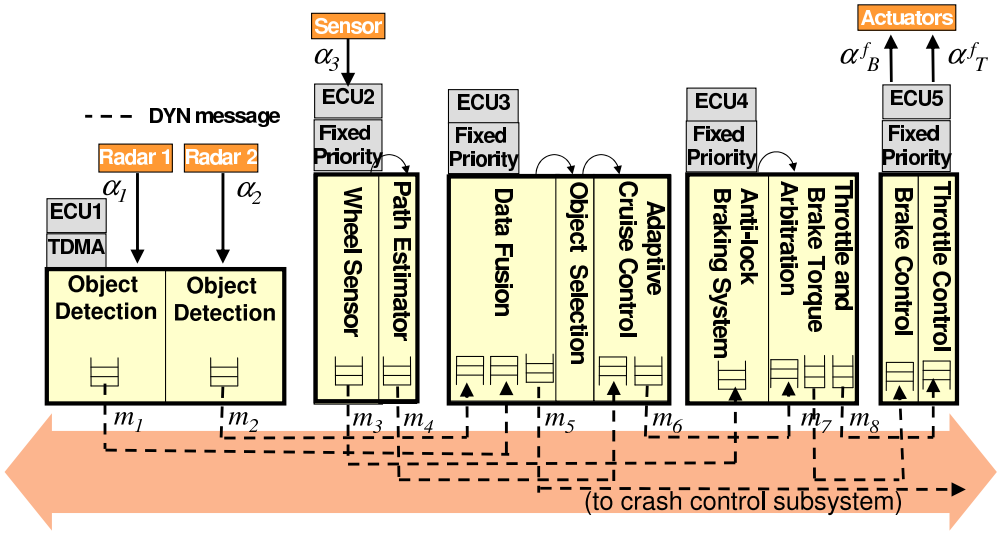


Figure 5.14: The system architecture of an Adaptive Cruise Control subsystem.

of data. $ECU1$ receives data from two radar sensors periodically every 60 ms, and $ECU2$ periodically receives data from a wheel sensor every 250 ms. Note that according to the FlexRay protocol specification [33], the communication cycle length may be upto 16 ms. Thus, the parameters that we have chosen here for our experiments conform to the FlexRay standard and may actually occur in real world design scenario.

The data received by $ECU1$ from each radar is processed by an *Object Detection* task. The processed data streams m_1 and m_2 are sent over the FlexRay bus to $ECU3$ to be processed by the *Data Fusion*, *Object Selection* and *Adaptive Cruise Control* tasks. The periodic data received by $ECU2$ from each radar is processed by the task *Wheel Sensor*. The data processed by this task is sent over the bus as the message stream m_3, which triggers the task *Anti-lock Braking System* at $ECU4$. The task *Adaptive Cruise Control* running at $ECU3$ also receives a message, m_4 from the task Path Estimator running on $ECU2$. The resulting data stream from $ECU3$, m_6, is transmitted over the the bus to $ECU4$ which runs the *Throttle and Brake Arbitration* task. The output from the *Throttle and Brake Arbitration* task is fed into the *Brake Control* and *Throttle Control* tasks ($ECU5$) via the messages

Bus		**ECUs**	
Message	*# Bytes*	*Task*	*WCET*
m_1	128	Data Fusion	10 ms
m_2	128	Object Selection	1 ms
m_3	64	Adaptive Cruise Control	4 ms
m_4	64	Arbitration	5 ms
m_5	128	Path Estimation	10 ms
m_6	64	Brake Control	2 ms
m_7	32	Throttle Control	2 ms
m_8	32	Anti-Lock Braking System	8 ms
		Wheel Sensor	4 ms
		Object detection	4 ms

Table 5.1: The workload on the bus and the ECUs for the ACC subsystem.

m_7 and m_8, which in turn send their outputs to two different actuators. These final output control signals are bounded by the functions α_B^f and α_T^f respectively. Finally, $ECU3$ also transmits a message stream m_4 to a *Crash Control* subsystem via the DYN segment of the bus.

In Figure 5.14, the dashed lines represent messages transmitted via the DYN segment of the FlexRay bus (m_1 has the highest priority, followed by m_2 and so on). The arrows between tasks in $ECU2$, $ECU3$, and $ECU4$ represent data dependencies (i.e. data from the incoming arrow flows into the task pointed to by the arrow). It may be noted that $ECU1$ uses a TDMA policy to schedule the tasks running on it, and the rest use a fixed-priority scheduler. Finally, Table 5.1 shows the lengths of the different messages and the execution times of the various tasks running on the different ECUs.

Design Space Exploration:

For the ACC subsystem described above, we computed performance metrics like end-to-end delays (radar to actuators), delays experienced by individual message streams, and buffer requirements at the ECUs. We show how to use our framework

to explore the optimal set of design parameters for such performance metrics.

In this work, we have used the Real Time Calculus (RTC) Toolbox [85] to perform the necessary calculations for the performance analysis of the ACC model. The RTC Toolbox is a toolbox within Matlab for system-level performance analysis of distributed real-time and embedded systems. A Java kernel carries out the computations on the curves based on the real-time calculus (see equations in Section 5.2) while a set of Matlab libraries connect the kernel to the Matlab command line. Thus, in essence, the toolbox provides us with a library of Matlab functions for compositional performance analysis. However, this helps to model only the basic framework described in Section 5.2 and is not adequate to implement a FlexRay based performance model. Therefore, we implemented the FlexRay model described in Section 5.4 using a combination of Java and Matlab, and followed a similar software architecture. This was then plugged into the existing RTC toolbox, thus creating a single unified framework for realizing our performance analysis models. Thus, our implementation framework can now model basic scheduling policies like fixed priority and TDMA, as outlined in Section 5.2 as well the FlexRay scheduling policy.

We then implemented the performance model of ACC system in our framework. Figure 5.15(a) shows the lower bounds on the resource availability for the DYN segment of the FlexRay bus. In this figure, β denotes the lower bound on the availability of the *unloaded* DYN segment of the bus. Similarly, β^f denotes the lower bound on the *remaining capacity* of this segment after accommodating all the message streams that have been mapped onto it. β'_{m1} and β'_{m2} denote lower bounds on the availability of the DYN segment *after* accommodating the message streams m_1 and m_2.

Figure 5.15(b) shows the lower bounds on the arrival rates of the data from the two radars and the wheel sensor. Since these data streams are periodic, the upper

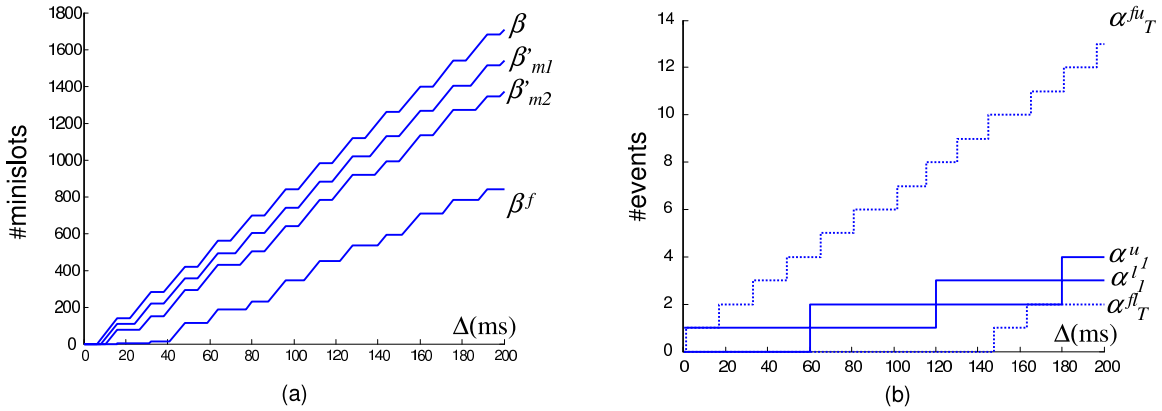


Figure 5.15: (a) The bounds on the resource curves for the DYN segment. (b) The bounds on the input and the output signals for the system.

bounds would be similar. This figure also shows the upper (α_T^{fu}) and lower (α_T^{fl}) bounds on the final output stream that feed into the throttle actuator. As explained in Section 5.2, from these bounds it is possible to compute the maximum jitter of this stream.

The computed end-to-end delay along the path from *Object Detection* to *Data Fusion* (via the FlexRay bus) to the crash control subsystem is equal to 109.86 ms. This delay includes the waiting time of a message at the two ECUs ($ECU1$ and $ECU2$), as well as the delay experienced in the bus. On the other hand, the end-to-end delay from the radars (Radar1) to the brake actuator is equal to 299 ms. The delay and buffer requirements of all the different message streams are listed in Table 5.2. These buffer sizes refer to the input buffers in which messages are stored while they wait to access the FlexRay bus.

Given a set of performance constraints, this framework can now be used to quickly evaluate whether a given design meets specified constraints. Further, it can also be used to evaluate delay and buffer requirements of individual message streams and ECUs, which can provide insights into performance bottlenecks and potential hotspots in an architecture. Such insights can also help in appropriate resource

	DYN Segment	
Message	*Delay*	*Buffer*
m_1	28.29 ms	128 Bytes
m_2	28.57 ms	128 Bytes
m_3	26.86 ms	128 Bytes
m_4	25.71 ms	64 Bytes
m_5	26.86 ms	128 Bytes
m_6	74.64 ms	512 Bytes
m_7	96.64 ms	160 Bytes
m_8	113.86 ms	224 Bytes

Table 5.2: Delay and buffer requirement of each message stream on the FlexRay bus.

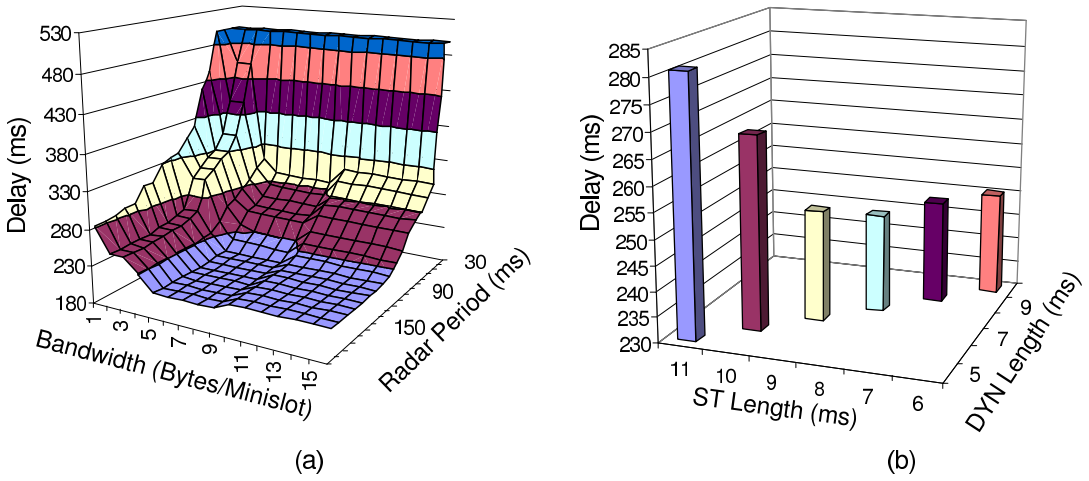


Figure 5.16: Design Space Exploration: (a) Influence of sampling rates and bandwidth on the end-to-end delay. (b) Influence of lengths of the static and dynamic segments on the end-to-end delay.

dimensioning. Finally, this framework can also help in determining appropriate combinations of scheduling parameters and activation rates of the different tasks for optimal performance under specified resource constraints. The design of all modern embedded systems involve determining the values of many system parameters, which influence each other in complex ways. As a result, their impact on various performance metrics is not immediately clear.

In what follows, we illustrate how our framework can be used to evaluate the impact of various parameters on the end-to-end delay from the radar to an actuator in

the ACC subsystem. Two such parameters that directly affect the delay are: (i) the bandwidth of the FlexRay bus, and (ii) the data arrival rates from the radars. Figure 5.16 (a) shows how the end-to-end delay varies for varying bandwidth of the FlexRay bus and the different sampling rates (or periods) of the radar. As the figure shows, the number of bytes per minislot is used as the measure for bandwidth. It can be seen that a larger period of the sensor leads to smaller end-to-end delays, albeit at the cost of some information loss due to the lower sampling rate. Similarly a larger bandwidth of the FlexRay bus in leads to smaller delays.

So far, we have considered a system architecture where all the messages have been mapped to the DYN segment. In reality however, the designer has the choice of mapping certain messages to ST segment and others to the DYN segment. In such a design scenario, deciding reasonable lengths for both the ST and the DYN segment is a tedious task because this choice has a direct impact on the performance metrics like end-to-end delay. Once again our analytical framework proves to be a convenient tool towards sorting such design issues. Suppose m_1 and m_2 in Figure 5.14 are mapped to the ST segment instead of the DYN segment. Figure 5.16(b) now shows how the end-to-end delay varies for various combinations of ST and DYN segment lengths for a bus cycle length (or period) of 16 ms (with all the other parameters being as described in the system architecture).

It should be mentioned here that we have adopted an analytic method to determine the performance metrics. Such methods provide hard performance bounds, but they are typically not able to model state-dependent behavior, which leads to pessimistic (but still correct) analysis results. Recently, [62] provided some interesting insights accuracy of the performance predictions provided by such analytic tools. It should be noted here that although simulation based approaches give tighter results, they suffer from insufficient corner case coverage. In future, it would be interesting to develop formal methods for FlexRay analysis based on

timed automata [6] which provide the exact performance predictions. However, we note that such formal techniques giving exact results are often paid for by a large analysis effort, i.e. may require long (or potentially unbounded) verification times.

5.6 Summary

In this chapter we presented a compositional performance model for a network of heterogeneous ECUs communicating via a FlexRay bus. FlexRay, which is backed by world's automotive industry, is in the most likely position to become the standard protocol in the industry. As such, of late there has been lot of interest in performance analysis of FlexRay-based networks. Our main contribution was a formal model of the protocol governing the DYN segment of FlexRay. We also showed how our framework may be exploited for design space exploration and thus assist the designer in choosing the optimal set of system parameters for his design constraints. We developed a tool for our framework, and demonstrated the applicability of our methods by evaluating a real world case study from the automotive domain.

Chapter 6

Conclusion

In this thesis we looked into several issues that lead to tedious interactive design exploration sessions for some common system-level analysis, namely, timing and scheduling analysis and multi-objective hardware/software co-design. Although these topics have already been widely studied, none of these studies focused on challenges arising in the context of interactive design cycles. Our thesis has made contributions in this direction, and the main results are summarized below.

- In this thesis, we presented a novel scheme for efficient schedulability analysis of *recurring real-time task sets*, to be used in interactive design sessions where the schedulability analysis is repeatedly invoked with small modifications in the task set. Since this scheme is used in an interactive fashion, we referred to it as *interactive schedulability analysis*.

  This concept of interactive schedulability analysis is fairly general and can be applied to a number of well-known task models. Our experimental results show that using our scheme can lead to more than 20× speedup for each invocation of the schedulability analysis algorithm, compared to the case where the full algorithm is run. In our work, we have also devised a technique

using which a system designer can be provided some feedback regarding which system parameter(s) should be changed that would likely yield a feasible solution.

- We developed an efficient scheme for multi-objective design space exploration in the context of evaluating *cost-utilization* tradeoffs for real-time systems. We derived a polynomial-time approximation algorithm for solving this NP-hard multi-criteria problem. Traditional approaches address these problems from an engineering perspective and rely on heuristics and randomized search techniques such as evolutionary algorithms. Our work in this thesis differs from these approaches by taking a classical approximation algorithms standpoint, where the goal is to provide formal guarantees on the quality of the results obtained.

  Our work is also interesting because there can be an exponentially large number of points in the Pareto front, which makes it impossible to compute this entire set in polynomial time. Hence, our polynomial-time approximation algorithm by default also implies approximating the (potentially exponential size) set with only a polynomial number of points. In a typical design or performance debugging scenario, a system designer inspects all the tradeoffs in the set and then selects one, or at most a few implementations. Hence, from a practical perspective, it is more meaningful if the designer is presented with a reasonably few well-distinguishable tradeoffs in the set, rather than an exponentially large number of solutions, many of which are very similar to each other. Our approximation algorithm is therefore not only attractive in terms of time-complexity, but also returns more meaningful solutions.

- Using two case studies, we showed that modern commodity graphics hardware may be exploited to accelerate computationally expensive kernels in design space exploration tools. In particular, we reformulated a schedulability

analysis algorithm and a multi-criteria design space exploration as *streaming* applications so that they maybe implemented on graphics hardware. We showed that our implementation achieves very attractive speedups compared to a standard CPU-based implementation.

Our contribution might also be valuable in light of the fact that the core problems solved are a variant of a classic optimization problem – the knapsack problem. This NP-hard problem is at the heart of numerous problems arising in the context of EDA and other areas of computer science and engineering. We believe that the generality of this problem might serve as a motivation to explore the possibility of exploiting GPUs for a variety of other combinatorial optimization problems. It might also be feasible to develop a toolbox for mapping a class of optimization problems to the GPU.

- We presented a compositional performance model for a network of heterogeneous ECUs communicating via a FlexRay bus. Our main contribution was a formal model of the protocol governing the dynamic segment of FlexRay. We also showed how our framework may be exploited for design space exploration and thus assist the designer in choosing the optimal set of system parameters for his design constraints. We developed a tool for our framework, and demonstrated the applicability of our methods by evaluating a real world case study from the automotive domain. Because we rely on analytical models, our tool returns results in a matter of few seconds, and is ideal for fast analysis in interactive design cycles. This is a distinct advantage over the existing simulation based tools, which take long running times during design processes.

6.1 Future Work

In this thesis, we could successfully establish that it is possible to ease the tedious interactive design space exploration sessions associated with some common performance analysis problems using various novel techniques. However, more work remains to be done to assess how relevant these methods and the results are in the design process (i) of other system-level performance analysis problem as well as (ii) of realistic systems in a practical/industrial setting. Towards this vision for future, our work spawns many new and promising research directions and poses some very interesting open questions, which are discussed below.

- Our framework for "interactive" schedulability analysis was established by demonstrating the concept with respect to a particular parameter i.e the task deadlines. However, in real-life designs the designer might like to have the flexibility to alter a different parameter like the execution times of the tasks, or the structure of the task graph. Extending the "interactive" framework for all such parameters would yield very exciting results and make it a very usable method.

  We also believe that it would be interesting to identify specific classes of changes for which the *interactive analysis* can be done in polynomial time. Further work should also be done towards providing more directed feedback to a system designer, compared to what we have presented in this thesis. Lastly, there are a number of recently developed tools for timing/schedulability analysis of embedded systems (see for example, [6, 38]). It would certainly be meaningful to explore if our analysis can be incorporated inside these tools in a smooth way.

  Although in this thesis, we have focused on the specific problem of schedulability analysis, we believe that such a *interactive* scheme can be used for a

variety of timing analysis problems e.g. worst-case execution time analysis of programs using program path analysis techniques. Moreover, many system-level design tools in the electronic design automation domain are being used by the designers in a interactive fashion. Since, our method exploits this repeated invocation of the algorithm to achieve speed-ups as well to provide feedback, it has the potential to be applied to all such problems.

- Our work on hardware/software design space exploration raises interesting questions (Chapter 3) as well. In this thesis we derived a fully polynomial-time approximation scheme (FPTAS) for solving this computationally intractable problem, and showed the validity of the results from a performance debugging perspective. However, this was solved only in a uni-processor environment. Nowadays multi-core platforms are increasingly becoming popular for design of real-time applications. Interestingly, the extension of the existing framework to the even to the dual-processor case seems intuitively difficult, and and FPTAS might not exist in this case. It would be interesting to establish the complexity of the problem, and if the problem is intractable, the challenge would be to propose suitable heuristics to solve the problem. It will also be interesting to fill in the details to extend our algorithm to more involved task models (e.g. the recurring real time task model).

  It may be also be noted that although our algorithm generated polynomial-sized $\mathcal{P}_\epsilon$ curves, they need not necessarily contain the fewest possible points required to represent an ϵ-approximate Pareto curve. It would be interesting to see whether it is possible to generate the smallest sized $\mathcal{P}_\epsilon$ in our setting, based on the recent results from [81].

- Using specific case studies, this thesis showed that modern commodity graphics hardware may be exploited to accelerate computationally expensive kernels in design space exploration cycles. Our contribution is valuable in light

of the fact that the core problem solved is a variant of a classic optimization problem, viz. the knapsack problem. This NP-hard problem is at the heart of numerous problems arising in the context of system-level design processes and other areas of computer science and engineering. We believe that the generality of this problem might motivate other researchers to explore the possibility of exploiting GPUs for a variety of other system-level design problems as well (e.g. schedulability analysis problems in multiprocessor settings).

- Our work with regards to performance analysis of automotive networks may also be extended in different dimensions. In practice multiple subsystems of FlexRay and other bus protocols like CAN would be connected by gateways to form larger networks. It would be meaningful to extend the existing framework to model the components like gateways in order to analyze larger networks. It would also be a practical extension to do a more *formal* backward analysis i.e. to find the suitable periods at which the sensors might be sampled, in order to meet a desired end-to-end delay.

  Further, we have assumed that the designer has taken the appropriate decisions regarding the issues of architecture selection, mapping and scheduling policies beforehand. It would be particularly interesting to investigate how these decisions affect the performance metrics. Also, in this thesis we have assumed all messages from a specified task to be of constant (worst-case) length. Relaxing this constraint to account for variable length messages will require certain modifications to our framework which would be interesting to explore. Again, from a practical perspective one would also like to explore the possibilities of integrating our implementation into standard tools for designing FlexRay-based systems such as those from DECOMSYS.

Bibliography

[1] M. Abramovici, J. T. de Sousa, and D. Saab. A massively-parallel easily-scalable satisfiability solver using reconfigurable hardware. In *Proc. 36th Design Automation Conference (DAC)*, pages 684–690. ACM Press, 1999.

[2] P. K. Agarwal, S. Krishnan, N. H. Mustafa, and S. Venkatasubramanian. Streaming geometric optimization using graphics hardware. In *Proc. 11th European Symposium on Algorithms (ESA)*, Lecture Notes in Computer Science 2832, pages 544–555. Springer, 2003.

[3] A. Ailamaki, N. K. Govindaraju, S. Harizopoulos, and D. Manocha. Query co-processing on commodity processors. In *Proc. 32nd International Conference on Very Large Data Bases (VLDB)*, pages 1267–1267. VLDB Endowment, 2006.

[4] K. Albers and F. Slomka. An event stream driven approximation for the analysis of real-time systems. In *Proc. 16th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 187–195. IEEE Computer Society, 2004.

[5] A. Albert. Comparison of event-triggered and time-triggered concepts with regard to distributed control systems. In *Embedded World*, Nürnberg, Germany, 2004. `www.semiconductors.bosch.de/pdf/embedded_world_04_albert.pdf`.

[6] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES: A tool for schedulability analysis and code generation of real-time systems. In *International Workshop on Formal Modeling and Analysis of Timed Systems*, Lecture Notes in Computer Science 2791, Marseille, France, 2003. Springer-Verlag.

[7] S. Baruah. Feasibility analysis of recurring branching tasks. In *Proc. 10th Euromicro Workshop on Real-Time Systems (ECRTS)*, pages 138–145. IEEE Computer Society, 1998.

[8] S. Baruah. A general model for recurring real-time tasks. In *Proc. 19th IEEE Real-Time Systems Symposium*, pages 114–122. IEEE Computer Society Press, 1998.

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

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

[11] S. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proc. 11th IEEE Real-Time Systems Symposium*, pages 182–190. IEEE Computer Society Press, 1990.

[12] M. Bauer, W. Ecker, R. Henftling, and A. Zinn. A method for accelerating test environments. In *Euromicro*, volume 01, page 1477, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

[13] G. Bernat and A. Burns. Three obstacles to flexible scheduling. In *Proc. 13th Euromicro Conference on Real-Time Systems (ECRTS)*, page 11. IEEE Computer Society, 2001.

[14] E. Bini and M. Di Natale. Optimal task rate selection in fixed priority systems. In *Proc. 26th IEEE International Real-Time Systems Symposium (RTSS)*, pages 399–409. IEEE Computer Society, 2005.

[15] E. Bini, M. Di Natale, and G. C. Buttazzo. Sensitivity analysis for fixed priority real-time systems. In *Proc. 18th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 13–22. IEEE Computer Society, 2006.

[16] J. L. Boudec and P. Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Springer-Verlag, New York, NY, USA, 2001.

[17] G.C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Boston, 1997.

[18] CAN Specification, Ver 2.0, Robert Bosch GmbH. `www.semiconductors.bosch.de/pdf/can2spec.pdf`, 1991.

[19] S. Chakraborty, T. Erlebach, S. Künzli, and L. Thiele. Schedulability of event-driven code blocks in real-time embedded systems. In *Proc. 39th Design Automation Conference (DAC)*, pages 616–621. ACM Press, 2002.

[20] S. Chakraborty, T. Erlebach, and L. Thiele. On the complexity of scheduling conditional real-time code. In *Proc. 7th International Workshop on Algorithms and Data Structures (WADS)*, Lecture Notes in Computer Science 2125, pages 38–49, 2001.

[21] S. Chakraborty, S. Künzli, and L. Thiele. Approximate schedulability analysis. In *Proc. 23rd IEEE Real-Time Systems Symposium (RTSS)*, page 159. IEEE Computer Society, 2002.

[22] S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Proc. 6th Conference on Design, Automation and Test in Europe (DATE)*, page 10190. IEEE Computer Society, 2003.

[23] C. A. C. Coello, A. Hernández Aguirre, and E. Zitzler, editors. *Proc. 3rd International Conference on Evolutionary Multi-Criterion Optimization*. Lecture Notes in Computer Science 3410, Springer-Verlag, 2005.

[24] J. Csirik, J. B. G. Frenk, M. Labbé, and S. Zhang. Two simple algorithms for bin covering. *Acta Cybernetica*, 14(1):13–25, 1999.

[25] nVIDIA CUDA Zone, http://www.nvidia.com/object/cuda_home.html.

[26] K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, 2001.

[27] DECOMSYS - Dependable Computer Systems, Hardware und Software Entwicklung GmbH. `www.decomsys.com`.

[28] dSPACE GmbH. `www.dspace.de`.

[29] R. Dutta, J. Roy, and R. Vemuri. Distributed design-space exploration for high-level synthesis systems. In *Proc. 29th Design Automation Conference (DAC)*, pages 644–650. IEEE Computer Society Press, 1992.

[30] R. Esser and J. W. Janneck. MOSES - a tool suite for visual modeling of discrete-event systems. In *IEEE International Symposium on Human-Centric Computing Languages and Environments*, Stresa, Italy, 2001. IEEE Computer Society. http://www.tik.ee.ethz.ch/∼moses/.

[31] J. Ferreira, P. Pedreiras, L. Almeida, and J. A. Fonseca. The FTT-CAN protocol for flexibility in safety-critical systems. *IEEE Micro*, 22(4):46–55, 2002.

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

[33] The FlexRay Communications System Specifications, Ver. 2.1. `www.flexray.com`.

[34] R. E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, 2000.

[35] N. Goodnight, C. Woolley, G. Lewin, D. Luebke, and G. Humphreys. A multigrid solver for boundary value problems using programmable graphics hardware. In *Proc. SIGGRAPH/Eurographics Conference on Graphics Hardware*, pages 102–111. Eurographics Association, 2003.

[36] M. Gries. Methods for evaluating and covering the design space during early design development. *Integration, the VLSI Journal*, 38(2):131–183, 2004.

[37] A. Hamann and R. Ernst. Efficient priority optimization in complex distributed embedded systems through search space adaptation. In *Proc. 9th Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1517–1517. ACM, 2007.

[38] A. Hamann, M. Jersak, K. Richter, and R. Ernst. Design space exploration and system optimization with SymTA/S – symbolic timing analysis for systems. In *Proc. 25th Real-Time Systems Symposium (RTSS)*, pages 469–478, 2004.

[39] R. Henftling, A. Zinn, M. Bauer, M. Zambaldi, and W. Ecker. Re-use-centric architecture for a fully accelerated testbench environment. In *Proc. 40th Design Automation Conference (DAC)*, pages 372–375. ACM, 2003.

[40] D. S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, Boston, 1997.

[41] Class B Data Communications Network Interface, SAE J1850 Srandard, Rev. 2, Nov. 1996. `www.interfacebus.com/Automotive_SAE_J1850_Bus.html`.

[42] B. Kienhuis, E. F. Deprettere, K. A. Vissers, and P. van der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In *International Conference on Application-Specific Systems, Architectures, and Processors*, pages 338–349. IEEE, 1997.

[43] D. Kim, S. Ha, and R. Gupta. Parallel co-simulation using virtual synchronization with redundant host execution. In *Proc. 9th Conference on Design, Automation and Test in Europe (DATE)*, pages 1151–1156. European Design and Automation Association, 2006.

[44] Y-I. Kim, W. Yang, Y-S. Kwon, and C-M. Kyung. Communication-efficient hardware acceleration for fast functional simulation. In *Proc. 41st Design Automation Conference (DAC)*, pages 293–298. ACM, 2004.

[45] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics*, 22(3):908–916, 2003.

[46] M. Laumanns, L. Thiele, K. Deb, and E. Zitzler. Combining convergence and diversity in evolutionary multiobjective optimization. *Evolutionary Computation*, 10(3):263–282, 2002.

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

[48] L-F. Leung and C-Y. Tsui. Energy-aware synthesis of networks-on-chip implemented with voltage islands. In *Proc. 44th Design Automation Conference (DAC)*, pages 128–131. ACM, 2007.

[49] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-software co-design of embedded reconfigurable architectures. In *Proc. 37th Design Automation Conference (DAC)*, pages 507–512. ACM, 2000.

[50] P. Lieverse, P. van der Wolf, K. A.Vissers, and E. F. Deprettere. A methodology for architecture exploration of heterogeneous signal processing systems. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, 29(3):197–207, 2001.

[51] Local Interconnect Network Specification, Lin Consortium. `www.lin-subbus.org`.

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

[53] W. Liu, B. Schmidt, G. Voss, and W. Müller-Wittig. GPU-ClustalW: Using graphics hardware to accelerate multiple sequence alignment. In *Proc. 13th International Conference on High Performance Computing (HiPC)*, pages 363–374. Springer, 2006.

[54] C. A. Lupini, T. J. Haggerty, and T. A. Braun. Class 2: General Motors' version of SAE J1850. In *Proc. 8th International Conference on Automotive Electronics*, pages 74–78, 1991.

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

[56] K. Mueller and F. Xu. Ultra-fast 3d filtered backprojection on commodity graphics hardware. In *Proc. 1st International Symposium on Biomedical Imaging*, pages 571–574, 2004.

[57] N. Navet, Y. Q. Song, F. Simonot-Lion, and C. Wilwert. Trends in automotive communication systems. *Proc. of the IEEE (special issue on Industrial Communications Systems)*, 96(6):1204–1223, 2005.

[58] N. Neophytou and K. Mueller. GPU accelerated image aligned splatting. In *Proc. 4th Eurographics / IEEE Visualization and Graphics Technical Committee (VGTC) Workshop on Volume Graphics*, pages 197–205. Eurographics Association, 2005.

[59] J. D. Owens, D. L., N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[60] M. Palesi and T. Givargis. Multi-objective design space exploration using genetic algorithms. In *Proc. 10th International Symposium on Hardware/software Codesign (CODES)*, pages 67–72. ACM, 2002.

[61] C. H. Papadimitriou and M. Yannakakis. On the approximability of trade-offs and optimal access of web sources. In *Proc. 41st Foundations of Computer Science (FOCS)*, page 86. IEEE Computer Society, 2000.

[62] S. Perathoner, E. Wandeler, L. Thiele, A. Hamann, S. Schliecker, R. Henia, R. Racu, R. Ernst, and M. González Harbour. Influence of different system abstractions on the performance analysis of distributed real-time systems. In *Proc. 7th International Conference on Embedded Software (EMSOFT)*, pages 193–202. ACM, 2007.

[63] P. Pop, P. Eles, and Z. Peng. Schedulability-driven communication synthesis for time-triggered embedded systems. *Real-Time Systems*, 26(3):297–325, 2004.

[64] T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei. Timing analysis of the flexray communication protocol. *Real-Time Systems*, 39(1-3):205–235, 2008.

[65] P. Popp, M. Di Natale, P. Giusto, S. Kanajan, and C. Pinello. Interactive presentation: Towards a methodology for the quantitative evaluation of automotive architectures. In *Proc. 11th Conference on Design, automation and test in Europe (DATE)*, pages 504–509. EDA Consortium, 2007.

[66] S. Punnekkat, R. Davis, and A. Burns. Sensitivity analysis of real-time task sets. In *Asian Computing Science Conference on Advances in Computing Science (ASIAN)*, 1997.

[67] S. P. Zwart R. G. Belleman, J. Bedorf. High performance direct gravitational n-body simulations on graphics processing units. *New Astronomy*, 13(2):103–112, 2008.

[68] R. Racu, A. Hamann, and R. Ernst. A formal approach to multi-dimensional sensitivity analysis of embedded real-time systems. In *Proc. 18th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3–12. IEEE Computer Society, 2006.

[69] R. Racu, M. Jersak, and R. Ernst. Applying sensitivity analysis in real-time distributed systems. In *Proc. 11th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 160–169. IEEE Computer Society, 2005.

[70] A. Rajnak and A. Kumar. Computer-aided architecture design & optimized implementation of distributed automotive ee systems. In *Proc. 44th Design Automation Conference (DAC)*, pages 556–561. ACM, 2007.

[71] R. J. Rost. *OpenGL Shading Language*. Addison-Wesley, 2006.

[72] I. Skliarova and A. B. Ferrari. A software/reconfigurable hardware sat solver. *IEEE Transactions on VLSI Systems*, 12(4):408–419, 2004.

[73] L. Soulé and T. Blank. Parallel logic simulation on general purpose machines. In *Proc. 25th Design Automation Conference (DAC)*, pages 166–171. IEEE Computer Society Press, 1988.

[74] H. Takada and K. Sakamura. Schedulability of generalized multiframe task sets under static priority assignment. In *Proc. 4th International Workshop on Real-Time Computing Systems and Applications (RTCSA)*, pages 80–86. IEEE Computer Society, 1997.

[75] K. Tindell, A. Burns, and A. Wellings. Calculating Controller Area Network (CAN) message response times. *Control Engineering Practice*, 3(8):1163–1169, 1995.

[76] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40(2-3):117 – 134, 1994.

[77] K. Tindell, H. Hanssmon, and A. J. Wellings. Analysing real-time communications: Controller Area Network (CAN). In *Proc. 15th Real-Time Systems Symposium (RTSS)*, pages 259 – 263. IEEE Society Press, 1994.

[78] V. T'kindt, J. C. Billaut, and H. Scott. *Multicriteria Scheduling : Theory, Models and Algorithms*. Springer-Verlag, 2006.

[79] H. Tokuda and M. Kotera. Scheduler 1-2-3: An interactive schedulability analyzer for real-time systems. In *Proc. 12th International Computer Software and Applications Conference (COMPSAC)*, pages 211 – 219. IEEE, 1988.

[80] ISO/CD11898-4, Road Vehicles Controller Area Network (CAN) Part 4: Time-Triggered Communication, International Standards Organization, Geneva, 2000.

[81] S. Vassilvitskii and M. Yannakakis. Efficiently computing succinct trade-off curves. *Theoritical Computer Science*, 348(2), 2005.

[82] S. Venkatasubramanian. The graphics card as a stream computer. In *SIGMOD-DIMACS Workshop on Management and Processing of Data Streams*, 2003.

[83] S. Vestal. Fixed-priority sensitivity analysis for linear compute time models. *IEEE Transactions on Software Engineering*, 20(4):308–317, 1994.

[84] E. Wandeler, A. Maxiaguine, and L. Thiele. Quantitative characterization of event streams in analysis of hard real-time applications. *Real-Time Systems*, 29(2-3):205–225, 2005.

[85] E. Wandeler and L. Thiele. Real-Time Calculus (RTC) Toolbox. `http://www.mpa.ethz.ch/Rtctoolbox`.

[86] N. H. Zamora, X. Hu, and R. Marculescu. System-level performance/power analysis for platform-based design of multimedia applications. *Transactions on Design Automation of Electronics Systems*, 12(1):2, 2007.

[87] M. R. Zargham. Parallel channel routing. In *Proc. 25th Design Automation Conference (DAC)*, pages 128–133. IEEE Computer Society Press, 1988.

[88] P. Zhong, M. Martonosi, P. Ashar, and S. Malik. Using configurable computing to accelerate boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):861–868, 1999.