ABSTRACT

A RESOURCE ALLOCATION MECHANISM BASED ON COST FUNCTION SYNTHESIS IN COMPLEX SYSTEMS

by
Carlos C. Amaro

While the management of resources in computer systems can greatly impact the usefulness and integrity of the system, finding an optimal solution to the management problem is unfortunately NP hard. Adding to the complexity, today's 'modern' systems — such as in multimedia, medical, and military systems — may be, and often are, comprised of interacting real and non-real-time components. In addition, these systems can be driven by a host of non-functional objectives – often differing not only in nature, importance, and form, but also in dimensional units and range, and themselves interacting in complex ways. We refer to systems exhibiting such characteristics as Complex Systems (CS).

We present a method for handling the multiple non-functional system objectives in CS, by addressing decomposition, quantification, and evaluation issues. Our method will result in better allocations, improve objective satisfaction, improve the overall performance of the system, and reduce cost —in a global sense. Moreover, we consider the problem of formulating the cost of an allocation driven by system objectives. We start by discussing issues and relationships among global objectives, their decomposition, and cost functions for evaluation of system objective. Then, as an example of objective and cost function development, we introduce the concept of deadline balancing. Next, we proceed by proving the existence of combining models and their underlying conditions. Then, we describe a hierarchical model for system objective function synthesis. This synthesis is performed solely for the purpose of measuring the level of objective satisfaction in a proposed hardware to software

allocation, not for design of individual software modules. Then, Examples are given to show how the model applies to actual multi-objective problems.

In addition the concept of deadline balancing is extended to a new scheduling concept, namely Inter-Completion-Time Scheduling (*ICTS*). Finally, experiments based on simulation have been conducted to capture various properties of the synthesis approach as well as ICTS. A prototype implementation of the cost functions synthesis and evaluation environment is described, highlighting the applicability and usefulness of the synthesis in realistic applications.

A RESOURCE ALLOCATION MECHANISM BASED ON COST FUNCTION SYNTHESIS IN COMPLEX SYSTEMS

by
Carlos C. Amaro

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

Department of Computer and Information Science

August 1998

APPROVAL PAGE
(Page 1 of 2)

A RESOURCE ALLOCATION MECHANISM BASED ON COST FUNCTION SYNTHESIS IN COMPLEX SYSTEMS

Carlos C. Amaro

8/6/98

Dr. Alexander D. Stoyen, Dissertation Advisor — Date
Associate Professor of Computer and Information Science and
Electrical and Computing Engineering, NJIT

8/6/98

Dr. Murat M. Tanik, Committee Member — Date
Director, Electronic Enterprise Engineering
of Computer and Information Science, NJIT

08/06/1998

Dr. Sanjoy K. Baruah, External Committee Member — Date
Assistant Professor of Computer Science, University of Vermont,
Burlington, Vermont

8/4/98

Dr. Michael G. Hinchey, Committee Member — Date
Assistant Professor of Computer and Information Science, NJIT

APPROVAL PAGE
(Page 2 of 2)

A RESOURCE ALLOCATION MECHANISM BASED ON COST FUNCTION SYNTHESIS IN COMPLEX SYSTEMS

Carlos C. Amaro

7/24/98

Dr. Phillip A. Laplante, External Committee Member — Date
President, Pennsylvania Institute of Technology,
Media, Pensylvania

08/06/98

Dr. Peter A. Ng, Committee Member — Date
Professor of Computer and Information Science, NJIT

8/6/98

Dr. Donald H. Sebastian , Committee Member — Date
Executive Director, Center for Manufacturing Systems
Professor of Industrial & Manufacturing Engineering, NJIT

BIOGRAPHICAL SKETCH

Author: Carlos C. Amaro

Degree: Doctor of Philosophy

Date: August 1998

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 1998
- Master of Science in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 1993
- Bachelor of Arts in Computer Science,
  RUTGERS, The State University of New Jersey, Newark, NJ, 1991

Major: Computer Science

Presentations and Publications:

C.C. Amaro, S.K. Baruah, and A.D. Stoyen, "Inter-Completion Time Scheduling (ICTS): Non-preemptive scheduling to maximize the minimum inter-completion time," in *Proceedings of IEEE Fourth International Conference on Engineering of Complex Computer Systems*, Monterey, California, August 1998.

C.C. Amaro, S.K. Baruah, A.D. Stoyen, and W.A. Halang, "Non-preemptive scheduling to maximize the minimum global inter-completion time (MGICT)," in *Proceedings of the 23rd IFAC/IFIP Workshop on Real-Time Programming*, Shantou, Guandong Province, P.R. China, June 1998.

C.C. Amaro, S.K. Baruah, T.J. Marlowe, and A.D. Stoyen, "Non-preemptive scheduling to maximize the minimum intercompletion time," to appear *The Journal of Combinatorial Mathematics and Combinatorial Computing.* Also, Technical Report NJIT/CIS-96-13, Real-Time Computing Laboratory, Department of Computer and Information Science, NJIT, April 1996.

C.C. Amaro, T. J. Marlowe, and A. D. Stoyenko, "Objective Function Synthesis For Complex Real-Time Systems," in *Proceedings of the 21st IFAC/IFIP Workshop on Real-Time Programming*, Gramado - RS, Brazil, November 1996.

M. Harelick, A.D. Styenko, C.C. Amaro, and R. Scherl, "Operator Resources For Large Complex Systems," in *Proceedings of IEEE 2nd International Conference on Engineering of Complex Computer Systems*, Montreal, Canada, October 1996.

C.C. Amaro, T.J. Marlowe, and A.D. Stoyenko, "Objective Function Synthesis: The REAL Resource Allocator Approach," Technical Report NJIT/CIS-95-27, Real-Time Computing Laboratory, Department of Computer and Information Science, NJIT, December 1995.

T. Marlowe, A. Stoyenko, P. Laplante, N. Jones , C.C. Amaro, P. Sinha, B.C. Cheng, M. Harelick, " Testing Network Assignment Algorith with a Structured Workload Generator," in *Proceedings of the Seventh Annual Software Technology Conference*, Salt Lake City, Utah, April, 1995.

A. Stoyenko, T. Marlowe, M. Younis, A. Ganesh, C.C. Amaro, P. Laplante, A. Silberman, P. Sinha, "Towards A Language Paradigm for Construction and Development of Complex Computer Systems," in *Proceeding of IEEE Workshop on Composability of Fault-Resilient Real Time Systems*, San Juan, Pureto Rico, Dec 1994.

C.C. Amaro, M. Harelick, P. Sinha, A.D. Stoyenko, T.J Marlowe, P. Laplante, A. Silberman, N. Jones, B.C. Cheng, T. Tugcu, "Economics of Resource Allocation," in *Proceeding of the Complex Systems Engineering Synthesis and Assessment Technology Workshop*, Silver Springs, Maryland, pp. 195-201, July 1994.

T. Marlowe, A. Stoyenko, P. Laplante, R. Daita, C.C. Amaro, C. Nguyen, S. Howell, "Multiple-Goal Objective Functions for Optimization of Task Assignment in Computer Systems," *Elsevier, Control Engineering Practice*, Vol. 4, Iss. 2, February 1996. Earlier version: in *Proceeding of the 19th IFAC/IFIP Workshop on Real Time Programming*, Isle of Reichenau, Lake Constance, Germany, June 1994.

T. Marlowe, A. Stoyenko, C. Nguyen, S. Howell, P. Laplante, R. Daita, C.C. Amaro, "Multiple-Stage Dynamic-Programming Heuristic for Assignment and Scheduling in Destination," Technical Report NJIT/CIS-93-13, Real-Time Computing Laboratory, Department of Computer and Information Science, NJIT, October 1993.

A.D. Stoyenko, L.R. Welch, P. Laplante, T.J. Marlowe, C.C. Amaro, B.C. Cheng, A.K. Ganesh, M. Harelick, X. Jin, M. Younis, G. Yu, "A Platform for Complex Real-Time Applications," in *Proceeding of the Complex Systems Engineering Synthesis and Assessment Technology Workshop*, Silver Springs, Maryland, pp. 152-159, June 1993.

This work is dedicated to
my wife

ACKNOWLEDGMENT

I would like to express my sincere gratitude to my advisor Prof. Alexander Stoyen for his leadership and valuable advice. He guided me in my work and provided me with technical and moral support. I am very grateful to my external readers and my thesis committee for their time and the many suggestions which led to a considerable improvement of this thesis. I am especially indebted to Prof. Murat Tanik for his technical advice and guidance. Thanks are also due to all members and visitors of the Dependable Real-Time Systems Laboratory at NJIT for their friendship and support, as well as their constructive criticism and technical expertise. I am especially grateful to Dr. Thomas Marlowe for his time and guidance in the early stages of my research. Special thanks is due to Dr. Roman Nossal whose technical expertise, friendship, and support, were critical in the completion of this work.

I am truly indebted to the Office of Naval Research, the Naval Surface Warfare Center, Dahlgren Division, and AT&T for sponsoring our Lab's efforts. I would like also to thank these agencies for financially supporting my study and my trips to various conferences and research meetings. In addition, I am very grateful to the Graduate Student Association at NJIT for their financial support of my presentations at multiple workshops and conferences.

My deep thanks to my wife for her support and assistance throughout my study and to my son, who's arrival expedited this matter. Finally, I would like to thank my parents who encouraged me all the way and blessed me with their prayers and my wife's parents for their patience and generosity.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Continued)

Chapter | Page

TABLE OF CONTENTS

(Continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES
(Continued)

LIST OF FIGURES

(Continued)

CHAPTER 1

INTRODUCTION

Complex Systems can be characterized as large applications running on a distributed and heterogeneous network with an arbitrary but known topology, driven by various *non-functional goals/objectives* such as performance, real-time behavior, human factors, reliability, and fault tolerance [47]. These objectives frequently conflict, are often non-commensurable, and sometimes compositionally imprecise. Satisfaction of these objectives interacts strongly with assignment of system components to resources in a distributed environment. Devising mechanisms capable of measuring and differentiating the level of satisfaction of such objectives for a given allocation is itself a very complex problem [54, 56, 80].

The problem of resource allocation has been addressed in many fields, including economics [8, 49, 60] and operations research [12, 16, 20, 50, 51]. In all instances, resource allocation is perceived as *the science concerned with the problem of using or administering scarce resources so as to attain the greatest or maximum fulfillment of society's unlimited wants* [49]. Likewise, in computer science, resource allocation is concerned with the problem of managing system resources — i.e., memory, communication links and switches, processors, display devices, sensors, actuators, semaphores, stacks, locks, data, buffers, etc. — so as to attain the greatest fulfillment of the system's objectives.

In essence, resource allocation deals with the timely satisfaction of a task's needs, while at the same time, leaning towards certain desired characteristic behavior as a consequence of the mapping. These desired characteristics/objectives do not have an absolute required level of satisfaction to achieve; rather, they are simply criteria the values of which do not invalidate the allocation, but differentiate better allocations from worse ones.

Previous work on resource allocation mainly assumes only a single or at a maximum two goals. With resource allocation almost all methods published to date concern themselves with a cost function based on a variation of the two objectives, load balancing (LB) and inter-processor communication (IPC) minimization [1, 15, 45]. However, today's real-time systems demand much more than just these two simple aspirations.

A common characteristic of most of the related work is that their cost functions are not constructed in a well-defined manner. While the decomposition of the overall goals into smaller, tractable objectives is present in most of the papers, the second step, which is the synthesis of the system cost function, is not covered in detail.

This thesis presents an approach to the synthesis of the cost function for resource allocation in the development of complex real-time systems. Originating from a user-defined decomposition of the objectives it is shown which steps have to be followed in the derivation of the system cost function.

Also, objective deadline balancing (DLB) is introduced along with representative cost functions. In addition, this notion of DLB is extended into the concept of scheduling to maximize the minimum inter-completion time — Inter-Completion Time Scheduling (ICTS). These approaches, to my best knowledge, have not been used before in complex systems.

This chapter provides a motivation for our study and application of cost function synthesis techniques for resource allocation in complex systems, highlights the difficulties associated with performing such techniques and points out the contribution and the organization of the thesis. In the following section, those requirements that distinguish complex systems are presented. Next, objectives, cost functions, and their relationships are discussed. Then, the complexity of the basic resource optimization problem is examined followed by a discussion on additional requirements imposed by complex systems. Next, a tool, satisfying these

requirements is described. Finally, a summary of the major contribution of this work is provided, concluding with an outline of the balance of the dissertation.

1.1 Complex System Requirements

While classical computer systems are driven by functional requirements and have no timing constraints associated with them, conventional real-time systems differentiate themselves by having a conceptual notion of time to which they must adhere to. As Halang and Stoyen [Stoyenko] explain:

> "... real time operation distinguishes itself from other forms of data processing by the explicit involvement of the dimension of *time*. This is expressed by the following two fundamental user requirements, which real time systems must fulfill under all, including extreme, load conditions:
>
> - timeliness and
> - simultaneity.
>
> These requirements are supplemented by two further ones of equal importance:
>
> - predictability and
> - dependability.
>
> Upon request from the external process, data acquisition, evaluation, and appropriate reactions must be performed on time."[30]

Hence, the absolute speed at which results are attained is of little consequence; but rather, the timeliness, within predefined and predictable time-bounds, at which results are observed, their correctness, and their completeness are decisive.

Depending on the criticality of timing constraints[1] imposed on the system by the external environment, real-time systems may be classified as *hard* or *soft*. These

[1] In this thesis we are not concerned with differentiating between system 'requirements' and 'constraints' —we are addressing non-functional objectives. Therefore, we shall use the terms interchangeably.

are distinguishable by the effects of violating the timeliness requirement; that is, missed deadlines. In *soft real-time* environments, costs rise with increased tardiness of observables, whereas in *hard real-time* systems, the cost of a missed deadline may be infinitely high, resulting in irreversible catastrophic consequences.

Unlike conventional real-time systems which consist solely of real-time tasks, Complex Systems may be comprised of three task types defined by their degree of time criticality, namely: hard real-time, soft real-time, and non-real-time. In addition to the constraints imposed by each type, their coexistence demands their interaction to be deterministic, dependable and correct. There may be other constraints induced by the coexistence. However, we will not concern ourselves with them in this thesis.

Examples of such systems can be found in multimedia (video on demand), medical (patient-monitoring), and military (theater of operation) scenarios.

1.2 Objectives and Cost Function Synthesis

Objectives are descriptions of non-functional system properties, which can be qualitatively decomposed into smaller scoped objectives and finally into attributes as suggested by Keeney and Raiffa in [38]. Thus *attributes* are atomic characteristic behavior of the system, i.e., lowest-level objectives. *Cost functions*, on the other hand, are mathematical expressions that measure and assign values to attributes and objectives.

The top-level objectives of a real-time system constitutes the overall target of the optimization process of the resource allocation strategy for a system, as in [47]. These top-level goals can be identified by so-called *decision makers* and can encompass a wide area or scope. The intended system is investigated and the desired non-functional aspects are identified as in [54]. Some of the desired characteristics may include: timeliness, predictability, fault tolerance, performance, human factors, and security. These then must be broken down into smaller scoped objectives. These

non-functional goals may be categorized into *constraints*, whose failure will result in the rejection of a proposed allocation, and *objectives*, whose relative degree of satisfaction affects in the acceptability of a proposed allocation.

Constraints typically arise from physical restrictions on the underlying application, or on the partially-specified platform or design, but can also be hard user requirements on acceptable designs. Objectives, in contrast, represent behavior desired for the application; objectives frequently can be satisfied to a greater or lesser degree. Measuring the relative level of satisfaction of an objective or set of objectives will call for quantification of these levels.

In this paper we do not address constraints. Rather, we assume proposed allocations to be constraint satisfying. Our problem is more acute in that given a set $\mathbf{F}$ of feasible solutions, we are trying to differentiate amongst these solutions. Moreover, $\mathbf{F} = \bigcup_{i=1}^{n} F_i$, $\mathbf{F} \subseteq S$, where S is the set of all possible permutations of an allocation, see Figure 1.1.

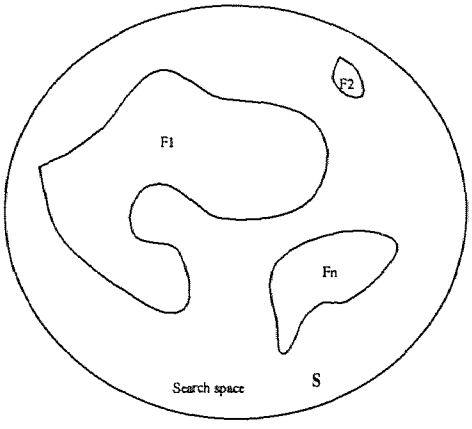


Figure 1.1 Constraint Satisfying Space **F**

Objectives typically embody goals for an entire system, but may involve only subsystems or single components. The *system objective* describes the overall target of the system and its subsystems. Failure to achieve a certain level of objective satisfaction is not in itself cause for rejecting the system; rather, better satisfaction

distinguishes good implementations. However, most research on resource allocation considers only one component of the system objective. For example, [1, 10, 15, 75] only use the objective *performance*, typically quantified as a summation of two cost metrics, interprocessor communication and execution cost. New methods need to be devised to be able to consider more than one or two metrics.

Objective satisfaction is quantified by cost functions. A single objective may have multiple cost functions. For example, communication optimization during resource allocation may be realized as "minimize overall communication", "maximize time separation between message transmissions", or "minimize the maximum of communication out of a single processor". But the problem of quantifying and managing trade-offs is significant. What are the degrees of importance for the individual cost functions for an objective? How can cost functions be computed or estimated effectively and efficiently at different stages of the design process?

Lower level *objectives* result from refinement. These smaller-scoped refinements are by themselves objectives but at a more detailed level. For example, performance can be refined into *throughput*, *response time*, and *load balancing*. Any of these objectives can be further refined. For instance, response time can entail both *average response time* and *minimize worst case response time*.

Cost functions evaluate individual system considerations used to determine inherent "goodness" as stipulated by system objectives. Some cost functions encountered in complex systems are communication cost, load balancing, failure rate, and deadline satisfaction percentage. In our model, cost functions are formulated so that minimization corresponds to better system performance, and minimizing the system cost function optimizes the degree of objective satisfaction.

Typically, high-level *cost functions* are synthesized from the amalgamation of lower-level cost functions taking into account their corresponding weights. The resulting multi-objective cost function may incorporate many objectives. Each can be

expressed in diverse units, such as time, money, count, risk or opportunity cost, and thus have fundamentally incompatible types and units. Meaningful combination of these objectives (sometimes with artificial units, and often of fundamentally different type) to obtain a single value requires *Data Fusion.*

Moreover, objectives cannot be satisfied independently, but are interrelated. For example, the cost associated with load balancing is inversely related to communication cost, and minimization of one tends to result in a larger cost for the other. Simultaneous minimization of such interrelated functions will be hard and sometimes even impossible. Measurements also need to be scaled to obtain approximately equal ranges and variances for different factor, which requires *Data Scaling.*

1.3 Resource Optimization is Hard

The simplest form of the general resource allocation optimization problem (single objective, two processing elements, and n tasks) is an NP-hard problem [1]. The problem is intractable for either mathematical programming or graph theoretical approaches [25]. Therefore, effective heuristics need to be developed.

To reduce complexity, some allocation methods optimize on a subset of the objectives. However, optimization cannot be performed on a single objective at a time. Objectives may conflict and optimizing one may worsen another. In addition, ranking tasks according to criticality and optimizing levels independently will preclude better allocations. Furthermore, optimal satisfaction of real-time task objectives may induce heavy costs for non-real-time task allocations.

Due to these inherent issues, current systems do not offer users many choices nor flexibility.

1.3.1 Resource Allocation in Complex Systems

Complex Systems like conventional real-time systems, by definition, must be deterministic, dependable, and correct. A system which satisfies these properties will almost inevitably be much more expensive than a comparable system which does not. This is because for real-time tasks, systems typically schedule and allocate resources in a pessimistic manner [66, 69, 82, 81], since they must always accommodate worst case execution scenarios. While this is not the case for non-real-time systems, processors in conventional real-time systems will by comparison tend to be underutilized, spend a lot of time idling, and inevitably costlier than any other type of system. However, Complex Systems can take advantage of this rarely used resource-time originally allocated by real-time schedulers. The unused time can be harvested and used by ready non-real-time preemptible tasks. With this advent the question then becomes one of allocation: how and where are tasks to be assigned?

In Complex Systems resource allocation can be validated by three components, specifically: schedulability, requirements conformability (other than timing), and level of objective satisfaction. Therefore, we discuss a *resource allocation process/tool* in the next section to accommodate these factors. Then, we present our contribution to such a tool.

1.3.2 A Resource Allocation Tool for Complex Systems

A general method toward the resolution of a multiobjective planning problem was presented by deNeufville and Stafford in [20] (1971). In their method five steps were proposed, they are: 1. Definition of objectives, 2. Formulation of measures of effectiveness, 3. Generation of alternatives, 4. Evaluation of the alternatives, and 5. Selection. A similar methodology, later introduced by Cohon in [16] (1978), augmented the model by combining the first two steps into 'Identification and quantification of objectives,' followed by two additional steps: 'definition of decision

variables and constraints' and 'data collection.' In addition, he added implementation as the last step to the planning phase.

While both methods may be adequate for social welfare resource allocation problems, they fall short when applied to complex computer systems. In the above models, objectives are viewed as single entities and are not decomposable — objectives cannot be comprised of other objectives. They assume their solution space to be of polynomial size and solvable. In addition, only a single set of alternatives is generated, and the preferred alternative selected through human interaction.

We will now define a similar model/tool for resource allocation in Complex Systems. We outline the tool in Figure 1.2 and describe its steps in Table 1.1.

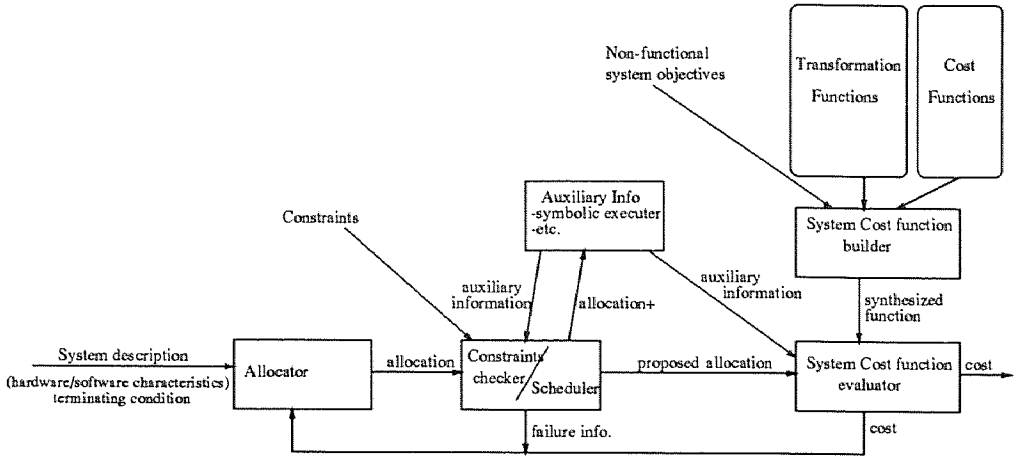


Figure 1.2 A Tool for Resource Allocation in Complex Systems.

The first step in solving the resource allocation problem is identifying the desired non-functional characteristics of the system. These characteristics are then classified as the top level objectives and are further decomposed into smaller-scoped, more specific characteristics by the *system cost function builder.* Decomposition may be finitely refined until lower level objectives can be quantified by *attribute cost functions.* This decomposition results in a hierarchical definition of system objectives. The Cost Functions in turn quantify specific atomic properties of the

Table 1.1 Steps for a Resource Allocation Tool

1	Identification of Objectives
2	Formulation & Decomposition of Objectives
3	Identification and Quantification of Cost Functions
4	Generation of allocations (constraint satisfiable)
5	Generation of simulated schedule
6	Evaluation of proposed feasible allocation
7	If termination conditions NOT satisfied go to step 4
8	Presentation of allocation(s) for implementation

allocation. Once the hierarchy is defined and quantified, the hierarchy is evaluated bottom-up by the *cost function evaluator*. Value *Transformation Functions* may be needed in order to combine sub-level results.

Allocations are then generated by an *allocator*, implementing search strategies and state space search algorithms. At each algorithmic step the constraints are checked for violations by the *constraint checker*. If any alteration by the search algorithm violates the constraints, then it is not applied and other alternatives are attempted. Each allocation is then checked to see if a feasible schedule (one that satisfies time constraints) exists for real-time tasks. If one does not exist, then the allocation is rejected and another proposed. Otherwise, a simulated schedule is derived by augmenting it with the remaining tasks (both soft and non-real-time). The derived symbolic-schedule can be used by time-driven objectives. Hence, every proposed allocation by the allocator is constraint-satisfiable —temporal and spatial.

All cost functions — static and semi-static — are then evaluated. While static functions can be measured from allocation-only data, semi-static functions need additional *auxiliary information* that can only be acquired through actual or symbolic execution of the tasks. Finally, their results may need to be transformed and combined accordingly to evaluate the objectives at each level in the hierarchy

This process continues until a *termination condition* is satisfied, some of which include: no significant improvement between iterations; maximum number of consecutive worsenings reached; maximum number of iterations allowed reached; search time expired; disparity between internal objective values exceeds limit. Afterwards, the lowest cost allocation from the set of all feasible allocations generated is selected.

This thesis will concentrate on the objective synthesis component of the general resource allocation tool. This includes, methods for objective decomposition and corresponding combining functions.

1.4 Contribution

In this dissertation, we mainly study how to synthesize cost functions for resource allocation in complex systems. The focus of this work is highlighted in Figure 1.3. We outline and discuss relationships among top-level objectives, cost functions, and the construction of the system cost function. We introduce a hierarchical model for such synthesis. We demonstrate, through example, how our model is applicable in complex systems. Finally, we implement the model in the REAL[2] prototype. Detailed description of the prototype is described in Chapter 7. The contribution of this dissertation can be summarized as follows:

- We have developed a hierarchy model to provide the basic mechanisms for objective decomposition and cost function synthesis. The decomposition takes the form of a Directed Acyclic Graph (DAG).

- We have specified data transformation and combining functions needed to evaluate objective satisfaction at each level in the hierarchy. We have identified conditions for the existence of combining models.

[2]The REsource ALlocation (REAL) prototype is partially Supported by U.S. ONR and U.S. NSWC grants, and the DESTINATION team.

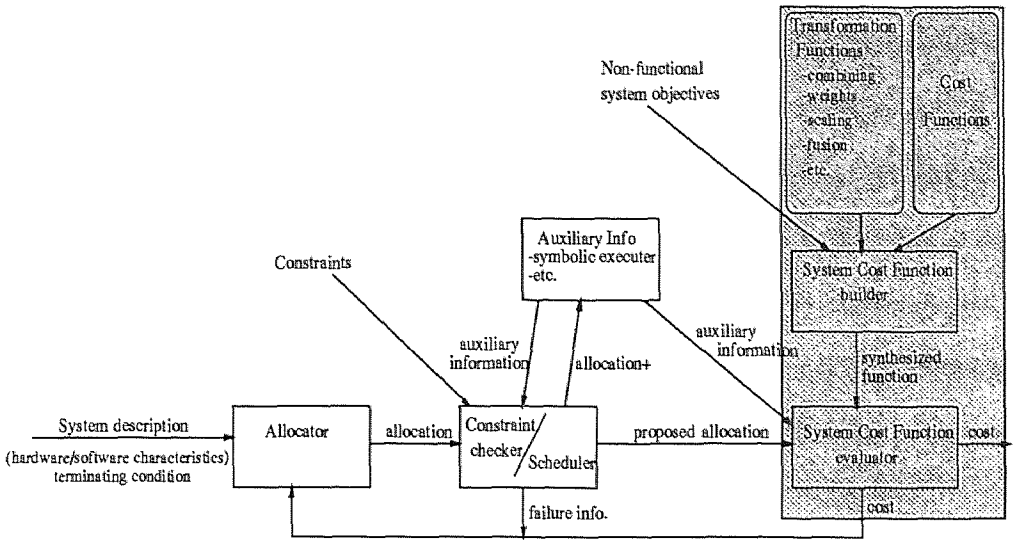


Figure 1.3 Outline of Our Contribution (Gray Area).

- We have introduced the concept of *DeadLine Balancing* (DLB) as an objective and extended it to *Intercompletion Time Scheduling* (ICTS). We describe algorithms and discuss the implication of scheduling real-time tasks so as to maximize their minimal inter-completion time.

  To validate our work empirically, we have done the following:

- We have examined the usefulness of our approach in a realistic application. The cost function synthesis approach is applied to a simple model of a stock exchange system. The objective decomposition is derived as well as three different cost function syntheses for this system. These cost functions are applied to various hardware and software scenarios. The behavior and the performance of the functions were evaluated by exhaustively calculating the function values and by using the cost functions as a fitness function for a Genetic Algorithm.

- We have studied the effects of the inter-completion time Scheduling strategy on Inter-Processor Communication and present results.

1.5 Organization

This dissertation is organized as follows. In the next chapter, a complex systems model which serves as a basis for this work is defined. Chapter 3 summarizes related work. The chapter is split into two parts, *approaches* and evaluation *methods*. Moreover, various types of approaches to solve the resource allocation problem are examined, as well as numerous allocation-cost evaluation methods. Chapter 4 presents our approach for evaluation of non-functional system objectives. The first sections cover development and evaluation of cost functions and introduce the concept of deadline balancing. The last sections describe construction and evaluation of objective functions. In Chapter 5, the concept of deadline balancing is extended to a new scheduling concept, namely Inter-Completion-Time Scheduling (*ICTS*). The algorithms are described and the implications of scheduling real-time tasks so as to maximize their minimal inter-completion time are discussed. Experiments based on simulation have been conducted to capture various properties of the synthesis approach as well as ICTS. In Chapter 6, the design and results of these experiments are illustrated. A prototype implementation of the cost functions synthesis and an evaluation environment are described in Chapter 7, highlighting the applicability and usefulness of cost function synthesis in realistic applications. Finally, Chapter 8 concludes this thesis and summarizes future research directions.

CHAPTER 2

COMPLEX SYSTEMS MODEL

In the previous chapter, we motivated our study and defined the problem that this thesis is trying to address. In this chapter, we define a complex systems model for this work. In addition, we provide definitions for some of the terms used throughout the thesis. In the next section we discuss an underlying distributed heterogeneous computer network model, followed by a discussion on the arbitrary software model.

Specific model characteristics that need to be identified are dependent on the metric to be evaluated. In addition, system resources need not be fast nor dependable. However, they must be predictable. The system itself need not be entirely fault-free, accurate, nor reliable; however, it must be deterministic. Moreover, its imprecisions or anomalies must be known and measurable at all times.

Furthermore, in the literature there are many ways of measuring atomic properties (both spatial and temporal) for reasonably real systems [30]. Therefore, we are really not concerned with measuring all of these in this thesis. Rather, what we are interested in is the *combining* of such properties. Therefore, if we have to build hardware and software models we will start with something simple but yet capable of exhibiting some of these properties.

2.1 Hardware Environment

In this section, the thesis' assumptions about complex system hardware environments are stated.

Real-time hardware (for example [30]) need not necessarily be very fast, but must provide predictable functionality enabling analysis of the system. Issues like caching, direct memory access, virtual addressing, pipelining, or asynchronous communication protocols can cause nondeterminism, and consequently should be

handled with care [85]. In this thesis, it is assumed that the execution time of each machine instruction is known. In addition, machines need not have identical instruction execution times. Furthermore, all machines are connected by a communication network. It is also assumed that the hardware does not introduce any unpredictably long delays into program execution or message propagation.

2.2 Software Environment

The software model consists of n tasks. Tasks communicate end-to-end. Moreover, tasks do not communicate while executing, only before execution or after. Furthermore, a task starts to execute when it has received all messages from all its predecessors, its release time has passed, and the processor to which it has been assigned becomes available. Moreover, no processor is executing more than one task at any given instant in time.

Tasks with their communication messages form a Directed Acyclic Graph (DAG). Missed deadlines of different tasks within the same time-category (recall: hard, soft, and non-real-time) are of equal importance and costs. In this model, each task T_i is characterized by several parameters – a *release time* r_i, an *execution requirement* $e_{i,j}$ $(j = 1, \ldots, m)$, a *deadline* d_i, and *interprocess communication* vector χ_i, with the interpretation that task T_i becomes ready for execution at time r_i, and needs to be executed for $e_{i,j}$ units of time over the interval $[r_i, d_i)$.

Such a model is sufficient for generating some real-time and performance measures.

CHAPTER 3

RELATED WORK

In this chapter we discuss work related to multi-objective resource allocation. We have divided this chapter into two parts. First, we discuss various approaches to the resource allocation problem. Then, in the second part, we look at methods used for evaluating the costs of an allocation.

Previous work on the subject matter is varied and extensive. However, unlike our model, none consider a mixed-task-type environment. That is, all related work assumes the task sets to be entirely real-time or entirely non-real-time. Nor do previous papers consider more than two objectives, as demanded by real computer systems. Let us now review some of these approaches.

3.1 Approaches

The mapping of software to hardware can be performed at various times/states in the life of the system, these approaches include: static, semi-static, or dynamic. For example, prior to running a system, one can evaluate possible alternatives and determine which is best for the selected objectives (*static*). Alternatively, during run time we can monitor the system periodically and migrate tasks accordingly (*dynamic*). However, we cannot always continuously monitor the system because the evaluation process may be too expensive. It may take longer to compute than the currently available free time, leaving less time then needed for tasks to run to completion. Therefore, a compromise must be reached, and heuristic solutions need to be employed. Finally, after tasks have finished running, one can determine the accuracy of our earlier speculations and adjust the allocation for the next time (*semi-static*). We will now compare the approaches and give examples.

Static methods determine the mapping prior to run time. One advantage over other methods is that static methods have ample time to derive solutions. However, there are limitations. Primarily, information may not yet be available for evaluating the criteria. This is a key point when it comes to measuring real-time characteristics of any proposed allocation. For example, timing information necessary for measuring slack, number of missed deadlines, and number of failed tasks will not be available prior to execution of the tasks. Hence, any results from analysis performed prior to run-time are pure speculation.

Run-time approaches are developed to circumvent the incomplete information drawback — only at run time will exact information be available. However, unlike static approaches where speed is not of concern, run-time approaches need to adapt to constantly changing conditions and resolve situations on the fly. These methods must be very quick and efficient, otherwise their solution may come too late.

In all approaches there are two components, *decision-time* — the time decisions are made, either at run-time (dynamic) or prior to run-time (static) — and *data-type* — data can either be fixed (static) or changing with time (dynamic). By making both of these two components the same, either static or dynamic, a method is classified accordingly. However, semi-static approaches are hybrids, characterized by having one of the two components dynamic and the other static.

Hence, semi-static approaches come in one of two flavors. First, the run-time decision can be made between statically generated alternatives. Second, decisions can be made prior to run-time while using dynamic information, such as feedback, profiling, conditional branch probabilities, actual data, and actual message sizes.

Some approaches that have included resource allocation to some degree are covered in the next sections.

3.1.1 Scheduling

Given one or more resources that need to be shared by a set of tasks, *scheduling* is the process of determining which task gets access to which resource, and when. Often, in systems with more than one processor, the hardware model will be homogeneous, with little or no notion of Interprocessor Communication (IPC) [11]. In addition, some scheduling methods bind tasks to processors prior to scheduling analysis, while others assign tasks based on the available processing time and needed execution cycles [11, 23].

Static scheduling approaches for real-time systems [11, 23, 42] typically allocate resources depending on timing requirements of tasks. Historically, these methods have given little consideration to any criteria other than timing. Typically, all processors are assumed to be identical and their number unlimited. Hence, "actual details of the assignment of tasks to processors can be ignored"[12].

More interestingly, some scheduling approaches [11, 18] have implemented resource allocation more directly. These approaches implement a variant of bin-packing [37]. Gupta and Spezialetti, for example, have developed a compile-time mechanism [29] for identifying clusters of tasks that may be interleaved and overlapped based on task busy-idle periods.

In *Dynamic scheduling*, tasks are allocated and scheduled as the system is made aware that they are available to be run [23]. The lookahead time, the difference between the time when a task declares its intention to execute and its latest allowable start time, may vary from 0 to ∞, depending on the application. A survey on scheduling is presented by D.G. Feitelson in [23].

3.1.2 Contention Resolution and Arbitration

Here, consumers/tasks compete a priori or at runtime for exclusive use of system resources. Various arbitration mechanisms [24, 74] have been developed to decide

which task or message is granted rights to the resource in question and when. For example, Stoica [74] assigns each task i an initial amount of symbolic money x_i at its scheduled start time st_i (usually $st_i = 0$) and an inflow rate xr_i. The tasks then bid for the resources they desire. Naturally, resource r_k goes to the highest bidder. The premise is that a task will bid higher and higher as it approaches a deadline or a critical point. This also suffers from the lack of addressing other criteria directly.

3.1.3 Dynamic Reassignment

Here tasks are reassigned during their course of execution to take advantage of changes in locality [10, 76]. At different phases of a task's life, a task may communicate with a multitude of other tasks scattered throughout the network. By migrating a task to another processor, possibly one closer to an upcoming communicating task, IPC traffic can be reduced as well as message delays. In addition, execution cost may be reduced by migrating to a less expensive now available processor.

3.1.4 Resource Reclaiming

This dynamic method recovers unused portions of previously allocated processor time and allocates it to other ready and waiting tasks. The reclaimed time typically arises from pessimistic scheduling policies. Schedules, typically, assume a worst case scenario. They allot processor time according to the longest case execution path of tasks. However, the actual execution time can sometimes be considerably less, resulting in slack which can be harvested and used by other tasks. While Sprunt *et al.* use this slack time primarily to schedule soft real-time aperiodic tasks in [69], Shen *et al.* in [66] used it to reschedule subsequent tasks in the task graph.

In hard-real-time systems, this approach has the effect of shifting ahead the start times of subsequent tasks. At best, the task set finishes early if tasks are

allowed to start earlier than their scheduled start time. The resulting processor idle time is equal to the sum of the differences between expected and actual execution times of all tasks in the task set. Furthermore, reclaiming will have no effect if tasks are unable to start earlier. In either case, processor idle periods are not utilized and processor utilization is not increased. However, in our model, each interval of slack may be allocated to awaiting non-real-time tasks, thereby increasing processor utilization.

3.1.5 Objectives

In computing systems resource allocation efforts are driven by an objective of sorts, be it maximum lateness, slack, makespan, fault tolerance, reliability, communication, or execution costs, just to name a few [1, 15, 36, 40, 75, 84]. However, none of these techniques alone are suitable for Complex Systems, because no single existing approach optimizes more than two non-correlated objectives simultaneously.

Granted, there has been much work done on techniques of addressing multi-objectives outside of computer science, some of which include: *Multiple Criteria Decision Making* (MCDM) [14, 38, 83, 87], *Multiple-Objective Mathematical programming* (MOMP) [16, 67], *Multiobjective Linear Programming* (MOLP) [19], *Multiobjective Goal Programming* (MOGP) [58], and *Multiobjective Fuzzy Programming* (MOFP) [51, 58, 59, 88]. However, there are many inadequacies. First, none handle incommensurability and trade-offs between objectives well in large computer systems. Typically, they divide individual objective values by the number of objectives and add (i.e., $\sum_{i=1}^{n} \frac{1}{n} f_i$ where n is the total number of objectives being optimized over and f_i is the value for objective i). Second, resource allocation in Complex Systems is not amenable to simple mathematical models, due to the high number of variables and coefficients demanded by each cost function. Finally, if it were possible to map the resource allocation problem to a simple mathematical model, it would render

the solution computationally infeasible when applied to Complex Systems, due to the non-monotonicity of aggregate objective cost functions.

Whereas in our method we aim to measure the cost of an allocation by the level of satisfaction of the multitude of non-functional system designs specified, Bailey and Pearson in [9] describe a tool for measuring system user satisfaction through evaluation of level of satisfaction for different factors. They use a linear sum of the weighted averages approach for measuring and analyzing the level of user satisfaction in information systems. They measure 39 factors by means of a questionnaire. Each factor is quantified by four bipolar adjective pairs (good, simple, readable, useful) and a 'scale' (satisfactory), both having the range $[-3, +3]$, and a weight (importance) ranging from 0.1 to 1.0, where the higher the value the more important the factor. The scale component is not meant to be used in the calculation, but rather, to test the internal validity of the tool. It is used in the normalizing step to eliminate side effects of factors that are not relevant to users.

Bailey and Pearson do handle a multitude of factors. However, all factors are assumed or defined to be of the same type and range. Therefore, all measures are commensurable and no scaling or fusion is performed. Furthermore, what they refer to as the 'weight' does not differentiate level of importance of different factors; it is simply a multiplicative factor. Also, neither the type nor the size of the system is taken into consideration. This approach is very subjective. It deals only with user opinion and perception, rather than concrete measures.

In this section we have been concerned with different types of resource allocation approaches and when to use them. In the next section, we turn our attention to the actual computation method.

3.2 Evaluation Methods

Several approaches to the allocation evaluation method have been used, including graph theoretic, integer programming, and heuristic techniques. All three approaches are addressed and surveyed in [10, 15, 21, 45, 76]. While mathematical methods provide an optimal solution, they are computationally intractable for non-trivial systems. Similarly, graph theoretic methods are NP-hard when more than two processors are included in the model. In addition, it is difficult to represent more than two optimization criteria in graph theoretical methods. Therefore, heuristics are typically developed to be fast and efficient but not necessarily optimal.

The next section is dedicated to reviewing the three methods, namely: graph theoretical, integer programming, and heuristic.

3.2.1 Graph Theoretic

In graph theoretical methods [1, 15, 65, 75, 76, 10], graphs are used to model the system. Typically, nodes are split into two sets, one for *processors* and the other for *tasks*. Edges between task nodes represent *interprocess communication* and those connecting nodes between the two sets, *execution cost*. Then graph theoretic techniques are applied to the graph in such a way as to minimize certain criteria. A problem with this method is that it is very difficult to handle more than one or two optimization criteria. Also it becomes computationally intractable when more than two or three processors are used.

Some of the graph theoretic techniques employed in resource allocation include:

- **[Min-cut]** In this approach, a graph is cut into n sets of nodes, each containing exactly one processor, in such a way as to minimize the sum of the weighted edges that were cut. Each set represent a mapping of task nodes to the processor node contained within that set. Stone and Bokhari in [75, 76, 10] showed this method provides the minimum cost allocation for 2 and 3 processor

systems in $\mathcal{O}(N \cdot E)$ where N is equal to number of nodes and E is the number of edges in the graph. However Chu *et al.* in [15] noted that it does not consider load balancing, limitations on memory storage sizes, queuing delays, nor does it consider precedence relations. Also, it quickly becomes computationally intractable as the number of processors rises above three.

- **[Clique partitioning and split graphs]** Given a graph G(E,V), a *clique* is defined as a set $U \subset V$ of pairwise adjacent nodes. A graph G can have more than one clique. In clique partitioning, nodes assume one of two possible types, a software task or processor. Edges between different node types represent execution cost. Edges between task nodes represent communication cost. The idea is then to minimize the sum of the total edge weights between cliques [1].

- **[Branch and Bound (B&B)]** Here, the state search space assumes a tree structure. Then, system constraints allow the exclusion of certain solutions from examination [45]. That is, constraints prune branches from the search tree, thereby, dramatically reducing the search space. With sufficient constraints the search space can be made manageable. However, due to non-monotonicity of cost functions, there is no clear path of decreasing costs from the root to a full allocation. Furthermore, B&B may inadvertently preclude better allocations if one is not careful.

3.2.2 Mathematical Programming

In mathematical programming approaches, the task allocation problem is modeled as an optimization problem. Mathematical programming techniques are then applied so as to maximize (or minimize) an objective function subject to a set of constraints which define feasibility [16]. Some of the mathematical programming techniques employed in resource allocation include: linear programming [15, 40, 45], goal programming, and multiobjective programming [9, 16, 50].

All of these techniques provide for a better representation of the task allocation environment over graph theoretical ones. Because these approaches permit the introduction of constraints into the model where appropriate for the system, whereas graph theoretical methods do not. However, these methods have shortfalls in Complex Systems. For example, mathematical programming models require attributes to be mutual preferentially independent, have like units, and like scales. Furthermore, allocation only data is insufficient for generating costs. Real-time cost-functions can not be computed without a schedule and/or symbolic execution data.

Moreover, mathematical programming approaches are limited by the amount of time available to compute solutions. That is, optimal solutions are computationally intractable. Typically, these methods require exponential time for computations. Therefore, new techniques need to be devised to reduce the amount of computing time required for the resource allocation problem. In addition, new techniques should be scalable to larger dimensional problems. Finally, an optimal solution may not be necessary where a near-optimal will suffice.

3.2.3 Heuristics

Resource allocation is NP hard [1, 25] when two or more resources are implemented. Therefore, heuristics, while not providing an optimal solution, need to be used to arrive at a solution for the given problem in a timely fashion. However, they should be as nearly optimal as possible.

In dynamic heuristic approaches [18, 66, 69], fast, efficient, and deterministic methods for finding a good allocation solution are crucial for tasks in real-time systems not to miss their deadlines. In static approaches, [1, 21, 65, 78, 84], fast timely solutions are not as crucial. However, consider that a small system with 5,000 tasks and 100 computers has a solution space of size 100^{5000}. Exhaustive search may

not be desirable nor affordable. Therefore, better heuristic methods are needed for static analysis.

Heuristic models tend to make assumptions and generalizations in order to simplify problems. The trade-offs with heuristics are that sometimes they oversimplify the environment and the solution they provide loses meaning with respect to the actual system. Furthermore, heuristics now available do not consider more than one or two objectives. Currently in real-time systems the concerns are utilization and completion times, while in non-real-time systems the focus of most research is on IPC, execution cost, and LB.

3.3 Where we Stand with Complex Systems

There are two issues at hand, *tractability* and *representation*. Heuristic methods may be tractable but their system model may not be representative of the actual system and their solution not applicable to the original problem. Graphic and mathematical methods may employ more representative models but their solutions are typically intractable.

Furthermore, as we have shown, current approaches fall short in representing the multitude of objectives found in Complex Systems. All consider the task sets to be exclusively comprised of hard, soft, or non-real-time tasks and are limited by two objectives. These restrictions weaken the model and the solution. All these deficiencies further justify the need for a heuristic *Resource Allocation Tool* in Complex Systems.

CHAPTER 4

EVALUATION OF NON-FUNCTIONAL SYSTEM DESIGNS

As we have seen in the previous chapter, little work has been directed at handling more than two non-functional objectives. However, the problem of structuring cost functions is not new. The issues are significant in and outside of Computer Science. One of the earliest mentionings in the literature is that of Lewis Carroll in [13], 1880. He suggests various cases which we will present in section 4.5.5.

In this chapter, work done on structuring and assessing of combining functions is introduced, adapted, and enhanced. Most of the work is outside of Computer Science, in fields such as Mathematics, Economics, and Operations Research. First, issues regarding objectives, objective decomposition, and attribute cost functions are discussed. Next, the existence of combining models is discussed. Then, a hierarchical model is presented along with steps for deriving the corresponding system cost function. Finally, an example is given.

4.1 Objectives

Recall that the non-functional goals of the system design process for an application may be divided into *constraints*, whose failure will result in the rejection of an allocation, and *objectives*, whose relative degree of satisfaction affects in the acceptability of a proposed design. Constraints typically arise from physical restrictions on the underlying application, or on the partially-specified platform or design, but can also be hard user requirements on acceptable designs. They can be either implicit or explicit, as in [32]. Objectives, in contrast, represent behavior desired for the system; objectives frequently can be satisfied to a greater or lesser degree. Measuring the relative level of satisfaction of an objective or set of objectives will call for quantification of these levels.

When a design decision has been taken, here an allocation, the outcome has to be evaluated with respect to the system objective. For evaluation purposes, a numerical value is assigned to the top-level objective, which corresponds to the quality of the decision. The need for quantification of objectives raises a number of issues: first, defining a suitable metric; second, obtaining values for the metric from an incompletely defined and non-executing platform. Many design decisions are made before the system itself is implemented. To obtain values for a design requires simulation. We do not address these two points here. However, the third issue related to quantification is the usage of these values to identify a good design, which is the topic of this thesis. This issue comprises an additional subtlety: a given objective can often be realized by multiple independent metrics; we would like the freedom to combine their results into a single, presumably more precise metric.

There are several obstacles to identifying a good allocation. First, since satisfaction of objectives is relative, designs cannot be evaluated in isolation; even a proposed design in hand, identification of a good design typically involves comparison. Second, since objectives may interact and even interfere, it is neither sufficient nor always possible to optimize for each objective separately. In general, we will again need to combine metrics, this time for different objectives, into a single numerical value reflecting the quality of a system design.

4.1.1 Quantification of Objectives / Identification of Attribute Cost Functions

Objective satisfaction is quantified through cost functions. Some of the many cost functions generally encountered in complex systems as outlined by Nguyen and Howell in [54] are communication cost, load balancing, elapsed time, failure rate, deadline satisfaction percentage, relative locality, and risk level.

In Figures 4.1 through 4.5 we quantify some of these objectives into cost functions, and give qualitative descriptions for each function. First, we define two

Performance:
- Allocation only
  - load balancing
    - $min\{\texttt{heaviest\_loaded\_PE} - \texttt{least\_loaded\_PE}\}$
  - reduce communication
    - $min\{\sum \texttt{Network\_communication\_traffic}\}$
    - $min\{max(\texttt{communication\_traffic\_through\_link})\}$
    - $min\{max(\texttt{communication\_traffic\_through\_switching\_node})\}$
- Symbolic execution
  - reduce message contention
    - $min\{\texttt{number\_of\_conflicts\_at\_all\_links}\}$
    - $min\{\max \texttt{number\_of\_conflicts\_at\_a\_link}\}$
  - reduce switching node contention
    - $min\{\texttt{number\_of\_conflicts\_at\_all\_nodes}\}$
    - $min\{\max \texttt{number\_of\_conflicts\_at\_a\_node}\}$
    - $min\{\texttt{number\_of\_conflicts\_at\_all\_nodes} + \texttt{number\_of\_conflicts\_at\_all\_links}\}$

Figure 4.1 Performance: Objective to Cost Function Mapping

types of cost functions: static and semi-static. While *static cost functions* can be measured from the allocation only data, *semi-static functions* need additional information that can only be acquired through actual or symbolic execution of the tasks, some of which include: task finish time, contention, and precedence related delays. Run-time cost functions can be defined. However, their evaluation will be postmortem/after the fact. Therefore, they are of no use in a dynamic sense for resource allocation.

In the section, we give an example of attribute cost function identification and objective quantification.

Real-Time:

- Allocation only
  - deadline balancing
    - minimize the minimal deadline difference
  - Symbolic execution
    - maximize slack
      - $max \sum deadline_i - finishtime_i \quad | deadline_i > finishtime_i$
    - reduce the number of missed deadlines
      - $min \sum i \quad | deadline_i < finishtime_i$

Figure 4.2 Real-Time: Objective to Cost Function Mapping

Security:

- Allocation only
  - provided vs. demanded (assume task assigned to PE)
    - $max_{tasks} max((TaskSecLevel - PESecLevel), 0)$
    - $\sum_{tasks} max((TaskSecLevel - PESecLevel), 0)$
    - $\sum_{tasks} max((TaskSecLevel / PESecLevel), 1)$
    - $max_{TaskSecLevel > PESecLevel} TaskSecLevel$
    - $\sum_{msgs} \sum_{link_i n_m sg} max((MsgSecLevel - LinkSecLevel), 0)$
    - $\sum_{msgs} \sum_{link_i n_m sg} \sum_{MsgSecLevel > LinkSecLevel} msg.size$
  - physical locality
    - $max_{source.loc != dest.loc} MsgSecLevel$
    - $max_{TaskSecLevel > LocSecLevel} TaskSecLevel$
    - `number of sites used`
  - Symbolic execution
    - $\sum_{TaskSecLevel > PESecLevel} task.time$
    - $\sum_{MsgSecLevel > LinkSecLevel} msg.size / link.rate$

Figure 4.3 Security: Objective to Cost Function Mapping

Pure Logic:

Allocation:

number of allocation preferences violated, e.g.,

$\sum_{PrefAssign(T,setAofP)}(T.proc_not_inA)$

$\sum_{PrefNoAssign(T,setAofP)}(T.proc_inA)$

number of collocality preferences violated

$\sum_{PrefCoAssign(T1,T2)}(T1.proc! = T2.proc)$

$\sum_{PrefNoCoAssign(T1,T2)}(T1.proc = T2.proc)$

number of cloning preferences violated

etc.

Symbolic Execution:

number of strong precedence preferences violated

$\sum_{PrefPrec(T1,T2)}(T1.end > T2.start)$

number of weak precedence preferences violated

$\sum_{PrefPrec(T1,T2)}(T1.start > T2.start)$

$\sum_{PrefPrec(T1,T2)}(T1.start > T2.end)$

Figure 4.4 Pure Logic: Objective to Cost Function Mapping

Human factors:

Allocation:

number of human operators needed [needs a lot more info]

max number of inputs managed by operator

Figure 4.5 Human Factors: Objective to Cost Function Mapping

4.1.2 Deadline Balancing: An Example

Here, as an example of objective and cost function identification, we introduce the concept of deadline balancing. We trace its development, step by step, from objective to attribute cost function.

The objective (o) in *Deadline Balancing* is to avoid allocations where the difference in time of the deadlines of tasks assigned to the same processor is small. In other words, we aim to avoid allocations with bunched-up deadlines on any particular processor. Problems may arise when tasks with approximately the same deadlines are assigned the same processor or processor group. These include:

- Processor saturation or overloaded states. All tasks assigned to a processor may have close deadlines within a interval of time.
- Processor idle states. If all tasks have to finish within a certain interval, afterwards, there will be no more work to be performed.
- Network communication bottlenecks. Certain communication links may always be busy during processor saturated intervals.
- Poor handling of sporadic or aperiodic tasks. Problems may occur when these tasks arrive during saturated states.
- Poor load balancing at any particular point in time. However, over an interval load balancing may be optimal. This effect is due to overloaded and idle states.

Since deadline balancing prefers allocations where the deadline difference between tasks assigned to the same processor is maximal, it will tend to spread out the deadlines over an interval, thereby reducing the occurrence of the saturated and idle states and their anomalies.

4.1.2.1 Definitions and Assumptions: The *deadline difference* $dd_{i,j}$ between two tasks i and j assigned to any one particular processor k is defined to be the absolute difference in time between their deadlines.

$$dd_{i,j} = \begin{cases} |dl_j - dl_i| & \\ \infty & \text{if at most one task has a deadline} \end{cases} \tag{4.1}$$

The *minimum deadline difference*, mdd, for processor k, is represented as:

$$mdd_k = \texttt{min}\ dd_{i,j}$$

where i and j range over all tasks assigned to k.

Allocations where task deadlines are most spread out are preferred, as opposed to ones where the deadlines are close to each other for those tasks assigned to any one particular processor. That is, the allocation with the highest *Minimal Deadline Difference*, MDD, is optimal. Therefore, we define MDD over m processors for a given allocation a as:

$$MDD_a = \texttt{min}\ (mdd_k) \quad | \ 1 \leq k \leq m$$

Note that the higher this value, the better the allocation.

Since minimization of cost functions represents the development towards a better allocation, we need to convert this into a function where approaching zero represents a better solution. Hence, *Deadline Balancing* (DLB) cost is defined as the variance in time between the minimal deadline difference and a significantly large constant β in the allocation a.

$$DLB_a = \begin{cases} \beta - MDD_a & |MDD_a \neq \infty \\ 0 & |MDD_a = \infty \end{cases} \tag{4.2}$$

The allocations with lowest DLB cost are the allocations that have tasks' deadlines most spread apart; in other words, for all processors, the smallest

separation between deadlines of tasks on the same processor is the greatest. The opposite can be said for allocations with the maximum DLB cost.

4.1.2.2 Computing Deadline Balancing Cost: The steps to find the DLB cost for a feasible allocation a are as follows:

1. For each processor k, where k ranges from 1 to m, create a heap H_k of the n_k tasks mapped to it, partially ordered on task deadlines. If a task does not have a deadline, disregard it, and do not insert it into the heap.

2. Remove all elements from heap H_k and compute the deadline difference $dd_{i,j}$ between successive elements, retaining the lowest value (mdd_k), once per processor.

3. Determine the global Minimum Deadline Difference, MDD_a, from the remaining m mdd's.

4. Evaluate the DLB_a, where β is equal to the largest global deadline.

4.1.2.3 Complexity: For n tasks and m processors, the first step of the algorithm, heapifying, is $\mathcal{O}(n \times lg\ n)$ in time while requiring $\mathcal{O}(n)$ space for storage. The second step is the most demanding, both in time and space. Each heap deletion requires $\mathcal{O}(lg\ n)$ compare and swap operations and a single subtraction. Since there may be n elements, total time for all deletions will be $\mathcal{O}(n \times lg\ n)$, with $\mathcal{O}(n-1)$ subtractions. $\mathcal{O}\ (m)$ additional space will be required to store individual mdd's results. For this step, total time required is $\mathcal{O}\ (n \times \lg\ n)$ and total space is $\mathcal{O}\ (m+n)$. The last two steps, determining MDD_a and DLB_a, are linear. The third requires $m-1$ comparisons to find MDD_a. Additional space complexity is $\mathcal{O}\ (1)$ for both.

Total time complexity is $\mathcal{O}\ (n \times \lg n + m)$ and total space complexity is $\mathcal{O}\ (m+n)$. This simple algorithm can be implemented on any machine.

In this section we have introduced, as an objective, deadline balancing (DLB) and its representative cost functions. We will extend this notion of DLB into the concept of scheduling to maximize inter-completion time —Inter-Completion Time Scheduling (ICTS)— in Chapter 5.

4.2 Combining Models

Combining models are aggregation function forms. They take as input two or more values and output a single value representative of the form used. Typically, they take the shape of a mathematical expression. In the following sections, the existence of combining models are discussed.

4.2.1 Arrow's Theorem

In this section the implications of Arrow's theorem are investigated. Simply put: Given the rankings of a set of alternatives, what should the overall ranking be? Arrow proposed some relatively reasonable axioms on the aggregation of the rankings, and studied their consequences. We briefly outline these axioms and theorem here with adaptation to our allocation problem. For a more in-depth discussion see [38, p. 523] or [60, p. 433]. The axioms are as follows.

Axiom 1 Complete domain. There are at least two individual decision makers, three alternatives, and a group ordering is specified for all possible individual orderings.

Axiom 2 Positive association of social and individual orderings. Given a group's ordering where alternative $\mathcal{A}$ is preferred to alternative $\mathcal{B}$, if individuals change their preference in $\mathcal{A}$'s favor then the group consensus must still prefer $\mathcal{A}$ to $\mathcal{B}$.

Axiom 3 Independence of irrelevant alternatives. If elimination of an alternative does not effect individual preference orders for the remaining alternatives, then the groups' consensus for the remaining alternatives should also be unchanged.

Axiom 4 Individual's sovereignty. For every alternative $\mathcal{A}$ there should be some set of individual orderings such that the groups consensus prefers $\mathcal{A}$ over any other alternative.

Axiom 5 Non-dictatorship. There is no individual decision maker who dictates the groups consensus regardless of other decision makers.

Theorem 1 (Arrow's Impossibility Theorem). Suppose a set of alternatives A has at least three elements and the number t of individuals (objectives) is at least two. Then axioms A1,A2,A3,A4, and A5 are inconsistent.

Proof: See [60, p. 440]. ■

Thus, there is no way of combining a set of rankings over a set of alternatives to obtain a single ranking simultaneously satisfying the five axioms. How are we to proceed? We have two possibilities:

1. relax some of the axioms, or

2. add information to the model.

We have chosen the second alternative in subsequent sections. For a detailed discussion on these possibilities see Luce and Raiffa [44].

Hence, simple combinations of ranks, regardless of scale type, will not work. A mechanism for combining values is needed.

4.2.2 Existence of Combining Models

Keeney and Raiffa show and prove that Arrow's axioms hold for additive conjoint measure on ordinal models in [38]. However, attributes must be mutually preferential independent. In this section we argue for a model that does not have this restriction of preferential independence.

Since objectives are constructed from subjective opinion, they may overlap to a certain degree. This overlap may suggest one of two possibilities. First, the specifier does not know what they are doing; or second, the specifier places significantly more importance on the overlapping attributes. The former case we may proceed by attempting to remove the doubly counted components as in set theory, union of sets. However, the later case is what will be argued in this section.

If the specifier is fully aware of her specification and intends it to be as such, this implies a weighted additive model. A small example follows.

Example 1 Suppose we are evaluating a system of m completed real time tasks by objective X with production $X ::= X_1, ..., X_n$. In addition, all lower level objectives (X_i) are mutually preferentially independent except for two. They are 'number of missed deadlines' X_{mdl} and 'lateness' X_{lat}, whose cost functions are defined as follows:

$$C_{mdl} = \sum_{i=1}^{m} 1 \qquad | \; ct_i > dl_i$$

$$C_{lat} = \sum_{i=1}^{m} ct_i - dl_i \qquad | \; ct_i > dl_i$$

where, $ct \stackrel{\text{def}}{=}$ completion time; $dl \stackrel{\text{def}}{=}$ deadline.

How should the corresponding costs $x_1, ..., x_n$ be combined? The additive forms, as stated by Keeney and Raiffa in [38, p. 111], cannot be used due to the dependence of X_{mdl} and X_{lat} on dl. However, in a weighted model, we can argue that the reason a specifier has created a dependency is because she places that much more importance on the overlapping attributes. Furthermore, she intentionally wants this bias to be

reflected in the aggregated evaluation. With this in mind we present the following Axiom:

Axiom 6 Intelligent specifier. An objective function specifier is fully aware and understands the implication of the objective structure she specifies.

This axiom asserts that an objective function specifier not only understands the structure she has specified but has done so intentionally. Hence, the overlapping of attributes implies an additional weight on the overlap and we present the following theorem:

Theorem 2 Given attributes $X_1, ..., X_n$ and subjective weights $w_1, ..., w_n$, an additive value function

$$v(x_1, x_2, ...x_n) = \sum_{i=1}^{n} w_i v_i(x_i) \tag{4.3}$$

(where v_i is a value function over X_i) exists if Axiom 6 is satisfied.

Proof: Let lower level objectives $X_1, ..., X_n$ and corresponding weights $w_1, ..., w_n$ define a production of X. Suppose X_a and X_b, with weights w_a, w_b, are not mutually preferential independent by correlation functions $r_{(a,b)}, r_{(b,a)}$. Assume for now that they are additive and following equation holds,

$$w_a X_a + w_b X_b = w'_a X_a = w'_b X_b \tag{4.4}$$

where $w'_a = w_a + r_{(a,b)}$ and $w'_b = w_b + r_{(b,a)}$.

Preferential independence now holds on

$$v(x_1, x_2, ...x_n) = ((\sum_{i=1}^{a-1} w_i v_i(x_i)) + w'_a X_a + \sum_{i=a+1}^{n} w_i v_i(x_i) \quad |i \neq a, b$$

and

$$v(x_1, x_2, ...x_n) = (\sum_{i=1}^{b-1} w_i v_i(x_i)) + w'_b X_b + \sum_{i=b+1}^{n} w_i v_i(x_i) \quad |i \neq a, b$$

Finally, substituting from equation 4.4 yields:

$$\begin{aligned} v(x_1, x_2, ...x_n) &= (\sum_{i=1}^{a-1} w_i v_i(x_i)) + w_a X_a + w_b X_b + \sum_{i=a+1}^{n} w_i v_i(x_i) \quad |i \neq a, b \\ &= \sum_{i=1}^{n} w_i v_i(x_i) \end{aligned}$$

■

Hence, dependencies correspond to increased weights on the dependent attributes in a corresponding mutual preferential independent form. We will be using this model for our work in the balance of this thesis.

4.3 A Hierarchy of Objectives

For the following considerations we have to distinguish two levels, a decision maker level and a system level. The former hosts the objectives and their decomposition, while attribute and atomic cost functions reside on the system level.

We view objectives as defined hierarchically as in [38]. However, our representation takes the form of a Directed Acyclic Graph (DAG), where as Keeney and Raiffa's [38] representation takes on the form of a tree (see Figure 4.6). There is root objective, the *system objective*, which in a given application comprises the top-level *system design objectives*, such as performance, real-time, security, and so on. These may in turn be refined into smaller scoped objectives. For example, *performance* may include issues of response time, throughput, load balancing, and so on, as in the original DESTINATION documentation [54], see Table 4.1. At some point the recursion stops. The lowest-level objectives that are not decomposed any further are also referred to as *attributes*. Each of the lower-level objectives is realized by one or more of the given predefined *attribute cost functions*, which constitute the interface between the decision maker and the system level.

On the system level there may be a similar decomposition, which is predefined and is not accessible to the decision maker. The attribute cost functions are made

Table 4.1 Objective Decomposition

Top-level Objectives		Low-level Objectives
Performance	allocation only	load balancing (LB) communication (IPC) throughput locality
	symbolic execution	elapsed time LB with contention IPC with contention queue size utilization
Real-Time	allocation only	deadline balancing (DB)
	symbolic execution	extended DB slack laxity latency missed deadlines consecutive missed deadlines % of missed deadlines highest priority missed
Fault-Tolerance	allocation only	active link failure active PE failure
	symbolic execution	same including time
Security	allocation only	provided vs. desired physical locality
	symbolic execution	same including time
Human Factors	allocation only	number of inputs number of operators
	symbolic execution	time between interactions
Pure Logic	allocation only	preferences violated cloning violations collocality violated
	symbolic execution	collocality violated w/migration

up of *atomic cost functions* according to combination rules that may be similar to those at the decision maker level.

It is important to note that the decision maker perceives only the attribute cost functions. Their decomposition is hidden to him. Furthermore, we do not discuss atomic cost functions. In addition, we assume a one to one relation between atomic and attribute cost functions.

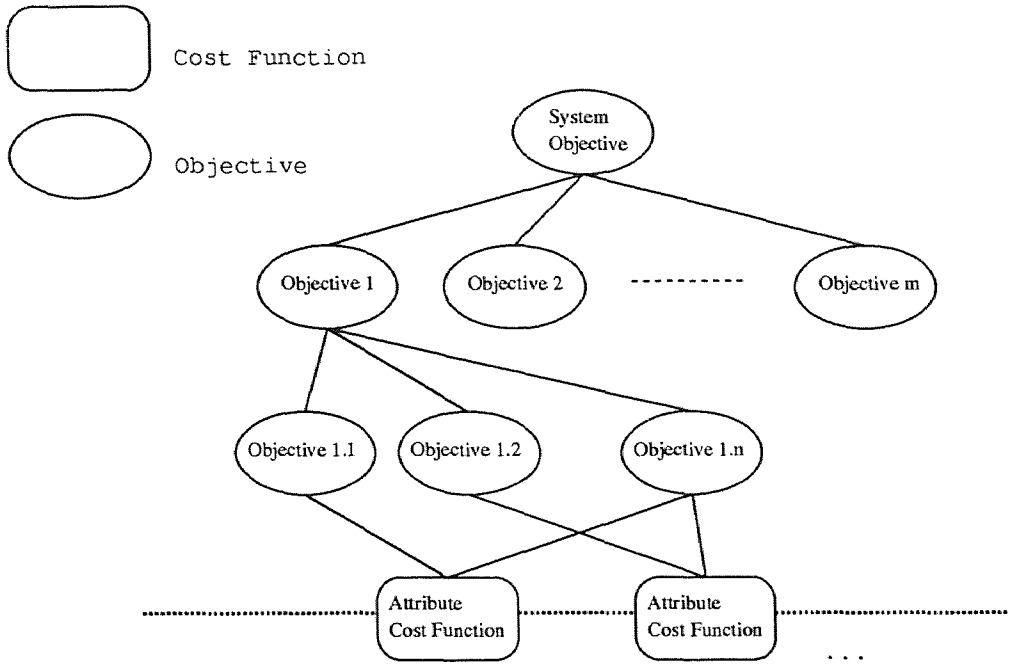


Figure 4.6 General Objective Structure

The upper and lower-level objectives and cost functions form a Directed Acyclic Graph (DAG). To ensure the logical flow of a hierarchy Mollaghasemi and Pet outline in [52] four tests that have been recommended by J. Gibson. We have adapted these four tests and present them as follows:

1. **[How]** Reading "down" any branch, each objective must answer the "how" of its immediately higher goal.

2. [**Why**] Reading "up" any branch, each higher objective answers "why" the objective above it is needed.

3. [**Necessary**] Reading "across" the objectives at a given level under any one general goal, the question *are all the more specific objectives necessary to accomplish the more general objective?* must be asked.

4. [**Sufficient**] Reading "across" objectives at a given level under any one general goal, the question *are the specific objectives sufficient to accomplish the more general objective?* must be asked.

4.4 System Cost Function Derivation

A two-step approach is taken to create a system cost function that reflects the system objective. The process is illustrated in Figure 4.7.

The approach starts with a given set of attribute cost functions that are available to the decision maker. The decomposition of these attribute cost functions into atomic cost functions is hidden behind the interface (Figure 4.7.a).

In the first step the system objective and the design objectives, which have been identified during analysis, are decomposed into smaller-scoped objectives. Finally, the lowest-level objectives are mapped to the attribute cost functions (Figure 4.7.b). Each objective can be expressed by one or more attribute cost functions.

During the second step—system cost function synthesis—the objective decomposition DAG is traversed in bottom-up manner. The cost function at each node is thereby built by combining the cost functions of its children (Figure 4.7.c). When the root node, the system objective is reached, the system cost function has been obtained. This important step is described in more detail in the following section.

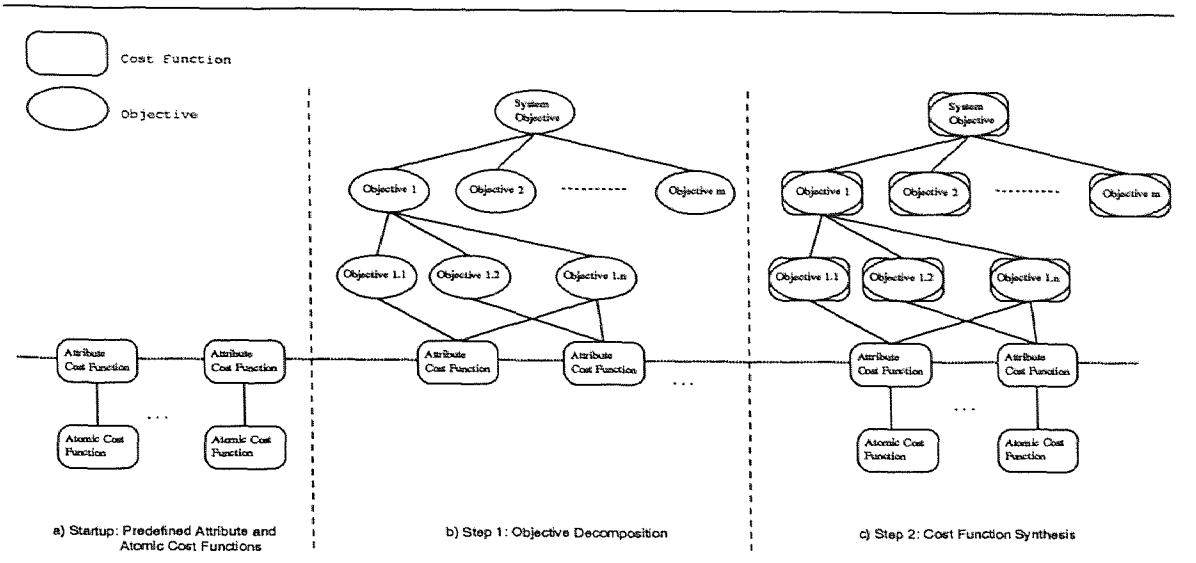


Figure 4.7 Steps Towards Cost Function Synthesis

4.5 Combining Cost Function Values

The upper and lower-level objectives and their corresponding cost functions, as well as the attribute and atomic cost functions form a Directed Acyclic Graph (DAG). This thesis aims at a general approach to combining values associated to children of a node into a value associated with the node itself. The value associated to the node *system* is then the relative measure of the "goodness" of the proposed design. Since each of these computations can be viewed as an attribute synthesis rule, we treat the result as a function defined in terms of the values at the children; composition of the functions at each node will define a synthesized function at the root. In other words, the attribute cost functions are combined in a bottom-up manner until the system cost function at the root node of the DAG is determined.

While in many cases the value at the parent results from a simple combining operator applied to values (possibly transformed) of the children, there may be cases where the interaction between child values is significant and algebraically complicated. In those cases, we usually will prefer to keep the combining operator

simple, and instead consider separate terms for the interaction. The combining rule is outlined in Figure 4.8.

```
if (at a leaf)
then evaluate the cost function
else
     evaluate the cost functions at the children
     for each child node
          transform the child values appropriately
     combine the resulting values
return (the resulting value)
```

Figure 4.8 Combining Rule

The rest of this section proceeds as follows: First, we consider the inputs to the evaluation process. Second, we discuss the transformations that may have to be applied to combine values. We then give the general form of the function synthesis rule and guidelines to combining functions.

4.5.1 Inputs for Cost Function Synthesis

The inputs to cost function synthesis include:

- The system description s
- The top-level system objectives o.
- The proposed solution a
- Auxiliary information φ derived by tools . For example, in order to evaluate a number of cost functions (e.g., percentage of soft deadlines satisfied), we need an approximate execution profile/static schedule.

4.5.2 Value Transformation

Transformations are required for various purposes in the process of combining values. These may be applied in any necessary order to the value(s) at a child node (or set of nodes) to derive a value used in the combining process. It may even be desirable to apply the same type of function several times in the process of transformation. These functions are not necessarily different in form, but rather are distinguished by purpose. The sequence of function applications can be considered to form one single transformation ($\mathcal{D}$).

The components of this *data transformation function* ($\mathcal{D}$)include:

- **[Scale Types $\mathcal{T}$]** Each measurement of an attribute by an attribute cost function belongs to a certain *Scale*. Different scale types are organized by the amount of information that they carry, which are in decreasing order: *Absolute, Ratio, Interval, Ordinal, and Nominal.* This classification is based upon the *meaningfulness* [60] of a statement of measurement. Moreover, scale types are defined by the type of admissible transformation allowed. This type of classification is due to the work of S.S. Stevens [72] and [73]. F.S. Roberts used this approach in [60] to derive the classification we outline in Table 4.2.

  The simplest example of a scale type is one where the only admissible transformation is identity ($\phi(x) = x$). Here, there is only one way to take measurements. Such a scale is called an *absolute scale.* As an example of an absolute scale consider *number of missed deadlines.* If a statement says there are n missed deadlines, it means exactly n, and there is no admissible transformation except the identity which changes this. For a more detailed explanation and examples see [60].

  Previously, we have mentioned that Arrow's axioms hold for additive conjoint measure on ordinal models [38]. Since absolute, ratio, and interval scales by

Table 4.2 Some Common Scale Types—Adapted From Roberts, 1976, p. 493.

Scale Type	Admissible Transformations	Strongest Test	Examples
Absolute	$\phi(x) = x$ (identity)	$a = 2b + \beta$	Counting # of missed deadlines
Ratio	$\phi(x) = \alpha x \quad \mid \alpha > 0$ Similarity transformation	$a > 2b$ $\frac{a}{b} > \frac{b}{c}$ $\frac{a}{b} > \frac{c}{d}$	Mass Temperature (Kelvin) Time(intervals) Lateness
Interval	$\phi(x) = \alpha x + \beta \quad \mid \alpha > 0$ Positive linear transformation	$a > b + \beta$ $a - b > c - d$	Temperature (Fahrenheit) (centigrade, etc.) Time (calendar)
Ordinal	$x \geq y$ iff $\phi(x) \geq \phi(y)$ (Strictly) monotone increasing transformation	$a > b$	Preference Hardness Air quality Grades of leather, lumber, etc.
Nominal	Any one-to-one ϕ	$a = b$	Labeled alternatives Numbers on uniforms

definition constitute stronger implications on a measurement they can all be coerced down to an ordinal scale. Therefore, we present the following theorem.

Theorem 3 Arrow's axioms hold on absolute, ratio, and interval structured conjoint measurement.

Proof: Proof is implicit by definition of scales types, since the axioms hold for additive conjoint measure in ordinal models. ■

The first step in aggregating measurements is to identify the scale type of the individual components – in our case, lower level cost functions. If they are found to be of different type then they must be coerced to a single scale before any combination can be performed. Coercion may proceed in one of two directions, up or down in the order of meaningfulness. Coercing down is easy

but information is lost. Coercing up is harder because we assume information about the measure, which may not be true.

- [**Scaling** $\mathcal{S}$] The value ranges or the variability of functions at different children may differ significantly. A scaling function transforms the anticipated values at those nodes into comparable values. There are two important issues. One is that we may not want to transform the full range of values, but only "typical values" or "typical values of good allocations"; thus construction of a scaling function may involve creating a test set and determining values, and computing statistical properties, such as maximum, minimum, mean, median, variance, and deciles, of the test set or a filtered subset, and using those to construct the scaling function. The other is that we may not always *want* to make the value ranges identical. If, for example, different child values represent different aspects of total time, we may just want to add times without scaling. Thus we obtain a more accurate estimate of total time than if we had scaled anticipated values to be comparable. Using this refinement requires more detailed semantic knowledge on the part of the evaluation engine.

- [**Fusion** $\mathcal{F}$] Cost functions and different lower level objectives may be expressed in incommensurable units. A fusion function creates commensurability. Some existing fusion functions include: normalization, conversion to money or time, and tradeoff functions. Fusion will be conceptually the most difficult of the transformations, due to the dependency on the data that it is operating on. It is defined by its current domain, where the addition of another data item (child) may disqualify its current instantiation at a node.

- [**Algebraic Transformation** ϕ] An algebraic transformation function corresponds to fitting a given "error" or "penalty" model, so that, for example, an exponential function penalizes a few large values in comparison to average

values for all inputs, while a square-root function does the opposite. Most often, the algebraic transformation function is identical for all child values.

- [**Weight** w] Weight functions (often simply constant multipliers, or powers in a multiplicative model) reflect the relative importance of the information at different children. These are typically provided by the decision maker or design elements for the hierarchy.

In addition, there are three sets of functions, of very different forms:

- [**Combining Functions** β] These take the transformed results for all child nodes, or pairs of child nodes and return a single result. Most often, the combining function is a simple binary associative operator, or such an operator followed by a simple unary operator such as reciprocal or negation.

- [**Interaction or Correlation** θ]Measures the contribution to the total cost based on the current values, and the nature of the interaction. For many applications, if there is interaction, it is nonetheless sufficient to consider interactions of pairs, as in statistical ANOVA and regression analysis. We need not concern ourselves with correlation functions. As argued in Section 4.2.2, dependencies correspond to increased weights on the dependent attributes in a corresponding mutual preferential independent form.

- [**Divergence** Δ] Objectives may not be fully quantifiable or their metrics for quantification not fully known. Moreover, it may be known that the metrics used in the quantification may over or under stipulate the intended meaning of the objective. Therefore, accountability for this deviation is needed. Hence, a divergence function relates the difference between an objective and the metrics used to quantify it. In addition, this function also serves as a reliability measure of the quantification. In this thesis, we will consider the objectives to be fully

quantifiable and the cost functions fully known. Hence, we will largely ignore divergence functions.

This multitude of transformation functions raises two fundamental questions: In what order should these functions be applied? Should each type of function be applied only once, or can the same class of function be multiply applied? It is clear that the answers are highly dependent on the domain and the sets of attribute cost functions used. Undoubtedly, reasonable transformations and sequences of transformations can only be constructed using a combination of domain characteristics, expert knowledge, and application parameters, refined by testing and past experience.

4.5.3 Notation

In this section we outline the notation used in the balance of the chapter.

s	system characteristics
o	top-level non-functional objectives
a	a solution
φ	auxiliary information
X_i	node i in the objective hierarchy
k_i	number of children of X_i
$\mathbf{x}_i$	children evaluation vector; $\mathbf{x}_i \equiv (x_1, x_2, ..., x_{k_i})$
o_i	objective i
C_i	cost function of o_i
I_j	the j^{th} atomic cost function
x_i	value for C_i
$\mathcal{I}$	identity function
$\mathcal{T}$	scale type function
$\mathcal{S}$	scaling function
$\mathcal{F}$	fusion function
ϕ	algebraic transformation function
w	weight function
β	combining function
$\mathcal{D}$	data transformation functions: $f(\mathcal{T}, \mathcal{S}, \mathcal{F}, \phi, w, \beta)$

In the next section we present the cost function synthesis rule.

4.5.4 Cost Function Synthesis Rule

Let a decomposition in the DAG describe the children of an arbitrary node X_i. Then the value for its objective o_i is given by:

$$C_i(s, o, a, \varphi) = \mathcal{D}_i(< \mathbf{x}_i(s, o, a, \varphi) >) \,, \tag{4.5}$$

where C_i is a cost function.

If X_i is a leaf node then $\mathcal{D}_i = \mathcal{I}$, where $\mathcal{I}$ is the identity function. Equation 4.5 reduces to:

$$C_i(s, o, a, \varphi) \;=\; I_j(s, o, a, \varphi) \tag{4.6}$$

Otherwise, we have an internal node, with

$$\mathbf{x}_i = (x_1, x_2, ..., x_k)$$

that is, objective o_i is divided into smaller scoped objectives (or cost functions) $X_1, \cdots, X_k$.

Let ϕ_i, $\mathcal{F}_i$, $\mathcal{S}_i$, and $\mathcal{T}_i$ be the data transformation functions for $\mathbf{x}_i$ ($\mathcal{I}$ for singletons.) Let β_i be the node combining function.

Then we recursively define the objective at X_i by

$$C_i(s, o, a, \varphi) \;=\; \beta_i \; (w_i \; (\phi_i \; (\mathcal{F}_i \; (\mathcal{S}_i \; (\mathcal{T}_i \; (< \mathbf{x}_i(s, o, a, \varphi) >)))))) \tag{4.7}$$

In our current model, we are, of course, not using the full generality described above. In particular:

- Our combinator β_i is usually either $+$ or $*$.
- The algebraic transformation functions we have considered to date have been quite simple. They include: linear sum, product, and sum-of-squares.
- While we are experimenting with $\mathcal{D}$, our current data transformation functions consist of "scales, scale, fuse, algebraically transform, and weigh linearly (that is, use multiplication if β is addition, and powers if β is multiplication)".

- The scaling function is typically affine. Moreover, the fusion and scaling functions (although definitely not the weight functions) are usually defined for each child considering only anticipated types, units, and ranges at that child, the desired unit at the parent, and the algebraic transformation and combining function to be used, but not the number of other children nor the anticipated characteristics of their values.

In our opinion, fusion functions are conceptually the most interesting and hardest to develop because they address non-commensurability. In the following sections we will see how they may be influenced by the combining function.

4.5.5 Guidelines for Choosing a Combining Function

Given a set of solutions $\{a_1, a_2, ...,\}$, objective X with child values $\mathbf{x} = \{x_1, x_2, ..., x_n\}$, and weights $w = \{w_1, w_2, ..., w_n\}$, how are these values to be combined? It all depends on how the user specifies the problem. In this section we shall describe situations and their corresponding combining forms.

One of the earliest mentionings in the literature is that of Lewis Carroll in [13]. He suggests various cases which will be presented in this section. In particular he suggests the following two general cases: 1) If results are to depend on *relative values* then use a **product** structure, with powers for weights if importances are not identical; 2) If units are of the same type and results are to depend on actual amounts then use an **additive** structure.

Before we start, the first problem that arises while combining values is one of *scale*–objectives may not be measured on the same scale type. Values need to be cohered to like scales before any combining can be performed [38]. In the following, the first five functions were suggested by L. Carroll in [13]. In addition, he states that "problems may evidently be set with many varying conditions, each requiring its own method of solution."

1. **Product** - When total cost x is to depend solely on relative cost of each value then use the product form:

$$C = \prod_{i=1}^{n} x_i \tag{4.8}$$

2. **Product of Powers** - If total cost x is to depend on relative cost of weighted values then product of powers or powers form:

$$C = \prod_{i=1}^{n} (x_i)^{w_i} \tag{4.9}$$

For example, if three objectives are weighted such that X_a is preferred twice as much, and X_b and X_c the same, then $w_a = \frac{1}{2}$, $r_b = \frac{1}{4}$ $r_c = \frac{1}{4}$, yielding: $C = x_a^{\frac{1}{2}} \times (x_b)^{\frac{1}{4}} \times (x_b)^{\frac{1}{4}}$.

3. **Sum** - When total cost x is to depend on actual amounts two additional problems arise. The first, measurements may be of different *unit type*, even if they are on the same scale. Second, *scaling*, measurements may have varying active domains and need to be scaled. Once these problems are resolved, moreover if the units are of equal scale, value type, and range, then the additive form applies:

$$C = \sum_{i=1}^{n} x_i \tag{4.10}$$

4. **Weighted Sum** - If the conditions are as before, but lower level objectives are weighted with weight w ($\sum_{1=1}^{k} w_i = 1$), then the form is that of a weighted sum:

$$C = \sum_{i=1}^{n} w_i x_i \tag{4.11}$$

If as before, three objectives are weighted such that X_a is preferred twice as much, and X_b and X_c the same, then then $w_a = \frac{1}{2}$, $r_b = \frac{1}{4}$ $r_c = \frac{1}{4}$, yielding: $C = \frac{1}{2}x_a + \frac{1}{4}x_b + \frac{1}{4}x_c$.

5. **Relative/Actual** - If the weights are to be such that, given an objective X_l , if all other objectives are equal, and the total cost is to depend on the relative cost of X_l. Moreover, $x_l = 0 \Longrightarrow C_{=}0$ and the remaining objectives are to effect the final cost collectively by their actual amounts. The resulting combining form is:

$$C = x_l(\sum_{i=1,i\neq l}^{n} x_i) \tag{4.12}$$

To continue our examples with X_a, X_b, and X_c , let $l = a$ then $C = x_a \times (x_b + x_c)$

6. **Actual/Relative** - If the weights are to be such that, given an objective X_l , if all other objectives are equal, then the total cost is to depend on their collective relative amount plus that of X_l. Moreover, $\exists i$ s.t.$x_{i(i\neq l)} = 0 \Longrightarrow C = x_l$. The resulting combining form is:

$$C = x_l + (\prod_{i=1,i\neq l}^{n} x_i) \tag{4.13}$$

Hence, in our examples with X_a, X_b, and X_c , let $l = a$ then $x = x_a + (x_b \times x_c)$.

7. **Dominance** - We would like one objective to dominate over another. Given two objectives X_a, X_b it is desired that the cost of X_a be much smaller relative to X_b's ($x_a << x_b$).

$$C = \frac{x_a}{x_b} \tag{4.14}$$

8. **Similarity** - We wish two objectives to be satisfied similarly.

$$C = |x_a - x_b|^{\alpha} \tag{4.15}$$

where α corresponds to an "error" or "penalty" model.

9. **Lowest** - If we are satisfied with having one out of n objectives low and we are willing to sacrifice the others then the form is:

$$C = MIN(x_1, x_2, ..., x_n)$$

For example, consider the problem of loading a computers memory bank under the following condition: there are n empty memory-chip slots on a board that accepts any chip type; all chips are of identical capacity, except that chips of different type are incompatible and will not run together. Then, we don't care how many chips we have or of what type as long as we have approximately n chips of the same type. Thereby, minimizing the number of empty slots.

4.6 Example

In this section an examples for the application of the function synthesis approach is provided. This example is worked out in detail, in particular the construction of its objective hierarchy and cost function synthesis is demonstrated.

Example 2 Consider an arbitrary software system, an arbitrary computer network, a set of feasible allocations mapping the software to hardware, execution schedules, and the attribute and atomic cost functions in Table 4.3. From the set of proposed allocations, we would like to choose the one that best matches our desired characteristics. They are: reduced latency (time by which a deadline is missed), good laxity (net amount of slack or lateness), reasonable deadline satisfaction, and short sequences of consecutive missed deadlines. Also, the combining of values should be linear when units are relatively of the same type, product otherwise, and a penalty incurred for larger values.

Clearly, in this example, all of the non-functional objectives are of the type "real-time" and they are to be combined as a sum or product of squares. Lower level

Table 4.3 Attributes and Attribute Cost Functions for the Resource Allocation Example. ct_i - completion time of task i, dl_i - deadline of task i, b - interval length, n - total number of tasks, k - maximum path length in the task graph. tx_j - transmission time of message j. rx_j - arrival time of message j. xt_j - expected duration of message j.

Attribute	Attribute Cost Function	Range	Unit type	Scale type
latency	$\sum_i \max(ct_i - dl_i, 0)$	$[0, n \cdot b]$	time	interval
laxity	$\sum_i (ct_i - dl_i)$	$[-\sum dl_i, n \cdot b]$	time	interval
% of missed deadlines	number of missed deadlines ÷ n	$[0, 1]$	bounded rational	ratio
missed deadlines	number of deadlines	$[0, n]$	integer	absolute
consecutive missed deadlines	Maximum number of consecutive missed deadlines on any execution path	$[0, k]$	integer	absolute
completion time	$max(ct_i)$	$[0, max(dl_i)]$	time	interval
communication delay	$\sum_i \max(0, (tx_j - rx_j) - xt_j)$	$[0, \sum_{j=2}^{xn}(\sum_{i=1}^{J-1})]$	time	interval
consecutive missed deadlines on a processor	Maximum number of consecutive missed deadlines on a processor	$[0, n]$	integer	absolute

objectives for this objective are: minimal latency, lowest laxity, greatest deadline satisfaction, minimal number of missed deadlines, and minimal consecutive missed deadlines. These attributes can be mapped to the following attribute cost functions respectively: latency, laxity, number of missed deadlines, and consecutive missed deadlines, which are given in Table 4.3.

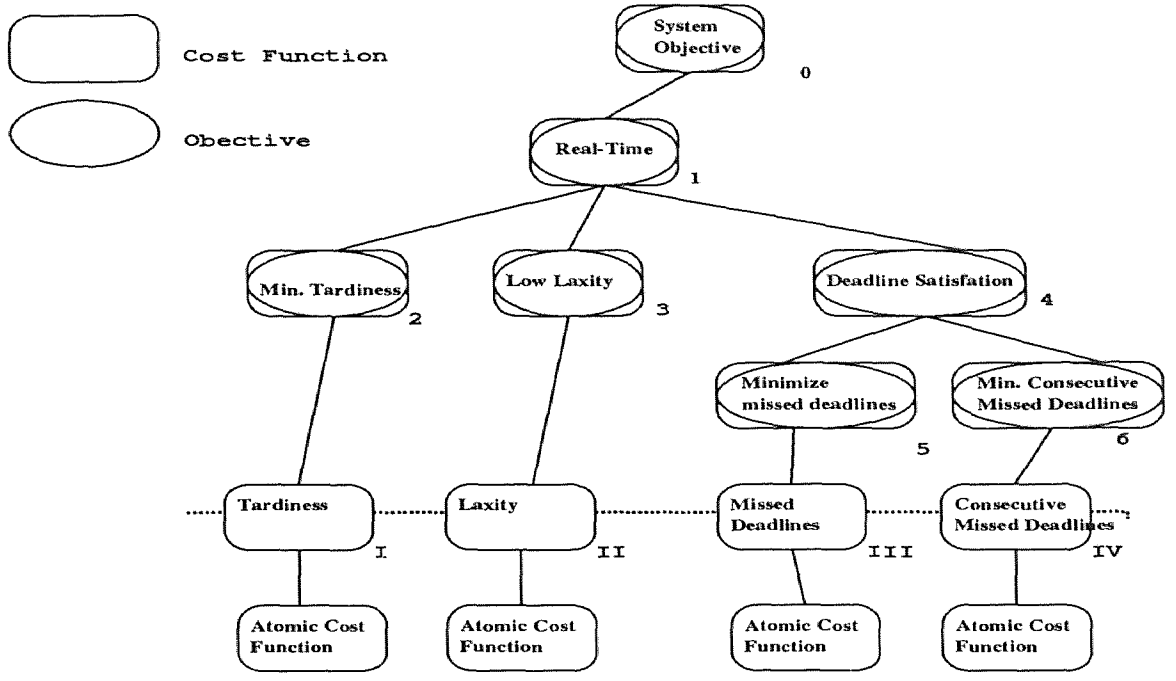


Figure 4.9 Objective Formulation for Example 2.

Therefore, in constructing the objective hierarchy and cost function synthesis for the example, we start with a set of predefined attribute cost functions. Next the design objectives are identified. Typically, objective decomposition is needed since the objectives as they are specified might not correspond to attribute cost functions. Finally, in a bottom-up fashion, the decision maker identifies the corresponding transformation functions for each node, Table 4.4, and constructs the system cost function. The objective decomposition and the attribute to attribute cost function mapping for this example is shown in Figure 4.9.

The final cost function is:

Table 4.4 Cost Function Synthesis Parameters for the Resource Allocation Example.

node	β	w	ϕ	$\mathcal{F}$	$\mathcal{S}$	$\mathcal{T}$
0	+	1	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}$
1	*	.33	$f(x) = x^2$	$c_1 \times u$	$u = c_2 \times C_2(s, o, a, \varphi)$	$\mathcal{I}$
		.33		$c_3 \times v$	$v = c_4 \times C_3(s, o, a, \varphi)$	$\mathcal{I}$
		.34		$c_5 \times w$	$w = c_6 \times w'$	$w' = c_7 \times C_4(s, o, a, \varphi)$
2	+	1	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}(C_I)$
3	+	1	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}(C_{II})$
4	+	.5	$f(x) = x^2$	$c_8 \times y$	$y = c_9 \times y'$	$y' = c_{10} \times C_5(s, o, a, \varphi)$
	+	.5		$c_{11} \times z$	$z = c_{12} \times z'$	$z' = c_{13} \times C_6(s, o, a, \varphi)$
5	+	1	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}(C_{III})$
6	+	1	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}(C_{IV})$

$$C_1(s, o, a, \varphi) =$$

$$((c_1 c_2 C_I)^2)^{.33} \times ((c_3 c_4 C_{II})^2)^{.33} \times ((c_5 c_6 c_7 (.5(c_8 c_9 c_{10} C_{III})^2 + .5(c_{11} c_{12} c_{13} C_{IV})^2))^2)^{.34}$$

Due to space constraints, here we omit the intermediate cost functions in the DAG of Figure 4.9.

It is quite obvious in this small example that the data are not of the same type nor range. Types vary from *time*, which can be expressed as an integer, to a *bounded rational* and an *integer*. Ranges within the same data type also vary (see Table 4.4). This example clearly illustrates that not all cost functions produce similar data; thereby, justifying the need for data transformation functions.

CHAPTER 5

INTER-COMPLETION TIME

In this chapter, we demonstrate the applicability of concepts originating from the multi-objective resource allocation problem to other areas. In particular, we extend the concept of deadline balancing of section 4.1.2 to a new scheduling concept, namely Inter-Completion-Time Scheduling (*ICTS*). We describe algorithms and discuss the implication of scheduling real-time tasks so as to maximize their minimal inter-completion time.

In *ICTS*, we try to avoid allocations where the difference in completion-times between consecutive tasks assigned to the same processor is small. The completion time of a task is the time at which a task fulfills all its functional requirements and stops executing. In other words we aim to avoid allocations with bunched up completion-times on any particular processor. As noted in Section 4.1.2 allocations having tasks with smaller deadline differences may suffer unwanted side effects, such as: saturation and idle states, communication bottlenecks, and/or poor handling of sporadic or aperiodic tasks. These anomalies may be far more evident in allocations where task inter-completion times are small.

The issue of shared resources, as mentioned in Chapter 1, is of prime importance in computer systems in general, and parallel and distributed applications in particular. Given one or several resources that need to be shared by a set of tasks, **scheduling** is the process of determining which task gets access to which resource, and when. Depending upon the characteristics of the application system under consideration, this process of schedule generation aims to optimize specified objectives. For hard real-time tasks, for example, the critical objective is that all tasks complete by their deadlines.

5.1 Model

Our attention in this research is restricted to the *non-preemptive* scheduling of independent real-time tasks. In our model, each task T_i is characterized by three parameters – a *release time* r_i, an *execution requirement* e_i and a *deadline* d_i, with the interpretation that task T_i becomes ready for execution at time r_i, and needs to be executed non-preemptively for e_i units of time over the interval $[r_i, d_i)$. Given a set $\tau = \{T_1, \ldots, T_n\}$ of n such tasks to be scheduled on m identical processors, the primary goal is to generate a schedule in which each task completes execution by its deadline.

Assuming that this primary goal can be met by several different schedules, secondary objectives may play a role in determining scheduling strategy. For example, in complex systems, where non-real-time tasks may coexist with real-time ones, it may be desirable to ensure that the real-time load is "balanced" among the various PE's. The focus of this research is one such secondary objective — that of *maximizing minimum inter-completion time*. In many applications, it is undesirable to have many different tasks which have been assigned to the same processor or processor group all complete within a small interval of time. Several potential problems arise if this is permitted to happen (recited from Section 4.1.2):

- Processor saturation during the intervals when a large number of tasks are completing execution.
- Network communication bottlenecks: Certain communication links may become overloaded during processor saturated intervals.
- Poor handling of sporadic tasks which arrive during saturated intervals.
- Poor load balancing at any particular point in time.

Scheduling to maximize the inter-completion time aims to "spread" the tasks' executions over time, thereby reducing the occurrence of the saturated and idle

states and the associated anomalies. When processors are clustered, as in systems with multiprocessor computers, similar problems occur but on a global scale within each cluster. Scheduling to maximize the global inter-completion time aims to spread each clusters' tasks' executions over time.

Inter-completion time. A schedule for τ on m processors is completely defined by specifying, for each $T_i \in \tau$, the processor p_i on which T_i is to execute, and the *start time* s_i at which it begins execution[1]. The time instant $c_i \stackrel{\text{def}}{=} s_i + e_i$ is called the *completion time* of task T_i in this schedule. For a given schedule, the *minimum inter-completion time on processor p* is defined to be the smallest difference between the completion-times of successive tasks that execute on processor p (if there is only one task that executes on a processor, then its minimum inter-completion time is defined to be ∞.). The *minimum inter-completion time (MICT) of a schedule* is defined to be the minimum, over all processors p, of the minimum inter-completion time of processor p. Similarly, for a given schedule, the *minimum global inter-completion time (MGICT) of a schedule* is defined to be the smallest difference between the completion-times of successive tasks, regardless of the processor they are executed on. If there is only one task that executes, then its minimum global inter-completion time is defined to be ∞. *MICT-scheduling* and *MGICT-scheduling* are the processes of generating a schedule with the largest possible minimum inter-completion time and minimum global inter-completion time, respectively.

Example 3 Consider a set of tasks $\tau = \{T_1 = (0, 4, 5), T_2 = (0, 3, 7), T_3 = (3, 2, 10), T_4 = (0, 1, 12), T_5 = (3, 1, 10), T_6 = (9, 1, 12)\}$. Figure 5.1 shows a schedule for this system on three processors. The minimum inter-completion time on Processor $P1$ is 5; since only once task completes on $P2$, its minimum inter-completion time is

[1] Of course, a valid schedule requires that each task executes within its release time and deadline (i.e., $s_i \geq r_i$ and $s_i + e_i \leq d_i$), and that no processor is assigned to more than one task at any given instant in time (i.e., if $p_i = p_j$, and $s_i \leq s_j$, then $s_i + e_i \leq s_j$).

set equal to ∞; the minimum inter-completion time on processor $P3$ is between the completions of tasks T_6 and T_3, and is equal to 4. The minimum inter-completion time for the schedule is therefore $\min(5, \infty, 4) = 4$, and the minimum global inter-completion time is $\min(2, 2, 3, 2, 2)$, which is 2.

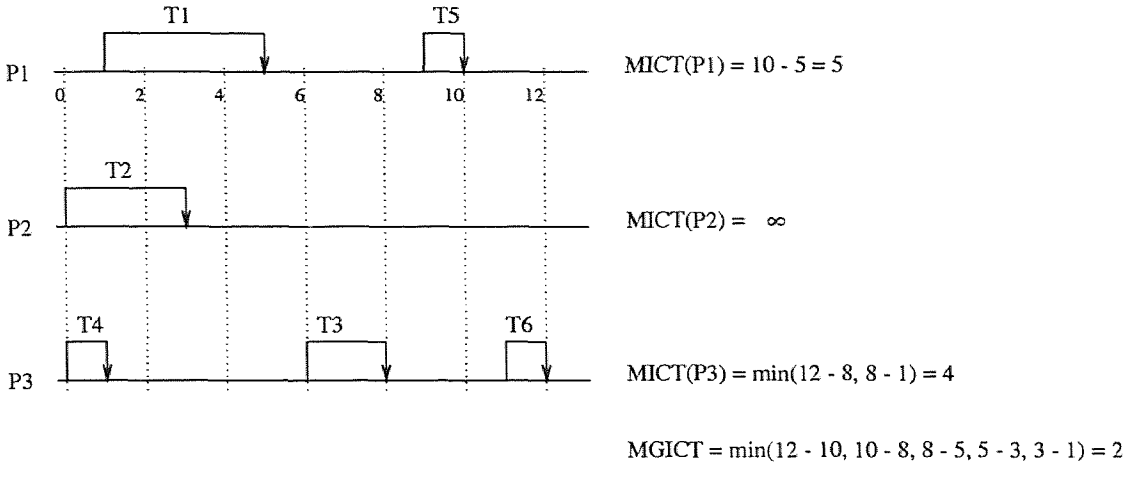


Figure 5.1 Schedule for Task System of Example 3

■

In traditional load-balancing the aim is to distribute the given set of tasks as evenly as possible among the available PE's, as in [75] and [78]. We view MICT/MGICT-scheduling as an extension to this view of load balancing, in that we are attempting to "balance" the load *temporally* as well as spatially (i.e., over the PE's). This perspective on load-balancing is particularly useful for handling of sporadic non-real-time tasks and in situations where some additional work needs to be done whenever a task completes execution, and we therefore wish to spread out these events as much as possible. It should be noted that these secondary jobs may involve tasks such as refreshing a display or updating system logs, and do *not* have hard deadlines associated with them. It is nevertheless desirable that they be completed within a reasonable interval of time.

5.2 Overview

In [5] we formalized the idea of spreading out over time the instants at which different tasks complete execution into the concept of *scheduling to maximize minimum inter-completion time* — MICT-scheduling. We focused on maximizing the minimum inter-completion time *on each processor*. We showed that the MICT-scheduling problem is, in general, NP-hard, and studied a wide variety of special cases of task systems. For a large number of these special cases, we presented very efficient scheduling algorithms; others we proved NP-hard. These results are summarized in Table 5.1. In addition, we have identified a very useful relationship between MICT-scheduling and the well-understood problem of scheduling to meet deadlines. In this thesis we review these special cases and extend the work to maximizing the minimum global inter-completion time.

Each task may be considered to enjoy three "degrees of freedom" — one for each of its parameters: r, e and d. In Section 5.3 (Theorems 4, 5, and 6), we show that we are unlikely to be able to obtain efficient MICT- & MGICT-scheduling algorithms that can schedule arbitrary sets of tasks, even on a single processor. We therefore investigate the issue of designing optimal MICT- & MGICT-scheduling algorithms when one or more of the degrees of freedom are curtailed. In Section 5.5, we consider sets of tasks in which all tasks are identical – i.e., each task has zero degrees of freedom. In Section 5.6, we focus on task sets in which each task has one degree of freedom. That is, we separately consider the cases where all tasks (i) have the same release time and execution requirement, but may have different deadlines, (ii) have the same execution requirement and deadline, but may have different release times, and (iii) have the same release time and deadline, but may have different execution requirements. In Section 5.7, we consider task sets where each task has two degrees of freedom — once again, we have three different possibilities, which are individually analyzed. For each of the cases listed above, we consider both uniprocessor and multi-

Table 5.1 Summary of Results in This Chapter (r - release time; e - execution requirement; d - deadline; $n \stackrel{\text{def}}{=}$ number of tasks; $\hat{d} \stackrel{\text{def}}{=}$ the largest deadline; the smallest release time is assumed to be 0)

	FIXED			MICT		MGICT
				Processors		
	r	e	d	Uniproc	Multiproc	
1	N	N	N	NP-hard (Theorem 4)	NP-hard (Theorem 5)	NP-hard (Theorem 6)
2	N	N	Y	$\mathcal{O}(n \log n \log \hat{d})$ (Sec. 5.7.1.1)	NP-hard (Theorem 11)	NP-hard (Theorem 12)
3	N	Y	N	$\mathcal{O}(n^2 \log n \log \hat{d})$ (Sec. 5.7.1.3)	open (Sec. 5.7.2)	open (Sec. 5.7.2)
4	Y	N	N	$\mathcal{O}(n^2 \log \hat{d})$ (Sec. 5.7.1.2)	NP-hard (Theorem 13)	NP-hard (Theorem 14)
5	N	Y	Y	$\mathcal{O}(n \log n)$ (Sec. 5.6.1)	$\mathcal{O}(n \log n)$ (Sec. 5.6.1)	$\mathcal{O}(n \log n)$ (Sec. 5.6.1.2)
6	Y	N	Y	$\mathcal{O}(n \log n)$ (Sec. 5.6.2)	NP-hard (Theorem 8)	NP-hard (Theorem 9)
7	Y	Y	N	$\mathcal{O}(n \log n)$ (Sec. 5.6.1)	$\mathcal{O}(n \log n)$ (Sec. 5.6.1)	$\mathcal{O}(n \log n)$ (Sec. 5.6.1.2)
8	Y	Y	Y	$\mathcal{O}(n)$ (Sec. 5.5)	$\mathcal{O}(n)$ (Sec. 5.5)	$\mathcal{O}(n)$ (Sec. 5.5.2)

processor MICT-scheduling and multiprocessor MGICT-scheduling. Our results are summarized in Table 5.1. Finally, in Section 6.11 we present simulation results.

5.3 Inter-Completion Time Scheduling is NP-hard

In this section, we prove that the general problem of obtaining a schedule with large minimum inter-completion time for an arbitrary task system is intractable. We start with some definitions.

A *task system* is specified by an ordered pair

$$\langle \tau = \cup_{i=1}^{n} \{T_i = (r_i, e_i, d_i)\}, m \rangle,$$

and represents a set of n tasks $T_1, T_2, \ldots, T_n$, to be scheduled on m identical processors, where task T_i is released at time r_i, has a deadline of d_i, and an execution requirement of e_i.

$\mathsf{mict}(\langle \tau, m \rangle) = \Delta$ indicates that there is a schedule for task system $\langle \tau, m \rangle$ with a minimum inter-completion time at least Δ. Similarly, $\mathsf{mgict}(\langle \tau, m \rangle) = \Delta$ indicates that there is a schedule for task system $\langle \tau, m \rangle$ with a minimum global inter-completion time at least Δ. Thus, asserting $\mathsf{mict}(\langle \tau, m \rangle) = 0$ and $\mathsf{mgict}(\langle \tau, m \rangle) = 0$ is equivalent to stating that τ is feasible on m processors.

Lemma 1 Given an arbitrary set of tasks τ and an arbitrary integer $\Delta > 0$, the problem of determining whether $\mathsf{mict}(\langle \tau, 1 \rangle) = \Delta$ is NP-complete in the strong sense.

Proof: Transformation from *Sequencing with release times and deadlines* [25, page 236]. ■

As a direct consequence of Lemma 1, we obtain the following theorem:

Theorem 4 Given an arbitrary set of tasks τ, it is NP-hard to schedule τ on one processor such that the minimum inter-completion time is maximized.

■

Theorems 5 and 6 immediately follow.

Theorem 5 Given an arbitrary set of tasks τ and m processors it is NP-hard to schedule τ on the m processors such that the minimum inter-completion time is maximized.

■

Theorem 6 Given an arbitrary set of tasks τ and m processors it is NP-hard to schedule τ on the m processors such that the minimum global inter-completion time is maximized.

■

5.4 Reducing MICT-Scheduling to Feasibility

While the issue of non-preemptive scheduling to maximize minimum inter-completion time has not been widely studied, there does exist a vast amount of literature devoted to feasibility analysis for non-preemptive scheduling. These include intractability results [25, Section A5], approximation algorithms [28, 27, 34], optimal algorithms for special cases [26, 68], etc. In this section, we attempt to exploit this wide body of research by establishing a relationship between MICT-scheduling and general non-preemptive scheduling.

The following theorem reduces the problem of determining schedules with specified minimum inter-completion times to the problem of determining feasibility of sets of tasks.

Theorem 7 Let $\tau \stackrel{\text{def}}{=} \cup_{i=1}^{n} \{T_i = (r_i, e_i, d_i)\}$ be a set of tasks. Let $\Delta \geq 0$. For each i, $1 \leq i \leq n$, define $\delta_i \stackrel{\text{def}}{=} \max(0, \Delta - e_i)$. Define $r(\tau, \Delta) = \cup_{i=1}^{n} \{T_i' = (r_i - \delta_i, e_i + \delta_i, d_i)\}$.

$$\mathsf{mict}(\langle \tau, m \rangle) = \Delta \quad \textbf{iff} \quad r(\tau, \Delta) \text{ is feasible on } m \text{ processors.}$$

Example 4 Consider a set of tasks $\tau = \{T_1 = (0, 3, 6), T_2 = (0, 1, 7), T_3 = (4, 6, 12)\}$. We wish to determine whether τ can be scheduled on one processor such that the minimum inter-completion time is at least five ($\Delta = 5$). Since $\delta_1 = 2, \delta_2 = 4, \delta_3 = 0$, Theorem 7 claims that this is equivalent to determining whether $r(\tau, 5) = \{T_1' = (-2, 5, 6), T_2' = (-4, 5, 7), T_3' = (4, 6, 12)\}$ can be scheduled on one processor:

Proof of Theorem 7: In this proof, let τ' denote the task set $r(\tau, \Delta)$.

<u>LHS ⇒ RHS.</u> $\mathsf{mict}(\langle \tau, m \rangle) = \Delta \Rightarrow \langle \tau', m \rangle$ is feasible:

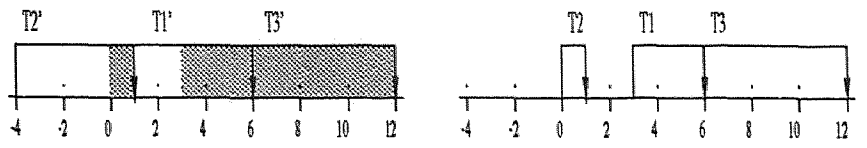


Figure 5.2 Schedule for $\tau(\tau, 5)$, and Schedule for τ with MICT 5

Suppose first that $\mathsf{mict}(\langle \tau, m \rangle) = \Delta$, and let S_o be an m-processor schedule for τ with a minimum inter-completion time $\geq \Delta$. We describe how to obtain a schedule S_1 for τ' on m processors.

For each i, $1 \leq i \leq n$, let $[t_s, t_s + e_i)$ denote the interval during which T_i was executed in schedule S_o. T_i' is executed on the same processor in S_1; its execution interval is determined as follows:

- If $\delta_i = 0$, then the execution requirements of T_i' and T_i are the same, and S_1 schedules T_i' over the interval $[t_s, t_s + e_i)$.
- If $\delta_i > 0$, observe that (i) the processor on which T_i is executed in S_o is idle over the interval $[t_s - \delta_i, t_s)$ (this follows from the fact that the minimum inter-completion time of S_o is at least Δ), and (ii) since $t_s \geq r_i$, it must be the case that $t_s - \delta_i \geq r_i - \delta_i$. Schedule S_1 therefore executes T_i' over the interval $[t_s - \delta_i, t_s + e_i)$.

<u>LHS $\Leftarrow$ RHS.</u> $\langle \tau', m \rangle$ is feasible $\Rightarrow \mathsf{mict}(\langle \tau, m \rangle) = \Delta$:

Suppose now that τ' is feasible on m processors, and let S_2 be an m-processor schedule for τ'. We describe below how to obtain a schedule S_3 for τ on m processors, which has a minimum inter-completion time of (at least) Δ.

For each i, $1 \leq i \leq n$, let $[t_s, t_s + e_i + \delta_i)$ denote the interval during which T_i' is scheduled in S_2. Then T_i is executed on the same processor in S_3; its execution interval is determined as follows:

- If $\delta_i = 0$, then the execution requirements of T_i' and T_i are the same, and S_3 schedules T_i over the interval $[t_s, t_s + e_i)$ as well. Since $e_i \geq \Delta$, the separation between the completion time of T_i and the task (if any) that was executed prior to it on the same processor is at least Δ.
- If $\delta_i > 0$, observe that $t_s \geq r_i - \delta_i$. S_3 assigns the processor to T_i over the interval $[t_s + \delta_i, t_s + e_i + \delta_i)$. Observe that (i) this interval is of size e_i, (ii) since $t_s \geq r_i - \delta_i$, this interval starts no earlier than r_i, and (iii) since $e_i + \delta_i = \Delta$, the separation between the completion time of T_i and the task (if any) that was executed prior to it on the same processor is exactly Δ.

■

Remark 1 The proof of Theorem 7 is *constructive* — given a schedule for $r(\tau, \Delta)$ on m processors, we can use the reduction defined in Case 2 of the proof to construct a schedule for τ on m processors with a minimum inter-completion time $\geq \Delta$. Furthermore, such a reduction can be performed in $\mathcal{O}(n)$ time.

5.5 Task Systems with no Degrees of Freedom

We start out by considering the very simple problem of scheduling a set of n identical tasks $\tau = \{T_1, \ldots, T_n\}$, where $T_i = (0, E, D)$, on m processors. First, we address the problem in MICT-scheduling then MGICT-scheduling.

5.5.1 MICT-Scheduling

Consider any schedule for $\langle \tau, m \rangle$. Since there are n tasks to be scheduled on m processors, some processor will be assigned at least $\lceil n/m \rceil$ tasks. The first task on this processor completes at (or after) time E; the interval $[E, D)$ is to be partitioned into at least $(\lceil n/m \rceil - 1)$ inter-completion times. Therefore, the minimum inter-completion time, obtained by partitioning $[E, D)$ as evenly as possible subject to

```
Δ ← ⌊(D−E)/(⌈n/m⌉−1)⌋;
If Δ < E return "not feasible";
t ← 0; /* time */
p ← 1; /* processor */
for i ← 1 to n do {
    if t + E > D {
        p ← p + 1;
        t ← 0;
    }
    Schedule T_i on processor p over the interval [t, t + E);
    t ← t + Δ
}
```

Figure 5.3 Algorithm for MICT-scheduling a Set of Identical Tasks

integer boundaries, is

$$\Delta_{max} \stackrel{\text{def}}{=} \lfloor \frac{D-E}{\lceil n/m \rceil - 1} \rfloor$$

An algorithm for generating a schedule with a minimum inter-completion time of $\Delta_{\max}$ is given in Figure 5.3; since its correctness is quite obvious, a formal proof of correctness is omitted. Observe that its run-time complexity is $\mathcal{O}(n)$, where n is the number of tasks.

5.5.2 MGICT-Scheduling

Consider any schedule for $\langle \tau, m \rangle$. Since there are n tasks to be scheduled on m processors, some processor will be assigned at least $\lceil n/m \rceil$ tasks. For this system to be feasible the sum of the execution times of all tasks on this processor must be $\leq D$.

The first task completes at (or after) time E; the interval $[E, D)$ is to be partitioned into at least $(n-1)$ inter-completion times. Therefore, if $\Delta \geq \frac{E}{m}$ the

minimum inter-completion time, obtained by partitioning $[E, D)$ as evenly as possible subject to integer boundaries, is

$$\Delta_{max} \stackrel{\text{def}}{=} \lfloor \frac{D - E}{n - 1} \rfloor$$

Otherwise, the minimum inter-completion time is

$$\Delta_{max} \stackrel{\text{def}}{=} \lfloor \frac{D - \lceil \frac{n}{m} \rceil \cdot E}{m - 1} \rfloor$$

This is because when Δ is less than $\frac{E}{m}$, given any two tasks, k^{th} and $k + m^{th}$, with start times s_k and $s_{k+m} = s_k + \Delta \cdot m$, and finish times $f_k = s_k + E$ and f_{k+m}. This implies that $s_{k+m} < f_k$ which cannot be since no processor can be assigned to more than one task at any given instant.

An algorithm for generating a schedule with a minimum global inter-completion time of Δ_{max} is given in Figure 5.4. Observe that its run-time complexity is $\mathcal{O}(n)$, where n is the number of tasks.

Lemma 2 Let Δ be the largest number such that $\text{mgict}(\langle \tau, m \rangle) = \Delta$. Algorithm MGICT-SCHEDULING generates a schedule for $\langle \tau, m \rangle$ with a minimum global inter-completion time equal to Δ.

Proof:

Consider any schedule for $\langle \tau, m \rangle$ that schedules all tasks. Recall that no processor can be assigned to more than one task at any given instant (i.e., $s_i + E \leq s_{i+m}$). Hence, there are three cases for the value of the inter-completion with respect to $\frac{E}{m}$. They are: $\lfloor \frac{D-E}{n-1} \rfloor \geq \frac{E}{m} \Rightarrow s_i + m \cdot \Delta \geq s_i + E$; $\frac{D-E}{n-1} = \frac{E}{m} \Rightarrow s_i + m \cdot \Delta = s_i + E$; and $\frac{D-E}{n-1} < \frac{E}{m} \Rightarrow s_i + m \cdot \Delta < s_i + E$.

First, if $\Delta \geq \frac{E}{m}$ or $\frac{D-E}{n-1} = \frac{E}{m}$ observe that algorithm MGICT-SCHEDULING generates a schedule with a minimum global inter-completion time of $\lfloor \frac{D-E}{n-1} \rfloor$. Since the first completion time is $\geq E$, and the k'th is $\leq (k-1) \cdot \Delta + E$, it follows that

If $D < \lceil \frac{n}{m} \rceil \cdot E$ return "not feasible";
$j \leftarrow 1$; /* processor */
$\Delta \leftarrow \lfloor \frac{D-E}{n-1} \rfloor$
if $\Delta \leq \frac{E}{m}$ then do {
    if $(\frac{D-E}{n-1} \neq \frac{E}{m})$ then $\Delta \leftarrow \lfloor \frac{D-\lceil \frac{n}{m} \rceil \cdot E}{m-1} \rfloor$
    $x \leftarrow 0$; /* number of tasks already on the current processor */
    for $i \leftarrow 1$ to n do {
        assign T_i to the j'th processor
        schedule task T_i on processor j over the interval
            $[E \cdot x + ((i-1) mod\ m) \cdot \Delta, (E \cdot x + ((i-1) mod\ m) \cdot \Delta + E)$
        if $(j < m)$ then $j \leftarrow j + 1$ else $j \leftarrow 1$, $x \leftarrow x + 1$
    }
} else do {
    for $i \leftarrow 1$ to n do {
        assign T_i to the j'th processor
        schedule task T_i on processor j over the interval
            $[(i-1) \cdot \Delta, (i-1) \cdot \Delta + E)$
        if $(j < m)$ then $j \leftarrow j + 1$ else $j \leftarrow 1$
    }
}

Figure 5.4 Algorithm for MGICT-SCHEDULING a Set of Identical Tasks

the minimum inter-completion time on this schedule is no smaller than $\lfloor((k-1)\cdot \Delta + E - E)/(k-1)\rfloor = \Delta$.

When $\Delta < \frac{E}{m}$, there will be at least one processor with $\lceil \frac{n}{m} \rceil$ tasks assigned to it. Therefore, the minimum inter-completion time, obtained by partitioning $D - \lceil \frac{n}{m} \rceil \cdot E$ as evenly as possible subject to integer boundaries, is

$$\Delta_{max} = \lfloor \frac{D - \lceil \frac{n}{m} \rceil \cdot E}{m-1} \rfloor$$

■

5.6 Task Systems with One Degree of Freedom

The case when the task system is allowed one degree of freedom is more interesting, and not quite as trivial as in the previous section. The results of this section are summarized in rows 5–7 of Table 5.1: observe that 7 of the 9 cases here are efficiently solvable while the eighth and ninth, perhaps surprisingly, are intractable. We also point out here that the $\mathcal{O}(n \log n)$ complexity of each of the tractable problems is due to the complexity of sorting n numbers; if the tasks are available in sorted order according to their non-fixed parameter, each of these problems can be solved in $\mathcal{O}(n)$ time.

5.6.1 Equal Release Times and Execution Requirements

We first consider task systems where all tasks (i) are released at the same instant, and (ii) have the same execution requirement. Without loss of generality, we assume that the common release time is 0, and let E denote the execution time of each task. The deadlines of different tasks may be different. (Since we are concerned with the off-line versions of the problem, in which all task parameters are known beforehand, the results here, by symmetry, apply also to the case when individual release times may differ, but all execution times are equal and all tasks have the same deadline.)

5.6.1.1 MICT-Scheduling with Equal Release Time and Execution Requirement:

One processor. First, we consider the case when set of tasks $\tau = \{T_1, T_2, \ldots, T_n\}$, with $T_i = (0, E, d_i)$, are to be scheduled on a single processor. Assume that the tasks are sorted by deadline (i.e., $d_i \leq d_{i+1}$ for all i) — given a set of n tasks, this can be achieved in $O(n \log n)$ time. Algorithm SCHEDULEPROC (Figure 5.5) generates a schedule for τ with the maximum possible minimum inter-completion time:

```
delmin ← ∞
for ℓ ← 2 to n do
    if (⌊(d_ℓ − E)/(ℓ − 1)⌋ ≤ delmin) then delmin ← ⌊(d_ℓ − E)/(ℓ − 1)⌋
    if (delmin < E) then return "not feasible"
for i ← 1 to n do
    schedule task T_i over the interval [(i − 1) · delmin, (i − 1) · delmin + E)
```

Figure 5.5 Algorithm SCHEDULEPROC

Lemma 3 Let Δ be the largest number such that $\mathsf{mict}(\langle \tau, 1 \rangle) \geq \Delta$. Algorithm SCHEDULEPROC generates a schedule for τ on one processor with a minimum inter-completion time equal to Δ.

Proof: Let $d_1, d_2, \ldots, d_n$ denote the deadlines of the tasks, arranged in order of non-decreasing deadline. Observe first that Algorithm SCHEDULEPROC generates a schedule with a minimum inter-completion time equal to $\min_{\ell=2}^{n}\{\lfloor (d_\ell - E)/(\ell - 1) \rfloor\}$

Consider now any schedule that schedules all the tasks. Since the first completion time is $\geq E$, and the k'th is $\leq d_k$, it follows that the minimum inter-completion time on this schedule is no smaller than $\lfloor (d_k - E)/(k-1) \rfloor$ for each integer k, $2 \leq k \leq n$. ■

Observe again that the run-time complexity of Algorithm SCHEDULEPROCS is $\mathcal{O}(n)$ if the tasks are already sorted by deadline. If the task deadlines are not already sorted, they can be sorted in $\mathcal{O}(n \log n)$ time.

Multiple processors. We now consider the case when $\tau = \bigcup_{i=1}^{n} \{T_i = (0, E, d_i)\}$ are to be scheduled on m processors, $m > 1$. Given such a system, Algorithm MULTIPROC (Figure 5.6) generates a schedule for τ with the maximum possible minimum inter-completion time.

Lemma 4 Let Δ be the largest number such that $\mathsf{mict}(\langle \tau, m \rangle) \geq \Delta$. Algorithm MULTIPROC generates a schedule for $\langle \tau, m \rangle$ with a minimum inter-completion time equal to Δ.

Before proving this Lemma, we need some auxiliary results. Let the deadlines of the n tasks, arranged in non-decreasing order, be $d_1, d_2, \ldots, d_n$.

Claim 4.1 Each k, $m < k \leq n$, imposes the restriction that

$$\Delta \leq \lfloor \frac{d_k - E}{\lceil k/m \rceil - 1} \rfloor \tag{5.1}$$

Proof: Observe that there are k tasks with deadline $\leq d_k$. By the pigeonhole principle, there is one processor which is assigned at least $\lceil k/m \rceil$ of these tasks. Since the first completion time on this processor is $\geq E$, and the completion time for each task with deadline no more than d_k is $\leq d_k$, it follows that the minimum inter-completion time is no more than $\lfloor (d_k - E)/(\lceil k/m \rceil - 1) \rfloor$. ■

We are now ready to prove Lemma 4.

Proof of Lemma 4: Suppose that Algorithm MULTIPROC generates a schedule with minimum inter-completion time Δ_{min}. Suppose that this minimum inter-completion time occurs on processor j, and is due to the assignment of the i'th-largest

1. Sort the tasks by deadline.
2. Assign the tasks, considered in deadline order, to the processors in a round-robin fashion. That is, let $T_1, T_2, \ldots, T_n$ denote the tasks sorted by deadline. Assign the tasks to the m processors, as follows:

```
j ← 1
for i ← 1 to n do
    assign T_i to the j'th processor
    if (j < m) then j ← j + 1 else j ← 1
```

3. Schedule each processor individually, by Algorithm SCHEDULEPROC.

Figure 5.6 Algorithm MULTIPROC

deadlined task[2]. That is,

$$\Delta_{\min} = \lfloor \frac{d_i - E}{n_{i,j} - 1} \rfloor$$

where $n_{i,j}$ denotes the number of tasks with deadline $\leq d_i$ that have been assigned to processor j in Step 2 of Algorithm MULTIPROC. Since the tasks are assigned to the processors in round-robin order, it is clear that exactly $\lceil i/m \rceil$ of the first i tasks are assigned to processor j; i.e., $n_{i,j} = \lceil i/m \rceil$. Therefore,

$$\Delta_{\min} = \lfloor \frac{d_i - E}{\lceil i/m \rceil - 1} \rfloor \tag{5.2}$$

By setting k in Equation 5.1 to i, it follows that no schedule can obtain a larger minimum inter-completion time. ■

Run-time complexity. Step 1 takes $\mathcal{O}(n \log n)$ time. Step 2 takes $\mathcal{O}(n)$ time. Let n_j denote the number of tasks allocated to processor j, $1 \leq j \leq m$, in Step 2. Step 3 requires calls to Algorithm SCHEDULEPROC on sets of tasks that are already sorted by deadline. The total complexity of this step is therefore

[2]It can be shown that j is necessarily 1, since the i^{th} deadline on processor 1 imposes a tighter constraint on completion of the first i tasks, for each i. However, this fact does not concern us here.

$\sum_{j=1}^{m} \mathcal{O}(n_j)$, which is equal to $\mathcal{O}(n)$. The dominant step is therefore Step 1, and the total complexity is $\mathcal{O}(n \log n)$.

5.6.1.2 MGICT-Scheduling with Equal Release Time and Execution Requirement:

We now consider the case when $\tau = \bigcup_{i=1}^{n} \{T_i = (0, E, d_i)\}$ are to be MGICT-scheduled on m processors, $m > 1$. Given such a system, Algorithm MGICTEQUALEXDL (Figure 5.7) generates a schedule for τ with the maximum possible minimum global inter-completion time.

Lemma 5 Let Δ be the largest number such that $\text{mgict}(\langle \tau, m \rangle) \geq \Delta$. Algorithm MGICTEQUALEXDL generates a schedule for $\langle \tau, m \rangle$ with a minimum global inter-completion time equal to Δ.

Proof: Similar to the proof of Lemma 2. ■

Run-time complexity. Step 1 takes $\mathcal{O}(n \log n)$ time. Step 2 takes $\mathcal{O}(n)$ time. The dominant step is therefore Step 1, and the total complexity is $\mathcal{O}(n \log n)$.

5.6.2 Equal Release Times and Deadlines

When all the release times and execution requirements are equal (or, by symmetry, when all the deadlines and execution requirements are equal), we have seen that the problem of scheduling to maximize minimum inter-completion can be very efficiently solved on any number of processors. We will now see that, when execution requirements may vary while release times and deadlines are fixed, the situation is not quite the same.

Theorem 8 Let $\tau = \bigcup_{i=1}^{n} \{T_i = (0, e_i, D)\}$ be a task system. The problem of MICT-scheduling τ on m processors is NP-hard, for arbitrary m.

1. Sort the tasks by deadline.

2. for $l \leftarrow 2$ to n do {
   if $(\lfloor \frac{d_l - E}{l-1} \rfloor \leq$ `delmin`$)$ then `delmin` $\leftarrow \lfloor \frac{d_l - E}{l-1} \rfloor$, `lmin` $\leftarrow l$
   }
   if $(d_l < \lceil \frac{l}{m} \rceil \cdot E)$ return "not feasible"
   $j \leftarrow 1$; /* processor */
   if `delmin` $< \frac{E}{m}$ then do {
      if $(\frac{D\texttt{lmin}-E}{\texttt{lmin}-1} \neq \frac{E}{m})$ then $\Delta \leftarrow \lfloor \frac{d_{\texttt{lmin}} - \lceil \frac{\texttt{lmin}}{m} \rceil \cdot E}{m-1} \rfloor$
      $x \leftarrow 0$; /* number of tasks already on the current processor */
      for $i \leftarrow 1$ to n do {
         assign T_i to the j'th processor
         schedule task T_i on processor j over the interval
            $[E \cdot x + ((i-1) mod\ m) \cdot \Delta, (E \cdot x + ((i-1) mod\ m) \cdot \Delta + E)$
         if $(j < m)$ then $j \leftarrow j + 1$ else $j \leftarrow 1$, $x \leftarrow x + 1$
      }
   } else do {
      $\Delta \leftarrow$ `delmin`
      for $i \leftarrow 1$ to n do {
         assign T_i to the j'th processor
         schedule task T_i on processor j over the interval
            $[(i-1) \cdot$ `delmin` $), (i-1) \cdot$ `delmin` $) + E)$
         if $(j < m)$ then $j \leftarrow j + 1$ else $j \leftarrow 1$
      }
   }

Figure 5.7 Algorithm MGICTEQUALEXDL for MGICT-scheduling Tasks with Identical Deadlines and Execution Requirement

Proof: We prove this theorem by showing that there is a polynomial transformation from the NP-hard problem of multiprocessor scheduling to MICT-scheduling with equal release times and deadlines.

The **multiprocessor scheduling problem** is defined as follows [25, page 238]:

> INSTANCE: Set T of tasks, number $m \in Z^+$ of processors, length $\ell(t) \in Z^+$ for each $t \in T$, and a deadline $D \in Z^+$.
> QUESTION: Is there an m-processor schedule for T that meets the overall deadline D, i.e., a function $\sigma : T \rightarrow Z_0^+$ such that, for all $u \geq 0$, the number of tasks $t \in T$ for which $\sigma(t) \leq u < \sigma(t) + \ell(t)$ is no more than m and such that, for all $t \in T$, $\sigma(t) + \ell(t) \leq D$?

Given an arbitrary instance of the multiprocessor scheduling problem, we obtain an instance of the problem of MICT-scheduling with equal release times and deadlines, by the following mechanism: for each task t in the multiprocessor scheduling problem instance, we define an MICT task with release-time 0, deadline D, and execution requirement $\ell(t)$. It is relatively straightforward to observe that this system of MICT tasks can be scheduled with an inter-completion time ≥ 0 if and only if the multiprocessor scheduling problem instance has a solution. ■

Theorem 9 immediately follows.

Theorem 9 Let $\tau = \cup_{i=1}^{n} \{T_i = (0, e_i, D)\}$ be a task system. The problem of MGICT-scheduling τ on m processors is NP-hard, for arbitrary m.

However, the situation is not quite as bleak on a single processor. Assume that the tasks are sorted by execution requirement (i.e., $e_i \leq e_{i+1}$ for all i) — given a set of n tasks, this can be achieved in $\mathcal{O}(n \log n)$ time. Observe that an MICT-schedule would have a task with smallest execution requirement (without loss of generality, T_1) scheduled over the interval $[0, e_1)$, and that this would leave the interval $[e_1, D)$ to be partitioned into $n - 1$ inter-completion intervals. We want $[e_1, D)$ to be partitioned as evenly as possible, subject to the constraint that each $e_i, i > 1$, has to "fit" within

Algorithm EQUALEXUNIPROC

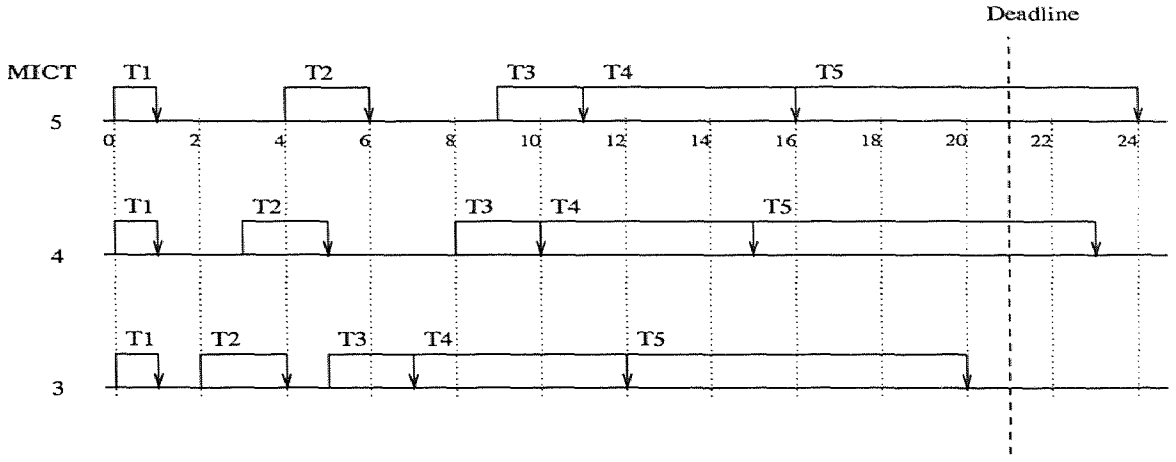
```
delmin ← ∞
for ℓ ← n down to 2 do
    if (⌊(D − e_1 − Σ_{j=ℓ+1}^{n} e_j)/(ℓ − 1)⌋ ≤ delmin)
        then delmin ← ⌊(D − e_1 − Σ_{j=ℓ+1}^{n} e_j)/(ℓ − 1)⌋
schedule task T_1 over [0, e_1)
c ← e_1 /* last completion time */
for i ← 2 to n do
    if (e_i ≤ delmin) {
        schedule task T_i over the interval [c + delmin − e_i, c + delmin)
        c ← c + delmin }
    else {
        schedule task T_i over the interval [c, c + e_i)
        c ← c + e_i }
```

Figure 5.8 Algorithm EQUALEXUNIPROC

an interval. Algorithm EQUALEXUNIPROC (Figure 5.8) generates such a schedule, in which the order of task-execution is $T_1, T_2, \ldots, T_n$. (The first for-loop accounts for the possibility that some of the later tasks have very large execution requirements, thus forcing the remaining tasks closer together.)

Example 5 Consider a set τ of 5 tasks, with $r_i = 0$ for all tasks, $d_i = 21$ for all tasks (i.e., $D = 21$),, and $e_1 = 1$, $e_2 = 2$, $e_3 = 2$, $e_4 = 5$, and $e_5 = 8$. We trace below the execution of Algorithm EQUALEXUNIPROC on τ: for each iteration of the first for loop, we indicate how the value of `delmin` gets updated. (The figure shows that `delmin` $= 5$ or 4 are unacceptable, illustrating the need to loop through $\ell = 5, 4, 3, 2$ to determine the optimal `delmin`.)

ℓ	$\lfloor (D - e_1 - \sum_{j=\ell+1}^{n} e_j)/(\ell - 1) \rfloor$	delmin
		∞
5	$\lfloor (21 - 1 - 0)/(5 - 1) \rfloor = 5$	5
4	$\lfloor (21 - 1 - 8)/(4 - 1) \rfloor = 4$	4
3	$\lfloor (21 - 1 - 13)/(3 - 1) \rfloor = 3$	3
2	$\lfloor (21 - 1 - 15)/(2 - 1) \rfloor = 5$	3



■

Theorem 10 Let Δ be the largest number such that $\mathsf{mict}(\langle \tau, 1 \rangle) \geq \Delta$. Algorithm EQUALEXUNIPROC generates a schedule for $\langle \tau, 1 \rangle$ with a minimum inter-completion time equal to Δ.

Proof: Similar to the proof of Lemma 3. ■

If the tasks are already sorted by execution requirement, the run-time complexity of Algorithm EQUALEXUNIPROC is $\mathcal{O}(n)$. Since sorting can be done in $\mathcal{O}(n \log n)$ time, the total complexity of MICT-scheduling a set of tasks with equal release times and deadlines on one processor is $\mathcal{O}(n \log n)$.

5.7 Task Systems with Two Degrees of Freedom

In Sections 5.5 and 5.6, we considered task systems with zero and one degrees of freedom. All of these were relatively straightforward to analyze and, with the

exception of Theorems 8 and 9, could be solved from first principles. Task systems with *two* degrees of freedom — the subject of this section — are a lot more challenging. For the uniprocessor problems, we make use of the reduction defined in Section 5.4 to transform MICT-scheduling to (general) non-preemptive scheduling, and then design efficient solutions to the resulting scheduling problems. Each of the three cases (Sections 5.7.1.1 - 5.7.1.3) require a fresh approach that differs significantly from the ones employed in the other two. Four of the six multiprocessor problems turn out to be intractable; the complexity of the others remains unresolved.

5.7.1 One Processor

Let $\tau = \bigcup_{i=1}^{n}\{T_i = (r_i, e_i, d_i)\}$ be a set of tasks to be scheduled on a single processor. Let $\hat{d} \stackrel{\text{def}}{=} \max_{i=1}^{n}\{d_i\}$, and assume without loss of generality that $\min_{i=1}^{n}\{r_i\} = 0$. Let $\Delta_{\max}$ denote the largest integer Δ for which $\mathsf{mict}(\langle\tau, 1\rangle) \geq \Delta$. Observe that $\lfloor\hat{d}/(n-1)\rfloor$ is a (loose) upper bound on the value of $\Delta_{\max}$. The aim in MICT-scheduling is to generate a schedule with a minimum inter-completion time equal to $\Delta_{\max}$.

Suppose now that we had an algorithm that, given τ and a positive integer Δ, determines whether τ can be scheduled with a minimum inter-completion time of at least Δ on a single processor (i.e., whether $\mathsf{mict}(\langle\tau, 1\rangle) \geq \Delta$); if so, it generates a schedule with minimum inter-completion time at least Δ. Then an MICT-schedule for τ — i.e., a schedule with minimum inter-completion time equal to $\Delta_{\max}$ — can be obtained by making $\mathcal{O}(\log\lfloor\hat{d}/(n-1)\rfloor)$ calls to this algorithm, by essentially performing "binary search" between the values 0 and $\lfloor\hat{d}/(n-1)\rfloor$. Since $\mathcal{O}(\log\lfloor\hat{d}/(n-1)\rfloor) = \mathcal{O}(\log\hat{d})$, the complexity of MICT-scheduling τ is therefore $\mathcal{O}(\log\hat{d})$ times the

complexity of generating a schedule with a specified minimum inter-completion time Δ, if it exists[3].

In the remainder of Section 5.7.1, we consider separately the three cases when τ is restricted in one of its degrees of freedom – fixed deadlines (Section 5.7.1.1), fixed release times (Section 5.7.1.2), and fixed execution requirements (Section 5.7.1.3). For each, we make use of the reduction defined in Theorem 7 to design an efficient algorithm which accepts as input a constrained task system τ and a positive integer Δ and, if $\mathsf{mict}(\langle \tau, 1 \rangle) \geq \Delta$, generates a schedule for τ with minimum inter-completion time of at least Δ.

5.7.1.1 Fixed Deadlines: When all the deadlines are equal, we may make use of Theorem 7 to reduce MICT-scheduling on a single processor to a tractable problem in (regular) scheduling.

Let $\tau = \cup_{i=1}^{n} \{(T_i = (r_i, e_i, D)\}$ be a task system in which all tasks have the same deadline D, and let Δ be a given positive integer. We apply the reduction r defined in Theorem 7 to τ, yielding the taskset $r(\tau, \Delta) \stackrel{\text{def}}{=} \cup_{i=1}^{n} \{(T_i' = (r_i - \delta_i, e_i + \delta_i, D)\}$, where $\delta_i \stackrel{\text{def}}{=} \max(0, \Delta - e_i)$. By Theorem 7, τ has a schedule with minimum inter-completion time Δ if and only if $r(\tau, \Delta)$ is feasible.

Since each task in $r(\tau, \Delta)$ has the same deadline D, it is trivial to determine whether $r(\tau, \Delta)$ is feasible, and to generate a schedule if the answer is yes: simply schedule the tasks according to earliest release times (ties broken arbitrarily), and report success if they all complete by time D, and failure otherwise. The run-time complexity is $\mathcal{O}(n \log n)$, with the dominant cost being the cost of sorting the tasks by order of non-decreasing release times. The overall complexity of determining $\Delta_{\max}$ is therefore $\mathcal{O}(n \log n \log \hat{d})$.

[3]Observe that $\log \hat{d}$ is polynomial in the size of the binary representation of τ; MICT-scheduling is therefore a polynomial-time operation, provided the problem of generating a schedule with specified minimum inter-completion time is in PTIME.

5.7.1.2 Fixed Release Times: When all the release times are equal, a technique similar to the one used in Section 5.7.1.1 may be used.

Let $\tau = \cup_{i=1}^{n}\{(T_i = (0, e_i, d_i)\}$ be a task system in which all tasks have the same release time (without loss of generality, we have assumed that this release time is 0). Let Δ be a given positive integer. We once again apply the reduction r defined in Theorem 7 to τ, yielding the taskset $r(\tau, \Delta) \stackrel{\text{def}}{=} \cup_{i=1}^{n}\{(T_i' = (-\delta_i, e_i + \delta_i, d_i)\}$, where $\delta_i \stackrel{\text{def}}{=} \max(0, \Delta - e_i)$. By Theorem 7, τ has a schedule with minimum inter-completion time Δ if and only if $r(\tau, \Delta)$ is feasible.

Observe that each task in $r(\tau, \Delta)$ may have a different release time, execution requirement, and deadline. Scheduling such systems is, in general, NP-hard in the strong sense (*Sequencing with release times and deadlines* [25, page 236]). Fortunately, $r(\tau, \Delta)$ is not quite general – notice that the interval between the release time $-\delta_i$, and the instant zero, is no larger than the execution requirement $e_i + \delta_i$, for every task T_i. We may therefore conclude that at most one task executes before time-instant zero in any schedule for $r(\tau, \Delta)$. In the pseudocode below, each iteration of the for loop "guesses" a different candidate T_ℓ for this first task. The rest of the tasks are all available by the time T_ℓ completes execution, and may therefore be executed in deadline order. Since some task T_j must execute first in a schedule for $r(\tau, \Delta)$, this algorithm will discover the schedule during the j'th iteration.

```
1 Assume that the tasks are available in order of non-decreasing deadlines
2 for ℓ ← 1 to n do{
    /*Task T_ℓ is executed first */
3   execute task T_ℓ over the interval [−δ_ℓ, e_ℓ)
4   execute the remaining tasks in EDF order
5   if all tasks meet their deadlines return "success"
  }
```

The run-time complexity may be computed as follows: It costs $\mathcal{O}(n \log n)$ to sort the tasks by deadline (line 1). Each iteration of the for loop (lines 2–5) takes $\mathcal{O}(n)$ time, and there could be up to n iterations, for a total complexity of $\mathcal{O}(n \log n + n^2)$, which equals $\mathcal{O}(n^2)$. The overall complexity of determining $\Delta_{\max}$ is therefore $\mathcal{O}(n^2 \log \hat{d})$.

5.7.1.3 Equal Execution Requirements: Let $\tau = \cup_{i=1}^{n} \{(T_i = (r_i, E, d_i)\}$ be a task system in which all tasks have the same execution requirement E, which we wish to schedule on a single processor. Let Δ be a given positive integer. We make use of the following result from [68] in determining whether $\mathsf{mict}(\langle \tau, 1 \rangle) \geq \Delta$:

Result 1 (Simons (1978)) Let τ be a set of n tasks, in which all tasks have the same execution requirement. Simons presented an $\mathcal{O}(n^2 \log n)$ algorithm to determine if τ can be non-preemptively scheduled on a single processor, and to generate such a schedule if it exists. We will refer to this algorithm as *Simons' Algorithm.*

We apply the reduction r defined in Theorem 7 to τ, yielding the taskset $r(\tau, \Delta) \stackrel{\text{def}}{=} \cup_{i=1}^{n} \{(T_i' = (r_i - \delta, E + \delta, d_i)\}$, where $\delta \stackrel{\text{def}}{=} \max(0, \Delta - E)$. By Theorem 7, τ has a schedule with minimum inter-completion time Δ if and only if $r(\tau, \Delta)$ is feasible.

The crucial observation is that the execution requirements of all tasks in $r(\tau, \Delta)$ are equal. We can therefore use Simons' Algorithm to determine in $\mathcal{O}(n^2 \log n)$ time if $r(\tau, \Delta)$ is feasible, and to generate a schedule if so. The total complexity of determining $\Delta_{\max}$ is therefore $\mathcal{O}(n^2 \log n \log \hat{d})$.

5.7.2 Multiple Processors

While all the problems studied above are seen to have efficient solutions, four of the corresponding problems on multiple processors turn out to be intractable. The complexity of the fifth and sixth remain open.

Theorem 11 Let $\tau = \cup_{i=1}^{n}\{T_i = (r_i, e_i, D)\}$. The problem of MICT-scheduling τ on m processors in NP-hard, for arbitrary m.

Proof: Directly follows from Theorem 8. ■

Theorem 12 Let $\tau = \cup_{i=1}^{n}\{T_i = (r_i, e_i, D)\}$. The problem of MGICT-scheduling τ on m processors in NP-hard, for arbitrary m.

Proof: Directly follows from Theorem 9. ■

Theorem 13 Let $\tau = \cup_{i=1}^{n}\{T_i = (0, e_i, d_i)\}$. The problem of MICT-scheduling τ on m processors in NP-hard, for arbitrary m.

Proof: Directly follows from Theorem 8. ■

Theorem 14 Let $\tau = \cup_{i=1}^{n}\{T_i = (0, e_i, d_i)\}$. The problem of MGICT-scheduling τ on m processors in NP-hard, for arbitrary m.

Proof: Directly follows from Theorem 9. ■

CHAPTER 6

EXPERIMENTAL VALIDATION

In previous chapters, an approach for cost function synthesis was presented. In addition a new scheduling concept, namely Inter-Completion-Time Scheduling (*ICTS*) was introduced. This chapter features the evaluation of these techniques.

First, the cost function synthesis approach is applied to a simple model of a stock exchange system. The objective decomposition is derived as well as three different cost function syntheses for this system. These cost functions are applied to various hardware and software scenarios. The behavior and the performance of the functions are evaluated by exhaustively calculating the function values and by using the cost functions as a fitness function for a Genetic Algorithm. Finally, the effects of the inter-completion time Scheduling strategy on Inter-Processor Communication is studied and results presented.

In the following sections, the application and experiments are explained in detail and comments on the conclusions drawn from them.

6.1 Application Description

Our application example models the transaction processing system for a stock brokerage firm. As the stock market changes continuously, trading stock is obviously a real-time process; orders have to be fulfilled within a certain amount of time.

The brokerage firm operates a computer system consisting of a number of computers interconnected by a LAN. In this system each broker has its own computer that acts as his or her front-end. Once a transaction has been started through the front-end, it can be executed on any of the firm's computers.

A transaction starts when a customer of the firm places an order to sell or buy shares. This order is entered into the computer system of the brokerage firm by a

broker, where the order is pre-processed. After that the computer system sends the order to the appropriate stock exchange and waits for the confirmation. Once this confirmation arrives, the order is post-processed and the customer is notified.

The above described process encompasses a multitude of tasks, which constitute a mix of real-time and non real-time activities. Real-time activities are for instance the interaction with the stock exchange system and the transfer of funds from and to the customer's account. Examples for non real-time activities are mailing confirmations and logging.

6.2 Model

In our model, a transaction is defined as a self-contained chain of communication and computation actions that is initiated by a stimulus and ends with a response. This chain is represented by an independent directed acyclic graph (DAG). Internally, a transaction consists of a set of tasks.

Each task T_i is characterized by three parameters – a *release time* r_i, an *execution requirement* $e_{i,j}$ $(j = 1, ..., m)$, a *deadline* d_i, and *communication vector* $\mathbf{c}_i = (c_1, c_2, \cdots, c_n)$ — with the interpretation that task T_i becomes ready for execution at time r_i, and needs to be executed for e_i units of time over the interval $[r_i, d_i)$. In addition, tasks are executed on processors. Firms may have multiple interconnected processors.

6.3 Objectives

The system should exhibit high degree of performance and real-time property characteristics. It should finish all transactions as quickly as possible and avoid communication delays. It should try to complete all transactions by their deadlines. If deadlines are missed, the sooner the transaction finishes the better. In addition, it

is better for one transaction to have many delays than for all to be delayed even a little.

6.4 Objective Decomposition

To ensure the logical flow of the hierarchy, the objectives described above are decomposed according to Gibson's guidelines as outlined in Section 4.3.

From the objective description in section 6.3 and the given attributes defined in Table 4.3, we start with the highest level objective, "Exhibit high degree of performance and real-time property characteristics." Obviously, this objective decomposes into performance and real-time. Real-time is to be described by deadline satisfaction and tardiness. Continuing this way, we obtain the following decomposition:

Performance

finish all transactions as quickly as possible

-minimize completion time

avoid communication delays

-minimize communication delays.

Real-time

try to complete the transactions by their deadlines.

it is better for one transaction to be

delayed a long time than for many to be delayed.

-minimize consecutive missed deadlines within a transaction

-minimize consecutive missed deadlines on a processor

if deadlines are missed the sooner they finish the better.

avoid missing deadlines

-minimize missed deadlines

quickly finish late transactions

-minimize latency

Now we are ready to match leaf nodes with appropriate attribute cost functions. Figure 6.1 gives the final qualitative decomposition and corresponding quantifiable attributes.

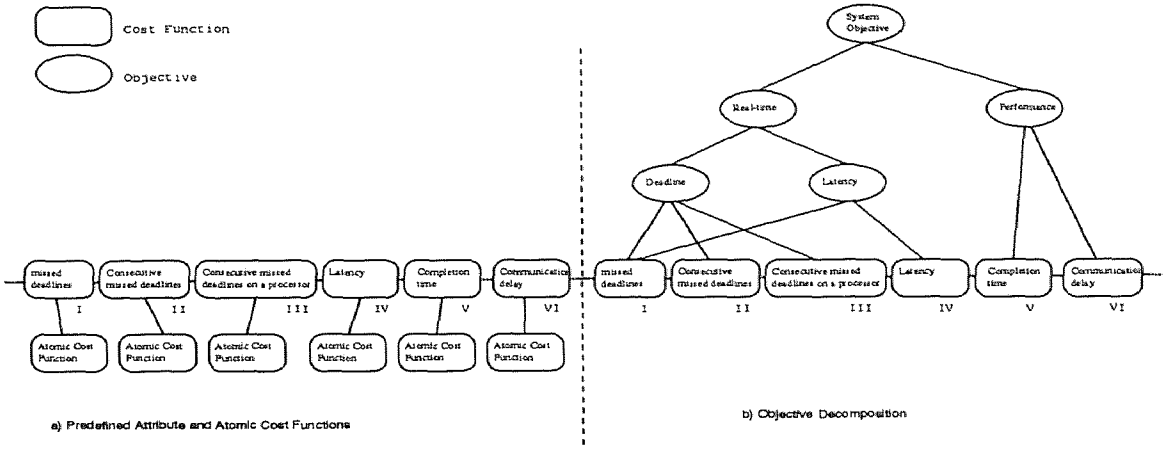


Figure 6.1 Application Decomposition

6.5 Cost Function Synthesis

Now that the decomposition with corresponding attribute has been determined, the system cost function is constructed in a bottom-up fashion. The resulting synthesis is shown in Figure 6.2.

From *missed deadlines* (I) and *total latency* (IV) we can obtain *average latency* (4). The lower the average latency the less amount of time tasks are delayed on average, for those tasks that do get delayed. This is sufficient for objective "if deadliness are missed the sooner they finish the better." In combining, first *missed deadlines* must be coerced down to an interval scale. Furthermore, since we are using a product structure no fusion is required, but we must make certain that we do not

divide by zero. Therefore, the value of 1 is added to the actual value of *missed deadlines*. Note that *missed deadlines* is 0 if and only if the *latency* is 0. Hence adding 1 to *missed deadlines* does not change the result in this case, i.e., the lowest possible value is still 0. The combination yields an interval type scale.

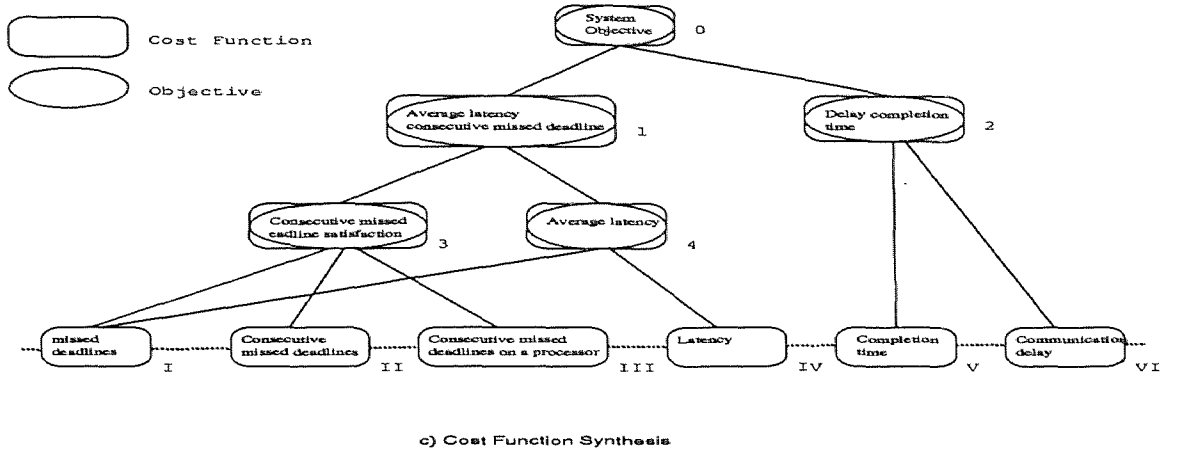


Figure 6.2 Brokerage Cost Function Synthesis

The *deadline*, or rather, the *consecutive missed deadlines* (3) component of the decomposition can be obtained from attribute cost functions *consecutive missed deadline*(II) and *consecutive missed deadline on a processor* (III). Both of these attributes are of the same scale type and very similar unit type. Therefore, we may use the additive structure without scale, scaling, or fusion transformations. The weights are chosen such that they favor *consecutive missed deadline on a processor*, since there is less of a preference on *consecutive missed deadlines*. Thus, *consecutive missed deadline satisfaction*, is sufficient for quantifying objective "it is better for one transaction to be delayed a long time than for many to be delayed even a little".

These two cost functions, *consecutive missed deadline satisfaction* (3) and *average latency* (4), must now be combined to quantify *real-time* (1) node in the decomposition. First, *consecutive missed deadline satisfaction* must be coerced to an interval scale. Again, we are using a product structure. Therefore, no fusion is required, but we must make certain that we do not multiply with zero. Hence, one

is added to both child values. This combination yields *average latency consecutive missed deadline.*

Turning to the performance side of the decomposition, node (2) can be obtained from attribute cost functions *completion time* and *communication delay.* Both are of interval scale type. Since we are using the product form once more, few transformations are required except that we must avoid zero values again. The weights are made such that they favor completion time over communication delays, since there is less emphasis on communication delay in the objectives description.

Finally, the *system* cost function, node (0), can be synthesized. It is quantified from the lower level cost functions *average latency consecutive missed deadline* and *delay completion time.* Both functions produce non-zero values of like scales, but different unit type. Since we are using the product structure, most transformations will not be required. In addition, since there is no preference between the two cost functions, both will be weighted evenly.

The resulting synthesis is shown in Figure 6.2 and node details in Table 6.1 *synthesis* 1. For the same decomposition, two additional syntheses (2 & 3) were constructed by modifying the combining function at one of the internal nodes. This was done to test the combining decision made. In synthesis 2, the combining function at node 2 was changed to an additive structure. While, in synthesis 3, node 3's combining function was modified to a product structure. All node details are described in Table 6.1.

6.6 Experiment Description

A basic requirement on any cost function for resource allocation is that it be capable of yielding good results in a multitude of environments. Hence we set up a series of experiments with varying numbers of processors and tasks to be allocated on those processors. Specifically, we used the following hardware/software systems:

Table 6.1 Brokerage Cost Function Parameters.

Node	Synthesis	β	w	ϕ	$\mathcal{F}$	$\mathcal{S}$	$\mathcal{T}$
0	1 & 3	$*$	.5	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}(C_1)$
			.5	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}(C_2)$
	2		.5	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}(C_1)$
			.5	$\mathcal{I}$	$\mathcal{I}$	$f(x) = x + 1$	$\mathcal{I}(C_2)$
1	1 & 2	$*$	.5	$\mathcal{I}$	$\mathcal{I}$	$f(x) = x + 1$	$\mathcal{I}(C_3)$
			.5	$\mathcal{I}$	$\mathcal{I}$	$f(x) = x + 1$	$\mathcal{I}(C_4)$
	3		.5	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}(C_3)$
			.5	$\mathcal{I}$	$\mathcal{I}$	$f(x) = x + 1$	$\mathcal{I}(C_4)$
2	1 & 3	$*$	.3	$\mathcal{I}$	$\mathcal{I}$	$f(x) = x + 1$	$\mathcal{I}(C_V)$
			.7	$\mathcal{I}$	$\mathcal{I}$	$f(x) = x + 1$	$\mathcal{I}(C_{VI})$
	2	$+$	.3	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}(C_V)$
			.7	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}(C_{VI})$
3	1 & 2	$+$	.0	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}(C_I)$
			.3	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}(C_{II})$
			.7	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}(C_{III})$
	3	$*$	.0	$\mathcal{I}$	$\mathcal{I}$	$f(x) = x + 1$	$\mathcal{I}(C_I)$
			.3	$\mathcal{I}$	$\mathcal{I}$	$f(x) = x + 1$	$\mathcal{I}(C_{II})$
			.7	$\mathcal{I}$	$\mathcal{I}$	$f(x) = x + 1$	$\mathcal{I}(C_{III})$
4		$\div$	.5	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}(C_{IV})$
			.5	$\mathcal{I}$	$\mathcal{I}$	$f(x) = x + 1$	$\mathcal{I}(C_I)$

Scenario	No. of Proc.	No. of Tasks
1	2	6
2	2	10
3	2	18
4	3	12

A special workload generator was used to create the scenarios 2 and 3. Scenarios 1 and 4, on the other hand, were designed manually such that they have a unique optimal solution in the unrestricted case. Unrestricted here means that no constraint checking was performed prior to the cost function evaluation. In other words, there is one optimal solution in the entire search space S. This solution need not necessarily be part of the constraint satisfying space F as shown in Figure 1.1. The optimal solution could of course be ruled out by the constraint checker. As mentioned, however, the experiments were focused on the unrestricted case, i.e., without constraint checking.

The functions were investigated with respect to their overall behavior as well as their performance in an optimization algorithm. Exhaustive evaluation of the cost functions for the entire input space exhibited their general behavior. We looked into aspects like monotony and value range. In a second step the functions were used as fitness functions in a resource allocation algorithm based on Genetic Algorithms (GA). In this case the convergence of the GA is of particular interest.

6.6.1 Exhaustive Evaluation

When evaluating the system cost function, questions like, "how many unique function values exist?", "is the function monotonic?", or "which size has the value range covered by the function values?" arise. These questions can only be answered, if the behavior of the cost function for all possible input spaces is known. Hence, we applied exhaustive evaluation, i.e., we created all input data sets and handed them to the cost functions for evaluation.

This provided us with the system cost function value for each input data set. A comparison of the three cost functions derived from the different syntheses was

undertaken with respect to the function values for all input data and the sorted function values. Furthermore, the number of unique function values compared to the size of the input data space was calculated.

We also used the exhaustive evaluation to obtain the best possible function value. This knowledge facilitates the assessment of the performance of the GA-based, stochastic resource allocator (see below).

6.6.2 GA-Based Resource Allocator

Besides the "static", exhaustive evaluation, the functions were applied in an optimization environment. More specifically they were integrated into a stochastic resource allocator. The stochastic search of the allocator is based on Genetic Algorithms (GA), a method used to solve NP-hard optimization problems in many fields. GAs constitute a so-called "uninformed" search strategy. This term refers to the fact that the algorithm itself does not have any knowledge on the problem it solves. The problem specific knowledge is entirely incorporated into the *fitness function*, which were implemented by the cost functions derived from our multi-objective syntheses. The fitness function calculates the *fitness value* of a candidate solution produced by the GA. The resulting fitness value is used as a feedback to the algorithm.

A problem to be solved by a GA has to be *encoded* in an appropriate way, because GAs act upon bit strings. Each bit string encodes one part of the information on the problem solution and is called a *gene*. Genes are grouped to *chromosomes*; one or more chromosomes form an *individual*. Each individual encodes a complete solution to the given problem. The quality of this solution, i.e., the fitness of the individual is assessed by the fitness function. A certain number of individuals form a *population*.

The GA imitates nature's process of evolution by taking one population as parent generation and creating an offspring generation. The algorithm selects the best, i.e., fittest, individuals of the parent population. Here the fitness of an individual comes into play. The better the fitness value of an individual, the greater is the probability that it is selected for reproduction. After the selection the GA mutates some of the genes of the selected individuals by flipping bits according to a given mutation rate and performs cross-over between the individuals by swapping parts of chromosomes. As a result of this process a new population, a child generation, is created, which is then evaluated. The whole process iterates until a solution of sufficient quality is found. For a comprehensive description of GAs and their function the reader is referred to [70]. The GA that has been used as the basis of the experiments is presented in [53].

6.7 Evaluation Criteria

In this section the graphs and the criteria that were applied to evaluate the performance of the three cost function types under the given scenarios are described.

Raw Data Graph. This graph compares the raw function values of the three cost function types for exhaustive evaluation. It covers the entire input space. This graph facilitates the evaluation of issues related to one cost function,

- the absolute value range,
- the shape regarding local minima and maxima, and
- the value distribution,

as well as issued related to the comparison of the cost function types,

- the input permutation yielding the optimal value, and
- the location of minima and maxima.

Sorted Data Graph. This graph also takes into account the entire space of input data. As opposed to the raw data graph this time the function values are sorted. It therefore allows to determine

- value range covered by the functions, and
- the number of unique function values. Several permutations resulting in the same function value are represented by plateaus in this graph.

Weight Comparison Graph. Only one function type at a time is depicted by this graph. It shows the raw function values for this function type as resulting from exhaustive evaluation. However, in this graph three instances of this function type are illustrated, which differ in the weight distribution at the system node of the synthesis. In other words, the influence of the real-time and the performance branch of the decomposition is varied for the instances. Using this graph one can determine the influence of either of the two branches on the system cost.

GA-Convergence Graph. While the above-mentioned graph types dealt with the results from exhaustive evaluation, the GA-convergence graph depicts the outcome of the stochastic allocator. The graph shows the fitness, i.e., the cost function value, of the best individual for each generation. By means of this graph the number of generations needed to converge as well as the best cost value given by the GA-based allocator can be derived.

Quality of Attribute Cost Function Selection. For any resource allocation problem the input data space comprises all possible allocations of the given tasks to the PEs. If n is the number of tasks and m is the number of PEs, then m^n is the size of the input space denoted by S. During the decomposition of the objectives one has to select the set of attribute cost functions to be used.

For each input data set there is a certain vector of attribute cost function values. Obviously, not all of these vectors are distinct. A stands for the number of unique attribute cost function vectors.

Given A and S, $\frac{A}{S}$ is a measure for the quality of the attribute cost function selection. A small value of the fraction means that a large number of allocations will have the same attribute cost function vector and thus the system cost function value regardless of the synthesis. Since in our model there is a one to one relation between attribute and atomic cost functions, results will also apply to atomic cost functions.

Quality of Synthesis. Depending on the particular synthesis, several input data sets may result in the same system cost function value. The number of unique system cost function values is denoted by U. Clearly, the maximum number of unique solutions is bounded by the number of different attribute cost function vectors A, i.e. $U \leq A$. The quality of the synthesis can be concluded from the fraction $\frac{U}{A}$, which gives the ratio of unique attribute cost function vectors resulting in unique system cost function values. A larger value means a better discrimination between different allocations.

In the following section the various criteria are evaluated. For each criterion we compare several or all of the 4 scenarios.

6.8 Observations

In this section we summarize the most interesting observations in the experiments.

6.8.1 Raw Data Graphs

As the input data spaces for scenarios 3 and 4 encompass 262,144 and 531,441 input data sets, respectively, the corresponding raw data graphs are too large to be

shown here. Hence we restrict the evaluation of this criterion to scenarios 1 and 2 (Figures 6.3 and 6.4).

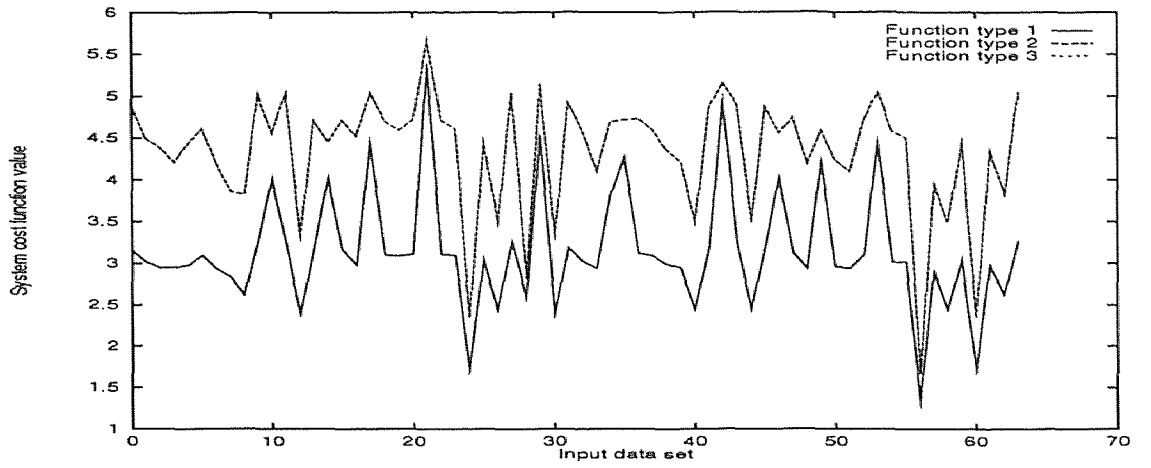


Figure 6.3 Scenario 1—Unsorted Cost Function Values.

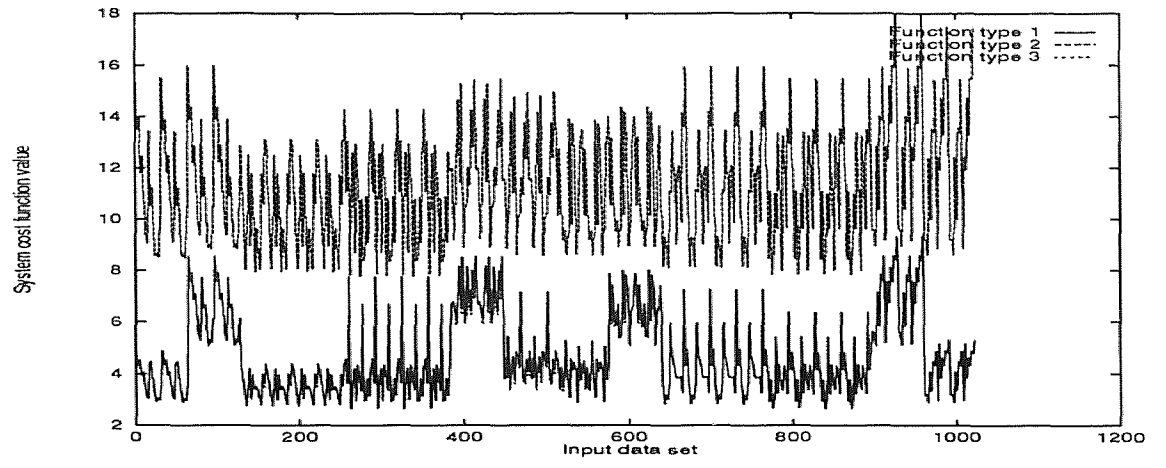


Figure 6.4 Scenario 2—Unsorted Cost Function Values.

The following observations can be concluded from the raw data graphs:

- At first glance it is obvious that function types 1 and 3 yield similar results. Recalling the discussion of the cost function syntheses, we see that function types 1 and 3 differ in the combining function of `consecutive missed deadlines` and `consecutive missed deadlines on a single PE`. As will be argued below, the results of the product and the sum combining function

are relatively similar, if the operands have similar values. This is exactly the case in the deadline branch of the objective DAG.

- Note that in both scenarios function type 2 has a value distribution similar to that of types 1 and 3. There are, however, some differences. For instance, for scenario 1 (Figure 6.3) function type 2 shows a local minimum for permutation 10, while function types 1 and 3 exhibit a local maximum at the same permutation.

- The monotony for all three function types is good; neither function shows large plateaus of neighboring equal values that could trap a search algorithm.

- For scenario 1 also the sole optimal solution is reflected properly by all three cost function types through a unique minimum function value.

6.8.2 Sorted Data Graphs

Next, we look at the sorted function values.

As scenario 1, depicted in Figure 6.5, has the smallest input data space it enables a more detailed investigation:

- In all cases the largest plateau covers 4 input data sets.

- There is a sufficient incline and function value range in all three function types with function type 2 being a little superior in terms of function value difference. Overall, there are 45 distinct function values out of a potential 64 different values for all types, which is about 70%.

Generally, the function types cover a sufficient value range for all scenarios. Also, for all scenarios there are only small plateaus of equal function values. The issue of the number of unique function values, i.e., the quality of the attribute cost function selection and the synthesis, will be dealt with later.

Also note the similarity in the function shape in the two manually constructed scenarios 1 and 4. This is due to the similar task structure in these two scenarios.

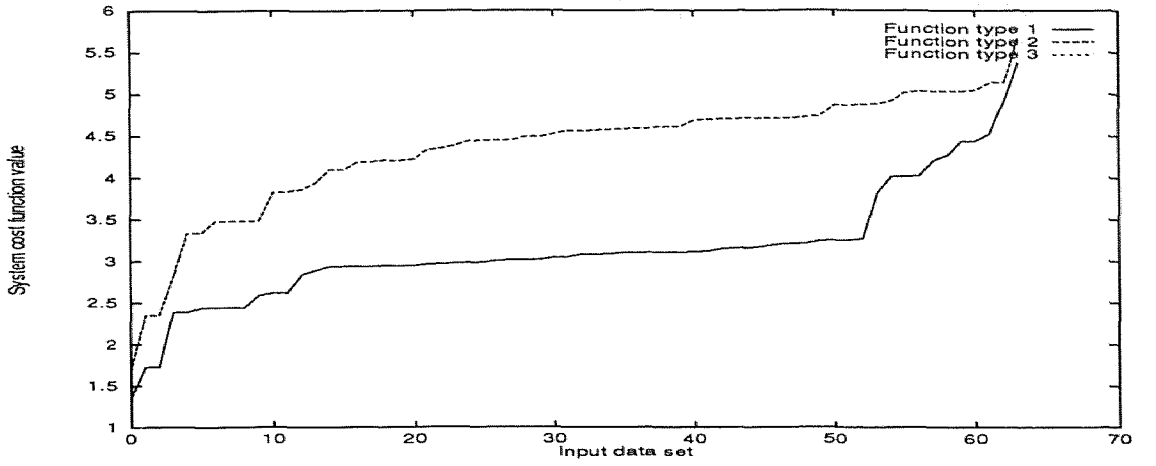


Figure 6.5 Scenario 1—Sorted Cost Function Values.

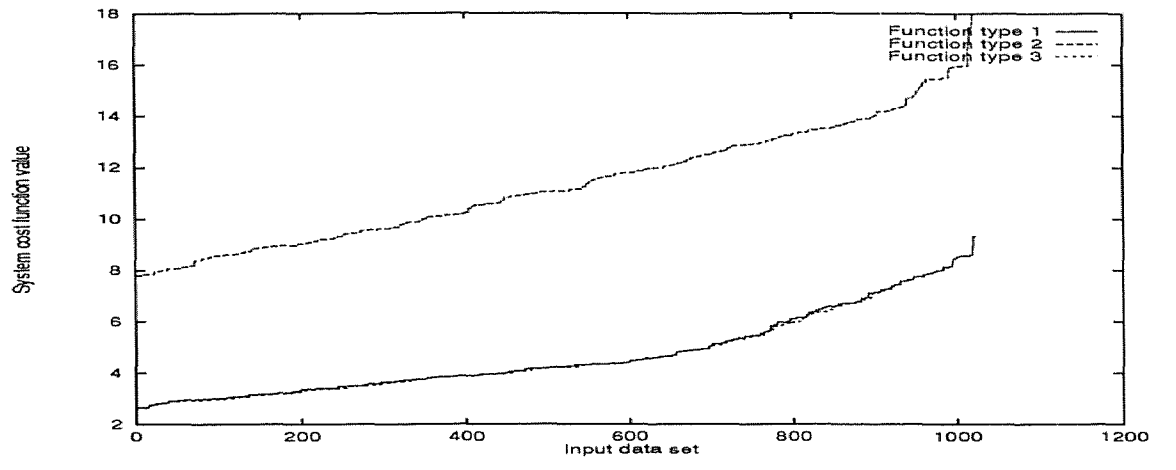


Figure 6.6 Scenario 2—Sorted Cost Function Values.

6.8.3 Weight Comparison Graphs

The weight comparison graphs put the values for different weight distributions for one cost function type into one figure and allow an interpretation of the influence of the branches on the system cost function. We restrict the evaluation of this graph

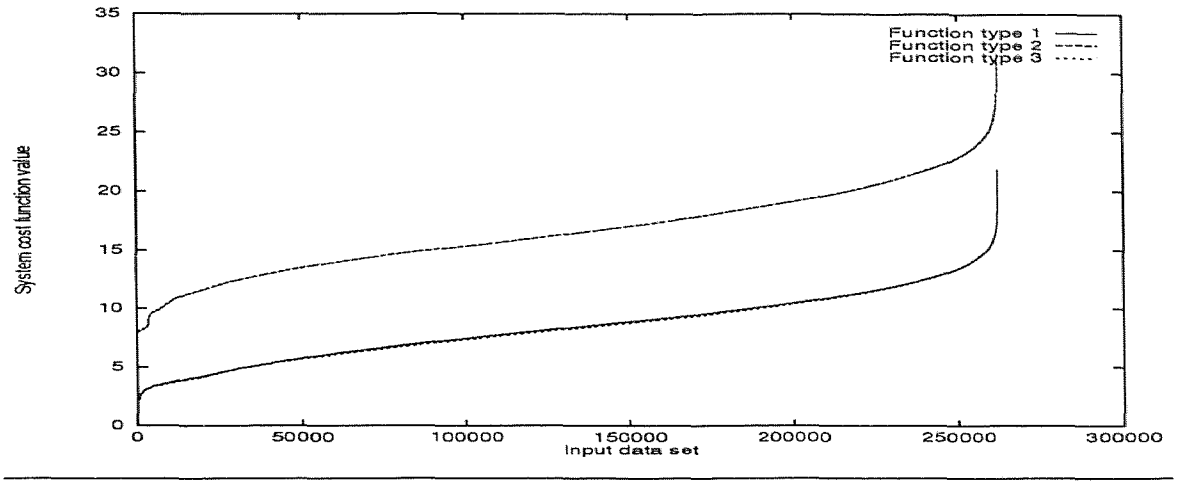


Figure 6.7 Scenario 3—Sorted Cost Function Values.

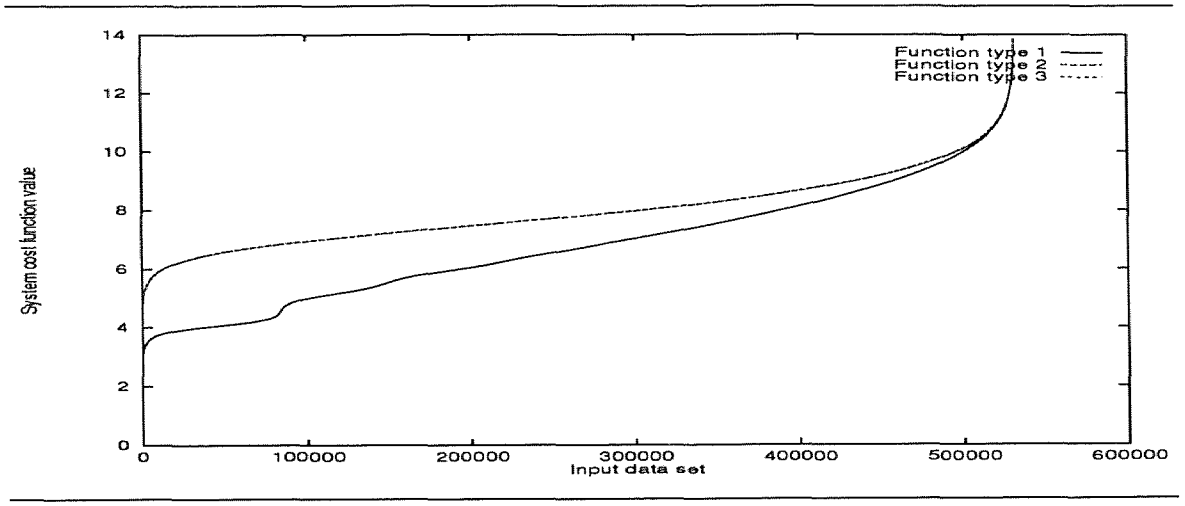


Figure 6.8 Scenario 4—Sorted Cost Function Values.

type to scenario 1 because of its small input data space. The graphs for scenarios 2, 3, and 4 feature a similar behavior.

Several important conclusions can be drawn from these graphs:

- In the case of function type 1, where the performance branch uses a product structure and thus has smaller values (see below), a reduction of the weight of this branch reduces the value range of the system cost function. For function type 2 utilizing a plus-structure, on the other hand, the value range of the performance branch is much bigger than that of the deadline branch. Thus this branch has a greater impact on the system cost function value even if its weight is reduced. For this reason the weight reduction of function type 2 does not yield a significant reduction in the value range of the system cost function.
- A reduction of the weight of the more influencing branch reduces the variation of the function values. Small differences between function values are, however, preserved. Hence the information content is not reduced by the weight change.
- A weight change can even reverse the bias of the system cost function. See, for example, permutation 47 of function type 2, where a local maximum is changed into a minimum by a shift to an 80-20 weight distribution.

6.8.4 GA-Convergence Graphs

Figures 6.11 to 6.14 depict the convergence behavior of the stochastic resource allocator using the three system cost functions.

6.8.4.1 Observations: In the following the most interesting observations regarding the results of the stochastic allocator are listed. An explanation for this behavior is provided in the subsequent section.

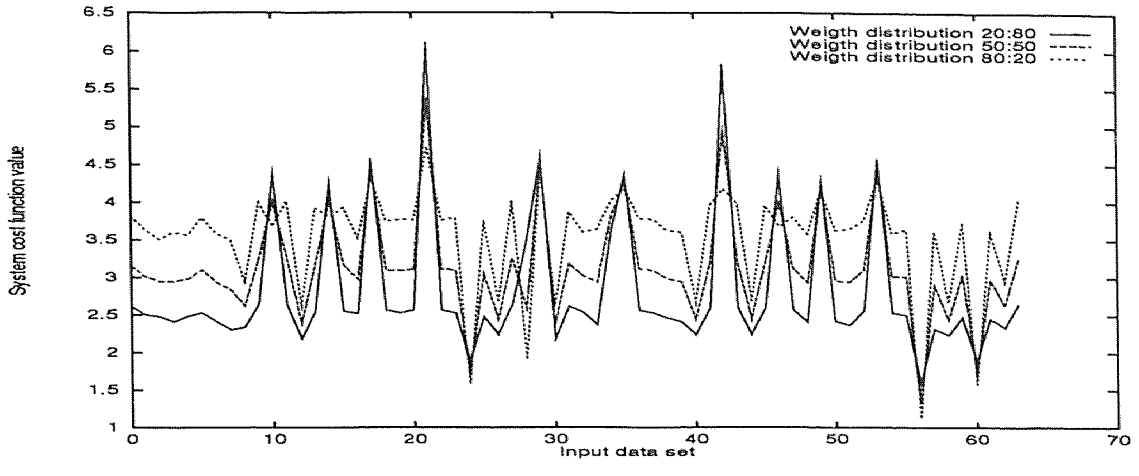


Figure 6.9 Scenario 1—Varying Weights for Function Type 1.

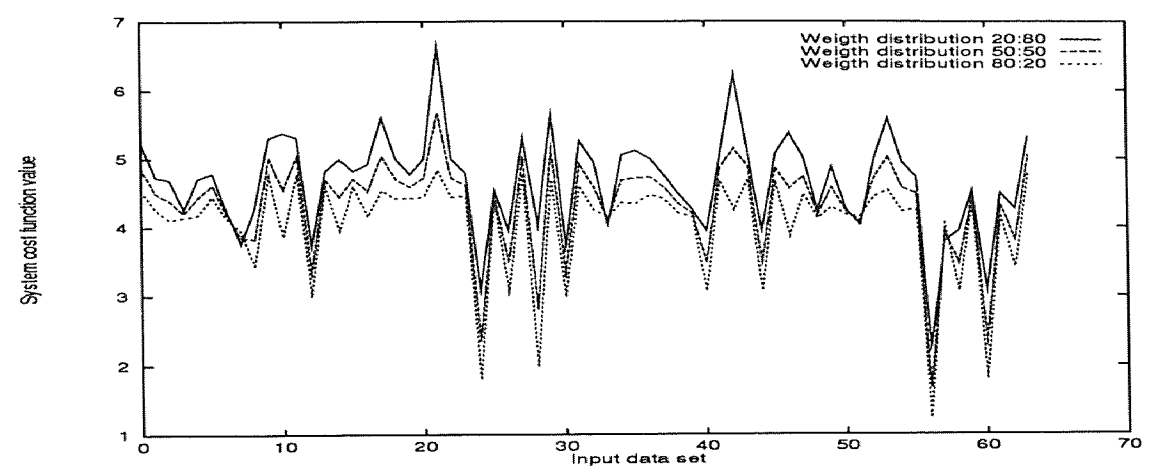


Figure 6.10 Scenario 1—Varying Weights for Function Type 2.

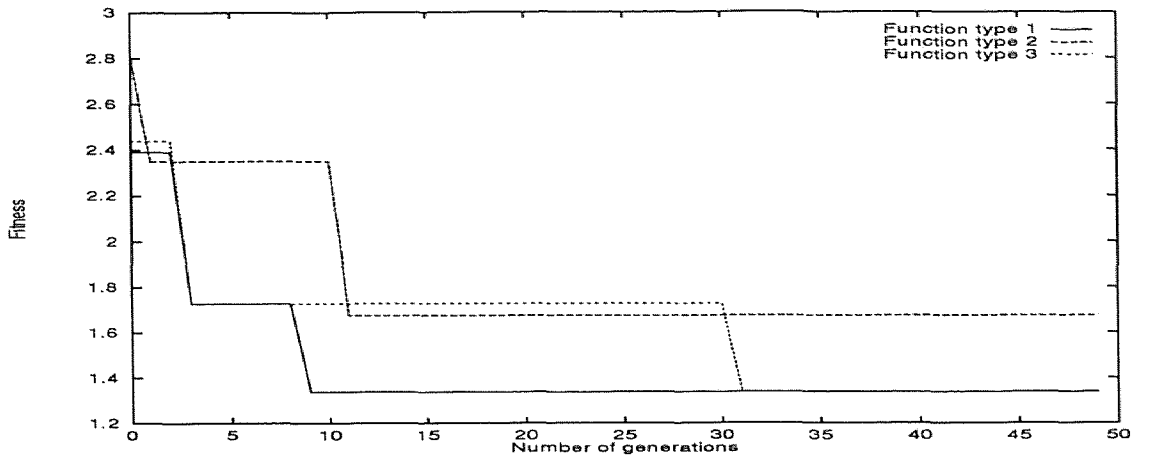


Figure 6.11 Scenario 1—Best Function Value of Each Generation of the GA-Based Allocator.

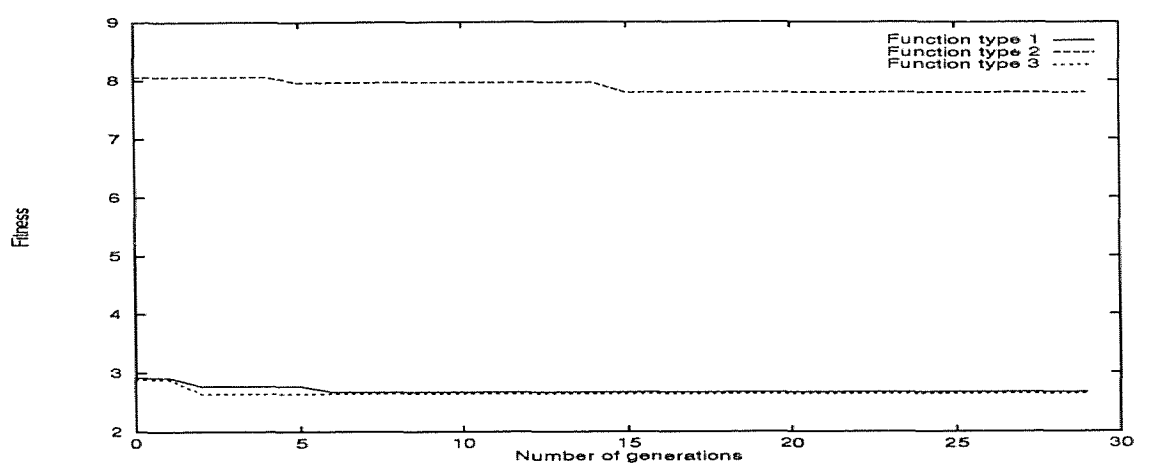


Figure 6.12 Scenario 2—Best Function Value of Each Generation of the GA-Based Allocator.

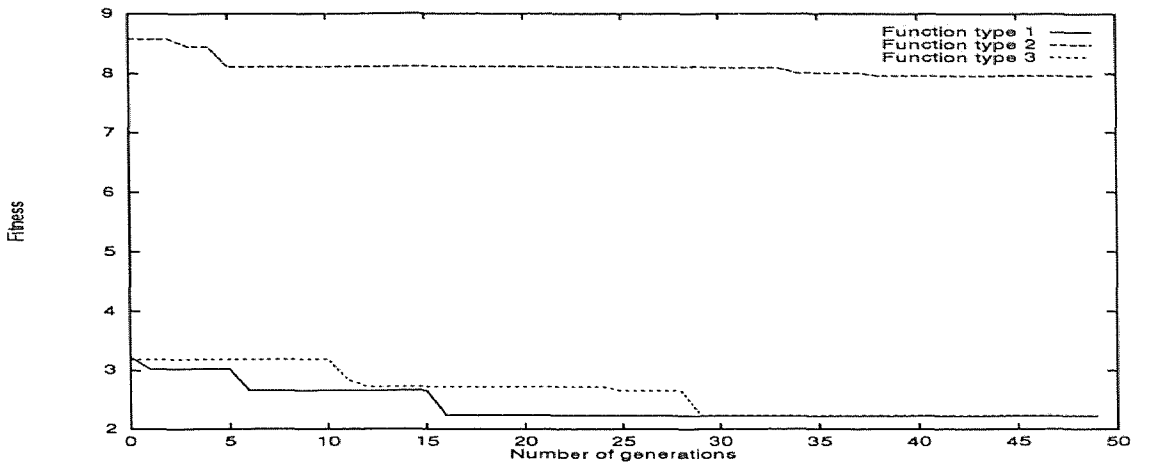


Figure 6.13 Scenario 3—Best Function Value of Each Generation of the GA-Based Allocator.

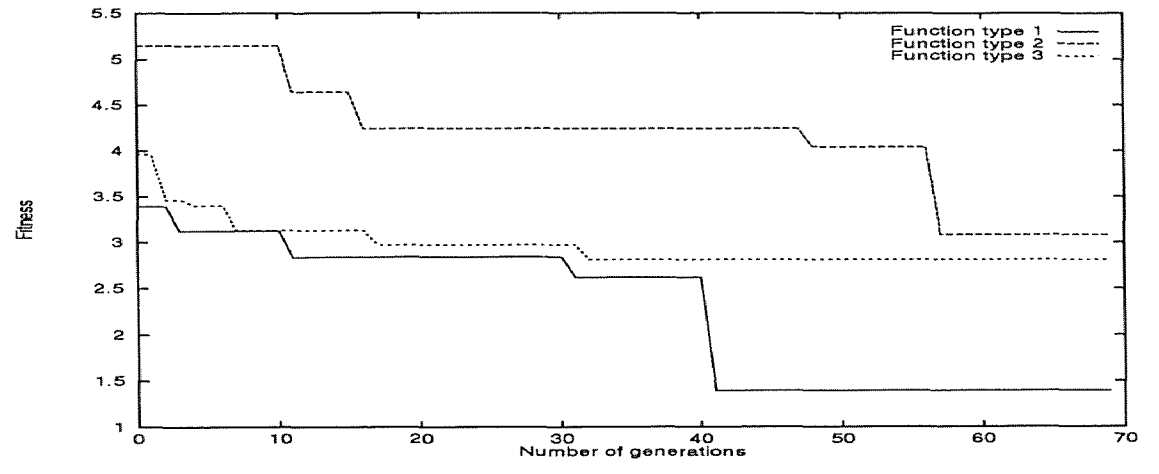


Figure 6.14 Scenario 4—Best Function Value of Each Generation of the GA-Based Allocator.

- At first glance it is obvious that the stochastic allocator does not always yield the best solution. Looking at Figure 6.14, for example, reveals that cost function type 3 gets stuck at a much higher cost value than function 1. On the other hand we know from the exhaustive evaluation that function types 1 and 3 have a similar shape and the same optimal cost in all cases.

- We next compared the best result achieved by the GA-based allocator to the optimal result known from the exhaustive evaluations. In the majority of the test cases the GA-based allocator was able to find the optimal solution. Exceptions are function type 1 in scenario 2 and function type 3 in scenario 4. In particular the latter yields a result that is more than twice the optimal value.

- Regarding the convergence speed one can observe a big difference between the function types and the scenarios. For scenarios 3 and 4, which have a large number of potential solutions, the convergence to the final value typically required up to 50 generations. For the smaller scenarios 1 and 2, on the other hand, the number of generations required to converge was about 20.

6.8.4.2 GA Behavior: All of the above results were obtained by a resource allocator based on Genetic Algorithms, a stochastic optimization method. Generally, a stochastic method does not guarantee to deliver the optimal result.

The performance of the GA cannot be measured by the number of generations alone. Other factors like runtime and memory requirements must also be accounted for. We did, however, not evaluate these additional measures as we were only interested in the GA's ability to achieve the optimal result based on the cost functions created with our approach.

Also note that the performance of the GA depends on the parameter settings. Increasing the population size, for example, normally reduces the number of generations required to find a solution. Other parameters like the crossover and the mutation rate influence the search process, too.

6.9 Decomposition/Synthesis Quality Measures

In Table 6.2 the numbers for the quality measures of the chosen decomposition and synthesis are listed. The table contains the ratio of the number of unique cost function vectors in relation to the size of the input space as well as the number of unique system cost values in relation to the number of unique cost function vectors.

Table 6.2 Quality Measures $\frac{A}{S}$ and $\frac{U}{A}$ for the Attribute Cost Function Selection and the Cost Function Synthesis.

Scenario	Cost Function	$\frac{A}{S}$	$\frac{U}{A}$
1	1	0.703	1.000
2	1	0.184	0.995
3	1	0.177	0.989
4	1	0.094	0.983

At first glance one will notice that the value of $\frac{A}{S}$ is the same for all cost function types within scenarios. All system cost function types are based on the same objective decomposition and attribute cost function selection. As $\frac{A}{S}$ only depends on the latter and the size of the input data space, but not on the synthesis, it must be equal for all function types.

Again, we summarize interesting observations regarding the quality measures:

- For scenarios 2 and 3 that were created by the workload generator, the ratio $\frac{A}{S}$ has an equal value. The ratio for the manually generated scenarios 1 and 4, however, differs from this value significantly. It is as high as 70% for the small scenario 1 and goes down to only 9% for the largest scenario 4.

- For an increasing size of A, the ratio $\frac{U}{A}$ decreased minimally. This means that for larger scenarios with larger sets of unique attribute cost function vectors, the system cost function discriminate between different vectors almost as well as for smaller scenarios.

- The three kinds of syntheses applied to the attribute cost functions preserved at least 99% of the variety compared to the set of chosen attribute cost function vectors. With the given cost function types it is possible to discriminate between these solutions in order to obtain the optimal solution under the given constraints.

- The numbers obtained for the ratio $\frac{A}{S}$ show that the selection of attribute cost functions reduces the variety of the input data space to about 10% for scenario 4 and about 20% for scenarios 2 and 3. Two conclusions can be drawn from these numbers. First, there are several solutions in the input data space that satisfy the given objectives equally well. At the same time this means that the cost functions we picked do not make best use of the variety of the input data space as they do not well discriminate between the potential solutions.

With respect to the last issue listed above, it has yet to be determined whether a better selection would have been possible. Doing the objective decomposition we are confronted with a set of attribute cost functions provided by the system. The variety offered by this set of functions determines the best achievable discrimination between solutions. Hence we derived the number of attribute cost function vectors comprising all available attribute cost functions (see Table 4.3) from the exhaustive evaluation. This number is denoted by C. In Table 6.3 the ratios $\frac{C}{S}$, $\frac{A}{S}$, and $\frac{A}{C}$ are listed.

According to the numbers in the first column giving the ratio between C and S the set of all available attribute cost functions preserves between 80 and 100% of the

Table 6.3 Comparison Between the Variety Offered by the Set of All Provided Attribute Cost Functions and the Variety Used by the Selected Cost Functions.

Scenario	$\frac{C}{S}$	$\frac{A}{S}$	$\frac{A}{C}$
1	1.000	0.703	0.703
2	1.000	0.184	0.184
3	1.000	0.177	0.177
4	1.000	0.094	0.094

variety of the input data space. This proves that the set of attribute cost functions provided by the system allows to discriminate between the potential solutions to a high extent.

As a consequence, one would assume that it is better to select as many attribute cost functions as possible. It seems to be a rule that the more attribute cost function one selects, the better is the discrimination of the input space. This, however, is true only in a limited number of cases. In general, one should consider two issues before applying the above guideline.

Firstly, there is a limit to discrimination given by the fraction $\frac{A}{S}$. If this ratio is already close to 100% like in scenario one in Table 6.3, there is no need for further discrimination. The cost functions chosen so far do already offer a fine-grained evaluation of the input space.

Secondly, the main concern during the objective decomposition is to represent all objectives governing the system design in an appropriate manner. If one adds an attribute cost function only in order to increase the number of unique cost function vectors, the resulting decomposition does *not* represent the initial system goals any more. In other words, if the objective decomposition and the mapping of attribute cost functions results in a small number of unique cost function vectors, one should

question the objectives rather than add attribute cost functions that do not reflect system objectives.

6.10 Interpretation

In this section we discuss general trends observed in the combining function. We focus mainly on the additive and multiplicative forms.

Throughout the experimentations it was observed that for some experiments the product structure yielded lower values than the additive (see Figure 6.15), while for others almost identical results were observed (see Figure 6.16). In addition, both methods produced identical best and worst solutions. These observations can be explained from the way the combining functions behave.

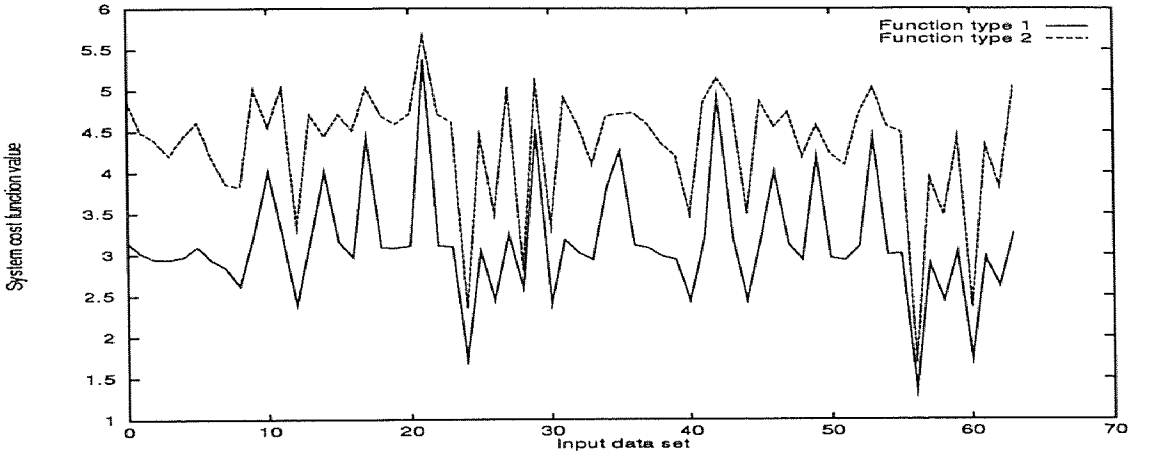


Figure 6.15 Sum yielding Larger Values Than Product Structure.

Given two cost function C_1 and C_2 with values x_1, x_2 and weights w_1, w_2 respectively, each graph in Figure 6.17 gives their relative aggregate cost for both product and additive structures under any weight configuration —since $w_1 + w_2 = 1$, than $w_1 = 1 - w_2$. Each graph corresponds to the disparity between x_1 and x_2. For example, The greater the difference between the two values the less the product

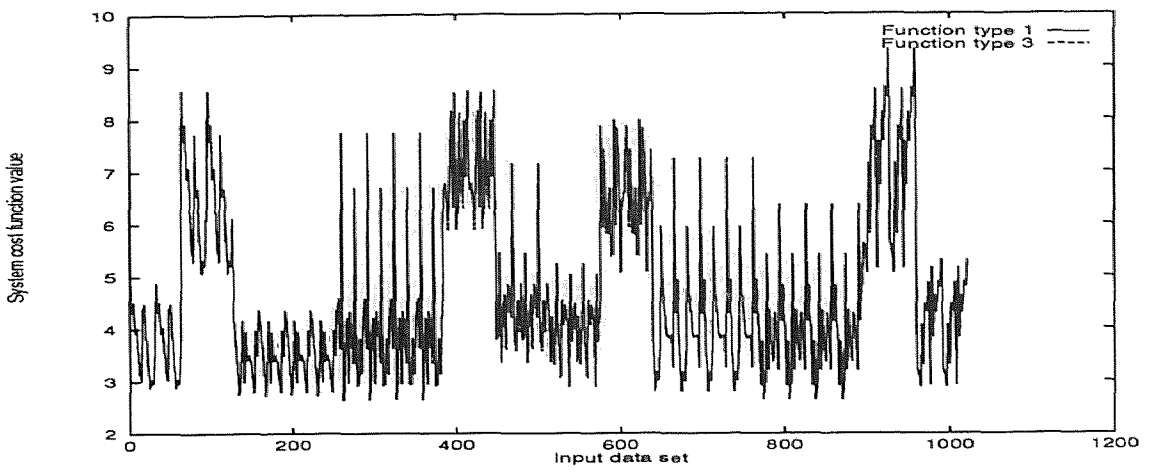


Figure 6.16 Sum and Product Yielding Similar Results.

structure will yield in comparison to the additive form, for any arbitrary set of weights.

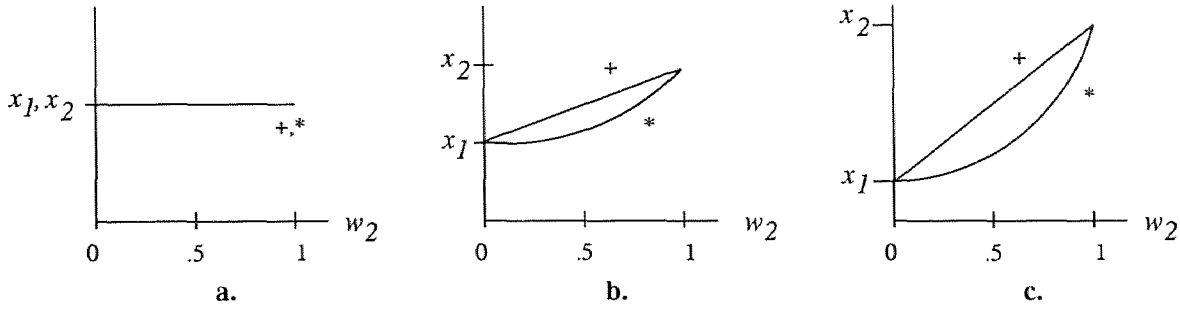


Figure 6.17 Effects of Sum and Product Structures in Combining

Hence, if the values being combined are relatively the same amount than it does not matter which combining function is used, either additive or product, see Figure 6.17.a. If on the other hand, the values are disproportionate then the sum yields higher values than multiplication, see Figure 6.17.c. This is because the Product structure favors lower values. An example is outlined Table 6.4.

Separately, an elastic affect was observed in some experiments. While sorted results of product structures were consistently lower than the additive and extremes nearly identical (see Section 6.8.2 Figure 6.8), product form outcomes were more

Table 6.4 Example of Sum and Product Behavior

x_1	x_2	w_1	w_2	Σ	Π
100	3	.3	.7	32.1	10.6
		.5	.5	53.5	20.1
		.8	.2	80.6	52
60	40	.3	.7	46	45.2
		.5	.5	50	48.9
		.8	.2	56.6	55.3
50	50	.3	.7	50	50
		.5	.5	50	50
		.8	.2	50	50

spread out. This is because at extremes operand values are similar, in between the values may be disproportionate. Thereby, stretching the product structure values lower, as seen in Figure 6.18.

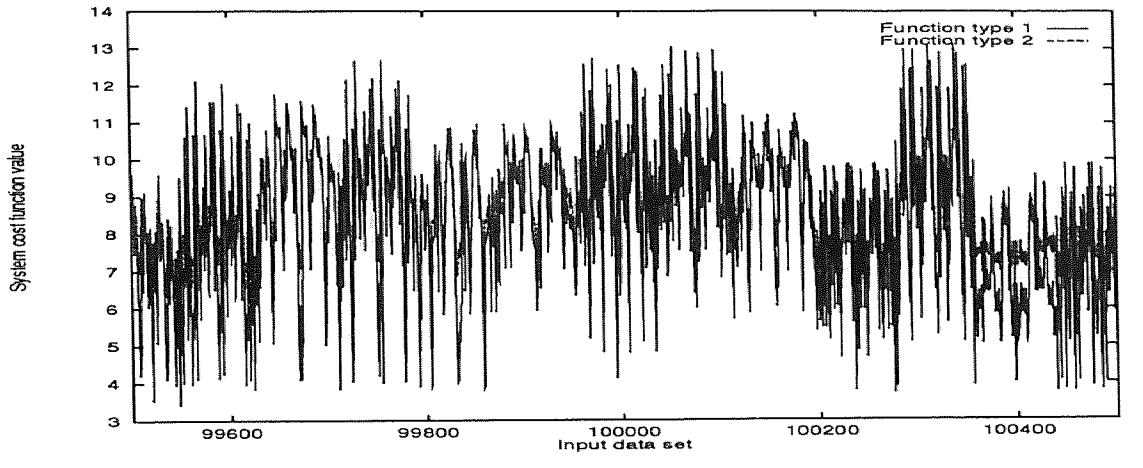


Figure 6.18 Sum and Product Elastic Affect.

6.10.1 Guidelines

Here, we outline general guidelines for using combining functions.

Guideline 1 All values must be of same scale regardless of combining function.

Guideline 2 When total cost x is to depend on actual amounts the values must be of same *unit type* and domain.

Guideline 3 When total cost x is to depend on relative amounts use a product structure.

Guideline 4 When using product structures avoid zero value operands.

6.11 Inter-completion Time Scheduling (*ICTS*)

In this section we study the effects of the inter-completion time Scheduling strategy on Inter-Processor Communication. The performance parameters of interest are number of messages delayed due to contention and total message delay. We will not compare our approach to other scheduling disciplines for obvious reasons.

In all our simulations, we have assumed a task system where each task, after completion, must send a message to a central task located on a dedicated processor p_0—such as in logging, voting, and multi-sensor single actuator systems. In all these systems, it is crucial that data not get old and stale. We have chosen this type of system to make demonstration and verification of inter-completion time scheduling easier. However, the trends should remain the same for arbitrary systems.

In our simulation we assume an arbitrary communication network and a single link with capacity $1packet/ms$ connecting processor p_0 to the network, see Figure 6.19. We also assumed the network links to be at least as fast as the link connecting p_0 and that all processors $p_1, \ldots, p_m$ are relatively the same time distance from p_0.

We arbitrarily chose task systems with 100 identical tasks, execution time requirements E of 100ms, 500ms, and 1000ms, a release time of 0, deadline D of $50 \times E$, and message size X. Since we want to have different message sizes X in relation to task size, we chose $X = \frac{1}{2}E, E, 2E$. We ran these systems on 2, 5, 10, and 20 identical processor networks.

Primarily, our interest is in network communication behavior under different inter-completion times. The results of the simulations are in Figures 6.20 thru 6.25. In all Figures inter-completion time is represented as a ratio to message size (i.e. $\frac{MGICT}{X}$).

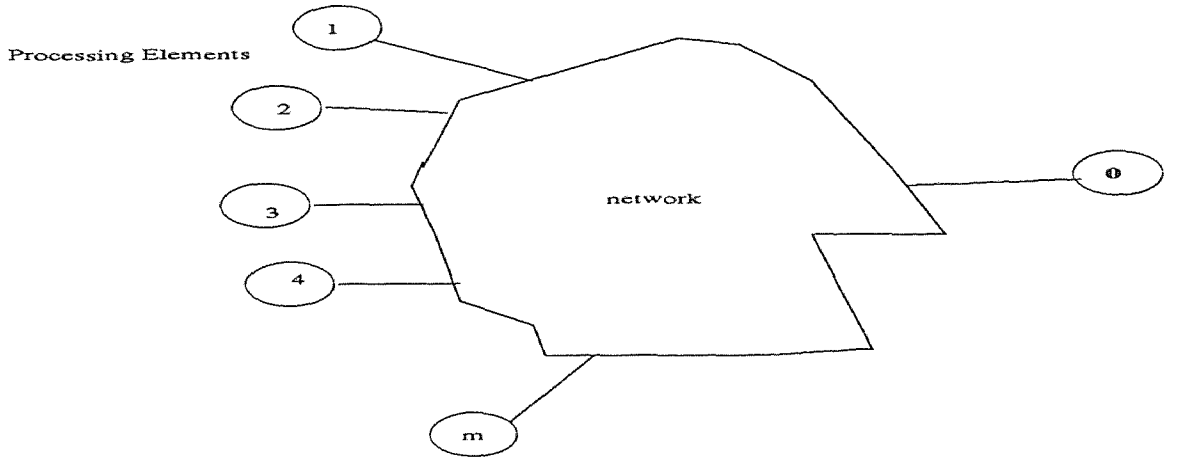


Figure 6.19 Sample Network

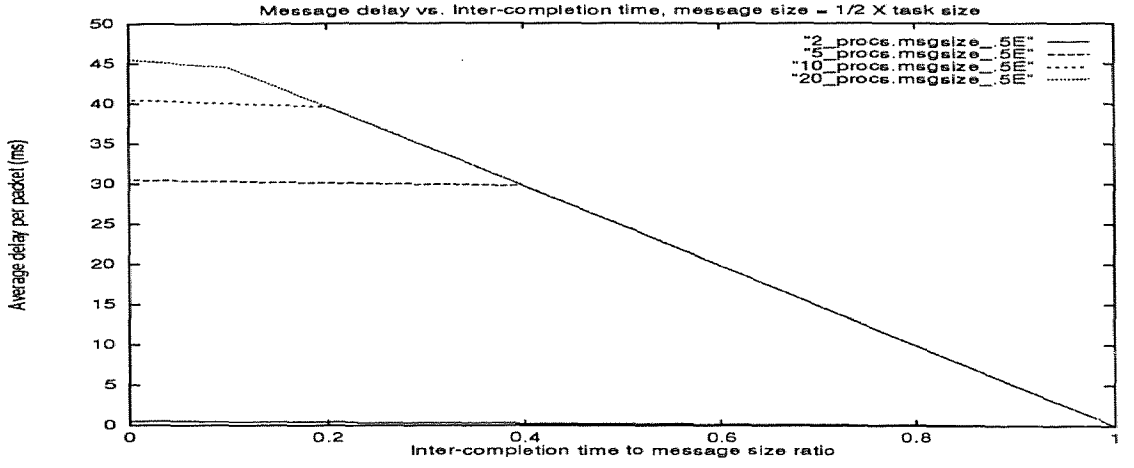


Figure 6.20 Average Communication Delay vs. Inter-Completion Time, $X = \frac{1}{2}E$.

The results presented in this section indicate that it is possible to achieve higher communication performance through inter-completion time scheduling. In all simulations, inter-completion time demonstrated a significant effect on both the number of packets delayed and for how long. When inter-completion time is zero

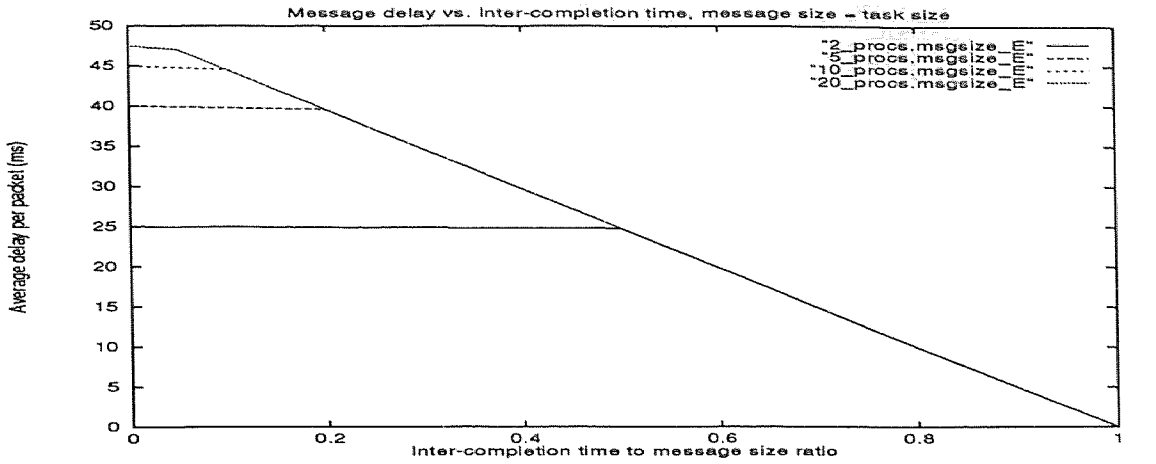


Figure 6.21 Average Communication Delay vs. Inter-Completion Time, $X = E$.

the communication delays are greatest. When inter-completion time is greater than or equal to message size communication delays are negligible. This is as expected, because there is sufficient time for messages to be received before subsequent messages arrive.

We also note that inter-completion time and processor utilization in general are inversely related. Therefore as inter-completion time is increased, utilization may decrease.

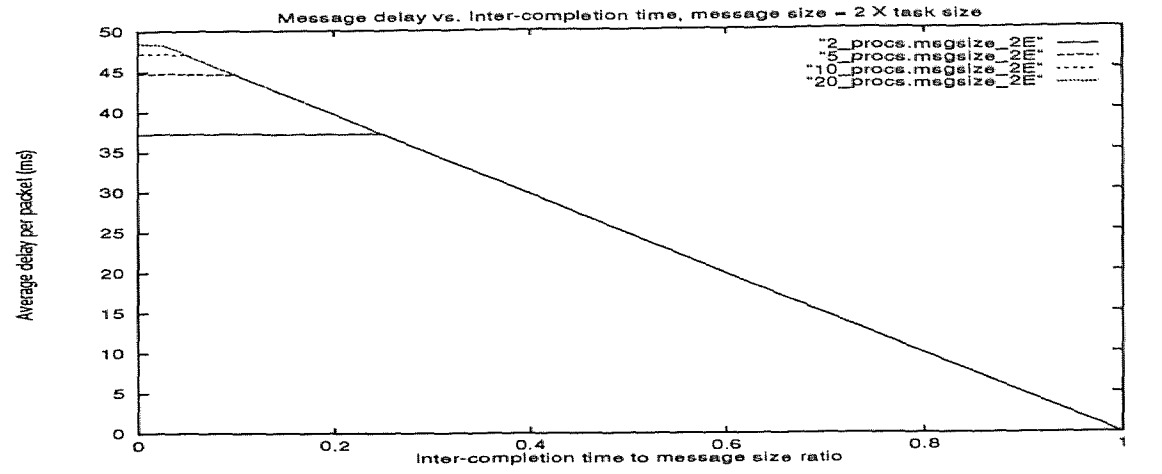


Figure 6.22 Average Communication Delay vs. Inter-Completion Time, $X = 2E$.

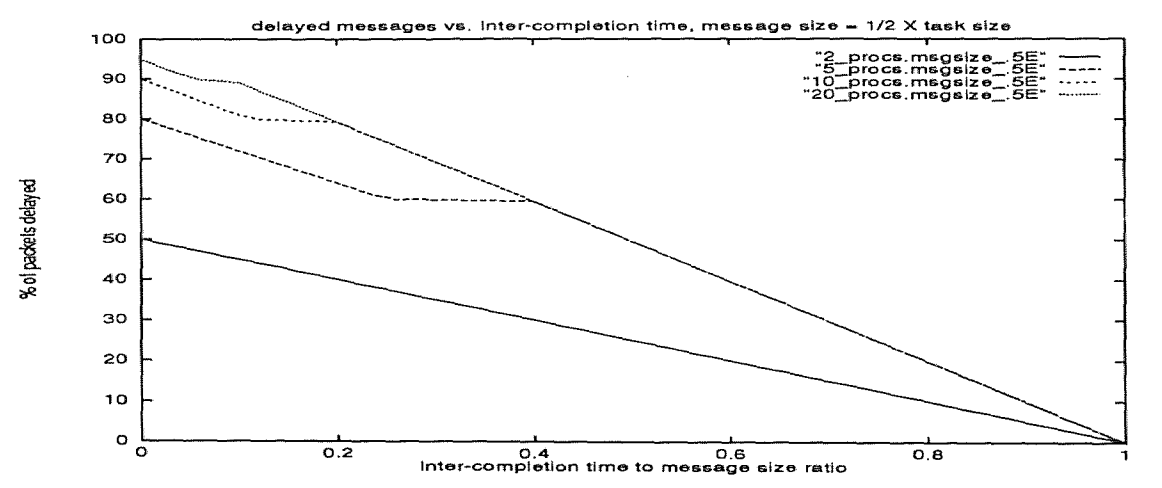


Figure 6.23 % of Packets Delayed vs. Inter-Completion Time, $X = \frac{1}{2}E$.

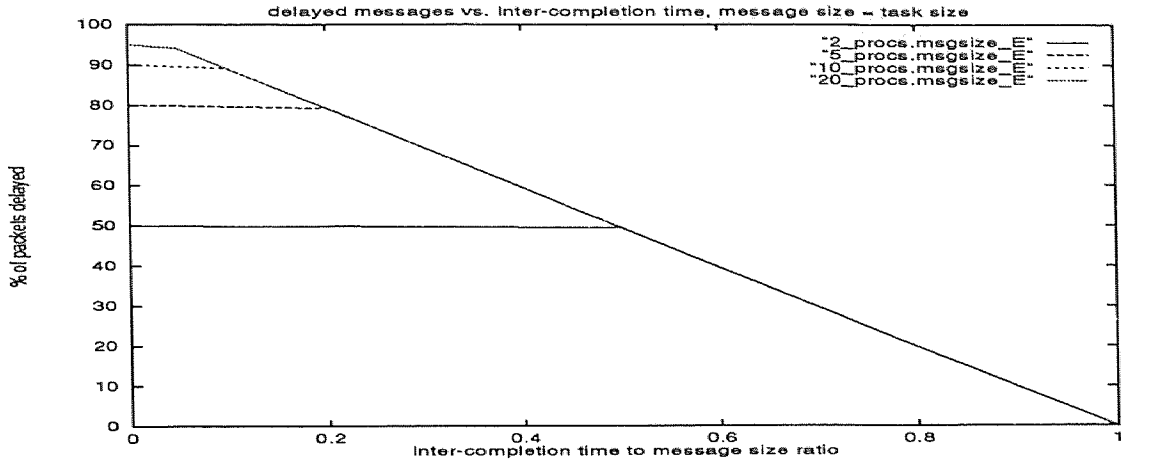


Figure 6.24 % of Packets Delayed vs. Inter-Completion Time, $X = E$.

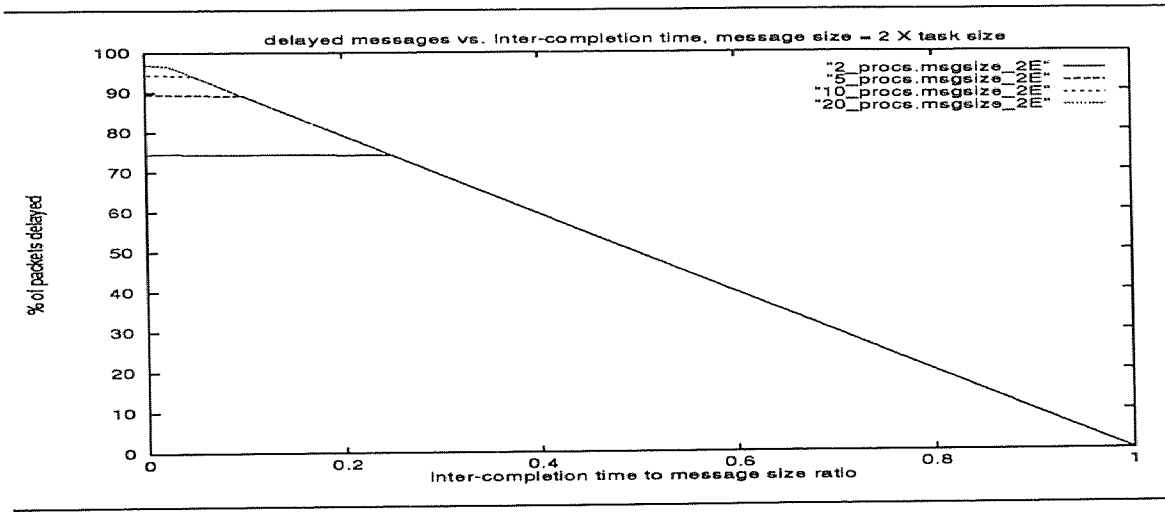


Figure 6.25 % of Packets Delayed vs. Inter-Completion Time $X = 2E$.

CHAPTER 7

IMPLEMENTATION AND TEST ENVIRONMENT

In the previous chapter, the impacts of various combining characteristics on the applicability of cost function synthesis have been studied. This chapter includes a description of the prototyping efforts to implement and test the applicability and usefulness of the data transformation functions of Chapter 4.

A REsource ALlocation prototype for complex systems, called REAL [6], is being built at the Dependable Real-Time Systems Laboratory at NJIT. The prototype includes an objective decomposer/cost function synthesizer supporting various types of transformations as discussed in Chapter 4, an evaluator for assessing cost functions, an allocator for task-to-processors assignment, a symbolic executer for estimating post runtime attribute costs, and a workload generator for creating random systems.

This chapter is organized as follows. First, the work load generator is discussed. Next, the synthesizer used to generate the cost function is described. Then, the allocation mechanisms employed in the platform are outlined. Next, the symbolic executer is discussed. Finally, the system cost function evaluator is described.

7.1 Workload Generator

The Workload Generator (WLG) is a principal component in the simulation and testing tool. For a given problem size, the Workload Generator creates a random task system. A system can comprise multiple independent task DAGs —each can either be real or non-real-time. This allows the existence of the mixed task type set for complex systems. Currently unrestricted models are generated. A later version of the prototype will incorporate a Constraint Manager for verifying constraint satisfaction.

no_of_pes	Number of processors in the hardware model
pe_types	Number of types of processors in the model
instruction_types	Number of instruction types in the software and hardware model
no_of_tasks	Number of tasks in the software model
max_no_of_dags	Maximum number of independent DAGs in software model
dag_size	Dag size factor [0, 1], 0-smallest 1-largest
max_children	Maximum number of children
max_exec_time	Maximum execution time for any instruction
min_exec_time	Minimum execution time for any instruction
min_mesg_size	Minimum size of any message
max_mesg_size	Maximum size of any message
min_sw_instructions	Minimum possible number of software instructions in a task
max_sw_instructions	Maximum possible number of software instructions in a task
seed	Random number generator SEED
rt_percentage	Real-time task-DAG percentage
current_time	current time
window_size_factor	Execution window size factor

Figure 7.1 Workload Generator Parameters

In the next sections the WLG's input parameters and output data files are described.

7.1.1 Input Parameters

The WLG requires quite a few number of parameters for input. Figure 7.1 describes these parameters. Figure 7.2 presents an example of the parameters for generating a two processor systems with 21 tasks. We will not discuss all parameters, since most are self explanatory. However, some do require a brief explanation. We discuss those in the following:

- **[pe_types]** Although, heterogeneous processors are supported, there may be some that are alike. *Pe_types* sets the maximum number of processor types. Setting this value to 1 creates homogeneous processor environment. The larger this value the greater the probability of obtaining unique processors.

- **[instruction_types]** The number of unique instructions executable by the entire hardware environment.

- **[dag_size]** Dag size sets the upper limit on the amount of tasks a task-DAG can have. For example, when *dag_size* = .3, at most 30% of the remaining tasks can be designated for the next task-DAG generation. For lesser values, many small task-DAGs will be generated. For larger values the reverse will be observed.

- **[max_children]** Max children sets the maximum number of descendants, in a control flow graph, any task can have. The larger this value the bushier the DAG will be. When set to 1, the result will be a sequential set of tasks. The minimum value is always 1.

- **[window_size_factor]** This value is used in determining the amount of time in which a task may execute. Moreover, it is used to compute the separation between *release time* and *deadline*. The greater the window size factor the greater the potential for a larger separation.

  In the next section we describe data file generated by the WLG.

7.1.2 System Description File

The output file, generated by the WLG, is split into 5 parts. The first part consists of general system information for the executer. Then, task execution time requirements are specified. Next, is the task communication matrix. Then, the initial allocation vector is specified. Finally, task timing characteristics are described. The file format is given in Figure 7.3. Each part is described in more detail in the following.

1. **[General Information]** The first value in the general system information is the *interval*. The *interval* delimits real-time from non-real-time tasks. Any

no_of_pes	2
pe_types	21
no_of_tasks	10
max_no_of_systems	3
system_size	.3
max_children	2
instruction_types	10
max_exec_time	2
min_exec_time	1
max_sw_instructions	2
min_sw_instructions	0
min_mesg_size	3
max_mesg_size	7
seed	3
rt_percentage	100
window_size_factor	1.6

Figure 7.2 Sample Parameter Values

task with a deadline greater than the interval is recognized as a non-real-time task. The next value is the *current time*, which is typically set to zero. The final two values are the number of processors m and the number of tasks n in the system respectively.

2. [**Execution Matrix**] The Execution time matrix follows the general system description. It is an table of size [no_of_pes $\times$ no_of_tasks]. For each task there are m entries, since processors may not be homogeneous.

3. [**Communication Matrix**] The communication matrix is a two dimensional table of size $n \times n$. Each i, j value corresponds to the size of the communication message between the i^{th} and j^{th} task.

4. [**Allocation Vector**] The allocation vector is a simple integer list of size n. Each value in the list corresponds to a processor in the system. Therefore, each value must lie in the interval [1,m].

Interval
Current Time
Number of processors
Number of tasks

Execution time matrix [no_of_pes × no_of_tasks]

Communication time matrix [no_of_tasks × no_of_tasks]

Allocation Vector [no_of_tasks]

Task variable matrix [no_of_tasks × no of variables]:
release_time scheduled_start_time deadline start_time completion_time
recovery_time failure_start_time

Figure 7.3 System Description File Format

5. [**Timing Information**] The last part constitutes detailed task timing information, such as: release time, deadline, and recovery time. The balance of the parameters serve as place holders for future implementation.

A sample description created by the Work Load Generator is given in Figure 7.4. In the next section, the Cost Function Synthesizer is described.

7.2 System Cost Function Synthesizer

The synthesizer is an integral component in the prototype. It is an interactive tool that assists in the decomposition of objectives and construction of the system cost function. It can build any decomposition, given a set of attribute cost functions and transformation functions. Also, it stores previous hierarchy models for facilitating decomposition decisions.

```
120
0
2 10

10 10 13 14 17 18 4 12 8 26
13 11 18 17 18 19 5 13 9 30

0 3 0 0 0 0 0 0 0 0
0 0 3 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0

0 0 0 0 3 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 7 0 0 0
0 0 0 0 0 0 0 4 4 0
0 0 0 0 0 0 0 0 0 4
0 0 0 0 0 0 0 0 0 6
0 0 0 0 0 0 0 0 0 0


1 1 1 1 1 1 1 1 1 1

33 0 46 0 0 0 0
49 0 65 0 0 0 0
68 0 95 0 0 0 0
48 0 68 0 0 0 0
71 0 97 0 0 0 0
0 0 19 0 0 0 0
26 0 45 0 0 0 0
49 0 75 0 0 0 0
49 0 68 0 0 0 0
79 0 120 0 0 0 0
```

Figure 7.4 Sample System Description Created by WLG

7.2.1 Input Parameters

There are two input channels for the Synthesizer, from disk and terminal. First, a list of available attribute cost functions and data transformations along with a file containing default decompositions is loaded. Default decompositions have the same form as system cost function description in the next section.

Then, as previously mentioned, the Synthesizer is an interactive tool. Therefore, key information reflecting the *decision makers* (DM) opinion must be entered via a terminal. For a given internal node in the hierarchy, the DM is given alternative decompositions specified in the default file. If those are insufficient, the DM may type in her own description. The decomposition continues in this fashion until all leaf nodes are attribute cost functions.

The synthesizer output file is described in the next section.

7.2.2 System Cost Function Description

The *cost function description file* consists of all the information needed for building the hierarchy. It contains all objective decompositions and corresponding transformation functions. The file is comprised of three parts, they are: node counts, internal node description, and edge information. A sample is given in Figure 7.5.

The first part, *node counts*, consists of two variables. These values describe the total number of nodes followed by the number of leaf nodes in the hierarchy. The number of internal nodes is intentionally omitted because it can be derived from these two given values.

Next, internal nodes are described. Information such as node names followed by combining function can be found here. Note, all functions must have three parameters, regardless if they are used or not.

Finally, the edges between nodes are described. On each line, the higher level node is specified first, followed by one of the lower level nodes. Then, all trans-

formation functions for the lower level node are defined. They are specified in the following order: scale, scaling,fusion,algebraic, and weight.

7.3 Resource Allocators

Two allocation mechanisms are utilized, exhaustive and Genetic. The Genetic mechanism was incorporated into the allocation component from [53].

The exhaustive mechanism takes three parameters: an integer vector, lowest processor number, and highest processor number. First, it initializes all vector fields to the minimum processor number. Then, it increments the first field. If the new value exceeds the maximum processor number, then the current field is initialized again and the next field incremented by one. This process continues until all vector field values equal the maximum processor value. In essence, it works much like a numerical counter with each vector field corresponding to a digit.

The genetic algorithm, as explained in the previous chapter, imitates nature's process of evolution by taking one population as parent generation and creating an offspring generation. The algorithm selects the best individuals of the parent population according to the fitness value which is generated by the system cost function. The better the fitness value of an individual, the greater is the probability that it is selected for reproduction. After the selection the genetic algorithm mutates some of the genes of the selected individuals by flipping bits according to a given mutation rate and performs cross-over between the individuals by swapping parts of chromosomes. As a result of this process a new population, a child generation, is created, which is then evaluated. The whole process iterates for a specified number of iteration. The genetic search strategy was implemented by Roman Nossal. Details on the algorithms implementation can be found in [53].

```
11 6

system product 1 1
real_time product 1 1
performance product 1 1
deadline_satisfaction sum 1 1
average_latency divide 1 1

latency sum 1 1
missed_deadline sum 1 1
consecutive_missed_deadline sum 1 1
consecutive_missed_deadline_pe sum 1 1
system_completion_time sum 1 1
communication_delay sum 1 1

system real_time identity 1 1 identity 1 1 identity 1 1 identity 1 1 power .5 1
system performance identity 1 1 identity 1 1 identity 1 1 identity 1 1 power .5 1
real_time deadline_satisfaction identity 1 1 sum 1 1 identity 1 1 identity 1 1 power .5 1
real_time average_latency identity 1 1 sum 1 1 identity 1 1 identity 1 1 power .5 1
deadline_satisfaction missed_deadline identity 1 1 identity 1 1 identity 1 1 identity 1 1
product 0 1
deadline_satisfaction consecutive_missed_deadline identity 1 1 identity 1 1 identity 1 1
identity 1 1 product .3 1
deadline_satisfaction consecutive_missed_deadline_pe identity 1 1 identity 1 1 identity 1 1
identity 1 1 product .7 1
average_latency latency identity 1 1 identity 1 1 identity 1 1 identity 1 1 power .5 1
average_latency missed_deadline identity 1 1 sum 1 1 identity 1 1 identity 1 1 power .5 1
performance communication_delay identity 1 1 sum 1 1 identity 1 1 identity 1 1 power .7 1
performance system_completion_time identity 1 1 sum 1 1 identity 1 1 identity 1 1 power
.3 1
```

Figure 7.5 Sample Cost Function Description File

7.4 Symbolic Executer

The symbolic execution of processes is handled by a single component. The executer reads the system description file generated by the work load generator and symbolically executes it in an Earliest Deadline First fashion [30].

The executers responsibilities include: administering queues, initiating execution of tasks, and managing communication. The processor interconnection topology and network is also simulated by the Executer. Moreover, it manages message propagation delays and queuing. Currently, the simulation supports a bus topology.

7.5 System Cost Function Evaluator

The evaluator reads the cost function file, Figure 7.5, generated by the synthesizer and creates the hierarchy. For every allocation the Evaluator computes the atomic attribute cost functions and then evaluates the system cost function in a recursive fashion.

Currently the following transformation functions are supported: exponential, power, linear, square root, identity, addition, multiplication, subtraction, and division. A detailed description is given in Figure 7.6. All functions accept three parameters, this is for form consistency. However, not all functions utilize all parameters. Unutilized parameters are ignored.

The evaluator applies the transformation functions in the following order: scale, scaling, fusion, algebraic transformation, weight, and finally the combining function is applied. However, since all functions have the same structural form, it would be quite simple to modify the order in which they are executed by changing the description file accordingly.

```
double exponential(double x, double y, double z)   { return(exp(x)); }
double power(double x, double y, double z)         { return(pow(x,y)); }
double linear(double x, double y, double z)        { return( y * x + z); }
double root(double x, double y, double z)          { return(sqrt(x)); }
double identity(double x, double y, double z)      { return(x); }
double sum(double x, double y, double z)           { return(x+y); }
double product(double x, double y, double z)       { return(x*y); }
double subtract(double x, double y, double z)      { return(x-y); }
double divide(double x, double y, double z)        { return(x/y); }
```

Figure 7.6 Supported Transformation Functions

```
system_completion_time
slack
latency
laxity
missed_deadline
consecutive_missed_deadline
consecutive_missed_deadline_pe
consecutive_missed_deadline_global
load_balance
communication_cost
PE_utilization
min_intercompletion_time
global_MICT
communication_delay
communication_overlap
communication_overlap_ratio
```

Figure 7.7 Available Attribute Cost Functions

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

In this thesis, we have concentrated on the construction of an objective hierarchy model, objective function formulation and evaluation, and cost-function development for identifying better allocations in complex systems. While there has been much work in the area of resource allocation in computer science and other fields, we have shown that none of the approaches are fully suitable for complex systems, insofar as they do not accommodate the multitude of objectives inherent to complex systems. In addition, we have introduced a new scheduling discipline, namely, Inter-Completion Time Scheduling (ICTS).

Moreover, we have outlined and discussed the relationships among top level system design, objectives, and attribute cost functions and the construction of system cost functions. We introduced a hierarchical model for such synthesis. We demonstrated, through example, how our model is applicable in complex real-time systems. Finally, we have implemented the model in a working platform.

We have seen how devising mechanisms to evaluate the inherent goodness of a given allocation is not a simple chore. It requires a thorough understanding of how individual design elements interact with each other. This interdependency may require that assumptions and approximations be made so that meaningful combination of multiple objectives and their cost functions are possible.

We have introduced, as an objective, deadline balancing (DLB) and representative cost functions. We extended and formalized this notion of DLB into the concept of scheduling to maximize inter-completion time —Inter-Completion Time Scheduling (ICTS). We have proposed the new task scheduling strategy for single and multiprocessor systems. Simulation results indicate that the proposed strategy achieves higher communication performance in multiprocessor systems.

We have shown that both MICT- and MGICT-scheduling problems are, in general, NP-hard, and have studied a wide variety of special cases of task systems, where each special case is distinguished by being restricted along some of its degrees of freedom. For a large number of these special cases, we presented very efficient scheduling algorithms; others we proved NP-hard.

8.1 Future Work

The work presented throughout this thesis can be extended in many directions. First, the system model can be extended to incorporate much more complex resource and task structures. Next, the tool may be enhanced. Also, the applicability of dynamic –as opposed to static– objective evaluation should be studied. Furthermore, the inter-completion time scheduling algorithm may be extended to more robust models. We will now discuss these to some detail.

From the decision makers point of view, the important development is a robust graphical tool for specifying and creating objective functions. This tool would give users the capability to select and construct objectives graphicly. In addition, automatic generation of functions may be possible. Given an English synopsis of system objectives, the tool could synthesize and/or recommend objective functions. Such a tool would be of great benefit for system planners because it would require less interaction and yield more standard results.

This thesis focussed on objective satisfaction and therefore largely ignored constraints. However, for complex systems, system constraints need to be addressed. Therefore, a constraint component also need to be build for proper system evaluation. Such a tool allow the decision maker to explicitly determine constraints on and between individual system elements. Any constraint manager should also include a consistency checker.

In addition the tool should be extended to support arbitrary network topologies, such as token ring, star, mesh, etc.. Additional transformation functions and attribute cost functions may also be needed to quantify other objectives, such as: fault-tolerance, security, and so on.

At some point it would be interesting to look at how to transform the static implementation to perform allocation evaluations at runtime. Dynamic evaluation would allow us to perform continuous optimization. We would be able to take advantage of situations otherwise undetected prior to run time.

Also, we would like to extend ICTS to more robust models. Our current research efforts include using the reduction theorem of Section 5.4 to obtain MICT- and MGICT-scheduling algorithms for other problems which have tractable corresponding feasibility problems.

In closing, we believe that our research enhances the confidence of complex system developers in allocating resources, by allowing the decision maker to directly address objective satisfaction. Our study provides guidelines for objective decomposition and cost function synthesis. We believe that the studies we are conducting are essential for the design and development of complex systems. The development of such systems will require the assistance of resource allocation tools and techniques to fine tune the 'performance' and enhance objective satisfaction without violating the requirements of the system, and thereby reducing cost. In addition, currently running applications can also benefit, by applying such a tool to investigate the existence of better allocations.

APPENDIX A

NOTATION

s	system characteristics
o	top-level non-functional objectives
a	a solution
φ	auxiliary information
X_i	node i in the objective hierarchy
k_i	number of children of X_i
$\mathbf{x}_i$	children evaluation vector; $\mathbf{x}_i \equiv (x_1, x_2, ..., x_{k_i})$
o_i	objective i
C_i	cost function of o_i
I_j	the j^{th} atomic cost function
x_i	value for C_i
$\mathcal{I}$	identity function
$\mathcal{T}$	scale type function
$\mathcal{S}$	scaling function
$\mathcal{F}$	fusion function
ϕ	algebraic transformation function
w	weight function
β	combining function
$\mathcal{D}$	data transformation functions: $f(\mathcal{T}, \mathcal{S}, \mathcal{F}, \phi, w, \beta)$

APPENDIX B

GLOSSARY

Algebraic transformation function (ϕ). An algebraic transformation function corresponds to fitting a given "error" or "penalty" model, so that, for example, an exponential function penalizes a few large values in comparison to average values for all inputs, while a square-root function does the opposite.

Allocation (a). An allocation is a mapping of tasks to processors. It specifies how and where tasks are to be assigned.

Attributes. Atomic characteristic behavior of a system.

Auxiliary information (φ). Information that cannot be derived from allocation only data. Typically, it is acquired through actual or symbolic execution of the tasks. For example, in order to evaluate a number of cost functions (e.g., percentage of soft deadlines satisfied), we need an approximate execution profile/static schedule.

Combining function (β). These functions take the transformed results for all child nodes, or pairs of child nodes and return a single result. Most often, the combining function is a simple binary associative operator, or such an operator followed by a simple unary operator such as reciprocal or negation.

Constraints. Are system goals whose failure will result in the rejection of a proposed design. Constraints typically arise from physical restrictions on the underlying application, or on the partially-specified platform or design, but can be hard user requirements on acceptable designs

Cost function (C). A mathematical expression that measures and assigns values to attributes and objectives.

Cost function instance (I). A particular cost function evaluation. For example, a cost function can be evaluated for the entire system, or for a single (not necessarily connected or autonomous) subsystem, or for multiple instantiations of subsystems. There is also a trivial instance for each cost function, which returns a constant, typically the identity for a combining operation – this allows use of a static hierarchy above instance level, using the trivial instance for unused cost functions.

Data transformation function ($\mathcal{D}$). The given sequence of functions $(\mathcal{I}, \mathcal{T}, \mathcal{S}, \mathcal{F}, \phi, w, \beta))$ applications can be considered to form one single transformation. We refer to this sequence as the *Data transformation function* .

Fusion function ($\mathcal{F}$). Cost functions and different lower level objectives may be expressed in incommensurable units. A fusion function creates commensurability . Some existing fusion functions include: normalization, conversion to money or time, and tradeoff functions.

Identity function ($\mathcal{I}$). A function that returns the same value or expression passed to it.

Objective (o). An objective is a description of non-functional system properties. Objectives can be qualitatively decomposed into smaller scoped objectives and finally into attributes. Objectives represent behavior desired for the application; objectives frequently can be satisfied to a greater or lesser degree. Violation of objectives does not necessarily result in the rejection of a system. However, their relative degree of satisfaction affects in the acceptability of a proposed design.

Objective hierarchy (X). A Directed Acyclic Graph (DAG) that describes the decomposition of objectives. There is a predefined root objective, *system*,

which in a given application comprises the top-level *system design factors*, such as performance, real-time, security, and so on. These may be refined into smaller scoped objectives represented as children in the DAG. For example, a node *performance* , may have as children *response time*, *throughput*, and *load balancing*. These can in turn be refined.

Scale type function ($\mathcal{T}$). These functions coerce values of different scale type to a singular scale. Coercion may proceed in one of two directions, up or down. Coercing down is easy but information is lost. Coercing up is harder because we assume information which may not be true about the measure.

Scaling function ($\mathcal{S}$). A scaling function transforms values of different cost functions to comparable ranges.

System characteristics (s). The description of the systems hardware and software models.

Top-level non-functional objectives (o). Constitute the overall goals of a system such as performance, real-time, security, and so on. These may be refined into smaller scoped objectives. For example, *performance* may include issues of response time, throughput, and load balancing.

Weight function (w). Weight functions reflect the relative importance of the information at different children. These are typically provided by the user or design elements for the upper levels of the hierarchy (user level) and by a function developer for the lower levels.

REFERENCES

1. H. Ali and H. El-Rewini, "Task Allocation in Distributed Systems: A Split Graph Model," *Journal of Combinatorial Mathematics and Combinatorial Computing*, Vol. 14, pp. 15-32, October 1993.

2. C.C. Amaro, S.K. Baruah, and A.D. Stoyen, "Inter-Completion Time Scheduling (ICTS): Non-preemptive scheduling to maximize the minimum inter-completion time," in *Proceedings of IEEE Fourth International Conference on Engineering of Complex Computer Systems*, Monterey, California, August 1998.

3. C.C. Amaro, S.K. Baruah, A.D. Stoyen, and W.A. Halang, "Non-preemptive scheduling to maximize the minimum global inter-completion time (MGICT)," in *Proceedings of the 23nd IFAC/IFIP Workshop on Real-Time Programming*, Shantou, Guandong Province, P.R. China, June 1998.

4. C.C. Amaro, T. J. Marlowe, and A. D. Stoyenko, "Objective Function Synthesis For Complex Real-Time Systems," in *Proc. 21st IFAC/IFIP Workshop on Real-Time Programming*, Gramado - RS, Brazil, November 1996.

5. C.C. Amaro, S.K. Baruah, T.J. Marlowe, and A.D. Stoyenko, "Non-Preemptive Scheduling to Maximize the Minimum Intercompletion Time," to appear in *The Journal of Combinatorial Mathematics and Combinatorial Computing.* Technical Report NJIT/CIS-96-13, Real-Time Computing Laboratory, Department of Computer and Information Science, NJIT, April 1996.

6. C.C. Amaro, T.J. Marlowe, and A.D. Stoyenko, "Objective Function Synthesis: The REAL Resource Allocator Approach," Technical Report NJIT/CIS-95-27, Real-Time Computing Laboratory, Department of Computer and Information Science, NJIT, December 1995.

7. C.C. Amaro, M. Harelick, P. Sinha, A.D. Stoyenko, T.J Marlowe, P. Laplante, A. Silberman, N. Jones, B.C. Cheng, T. Tugcu, "Economics of Resource Allocation," in *Proc. Complex Systems Synthesis and Assessment Engineering Workshop*, Silver Springs, Maryland, pp. 195-201, July 1994.

8. K.J. Arrow, *Social choice and individual values*, 2^{nd} edition, New York: Wiley, 1963.

9. J.E. Bailey and S.W. Pearson, "Development of a Tool For Measuring And Analyzing Computer User Satisfaction," *Management Science*, Vol. 29, No. 5, pp. 530-45, May 1983.

10. S.H. Bokhari, "Dual Processor Scheduling with Dynamic Reassignment," *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 4, pp. 341-49, July 1979.

11. A. Burchard, J. Liebeherr, Y. Oh, and S.H. Son, "New Strategies for Assigning Real-Time Tasks to Multiprocessor Systems," *IEEE Transactions on Computers*, Vol.44, No. 12, pp. 1429-42, December 1995.

12. P. Brucker, M.R. Garey, and D.S. Johnson, " Scheduling Equal-Length Tasks Under Treelike Precedence Constraints to Minimize Maximum Lateness," *Mathematics of Operations Research*, Vol. 2, No. 3, pp. 275-84, August 1977.

13. L. Carroll, "A Tangled Tale," *The Best of Lewis Carroll*, Book Sales, Inc., Secaucus, New Jersey , not dated, Originally appeared in *The Monthly Packet*, beginning in April 1880.

14. C. Carlsson and R. Fuller, "Multiple Criteria Decision Making: The Case For Interdependence," *Pergamon Computers Ops Res.*, Vol. 22, No. 3, pp.251-260, 1995.

15. W.W. Chu, L.J. Holloway, M. Lan, K. Efe, "Task Allocation in Distributed Systems," *IEEE Computer*, pp. 57-69, November 1980.

16. J.L. Cohon, *Multiobjective Programming and Planning*, Academic Press, 1978.

17. S. Davari and S.K. Dhall, "An On Line Algorithm for Real-Time Tasks Allocation," in *Proc. IEEE, Real-Time Systems Symposium*, pp. 194-200, 1986.

18. S. Davari and S.K. Dhall, "On a Periodic Real-Time Task Allocation Problem," in *Proc.* 19^{th} *Hawaii International Conference on Systems Science*, pp. 133-41, 1986.

19. M. Dror, P. Shoval, and A. Yellin, "Multiobjective Linear Programming, Another DSS," *Decision Support Systems*, No. 7, pp. 221-32, 1991.

20. R. deNeufville, J. Stafford, *Systems Analysis for Engineers and Managers*, New York: McGraw-Hill, 1971.

21. K. Efe, "Heuristic Models Of Task Assignment Scheduling in Distributed Systems," *IEEE Computer*, Vol. 15, No. 6, pp. 50-56, June 1982.

22. H. El-Rewini, T. Lewis, and H. Ali, *Task Scheduling in Parallel and Distributed Systems*, Prentice-Hall, 1994.

23. D.G. Feitelson, " A Survey of Scheduling in Multiprogrammed Parallel Systems," IBM Research Report RC 19790 (87657), T. J. Watson Research Center, revised February 1995.

24. R.A. Gagliano, M.D. Fraser, and M.E. Schaefer, "Auction Allocation of Computing Resources," *Communications of the ACM*, Vol. 38, No. 6, pp. 88-99, June 1995.

25. M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, San Francisco: W. H. Freeman and Company, 1979.

26. M. Garey and D. Johnson, "Two-processor scheduling with start-times and deadlines," *SIAM Journal of Computing*, Vol. 6, pp. 416-26, 1977.

27. R. Graham, "Bounds on multiprocessor timing anomalies," *SIAM Journal on Applied Mathematics*, Vol. 17, pp. 416-29, 1969.

28. R. Graham, "Bounds for certain multiprocessing anomalies," *Bell System Technical Journal*, Vol. 45, pp. 1563-81, 1966.

29. R. Gupta and M. Spezialetti, "Busy-Idle Profiles and Compact Task Graphs: Compile-time Support for Interleaved and Overlapped Scheduling of Real-Time Tasks," *IEEE, Real-Time Systems Symposium*, 1994.

30. W.A. Halang and A.D. Stoyenko, *Constructing Predictable Real Time Systems*, Boston-Dordrech-London: Kluwer Academic Publishers, 1991

31. M. Harelick, A.D. Styenko, C.C. Amaro, R. Scherl, "Operator Resources For Large Complex Systems," in *Proc. IEEE International Conference on Engineering of Complex Computer Systems*, pp. 112-115, Montreal, Canada, October 1996.

32. M.S. Harelick, T.J. Marlowe, A.D. Stoyenko, P. Sinha, "A Constraint Function Classification for Complex Systems Development," in *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems*, pp. 286-89, Ft. Lauderdale, Florida, November 1995.

33. J.L. Hellerstein, "Achieving Service Rate Objectives with Decay Usage Scheduling," em IEEE Transactions On Software Engineering, Vol. 19, N0. 8, pp. 813-25, August 1993.

34. D. S. Hochbaum and D. B. Shmoys, "A unified approach to approximation algorithms for bottleneck problems," *Journal of the ACM*, Vol. 33, pp. 533-50, 1986.

35. J.P. Ignizio, *Introduction to linear goal programming*, Sage, Series 7, No. 56, 1985.

36. J. Jehuda, G. Koren, and D.M. Berry, "A Time-Sharing Architecture for Complex Real-Time Systems," in *Proc. IEEE International Conference on Engineering of Complex Computer Systems*, pp. 9-15, Ft. Lauderdale, Florida, November 1995.

37. D.S. Johnson, "Fast algorithms for Bin-packing," *J. Comput. Systems Sci.*, Vol. 8, pp. 272-314, 1974.

38. R.L. Keeney, and H. Raiffa, *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*, New York: John Wiley & Son, 1976.

39. D.H. Krantz, R.D. Luce, P. Suppes, and A. Tversky, "Foundations of Measurement," Vol. I, Academic Press, New York, 1971.

40. R.P.-L. Lee, "Optimal Task and File Assignment in a Distributed Computing Network," Ph.D. Dissertation, Department of Computer Science, University of California, Los Angeles, 1977.

41. T. Lewis and H. El-Rewini, *Introduction to Parallel Computing*, Prentice-Hall, 1992.

42. C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a hard-real-time environment," *Journal of ACM*, Vol. 20, No. 1, pp. 46-61, 1973.

43. V.M. Lo, "Heuristic Algorithms for Task Assignment in Distributed Systems," *IEEE Transactions on Computers*, Vol.37, No. 11, pp. 1384-97, November 1988.

44. R.D. Luce and H. Raiffa, *Games and decisions*, New York: John Wiley & Sons, Inc., 1965.

45. P.R. Ma, E.Y. Lee, M. Tsuchiya, "A Task Allocation Model for Distributed Computing Systems," *IEEE Transactions On Computers*, Vol. c-31, No. 1, pp.41-7, January 1982.

46. T.J. Marlowe, A.D. Stoyenko, P. Laplante, N. Jones, C.C. Amaro, P. Sinha, B.C. Cheng, M. Harelick, "Testing Network Assignment Algorithm with a Structured Workload Generator," in *Proc. Seventh Annual Software Technology Conference*, Salt Lake City, Utah, April, 1995.

47. T.J. Marlowe, A.D. Stoyenko, P. Laplante, R. Daita, C.C. Amaro, C. Nguyen, S. Howell, "Multiple-Goal Objective Functions for Optimization of Task Assignment in Computer Systems," *Elsevier, Control Engineering Practice*, Vol. 4, Iss. 2, February 1996.
Earlier version: *Proc.* 19^{th} *IFAC/IFIP Workshop on Real Time Programming*, Isle of Reichenau, Lake Constance, Germany, June 1994.

48. T.J. Marlowe, A.D. Stoyenko, C. Nguyen, S. Howell, P. Laplante, R. Daita, C.C. Amaro, "Multiple-Stage Dynamic-Programming Heuristic for Assignment and Scheduling in Destination," Computer and Information Science Research Report CIS-93-13, New Jersey Institute of Technology, October 1993.

49. C. McConnell, *Economics*, 9^{th} edition, New York: McGraw-Hill, 1984.

50. K.B. Misra, "An Efficient approach for multiple criteria redundancy optimization problems," *Microelectronics and Reliability*, v 31, n 2-3, pp. 303-321, 1991.

51. B.K. Mohanty and T.A.S. Vijayaraghavan, "A multi-objective programming problem and its equivalent goal programming problem with appropriate priorities and aspiration levels: a fuzzy approach," *Pergamon, Computers And Operations Research*, vol. 22, No. 8, pp. 771-8, October 1995.

52. M. Mollaghasemi and J. Pet-Edwards, *Technical Briefing: Making Multiple-Objective Decision*, Los Alamitos, CA: IEEE Press, 1997.

53. R. Nossal and T.M. Galla, "Solving NP-Complete Problems in Real-Time System Design by Multichromosome Genetic Algorithms", In *Proceedings of the SIGPLAN 1997 Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 68–76. ACM SIGPLAN, June 1997.

54. C.M. Nguyen, S.L. Howell, "Systems Design Factors: The Essential Ingredients of Systems Design," Technical Report NAVSWC TR 92-268, Naval Surface Warfare Center, Silver Spring, Maryland 1992.

55. I. Page, T. Jacob, and E. Chern, "Fast Algorithms for Distributed Resource Allocation," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 2, pp. 188-97, February 1993.

56. J. Pukite, P.R. Pukite, "Complex System Optimization," in *Proc. Complex Systems Engineering Synthesis and Assessment Workshop*, Silver Springs, Maryland, 195-201, July 1994.

57. C.V. Ramamoorthy, K.M. Chandy, and M.J. Gonzalez, Jr., "Optimal Scheduling Strategies in a Multiprocessor System," *IEEE Transactions on Computers*, Vol. C-21, No.2, pp. 137-47, February 1972.

58. J.R. Rao, R.N. Tiwari, and B.K. Mohanty, "A Preference Structure on Aspiration Levels in a Goal Programming Problem –A Fuzzy Approach," *Fuzzy Sets Systems*, Vol. 25, pp. 175-182, 1988.

59. J.R. Rao, R.N. Tiwari, and B.K. Mohanty, "A Method for Finding Numerical Compensation for Fuzzy Multicriteria Decision Problem," *Fuzzy Sets Systems*, Vol. 25, pp. 33-41, 1988.

60. F.S. Roberts, *Discrete mathematical models, with applications to social, biological, and environmental problems*, Englewood Cliffs, New Jersey: Prentice-Hall, 1976.

61. F.S. Roberts, *Measurement Theory with Application to Decisionmaking, Utility, and Social Sciences*, Reading, Massachusetts: Addison-Wesley, 1979.

62. S. Selvakumar, C.S.R. Murthy, "Static task allocation of concurrent programs for distributed systems with processor and resource heterogeneity," *Elsevier, Parallel Computing*, Vol. 20, pp. 835-851, 1994.

63. D.D. Sharma and D.K. Pradham, "Processor Allocation in Hypercube Multicomputers: Fast and Efficient Strategies for Cubic and Noncubic Allocation," *IEEE Transactions on Parallel and Distributed Systems*, Vol.6, No. 10, pp. 1108-1123, October 1995.

64. K. Schwan and C. Gaimon, "Automating Resource Allocation for Multiprocessors," *The Journal of Systems and Software*, no. 9, pp. 51-66, Elsevier Science Publishing Co., Inc., 1989.

65. C.-C. Shen, W.-H. Tsai, "A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion," *IEEE Transactions On Computers*, Vol. c-34, no. 3, March 1985.

66. C. Shen, K. Ramamritham, John A. Stankovic, "Resource Reclaiming in Real-Time," in *Proc. Real-Time Systems Symposium*, Lake Buena Vista, Fl., USA, Dec 5-7 1990, IEEE 1991.

67. W.S. Shin and A. Ravindran, "An Interactive Method for Multi-Objective Mathematical Programming Problems," *Journal of Optimization Theory and Applications*, Vol. 68, No. 3, pp. 539-61, March 1991.

68. B. Simons, "A fast algorithm for single processor scheduling," in *Proceedings of the Ninteenth Annual Symposium on Foundations of Computer Science*, 1978.

69. B. Sprunt, J. Lehoczky, and L. Sha, "Exploiting Unused Periodic Time For Aperiodic Service Using The Extended Priority Exchange Algorithm," in *Proc. IEEE, Real-Time Systems Symposium*, pp. 251-258, July 1988.

70. M. Srinivas and L.M. Patnaik, "Genetic Algorithms: A Survey", *IEEE Computer*, pages 17–26, June 1994.

71. J.A. Stankovic, "Editorial: Resource Allocation in Real-Time Systems", *Real-Time Systems*, v5, n2-3, pp. i-vi, May 1993.

72. S.S. Stevens, "On the Theory of Scales of Measurement," *Science*, 103, pp.677-680, 1946.

73. S.S. Stevens, "Mathematics, Measurement and Psychophysics," in *Handbook of Experimental Psychology*, John Wiley & Son, Inc, New York, pp. 1-49, 1951.

74. I. Stoica, "A Unified Approach for Multiple Shared-Resource Allocation in Distributed Systems," Ph.D. Proposal, Department of Computer Science, Old Dominion University, Norfolk, Virginia, 1995.

75. H.S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, pp. 85-93, January 1977.

76. H.S. Stone and S.H. Bokhari, "Control of Distributed Processes," *IEEE Computer*, pp. 97-106, July 1978.

77. A.D. Stoyenko, "A Real-Time Language with A Schedulability Analyzer," Ph.D. Thesis, Department of Computer Science, University of Toronto, 1987.

78. A.D. Stoyenko, J. Bosch, M. Akşit and T.J. Marlowe, "Load Balanced Mapping of Distributed Objects to Minimize Network Communication," *Journal of Parallel and Distributed Processing*, to appear 1996.

79. A.D. Stoyenko, T.J. Marlowe, M. Younis, A. Ganesh, C.C. Amaro, P. Laplante, A. Silberman, P. Sinha, "Towards A Language Paradigm for Construction and Development of Complex Computer Systems," in *Proceeding of IEEE workshop on Composability of Fault-Resilient Real Time Systems*, San Juan, Puerto Rico, Dec. 1994.

80. A.D. Stoyenko, L.R. Welch, P. Laplante, T.J. Marlowe, C.C. Amaro, B.C. Cheng, A.K. Ganesh, M. Harelick, X. Jin, M. Younis, G. Yu,"A Platform for Complex Real-Time Applications," in *Proc. Complex Systems Engineering Synthesis and Assessment Workshop*, Silver Springs, Maryland, 152-159, June 1993.

81. A.D. Stoyenko, T.J. Marlowe, "Polynomial-Time Transformations and Schedulability Analysis of Parallel Real-Time Programs with Restricted Resource Contention," *Journal of Real-Time Systems*, Vol 4, No. 4, pp. 307-29, November 1992.

82. A.D. Stoyenko, V.C. Hamacher, R.C. Holt, "Analyzing Hard-Real-Time Programs for Guaranteed Schedulability," *IEEE Transactions on Software Engineering*, pp. 737-50, SE-17, No. 8, August 1991.

83. P. Vincke, *Multicriteria Decision-Aid*, New York, NY: John Wiley & Sons, Inc., 1992.

84. C.M. Woodside, G.G. Monforton, "Fast Allocation of Processes in Distributed and Parallel Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, No. 2, pp. 164-74, February 1993.

85. M.F. Younis, "Safe code transformations for speculative execution in real-time systems," Ph.D. Thesis, Department of Computer & Information Science, New Jersey Institute of Technology, Newark, New Jersey 1996.

86. M. Zeleny, *Multiple Criteria Decision Making*, McGraw-Hill, 1982.

87. W. Stadler, "Applications of Multicriterion Optimization in Engineering and The Sciences," *MCDM: Past Decades and Future Trends*, M. Zeleny, Greenwich, Connecticut: JAI PRESS INC., 1984.

88. H.J. Zimmerman, "Fuzzy Programming and Linear Programming with Several Objective Functions," *Fuzzy Sets Systems*, Vol. 1, pp. 45-43, 1978.