Using Application Benefit for Proactive Resource Allocation in Asynchronous Real-Time Distributed Systems

By

Tamir A. Hegazy

Thesis submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requir ements of the degree of

Master of Science
in
Computer Engineering

APPROVED:

_____
Binoy Ravindran, Chairman

_____ _____
Nathaniel J. Davis, IV Scott F. Midkiff

September 20, 2001
Blacksburg, VA

Keywords: Resource Management, Resource Allocation, Asynchronous Real-Time Systems, Distributed Real-time Systems, Proactive Resource Allocation, Best-Effort Resource Allocation

Using Application Benefit for Proactive Resource Allocation in Asynchronous Real-Time Distributed Systems

Tamir A. Hegazy

Abstract

This thesis presents two proactive resource allocation algorithms, RBA* and OBA, for asynchronous real-time distributed systems. The algorithms consider an application model where timeliness requirements are expressed using Jensen's benefit functions and propose adaptation functions to describe anticipated workload for future time intervals. Furthermore, an adaptation model is considered where processes are replicated for sharing workload increases. A real-time Ethernet system model is considered where message collisions are resolved. Given such models, the objective is to maximize aggregate application benefit and minimize aggregate missed deadline ratio. Since determining the optimal allocation is computationally intractable, the algorithms heuristically compute the allocation so that it is as "close" as possible to the optimal allocation. While RBA* analyzes process response times to determine the allocation, OBA analyzes processor overloads to compute the decision in a much faster way. RBA* incurs a quadratic amortized complexity in terms of subtask arrivals for the most computationally intensive component when DASA is used as the underlying process-scheduling algorithm, whereas OBA incurs a logarithmic amortized complexity for the corresponding component. To study how different process-scheduling and message-scheduling algorithms affect the performance of the algorithms and to compare their performances, benchmark-driven experiments were conducted. The experimental results reveal that RBA* produces higher aggregate benefit and lower missed deadline ratio when DASA is used for process scheduling and message scheduling. Furthermore, it is observed that RBA* produces higher aggregate benefit and lower missed deadline ratio than OBA, confirming the intuition that accurate response time analysis can lead to better results.

Acknowledgments

"In the Name of God, the Most Merciful, the Most Compassionate"

Special thanks go to my family in Egypt, especially my parents. No words can describe what you have been doing for me, and no words can describe how much I love you all.

I would like to acknowledge the fund of the sponsors, the U.S. Navy.

I would like to sincerely thank my advisor, Dr. Binoy Ravindran, for his continuous help and support throughout the past year and a half of my research work. Without your help, I would not have gotten this work done, and I would not have had all those publications that will certainly help me in my future career.

I would also like to thank committee members, Dr. Nathaniel Davis and Dr. Scott Midkiff, for their support and for their comments and suggestions regarding the content of this thesis.

I would like to thank Ms. Christina Zamboni for her constant encouragement and support.

I would like to thank all members of the real-time research group at Virginia Tech. With your company and support, I was able to get this done.

Table of Contents

Table of Figures

Chapter 1: Introduction

Real-time computer systems that are emerging for the purpose of strategic mission management are "asynchronous" in the sense that processing and communication latencies do not necessarily have known upper bounds, and event and task arrivals may be non-deterministically distributed. Another source of non-determinism is that some events and state changes are apparently spontaneous to the computer system per se, because their causal reasons are from "outside" the system. This emerging generation of real-time distributed computing systems is very important for (real-time) supervisory control in many domains, including the defense, industrial automation, and telecommunications.

The vast majority of the past efforts on real-time computing focus on hard real-time systems that are usually centralized, but occasionally distributed [Butta97, Liu00]. The hard real-time computing theory assumes that application properties such as task execution times, communication delays, and execution environment characteristics such as process scheduling and message scheduling overheads are deterministically known with absolute certainty. The theory exploits such a-priori information and provides the hard real-time guarantee of "always meet all deadlines" on the application behavior. However, the non-determinism inherent in the application and execution environment of asynchronous real-time distributed systems generally makes it non-cost-effective or even impossible to complete the execution of every real-time computation at its optimum time (e.g., before the deadline) [Jen92].

Recent advances in distributed systems research have produced quality of service (QoS) technologies such as [AAS97, AS98, BN+98, HBNB97, HSNL97, Rav00, RLLS97, RSYJ97] that allow applications to specify and negotiate service expectations such as timeliness and accuracy, which were not previously possible under classical real-time theory. The QoS techniques consider application models where tasks can operate at multiple, discrete "levels" of service. A level is a strategy for doing a tasks' work and is characterized by a resource usage such as CPU usage, a QoS dimension such as accuracy

of computational results, and a user-specified benefit. Therefore, given a set of QoS levels at which a task can operate, an adaptation mechanism can determine the "right level" of QoS depending upon the availability of internal resources to optimize a system-wide criteria such as aggregate task benefit.

In this thesis, we advance the QoS technology by presenting two polynomial-time heuristic resource allocation algorithms for asynchronous real-time distributed systems. We consider an application model that consists of trans-node periodic and aperiodic tasks that have end-to-end timeliness requirements, expressed using Jensen's benefit functions [Jen92]. Further, we propose the concept of adaptation functions for describing the anticipated application workload during future time intervals. The functions are user-specified, can have arbitrary shapes, and may be dynamically modified to express the future application workload that is anticipated at a given time. Furthermore, we consider an adaptation model for the application, where subtasks of application tasks may be replicated at run-time for sharing workload increases, and a real-time Ethernet system model where message collisions are resolved using a deterministic message scheduling algorithm.

Given such application, adaptation, and system models, our objective is to maximize the aggregate application benefit and to minimize the application missed deadline ratio during a future window of time with a known, but arbitrary workload pattern. This problem can be shown to be NP-complete[1]. Thus, RBA* and OBA heuristically compute the allocation decision in polynomial-time, such that the resulting allocation seeks to maximize the aggregate benefit and minimize the missed deadline ratio, but not necessarily produce the maximum aggregate benefit and minimum missed deadline ratio.

The RBA* algorithm analyzes response times of task subtasks to compute accurate allocations so that they will yield "good" aggregate benefit and aggregate missed deadline ratios. Hence, the algorithm is named: Response Time Analysis-Based Best-effort Resource Allocation (or RBA*.) However, due to the complexity involved in

[1] NP-completeness of the resource allocation problem is discussed in Section 4.2.

analyzing subtask response times when sophisticated schedulers are used on processors, RBA* "pays" for the accurate decisions by incurring a worst-case computational complexity of $O\left(p^2m^4n^4\lceil W/k\rceil^4\right)$ under the DASA scheduler [Cla90], given *n* tasks, a maximum of *m* subtasks per task, *p* processors, minimum task period of *k*, and an adaptation window of length *W*. The RBA* algorithm presented in this thesis is an improved version of the RBA algorithm that we had presented in [RH01]. RBA* overcomes many of the pessimistic assumptions made by RBA to analyze subtask response times. Also, the amortized complexity of the most computationally intensive component is $O\left(N^2\right)$ under DASA.

The OBA algorithm on the other hand, avoids response time analysis, but analyzes processor overload situations in computing the allocations. Hence the algorithm is named: Overload Analysis-Based Best-Effort Resource Allocation (or OBA.) Since analyzing overloads on processors is scheduler-independent and relatively inexpensive than analyzing subtask response times, OBA incurs a better worst-case complexity of $O\left(p^2m^2n^2\lceil W/k\rceil^2\log\left(mn\lceil W/k\rceil\right)\right)$. Further, the amortized complexity of the component that is corresponding to RBA's most computationally intensive component is only $O\left(\log N\right)$ instead of $O\left(N^2\right)$. However, we hypothesize that OBA may perform worse than RBA* during overload situations — conditions we are clearly interested in, due to the asynchronous nature of the system. Our hypothesis is due to the fact best-effort scheduling algorithms make scheduling decisions during overload situations that are difficult to predict than decisions that are made during under-load situations.[2] Thus, we conjecture that it may be difficult to make "good" resource allocations by ignoring the precise behavior of the schedulers, especially during overload situations when they make decisions that are difficult to predict, but may have a significant impact on the resource allocation decisions.

[2] During under-loaded situations, most best-effort schedulers such as DASA and RED [BS95] "mimic" EDF [SSRB98], since EDF is optimal during such situations.

Thus, to study how different scheduling and message scheduling algorithms affect the performance of the resource allocation algorithms and to determine the relative performance of the algorithms under a broad range of load situations, we conduct benchmark-driven experiments. The experimental results reveal that RBA* produces higher aggregate benefit and lower missed deadline ratio when the DASA algorithm is used for process scheduling and message scheduling than under other algorithms. Furthermore, we observe that RBA* produces higher aggregate benefit and lower missed deadline ratio than OBA, confirming our intuition that accurate response time analysis can lead to better results.

Thus, the major contribution of the thesis is the RBA* and OBA algorithms that seek to maximize aggregate benefit and minimize aggregate missed deadline ratio in asynchronous real-time distributed systems. To the best of our knowledge, we are not aware of any work that solves the problem solved by RBA* and OBA.

The rest of the thesis is organized as follows: Chapter 2 provides a summary of the related work. Chapter 3 presents our application, adaptation, and system models. We discuss the objectives of the work and informally state the problem in Chapter 4. Chapter 5 discusses the heuristics employed by RBA*, the rationale behind the heuristics, and the complexity analysis of the algorithm. In Chapter 6, we present the OBA algorithm; the heuristics employed by OBA, the rationale behind the heuristics, and the complexity analysis of the algorithm. We discuss the experimental evaluation of the algorithms in Chapter 7. Finally, the thesis concludes with a summary of the work, its contributions, and future work.

Chapter 2: Past and Related Efforts

In this Chapter we introduce the past and related works that overlap with the work presented in this thesis. The work presented in this thesis considers resource management of both periodic and aperiodic tasks that might coexist in a system. Thus, we start by shedding the light on the past work of scheduling aperiodic tasks along with periodic tasks (Section 2.1.) Section 2.2 briefly describes the Quality of Service (QoS) works that consider resource allocation in real-time distributed systems. In Section 2.3, we present some relevant works that deal with general resource allocation in distributed systems that are not necessarily real-time systems. Section 2.4 discusses two recent system models that are suitable for real-time distributed systems.

2.1 Aperiodic Task Scheduling

Traditionally, scheduling and resource management of aperiodic computations is performed through reservation of resources and on-line admission control schemes [CSR86, DTB93, KS92, LRT92, LSS87, MoK83, NS94, RCF97, RSZ89, RTL93, TLS96, SB96, SLS88, SR91, SRC85, SSL89, ZRS87a, ZRS87b]. Admission control strategies are employed to ensure the adequacy of resources for achieving the real-time requirements of aperiodic tasks before accepting the events that trigger their execution. The techniques assume that the (worst-case) parameters of the computations such as execution times of the tasks/subtasks on the processors, communication delays of messages between tasks/subtasks, and distributions of event-arrivals are deterministically known.

The approach of resource reservation based on the worst-case conditions may not be applicable in asynchronous real-time systems. Asynchronous real-time distributed systems are subject load patterns that cannot be deterministically characterized. Thus, parameters such as execution times and communication delays have unknown upper bounds. Moreover, event arrivals have non-deterministic distributions in such systems. Hence, determining the worst-case resource needs and performing an a-priori reservation

of resources is not a viable approach. Even using ”large enough” estimations of the worst-case execution times and communication delays may result in system designs that are non-cost-effective.

Also, it is practically infeasible to perform admission control for aperiodic events. Aperiodic computations in asynchronous real-time distributed systems are often triggered by events that occur in the application environment e.g., detection of hostile threats that need to be engaged and destroyed in an air-engagement combat system. Furthermore, the computations perform mission-critical calculations such as determining the launch velocities and flight trajectories of weapons (e.g., interceptor missiles) that are detonated to engage the threats. Thus the aperiodic events need to be accepted in timely manner, which might not be the case with performing admission control because of the computation and the communication overhead that comes along with the on-line admission control.

2.2 Quality of Service

The QoS techniques primarily focus on application models where the application tasks can tolerate multiple levels of service that vary in their quality of computational results and requirements of resources. Therefore, given a set of QoS levels at which the application can operate, an adaptation mechanism can determine the “right level” of QoS depending upon the availability of internal resources and external load. Such QoS adaptation models and algorithms are presented in [AAS97, AS98, CSSL97, HLR01, HSNL97, HBNB97, Lee99, LLSR99a, LLSR99b, RLLS97, RLLS98, RS99, RSY98, RSYJ97, Rav00]. In the following paragraphs, we discuss the most relevant works among the works cited above.

In [AAS97], the authors propose a model for QoS negotiation in building real-time services to meet both predictability and graceful degradation requirements. The QoS negotiation mechanism proposed permits clients to express in their service requests a spectrum of QoS levels they can accept from the provider and perceived utility (benefit)

of receiving service at each of these levels. The task model used consists of clients that are application tasks. The execution model is influenced by the requirements of a flight control application but is still sufficiently general for use in different applications. The task set includes both periodic and aperiodic tasks where aperiodic tasks are assumed to be handled by a periodic server. A task is composed of a set of modules and has a deadline by which all its modules must complete. The modules may have arbitrary precedence constraints among themselves. However, tasks are independent – there is not any precedence constraints among different tasks. The QoS negotiation mechanism was shown to outperform the conventional "binary" admission control schemes and achieve higher application utility. In [AS98], a QoS-optimization algorithm and communication system architecture are proposed for communication in real-time systems. The algorithm and the architecture satisfy the following objectives: (1) provide per-flow or per-service class guarantees, (1) maximize the aggregate utility (benefit) of the communication services across all the clients, (3) adapt gracefully to transient overload, and (4) avoid, if possible, starving lower-priority service classes during the period of sustained overload.

In [RLLS97], the authors present an analytical model for QoS management in real-time systems in which application needs must be satisfied along multiple dimensions such as timeliness, reliable delivery schemes, cryptographic security, and data quality. The problem was later shown as NP-hard in [RLLS98]. The model assumes multiple concurrent applications, each of which can operate at different levels of quality based on the system resources available to it. Each application has a minimal resource requirement, but can adapt its behavior if given more resources and provide additional utility. The goal of the model is to be able to allocate resources to the applications such that the overall system utility is maximized under the constraint that each application can meet its minimum needs. The authors identify resource profiles of applications which allow such decisions to be made efficiently and in real-time. They also identify application utility functions along different dimensions. Under a single resource, the authors provided an optimal (or near-optimal) resource allocation schemes. In [RLLS98], the authors deal with a converse problem of the problem presented in [RLLS97]. They consider the problem of apportioning multiple resources to satisfy a single QoS dimension. In

practice, this problem becomes complicated, since a single QoS dimension perceived by the user can be satisfied using different combinations of available resources. In [LLSR99a], the authors consider a more complex problem; the problem of apportioning multiple finite resources to satisfy the QoS needs of multiple applications along multiple QoS dimensions. In other words, each application needs multiple resources to satisfy its QoS requirements. The authors evaluate and compare three strategies to solve this provably NP-hard problem. They show that dynamic programming and mixed integer programming compute optimal solutions to this problem but exhibit very long running times. Then, the mixed integer-programming problem is adapted to yield near-optimal results with smaller running times. Finally, the authors present an approximation algorithm based on a local search technique that is less than 5% away from the optimal solution but which is more than two orders of magnitude faster. In [LLSR99b], the work is advanced to support discrete QoS operating points. The problem of maximizing system utility by allocating a single finite resource to satisfy the QoS requirements of multiple applications along multiple QoS dimensions is addressed. Two near-optimal algorithms are presented and compared. One algorithm yields a solution with a bounded distance from the optimal solution and the other yields a solution within a user-specified distance from the optimal solution.

In [RSYJ97], the authors consider adaptive resource allocation for reactive high performance applications that must meet well-defined real-time constraints in dynamic execution environments. Each such application consists of multiple interacting components capable of executing in a distributed environment consisting of parallel machines, embedded system components (e.g., signal processors), and user interface stations (e.g., workstations. Components are either sequential or parallel tasks and their resource needs may be data-dependent, varying with changes in the rate or content of data input. The components can be programmed to adapt their resource needs at runtime by changes in the their execution mode, algorithms, or specific attributes such as the level of parallelism and communication protocols. The authors developed an application resource usage model that captures the characteristics of parallel real-time tasks that are required for making reallocation decision. Thus, the adaptation is performed by changing

the configuration of the application with different levels of parallelism. In [RSY98], the authors, provide a framework, called *FARA*, that provides abstractions and mechanisms for building integrated adaptation and resource allocation services in complex real-time systems. FARA provides abstractions and mechanisms that enable integrated, runtime adaptation and resource allocation decisions in systems with multiple applications and varied resources, while it facilitates the portability across multiple implementation platforms and applications. It also offers a hierarchical model for describing the adaptation capabilities of multi-component applications, a description of adaptation costs based on which the enactment delay may be evaluated, and (3) mechanisms that reduce the decision overhead through collaborative manipulation of decision context and batch evaluation of allocation decisions. Then the authors advance the FARA model presented in [RSY98] to another model called *FARACost* [RS99]. FARACost aims to benefit from available adaptation capabilities yet avoid critical timing failures. FARACost as a dynamic resource allocation mechanism is assumed to be "aware" of the perturbation induced by its decisions. FARACost captures the impact of application-specific adaptation procedures and uses this information to evaluate adaptation choices. The authors experimentally demonstrate that the use of models like FARA-Cost reduces or prevents pending timing constraint failures, while leading to long-term performance improvements.

In [HLR01], a QoS model and QoS admission control policies for dynamic system (where application continuously enter and depart the system) are presented. The model considered in that work is a set of tasks that can operate at varying levels of QoS. Taks randomly joind the systems requesting resources and depart from the system when they are done. The goal of the work is to choose a resource allocation that for these tasks that will deliver the highest total QoS while minimizing variation of QoS levels. The work introduced and compared three admission control policies. It was shown that, in general, while homogeneous algorithms can achieve the highest global utility, when stability is considered, non-homogeneous provide better a utility/stability tradeoff.

However, it is difficult to apply the QoS adaptation models described earlier in this section for achieving the requirements of aperiodic computations. This is because it is difficult to determine the multiple levels of service that the aperiodic computations can tolerate, which vary in their quality of results and resource requirements. Quality and accuracies of the aperiodic tasks are often critical to mission success. Furthermore, the QoS adaptation algorithms that perform the adaptation—selection of the "right level" of service—*during event arrival*— are impractical for aperiodic tasks due to the high cost incurred in performing the adaptation.

2.3 General Resource Allocation in Distributed Systems

In the distributed resource allocation literature, several works have appeared. In [KLB98], a proposed model for resource management to support service is provided. The resource management architecture is application-driven. This means that users are allowed to specify quality of service expectations for their applications in terms of application-level metrics, as opposed to low-level resource needs. Furthermore, the architecture supports dynamic resource management to accommodate and adjust, to changing user needs and resource demands is applicable in a wide variety of scenarios, and facilitates the development of quality of service enabled applications. The model makes no assumptions about the underlying system architecture, network, or operating system, which makes the model widely acceptable in resource allocation in distributed system. Nevertheless, the model may need cycles of adjustments until the user expectations are met. However, compared to proactive resource allocation, one disadvantage of this technique is the time it would take the resource manager components to adapt to load changes might be long in a way that is not suitable for mission-critical systems where timeliness requirements are not tolerable. Another disadvantage is the fact that urgent aperiodic events are cannot be subject to several cycles of adjustments. They need to be executed once they arrive. In [CMM97], the authors provide a resource allocation technique using multi-agent system. The multi-agent system is called "*Challenger*". *Challenger* consists of agents that individually manage local resources.

These agents communicate with one another to share their resources for efficient use of the system.

Although generally useful in resource allocation in distributed systems, the focus of the work is not aimed at systems with tight timeliness requirements. This limits the real-time use of the model. For example, the application model assumes a general application model where applications are not assigned deadlines. So, the model does not account for meeting application deadlines, which is a very fundamental issue in real-time systems.

2.4 Recent System Models for Asynchronous Real-Time Distributed Systems

In this section we discuss two recent system models that are suitable for real-time distributed systems, namely, the timed asynchronous distributed system model and the timely base model.

2.4.1 The Timed Asynchronous Distributed System Model

In [CF99], the authors formally define a model called the *timed asynchronous distributed system model* (for short, the *timed model*). The model assumes that (1) all services are timed: Their specification prescribes the outputs and state transitions that should occur in response to inputs, in addition to the time intervals within which a client can expect these outputs and transitions to occur, (2) interprocess communication is via an unreliable datagram service with omission/performance failure semantics: The only failures that messages can suffer are omission (message is dropped) and performance failures (message is delivered late), (3) processes have crash/performance failure semantics: The only failures a process can suffer are crash and performance failures, (4) processes have access to hardware clocks that proceed within a linear envelope of real-time, and (5) no bound exists on the frequency of communication and process failures that can occur in a system. The times model assumes that the distributed system consists of a finite data set of processes that communicate via a datagram service. The model mainly makes assumptions about the hardware clocks, the datagram service, and the process management service.

Hardware Clocks

The clocks are mapped from the local clock time to the real-time using a well-defined function. Local processes access the same local clock while remote processes access different clocks at different nodes. The timed model does not require the clocks to be calibrated or synchronized. Rather, it only places a restriction on the drift value which has to be bounded. Note that he process management service that runs in each node uses the node's local clock to manage alarm clocks that allow the local processes to request to be awakened whenever desired

Datagram Service

The datagram service used by the timed system is assumed to have *send()*, *broadcast()*, and *deliver()* primitives. The main requirements of the datagram service are (1) Validity: If the datagram service delivers a message to a process at a certain time and identifies a sender, then the sender has indeed sent that message in a previous time (2) *No-duplication*: Each message has a unique sender and is delivered at a destination process at most once, (3) *Min-Delay*: Any message sent between two remote processes has a minimum transmission delay.

Process Management Service

The process management service assumes that each process can be in one of three states, namely *up*, *crashed*, and *recovering*. A process is said to be the *up* state, f it is executing its standard program code. A process is said to be in the *crashed* state, if it has stopped executing its code and has lost all its previous state. Finally, a process is said to be in the *recovering* state, if it is executing its state initialization. This can happen either after its creation or when it restarts after a crash.

The timed model is assumed to be suitable for applications with soft real-time constraints, since they need a notion of time. The timed model provides these applications with a sufficiently strong notion of time.

The system model considered in this thesis agrees with the timed model in the following aspects:

- It assumes that the hardware clocks are synchronized using a technique similar to the one described in [Mills95].
- It considers a datagram service for communication.

However, the process management service considered in the timed model is not the focus of this thesis. The application, adaptation, and system model description and motivations are presented later in Chapter 3.

2.4.2 The Timely Computing Base

In [VCF00], a model called TCB (Timely Computing Base) is proposed. The model has two main aspects. The fist aspect is that the heterogeneity in system synchronism is cast into the system architecture. There is a generic or *payload* system, over a global network or *payload* channel. This prefigures what is normally 'the system' in homogeneous architectures, that is, where applications run. Additionally, there is a *control* part, made of local TCB modules, interconnected by some form of medium, the *control* channel. The medium may be a virtual channel over the available physical network or an alternative network in its own right. The second aspect is that the TCB has well-defined synchronism properties. The TCB provides simple support services, such as the ability to detect failures, to measure durations, and to execute *timely* timed actions. For certain types of less critical applications, it is not necessary that all sites have TCBs.

The system is assumed to consist of processes that exchange messages and may exist in several nodes. Nodes are interconnected by a communication network. The system can have any degree of synchronism, that is, if bounds exist for processing or communication

delays, their magnitude may be uncertain. Also, local clocks may not exist or may not have a bounded drift rate. The system is assumed to follow omissive failure model, that is processes only have late timing failures. Further, the system model has uncertain timeliness, that is, that is bounds may be violated. Given all the above assumptions, the system still must be dependable with regard to time, that is, it must be capable of timely executing certain functions or detecting failures.

The TCB was shown to be capable of executing timely functions, although the rest of the system can be asynchronous. Special hardware is not mandatory to achieve synchrony of the TCB. The authors assumed a computational model based on the TCB to support applications (algorithms, services, etc.) based on any synchrony of the payload system, from asynchronous to synchronous. From a system design viewpoint, this is the same as saying from non real-time to hard real-time. Thus, they proposed a few services for the TCB such as (1) timely execution, (2) duration measurement, and (3) timing failure detection. The authors also devised an application-programming interface allowing the notion of time to propagate from the TCB to payload applications.

The application/adaptation model considered in this thesis agrees with the timely computing base model in the fact that processes can have only timing failures. On the other hand, unlike the timely computing base model, the system model considered in this thesis does assume that the hard clocks are synchronized using a technique similar to that described in [Mills95]. The application, adaptation, and system model description and motivations are presented later in Chapter 3.

As discussed above, the idea of performing proactive resource allocation is asynchronous real-time distributed systems is not well addressed by the past efforts. The goal of this thesis is to provide a solution to the problem where maximizing the application level aggregate benefit and minimizing the missed deadline ratio are the main objective. The solution should neither assume upper bounds on the workloads nor assume any probabilistic distribution on the aperiodic events arrival. The problem addressed in this

thesis is not addressed by either the timed asynchronous distributed system model or the timely based model.

Chapter 3: The Application, Adaptation, and System Models

3.1 Application Model

In this section, we present the application model used in this work. The following subsections introduce the task model, timing requirements, and the application profiles.

3.1.1 Task Model

The application model assumes a system with a set of periodic and aperiodic tasks. A periodic task executes periodically usually at a fixed period. A periodic task in a real-time system usually processes data that comes out of a sensor such as a radar system in a mission critical system or a video camera in a multimedia system. For example, a periodic sensing task periodically processes the data of the objects detected by a radar system. Another example is a periodic image-processing task that periodically processes the digital image samples generated by a video camera. On the other hand, an aperiodic task is a task that gets triggered at arbitrary points of time. The events that trigger the aperiodic tasks do not posses a regular arrival pattern. However, in many cases, the aperiodic events arrival can be characterized as a probability distribution of event inter-arrival time.

The task model in this thesis is this model is motivated by the US Navy's AAW system [WRSB98]. The system is assumed to be fully asynchronous. As a result, it is assumed that there is not known upper bounds on the system loads or probability distributions that can characterize the event arrivals. Also, it is assumed that an aperiodic task is triggered upon the completion of the execution of some instants of a periodic task. We denote the set of tasks in the application by the set $T = \{T_1, T_2, T_3 ...\}$, where a task can be either periodic or aperiodic. Each aperiodic task T_j has a "triggering" periodic task T_k that triggers its execution. After a periodic task T_k completes its execution, it may generate zero or more number of events that trigger the execution of the corresponding aperiodic task T_j.

The period of a task T_i is denoted as *period*(T_i), if T_i is periodic. If T_i is aperiodic, *period*(T_i) denotes the period of the periodic task that triggers task T_i. The periods of periodic tasks need not be the same and can be different for different tasks.

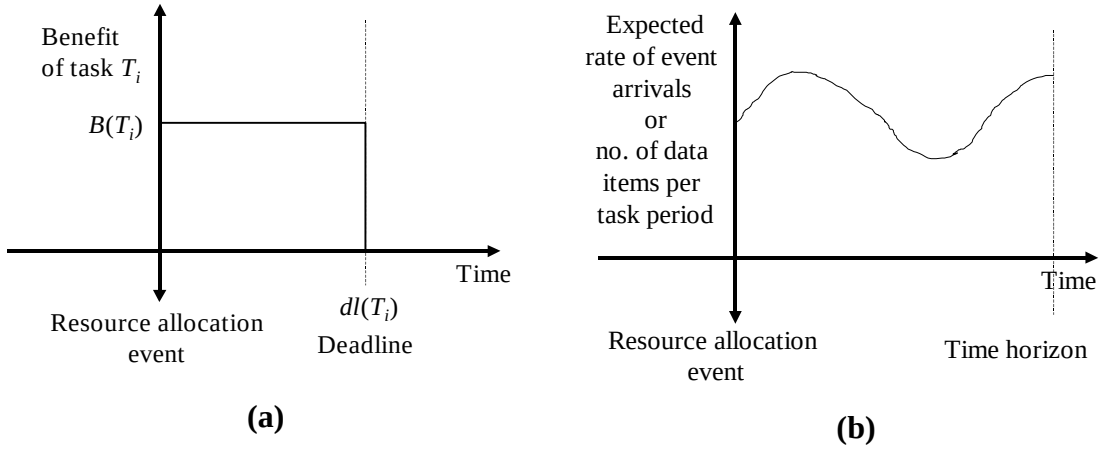


Figure 3.1. (a) A Benefit Function and (b) an Adaptation Function

Each task T_i is assumed to consist of a set of subtasks (executable programs), which execute in a *serial* fashion. We use the notation $T_i = [st_1^i, m_1^i, st_2^i, m_2^i, ..., st_n^i, m_n^i]$ to represent a task T_i that consists of *n* subtasks and *n* messages to be executed in series. That is, subtask $st_j^i (i > 1)$ cannot execute before message m_{j-1}^i arrives. The set of subtasks of a task T_i is denoted as $ST(T_i) = \{st_1^i, st_2^i, st_3^i, ..., st_n^i\}$ and the set of inter-subtask messages of a task T_i is denoted as $MS(T_i) = \{m_1^i, m_2^i, m_3^i, m_4^i, ..., m_n^i\}$.

3.1.2 Timing Requirements

We use Jensen's benefit functions for expressing application timeliness requirements [Jen92]. Such benefit functions express the benefit that the application gains by completing a task as a function of the completion time. Benefit functions can have arbitrary shapes. For example, it can be a decreasing ramp function, which means that the benefit drops linearly as a function of completion time. It can also be a rectangular function (See Figure 3.1(a).) In some cases, it may be desirable to still achieve some

benefit after the deadline is missed. In other cases, the application does not achieve any benefit if it completes execution any time after its deadline, in which case the application is said to have achieved zero benefit. Sometimes, it makes sense to measure the "lateness" of the task completion by considering a negative benefit if the task finishes later than the deadline. Furthermore, benefit functions can be classified as unimodal and multimodal. The benefit functions can express any timing constraint model including hard timing constraints, soft timing constraints, and general timing constraint. We assume rectangular benefit functions for all tasks such as the one shown in Figure 3.1(a). Thus, completing a task anytime before its deadline will result in uniform benefit; completing it after the deadline will result in zero benefit. We denote the benefit of a task T_i —the height of T_i's benefit function—as $B(T_i)$ and the deadline of a task T_i as $dl(T_i)$, respectively.

Jensen's benefit functions are considered for expressing timing requirements because they capture both the deadline of a task and the importance or the "contribution" of this task to the whole system. Note that using only deadline to express the timing requirement does not capture the importance of the task to the system. As a result, the goal of minimizing the missed deadline ratio may not be sufficient to achieve optimal or near-optimal application benefit. Further, rectangular benefit functions are considered in the work presented in this thesis, because some of the best-effort scheduling algorithms, such as DASA [Cla90], considered support only rectangular benefit functions.

3.1.3 Application Profiles

We model the workload of a subtask as the number of data items that it needs to process, and we model the workload of an inter-subtask message as the number of data items that it needs to transmit. Our motivation for this is the fact that the number of sensor reports and aperiodic events (or "data items") processed and transmitted by subtasks and messages, respectively, constitute the most significant part of the application workload in many real-time distributed applications that we regard as asynchronous [WRSB98]. Furthermore, the major element of uncertainty in the processing and communication

latencies in these systems is due to the uncertainty in the number of sensor objects and aperiodic events that the application has to process, as they are dependent upon the applications' external environment.

We assume that application-profile functions that can estimate subtask execution times as a function of data size workloads are available. Such execution time estimates can be regarded as a worst-case lower bound for processing a given data size workload. The profile functions can be determined by application profiling and measurement as described in [Deva01]. We denote the lower bound on the estimated execution time of a subtask st_j^i for processing a data size d as $eex(st_j^i, d)$.and the estimated communication delay of an inter-subtask message m_j^i to transport a data size d as $ecd(m_j^i, d)$.

3.2 Adaptation Model

We assume an adaptation model for the application where subtasks of tasks can be *replicated* at run-time. The idea behind replication of subtasks is that once a subtask is replicated, replicas of the subtask can share the workload that was processed by the original subtask. Furthermore, concurrency can be exploited by executing the replicas on different processors and thereby the end-to-end task latency can be reduced. Thus, replication is allowed as a means to reduce task latencies when task workloads increase at run-time. For simplicity in the design of the application and the resulting application model, we assume that the workload of a subtask is equally distributed among all its replicas.

The state consistency of the subtask replicas is not considered in this work, as we assume that the tasks process data objects that are "continuous" in the sense that their values are obtained directly from a sensor in the application environment, or computed from values of other such objects. The replicas are thus assumed to be *temporally consistent* without applying every change in value, due to the continuity of physical phenomena.

We propose adaptation functions for expressing anticipated workload scenarios of the application during future time intervals. The adaptation functions can be specified by the user. It is believed that the "human-in-the-loop" approach with the concept of adaptation functions will enable strategic allocation of resources that will yield greater system utility. The adaptation functions describe the anticipated application workload as a function of the time (or a reference point such as task period) at which it is anticipated to occur. The functions can have arbitrary shapes and are user-specified for each application task. We regard the origin of the functions' axes as a "resource allocation event" in the sense of the start time of the expected scenario for the workload. The function is specified for a fixed duration of time into the future and ends at a time instant called the "time horizon." The anticipated workload scenarios—and thus the adaptation functions—may be dynamically modified, as the user's perception regarding the future workload changes. We regard adaptations functions to be derived from the requirements and operational scenarios of the application. An example adaptation function is shown in Figure 3.1(b).

In this work, we use task periods as reference points for describing anticipated workloads. Thus, we denote the anticipated workload of a task T_i during a period p as $Adapt(T_i, p)$. For a periodic task, the anticipated workload is defined as the number of data items that is anticipated during a task period. For an aperiodic task, the anticipated workload is defined as the number of events that the triggering periodic task (of the aperiodic task) is anticipated to produce at the end of its period. The anticipated workload is assumed to be a constant during the task period, but may be different for different periods.

Note that the anticipated workload is not a necessity for resource allocation. If the anticipated workload is not available, resource allocation can still be done by using either (1) the workload that was observed during past time intervals or (2) a projected workload from the past workloads. Thus, it is important to note that resource allocation is independent of adaptation functions. If the functions are available, they will facilitate user-specified adaptation of the system.

3.3 System Model

We consider a LAN-like distributed system where hosts are interconnected through a message-scheduling server in a star-topology fashion. The message-scheduling server is equipped with a multi-port Ethernet adapter such as [Znyx01]. Each host is connected to the message-scheduling server using a full-duplex Ethernet link (IEEE 802.3) and to a port at the message-scheduling server that is dedicated for the host. Thus, the link between each processor and the message-scheduling server becomes a dedicated link for simultaneous two-way communication link A message that is sent from an application subtask is first transmitted from the processor of the subtask to the message-scheduling server and then from the message-scheduling server to the processor of the destination subtask (See Figure 3.2.)

There are several reasons behind considering this Ethernet model in this thesis. First, Ethernet is a fast, cheap, and a widely available technology. Second, there are some problems in considering other network technologies. For example, ATM is very expensive compared to Ethernet. Also, some technologies such as FDDI are outdated.

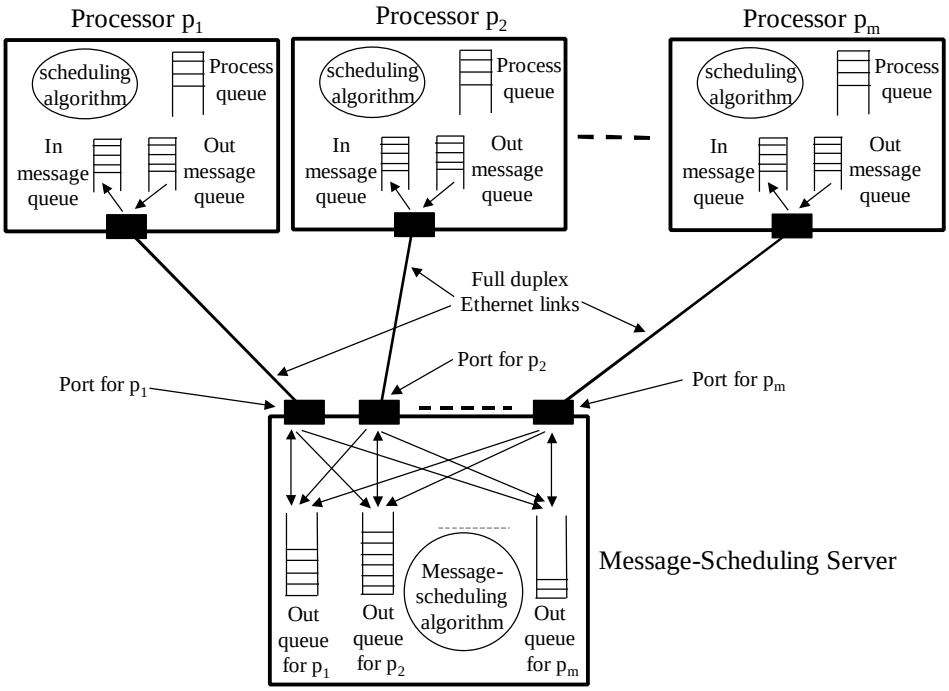


Figure 3.2. The Real-Time Ethernet Network System Model

The message-scheduling server simply maintains a list of message ready queues, one queue per host. Each queue stores the messages that are destined for the corresponding host. When the outgoing link from the message-scheduling server to a destination host becomes free for transmission, the message-scheduling server uses a message-scheduling algorithm that selects a message for transmission from the message ready queue of that host. Thus, messages collisions that occur due to access contention for the network is resolved using the deterministic message-scheduling algorithm. Recall that the links between the hosts on one side and the message-scheduling server on the other side are dedicated full-duplex links. So, collisions are not likely to happen, since there is no contention for the links.

The set of processors on the hosts is denoted by the set $PR = \{p_1, p_2, p_3, ...\}$. We assume that the clocks of the processors are synchronized using an algorithm such as [Mills95]. Furthermore, we use the non-preemptive version of the scheduling algorithm used at the local processors for scheduling messages at the message-scheduling server. This is done for system homogeneity and the consequent simplicity that we obtain in the system model.

For process scheduling and message scheduling, best-effort real-time scheduling algorithms such as DASA [Cla90] LBESA [Loc86], RED [BS95], and RHD [Butta97] are considered (An explanation of each of the scheduling algorithms is provided Appendix A). We consider best-effort algorithms, as they are known to outperform EDF (Earliest Deadline First real-time scheduling algorithm) during overloaded situations and perform the same as EDF during under-loaded situations where EDF is optimal.

We believe that this model is an extension of existing standards. Consider the IEEE 802.1p standard [IEEE802a, IEEE802b] that allows attaching priorities to messages. It establishes eight levels of priority similar to IP Precedence. Network adapters and switches route traffic based on the priority level. Using Layer 3 (of the OSI model) switches allows mapping 802.1p prioritization to IP Precedence before forwarding to

routers. From the scheduling point of view, we believe that our message-scheduling server model is a normal extension to the 802.1p model. It allows applying any (non-preemptive) scheduling algorithm for message scheduling that could be more that just mere priority scheduling (e.g. best-effort scheduling algorithm like DASA [Cla90]). Where the message-scheduling algorithm on the server is implemented within the OSI model is left open to make it general.

Chapter 4: The Objectives and Problem Statement

This chapter introduces the resource allocation problem, given the application, adaptation and system model described in the previous chapter. The objectives and problem statement are introduced in Section 4.1. Section 4.2 discusses the NP-completeness of the resource allocation problem.

4.1 Problem Statement

Given the application, adaptation, and system models described in Chapter 3, our objective is to maximize the aggregate task benefit and minimize the aggregate task missed deadline ratio during the time window of the adaptation functions. We define the aggregate task benefit as the sum of the benefit accrued by the execution of each task. During the future time window, each task T_i may execute more than once. At the end of each execution, if the task is completed by its deadline, the benefit of the task $B(T_i)$ is added to the aggregate benefit. (Recall that we consider only rectangular benefit functions as described in Chapter 3.) The aggregate missed deadline ratio is defined as the number of all the task executions within the future time window that missed their deadlines to the total number of task executions during that time window.

Thus, the problem that we are addressing in this thesis can be informally stated as follows:

"Given adaptation functions for each task in the application that may have arbitrary shapes and thus define an arbitrary task workload, what is the number of replicas needed for each task subtask for each possible execution? And what is the processor assignment for executing those replicas such that the resulting resource allocation will maximize the aggregate task benefit and minimize the aggregate task missed deadline ratio, during the time window of the adaptation functions?"

This problem can be shown to be an NP-complete problem as described in the next section. Thus, RBA* and OBA are heuristic algorithms that solve the problem in polynomial time, but do not necessarily determine the number of subtask replicas and their processor assignment that will yield the maximum aggregate task benefit and minimum aggregate task missed deadline ratio.

Besides determining a resource allocation that is as "close" as possible to the optimal allocation, we are also interested in determining the impact that different subtask scheduling and message scheduling algorithms have on the performance of the resource allocation algorithm. We experimentally evaluate this through a benchmark-driven experimental study. So, we study the effect of using DASA [Cla90], LBESA[Loc86], RED [BS95], and RHD [Butta97] as the underlying process-scheduling and message-scheduling algorithms for the developed resource allocation algorithms that are used to solve this problem.

4.2 NP-Completeness of the Resource Allocation Problem

We prove that (a special case of) the resource allocation problem is NP-complete by mapping a special case of the problem to the well-known zero-one knapsack problem. The zero-one knapsack problem is informally specified as follows "Given a set of items, where each item has an integer value and an integer weight, find the subset of items that will not exceed a total weight *C*, for a given *C*, but result in the maximum possible total value." Mathematically, this can be formulated as follows.

Find the values of the boolean variables x_i^* such that $\sum_{i=1}^{n} x_i^* v_i$ is maximal. This is subject to $\sum_{i=1}^{n} x_i^* w_i \le C$; where v_i and w_i are the value and the weight of the item *i*, respectively. *C* is the capacity of the knapsack. If the item *i* is selected, then x_i^* has a value 1. Otherwise, x_i^* has a value 0.

Now, consider a special case of our resource allocation problem, where each subtask of each task can have only one replica. Consider having periodic tasks only that have the same period length. Each task consumes a certain amount of resources and has a user-specified benefit. Consider allocating resources for one period *p* of execution of the tasks. The special case resource allocation problem becomes: Find the values of the boolean variables x_i^* such that $\sum_{i=1}^{n} x_i^* B(T_i)$ is maximal. This is subject to $\sum_{i=1}^{n} x_i^* u_i \le U$ Where $B(T_i)$ is the benefit of the task T_i (as defined in section 3.1), U is the maximum achievable utilization of the distributed system, and u_i is the resources usage percentage due to executing task T_i during period p. u_i is defined as the ratio of the summation of the "pure" (without interference from other subtasks) execution of the subtasks and the pure transmission times of the messages to the period of the task T_i.

$$u_i = \frac{\sum_{st_j^i \in ST(T_i)} eex\left(st_j^i, Adapt\left(T_i, p\right)\right) + \sum_{m_j^i \in MS(T_i)} \frac{Adapt\left(T_i, p\right)}{l_s}}{period\left(T_i\right)}$$

where l_s is the speed of the link that carries the message.

From the discussion presented above, we conclude that a special case of the resource allocation problem is equivalent to the zero-one knapsack problem. Hence, the special case of the resource allocation problem is NP-complete. This completes the proof.

For the general resource allocation problem, the periods of the tasks need not be equal. Tasks could have arbitrary periods. Further, the number of subtask replicas need not be one replica for all subtasks of all tasks. Subtasks can have multiple replicas. The number of replicas of a subtask could be different from the number of replicas of the other subtasks within the same task and the other subtasks of other tasks. All those factors add to the intractability of the problem.

Chapter 5: The RBA* Algorithm: Heuristics and Rationale, and Complexity Analysis

Since the objective of resource allocation is to maximize aggregate benefit and minimize aggregate missed deadline ratio, the desired properties of the RBA* algorithm include:

(1) *Allocate resources in the decreasing order of task benefits*. By doing so, we increase the possibility of maximizing aggregate benefit, as the task selected next for resource allocation is always the one with the largest benefit among the unallocated tasks;

(2) *Allocate resources for each task until its deadline is satisfied*. In other words, for a given task, the algorithm should allocate the minimum number of replicas for its subtasks that satisfies the deadline of that task. By this, we maximize the possibility of minimizing the aggregate task missed deadlines. Furthermore, there is no reason to allocate resources for a task once its deadline is satisfied, since the task benefit functions are rectangular and drop to zero benefit after the deadline. Furthermore, note that having more replicas than it is necessary generates more messages communication overhead due to the increase in the number of the messages between the replicas;

(3) *De-allocate resources for a task if its deadline cannot be satisfied*. A task that cannot satisfy its deadline contributes a zero benefit to the aggregate benefit. Hence, running such task is useless from the overall system benefit point of view. Not running such task will not decrease the system benefit. So, all the resources that were allocated to that task should be de-allocated. By doing so, we save system resources, which can be potentially used for satisfying deadlines of lower benefit tasks. This will increase the possibility of satisfying the deadlines of greater number of lower benefit tasks, resulting in potential contributions of non-zero benefit from them toward aggregate benefit.

(4) *Not allow a lower benefit task affect the timeliness of a higher benefit task.* Note that when resources are being allocated for a task, we may reach a point before the satisfaction of the task deadline, after which any more increase in resources for the task will negatively affect the timeliness of higher benefit tasks, decreasing the aggregate task benefit that is accrued so far. At such points, it is not obvious what choice—whether to continue the allocation for the task or to stop and de-allocate—can yield higher aggregate benefit. For example, it may be possible that continuing the resource allocation for the task may eventually satisfy it's deadline (at the expense of one or more higher benefit tasks.) However, this may also satisfy the deadlines of greater number of lower benefit tasks resulting in greater aggregate task benefit than the benefit that would be achieved if we were to de-allocate the task and proceed to the next lower benefit task. At such points of "diminishing returns," RBA* makes the choice of de-allocating all resources allocated to the task. The rationale behind this choice is that since it is not clear how many higher benefit tasks will have to "pay" for satisfying the task deadline, it maybe best not to "disturb" the aggregate benefit that is accrued so far. Moreover, since resources are always allocated in decreasing order of task benefits, the chances of obtaining a higher aggregate benefit may be higher by satisfying as many high benefit tasks as possible.

Thus, RBA* performs resource allocation according to the heuristic choices discussed here. We now summarize the algorithm as follows:

RBA* performs resource allocations when user-modifications to adaptation functions of application tasks are detected. Since the anticipated workload may be different for different task periods in the time window specified by the adaptation functions, the algorithm allocates resources for each period in the time window of the adaptation function, starting from the earliest period and proceeding to the latest. When triggered, the algorithm first sorts all tasks according to their benefits. For each task and for each adaptation period (in decreasing order of task benefits and period occurrences,

respectively), RBA* determines the number of replicas needed for each task subtask and their processor assignment that will satisfy the task deadline for the current period. While computing the number of subtask replicas for the task, if the timeliness of a higher benefit task is affected or if the task is found to be infeasible, the algorithm de-allocates all allocated replicas and proceeds to the next adaptation period.

The pseudo-code of RBA* at the highest level of abstraction is shown in Figure 5.1. In the first step, the algorithm constructs a heap that uses task benefit as the key values. In the second step, it iterates over the sorted set of tasks to allocate resources for each task. The algorithm allocates resources for all the tasks for each possible execution (Recall that periodic tasks execute every period, and aperiodic tasks execute whenever they get triggered by the corresponding triggering periodic tasks). For each task execution, the algorithm allocates replicas for all the task subtasks. The replicas are allocated for a subtask until the task deadline is satisfied.

```
RBA_Algorithm(T, W)  /* T is the task set and W is the future time window */
1. Construct a heap of tasks using task benefit as the key value;
2. For each task i = 1 to |T|
    2.1 Extract the task T_i from the heap; /* tasks will be considered in decreasing-benefit order */
    2.2 For each period j = 1 to ⌈W / period(T_i)⌉
        2.2.1 For each subtask k = 1 to |ST(T_i)|
                RBA_DetermineReplicasAndProcessors(st_k^i, j, Adapt(T_i, j));
```

Figure 5.1. Pseudo-code of the RBA* Algorithm at the Highest Level of Abstraction

Determining the number of replicas needed for each subtask of a task and their processor assignment that will satisfy the task deadline for a given adaptation period is non-trivial. The main difficulty lies in determining the end-to-end task response time given some number of replicas for each task subtask, as it requires holistic analysis of the system, which can be computationally intractable. To solve this problem, the algorithm employs the following important heuristic decision:

Assign deadlines to subtasks and messages of the task from the task deadline in such a way that if all subtasks and messages of the task can meet their respective deadlines, then the task will able to meet its deadline.

Based on this heuristic, we can determine the number of replicas needed for each task subtask that will satisfy the task deadline by solving the problem at the subtask-level i.e., by determining the number of replicas needed for each subtask that will satisfy the *subtask deadline*, as deadline satisfaction of all subtasks of the task is assumed to automatically satisfy the task deadline.

We now discuss the steps involved in determining the number of subtask replicas and their processor assignment. Section 5.1 discusses subtask and message deadline assignment. Estimation of subtask response time and message delays is discussed in Sections 5.2. Section 5.3 discusses how RBA* computes the number of replicas and their processors. In Section 5.4, we analyze the complexity of the algorithm. We summarize this chapter in Section 5.5.

5.1 Deadline Assignment of Subtasks and Messages

The problem of subtask and message deadline assignment from task deadlines has been studied in a different context [KG97]. The equal flexibility (EQF) strategy presented in [KG97] assigns deadlines to subtasks and messages from the task deadline in a way that is proportional to their execution times and communication delays, respectively. Suppose we have a task that consists of subtasks and messages with known execution times and communication delays, respectively. We define the task slack time to be the difference between the task absolute deadline and the task completion time, which is the same as the difference between the relative deadline of the task and the sum of the execution times and the communication delays of the subtasks and the message, respectively.

What EQF suggests is that the deadline of the task should be divided among the subtasks (and the messages) such that the subtask (or the message) individual relative deadline is

equal to the execution time (or the communication delay) plus a ratio of the slack that is equal to the ratio of the execution time (or the communication delay) of the subtask (or the message) to the total task execution time. The total task execution time is the sum of the execution times of the messages and the communication delays of the messages. Experimentally, EQF is shown to outperform other deadline assignment methods regarding the end-to-end missed deadline ratio [KG97].

RBA* uses EQF in the following way: The algorithm estimates subtask execution times and message communication delays using application-profile functions for a workload that is anticipated on the average. The estimated execution times and message delays are then used to assign subtask and message deadlines, according to EQF, respectively. Thus, the deadline of a subtask st_i^k for an assumed average data size of d_{avg} is given by:

$$dl\left(st_i^k\right) = eex\left(st_i^k, d_{avg}\right) + \left(dl\left(T_k\right) - \sum_{j=i}^{m} eex\left(st_j^k, d_{avg}\right) - \sum_{j=i+1}^{m} ecd\left(m_j^k, d_{avg}\right) \right) \times \left[eex\left(st_i^k, d_{avg}\right) \Big/ \left(\sum_{j=i}^{m} eex\left(st_j^k, d_{avg}\right) + \sum_{j=i+1}^{m} ecd\left(m_j^k, d_{avg}\right) \right) \right]$$

The deadline of a message m_i^k for an assumed average data size of d_{avg} is given by:

$$dl\left(m_i^k\right) = ecd\left(m_i^k, d_{avg}\right) + \left(dl\left(T_k\right) - \sum_{j=i}^{m} ecd\left(m_j^k, d_{avg}\right) - \sum_{j=i+1}^{m} eex\left(st_j^k, d_{avg}\right) \right) \times \left[ecd\left(m_i^k, d_{avg}\right) \Big/ \left(\sum_{j=i}^{m} ecd\left(m_j^k, d_{avg}\right) + \sum_{j=i+1}^{m} eex\left(st_j^k, d_{avg}\right) \right) \right]$$

5.2 Estimating Subtask Response Time

The response time of a subtask st_i^j of a task T_j under fixed priority schedulers is given by the classical equation $R_i^j = C_i^j + I_i^j$, where R_i^j is the subtask response time, C_i^j is the subtask execution time, and I_i^j is the interference that the subtask experiences from other

subtasks [ABR+93]. However, this equation is insufficient for best-effort schedulers such as DASA and LBESA that we are considering in this work, as they make decisions at each scheduling event that are functions of the remaining subtask execution times at the event. The remaining execution time of a subtask at a given time instant is defined as the difference between the total execution time of the subtask and the time that the subtask has already spent being executed on the processor up to this time instant.

To determine the response time of a subtask on a processor, we need to know the scheduling events, which are the time instants the scheduler has to execute to select a subtask from the ready queue. Those scheduling events are the arrivals and completion times of the subtasks. To determine subtask arrival times, we assume the following:

A1: Each periodic task arrives at the beginning of its period;
A2: Each aperiodic task arrives when the triggering message from its triggering periodic task arrives (According to the application model, an aperiodic task gets triggered upon the completion of the execution of its triggering periodic task);
A3: The response time of a subtask is the longest response time among all its replicas;
A4: The transmission delay of a message is equal to the deadline assigned to the message using EQF;
A5: The first subtask of a task is assumed to arrive at the beginning of the period of its parent task; any other subtask is assumed to arrive after the elapse of an interval of time (since the start of the task execution) equal to the sum of the message delays and subtask response times of all of that subtask's predecessor messages and all predecessor subtasks, respectively.

A1 and **A2** are straightforward assumptions. According to **A3**, a subtask is considered to finish execution when all of its replicas finish processing data objects passed to them. If only a subset of the replicas of a subtask completes execution by a certain time, the data objects that were passed to the subtask are not all processed by that time. This is because some data objects are still being processed by other replicas. As a result, the subtask cannot be considered to have finished execution, if only a subset of its replicas completed

processing their data objects. All replicas have to finish before we can say that the subtask has finished execution. The last replica to finish will determine the response time of the whole subtask. Therefore, the response time of a subtask is the longest response time among all replicas. **A4** assumes the worst-case communication delay of the messages, if the messages were to arrive by the deadline. To determine more accurate communication delays given a certain message scheduling-algorithm at the message-scheduling server, this would require constructing a list of all message arrivals to perform holistic analysis at the message-scheduling server, at the sender host, and at the receiver host, which is a very complicated process that adds more complexity to the algorithm. Instead, we consider the worst-case communication delays as a rough estimate for the communication delays. **A5** is directly derived from the precedence relationship between tasks and messages (see Section 3.1.) Although some of these assumptions are not realistic, we believe that they are reasonable for estimating the number of subtask replicas.

Given the previous assumptions, the arrival time of a subtask can be determined as the sum of the response times of all subtasks and the deadlines of all messages that precede the subtask (under consideration) plus the arrival time of the parent task of the subtask. Thus, given the arrival time of a task T_i, the arrival time of a subtask st_j^i of the task is given by:

$$arrival\left(st_j^i\right) = arrival\left(T_i\right) + \sum_{\forall k:k<j} \left[response\left(st_k^i\right) + dl\left(m_k^i\right) \right]$$

The arrival time of each subtask on a processor can thus be determined and an arrival list can be constructed. Note that the algorithm considers subtasks within a task according to their precedence order. So, by the time the algorithm attempts to determine the arrival time of a subtask, the response times of its predecessor subtasks will have been determined.

Our eventual goal is to determine the subtask response times by examining the arrival list in increasing order of arrival times and applying the scheduling function at each arrival time. For this purpose, the arrival list must be sorted according to the arrival times. This

can be accomplished by inserting the arrival time of a subtask into an integer-ordered list at an integer position that corresponds to the subtask arrival time. Thus, when all subtask arrival times are determined and inserted into the list, the list automatically gets ordered according to arrival times.

Once the arrival times of subtasks are determined, RBA* estimates the anticipated workload during each task adaptation period using the task adaptation functions. (For aperiodic tasks, the algorithm uses the period of their triggering periodic tasks as the task period.) The anticipated workloads are then "plugged into" the application profile functions to estimate the subtask execution times (without interference) during the task periods. The pure execution times are used to determine the response times as will be presented later in this section.

The algorithm now estimates the subtask response times by determining the scheduling events that occur during the time window and by applying the scheduling algorithm at each scheduling event to determine the scheduling decision. Note that it is impossible to determine the subtask response times without determining the scheduling events (and the decision made at each event) for algorithms such as DASA and LBESA as their decisions at each event depends on the remaining subtask execution times at the event.

We illustrate how the response time of a subtask can be determined with a simple example in Section 5.2.1. Then, we present the algorithm for response time analysis in Section 5.2.2.

5.2.1 An Example Subtask Response Time Analysis

Consider the task set shown in Table 5.1 where the (relative) deadlines of subtasks and messages are computed from the end-to-end task deadline using EQF. Note that the sum of the subtask and message deadlines is equal to the end-to-end task deadline for all the tasks.

We now consider the problem of determining the response time of subtask st_2^3 during the third period of task T_3. Since T_3 is periodic, then its arrival time is assumed to be the beginning of the third period i.e., at time $t = (3-1)\times 20 = 40$. Then, the arrival time of st_2^3 is computed by adding the previously determined response time of the subtask st_1^3; say 6, and the relative deadline of the message m_1^3 to the arrival time of T_3. Thus, st_2^3 is assumed to arrive at time $t = 40+6+1 = 47$.

Table 5.1. An Example Task Set

Task	T_1	T_2	T_3	T_4
Type	aperiodic	aperiodic	periodic	periodic
Benefit	40	35	24	15
Period	--	--	20	40
Triggering Task	T_3	T_4	--	--
Subtasks & messages	$st_1^1, m_1^1, st_2^1, m_2^1$	$st_1^2, m_1^2, st_2^2, m_2^2, st_3^2, m_3^2$	$st_1^3, m_1^3, st_2^3, m_2^3$	st_1^4, m_1^4
End-to-end deadline	25	40	20	40
Subtask & message deadlines	8, 2, 13, 2	12, 2, 13, 2, 9, 2	7, 1, 11, 1	34, 6

We consider a time window from $t = 0$ to the absolute end-to-end deadline of the parent task T_3 during this execution. Since T_3 is assumed to arrive at $t = 40$, the absolute deadline is the sum of the arrival time and the end-to-end deadline of T_3, which is at $t = 40 + 20 = 60$. Thus, the time window is the interval [0, 60]. Throughout the time window [0, 60], we consider the arrivals of all subtasks of higher benefit tasks on the processor where replicas of the subtasks are executing.

Consider the subtasks st_2^1 and st_1^2 of the aperiodic tasks T_1 and T_2, respectively, that have benefits higher than task T_3. Suppose that st_2^1 and st_1^2 have replicas that are executing on the same processor as that of subtask st_2^3. Further, assume that the replica of st_2^1 will be executing on this processor for the entire duration of the time window [0, 60].

Furthermore, assume that the replica of st_1^2 will be executing only during the first period of task T_4 when the parent task T_2 (of st_1^2) is triggered by task T_4 (Suppose this information is known after replicating the subtasks of T_1 and T_2 and st_1^3 of T_3.) We compute the subtask arrival times as described previously (Assumption A5) given what the response times are for the subtasks that are running on the processor under consideration. The response times of the other subtasks are determined after we have allocated resources for them. Now, throughout the time window [0, 60], the arrivals of the tasks will be as shown in Figure 5.2, assuming knowledge of the response times of their predecessor subtasks.

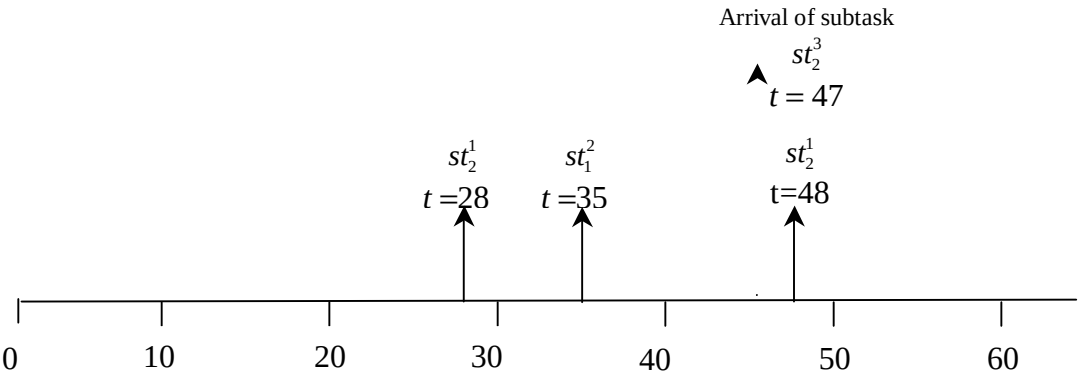


Figure 5.2. Example of Arrival Time Computations Used for Response Time Analysis

Once the arrival times of subtasks are determined, the algorithm estimates the subtask execution times using the adaptation functions to determine the expected load at each task period. Recall that given a task period, the adaptation function returns the expected task load during that period. For the aperiodic tasks, we use the period of their triggering periodic tasks as the task period. The algorithm estimates the subtask response times by determining the scheduling events that occur during the time window and by applying the process-scheduling algorithm (used at the local processors) at each scheduling event to determine the scheduling decision. Note that it is impossible to determine the subtask response times without determining the scheduling events (and decisions made at each event) for scheduling algorithms such as DASA and LBESA as they make decisions at

each scheduling event that are functions of the remaining subtask execution times at the event.

Table 5.2 illustrates the procedure followed by the algorithm assuming that LBESA [Loc86] is used as the process-scheduling algorithm. The algorithm uses two lists for determining the response times: *Arrival* and *Process*. The lists have three fields for each subtask arrival: (1) the subtask that is expected to arrive, (2) the time at which the arrival occurs, and (3) the remaining execution time of the subtask. The list *Process* represents the ready queue of the processor. The algorithm first initializes the list *Arrival* to contain all the expected arrival events and initializes the list *Process* to "empty." The earliest event is then removed from the *Arrival* list and is placed in the list *Process*. For the task set example, the earliest arrival is the arrival of st_2^1 at $t = 28$. Also, the arrival time of the earliest event (i.e., 28) is saved as the "*current time*."

The algorithm uses the process-scheduling algorithm to determine which subtask from the ready queue (i.e., the list *Process*) will be selected to execute. In our case, there is only one subtask in the ready queue and therefore will be selected by the scheduler for execution. The algorithm then computes the expected finishing time of the selected subtask by adding the current time to the remaining execution time of the subtask. The expected finishing time is then compared with the earliest subtask arrival time in the list *Arrival* to determine the next earliest event. The algorithm then processes this next earliest event.

Whenever the expected finishing time of a subtask becomes earlier than the next arrival event, the algorithm updates the current-time variable with the subtask finishing time. The process-scheduling algorithm is then used to select a subtask from the ready queue. On the other hand, if the next arrival event is earlier than the finishing time of the selected subtask, the algorithm updates the current time variable with the next subtask arrival time. The difference between this (next) arrival time and the time of the last scheduling decision is then determined and subtracted from the remaining execution time of the currently running subtask. The next arrival event is removed from the *Arrival List*

and is placed in the list *Process*. The scheduling algorithm is then applied on the list *Process* to determine the new scheduling decision.

Table 5.2. Determining Subtask Response Times under LBESA

Iteration #	*Time*	*Arrival List*	*Process List*	*Scheduler (LBESA) decision*	*Expected finishing time of the scheduled subtask*	*Next Event*
0	0	[st_2^1 , 28, 8] [st_2^1 , 48, 9] [st_1^2 , 35, 7] [st_2^3 , 47, 6]	Empty	N/A	N/A	Arrival [st_2^1 , 28, 8]
1	28	[st_2^1 , 48, 9] [st_1^2 , 35, 7] [st_2^3 , 47, 6]	[st_2^1 , 28, 8]	st_2^1	Current time + remaining time = 28 + 8 = 36	Arrival [st_1^2 , 35, 7]
2	35	[st_2^1 , 48, 9] [st_2^3 , 47, 6]	[st_2^1 , 28, 1] [st_1^2 , 35, 7]	st_2^1	35 + 1 = 36	Termination of st_2^1 at $t = 36$
3	36	[st_2^1 , 48, 9] [st_2^3 , 47, 6]	[st_1^2 , 35, 7]	st_1^2	36 + 7 = 43	Termination of st_1^2 at $t = 43$
4	43	[st_2^1 , 48, 9] [st_2^3 , 47, 6]	Empty	N/A	N/A	Arrival [st_2^3 , 47, 6]
5	47	[st_2^1 , 48, 9]	[st_2^3 , 47, 6]	st_2^3	47 + 6 = 53	Arrival [st_2^3 , 48, 6]
6	48	Empty	[st_2^1 , 48, 9] [st_2^3 , 47, 5]	st_2^1 (Preempt st_2^3)	48 + 9 = 57	Termination of st_2^1 at $t = 57$
7	57	Empty	[st_2^3 , 47, 5]	st_2^3	57 + 5 = 62	Termination of st_2^3 at $t = 62$
8	62	Empty	Empty	N/A	N/A	N/A

Table 5.3. Response Times of Subtasks for the Example Task Set

Subtask	*Arrival time*	*Finishing time*	*Response time*	*Relative deadline*
st_2^1 (1st period of the triggering periodic task)	28	36	8	13
st_2^1 (2nd period of the triggering periodic task)	48	57	9	13
st_1^2	35	43	8	12
st_2^3	**47**	**62**	**15**	**11**

$RBA\_AnalyzeResponse(s, p, q, l)$ /* s is the subtask, p is the period number, q is the processor, l is the workload */

1. $ArrivalEvents[\ ] = \varnothing$; $Process[\ ] = \varnothing$;

/* Lists $ArrivalEvents[\ ]$ and $Process[\ ]$ have 3 fields: time, subtask, remaining_time */

2. $x = ParentTaskNumber(s)$; $y = SubtaskNumber(y, x)$; /* y is subtask # of s within its parent task x */

3. If $type(s) = PERIODIC$ /* $type(s)$ returns PERIODIC if s is periodic; otherwise returns APERIODIC */

   3.1 $ArrivalTime(s) = (p-1) \times period(T_x) + \sum_{r<y} \left(ResponseTime(st_r^x) + dl(m_r^x) \right)$;

   3.2 $StopTime = p \times period(T_x)$;

4. else /* s is aperiodic */

   4.1 $ArrivalTime(s) = p \times period(T_x) + \sum_{i<y} \left(ResponseTime(st_i^x) + dl(m_i^x) \right)$;

   4.2 $StopTime = (p+1) \times period(T_x)$;

5. $ArrivalEvents = ArrivalEvents \cup \left[st_y^x, ArrivalTime(s), eex(s,l) \right]$;

6. For each task i = 1 to $x-1$ /* consider all tasks that have higher benefits */

   6.1 For each period $j = 1$ to $\lceil StopTime / period(T_i) \rceil$

      6.1.1 For each subtask $k = 1$ to $|ST(T_i)|$

         If subtask st_k^i has a replica executing on processor q during period j

            1. $ArrivalTime(st_k^i) = p \times period(T_i) + \sum_{r<k} \left(ResponseTime(st_r^i) + dl(m_r^i) \right)$;

            2. If $type(st_k^i) = APERIODIC$ $ArrivalTime(st_k^i) = ArrivalTime(st_k^i) + period(T_i)$;

            3. $ArrivalEvents = ArrivalEvents \cup \left[st_k^i, ArrivalTime(st_k^i), eex(s,l) \right]$;

            /* the arrival time of subtask st_k^i is inserted at the $ArrivalTime(st_k^i)^{th}$ position in $ArrivalEvents$ */

7. $TerminationTime = \infty$; $SchedulerTime = 0$; $CurrentProcess = 1$;

8. For each arrival event k from $ArrivalEvents$ /* in increasing order of arrival events */

   8.1 If $ArrivalEvents[k].time < TerminationTime$

      8.1.1 $Process[CurrentProcess].remaining\_time = Process[CurrentProcess].remaining\_time - ArrivalEvents[k].time - t$;

      8.1.2 $t = ArrivalEvents[k].time$;

      8.1.3 $Process = Process + \{ ArrivalEvents[k] \}$;

      8.1.4 $ArrivalEvents = ArrivalEvents - \{ ArrivalEvents[k] \}$;

      8.1.5 $CurrentProcess = LocalScheduler(Process[\ ])$;

      8.1.6 $TerminationTime = t + Process[CurrentProcess].remaining\_time$;

   8.2 else

      8.2.1 $t = TerminationTime$;

      8.2.2 If $Process[CurrentProcess].subtask = s$

         $ResponseTime = t - ArrivalTime(s)$;

      8.2.3 If $t > Process[CurrentProcess].time + dl(Process[CurrentProcess].subtask)$

         return ∞; /*one of the subtasks of the higher benefit tasks missed its deadline*/

      8.2.4 $Process = Process - \{ Process[CurrentProcess] \}$;

9. return $ResponseTime$;

Figure 5.3. The *RBA_AnalyzeResponse* Procedure

The algorithm repeats the process until both lists become empty. The response time of a subtask is computed as the difference between its arrival time and finishing time (See Table 5.2.). The response times of the subtasks for the example task set are shown in Table 5.3. The last row of the table shows the expected execution time of st_2^3 on the processor under consideration during the third period of its parent task T_3.

5.2.2 The Subtask Response Time Analysis Algorithm

The pseudo-code of the subtask response time analysis algorithm is shown in Figure 5.3. The procedure *RBA_AnalyzeResponse*, accepts a subtask *s*, a task period *p*, a processor *q* on which the response time of *s* needs to be determined, and the workload of the subtask as its arguments. It computes the response time of the subtask *s* during the period *p* on the processor *q*. As a byproduct, the procedure determines the response times of all subtasks that are assigned to processor *q*. It then compares the subtasks' response times with the subtasks' deadlines. If all subtasks satisfy their deadlines, the procedure returns the response time of the subtask *s*. If any subtask is found to miss its deadline, the algorithm returns a "failure" value, indicating that replicating subtask *s* on processor *q* will either not satisfy the deadline of *s* or affect the timeliness of higher benefit tasks. Note that whenever the procedure *RBA_AnalyzeResponse* is invoked for a subtask *s* for a processor *q*, all existing subtasks on *q* must belong to higher benefit tasks than the task of *s*, since RBA* allocates replicas to tasks in decreasing order of their benefits.

The layout of the procedure *RBA_AnalyzeResponse* does not change with the change of the scheduling algorithm used on the processors and the message-scheduling server. The procedure only invokes an abstract function called *LocalScheduler* (Figure 5.3, line 8.1.5) that is responsible for selecting the next subtask from the process ready queue depending upon the algorithm used. Since we are considering best-effort algorithms such as DASA, LBESA, and RED for subtask scheduling and message scheduling, we need to map task-level benefit (that is user-specified) into benefit values for subtasks and messages of the task. This is because best-effort algorithms use benefit values of subtasks and messages in making their scheduling decisions. Thus, we define the benefit of a subtask and a

message as simply the benefit of its parent task. Hence, all subtasks and messages of a task inherit the task benefit. Local schedulers and the message-scheduler deal with subtasks and messages, as opposed to tasks. At the local scheduler level, the benefit of a task is reflected by the benefit of its subtasks and messages. Assigning the benefit of the task to its subtasks and messages is an intuitive and straightforward way that captures this issue.

5.3 Determining Number of Subtask Replicas and Their Processors

RBA* determines the number of subtask replicas and their processor assignment as follows: Let the subtask st_q^i be considered for replication, which currently has a single replica, denoted as, say $st_{q,0}^i$. The algorithm first analyzes the response time of $st_{q,0}^i$ on its processor, say p_i. If the response time of $st_{q,0}^i$ is found to be less than the subtask deadline without affecting the timeliness of subtasks of higher benefit tasks, the algorithm concludes that the single replica $st_{q,0}^i$ is enough to satisfy its deadline and then proceeds to the next subtask.

If the response time of $st_{q,0}^i$ is larger than the subtask deadline or if executing $st_{q,0}^i$ on processor p_i makes one or more subtasks of higher benefit tasks miss their deadlines, the algorithm reduces the data size load of $st_{q,0}^i$ by replication. The algorithm considers a second replica for the subtask, denoted as, say $st_{q,1}^i$, which will reduce the data size load of the existing replica by half. To determine the processor for executing $st_{q,1}^i$, the algorithm analyzes the subtask response time for processing half the data size on each of the processors, excluding p_i, as described in Section 5.2. The processor that gives the shortest response time, say p_j, is selected. The algorithm re-computes the response time of the first replica $st_{q,0}^i$ on processor p_i under half the data size, since $st_{q,1}^i$ will now process the other half of the data size. If the response time of both the replicas is found to be shorter than the subtask deadline and execution of the replicas on the processors p_i and

p_j is not found to affect the timeliness of higher benefit tasks, then two replicas are considered to be sufficient by the algorithm. Otherwise, a third replica will be considered, and so on.

RBA* repeats the process until each replica is able to satisfy the subtask deadline. Note that as the number of replicas increases, the workload share of each replica will be reduced. Every time the algorithm considers adding a new replica, it checks whether the existing ones will be able to satisfy their deadlines under the reduced load without affecting the timeliness of the higher benefit tasks. If the algorithm determines that executing the maximum possible number of replicas for a subtask (which is equal to the number of processors in the system, for exploiting maximum concurrency) does not satisfy the subtask deadline, it assumes that the subtask, and hence the whole task, will miss their deadlines. Then, all the resources allocated to the task so far will be de-allocated (as discussed in the RBA* properties described earlier in this chapter.)

RBA_DetermineReplicasProcessors(s,i,l) /* s is the subtask, i is the period, and l is the anticipated load */

1. if *EstimateResponse*$(s,i,prr(s_0),l) < dl(s)$ return SUCCESS; /* done, if first replica is enough */
   /* s_0 is the first replica of s; $prr(s_0)$ is the processor that is executing s_0 */
2. $PT = PR - pr(s)$; /* $pr(s)$ is the set of processors that are executing replicas of s;
   PT is thus the set of processors that are not executing replicas of s */
3. if $PT = \varnothing$ return FAILURE;
4. For each processor $q \in PT$
   4.1 *ResponseTime* $= RBA\_AnalyzeResponse\left(s,i,q,l/\left(|pr(s)|+1\right)\right)$;
   /* $\left(|pr(s)|+1\right)$ is the number of replicas after increasing it by one and $l/\left(|pr(s)|+1\right)$ is the new load */
   4.2 Determine the minimum response time and the corresponding processor. Save the value of the minimum response time in *MinResponse* and the corresponding processor ID in $p_{\min}$;
5. $PT = PT - \{p_{\min}\}$;
6. $pr(s) = pr(s) \cup \{p_{\min}\}$;
7. if $MinResponse > dl(s)$ return to step 3;
8. For each processor $q \in pr\left(st_j^i\right) - \{p_{\min}\}$
   8.1 if $RBA\_AnalyzeResponse\left(s,i,q,l/|pr(s)|\right) > dl(s)$ return to step 3;
9. return SUCCESS;

Figure 5.4. Algorithm for Determining Number of Subtask Replicas and their Processors

Figure 5.4 shows the pseudo-code of the algorithm that determines the number of the subtask replicas and their processor assignment. The procedure *RBA_DetermineReplicasProcessors* accepts a subtask *s*, a period *i*, an anticipated workload *l* during the period *i*, and determines the number of replicas for *s* and their processors. Recall that the procedure *RBA_Algorithm* (Figure 5.1) invokes the procedure *RBA_DetermineReplicasProcessors* for all the subtask executions in the system.

5.4 The Complexity of RBA*

In Section 5.4.1, we analyze the worst-case complexity of RBA*. In section 5.4.2, we analyze the amortized complexity of the most computationally intensive component of RBA* which is *RBA_AnalyzeResponse*. For that procedure, amortized complexity is considered since it generally provides a "tighter" upper bound than the worst-case complexity.

For the following subsections, let *n* denote the number of tasks in the application, *p* denote the number of processors in the system, *m* denote the maximum number of subtasks of a task (thus in the worst-case, all *n* tasks will have *m* subtasks), *k* denote the smallest task period among all tasks (thus in the worst-case, all *n* tasks will have a period *k*), and *W* denote the length of the adaptation function.

5.4.1 Worst-Case Complexity of RBA*

The complexity of the *RBA_Algorithm* procedure (Figure 5.1) depends upon the complexity of the procedure *RBA_DetermineReplicasProcessors* (Figure 5.4). The complexity of *RBA_DetermineReplicasProcessors* depends on the procedure *RBA_AnalyzeResponse* (Figure 5.3) that determines the response time of a subtask. First, we consider the complexity of *RBA_AnalyzeResponse*. Second, we discuss the complexity of *RBA_DetermineReplicasProcessors*. Finally, we discuss the complexity of *RBA_Algorithm*.

I. Complexity of *RBA_AnalyzeResponse*

The complexity of *RBA_AnalyzeResponse* consists of two components. First, given a subtask and a processor on which the response time of the subtask is to be determined, procedure *RBA_AnalyzeResponse* first constructs an arrival list for all subtasks on the processor. Second, for each subtask in the constructed arrival list, the procedure then invokes the scheduler for each of its arrival and departure within the length of the adaptation function. Thus, the cost of *RBA_AnalyzeResponse* is simply the sum of the cost of constructing the arrival list and the cost of invoking the scheduler for each scheduling event i.e., for each arrival and termination event of a subtask.

a. Arrival List Construction

Since a subtask can be replicated for a maximum number of p times and since RBA* does not assign two or more replicas of the same subtask on the same processor, the maximum number of subtask replicas that can be assigned by RBA* to a processor is mn (i.e., all m subtasks of a task $\times$ n tasks.) Each of the mn subtasks can arrive during all the periods of its parent task throughout the adaptation function widow W. The largest possible number of arrivals of a subtask is $\lceil W/k \rceil$. Thus the largest arrival list will have a size of $mn\lceil W/k \rceil$.

To construct the arrival list, *RBA_AnalyzeResponse* determines the arrival time of each replica on the processor. To determine the arrival time of a replica, *RBA_AnalyzeResponse* examines each predecessor subtask and message of the replica. Thus, the cost of determining the arrival time of a single replica involves examining d predecessor subtasks and d predecessor messages, incurring a total cost of $O(d)$, where d is the number of the predecessor subtasks of the subtask under consideration.

Once the arrival time of a subtask is determined, the procedure *RBA_AnalyzeResponse* inserts the subtask arrival time into a heap using a key value that corresponds to the subtask arrival time. Recall that the largest list size was determined to be $mn\lceil W/k \rceil$. The insertion cost in a heap is $O\left(\log\left(mn\lceil W/k \rceil\right)\right)$. Finally, the cost of constructing the ordered arrival list for all the $mn\lceil W/k \rceil$ subtask arrivals on a processor is

$$mn\lceil W/k \rceil \times O\left(d + \log\left(mn\lceil W/k \rceil\right)\right) = O\left(mdn\lceil W/k \rceil + mn\lceil W/k \rceil \log\left(mn\lceil W/k \rceil\right)\right).$$

b. Response Time Analysis

The response time analysis is performed by invoking the local scheduler at each subtask arrival and departure. The cost of invoking the scheduler is obviously dependent on the scheduling algorithm employed. If we consider the DASA/ND algorithm, then the cost of computing a scheduling decision, given *r* processes in the ready queue of the processor, is $O\left(r^2\right)$ (Worst-case complexity of DASA/ND is discussed later in this section). Since we can have up to $mn\lceil W/k \rceil$ arrival list size on a processor in the worst-case, the cost of invoking DASA for a single scheduling event is $O\left(m^2n^2\lceil W/k \rceil^2\right)$. Before invoking DASA/ND, the next subtask arrival should be extracted from the heap, which costs $O\left(\log\left(mn\lceil W/k \rceil\right)\right)$. Thus, the sequence of extracting the next arrival from the heap and calling DASA-NP repeats $2mn\lceil W/k \rceil$ times. So, the complexity of the response time analysis is

$$2 \times mn\lceil W/k \rceil \times \left(\log\left(mn\lceil W/k \rceil\right) + m^2n^2\lceil W/k \rceil^2\right) = O\left(m^3n^3\lceil W/k \rceil^3\right).$$

Now, the complexity of *RBA_AnalyzeResponse* is the sum of the cost of the arrival list construction and the cost of the scheduler invocations. That is,

$$O\left(mdn\lceil W/k \rceil + mn\lceil W/k \rceil \log\left(mn\lceil W/k \rceil\right)\right) + O\left(m^3n^3\lceil W/k \rceil^3\right) = O\left(m^3n^3\lceil W/k \rceil^3\right)$$

II. Complexity of *RBA_ DetermineReplicasProcessors*

The procedure *RBA_DetermineReplicasProcessors* determines the number of replicas and processors needed for a given subtask in an iterative manner by starting with a single replica, and incrementing until the maximum possible number of replicas (equal to the number of processors, *p*) is reached. During each iterative step, the procedure invokes *RBA_AnalyzeResponse* a maximum of *p* number of times to determine the response time of the replica (considered in the step) on all *p* processors. Thus, the procedure *RBA_DetermineReplicasProcessors* invokes the procedure *RBA_AnalyzeResponse* p^2 number of times and has a complexity of $p^2 \times O\left(m^3 n^3 \lceil W/k \rceil^3\right)$ which is $O\left(p^2 m^3 n^3 \lceil W/k \rceil^3\right)$.

III. Complexity of *RBA_Algorithm*

The cost of the main procedure, *RBA_Algorithm*, has two components. First, *RBA_Algorithm* constructs a heap that uses task benefit as the key values. The cost of building the heap for *n* tasks is $O(n)$. Second, it invokes *RBA_DetermineReplicasProcessors* for each task subtask and for each period. Recall we have *n* tasks, maximum *m* subtasks per task, and minimum period of *k* for each task. This means that *RBA_DetermineReplicasProcessors* is invoked $mn\lceil W/k \rceil$ times by *RBA_Algorithm*. Before calling *RBA_DetermineReplicasProcessors*, the next highest benefit task needs to be extracted form the heap. The cost of extracting and deleting the task with the maximum benefit from the heap is $O(\log n)$. So cost of the second component is

$$mn\lceil W/k \rceil \times O\left(\log n + p^2 m^3 n^3 \lceil W/k \rceil^3\right) = O\left(p^2 m^4 n^4 \lceil W/k \rceil^4\right)$$

The worst case complexity of *RBA_Algorithm* is the summation of the costs of its components, which is

$$O(n) + O\left(p^2 m^4 n^4 \lceil W/k \rceil^4\right)$$

Thus, the worst-case complexity of RBA* when DASA is used as the underlying scheduling algorithm is asymptotically bounded by $O\left(p^2 m^4 n^4 \lceil W/k \rceil^4\right)$.

IV. Worst-Case Complexity of DASA/ND

Now we analyze the worst-case complexity of DASA/ND, which was used in determining the worst-case complexity of RBA*. Given a list of *r* processes, DASA/ND takes the following steps:

(i) It computes the potential benefit density of each process.

(ii) It sorts the processes according to their potential benefit density.

(iii) It iterates over all the processes. For each iteration, it inserts the process into a tentative schedule and examines the feasibility of the schedule (For more details, please see Appendix A and [Cla90])

The worst-case cost of the first component is $O(r)$. The worst–case cost of the second step, sorting, is $O(r \log r)$. In the third step, inserting an element in the schedule (a linear ordered list) costs $O(r)$ in the worst case. The feasibility check that is done in the third step is done for all the *r* processes in the worst case. So, the feasibility check costs $O(r)$. Since the third step iterates over all the processes and performs feasibility check at each iteration, the worst-case complexity of the third step is $r \times (O(r) + O(r)) = O(r^2)$. Thus, the cost of DASA/ND is the summation of the complexity of each step: $O(r) + O(r \log r) + O(r^2)$, that is $O(r^2)$.

5.4.2 Amortized Complexity of *RBA_AnalyzeResponse*

We now analyze the amortized complexity of the *RBA_AnalyzeResponse* procedure, since it is the most computationally intensive procedure in the RBA* algorithm.

Amortized complexity is considered since it generally provides a "tighter" upper bound than the worst-case complexity. The *RBA_AnalyzeResponse* procedure invokes another procedure called *LocalScheduler* (See Figure 5.3), which represents the underlying scheduling algorithm that selects a process from the process ready queue. In the following subsections, we discuss the complexity of the DASA/ND [Cla90] as the local scheduling algorithm. Then, we analyze the amortized complexity of *RBA_analyzeResponse*.

I. Total Cost of DASA/ND

Given a list of *r* processes, DASA/ND takes the following steps:

(iv) It computes the potential benefit density of each process.

(v) It sorts the processes according to their potential benefit density.

(vi) It iterates over all the processes. For each iteration, it inserts the process into a tentative schedule and examines the feasibility of the schedule (For more details, please see Appendix A and [Cla90])

Step (i) costs $O(r)$, since it computes the potential benefit density for all processes through one pass. Step (ii), sorting, costs $O(r \log r)$. Step (iii) costs $\sum_{k=1}^{r}(k+k)$ steps, where for the k^{th} process, it costs $(k-1)$ steps to insert the process in the schedule (implemented as a linear sorted list) in the worst case. Checking the feasibility, after inserting the k^{th} process, costs *k* steps, since it can be done by making a single pass over the schedule.

It is clear that step (iii) dominates because it yields a cost of $O(r^2)$, while steps (i) and (ii) cost $O(r)$ and $O(r \log r)$, respectively. So, the total cost of DASA/ND is $O(r^2)$.

II. Amortized Complexity of *RBA_analyzeResponse*

As mentioned in Section 5.4.2, the cost of *RBA_analyzeResponse* consists of two components: constructing the heap with subtask arrival times as key values and analyzing subtask response times. Given, *N* subtask arrivals, the total number of steps of the first component is $\sum_{k=1}^{N} \log k$ steps, because it costs *O(*$\log k$*)* to insert the k^{th} element in the heap.

For the response time analysis, DASA/ND is called *2N* times. The worst case happens when none of the *N* processes terminates until all of them arrive. In this case, the total number of steps is $\sum_{k=1}^{N}\left(\log\left(N-k\right)+k^2\right)+\sum_{k=N-1}^{1} k^2$, where k^2 is the cost of invoking DASA/ND for *k* processes (Section 5.4.2), $\log\left(N-k\right)$ is the cost of extracting the k^{th} element from the heap. $\sum_{k=1}^{N}\left(\log\left(N-k\right)+k^2\right)$ is the cost of extracting the k^{th} element and then calling DASA/ND, with no process leaving the ready queue until the *N* processes arrive. Every time a process arrives, the queue size increases until it becomes *N*. Then, the first termination occurs. At that time, DASA/ND will be invoked for *N* processes in the ready queue. Then, the number of processes in the queue will decrease until the queue is empty. The cost of invoking DASA/ND at each of the terminations is $\sum_{k=N}^{1} k^2$ steps.

Thus, the amortized complexity of *RBA_analyzeResponse* is

$$\frac{\text{Total Number of Steps}}{\text{N}} = \frac{\sum_{k=1}^{N}\log k+\left(\sum_{k=1}^{N}\left(\log\left(N-k\right)+k^2\right)+\sum_{k=N}^{1} k^2\right)}{N}$$

Note that the dominant term in the numerator is $\sum_{k=1}^{N} k^2$ which is $O\left(N^3\right)$. So, The amortized complexity of *RBA_analyzeResponse* is $O\left(N^3\right)\big/_{N} = O\left(N^2\right)$.

5.5 Summary

The RBA* algorithm was developed to solve the problem presented in Chapter 4. The problem specifies an application, adaptation, and system models as described in Chapter 3. The problem assumes that the anticipated load is expressed using adaptation functions. The problem require a solution that determines the resource allocation in terms of number of replicas for each task subtask during all task executions and the processor assignment of those replicas. The resulting resource allocation should lead to the maximum aggregate benefit and the minimum aggregate missed deadlines ratio. As the problem was found to be NP-complete (Section 4.2), we developed the heuristic algorithm RBA* for determining such resource allocation. In this chapter, we introduced the heuristics of the RBA* algorithm and the rationale behind each heuristic. The algorithm mainly adopted the following heuristics to satisfy the objective of maximizing the aggregate benefit:

- Allocate resources in the decreasing order of task benefits
- De-allocate resources for a task if its deadline cannot be satisfied
- Not allow a lower benefit task affect the timeliness of a higher benefit task

For satisfying the objective of minimizing the aggregate missed deadlines ratio, the algorithm mainly adopted the following heuristics:

- Allocate resources for each task until its deadline is satisfied
- Assign deadlines to subtasks and messages of the task from the task deadline in such a way that if all subtasks and messages of the task can meet their respective deadlines, then the task will able to meet its deadline.

The rationale behind each heuristic was presented in this chapter. As presented in this chapter, RBA* (using the heuristics stated above) computes (1) the number of replicas of each subtask for each task execution, and (2) the processor assignment needed to run such replicas. So, for the given application, adaptation, and system models, and for a given adaptation functions that describe the anticipated load of all the tasks in the system within a future time window, running RBA* results in a resource allocation that will lead to an aggregate benefit that is close to the maximum and aggregate missed deadlines ratio that is close to the minimum.

The worst-case complexity of RBA* was analyzed and determined to be $O\left(p^2 m^4 n^4 \lceil W/k \rceil^4\right)$, where n denotes the number of tasks in the application, p denotes the number of processors in the system, m denotes the maximum number of subtasks of a task (thus in the worst-case, all n tasks will have m subtasks), k denotes the smallest task period among all tasks (thus in the worst-case, all n tasks will have a period k), and W denotes the length of the adaptation function. Since the procedure *RBA_AnalyzeResponse* was found to be the most computationally expensive component of the algorithm, the amortized complexity of such component was analyzed and determined to be $O\left(N^2\right)$. This was done for comparison purposes, as this algorithm will be compared to the other heuristic algorithm presented in the next chapter.

Chapter 6: The OBA Algorithm: Heuristics and Rationale, and Complexity Analysis

The objectives of the OBA algorithm include all the objectives of the RBA* algorithm i.e., to maximize aggregate application benefit and minimize aggregate missed deadline ratio. Thus, the desired properties of OBA include all that of RBA* discussed in the previous chapter. Thus, OBA employs the same heuristics as that employed by RBA* and the rationale behind the heuristics also holds for OBA. However, having determined the worst-case complexity of RBA*, which is a polynomial of the 4^{th} degree, we now would like to design OBA such that it achieves the objectives much faster than RBA*.

6.1 Feasibility Test Speedup

A careful observation of the RBA* reveals that the procedure that costs the algorithm the most is the "heart" of the algorithm, namely, *RBA_AnalyzeResponse*. All other properties of the algorithm such as (1) allocating replica resources to tasks in the decreasing order of task benefits, (2) allocating resources for each task until its deadline is satisfied, (3) de-allocating resources for a task if it's deadline cannot be satisfied, and (4) determining the number of replicas needed for each task subtask that will satisfy the task deadline by determining the number of replicas needed for each subtask that will satisfy the subtask deadline are desirable features of OBA as well.

Recall that the procedure *RBA_AnalyzeResponse* analyzes the response time of a subtask on a given processor and for a given workload by constructing an arrival list for all subtasks on the processor and invoking the scheduling algorithm for each subtask arrival and completion within the length of the adaptation function. Here, the complexity of the procedure is dominated by the complexity of invoking the scheduling algorithm for all the scheduling events i.e., $O\left(m^3 n^3 \lceil W/k \rceil^3\right)$.

We can speed up the *RBA_AnalyzeResponse* procedure by avoiding the execution of the scheduling algorithm and doing an *overload test* on the processor instead. Note that the objective of invoking the scheduling algorithm is to determine the subtask feasibility (which is done by determining the subtask response time and comparing the response time against the subtask deadline.) We can also determine the subtask feasibility by doing an overload test on the processor. If the processor is under-loaded, then clearly, the subtask must be able to complete by it's deadline as we are considering best-effort scheduling algorithms that "mimic" EDF (Earliest Deadline First real-time scheduling algorithm) during under-loaded situations, where EDF guarantees all deadlines. So if the processor is under-loaded, we can conclude that the processor is a candidate for the subtask. If, on the other hand, the processor is overloaded (one or more subtasks will miss the deadline using EDF), we may then reduce the workload share of the subtask by considering another replica for the subtask. The subtask feasibility can then again be determined through the overload test and the whole procedure can be repeated in a way similar to that of RBA*.

Given *N* subtask arrivals on a processor that are deadline-ordered, we can perform the overload test in $O(N)$ time [Cla90, BS95]. Thus, the cost of performing the overload test on a processor, given $mn\lceil W/k \rceil$ subtask arrivals on the processor in the worst-case is given by $O(mn\lceil W/k \rceil)$, assuming that we are given a deadline-ordered subtask arrival list. Recall from the previous chapter that the complexity of *RBA_AnalyzeResponse* has two components: the arrival list construction and the response time analysis. Thus, the complexity of OBA's counterpart of *RBA_AnalyzeResponse* becomes equal to the sum of the cost of constructing the deadline-ordered subtask list and the cost of performing the overload test.

We can easily modify the procedure *RBA_AnalyzeResponse* so that it constructs the subtask arrival list that is ordered by subtask deadlines instead of arrival times at a cost of $O(mn\lceil W/k \rceil \log(mn\lceil W/k \rceil))$ using heaps (details of OBA's complexity are presented later in this chapter). Therefore, the total cost of OBA's version of the

RBA_AnalyzeResponse procedure becomes $O\left(mn\lceil W/k\rceil \log\left(mn\lceil W/k\rceil\right)\right)$. This cost will significantly speed up OBA with respect to RBA*, which had a cost of $O\left(m^3n^3\lceil W/k\rceil^3\right)$ for the procedure *RBA_AnalyzeResponse*[3] when DASA is used as the underlying scheduling algorithms at all local processors.

6.2 The OBA Algorithm

Thus, at the highest level of abstraction, OBA follows the exact same steps as RBA*. The pseudo-code of OBA at the highest level of abstraction is shown in Figure 6.1. The difference comes in determining the number of replicas needed for each task subtask and their processor assignment.

We now discuss how OBA performs the overload test in Section 6.2.1. Section 6.2.2 describes how OBA determines the number of subtask replicas and their processor assignment.

```
OBA_Algorithm(T,W) /* T is the task set and W is the future time window */
1. Construct a heap of tasks using task benefit as the key value;
2. For each task i = 1 to |T|
    2.1 Extract the task T_i from the heap; /* tasks will be considered in decreasing-benefit order */
    2.2 For each period j = 1 to ⌈W/period(T_i)⌉
        2.2.1 For each subtask k = 1 to |ST(T_i)|
                OBA_DetermineReplicasAndProcessors(st_k^i, j, Adapt(T_i, j));
```

Figure 6.1. Pseudo-code of the OBA Algorithm at the Highest Level of Abstraction

6.2.1 Overload Analysis

To check whether the presence of a subtask will result in an overload, the algorithm first constructs a list of subtask arrival times similar to the one constructed by RBA*'s *RBA_AnalyzeResponse* procedure (Section 5.2.2), except that the list is deadline-ordered. As discussed in Section 5.2, the algorithm constructs a deadline-ordered list by inserting

[3] Later, in this chapter, we provide a detailed analysis of the complexity of OBA.

a subtask arrival event into an integer-ordered list at the subtask deadline position, once it determines the arrival time of a subtask.

Once the deadline-ordered arrival list is constructed, the algorithm examines the subtask deadlines in the arrival list in decreasing order of deadlines. For each subtask deadline d_i, the algorithm computes the sum of the remaining execution times of all subtasks having deadlines less than d_i and compares the sum against d_i. If the sum is greater than the deadline d_i for any deadline, then there exists an overload on the processor, as it indicates that there exists at least one subtask on the processor that is unable to complete before its deadline. If this sum is less than the subtask deadline for each deadline, then the processor is considered to be running an under-load situation.

Figure 6.2 shows a pseudo-code of OBA's overload test procedure called *OBA_OverloadCheck* that checks whether executing a subtask replica *s* on a processor *q* during the subtask period *p* will cause an overload situation. The procedure starts by constructing a list of subtask arrival the same way *RBA_AnalyzeResponse* constructs its arrival list. After the list is constructed, the overload check is run in a single pass on the list. The procedure returns a *SUCCESS* value if no overload is detected. Otherwise, it returns a *FAILURE* value.

6.2.2 Determining Number of Subtask Replicas and Their Processors

OBA determines the number of subtask replicas and their processor assignment as follows:

Let the subtask st_q^i be considered for replication, which currently has a single replica, denoted as, $st_{q,0}^i$. The algorithm first checks whether there is an overload on the processor where the replica is assigned. If no overload is detected, the algorithm concludes that the single replica $st_{q,0}^i$ can process the entire subtask workload on the processor that was considered, complete it's execution before the subtask deadline (since no overload is detected on the processor, all subtasks must be able to compete by their deadlines), and

thus cannot affect the timeliness of the higher benefit tasks[4]. Thus, by detecting an under-load on a processor, OBA makes the same conclusions that are made by RBA* regarding subtask feasibility and timeliness of higher benefit tasks. So, if a subtask execution on a processor is found to negatively affect the timeliness of another subtask that belongs to a higher benefit task, then running that subtask on that processor under the current load is not feasible.

If an overload is detected on the processor of $st^i_{q,0}$, the algorithm reduces the data size load of the subtask by replication. The algorithm considers a second replica for the subtask, denoted as $st^i_{q,1}$, on a processor that does not have the existing subtask replica $st^i_{q,0}$ assigned to it. Note that by considering a second replica for the subtask, we reduce the workload share of each of the two subtask replicas, thereby reducing the execution time of the subtask replicas. This may resolve the overload situation on the processors where a subtask replica is assigned. The algorithm now tests for overload on the processors of (both) $st^i_{q,1}$ and $st^i_{q,0}$. If no overload is detected on the processors of $st^i_{q,1}$ and $st^i_{q,0}$, the algorithm concludes that two replicas are sufficient to satisfy the subtask deadline and proceeds to the next subtask. Otherwise, OBA considers yet another replica for the subtask.

The algorithm thus, repeats the process of replicating and overload testing until either (1) no overload is detected on any of the processors of the subtask replicas or (2) the maximum possible number of replicas for the subtask (equal to the number of processors in the system, for exploiting maximum concurrency) is reached. If OBA determines that executing the maximum number of replicas for a subtask does not resolve the overload situation and thus does not satisfy the subtask deadline, it de-allocates the task as discussed in Chapter 5.

[4] Note that whenever we consider the execution of a subtask replica *s* on a processor *q* and test for overload on *q*, all existing subtasks on *q* must belong to higher benefit tasks than the task of *s*, since OBA allocates replicas to tasks in decreasing order of their benefits.

Figure 6.3 shows the pseudo-code of the procedure *OBA_DetermineReplicasProcessors* that determines the number of replicas necessary for each subtask and their processors. This procedure calls the procedure *OBA_OverloadCheck* (Figure 6.2) to test processor overloads during the resource allocation process. Recall that the main procedure of the OBA algorithm, *OBA_Algorithm* (Figure 6.1) invokes the procedure *OBA_DetermineReplicasProcessors* for each subtask execution during the future time window.

OBA_OverloadCheck(s, p, q, l) /* s is the subtask, p is the period number, q is the processor, l is the workload */

1. $ArrivalEvents[\ ] = \varnothing$; /* List $ArrivalEvents[\ ]$ has 3 fields: time, subtask, remaining_time */
2. $x = ParentTaskNumber(s)$; $y = SubtaskNumber(s, x)$; /* y is subtask # of s within its parent task x */
3. If $type(s) = PERIODIC$ /* $type(s)$ returns PERIODIC if s is periodic; otherwise returns APERIODIC */
   - 3.1 $ArrivalTime(s) = (p-1) \times period(T_x) + \sum_{\forall r: r<y} \left(ResponseTime(st_r^x) + dl(m_r^x) \right)$;
   - 3.2 $StopTime = p \times period(T_x)$;
4. Else /* s is aperiodic */
   - 4.1 $ArrivalTime(s) = p \times period(T_x) + \sum_{i<y} \left(ResponseTime(st_i^x) + dl(m_i^x) \right)$;
   - 4.2 $StopTime = (p+1) \times period(T_x)$;
5. $ArrivalEvents = ArrivalEvents \cup \left[st_y^x, ArrivalTime(s), eex(s, l) \right]$;
6. For each task $i = 1$ to $x-1$ /* consider all tasks that have higher benefits */
   - 6.1 For each period $j = 1$ to $\lceil StopTime / period(T_i) \rceil$
     - 6.1.1 For each subtask $k = 1$ to $|ST(T_i)|$
       - If subtask st_k^i does not have a replica executing on processor q during period j, skip current iteration
         1. If $type(st_k^i) = PERIODIC$
            $ArrivalTime(st_k^i) = p \times period(T_i) + \sum_{\forall r: r<k} \left(ResponseTime(st_r^i) + dl(m_r^i) \right)$;
         2. Else /* st_k^i is *APERIODIC* */
            $ArrivalTime(st_k^i) = (p+1) \times period(T_i) + \sum_{\forall r: r<k} \left(ResponseTime(st_r^i) + dl(m_r^i) \right)$;
         3. $ArrivalEvents = ArrivalEvents \cup \left[st_k^i, ArrivalTime(st_k^i), eex(st_k^i, l) \right]$;

            /* the arrival time of subtask st_k^i is inserted at the $adl(st_k^i)^{th}$ position in *ArrivalEvents*; where $adl(st_k^i)$ is the absolute deadline of st_k^i */
7. $Sum = 0$;
8. For each subtask arrival event $k \in ArrivalEvents[\ ]$
   - 8.1 $Sum = Sum + ArrivalEvents[k].remaining\_time$;
   - 8.2 If $Sum > dl(ArrivalEvents[k].subtask)$
     - return *OVERLOAD*;
9. return *SUCCESS*;

Figure 6.2. The *OBA_OverloadCheck* Procedure

OBA_DetermineReplicasProcessors(s,i,l) /* s is the subtask, i is the period, and l is the anticipated load */

1. For each processor $q \in pr(st)$

    1.1 If *OBA_OverloadCheck*$\left(s,i,q,l/\left|pr(s)\right|\right) = OVERLOAD$

        Goto to step 2;

    1.2 Else

        Return *SUCCESS*;

2. If $\left|pr(s)\right| = MaximumReplicas$

    Return *FAILURE*;

3. $PT = PR - pr(s)$, $G = \{r\}$;

/* $pr(s)$ is the set of processors that are executing replicas of s; PT is thus the set of processors that are not executing replicas of s, r is any element in PT */

4. For each processor $q \in PT$

    4.1 If *OBA_OverloadCheck*$\left(s,i,q,l/\left(\left|pr(s)\right|+1\right)\right) = SUCCESS$

        /* $\left(\left|pr(s)\right|+1\right)$ is the number of replicas after increasing it by one; $l/\left(\left|pr(s)\right|+1\right)$ is the new load */

        4.1.1 $G = \{q\}$;

        4.1.2 Goto Step 6;

5. $PT = PT - G$;

6. $pr(s) = pr(s) \cup G$;

7. Goto Step 1;

Figure 6.3. The *OBA_DetermineReplicasProcessors* Procedure

6.3 The Complexity of OBA

In section 6.3.1, we analyze the worst-case complexity of RBA*. In section 6.3.2, the amortized complexity of the *OBA_OverloadCheck* procedure is analyzed.

6.3.1 Worst-Case Complexity of OBA

The analysis of the worst-case computational complexity of OBA is similar to that of RBA*: OBA's complexity depends upon the complexity of the procedure *OBA_DetermineReplicasProcessors*, the complexity of which depends on the procedure *OBA_OverloadCheck*.

As discussed in Section 6.2.1, the complexity of the procedure *OBA_OverloadCheck* is equal to the sum of the cost of constructing the heap using subtask deadlines as key

values and the cost of performing the overload test. The cost for constructing the subtask deadline heap is $O\left(mn\lceil W/k\rceil \log\left(mn\lceil W/k\rceil\right)\right)$, since we use the same approach used by procedure *RBA_AnalyzeResponse* as discussed in the previous chapter. Note that the term $mdn\lceil W/k\rceil$ does not appear here, because the algorithm does not need to compute the arrival times of the subtasks. It only needs the absolute deadlines to perform the overload test.

Given the deadline heap, we test for overload by making a single pass over the list. Each subtask deadline is examined in their increasing order and the cumulative sum of the remaining execution times of all subtasks with lesser deadlines is compared to the current deadline. It costs $\log\left(mn\lceil W/k\rceil\right)$ to extract and delete the earliest deadline subtask from the heap. Since this process is repeated for all the $mn\lceil W/k\rceil$ nodes of the heap, the cost of the overload test is $O\left(mn\lceil W/k\rceil \log\left(mn\lceil W/k\rceil\right)\right)$ Thus, the total cost of *OBA_OverloadCheck* is given by

$$O\left(mn\lceil W/k\rceil \log\left(mn\lceil W/k\rceil\right) + mn\lceil W/k\rceil \log\left(mn\lceil W/k\rceil\right)\right) = O\left(mn\lceil W/k\rceil \log\left(mn\lceil W/k\rceil\right)\right)$$

.

The procedure *OBA_DetermineReplicasProcessors* determines the number of replicas and processors needed for a given subtask in an iterative manner by starting with a single replica, and incrementing until the maximum possible number of replicas (equal to the number of processors, *p*) is reached. During each iterative step, the procedure invokes *OBA_OverloadCheck* a maximum of *p* number of times to test for overload on all *p* processors for the replica considered in the step. Thus, the procedure *OBA_DetermineReplicasProcessors* invokes the procedure *OBA_OverloadCheck* p^2 number of times and has a complexity of:

$$p^2 \times O\left(mn\lceil W/k\rceil \log\left(mn\lceil W/k\rceil\right)\right) = O\left(p^2 mn\lceil W/k\rceil \log\left(mn\lceil W/k\rceil\right)\right).$$

The cost of the main procedure, *OBA_Algorithm*, has two components. First, O*BA_Algorithm* constructs a heap using the task benefit as the key values. The cost of building the heap for *n* tasks is $O(n)$. Second, it invokes *OBA_DetermineReplicasProcessors* for each task subtask and for each period. Recall we have *n* tasks, maximum *m* subtasks per task, and minimum period of *k* for each task. This means that *RBA_DetermineReplicasProcessors* is invoked $mn\lceil W/k \rceil$ times by *RBA_Algorithm*. Before calling *OBA_DetermineReplicasProcessors*, the next highest benefit task needs to be extracted form the heap. The cost of extracting and deleting the task with maximum benefit from the heap is $O(\log n)$. So cost of the second component is given by:

$$mn\lceil W/k \rceil \times O\left(\log n + p^2 mn\lceil W/k \rceil \log\left(mn\lceil W/k \rceil\right)\right) = O\left(p^2 m^2 n^2 \lceil W/k \rceil^2 \log\left(mn\lceil W/k \rceil\right)\right)$$

The worst case complexity of *OBA_Algorithm* is the sum of the costs of its components, which is $O(n) + O\left(p^2 m^2 n^2 \lceil W/k \rceil^2 \log\left(mn\lceil W/k \rceil\right)\right)$. Thus, the worst-case complexity of OBA is asymptotically bounded by $O\left(p^2 m^2 n^2 \lceil W/k \rceil^2 \log\left(mn\lceil W/k \rceil\right)\right)$.

6.3.2 Amortized Complexity of *OBA_OverloadCheck*

In this section we consider the amortized complexity of *OBA_OverloadCheck* since it represents the main difference between OBA and RBA*. Recall that the corresponding procedure in RBA* is called *RBA_AnalyzeResponse*, which was shown to be computationally intensive (Chapter 5). In this section we show that *OBA_OverloadCheck* is faster than *RBA_AnalyzeResponse*.

The cost of *OBA_OverloadCheck* has two components. The first component corresponds to constructing the deadline heap and the second component is for the overload check. Given *N* subtask arrivals, the total number of steps for constructing the deadline heap

is $\sum_{k=1}^{N} \log k$. (This is the same as constructing the arrival list in *RBA_AnalyzeResponse* in Section 5.4.1.) The overload check then takes a total number of *N* iterations. At each iteration, the next subtask needs to be extracted from the heap, which costs $O\left(\log\left(N-k\right)\right)$. So, the overload check component costs $\sum_{k=1}^{N} \log\left(N-k\right)$.

The amortized complexity of *OBA_OverloadCheck* is

$$\frac{\text{Total Number of Steps}}{N} = \frac{\sum_{k=1}^{N} \log k + \sum_{k=N}^{1} \log\left(N-k\right)}{N}$$

Note that both terms in the numerator yield $O\left(N \log N\right)$. So, the amortized complexity of *OBA_OverloadCheack* is $O\left(N \log N\right)/N = O\left(\log N\right)$.

6.4 Summary

The OBA algorithm was developed to solve the problem presented in Chapter 4. The problem specifies an application, adaptation, and system models as described in Chapter 3. The problem assumes that the anticipated load is expressed using adaptation functions. Required is a solution that determines the resource allocation in terms of number of replicas for each task subtask during all task executions and the processor assignment of those replicas. The resulting resource allocation should lead to the maximum aggregate benefit and the minimum aggregate missed deadlines ratio. As the problem was found to be NP-complete (Section 4.2), we developed the heuristic algorithm OBA for determining such resource allocation. In this chapter, we introduced the heuristics of the OBA algorithm and the rationale behind each heuristic. The algorithm has one more objective, which is to compute the resource allocation decisions faster than RBA* that is relatively computationally expensive. The algorithm mainly adopted the following heuristics to satisfy the objective of maximizing the aggregate benefit:

- Allocate resources in the decreasing order of task benefits
- De-allocate resources for a task if its deadline cannot be satisfied

- Not allow a lower benefit task affect the timeliness of a higher benefit task

For satisfying the objective of minimizing the aggregate missed deadlines ratio, the algorithm mainly adopted the following heuristics:

- Allocate resources for each task until its deadline is satisfied
- Assign deadlines to subtasks and messages of the task from the task deadline in such a way that if all subtasks and messages of the task can meet their respective deadlines, then the task will able to meet its deadline.

The rationale behind each heuristic was presented in this chapter. To achieve the objective of speeding up the resource allocation decision, the algorithm adopted the concept of overload check as a feasibility check (as opposed to response time analysis adopted by OBA) As presented in this chapter, OBA (using the heuristics stated above) computes (1) the number of replicas of each subtask for each task execution, and (2) the processor assignment needed to run such replicas. So, for the given application, adaptation, and system models, and for a given adaptation functions that describe the anticipated load of all the tasks in the system within a future time window, running RBA* results in a resource allocation that will lead to an aggregate benefit that is close to the maximum and aggregate missed deadlines ratio that is close to the minimum.

The worst-case complexity of OBA was analyzed and determined to be $O\left(p^2m^2n^2\lceil W/k\rceil^2\log\left(mn\lceil W/k\rceil\right)\right)$, where n denotes the number of tasks in the application, p denotes the number of processors in the system, m denotes the maximum number of subtasks of a task (thus in the worst-case, all n tasks will have m subtasks), k denotes the smallest task period among all tasks (thus in the worst-case, all n tasks will have a period k), and W denotes the length of the adaptation function. This means that the algorithm achieved the objective of making resource allocation decisions is a shorter time. The amortized complexity of *OBA_OverloadCheck* (which corresponds to the RBA*'s *RBA_AnalyzeResponse*) component was analyzed and determined to be $O(\log N)$. This is faster than the amortized complexity of *RBA_AnalyzeResponse* that is $O(N^2)$.

Chapter 7: Experimental Evaluation of RBA* and OBA

In this chapter, we experimentally evaluate the performance of RBA* and OBA through a set of simulation experiments under several best-effort real-time scheduling algorithms. In Section 7.1, we present the baseline parameters of simulation. In Section 7.2, we present the experimental results and the reasoning behind them.

7.1 Baseline Parameters

Table 7.1 summarizes the baseline parameters of our experimental set up. The baseline parameters are derived from the real-time benchmark that has resulted from our past work [WS99]. The software architecture of the periodic and aperiodic tasks that we have used corresponds directly to that of the benchmark.

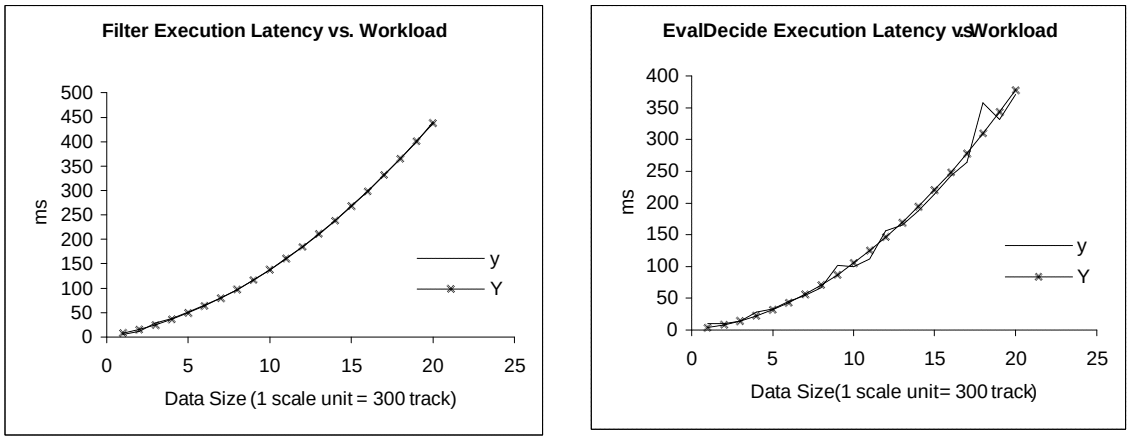
Table 7.1. Baseline Parameters

Number of periodic Tasks	10
Number of aperiodic Tasks	10
Number of nodes	15
Maximum number of replicas for each subtask	10
Range of number of subtasks per task	1-8
Size of a data item (track)	80 bytes
Range of periods lengths of periodic tasks	200 ms — 4 sec
Relative end-to-end deadline of periodic and aperiodic tasks	same as corresponding periods
Range of task absolute benefits	200 — 4500
Task adaptation functions	constant, increasing ramp

Recall that the algorithms assume that offline profiles of the subtasks are available as described in Section 3.1.3. Those profiles are used by the resource allocation algorithms (RBA* and OBA) to estimate the subtask execution times given the size of the workload that the subtask has to process. To determine the function $eex(st_j^i, d)$ that can provide a reasonable approximation of subtask execution latencies as a function of the workload data size, we measure the execution time of the subtasks of the benchmark for a set of data sizes without any interference from other application subtasks. The measurements

are then used to define a regression equation that computes execution latency as a function of subtask data size.

Figures 7.1(a) and 7.1(b) show sample plots of execution latency measurements of two application subtasks of the benchmark called *Filter* and *EvalDecide* [WS99] under Redhat Linux 6.2. The solid lines (called "y" in the figure) show the observed execution time and the dashed lines (called "Y" in the figure) show the second order curve that is computed from the observed execution times. This second order curve is defined by the second order regression equation $C_i^j(d) = a_i^j d^2 + b_i^j d$, where C_j^i is the execution time of the subtask st_j^i, d is the data size in hundreds of data items, and a_i^j and b_i^j are constants that are dependent upon the application subtask.



(a) **(b)**

Figure 7.1. Execution Latency Profiles of (a) *Filter* and (b) *EvalDecide* Subtasks Under Varying Data Sizes with no Contention

The coefficients of the regression equation for five application subtasks of the benchmark are shown in Table 7.2, where *d* is measured in hundreds of data items (tracks), and the execution time is given in milliseconds.

Table 7.2. Coefficients of the Execution Latency Regression Equation

Subtask	*a*	*b*
1	.009655706	1.23350748
2	0.003556645	0.6108486
3	0.090740954	1.651962884
4	.002994002	0.245154102
5	0.093582979	0.484474288

7.2 Experimental Results

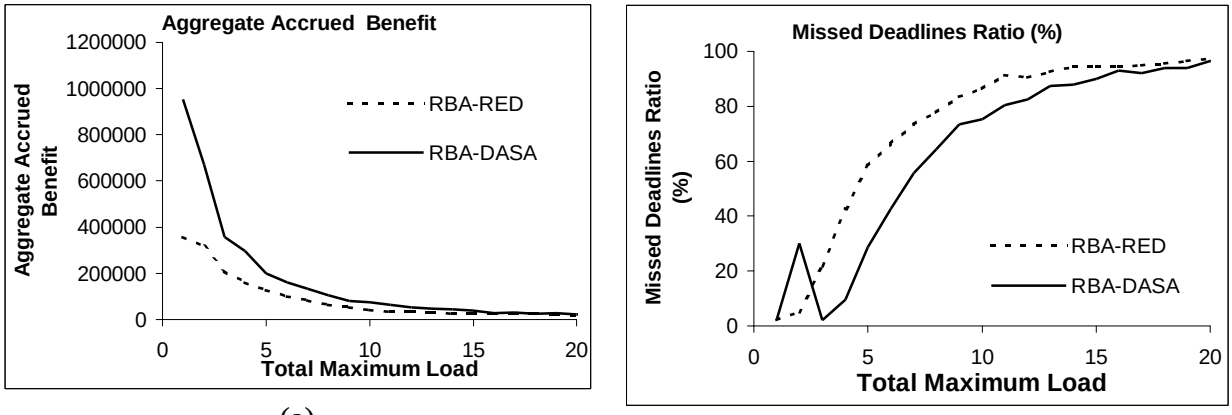
7.2.1 Performance of RBA* under Different Scheduling Algorithms

In this section, we provide experimental evaluation for RBA* under different best-effort scheduling algorithms. Simulation experiments were performed using DASA, LBESA, RED, and RHD as the local scheduling algorithms on each of the processors in the distributed system. In each case, the function *LocalScheduler* used by RBA* (shown in Figure 5.3, line 8.1.5) was used to mimic the local scheduling algorithm. First, RBA* was run to determine the number of subtask replicas and the processor assignment for each subtask execution during the future time window defined in the baseline parameters. Second, the set of tasks were run in the system (during the time interval defined by the future time window) given the allocation that was determined by RBA*. The aggregate benefit was computed over this time interval. To compute the aggregate benefit by adding the benefit $B(T_i)$ of a task T_i whenever the task completes its current execution by the deadline and by adding zero if it does not complete by the deadline. This is because rectangular benefit functions were used to express the timeliness of the tasks as described in Chapter 3.

Figure 7.2(a) shows the aggregate benefit accrued by RBA* under DASA and under RED as the local scheduling algorithms when the workload of all tasks given by the task adaptation functions increase linearly. In other words, the task adaptation functions are increasing "ramp" functions. We use ramp functions to test the performance of the algorithm under increasing loads that lead to overload situations. Note that asynchronous

systems are prone to overload situations due to the lack of known upper bounds on the workloads. Each point on the plot was obtained by conducting a complete experiment as described in the previous paragraph. All the experiments have the same adaptation window size. However, they differ in the slope of the ramp function and hence in the maximum load. Figure 7.2(b) shows the corresponding missed deadlines ratio. The metrics are shown as a function of system load, where the system load is defined as the total maximum data items processed by all the periodic and aperiodic tasks in the system. The load is measured in terms of tracks in the case of periodic tasks and in terms of events in the case of aperiodic tasks.

Figure 7.3 shows the performance of RBA* under DASA and under RED when the load of all periodic tasks remains a constant and the load of all aperiodic tasks increase linearly over the time window. We consider such load pattern to focus on the performance of aperiodic tasks when they experience increasing loads that lead to overload situations.



(a) (b)

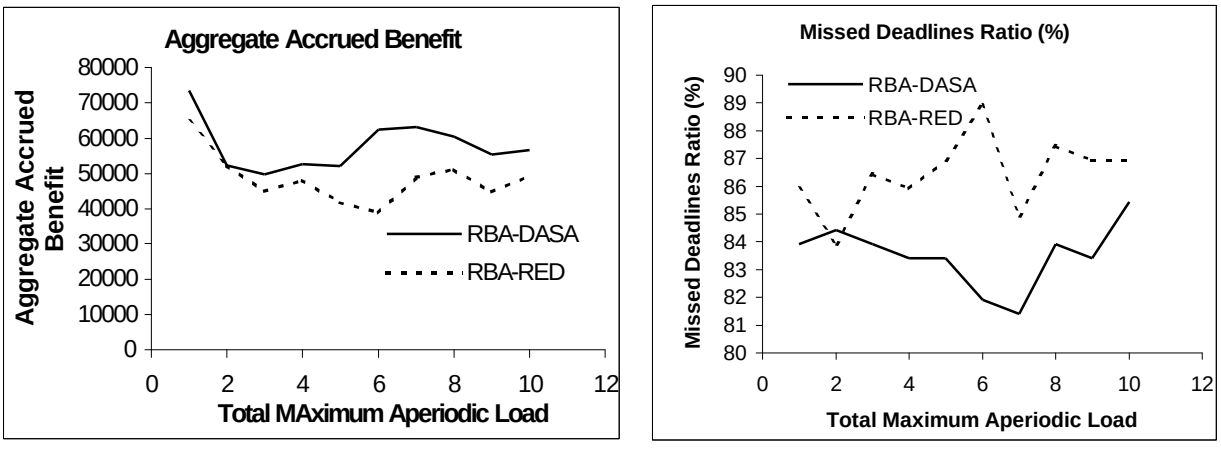
Figure 7.2. Performance of RBA* under DASA and RED For Increasing Ramp Adaptation Functions: (a) Aggregate Benefit Accrued and (b) Aggregate Missed Deadline Ratio

The same experiments described above were conducted with LBESA and RHD as the local scheduling algorithms. We observed that the performance of RBA* under LBESA

and under RHD is very close to that under RED for the load situations considered in the Figures 7.2 and 7.3. Therefore, for clarity, we omit the performance of RBA* under LBESA and under RHD from the figures.

From the Figures 7.2 and 7.3, we observe that RBA* under DASA produce higher aggregate benefit and lower missed deadline ratio than that under RED, LBESA, and RHD. Thus, the experimental results illustrate the superiority of RBA* under the DASA best-effort scheduler. This is believed to be due to two main reasons:

- RBA* exploits the process scheduling algorithm to allocate resources, at the same time DASA as a local scheduler is known to be superior among several other best-effort scheduling algorithms. For example, DASA performs better than LBESA in most cases [Cla90]. Thus, the performances of the local schedulers are expected to be "transferred" to the performance of RBA*.
- RBA* mimics DASA at the high level of the resource allocation. For example, RBA* allocates resources and tests the feasibility of the tasks in the decreasing benefit order. DASA also tests the feasibility of phases in a decreasing benefit density order (see Appendix A and [Cla90].)



(a) (b)

Figure 7.3. Performance of RBA* under DASA and RED for Fixed Periodic Load and Increasing Aperiodic Load: a) Aggregate Accrued Benefit and (b) Missed Deadline Ratio

We also studied the performance of the algorithms during situations when the actual workload is different from the anticipated workload given by the adaptation functions. To characterize the algorithm's performance during such situations, we define a *relative load error* term as follows:

$$e_r = \frac{(actual\ load - anticipated\ load)}{anticipated\ \ load}$$

Figure 7.4 shows the performance of the algorithm under a range of relative load errors from –0.9 to +0.9, under a fixed anticipated workload. A load error of 0.9 means that the actual load is 190% of the anticipated load. The y-axis shows the *relative change in aggregate benefit*. We define the *change in aggregate benefit* for a certain value of e_r as the difference between the aggregate benefit under this value of e_r and the aggregate benefit under zero relative load error. The *relative* change in aggregate benefit is defined as the ratio between the change in aggregate benefit and the aggregate benefit under zero relative load error. Figure 7.4 shows that RBA* generally performs better under error when DASA is used as the underlying scheduling algorithm than when RED is used.

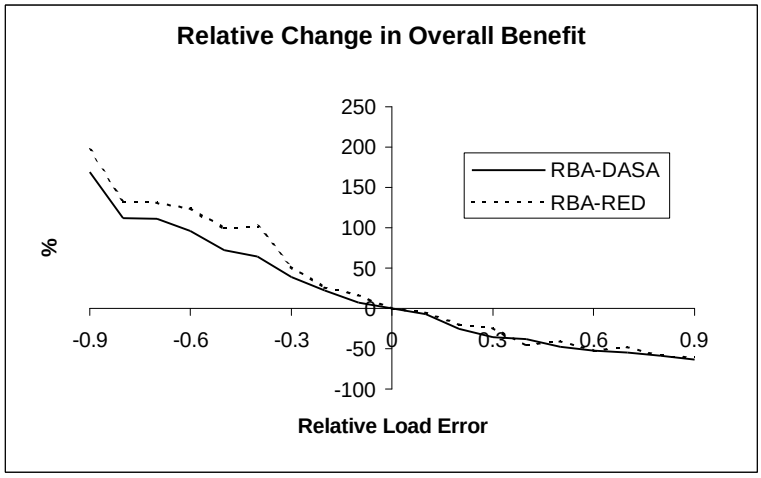
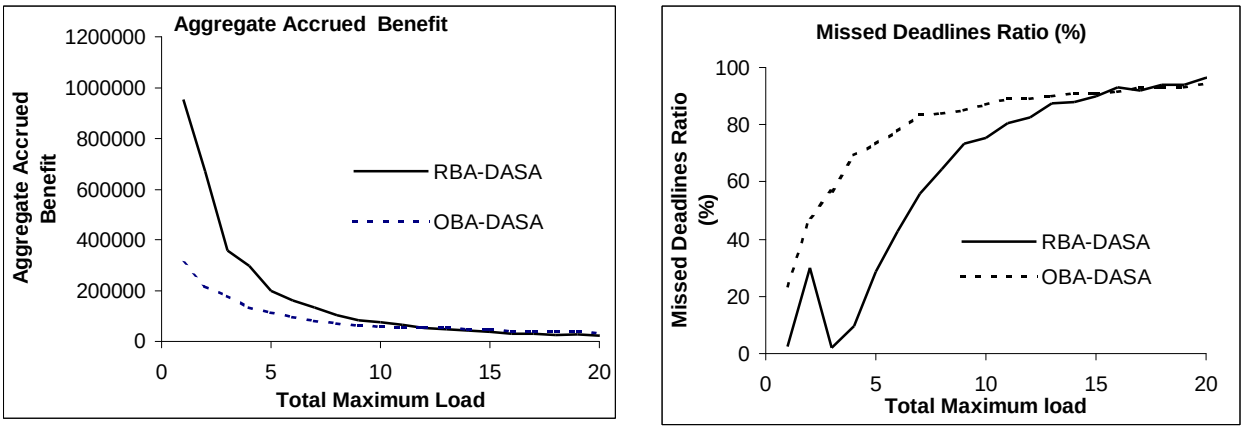


Figure 7.4. Effect of Error in Anticipated Load on RBA*'s Performance

7.2.2 Performance of OBA versus RBA*

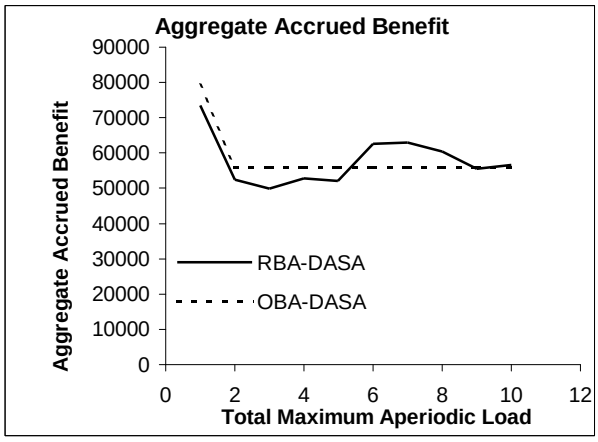
Because DASA performed the best among the four best-effort scheduling algorithms we considered, we compare the performance of RBA* and OBA only under DASA. The same experiments that yielded the results shown in Figures 7.2, 7.3, and 7.4 were repeated for OBA using DASA as the underlying local scheduling algorithm at the processors and the message-scheduling server.

Figure 7.5 shows the performance of OBA-DASA compared to RBA*-DASA when increasing-ramp adaptation functions were used for all tasks. We observe that RBA*-DASA produces higher aggregate benefit and lower missed deadlines than OBA-DASA. The same observation was obtained when using constant periodic adaptation function and increasing ramp aperiodic adaptation functions for all tasks (except that on the aggregate benefit plot in Figure 7.6, the aggregate benefit curves overlap at some points.) In general RBA* is expected to result in resource allocations that lead to higher aggregate benefit and lower missed deadlines because it bases its allocation decisions on accurate response time estimation that takes the local process scheduler into account, whereas OBA uses only overload analysis to perform the allocation. Overload analysis does not provide an accurate estimate for the subtask response time.

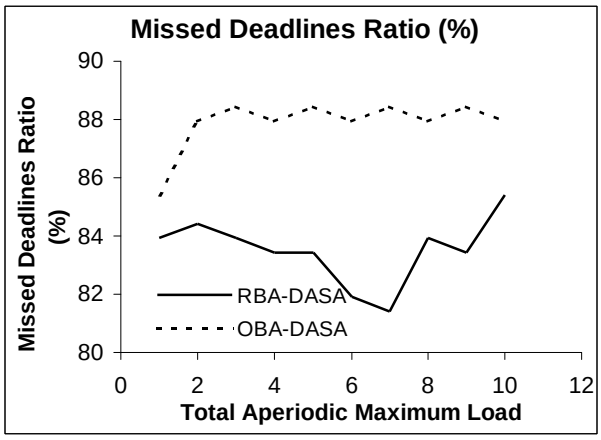


(a) (b)

Figure 7.5. Performance of OBA and RBA* For Increasing Ramp Adaptation Functions: (a) Aggregate Accrued Benefit and (b) Missed Deadline Ratio



(a)



(b)

Figure 7.6. Performance of OBA and RBA* for Fixed Periodic Load and Increasing Aperiodic Load: (a) Aggregate Accrued Benefit and (b) Missed Deadline Ratio

The error analysis results are shown in Figure 7.7 for OBA-DASA and RBA*-DASA. From the figure, we notice that RBA* gives a better performance under errors in anticipated loads, for the same reasons discussed in the last paragraph.

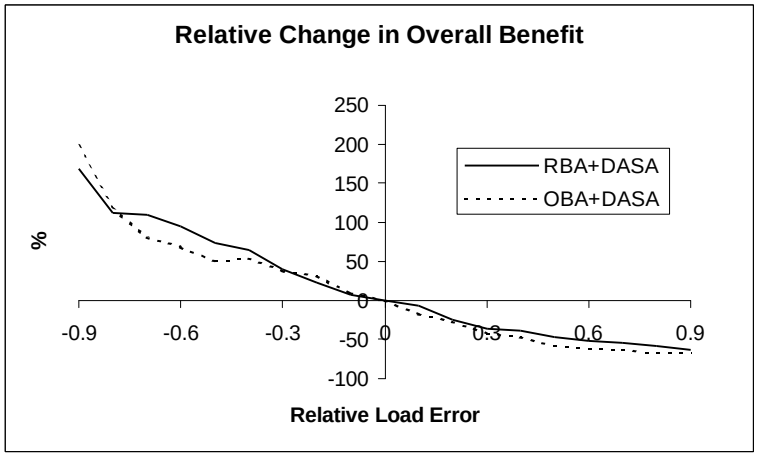


Figure 7.7. Effect of Error in Anticipated Load on OBA's Performance

Chapter 8: Conclusions and Future Work

In this thesis, we present two resource allocation algorithms called RBA* and OBA for proactive resource allocation in asynchronous real-time distributed systems. We consider an application model where application timeliness requirements are expressed using Jensen's benefit functions and propose adaptation functions to describe the anticipated application workload during future time intervals. Furthermore, we consider an adaptation model where subtasks of application tasks are replicated at run-time for sharing workload increases, and a real-time Ethernet system model where message collisions are resolved using a deterministic routing algorithm. Given such application, adaptation, and system models, our objective is to maximize aggregate application benefit and minimize aggregate missed deadline ratio.

This problem can be shown to be NP-complete. As a result, finding the optimal solution is computationally intractable. Thus, RBA* and OBA heuristically compute the number of replicas needed for task subtasks and their processor assignment, in polynomial time, but do not necessarily yield the optimal allocation. The algorithms aim to maximize the aggregate benefit by allocating resources to the higher benefit tasks before the lower benefit tasks, by de-allocating resources for the tasks that cannot complete execution by their deadlines, and by not allowing a lower benefit task to affect the timeliness of a higher benefit task.

The RBA* algorithm analyzes response times of task subtasks to get accurate estimate of the subtasks response times. Response times are then compared to the deadlines to determine how many replicas are needed for a subtask. RBA* incurs a worst-case computational complexity of $O\left(p^2m^4n^4\lceil W/k\rceil^4\right)$ under the DASA scheduler and an amortized complexity of $O\left(N^2\right)$ for the most computationally intensive component. The OBA algorithm, on the other hand, analyzes processor overload situations for computing the allocations and incurs a better worst-case complexity of $O\left(p^2m^2n^2\lceil W/k\rceil^2\log\left(mn\lceil W/k\rceil\right)\right)$ and amortized complexity of $O(\log N)$ for the

component that corresponds to RBA*'s most computationally intensive component. However, we hypothesize that OBA may perform worse than RBA* due to the difficulty in making "good" resource allocations by ignoring the precise behavior of the schedulers, especially during overload situations.

Thus, to study the relative performance of the algorithms and to determine how different scheduling and routing algorithms affect their performance, we conduct benchmark-driven experiments. The experiments study the effect of increasing the load in general on the performance of the algorithms. They also focus on the effect of increasing the aperiodic load and the effect of errors in the anticipated load. The experimental results reveal that RBA* produces higher aggregate benefit and lower missed deadline ratio when the DASA algorithm is used for scheduling and routing than under other algorithms. Furthermore, we observe that RBA* produces higher aggregate benefit and lower missed deadline ratio than OBA, confirming our intuition that accurate response time analysis can lead to better results.

Thus, the major contribution of the thesis is the RBA* and OBA algorithms that seek to maximize aggregate benefit and minimize aggregate missed deadline ratio in asynchronous real-time distributed systems by performing the resource allocation proactively. To the best of our knowledge, we are not aware of any efforts that proactively allocate resources in asynchronous real-time distributed systems.

Several aspects of this work are under further investigation. RBA* and OBA are centralized resource allocation algorithms, which may potentially affect their scalability. For example, in large systems, it may take a considerable amount of time to convey (and enact) the decisions of the centralized resource allocation algorithm from the host that is running the algorithm to the computing hosts. Therefore, a centralized resource allocation algorithm is expected to be more scalable. Another aspect is that the adaptation functions are deterministic in the sense that the user anticipates the future workload without uncertainties (though we experimentally study the algorithm's performance in the presence of uncertainties.) It may be possible to define adaptation functions in a

probabilistic setting, thereby enabling probabilistic decision-making for adaptation. Furthermore, fault-tolerance is a key requirement in asynchronous real-time distributed systems, in addition to timeliness. All these issues are currently being studied. For instance, consider the case where the host that is running the resource allocation algorithm fails during computing the allocation. Such failure will cause the whole system to fail at least for the future time window under consideration, since the application tasks will not have the required resources to execute.

Appendix A: Best-Effort Scheduling Algorithms

In this appendix, we provide an explanation of the best-effort real-time scheduling algorithms that were used as the underlying scheduling algorithms for processes at the local processors and messages at the message-scheduling server. In the experimental evaluation of RBA* and OBA, we used the best-effort algorithms known as DASA [Cla90] LBESA [Loc86], RED [BS95], and RHD [Butta97] for process scheduling and message scheduling. In the following subsections, we discuss each of the algorithms.

A.1 DASA Best-Effort Scheduling Algorithm

DASA (Dependent Activity Scheduling Algorithm) [Cla90] was developed by R. K. Clark in 1990. DASA was designed for single-processor systems. The algorithm uses the notion of the benefit density, which is defined as the ratio of task benefit to (remaining) task execution time. The point behind using this notion of benefit density of a task is to capture the benefit accrued per time unit as a result of executing the task (or the "return for investment".)

The main objectives of DASA are maximizing the aggregate accrued benefit and maximizing the aggregate achieved deadlines. In that regard, DASA does not differentiate between periodic or aperiodic tasks. Therefore, the scheduling unit is called a phase, which represents an instant of task execution. Each phase is defined by a worst-case execution time, an absolute deadline, and a benefit. DASA assumes rectangular benefit functions for all tasks. As a preemptive algorithm, the arrival and the termination of a phase provide the scheduling events where the algorithm makes a decision to determine the next phase to execute. DASA can schedule tasks with dependencies between phases., that is a phase cannot execute unless some other . A simpler version of DASA, called DASA/ND can be applied if there are no dependencies between tasks.

Figure A.1 shows a simplified procedural description of DASA. Note that in the case of DASA/ND, steps 2 and 4 are skipped. First, the algorithm starts by creating an empty schedule. Second, if there are any dependencies, it checks the dependencies among the

phases. Third, it computes the benefit densities of all phases. Then, any deadlocks are to be resolved. Because the algorithm examines the phases in the decreasing benefit density order, sorting the phases according to the value density is necessary. Having sorted the phases, the algorithm iteratively tests the phases by inserting the first phase in the empty schedule, examining the feasibility of the schedule, inserting the second phase, and so on. At any point if the schedule is not feasible due to the insertion of the last phase, the phase is removed from the schedule and the next phase in the decreasing value density order is considered.

1. Create an empty schedule
2. Determine dependencies among phases
3. Calculate benefit density for each phase
4. If deadlock detected, resolve it
5. Sort phases by potential utility density
6. Examine each phase in order of decreasing benefit density
    6.1 tentatively add the phase, and its dependencies, to the schedule in deadline order
    6.2 test the feasibility of the schedule
    6.3 if feasible, make tentative changes, else discard them
    6.4 apply optimizations to reduce schedule if possible
7. Select the phase with the earliest deadline

Figure A.1 Simple Procedural Description of DASA Algorithm

A.2 LBESA Best-Effort Scheduling Algorithm

LBESA (Locke's Best Effort Scheduling Algorithm) was developed by C. D. Locke in 1986. It has the same objectives as DASA, which are maximizing the aggregate benefit and maximizing the aggregate missed deadlines. However LBESA examines the phases from the earliest absolute deadline to the latest absolute deadline, whereas DASA examines the phases in a decreasing benefit density order. Another difference is that, according to LBESA, when an overload is detected, LBESA keeps rejecting phases with the lowest value densities until the overload is resolved. Figure A.2 shows a simple procedural description of LBESA.

Compared to DASA, LBESA supports some features that are not supported by DASA. For example, LBESA supports non-rectangular benefit functions that are not supported by DASA (DASA supports rectangular benefit functions only). Moreover, LBESA allows task execution times to be described stochastically using probability distribution functions, while DASA assumes that execution times are deterministic. However, unlike DASA, LBESA assumes no dependencies between phases.

1. Create an empty schedule
2. Sort all phases according to deadlines
3. For each phase (in increasing deadline)
   3.1 Insert phase in schedule at its deadline position
   3.2 Check for schedule feasibility
   3.3while (schedule is not feasible and schedule not empty)
      3.3.1 Remove least-value-density phase from schedule
5. Select phase in schedule with earliest deadline

Figure A.2 Simple Procedural Description of LBESA Algorithm

A.3 The RED Algorithm

RED (Robust Earliest Deadline) is a robust scheduling algorithm proposed by Buttazzo and Stankovic for dealing with firm aperiodic tasks in overloaded environments.The algorithm combines many features including graceful degradation in overloads and deadline tolerance. It operates in normal and overload conditions, and it is able to predict not only deadline misses but also the size of the overload, its duration and, and its overall impact on the system [Butta97].

In RED, each task is characterized by four parameters: a worst-case execution time (C_i), a relative deadline (D_i), a deadline tolerance (M_i), and a benefit (V_i). The deadline tolerance is the amount of time by which a task is allowed to execute after the deadline and still produce a valid result. In RED, the primary deadline plus the deadline tolerance provides a sort of a secondary deadline, which is used to run the acceptance test in

overload conditions. Note that having a tolerance greater than zero is different from having a longer deadline because the tasks are scheduled according to their primary deadlines but accepted according their secondary deadlines. Thus, a schedule is said to be *strictly feasible* if all tasks complete before their primary deadlines, whereas is said to be *tolerant* if there exists some task that executes after its primary deadline but completes within its secondary deadline [Butta97].

$RED\_acceptance(J, J_{new})$
/* J is the existing task set, J_{new} is a newly arrived task */
1. $E = 0$
2. $L_0 = 0$
3. $d_0 = current\_time(\ )$
4. $J' = J \cup \{J_{new}\}$
5. $k = \langle$position of J_{new} in the task set $J' \rangle$
6. for each task J_i' such that $i \geq k$ do
/* compute the maximum exceeding time */
   6.1 $L_i = L_{i-1} + (d_i - d_{i-1}) - c_i$ /* c_i is the remaining worst-case execution time */
   6.2 if $(L_i + M_i < -E)$ then $E = -(L_i + M_i)$
7. if $(E > 0)$
   7.1 $\langle$select J^* of least benefit tasks to be rejected$\rangle$
   7.2 $\langle$reject all tasks in $J^* \rangle$

Figure A.3 The RED Acceptance Test

RED performs an acceptance test based on the task residual laxity. The residual laxity (L_i) of a task is defined as the interval between its estimated finishing time (f_i) and its primary absolute deadline (d_i). RED also defines the exceeding time (E_i) as the time that a task executes after its secondary deadline (E_i is considered to be zero if the task completes by its primary deadline.) Thus, the schedule will be strictly feasible if and only if the residual laxity values for all the tasks is nonnegative, whereas it will be tolerant if some residual laxities are of negative values but the maximum exceeding time among the tasks is zero. Figure A.3 shows the RED acceptance test.

A.4 The RHD Algorithm

RHD (Robust High Density) is a quite simple but efficient algorithm. In RHD, the task with the highest benefit density is scheduled first, regardless of its deadline [Butta97].

References

[AAS97] T. Abdelzaher, E. Atkins, and K. Shin, "QoS Negotiation in Real-Time Systems and its Application to Automated Flight Control," *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pages 228-238, June 1997.

[ABR+93] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, "Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling," *Software Engineering Journal*, Vol. 8, No. 5, pages 284-292, September 1993.

[AS98] T. Abdelzaher and K. Shin, "End-host Architecture for QoS-Adaptive Communication," *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium*, pages 121-130, June 1998.

[Butta97] Giorgio Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, Kluwer Academic Publishers, Norwell, Massachusetts, 1997.

[BS95] G. Buttazzo and J. Stankovic, "Adding Robustness in Dynamic Preemptive Scheduling," Book Chapter, *Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems*, D. Fussel and M. Malek (eds), Kluwer Academic Publishers, Boston, Massachusetts, pages 67-88, 1995.

[BN+98] S. Brandt, G. Nutt, et al., "A Dynamic Quality of Service Middleware Agent for Mediating Application Resource Usage," *Proceedings of the IEEE Real-Time Systems Symposium*, pages 307-317, December 1998.

[CF99] F. Cristian and C. Fetzer, " The Timed Asynchronous Distributed System Model," *IEEE Transactions on Parallel and Distributed Systems*, Volume 10, Number 6, pages 642-657, June 1999.

[Cla90] R. K. Clark, "Scheduling Dependent Real-Time Activities," CMU-CS-90-155 (Ph.D. Thesis), Department of Computer Science, Carnegie Mellon University, 1990.

[CMM97] A. Chavez, A. Moukas and P. Maes, "Challenger: a Multi-Agent System for Distributed Resource Allocation," *Proceedings of the First International Conference on Autonomous Agents*, pages 323-331, 1997.

[CSR86] S. Cheng, J. Stankovic, and K. Ramamritham, "Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-time Systems," *Proceedings of the IEEE Real-Time Systems Symposium*, pages 166-174, 1986.

[CSSL97] S. Chatterjee, J. Snydir, B. Sabata, and T. Lawrence, "Modeling Applications for Adaptive QoS-based Resource Management," *Proceedings of the Second IEEE High-Assurance System Engineering Workshop*, pages 194-201, August 1997.

[Deva01] Ravi Devarasetty, "Heuristic Algorithms for Adaptive Resource Management of Periodic Tasks in Soft Real-Time Distributed Systems," M.S. Thesis, The Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, 2001.

[DTB93] R. Davis, K. Tindell, and A. Burns, "Scheduling Slack Time in Fixed-Priority Preemptive Systems," *Proceedings of the IEEE Real-Time Systems Symposium*, pages 222-231, December 1993.

[HBNB97] M. Humphrey, S. Brandt, G. Nutt, and T. Berk, "The DQM Architecture: Middleware for Application Centered QoS Resource Management," *Proceedings of the IEEE Workshop on Middleware for Distributed Real-Time Systems and Services, the Eighteenth IEEE Real-Time Systems Symposium*, pages 97-104, December 1997.

[HL98] J.-F. Hermant and G. Le Lann, "A Protocol and Correctness Proofs for Real-Time High-Performance Broadcast Networks," *Proceedings of the Eighteenth International Conference on Distributed Computing Systems*, pages, pages 360-369, 1998.

[HLR01] J. Hansen, J. Lehoczky, and R. Rajkumar, "Optimization of Quality of Service in Dynamic Systems," *Proceedings of the Fifteenth International Parallel and Distributed Processing Symposium*, pages 1001-1008, 2001.

[HSNL97] D. Hull, A. Shankar, K. Nahrstedt and J. W. S. Liu, "An End-to-End QoS Model and Management Architecture," *Proceedings of the IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, pages 82-89, 1997.

[IEEE802a] IEEE's 802.1p Standard, available at: http://standards.ieee.org/catalog/IEEE802.1.html

[IEEE802b] IEEE's 802.1p Standard (discussion), available at: http://www.nwfusion.com/news/tech/0907tech.html

[Jen92] E. Douglas Jensen, "Asynchronous Decentralized Real-Time Computer Systems," Book Chapter, *Real-Time Computing, Proceedings of the NATO Advanced Study Institute*, W. A. Halang and A. D. Stoyenko (eds), St. Martin, Springer-Verlag, October 1992.

[KG97] B. Kao and H. Garcia-Molina, "Deadline Assignment in a Distributed Soft Real-time System," *IEEE Transactions on Parallel and Distributed Systems*, Volume: 8 Issue: 12, pages 1268-1274, December 1997.

[KLB98] M. Katchabaw, H. Lutfiyya and M. Bauer, "Driving Resource Management with Application-Level Quality of Service Specifications," *Proceedings of the First International Conference on Information and Computation Economies*, pages 83-91, 1998.

[KS92] G. Koren and D. Shasha, "D-Over: An Optimal On-line Scheduling Algorithm for Overloaded Real-Time Systems," *Proceedings of the IEEE Real-Time Systems Symposium*, pages 290-299, December 1992.

[Lee99] Chen Lee, "On Quality of Service Optimization with Discrete QoS Options," Ph.D. Thesis, Carnegie Mellon University, 1999.

[Liu00] J. W. S. Liu, *Real-Time Systems*, Prentice Hall, Upper Saddle Point, New Jersey, 2000.

[LLSR99a] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar and J. Hansen, "A Scalable Solution to the Multi-Resource QoS Problem," *Processdings of the Twentieth IEEE Real-Time Systems Symposium*, pages 315-326, December 1999.

[LLSR99b] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, "On Quality of Service Optimization with Discrete QoS Options," *Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium*, pages 276-286, 1999.

[Loc86] C. D. Locke, “Best-Effort Decision Making for Real-Time Scheduling,” CMU-CS-86-134 (Ph.D. Thesis), Department of Computer Science, Carnegie Mellon University, 1986.

[LRT92] J. P. Lehoczky and S. Ramos-Thuel, “An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems,” *Proceedings of the IEEE Real-Time Systems Symposium*, pages 110-123, 1992.

[LSS87] J. Lehoczky, L. Sha, and J. Strosnider, “Enhanced Aperiodic Responsiveness in Hard Real-time Environments,” *Proceedings of the Eighth IEEE Real-Time Systems Symposium*, pages 261-270, 1987.

[Mills95] D. L. Mills, “Improved Algorithms for Synchronizing Computer Network Clocks,” *IEEE/ACM Transactions on Networks*, pages 245-254, June 1995.

[Mok83] A. K. Mok, “Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment,” Ph.D. Thesis, Massachusetts Institute of Technology, 1983.

[NS94] M. Natale and J. Stankovic, “Dynamic End-to-End Guarantees in Distributed Real-Time Systems,” *Proceedings of the IEEE Real-Time Systems Symposium*, pages 216-227, December 1994.

[RCF97] I. Ripoll, A. Crespo, and A. Fornes, “An Optimal Algorithm for Scheduling Soft Aperiodic Tasks in Dynamic Priority Preemptive Systems,” *IEEE Transactions on Software Engineering*, Volume 23, Number 6, pages 388-400, June 1997.

[RH01] B. Ravindran and T. Hegazy, "RBA: A Best Effort Resource Allocation Algorithm for Asynchronous, Real-Time Distributed Systems," *Journal of Research and Practice in Information Technology, Special Issue on Distributed Systems*, Volume 33, Number 3, pages 213-227, August 2001

[RLLS97] R. Rajkumar, C. Lee, J. Lehoczky and D. Siewiorek, "A Resource Allocation Model for QoS Management," *Proceedings of the Eighteenth IEEE Real-Time Systems Symposium*, pages 298-307, December 1997.

[RLLS98] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek, "Practical Solutions for QoS-based Resource Allocation Problems," *Proceedings of the Nineteenth IEEE Real-Time Systems Symposium*, pages 296-306, December 1998.

[RS99] D. Rosu and K. Schwan, "FARACost: An Adaptation Cost Model Aware of Pending Constraints," *Proceedings of the Twentieth IEEE Real-Time Systems Symposium*, pages 224-233, December 1999.

[RSY98] D. Rosu, K. Schwan, and S. Yalamanchili, "FARA – A Framework for Adaptive Resource Allocation in Complex Real-Time Systems," *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pages 79-84, June 1998.

[RSYJ97] D. Rosu, K. Schwan, S. Yalamanchili, and R. Jha, "On Adaptive Resource Allocation for Complex Real-Time Applications," *Proceedings of the Eighteenth IEEE Real-Time Systems Symposium*, pages 320-329, December 1997.

[RSZ89] K. Ramamritham, J. Stankovic, and W. Zhao, "Distributed Scheduling of Tasks with Deadlines and Resource Requirements," *IEEE Transactions on Computers*, Volume 38, Number 8, pages 1110-1123, August 1989.

[RTL93] S. Ramos-Thuel and J. Lehoczky, "On-line Scheduling of Hard Deadline Aperiodic tasks in Fixed-Priority Systems," *Proceedings of the IEEE Real-Time Systems Symposium*, pages 160-171, 1993.

[Rav00] B. Ravindran, "Engineering Dynamic Real-Time Distributed Systems: Architecture, System Description Language, and Middleware," *IEEE Transactions on Software Engineering*, To appear in Volume 27, Number 12, December 2001.

[SB96] M. Spuri and G. Buttazzo, "Scheduling Aperiodic Tasks in Dynamic Priority Systems," *Journal of Real-Time Systems*, Volume 10, pages 179-210, March 1996.

[SLS88] B. Sprunt, J. Lehoczky, and L. Sha, "Exploiting Unused Periodic time for Aperiodic Service Using the Extended Priority Exchange Algorithm," *Proceedings of the IEEE Real-Time Systems Symposium*, pages 251-258, 1988.

[SR91] J. Stankovic and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real-time Systems," *IEEE Software*, Volume 8, Number 3, pages 62-72, May 1991.

[SRC85] J. Stankovic, K. Ramamritham, and S. Cheng, "Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-time Systems," *IEEE Transactions on Computers*, Volume C-34, Number 12, pages 1130-1141, December 1985.

[SSL89] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic Task Scheduling in Hard Real-time Systems," *Journal of Real-Time Systems*, Volume 1, Number 1, pages 27-60, 1989.

[TLS96] T-S. Tia, J.W.-S. Liu, and M. Shankar, "Algorithms and Optimality of Scheduling Soft Aperiodic Requests in Fixed-Priority Preemptive Systems," *Journal of Real-Time Systems*, Volume 10, pages 23-43, January 1996.

[VCF00] P. Veríssomo, A. Casimiro, and C. Fetzer, "The Timely Computing Base: Timely Actions in the Presence of Uncertain Timeliness," *Proceedings of International Conference of Dependable Systems and Networks*, pages 533-542, 2000.

[WS99] L. Welch and B. Shirazi, "A Dynamic Real-Time Benchmark for Assessment of QoS and Resource Management Technology," *Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium*, pages 36-45, June 1999.

[WRSB98] L. Welch, B. Ravindran, B. Shirazi, and C. Bruggeman, "Specification and Modeling of Dynamic, Distributed Real-Time Systems," *Proceedings of the Nineteenth IEEE Real-Time Systems Symposium*, pages 72-81, December 1998.

[ZRS87a] W. Zhao, K. Ramamritham, and J. Stankovic, "Preemptive Scheduling Under Time and Resource Constraints," *IEEE Transactions on Computers*, Volume 36, Number 8, pages 949-960, August 1987.

[ZRS87b] W. Zhao, K. Ramamritham, and J. Stankovic, "Scheduling Tasks With Resource Requirements in Hard Real-Time Systems," *IEEE Transactions on Software Engineering*, Volume 13, Number 5, pages 564-577, May 1987.

[Znyx01] ZNYX Networks - Product ZX340Q Series, available at:

http://www.znyx.com/products/netblaster/zx340q.htm
(Accessed in October 2001).

Tamir A. Hegazy

Tamir A. Hegazy is receiving his M.S. degree from the Computer Engineering at the Bradley Department of Electrical and Computer Engineering, Virginia Tech. He received his B.S. degree in Electronic Engineering from Mansoura University, Egypt in 1997. He is the recipient of the "Honor Degree" and the "Excellence Prize" from Mansoura University for distinguished academic performance during the B.S. degree. His current research focus is on asynchronous real-time distributed systems. He has several publications on the topic of adaptive resource allocation in asynchronous real-time distributed systems in IEEE journals and conferences.