On Quality of Service Management

Chen Lee

August 1999

CMU-CS-99-165

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Engineering

Thesis Committee:
Daniel Siewiorek, Chair
Ragunathan Rajkumar
John Lehoczky
Christos Faloutsos

This work was supported in part by the Defense Advanced Research Projects Agency under agreements N66001-97-C-8527 and E30602-97-2-0287. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied of the Defense Advanced Research Projects Agency or the U.S. government.

Dedicated to my mother

Keywords: QoS management and optimization, QoS index and utility model, QoS tradeoff, resource tradeoff, real-time, multi-media, policy-driven and adaptive systems

Abstract

A quality of service (QoS) management framework for systems is presented that satisfies application needs along multiple dimensions such as timeliness, reliability, cryptographic security and other application-specific quality requirements. In this model, end users' quality preferences are taken into account when system resources are apportioned across multiple applications such that the net system utility accrued to the end-users is maximized. The framework facilitates QoS tradeoff through a semantically rich (in terms of expressiveness and customizability) QoS specification interface that enables the end users to give guidance on the qualities they care about and the tradeoffs they are willing to make under potential resource shortages. The interface also allows the user or system administrator to define fine-grained service requests easily for multi-dimensional complex QoS provisioning. Furthermore, by introducing the abstraction of *Quality Index*, which maps qualities to indices in a uniform way, and by the mathematical modeling of *QoS Tradeoff* and *Resource Tradeoff*, we transform the QoS management problem into a combinatorial optimization which ultimately enables us to quantitatively measure QoS, and to analytically plan and allocate resources.

A series of optimization algorithms is developed that tackle the QoS management problem which is provably NP-hard. The first set of algorithms treats the problem of maximizing system utility by allocating a *single* finite resource to satisfy the QoS requirements of *multiple* applications along *multiple* QoS dimensions. Two near-optimal algorithms are developed to solve this problem. The first yields an allocation within a known distance from the optimal solution, and the second yields an allocation whose distance bound from the optimal solution can be *explicitly controlled* by a QoS manager. We compare the run-times of these near-optimal algorithms and their solution quality relative to the optimal allocation, which in turn is computed using

dynamic programming.

The second set of algorithms deals with apportioning *multiple* finite resources to satisfy the QoS needs of *multiple* applications along *multiple* QoS dimensions. Three strategies are evaluated and compared First, dynamic programming and integer programming with branch-and-bound compute optimal solutions to this problem but exhibit very high running times. Then the integer programming approach is adapt to yield near-optimal results with faster running times. Finally, an approximation algorithm based on an extended *local search* technique is presented that is less than a few percent from the optimal solution but which is more than two orders of magnitude faster than the optimal scheme of dynamic programming. Perhaps more significantly, the local search technique turns out to be very scalable and robust as the number of resources under management increases. These detailed evaluations provide practical insight into which of these algorithms can be used online in real-time systems.

Acknowledgements

First and foremost I would like to thank my advisor Dan Siewiorek for his guidance and support. I cannot imagine a better supervisory style than his. Dan gives student complete freedom on defining and pursuing research goal and possesses broad knowledge on almost every aspects of computer system research so he could handily provide guidance whenever I need. Thanks also goes to my co-advisor Raj Rajkumar, and to John Lehoczky and John Hooker who although they are not my advisors, have provided me much advice. Raj led me into the real-time research, offered me the opportunity to hack deep into the kernel of a real-time operating system. Raj's expertise on real-time systems and theory are an invaluable source of guidance. He has a way of asking the questions that often opens up new angles on challenging research issues. John Lehoczky and John Hooker's deep understanding and research achievement of related research problems steered me towards the effective courses of research, and they also have been a constant source of ideas and inspiration.

I would like to thank the other members of my thesis committee, Christos Faloutsos and Nelu Mihai for giving very useful comments on my thesis proposal and thesis draft.

Especially I want to thank Morten Welinder and Andrzej Filinski. They have given me invaluable and tremendous help and support through the course of my graduate studies. I have a deep debt of gratitude to them.

Thanks also go to Jeannette Wing who always kindly reminded and prodded me to finish up my thesis work, to Julia Deems for providing help and advice with English writing, and to my officemates and coworkers as well as great friends Cliff Mercer and Jim Zelenka. They made my research and system hacking experience a cheerful one.

I am delighted to have had the company of so many wonderful friends and schoolmates during almost six years of stay in CMU. Here I will just try to list some that I have had relatively more frequent contact with recently: Ali, Arup, Bennet, Bryan, David, Eduardo, Eli, Erik, Girija, Jie, Jim, John, Jonathan, Jorjeta, Khalil, Lemonte, Mark, Mei-Yuh, Peter, Qi, Rahul, Robert, Rony, Srihari, Tom, Vince, Xudong, Yeyi, and great volleyball teammates as well as friends Corey, Darrell, Dennis, Dushyanth, and Marius.

I also benefited from many and varied interactions with faculty members in different research areas. Jeannette Wing, John Shen, Pradeep Khosla, Robert White, John Reynolds, Hide Takuda, Mahadev Satya, Garth Gibson, David Nagle, Hui Zhang, John Wilkes, David Johnson, Peter Steenkist, Bruce Maggs and Tom Gross all showed me how to conduct system research from various angles.

I also owe thanks to all the marvelous people at Carnegie Mellon who keep things running smoothly and who cheerfully and efficiently make all the administrative hassles go away, especially Laura Forsythe, Joan Maddamma, Lynn Philibin, Elain Lawrence, Marian D'Amico, Catherine Copetas, Karen Olack, Patty Mackiewicz and everyone on the CS facilities staff.

Thanks also go to Ying Li and Jiabin Pan of the Pittsburgh Ballet. Everyone needs a life outside of science and research.

Finally, I would like to thank my family, especially my late mother, an electrical engineer, and my father, an economist. They always have had faith in me and believed that human creativity knows no limitation and that I could do anything. I will always remember those countless evenings my mother patiently answered my curious questions when she was building the first television set for my family. She was the one eventually got me interested in computer science. This thesis is dedicated to her.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Quality of Service (QoS) control is receiving widespread attention in computer network and real-time multimedia system research as well as commercial markets. Typically, service characteristics in existing multimedia and networked systems are fixed when systems are built, therefore they often do not give users any real influence over the QoS they can obtain. On the other hand, multimedia applications and their users can differ enormously in their requirements for service quality and the resources available to them at the time of application use. Therefore, there is an increasing need for customizable services that can be tailored for the end users' specific requirements.

In the meantime, new and improved systems such as the one proposed by the Amaranth project at Carnegie Mellon University [59] are placing more and more complex demands on the quality of service that are reflected in multiple criteria over multiple quality dimensions. These QoS requirements can be objective in some aspects and subjective in others. Moreover, because of the manifold and subjective nature of user quality demands, it is very hard to measure whether the provided quality fulfills the stated demands without guidance and input from end clients.

One issue is *QoS Tradeoff* where a user of an application might want to emphasize certain aspects of quality, but not necessarily others. Users might tolerate different levels of service, or could be satisfied with different quality combination choices, but the available system resources might only be able to accommodate some choices but not others. In situations where a user is able to identify a number of desirable objective qualities and rate them (subjectively), the system should be able to reconcile different demands to maximize the user's preference and to make the most effective use of the system. So it is important for a system to provide a large variety of service qualities and to accommodate specific user quality requirements and deliver as good service as it can from the users' perspective.

An issue related to QoS tradeoff is *Resource Tradeoff*. In this case, the tradeoff refers to reconciling or balancing competing resource demands. Resource tradeoff is often transparent to the user but can be of great help in accommodating user requirements including QoS tradeoff, especially when the availability of several different resources is not balanced. It arises when an application is able to use an excess of one resource, say CPU power or cache, to lower its demands on another, say network bandwidth, while maintaining the same, or similar, level of QoS. For example:

- Video conferencing systems often use compression schemes that are effective, but computationally intensive, to trade CPU time for network bandwidth. If the bandwidth is congested on some intermediate links (which is often the case), this benefits the system as a whole.

- In the case of a mobile client with limited CPU and memory capacity but sufficient link speed with a nearby intermediate powerful server, computationally expensive speech recognition, silence detection and cancellation, and video compression could be carried out on the nearby server.

- For proxy servers which act as transcoders/transceivers besides caching data, the proxy servers can distill data for low bandwidth clients (when both server and client have fast CPU, memory and disk bandwidth, but the network link speed in between is limited).

Consider a video-conferencing application. The audio streams in this application have multiple QoS characteristics: the sampling rate of the audio data, the resolution (number of bits) of each audio sample, and the end-to-end latency of the audio stream. Similarly, the video streams must deal with multiple QoS dimensions: the video frame rate, the size of the video window, the number of bits per pixel and so on. Given an operating point along each of these QoS dimensions, the application requires processing and network bandwidth resources at the application end-hosts and all intermediate links that the audio/video streams traverse.

We envision an environment where many such time-critical, real-time and non-real-time applications each with multiple QoS dimensions co-exist in a system with a set of finite resources under management. During loaded periods, the system may not have sufficient resources to deliver the maximum quality possible to every application along each of its QoS dimensions. Hence, decisions must be made by the underlying resource manager to apportion available resources to these applications such that a global objective is maximized.

1.2 Related Work

Research on Quality of Service for multimedia applications has gained significant momentum over the last few years. Much research has been conducted on the end-system or end-to-end architectures for QoS support [12, 17, 6, 39, 40, 4, 36, 24, 8, 60, 48, 26, 21], and much more is on link, network and transport layer ([62, 61, 11, 52, 55, 33, 54, 5] to name a few). Most of this research has been focused on low-level system mechanisms. The authors consider and work on such parameters as period, buffer size, jitter, bandwidth and so on. While these issues are important factors for QoS control, we believe that they are not sufficient for the ultimate end-users who experience the resulting QoS.

Research on adaptive QoS control [57, 56, 35, 29, 41] brings us a step closer to the QoS support from a user's perspective by providing a mechanism in an application to accommodate potential dynamic changes in the operating environment. But these mechanisms are still mainly system-oriented in that a user has limited influence over the quality of the service to be delivered or adapted.

In coping with the shortage of QoS support from an end-user point of view, we proposed a basic framework [25, 30, 46] that enables the end users to give guidance on the qualities they care about and the tradeoffs they are willing to make under potential resource constraints. Working from the user's perspective and maximizing the user perceived quality or utility has also been addressed in [19, 2, 3]. In [19], a user-centric approach is taken, where a user's preferences are considered for application runtime behavior control and resource allocation planning. Example preferences include statements that a video-phone call should pause a movie unless it's being recorded and that video should be degraded before audio when all desired resources are not available. These are useful hints for high-level QoS control and resource planning, but are inadequate for quantitatively measuring QoS, or analytically planning and allocating resources.

The notion of using utility functions to represent varying satisfaction with QoS changes is certainly not new. Jensen et al. [18] and Locke [32] are perhaps among the first to study "value functions" to represent the benefit of different completion times of a task. Their value function model is a utility function along the latency quality dimension of real-time tasks. Our model can be viewed as extending this notion to include quality dimensions other than timeliness. The *imprecise computation* model proposed by Liu et al. [31] considered the problem of optimally allocating CPU cycles to applications which must satisfy minimum CPU requirements, but can produce better results with additional CPU cycles. The frequency of each application remains constant, while the computation time per instance of an application can be varied. The results were generally assumed to improve linearly with additional resources. The model described in this thesis can be considered to be a generalization of the above model from single quality dimension to multiple QoS dimension optimization with potential QoS tradeoffs, and from single resource to multiple resource allocation with potential resource tradeoffs. Applying utility model for QoS control is also studied in [2, 22]. In [2], the authors propose a mechanism for QoS (re)negotiation as a way to ensure graceful degradation. The authors suggest that a user should be able to express, in his/her service requests, the spectrum of QoS levels the user can accept from the provider, as well as the perceived utility of receiving service at each of these levels. A similar approach is taken in [22]. But neither of the authors address the resource tradeoff problem. Also, neither has developed the specification method and mechanism to facilitate utility data acquisition. We will return to [22] in Chapter 6 for more related work and comparisons.

Interesting research have been conducted in [3] and [37, 20]. In [3], the authors present a framework for the construction of network-aware applications. The basic idea is to allow an application to adapt to its network environment, e.g. by trading off the volume (and with it the quality) of the data to be transfered and the time needed for the transfer. Their mechanism coincides with one of our schemes for implementing the resource tradeoff ($r \models_i q$). The model defined in [30] can be considered a generalization of [3]. The authors in [37] thoroughly explored building an analytical framework for adaptive terminal IO service, which can be viewed as good mechanism for resource tradeoff. We believe that their work can operate well together with the work presented in this thesis.

1.3 Approach

Figure 1.1 gives a pictorial view of our QoS management optimization system.

The QoS architecture [27] we consider consists of a QoS specification interface, a quality tradeoff specification model, and a unified QoS-based admission control and resource allocation model. The QoS specification interface allows multiple QoS requirements to be specified, and is semantically rich both in terms of expressiveness and customizability. Note that our QoS management framework is translucent in a sense that some aspects are made visible to the end-users so that users can control the delivered QoS parameters, while at the same time hiding how the requested delivery is accomplished. In the model, end users' quality preferences are elicited when system resources are apportioned across multiple applications such that the net utility that accrues to the end-users is maximized. Specifically, the QoS tradeoff specification interface allows applications and users to assign values (utilities) to different levels of service that a system can provide. A QoS resource manager, taking QoS profiles and resource profiles of arriving applications as its inputs and exploring fully the QoS tradeoffs and resource tradeoffs, makes resource allocations to these applications so as to maximize the global utility derived by these systems. Finally, by introducing the abstraction of *Quality Index*, which maps qualities to indices in a uniform way, and by the mathematical modelling of *QoS Tradeoff* and *Resource Tradeoff*, we transform the QoS management problem into combinatorial optimization which ultimately enables us to quantitatively measure QoS, and to analytically plan and allocate resources.

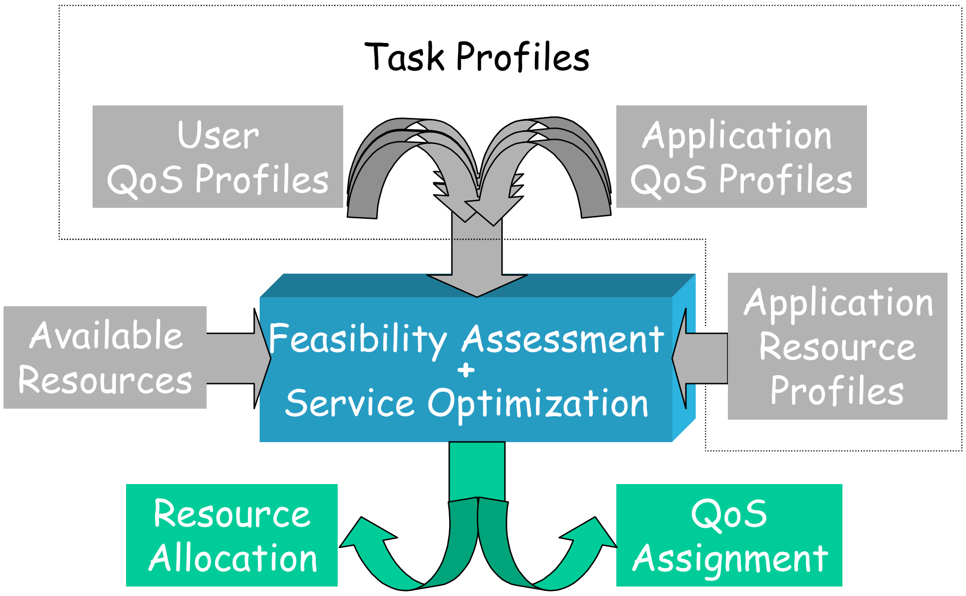

Figure 1.1: Input and Output of the QoS Management Optimization Module

1.4 Organization of the Dissertation

The rest of the dissertation is organized as follows: In Chapter 2 we formally introduce our QoS management model. We start the chapter with example quality dimensions. We then introduce the abstraction of *Quality Index*. Then *QoS Tradeoff* and *Resource Tradeoff* are formally modeled. We conclude this chapter by formulating our QoS management problem and demontrating its power in terms of generality and expressiveness. In Chapter 3 the user specification interface and mechanisms for QoS specification acquisition are addressed. In Chapter 4 we clasify our QoS management problem and discuss the design principles for the corresponding algorithms. In Chapter 5 and Chapter 6 we presents our algorithms for single resource multiple QoS dimension and multiple resource multiple QoS dimension problem respectively, with theoretical measure of their performances. Experimental performance evaluation is conducted in Chapter 7. Finally, we draw conclusions and discuss future work in Chapter 8.

Chapter 2

Quality Index and QoS Modeling

In this chapter, we are going to formalize our QoS management problem. First, however, we need to regularize the problem using a concept we call *quality index.* This is a mapping from qualities to indices in uniform way. Using quality index as basis, we transform the QoS management problem into a optimization problem which utltimately enables us to quantitatively measure QoS, and to analytically plan and allocate resources.

2.1 Quality Dimensions

Consider a video-streaming system which deals with real-time audio and video data streams being encrypted and transmitted across potentially unreliable networks. In this context, we consider the following example quality dimensions:

- Cryptographic Security (encryption key-length)
  - 0(off), 56, 64, 128
- Data Delivery Reliability, which could be
  - maximum packet loss: measured in percentage
  - expected packet loss: measured in percentage
  - packet loss occurrence: measured as the probability that one or more packets are lost over the length of the session.

- Video Related Quality
  - picture format[1]: SQCIF, QCIF, CIF, 4CIF, 16CIF
  - color depth(bits): 1, 3, 8, 16, 24, ...
    black/white, grey scale to high color
  - video timeliness — frame rate(fps): 1, 2, ..., 30
    low rate animation to high motion picture video
- Audio Related Quality
  - sampling rate(kHz): 8, 16, 24, 44, ...
    AM, FM, CD quality to higher fidelity audio
  - sample bit(bits): 8, 16, ...
  - audio timeliness, or end-to-end delay(ms)
    ..., 25, 50, 75, 100, ...

For the sake of this example we assume that cryptographic key lengths are indicative of the level-of-security provided. This is not generally true. For example, a 56-bit DES encryption is a vast improvement over a 128-bit RSA encryption. But as we shall see shortly, all we require is the ability to place an ordering on the choices.

Notice that all quality dimensions have a discrete set of options. For some, picture format for instance, this is natural; for others, audio timeliness for example, we require that a discrete set of choices be selected. It is not explicit from the example above, but we also require that the set of choices be finite.

2.2 Quality Index

As the example above shows, quality dimensions can differ radically from each other in nature. Certain quality dimensions, such as frame rate, can have a regular quality specification, while others, such as picture format, color depth and end-to-end delay, are non-numeric, non-uniform, or in non-increasing order. For example, there is color

[1]The choices listed here come from [16] [51]. Other standards, such as MPEG could have been used instead.

depth where high numbers corresponds to high quality. Then there is audio timeliness, where high numbers means low quality.

In order to present a coherent formalization of the QoS management problem, we need to regularize the quality dimensions. To this end, we introduce the notion of *Quality Index,* which is a mapping between qualities to integers ("indices") starting with one in such a way that higher indices correspond to higher quality.

For the video application (assume it is T_i in the system) introduced ealier, we would have the following quality indices.

PICTURE FORMAT: Assume it uses the H263 [16] standard format

Format:	SQCIF	QCIF	CIF	4CIF	16CIF
Quality Index:	1	2	3	4	5

The corresponding Quality Index is therefore $Q_{i1} = \{1, 2, 3, 4, 5\}$.

COLOR DEPTH: Assume that T_i has 1, 3, 8, 16, and 24 bit color depths available for the user to choose.

Depth:	1	3	8	16	24
Quality Index:	1	2	3	4	5

Therefore $Q_{i2} = \{1, 2, 3, 4, 5\}$.

FRAME RATE: T_i allows frame rates ranging from 1 fps to 30 fps in steps of 1 fps. These will map directly onto $Q_{i3} = \{1, 2, \ldots, 30\}$.

Rate (fps):	1	2	...	30
Quality Index:	1	2	...	30

ENCRYPTION KEY LENGTH: For T_i, encryption will be either on with 56-bit encryption or off[2]. Therefore we have $Q_{i4} = \{1, 2\}$.

Key length:	(none)	56-bit
Quality Index:	1	2

AUDIO SAMPLING RATE: Assume T_i provides audio sampling rates from AM-quality (8 kHz) to CD-quality (44 kHz).

[2]For the simplicity of illustration, we reduced the levels of encryption to 2 from the 4 in previous example in 2.1. We also omited the quality dimensions related with data delivery reliability here.

Sampling rate (kHz):	8	16	24	44
Quality Index:	1	2	3	4

Thus we have $Q_{i5} = \{1, 2, 3, 4\}$.

AUDIO BIT COUNT: Assume that T_i provides only two sampling sizes, 8 bits and 16 bits.

Bit count:	8	16
Quality Index:	1	2

Therefore $Q_{i6} = \{1, 2\}$.

END-TO-END DELAY: Assume that end-to-end delays ranging from 125 ms to 25 ms in steps of 25 ms. Since high numbers for end-to-end delay are worse than low numbers, high numbers are mapped to low indices in the set $Q_{i7} = \{1, 2, \ldots, 5\}$.

Delay (ms):	125	100	...	25
Quality Index:	1	2	...	5

Quality Index Hence *Quality Index* is essentially a bijective function between a task's dimensional quality space and natual numbers

$$f_{ij} : Q_{ij} \to \{1, 2, \ldots, |Q_{ij}|\}$$

that provides a uniform and consistent ordering of dimensional quality space. That is, if q_1 is "better than" q_2, then $f_{ij}(q_1) > f_{ij}(q_2)$. Since f_{ij} is bijective, it has an inverse $f_{ij}^{-1} = \{(y, x) \mid (x, y) \in f\}$, i.e. $f_{ij}^{-1} : \{1, 2, \ldots, |Q_{ij}|\} \to Q_{ij}$.

From now on, we will mostly use Q_{ij} and q_{ij} to represent their corresponding f_{ij}–indexed quality sets and quality points, except in occasional cases. This should not cause any confusion as the context clearly determines whether the original quality specification or index value is under consideration.

Note that this abstraction forms the basis of our quantitative and analytic approach for QoS management, including the QoS based resource allocation.

2.3 QoS Tradeoff and Application Utility

By *QoS tradeoff* or *quality tradeoff* we mean an assignment of value to each possible quality point. This valuation is central to our model and it allows the management system to take the task's and the user's preferences into account when allocating resources. The function that describes a task's valuation is known as the application's *utility function*, $u_i : Q_i \rightarrow \mathbb{R}$ where Q_i represents the set of all possible quality points, and $\mathbb{R}$ is the set of real numbers.

In a system where resources are in high demand and in which not every task can be allocated the resources it wants, tasks can benefit from QoS tradeoff. In order to see this, consider first a task in a system that does not have QoS tradeoff. If a resource used by this task gets over-subscribed, the system will have to either reject the task, or lower the assigned resource and thus quality and utility at its own discretion.

With QoS tradeoff, however, if the system cannot fulfill the resouce demands for a particular quality mode, it can use the QoS tradeoff information to make smart decision on alternative and perhaps equally desired mode. If the resource usages of the equally desired choices cannot be satisfied, the system can lower the service quality in a way that affects the user minimally. As a result, the task might be able to run at an acceptable, if sub-optimal, level of service. Since quality is at least partially subjective, it is important that the user be allowed to influence the QoS tradeoffs. Therefore it is to the user's significant advantage for a system to provide the capability and interface that allows the user to make implicit or explicit quality tradeoffs. We will consider user interfaces implications in Chapter 3.

With QoS tradeoff, our QoS management optimization engine will work most effectively to help each task achieve as high level of quality as possible, subject to resource constraints and the management policy deployed in the system.

The benefits of QoS tradeoff become even larger in a dynamic environment, where resources, processing power and the link speed, on or between the end and intermediate nodes might dynamically change, and an application's resource allocation fluctuates during the course of operation.

2.4 Resource Tradeoff

The more resource an application is given, the better quality it can provide. With this in mind, one might naively think that the relationship between resource and quality could be described as a function.

Unfortunately, this does not work in the context of multiple resources and quality dimensions. This is because an application can choose between two or more algorithms that achieve the same quality but using different resources. Consider data transmission where two different compression algorithms, A_1 and A_2 are available to use. Assume that A_1 has a relatively low compression rate, but is computationally cheap, whereas A_2 has a high compression rate, but is computationally more expensive. To the user, the end result after decoding will be the same no matter what algorithm is in use. But using A_2 results in more CPU processing power and less network bandwidth compared to that of A_1.

$$
\begin{tikzcd}
 & r_{A_1} = \langle r_1, \cdots, r_m \rangle \\
q = \langle q_1, \cdots, q_d \rangle \arrow[ur, "A_1"] \arrow[dr, "A_2"'] & \vdots \\
 & r_{A_2} = \langle r'_1, \cdots, r'_m \rangle
\end{tikzcd}
$$

This example shows that one quality point can correspond to multiple resource usage points. Thus we cannot describe the situation as a function from quality space to resource space.

Likewise, given a resource allocation, an application can use this to improve quality along one of several quality dimensions, which yields different quality results. For example, in a video conferencing session with limited network capacity, the bandwidth can be used primarily for video to improve the picture quality, or used primarily for audio thereby increasing the sound quality.

$$
\begin{tikzcd}
 & q_{A_1} = \langle q_1, \cdots, q_d \rangle \\
r = \langle r_1, \cdots, r_m \rangle \arrow[ur, "A_1"] \arrow[dr, "A_2"'] & \vdots \\
 & q_{A_2} = \langle q'_1, \cdots, q'_d \rangle
\end{tikzcd}
$$

This example shows that one resource point can correspond to multiple quality points. Therefore we cannot expected to have a function from resource space to quality space that describes the quality in terms of resource allocation. Instead, only a scatter for R and Q can be drawn.

Consequently, only a relation, but not a function, can be defined between Q_i and R. We elect to model the relationship between the resource and quality in the most general fashion, that is as a mathmatical relation:

$$r \models_i q$$

A resource choice, $r \in R$, and a quality point, $q \in Q_i$, are in relation if task T_i can achieve quality q using resource r, but not less.

Note that both R and Q_i have partial orderings which $\models_i$ must respect, i.e., more resource must not lead to lower quality. That is, if $r_1 \models_i q_1$, $r_2 \models_i q_2$, and $r_1 > r_2$, then we must have $q_1 \not< q_2$.

This requirement ensures that utility is non-decreasing with respect to resources. In other words, more resources should not lead to reduced quality (and thus utility), which is reasonable and natural.

2.5 Problem Formulation

In this section, we are going to formally describe the QoS management problem including the QoS based resouce allocation. We will formalize it as a optimization problem.

Consider a system with multiple independent applications and multiple resources. Each application, with its own quality-of-service requirements, contends with others for finite system resources. Let the following be given

$T_1, T_2, \ldots, T_n$ — tasks (or applications)
$R_1, R_2, \ldots, R_m$ — shared system resources
$Q_{i1}, Q_{i2}, \ldots, Q_{id_i}$ — QoS dimensions for task T_i

Each R_i is a set of non-negative values representing the possible allocation choices of the ith shared resource. The set of possible resource vectors, denoted as R, is given by $R = R_1 \times \cdots \times R_m$. The available amount of each shared resource is finite, so we also have $r^{\max} = \langle r_1^{\max}, \ldots, r_m^{\max} \rangle$.

Similarly, each Q_{ij} is a finite set of quality choices for the ith task's jth QoS dimension, and we define the set of possible quality vectors for task T_i by $Q_i = Q_{i1} \times \cdots \times Q_{id_i}$.

2.5.1 Task Profile

Associated with each T_i is a *task profile,* which is a characterization of T_i on quality and resource requirement. It consists of an *application profile* and a *user profile.* An application profile comes from an application designer, while a user profile provides user-specific quality requirements associated with each session.

A user can either instantiate the quality attributes of the default application profile, by selecting one of many templates supplied with the application, or the user can supply their own reward functions with respect to different levels of qualities.

2.5.1.1 Application Profile

An *Application Profile* consists of a *QoS Profile* and a *Resource Profile.*

QoS Profile A *QoS Profile* consists of

- Quality Indices — Q_{ij}, $1 \leq j \leq d_i$.
- Quality Space — $Q_i = Q_{i1} \times \cdots \times Q_{id_i}$.
- *Application Utility* — a QoS or rate of service measure

$$u_i : Q_i \to \mathbb{R}$$

  which for each quality point specifies a utility in the form of a non-negative number.

Resource Profile A *Resource Profile* for T_i is a description of the application's resource usage at different levels of quality. Due to resource tradeoff and quality tradeoff as described earlier, we cannot expect this to be a function (the scatter plot in Figure 2.1, which depicts a possible relation between resource and quality, might help us visualize this). Instead we use the relation $r \models_i q$.

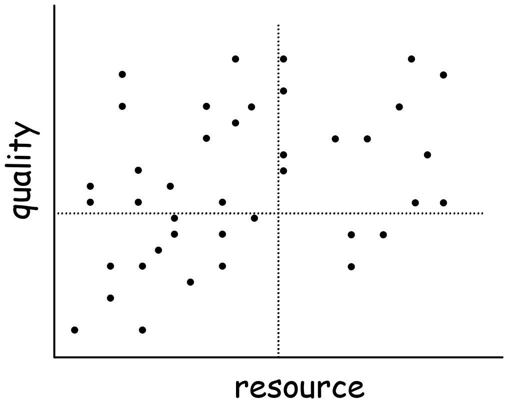

Figure 2.1: Scatter of Resource and Quality

2.5.1.2 User Profile

A *User Profile* is an instantiation of the application profile with perhaps some additional QoS requirement.

- The QoS profile part of application profile provides a template which a user can instantiate to create a user profile. A user can also supply his/her own QoS profile which supersedes those provided by the application.

- **QoS Constraint:** is the minimum QoS requirement specification

$$q_i^{\min} = \langle q_{i1}^{\min}, q_{i2}^{\min}, \ldots, q_{id_i}^{\min} \rangle.$$

  The semantics of $q_i^{\min}$ is that if the minimum requirements cannot be satisfied, then the application cannot run or the user prefers not to run T_i at all.

- **Saturation point:** A user may explicitly specify a cap, or *saturation point,* $q_i^{\max}$, on its quality requirement to indicate that further improvements beyond it are not likely to be perceived or appreciated. Similar to the discussion of $q^{\min}$ above, the maximum quality constraint could be handled by setting

$$u_i(q) := u_i(q_i^{\max}) \text{ for all } q > q_i^{\max}.$$

  To simplify algorithm descriptions, we will not explicitly use this aspect in the algorithms presented later in this thesis.

In general, **Task Profile** will be used to represent the effect of using an application profile that has been instantiated by the user profile. Occasionally we will not distinguish the two sources and use *task profile* for *application profile* and *user profile* as well.

For the overall system, with multiple applications possibly requiring multiple resources, system utility can be introduced and defined.

2.5.2 System Utility

Each task has a utility function that measures the value it puts on a quality assignment. We will use these utility functions to define an overall *System Utility*, $u : Q_1 \times \cdots \times Q_n \rightarrow \mathbb{R}$. Many different definitions are possible, depending on the system or policy in question. Examples include:

- $u = u_w$, a (weighted) sum of application utilities

$$u_w(q_1, \ldots, q_n) = \sum_{i=1}^{n} w_i u_i(q_i)$$

  for *differential services,* where u_i is non-decreasing, and $0 \leq w_i \leq 1$ could be the priority of T_i, or

- $u = u^*$, where

$$u^*(q_1, \ldots, q_n) = \min_{i=1 \ldots n} u_i(q_i)$$

  for *"fair" sharing.*

Note that the algorithms or schemes presented in this thesis are for the weighted sum where the weights are set to 1 for simplification to present the algorithms.

As the reader can see, the QoS management model is very flexible and powerful. It can be adapted to fit a variety of situations. In this thesis, however, the weighted-sum definition is used for system utility.

2.5.3 Optimization Formulation

Given a set of task profile, our goal is to assign qualities (q_i) and allocate resources (r_i) to tasks or applications, such that the system utility u is maximized. Therefore we have the following *Problem Function* formulation

$$\begin{aligned}
&\text{maximize} && u(q_1, \ldots, q_n) && && \\
&\text{subject to} && q_i \geq q_i^{\min} \text{ or } q_i = 0\ , && i = 1, \ldots, n, && \textbf{(QoS Constraints)} \\
& && \sum_{i=1}^{n} r_{ij} \leq r_j^{\max}\ , && j = 1, \ldots, m, && \textbf{(Resource Constraints)} \\
& && r_i \models_i q_i\ , && i = 1, \ldots, n. &&
\end{aligned} \tag{2.1}$$

Due to the presence of a general relation ($r \models_i q$) in the constraints, this optimization problem is different from what is found in the general optimization literature. But we shall see in Section 4.2 that the problem can be transformed into a general combinatorial optimization problem.

If maximizing the profit margin, rather than the users' appreciation of the quality, is the objective goal of the system, the objective function $u(q_1, \ldots, q_n)$ in Equation 2.1 would be $\underline{u}\,((q_1, r_1), \ldots, (q_n, r_n))$ which is defined as

$$\underline{u}\,((q_1, r_1), \ldots, (q_n, r_n)) = \sum_{i=1}^{n} (u_i(q_i) - \mathit{cost}(r_i)) \tag{2.2}$$

The algorithms to be described in the Chapters 5 and 6 for Formulation 2.1 can be used straightforwardly with only trivial changes for the problem with modified objective function in Equation 2.2. See Section 2.6.3 for further discussion on the profit-based QoS management problem.

2.6 Expressive Power of Model

In order to show that the model described in last section is semantically rich, we will now show how some other models can be embedded into it.

2.6.1 Basic Priority Scheme

With the basic priority scheme, a task with high priority is selected or admitted to run over lower priority tasks, unless it is not possible to run the high priority task at all.

Let $p(T_i)$ be the priority of T_i. Let us further assume that tasks with high priorities are assigned low numbers and low priorities assigned high numbers.

To ensure that the system exhibits the basic priority behavior, we need to control the task admission in a way that a task, say T_j, will not be accepted to run unless all tasks with higher priority have been accepted or resources required by T_j exceed the available resource after tasks with higher priority than T_j are admitted. We will show that this basic priority policy can be realized in Formulation 2.1 through weight assignment and by giving a task utility 1 if it runs (and utility 0 if it does not run).

Let

$$p : \{T_1, \ldots, T_n\} \to \{1, \ldots, n\}$$

be a bijective function that describes the tasks' priorities with 1 as the highest and n as the lowest priority. We shall see later, that this function actually need not be bijective, and a group of tasks are allowed to have the same priority. We require bijectivity here for notational simplicity so we can make reference to p's inverse function, p^{-1}.

We want to show, that if T_i is admitted to run (i.e., it is allocated non-zero resource) then all T_j with higher priority, $p(T_j) < p(T_i)$, are also admitted to run. We can ensure this by simply assigning task T_i a weight of $w_i := 2^{-p(T_i)}$, where $p(T_i)$ is the priority of T_i.

We will demonstrate that such a assignment guarantees that a task with a higher priority will contribute more towards the system utility than the utilities combined by all tasks with lower priorities, therefore would be admitted in the system unless its resource requirement exceeds the reminding resources after higher priority tasks are alloted. In other words, we need to show that:

$$w_i u_i(q) \quad > \quad \sum_{j=p(T_i)+1}^{n} w_{p^{-1}(j)} u_{p^{-1}(j)}(q)$$

Substituting the assignment in for w_i and each w_j, we have

$$
\begin{array}{rrcl}
 & 2^{-p(T_i)}u_i(q) & > & \displaystyle\sum_{j=p(T_i)+1}^{n} 2^{-j}u_{p^{-1}(j)}(q) \\
\Updownarrow & & & \\
 & \left(\frac{1}{2}\right)^{p(T_i)} 1 & > & \displaystyle\sum_{j=p(T_i)+1}^{n} \left(\frac{1}{2}\right)^{j} 1 \\
\Uparrow & & & \\
 & \left(\frac{1}{2}\right)^{p(T_i)} & \geq & \displaystyle\sum_{j=p(T_i)+1}^{\infty} \left(\frac{1}{2}\right)^{j} \\
\Updownarrow & & & \\
 & \left(\frac{1}{2}\right)^{p(T_i)} & \geq & \left(\frac{1}{2}\right)^{p(T_i)+1} \displaystyle\sum_{j=0}^{\infty} \left(\frac{1}{2}\right)^{j} \\
\Updownarrow & & & \\
 & 2 & \geq & \displaystyle\sum_{j=0}^{\infty} \left(\frac{1}{2}\right)^{j}
\end{array}
$$

The right hand side of the last inequality is an infinite decreasing geometric series, which converges to 2. Thus Problem Formulation 2.1 with the weights set above indeed realizes the basic priority scheme.

If multiple tasks have the the same priority, the weights can be adjusted in the following way. First, give all tasks a distinct priority, and let the tasks to have the same priority be adjacent, but their relative order can be arbitrary. Then designate weights using $p(\cdot)$. Subsequently, for groups of tasks that have the same priority, use the lowests weight of the group for all the tasks in this group. Since doing this only lowers the weight of each task to the lowest of this group, the utility condition still holds.

2.6.2 Enhanced Prioritized Allocation

Assume that tasks in the systems are in adaptive nature as well as prioritized. One might wonder whether we can have a system that lets admitted task run with various levels of quality (therefore potentially different utility) while still enforcing that tasks are admitted based on their priority. We will refer such policy *adaptive prioritized allocation*, and we will show that Problem Formulation 2.1 can realize this adaptive prioritized allocation scheme. To see this, let again

$$p : \{T_1, \dots, T_n\} \rightarrow \{1, \dots, n\}$$

be a bijective function that describes the tasks' priorities with 1 as the highest and n as the lowest priority. Again this function need not be bijective, and tasks are allowed to have the same priority. As in Section 2.6.1 we require bijectivity here for notational simplicity.

Assume that

$$u_i(q) \in [\delta_i, 1] \qquad \text{for all } i \text{ and } q \in Q_i$$

where $\delta_i \in (0, 1]$ describes the utility range of T_i.

We want to show that if T_i is admitted to run, no matter upon which level of quality it will operate, then all T_j with higher priority, $p(T_j) < p(T_i)$, are also admitted to run with one of the quality modes specified in their corresponding quality spaces, unless resources required by T_j exceeds the available amount. We can ensure this by assigning proper weights, w_i, in the following way.

Let $\delta = \min\limits_{i=1 \dots n} \delta_i$, $w_i = (\delta/2)^{p(T_i)}$ and let p^{-1} be the inverse function of p. We will now prove that such weight assignments guarantees that a task with a higher priority contributes more towards the system utility than the utilities combined by all tasks with lower priorities:

$$
\begin{array}{rcl}
w_i u_i(q) & > & \displaystyle\sum_{j=p(T_i)+1}^{n} w_{p^{-1}(j)} u_{p^{-1}(j)}(q) \\
\Updownarrow & & \\
(\delta/2)^{p(T_i)} u_i(q) & > & \displaystyle\sum_{j=p(T_i)+1}^{n} (\delta/2)^j u_{p^{-1}(j)}(q) \\
\Uparrow & & \\
(\delta/2)^{p(T_i)} \delta & > & \displaystyle\sum_{j=p(T_i)+1}^{n} (\delta/2)^j 1 \\
\Uparrow & & \\
(\delta/2)^{p(T_i)} \delta & \geq & \displaystyle\sum_{j=p(T_i)+1}^{\infty} (\delta/2)^j \\
\Updownarrow & & \\
(\delta/2)^{p(T_i)} \delta & \geq & (\delta/2)^{p(T_i)+1} \displaystyle\sum_{j=0}^{\infty} (\delta/2)^j \\
\Updownarrow & & \\
(\delta/2)^{p(T_i)} \delta & \geq & (\delta/2)^{p(T_i)+1} \dfrac{1}{1-\delta/2} \\
\Updownarrow & & \\
2 & \geq & \dfrac{1}{1-\delta/2} \\
\Updownarrow & & \\
1 & \geq & \delta
\end{array}
$$

The latter inequality is true by assumption, so the former is always true. Thus the model in 2.1 with the weights set as claimed realizes the QoS adaptive, or enhanced, prioritized allocation.

2.6.3 Profit as Utility

As mentioned above, the system with objective funtion defined as in Equation 2.2 can be solved directly using the algorithms to be described later in this thesis. Nevertheless we will now show how it can also be embedded into the model in 2.1 by reducing the profit-based problem to an instance of the problem described in 2.1. In order to differentiate the symbols of one problem from the symbols of the other, the symbols of the profit-based problem will be underlined.

$$
\begin{aligned}
R &= \underline{R} \\
Q_i &= \{(q_i, r_i) \in \underline{Q}_i \times \underline{R} \mid r_i \underline{\models_i} q_i\} \\
q_i^{\min} &= (\underline{q}_i^{\min}, 0) \\
r_i \models_i q_i &\Leftrightarrow r_i = \mathcal{E}^r(q_i) \\
u_i(q_i) &= \underline{u}_i(\mathcal{E}^q(q_i)) - \underline{cost}(\mathcal{E}^r(q_i)) + C
\end{aligned}
$$

where $\mathcal{E}^r$ is a function that extracts the r component from a q-r-pair and $\mathcal{E}^q$ extracts the q component. The (partial) ordering of elements in Q_i is inherited from $\underline{Q}_i$. The constant C is defined as $\max_{\underline{r}} cost(\underline{r})$. It is used to ensures that utilities remain non-negative, and it does not influence the optimization result in any other way.

It can now be seen that this is a problem instance of Equation 2.1 and that it solves the profit-based management problem.

2.6.4 Connected or Conditional QoS Requirements

With the introduction of quality indices and quality dimensions, connected or conditional QoS requirements can also be easily specified in the QoS management system.

Assume that the user wants to add conditions on the quality apportioning across quality dimensions, e.g., the user might require that the quality on one dimension is contingent on the offered quality of another dimension. For example, for a video streaming session, a user might require that the video frame rate follow frame resolution in either the same or opposite direction, such as

$$\{\langle 5\,\mathrm{fps}, \mathrm{QCIF}\rangle, \langle 10\,\mathrm{fps}, \mathrm{CIF}\rangle, \ldots, \langle 15\,\mathrm{fps}, 16\mathrm{CIF}\rangle\}$$

where the rates follow each other, or

$$\{\langle 5\,\mathrm{fps}, 16\mathrm{CIF}\rangle, \langle 10\,\mathrm{fps}, 4\mathrm{CIF}\rangle, \ldots, \langle 15\,\mathrm{fps}, \mathrm{QCIF}\rangle\}$$

where the rates go against each other. This can be viewed as the user explicitly disallowing certain quality points, or alternatively as restricting the set of valid QoS points. This conditional QoS requirements can be accommodated in one of the following ways:

Pre-digitize Quality Space in User Profile When the size of the qualified (or disqualified) quality space is small, we can let the *user* specify the qualifying (or disqualifying if the number of disallowed points is smaller) points.

Augment QoS Constraints Alternatively, we can simply let the user describe the connected QoS requirements through predicates as augmented QoS constraints.

Augment Resource Profile Furthermore, the system can faciliate such conditional QoS requirements through restricting $r \models_i q$. Let $Q_i^c \subseteq Q_i$ be the qualifying points. We can embed this into the model described in 2.1 by simply making sure that $r \models_i q$ is never true for any such q, i.e., using

$$r \mathrel{\widehat{\models}}_i q \quad \equiv \quad q \in Q_i^c \wedge r \models_i q$$

as our resource profile.

2.6.5 Functional Q_i-R Relationship

Many authors, for example [53, 22], assume a functional relationship between resource and quality, i.e., either $q_i = f_q(r_i)$ or $r_i = f_r(q_i)$ for suitable f_q or f_r. Such functions are special cases of our general relation, $r \models_i q$.

2.7 Chapter Summary

This chapter formalized our QoS management problem and introduced the basic entities of our model. It first introduced the notion of *quality dimensions* across which optimization will need to be performed. The concept of *quality index* allows a mapping from non-uniform QoS dimensions to integers. The notion of *application utility* is used to quantify the relative merits of various QoS points. *QoS tradeoffs* can therefore be made based on application utilities. The same QoS operating point can be satisfied by making tradeoffs among resources. For example, CPU time might be traded for network bandwidth by compressing or not compressing a video stream.

Profiles are used to characterize applications and their requirements. A *task profile* characterizes the relationship between QoS and resource requirements, and consists of an application profile and a user profile. An *application profile* consists of a QoS profile and a resource profile. The *QoS profile* specifies the utility of all possible QoS points, or modes, while the *resource profile* specifies the relation between the QoS points and the resources necessary to achieve them. The application profile is typically created by the application designer. The *user profile* is a possibly customized instantiation of an application profile by the end-user.

The goal of our optimization is to assign qualities and the corresponding allocation of resources to applications such that the total system utility is maximized. This optimization model is semantically rich and can also be used to model traditional schemes such as prioritized allocation and "fair" sharing schemes. The model also supports conditional QoS requirements, where the quality along one dimension is contingent on the offered quality along another dimension.

In conclusion, our QoS management model can be very flexible and expressive. Here we only describe several examples of realization schemes. We expect that this formulation is capable of modelling and supporting the ever demanding and sophisticated QoS requirements.

Chapter 3

Specification Methodology for QoS Provision

At the crux of our translucent QoS management optimization system lies the QoS specification. It is important that we provide a powerful and semantically rich QoS specification for the system to use for service optimization. Equally important we need to provide a *user friendly interface* that facilitates specification acquisition.

The reason for the emphasis on QoS specification and interface design might not be obvious, but the reader should see the point shortly when we take a closer look at the quality space of a typical multimedia systems.

QoS specifications represent the combined wishes of the user and the application writer. But while the application writer can be assumed to be willing to spend substantial effort in order to produce a suitable QoS specification for optimization, the same is not true for the user who, after all, is interested in using the application, not in pleasing the optimization system. On one hand, this puts a severe constraint on the level of complexity that the application can present to the user for purposes of QoS control. On the other hand, the user must be presented with enough options that his or her desires can be adequately expressed. Therefore, the user interface must strike a careful balance.

In our previous work [29], we presented RT-Phone (Figure 3.1), an IP video conferencing system with some preliminary QoS control. This system had two modes of user control over quality — basic and advanced.

In the basic mode, we had a simple interface in which overall quality of service was

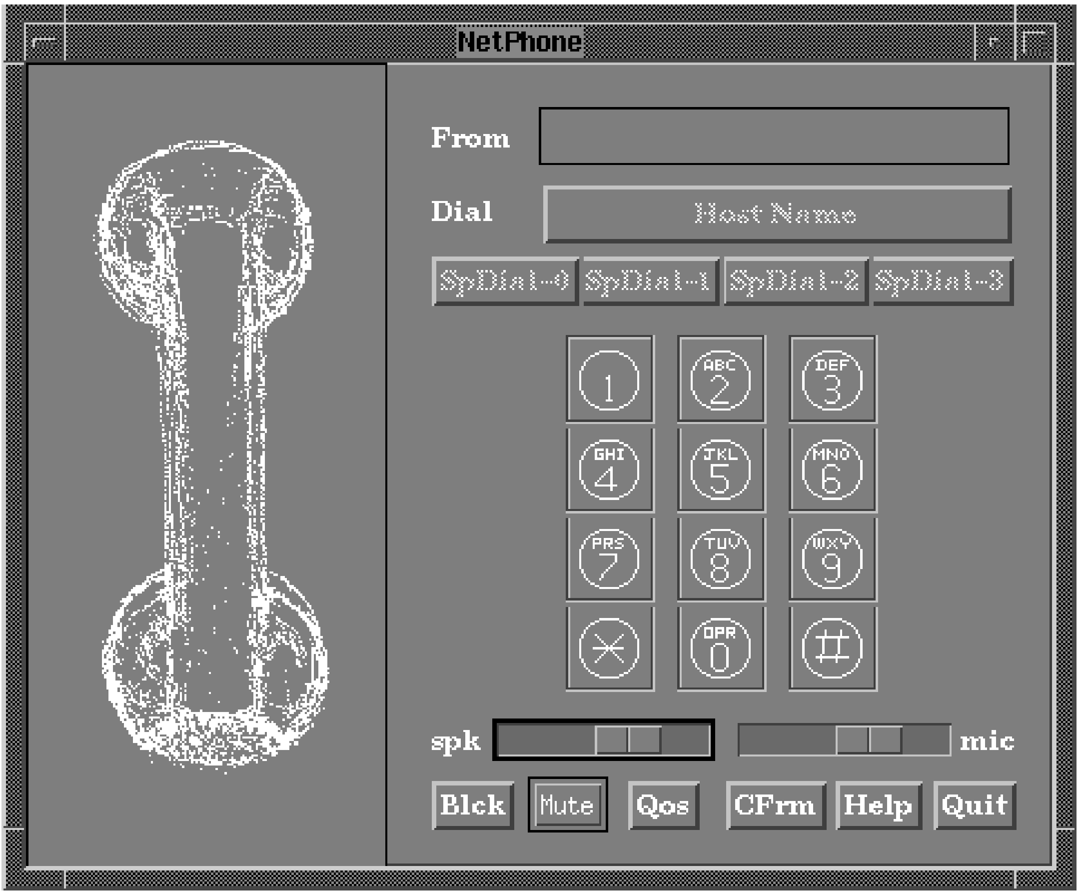

Figure 3.1: A Screendump of RT-Phone

selected on a scale from one to ten. Each point of the increasing scale corresponded to an *application-defined* quality point. Selecting a quality point for the RT-Phone model is equivalent to setting both $q_i^{\min}$ and $q_i^{\max}$ to the same value in the model of this thesis, that is to say the RT-Phone model has no utility concept. It should be clear that just selecting on a scale from one to ten is a fairly coarse way of specifying quality, as we can think about it as a "QoS digitization" by the application designer (rather than the end user) where many of the quality choices will be ruled out.

The RT-Phone system therefore also had an advanced mode that allowed users control over individual quality dimensions (see Figure 3.2). Since the RT-Phone QoS model did not have the quality index and utility concepts, the power of QoS control is of a much lesser degree than suggested in the example of Section 2.1. We believe that the principles of facilitating the dimensional quality control in advanced mode are sound, so we have adapted them into the QoS specification model defined in [25, 30, 27] and this thesis.

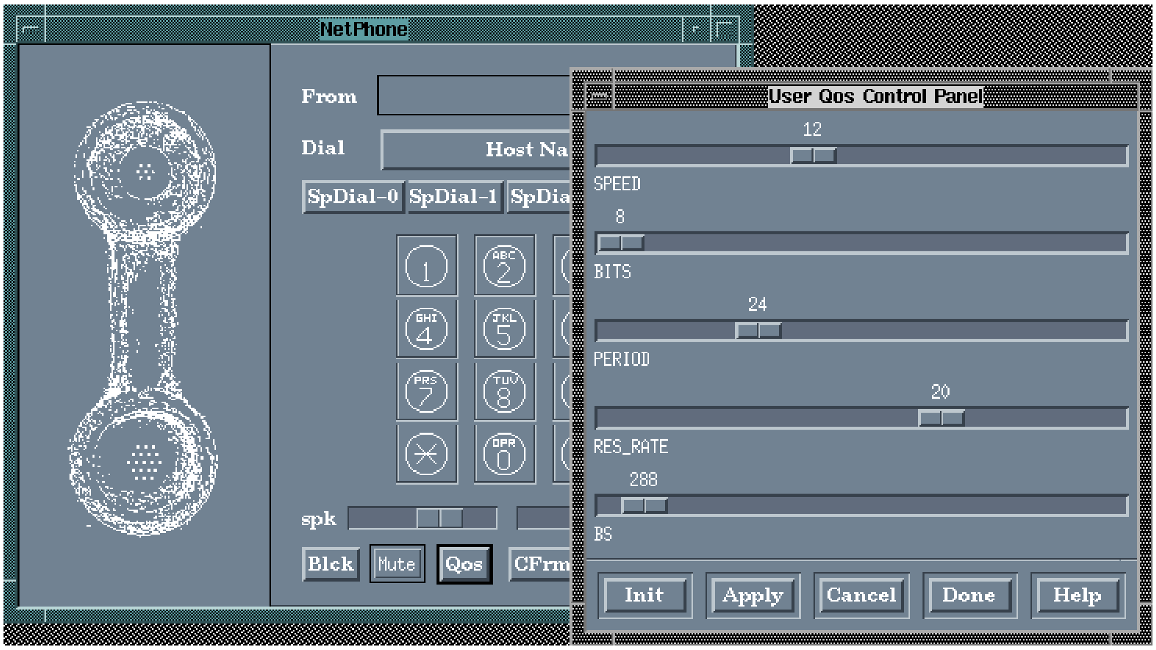

Figure 3.2: RT-Phone QoS Control

3.1 Quality Space

The application utility is a function defined on the set of quality points, Q_i. Therefore, producing a QoS specification consists of determining a value for each element in Q_i, either directly or through some indirect means.

To illustrate this, let us revisit some of the quality dimensions for the video conferencing system shown in section 2.1

Quality	Choices	Choice Count
Cryptographic security	0, 56	2
Picture format	SQCIF, ..., 16CIF	5
Color depth(bits)	1, 3, 8, 16, 24	5
Frame rate(fps)	1, 2, ..., 30	30
Sampling rate(kHz)	8, 16, 24, 44	4
Sample size	8, 16	2
Audio end-to-end delay(ms)	25, 50, 75, 100	4

The original specification had ellipses, "...", representing more choices than shown above. We will ignore those here.

The size of the QoS specification, that is the total number of different quality points, in this example is

$$|Q_i| = \prod_{j=1}^{d_i} |Q_{ij}| = 2 \times 5 \times 5 \times 30 \times 4 \times 2 \times 4 = 48000.$$

With this many quality points, it is clearly infeasible to have the user specify the utility on a point-by-point basis; there are simply too many choices. Therefore a pragmatic scheme is needed to address the issue.

3.2 Dimensional Utilities

Obvious way of trying to solve the specification problem is to have the user specify the utility of selected quality points and then interpolate. Unfortunately, this does not work well in a multidimensional quality space, where the quality points are not completely ordered. For example, if we have two dimensions each with two points, there will be four quality points:

$$Q_i = \{\langle 1, 1\rangle, \langle 1, 2\rangle, \langle 2, 1\rangle, \langle 2, 2\rangle\}.$$

The points $\langle 1, 2\rangle$ and $\langle 2, 1\rangle$ are unordered. The lack of order increases with the number of dimensions and turns interpolation into extrapolation.

Gaining insight from decision science as well as our previous experience, we therefore provide the user with the capability to specify dimensional quality utilities. The application utility can then be defined in term of dimensional utilities. In particularly, we embed the dimensional QoS control capability scheme into our utility based QoS management model by allowing task T_i to specify functions $u_{ij} : Q_{ij} \rightarrow \mathbb{R}$, one for each quality dimension $j \in \{1, \ldots, d_i\}$. We call these "dimentional utility functions" and combine them to form the task's utility function as we shall see in Section 3.3.

The quality points in the multi-dimensional case do not have a complete ordering, but the individual dimensions do. Moreover, some common properties associated with dimensional quality utility are observed including: non-decreasing, often quasi-continuous and piecewise concave. Figure 3.3 depicts some typical utility function shapes. Therefore, the application writer can easily define and supply the user various such function classes that can be used as templates for the user to instantiate for the dimensional utility function.

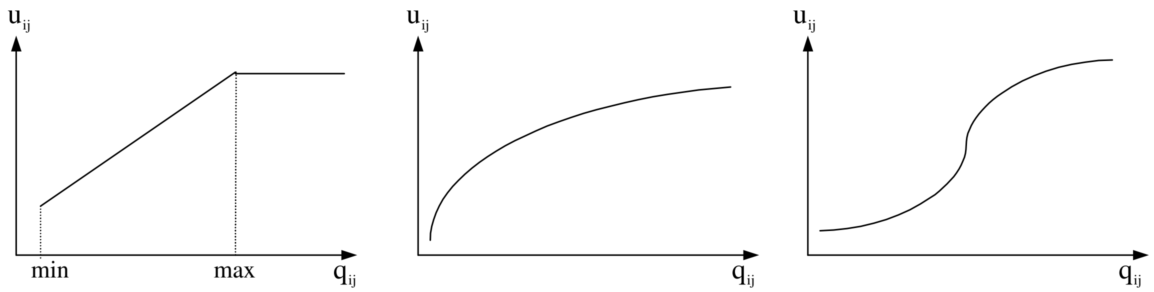

Figure 3.3: Typical Dimensional Utility Functions

Note that the algorithms presented in Chapters 5 and 6 rely only on the (quite natural) property of non-decreasing utility. The others are only used for user interface purposes — the continuous-looking graphical presentation of a function template only serves the purpose of easing the user defining his QoS utility function.

These utility function classes will be put into the general dimensional utility function template pool. Upon sighting a matching template, the user can customize it with actual parameter instantiation.

3.3 Construction of Application Utility

In the same way that application utilities were combined into an overall system utility — see Section 2.5.2 — we can now combine the dimentional utility functions into application utility functions. One reasonable way of doing this is to use a weighted sum

$$u_i(q_i) = \sum_{j=1}^{d_i} w_{ij} u_{ij}(q_{ij})$$

and for the remainder of this thesis, we shall do just that. The weights allow us to emphasize certain quality dimensions at the expense of others.

This creates an interesting issue regarding how weights should be assigned. Currently, the Analytic Hierarchy Process (AHP) [49] is used to cope with the problem.

By using dimensional utility functions, we have reduced the number of utility values that need to be determined from 48000 above to

$$\sum_{j=1}^{d_i} |Q_{ij}| = 2 + 5 + 5 + 30 + 4 + 2 + 4 = 52$$

if we ignore the weights. This may or may not still be too much to demand from the user, but we shall see shortly that there are ways of bringing it down even further. Still, a reduction of three orders of magnitude is a good start.

3.4 Example Dimensional and Application Utility

Again, an example task profile will be presented to illustrate the possible structure of dimensional utility functions and application utility functions.

Recall that application utility u_i for T_i is defined as a weighted sum of the dimensional quality utilities.

$$u_i(q_i) = \sum_{j=1}^{d_i} w_{ij} u_{ij}(q_{ij})$$

where u_{ij} are the dimensional utility functions. Example definitions could be:

Function	**Comments**
$u_{i1}(q_{i1}) = 20q_{i1}$	Picture format: linear.
$u_{i2}(q_{i2}) = 100q_{i2}/3$	Color depth: linear.
$u_{i3}(q_{i3}) = 100(1 - e^{aq_{i3}+b})$	Frame rate: exponential decay, assume T_i achieves 50% at $q_{i3} = 5$ and 95% at $q_{i3} = 20$. Therefore $a = -0.1535, b = 0.0744$.
$u_{i4}(q_{i4}) = 20(q_{i4} - 1)$	Encryption: linear.
$u_{i5}(q_{i5}) = 100(1 - e^{-1.5q_{i5}})$	Audio sampling rate: exponential decay, T_i achieves 95% at $q_{i5} = 2$ or 16 kHz.
$u_{i6}(q_{i6}) = 50q_{i6}$	Audio sampling bits: linear.
$u_{i7}(q_{i7}) = 20q_{i7}$	End-to-end delay: linear, achieves 100% at the best quality point, $q_{i7} = 5$ or 25 ms delay.

Figure 3.4 depicts the utility curve described above for frame rate.

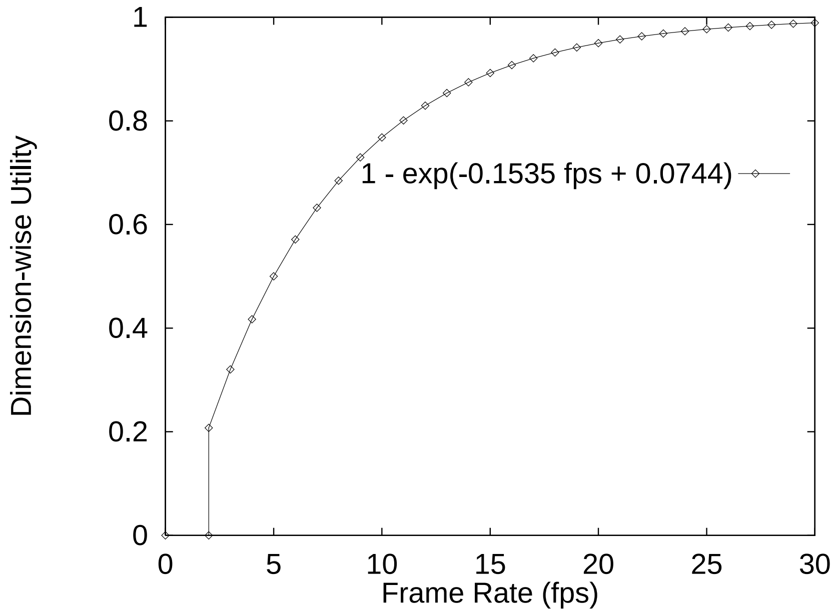

Figure 3.4: Dimensional Utility Function for Frame Rate

Suppose T_i is a remote surveillance system, where video is much more important to the user than audio. Assume that SQCIF, gray-scale, low frame rate is fine for video, and there is no need for encryption. Therefore, in the example system of Section 2.2, we could have the following minimum quality specification

$$q_i^{\min} = \langle 1, 1, 2, 1, 1, 1, 2 \rangle$$

which corresponds to the following minimum quality

$$\langle \text{SQCIF}, 1\,\text{bpp}, 2\,\text{fps}, \text{no encryption}, 8\,\text{kHz}, 8\,\text{bps}, 75\,\text{ms} \rangle.$$

Since video is more important to the user than audio, an example application utility function for T_i could be:

$$u_i(q_1, \ldots, q_7) = 5\Big(\underbrace{u_{i1}(q_{i1}) + \cdots + u_{i4}(q_{i4})}_{\text{video}}\Big) + 1\Big(\underbrace{u_{i5}(q_{i5}) + \cdots + u_{i7}(q_{i7})}_{\text{audio}}\Big)$$

where video quality is weighted five times more than that of audio.

3.5 Other Interface Issues

If a user were to choose quality on a scale of 1 to 10 with some pre-determined quality choices preset by the system, the interface would be very simple, but such built-in "QoS digitization" can severely limits the degree of customization.

Satisfaction Knee Points A more flexible, but also more sophisticated, scheme would be to have a set of parameterized utility curves available for each quality dimension, and to have the user pick the curves and instantiate appropriate parameters/coefficients. In our system, the instantiation is carried out by letting the user graphically specify *Satisfaction Knee Point parameters.* For the exponential-decay used in the previous example ($u_{i3}(q_{i3}) = 1 - e^{aq_{i3}+b}$), the user could specify the 50% and 95% levels. This is enough to uniquely determine a and b. For example, a user could specify $\langle 5\,\text{fps}, 0.50\rangle$ and $\langle 20\,\text{fps}, 0.95\rangle$, and the corresponding utility curve would then be the one shown in Figure 3.4, with $a = -0.1535$ and $b = 0.0744$.

Satisfaction Knee point choice can be realized through a drag-able marker graphic interface. Upon choosing a utility function graph outline, a user can drag markers, representing satisfaction knee points, to the desired positions. The corresponding dimensional utility function can therefore be determined and fed into system automatically.

By using Satisfaction Knee points, we can further reduced the number of utility values that need to be determined from 52 above to approximately two for each dimension in most cases, which amounts to a total of 12 for the example above.

It would be ideal to have an interface that can *assist the user* digitize the quality to a certain range of scale, and acquire the corresponding utility accordingly. Note that such mapping process is different from the built-in QoS digitization, where the quality choices are pre-determined by application designer.

One way could be to move the dimensional utility function method to the user interface part to synthesize or digitize quality-utility data, as it could significantly reduce the quality space searched by the QoS management optimizer. This especially crucial if the management system is to be deployed on a gate or route of a mediume or larger area network.

User QoS Constraint Recall the definition of *User Profile* in Section 2.5 we allow a user to specify its quality constraints explicitly through $q_i^{\min}$. Alternatively we could let the user specify the constraints implicitly through utility functions by setting $u_i(q) = 0$ for all $q < q_i^{\min}$. We have yet to complete a user-interface study to decide whether this approach will compromise the simplicity of the user-interface. For now, we will use this QoS Constraint approach. In this thesis, we will use this explicit QoS Constraint approach.

3.6 Chapter Summary

This chapter presented our QoS specification methodology with emphasis on the application user. The *application utility* is a function defined on the set of quality points, but the number of quality points to be considered can sometimes become unwieldy. *Dimensional utility functions* represent a pragmatic way of specifying utility values for a large number of quality points. Affine, concave and s-curves are three typical dimensional utility functions. The dimensional utility functions also enable the expression of conditional QoS requirements, and can be individually modified by the end-user. Our dimensional utilities are guided by the results from decision science which allow relative weights to be assigned to various dimensions.

Chapter 4

Problem Complexity and Algorithm Design

4.1 Problem Taxonomy

We assume that multiple applications similar to the one described in Chapter 2 can co-exist in a system. We classify (Figure 4.1) our QoS management problem based

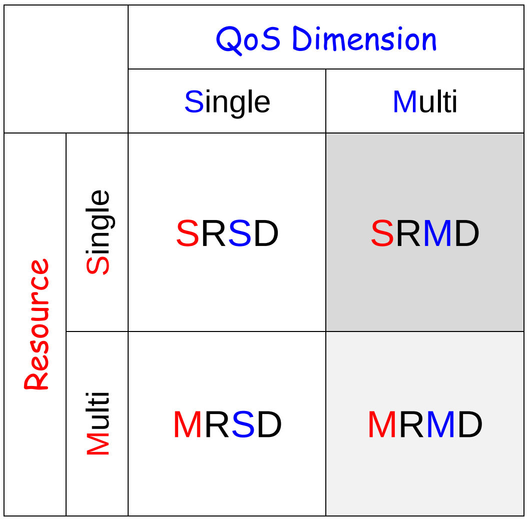

Figure 4.1: Problem Classification.

on whether the system deals just with a single resource or with any number, and on whether there is a single QoS dimension or multiple. We have the following four problem classes:

- Single Resource and Single QoS Dimension: SRSD
- Single Resource and Multiple QoS Dimensions: SRMD
- Multiple Resources and Single QoS Dimension: MRSD
- Multiple Resources and Multiple QoS Dimensions: MRMD

In this thesis, we are going to address the SRMD and MRMD problems directly. Since MRMD is a superset of the other problem classes, we could have chosen to address that class only, but it turns out that there are significant computational benefits to addressing SRMD separately. We have developed efficient schemes for SRMD that are not easily achievable for MRMD. The schemes we have for SRMD readily lead us to a QoS-driven single resource allocation when only a single resource is of concern (either it is the only resource under consideration, or it is relatively more scarce and other resources are abundant). For instance, these schemes can be used for QoS-driven disk, memory, network bandwidth as well as for processor scheduling.

We treat the two remaining classes only indirectly: an SRSD problem is also an SRMD problem and an MRSD problem is also an MRMD problem.

4.2 Problem Complexity

This combinatorial problem could also be formulated as follows. Let $\kappa_{i1}, \ldots, \kappa_{i|Q_i|}$ be an enumeration of the quality space, Q_i, for task T_i. Let $\rho_{ij1}, \ldots, \rho_{ijN_{ij}}$ be an enumeration of the resource usage choices (tradeoffs among different resources) associated with κ_{ij} for T_i, where N_{ij} is the number of such resource usage choices. (In particular we should always have $\rho_{ijk} \models_i \kappa_{ij}$.)

Let $x_{ijk} = 1$ if task T_i has been given quality point κ_{ij} and resource consumption ρ_{ijk}, and $x_{ijk} = 0$ otherwise. We can now reformulate system 2.1 as:

$$\begin{aligned}
\text{maximize} \quad & \sum_{i=1}^{n} \sum_{j=1}^{|Q_i|} \sum_{k=1}^{N_{ij}} x_{ijk} u_i(\kappa_{ij}) \\
\text{subject to} \quad & \sum_{i=1}^{n} \sum_{j=1}^{|Q_i|} \sum_{k=1}^{N_{ij}} x_{ijk} \rho_{ijk\ell} \leq r_\ell^{\max}, && \ell = 1, \ldots, m, \\
& \sum_{j=1}^{|Q_i|} \sum_{k=1}^{N_{ij}} x_{ijk} \leq 1 \ , && i = 1, \ldots, n, \\
& x_{ijk} \in \{0, 1\}, && i = 1, \ldots, n, \ j = 1, \ldots, |Q_i| \, , \ k = 1, \ldots, N_{ij}.
\end{aligned} \tag{4.1}$$

The three constraints express the following. The first one ensures that we will not oversubscribe resources. The second ensures that we assign at most one quality-resource allotment per task. The third one expresses the boolean nature of the choice variables x_{ijk}.

Note, that $\rho_{ijk\ell}$ is just the ℓth coordinate of the vector ρ_{ijk}.

Therefore all the instances of our problem can be viewed as special cases of the general integer or nonlinear programming problems.

Proposition 1 *SRSD, SRMD, MRSD, and MRMD are all NP-hard problems.*

Proof Since SRSD is a special case of the other three, we only have to show that SRSD is NP-hard.

For SRSD, we have $m = N_{ij} = 1$ and thus $k = \ell = 1$. System (4.1) becomes

$$\begin{aligned}
\text{maximize} \quad & \sum_{i=1}^{n} \sum_{j=1}^{|Q_i|} x_{ij1} u_i(\kappa_{ij}) \\
\text{subject to} \quad & \sum_{i=1}^{n} \sum_{j=1}^{|Q_i|} x_{ij1} \rho_{ij11} \leq r_1^{\max}, \\
& \sum_{j=1}^{|Q_i|} x_{ij1} \leq 1, && i = 1, \ldots, n, \\
& x_{ij1} \in \{0, 1\}, && i = 1, \ldots, n, \ j = 1, \ldots, |Q_i| \, .
\end{aligned} \tag{4.2}$$

The 0–1 Knapsack Problem is known to be NP-hard [34]. It can be described as follows. Given a set of n items and a knapsack of capacity c, with p_i and w_i the profit and weight of item i respectively, select a subset of the items so as to

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} p_i x_i \\ \text{subject to} \quad & \sum_{i=1}^{n} w_i x_i \leq c \\ & x_i \in \{0,1\}, \quad i = 1, \ldots, n, \end{aligned} \tag{4.3}$$

We can therefore reduce the 0–1 Knapsack Problem to SRSD by setting

$$\begin{aligned} Q_i &= \{(1)\} \\ u_i(q_i) &= p_i \\ r_1^{\max} &= c \\ \rho_{i111} &= w_i \end{aligned}$$

and have the 0–1 Knapsack Problem's x_i represented by x_{i11} in the SRSD case. Thus SRSD is as least as hard as the 0–1 Knapsack problem and therefore it is NP-hard.□

4.3 Algorithm Design Issues

As just shown and also in [30][28], the QoS management optimization problems are NP-hard. As a consequence, there are no optimal solution techniques other than a (possibly complete) enumeration of the solution space. On the other hand, QoS management calls for on-line solutions as the optimization module will ideally be in the heart of an admission control and adaptive QoS management system. Therefore the goal is to strike the right balance between solution quality and computational complexity.

For more than two decades, many researchers from the fields of mathematics, computer science and operations research have been working on the combinatorial optimization and solving NP-hard problems. There are three algorithmic approaches [1] [34] that have been well studied and widely used:

Enumerative methods

that are guaranteed to produce an optimal solution [13][14],

Approximation algorithms
that run in polynomial time [50][15], and

Heuristic techniques (under the general heading of local search)
that do not have *a priori* guarantee in terms of solution quality or running time, but provide a robust approach to obtaining a high-quality solution to problems of a realistic size in reasonable time [1].

An important attribute is the incremental and state-reuse property of a scheme, so as to avoid having to completely redo expensive computations to accommodate the dynamic arrival and departure of tasks. Also, we ensure that all algorithms should be formulated so that the the search for an optimal solution can be terminate at any time while still reaching a *feasible,* but sub-optimal and hopefully good, solution. These two properties are essential for an algorithm to be used in an online (or near-online) environment.

Therefore a series of schemes have been developed that give approximation, approximation with bound, and exact solutions, with increased asymptotic computational complexity. These algorithms use various optimization techniques including linear and nonlinear programming, constraint relaxation, basic dynamic programming, branch-bound, advanced dynamic programming with addition of dominance rules, direct and local search schemes.

It will be necessary to conduct extensive empirical studies to evaluate the practical performance of these algorithms when deployed under different system setups and task profiles. For instance, the systems to which QoS management optimization engine could be deployed could range from an end-node multi-media workstation, small or medium scale proxy/transceiver[1] servers, medium or large (with firewall and routing capability) gates [23], and on-demand media (news, video, stock quote, game) servers. These studies allow comparison of the relative performance of the the algorithms and answer questions such as whether algorithms are robust [44] enough to cover multiple cases, or whether combination algorithms might prove useful. In the latter case, the QoS management optimization engine could fire an algorithm based on the particular data instance exhibited by the profiles of application/user sessions in the system.

[1]Data distillation for low-link-speed mobile or other clients for instance.

Another important issue, which is policy-dependent but would affect the actual algorithm design, is the stability of the task quality assigned to existing tasks in the system. In the case where policy requires that quality not degrade for certain tasks, some algorithms might not be suitable , while others might be more appropriate.

4.4 Structure Composition of Resource & Utility

In Section 2.5 we explained that no *function* relationship can be described between quailty and resource. Similarly, there is no direct function relationship present between resource and utility, as utility is a function of quality. Moreover, due to the multi-dimensional and potentially subjective nature of quality of services, there is often no complete ordering among quality-of-service points, even for individual tasks.

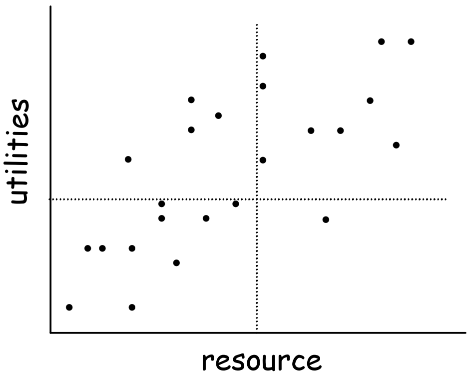

Figure 4.2: Scatter of Resource and Utilities

As in the case of resource and quality, only a scatter plot for R and U can be drawn; an illustration is shown in Figure 4.2. Therefore some structural composition or processing is required for those algorithms that call for mapping from resource to utility as heuristic. Fortunately, an *R-U* (resource to utility) function/graph can constructed for each task through *QoS Profile* and *Resource Profile.* Such an *R-U* graph can be drawn by listing each valid quality point's resource usage(s) and its corresponding utility through the steps described below.

Recall that given a resource allocation to a task, one could use the resource to improve different QoS dimensions, which could therefore lead to different utility values.

But the most valued QoS point for each resource value can be picked (as depicted in Figure 4.3, assume that $u_1 > u_2$) as intuitively, we certainly want to assign resources to those quality points with the highest utility value.

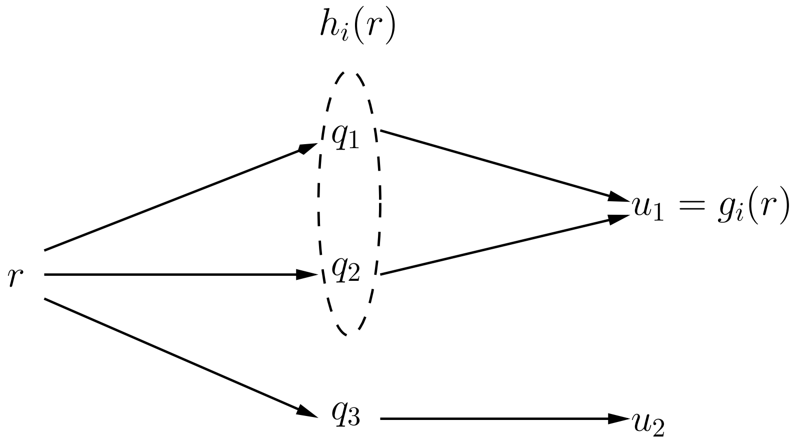

Figure 4.3: Resource-Utility Structure Composition

We therefore can define a function $g_i : R \to \mathbb{R}$ by

$$g_i(r) = \max\{\, u_i(q) \mid r \models_i q \,\} \tag{4.4}$$

and define $h_i : R \to \mathcal{P}(Q_i)$ (see Figure 4.3) to retain the quality points associated with the utility value $g_i(r)$:

$$h_i(r) = \{\, q \in Q_i \mid u_i(q) = g_i(r) \wedge r \models_i q \,\} \tag{4.5}$$

Then an *R-U* graph can be generated for each task, each of which would be a step function (perhaps with multiple level of steps).

When utility is defined as profit, functions g_i and h_i above will be defined as

$$g_i(r) = \max\{\, \bar{u}_i(q, r) \mid r \models_i q \,\} \tag{4.6}$$

$$h_i(r) = \{\, q \in Q_i \mid \bar{u}_i(q) = g_i(r) \wedge r \models_i q \,\} \tag{4.7}$$

4.5 Chapter Summary

This chapter classify our problems into four subcategories, two of which will be given direct treatment in the following chapters. Complexity, algorithm design issues and structure composition was discussed.

The taxonomy of the problem space was defined based on our quality optimization model. The following four classes of problems are identified: Single-Resource Single QoS Dimension (SRSD), Single-Resource Multiple QoS Dimensions (SRMD), Multiple-Resource Single QoS Dimension (MRSD) and Multiple-Resource Multiple QoS Dimensions (MRMD). All these four problem classes are then shown to be NP-hard. Enumerative techniques, approximation algorithms or heuristics must therefore be applied to solve our optimization problem. Since a simple functional relationship does not exist between quality and resource in our case, a structural composition processing scheme is introduced that produces resource-utility heuristics for the approximation algorithms to be presented in Chapter 5 and 6.

Chapter 5

SRMD Algorithms

In this chapter we focus on the SRMD problem where just one resource under management. We present two near-optimal algorithms to solve this problem. The first yields an allocation within a known bounded distance from the optimal solution, and the second yields an allocation whose distance from the optimal solution can be explicitly controlled by the QoS manager. We compare the *asymptotic* performance of the approximation algorithms to an exact algorithm which in turn is designed using dynamic programming. Their *practical* performance evaluations will be presented in Chapter 7.

5.1 An Optimal Solution Scheme

Assume that the resources are allocated in units of $r^{\max}/P$ for some integer P. If, for example, $P = 100$ this would mean that allocation is in integer percentage. Under this assumption, we can characterize the structure of the optimal solution and recursively define its value as follows.

Denote by $v(i, p)$ the maximum utility achievable when the first i of n tasks are considered with resource $r^{\max}p/P$ available for allocation. We can describe $v(i, p)$ in terms of $v(i-1, \cdot)$ by considering all allocation choices for the ith task:

$$v(i,p) = \max_{p' \in \{0,\ldots,p\}} \{g_i(p') + v(i-1, p-p')\} \tag{5.1}$$

Obviously, $v(0, p) = 0$. As optimization, the set of interesting p' values to consider is in fact just all the (starting) discontinuity points of g_i (see Definition 4.4).

Therefore $v(n, P)$ will be the maximum utility achievable by allocating up to $r^{\max}$ to the n tasks, i.e., the best allocation overall.

Based on Equation 5.1, the following algorithm `srmd` can be constructed through dynamic programming. Let

$$C_i = \left\langle \begin{pmatrix} u_{i1} \\ r_{i1} \end{pmatrix}, \ldots, \begin{pmatrix} u_{ik_i} \\ r_{ik_i} \end{pmatrix} \right\rangle$$

denote the utility function g_i's discontinuity points in increasing u-order, and $qos(i, p)$ the list of QoS allocation choices for T_1 through T_i towards $v(i, p)$.

srmd$(n, P, C_1, \ldots, C_n)$

for $p = 0$ **to** P **do**
    $qos(0, p) := nil$
    $v(0, p) := 0$
    $r(0, p) := 0$
for $i = 1$ **to** n **do**
    $qos(i, 0) := nil$
    $v(i, 0) := 0$
    for $p = 1$ **to** P **do**
        $u^* := 0$
        $r^* := 0$
        $j^* := 0$
        for $j = 1$ **to** $|C_i|$ **do**
            if $(r_{ij} > p$ **or** $h_i(r_{ij}) < q_i^{\min})$ **break**
            $u := u_{ij} + v(i - 1, p - r_{ij})$
            if $u > u^*$ **then**
                $u^* := u$
                $r^* := r_{ij}$
                $j^* := j$
        $qos(i, p) := qos(i - 1, p - r_{ij^*})$ **concat** $[h_i(r_{ij^*})]$
        $v(i, p) := u^*$
        $r(i, p) := r^*$
$p := r^{\max}$
for $i = n$ **downto** 1 **do**

```
        resource(i) := r(i, p)
        u(i) := v(i, p)
        p := p − resource(i)
    return v(n, P), qos(n, P) resource(1), ...,resource(n),
                u(1), u(2) − u(1), ...,u(n) − u(n − 1)
```

The result $v(n, P)$, the utility accrued when 100% of the resource is available, is optimal. Let $L = \max_{i=1}^{n} |C_i|$. The time complexity of the algorithm is $O(nLP)$ or $O(nP^2)$, which is pseudo-polynomial.

One of the plus sides of this scheme (also true for the MRMD scheme described in Chapter 6) is its incremental and state-reuse property in which when a new task arrives, previous results can be directly reused to avoid the expensive recomputation of the complete new task set.

When the session length information of tasks are available, the task lists are generally ordered in decreasing session length order, so when a task T_n finishes and departs the system (and therefore releases some resources), the result for T_i, $i = 1, \ldots, n-1$ is already computed and kept in the system, that could be reused to make a quick decision (not necessarily to be optimal especially when a stability policy is in use) on which tasks' qualities could be improved.

When a priority-based policy alone is emphasized, the task list to be fed into the algorithm will be in non-increasing order of task priorities.

Algorithm `srmd` could be a practical method for QoS-driven single resource allocation, such as processor scheduling in operating systems which support QoS. The algorithm, with minor change, would be suitable to deal with the *stability* problem when a user prefers (or a policy requires) a relative consistent quality.

5.2 An Approximation Scheme

By constructing the convex hull for each of g_i (see Definition 4.4) functions we get piece-wise linear relaxation functions g_i°, $i = 1, \ldots, n$. The gradients of of g_i° can be used as a heuristic to allocate resources among these tasks. Let

$$C_i = \left\langle \begin{pmatrix} u_{i1} \\ r_{i1} \end{pmatrix}, \ldots, \begin{pmatrix} u_{ik_i} \\ r_{ik_i} \end{pmatrix} \right\rangle$$

be the utility function g_i's discontinuity points in increasing r-order (therefore increasing u-order as well). We will refer to these lists as r-u-pair lists. Denote by r^c the current remaining resource capacity after certain resource has been allocated; s_list$[i].t$, s_list$[i].r$, s_list$[i].u$ the task id, the associated r-value and u-value of the corresponding r-u-pair list; and r$[i]$ the resource allocated for T_i.

asrmd1$(n, C_1, \ldots, C_n)$

1. **for** $i = 1$ **to** n **do**
2. $C'_i :=$ **convex_hull_frontier**(C_i)
3. $u[i] := 0$
4. $r[i] := 0$
5. *s_list*= **merge**$(C'_1, \ldots, C'_n)$
6. $r^c := r^{\max}$
7. $u := 0$
8. **for** $j = 1$ **to** $|s\_list|$ **do**
9. $i := s\_list[j].t$
10. $\beta = s\_list[j].r - r[i]$
11.   **if** $(\beta \le r^c)$ **then**
12. $r^c := r^c - \beta$
13. $r[i] := s\_list[j].r$
14. $u[i] := s\_list[j].u$ /* *Update allocation info for* T_i. */
15.   **else**
16.   **break**
17. **for** $i = 1$ **to** n **do**
18. $q[i] := h_i(r[i])$ /* *See Definition 4.5.* */
19. $u := u + u[i]$
20. **return** $(u, q[1], \ldots, q[n], r[1], \ldots, r[n], u[1], \ldots, u[n])$

Note that each $q[i]$ provides a set of quality choices from which T_i (its user, or session manager) could choose to make further QoS tradeoffs.

Notice that in implementation, we actually replace "break" in line 16 with continue (i.e., let the loop continue when condition at step 11 does not hold). This means that after the optimality condition (to be described shortly) is violated, the residual capacity (r^c) will be greedily filled. The continuation can be thought as a post-optimization process. The error bound property to be proved below holds for either case.

Let $L = \max_{i=1}^{n} |C_i|$. After the procedure *convex_hull_frontier*[1] (which takes time $O(nL)$) a convex hull frontier with non-increasing slope segments (piece-wise concave) is obtained for each task. The segments are merged at step 5 using a divide-and-conquer approach with $\log_2 n$ levels, each level having nL comparisons. Merging thus takes time $O(nL \log n)$. Steps 8 through 16 require $O(|s\_list|) = O(nL)$. Steps 17 through 19 take $O(n)$. The total running time of the algorithm is thus $O(nL \log n) + O(nL) = O(nL \log n)$.

Denote by δ_i the maximum utility difference between adjacent discontinuity points of C'_i, i.e., the largest increase in utility for task T_i on the convex hull frontier. Let $\chi = \max_{i=1}^{n} \delta_i$. Denote by U_{opt} the optimal utility result and U_{asrmd1} the approximation result obtained by algorithm `asrmd1`.

Theorem 1 *U_{asrmd1} is within χ of U_{opt}, i.e. $U_{opt} - \chi < U_{asrmd1} \leq U_{opt}$.*

Proof Note first, that if the residual resource, r^c, ends up being zero before executing "**break**" at step 15 (or if j reaches the end of $|s\_list|$), then the solution found is in fact optimal based on the Kuhn-Tucker condition[43], as each g_i° (represented by C'_i in `asrmd1`) is essentially a piece-wise concave function.

Algorithm `asrmd1` produces a utility value, U_{asrmd1}, which is feasible. Therefore we have $U_{\text{asrmd1}} \leq U_{\text{opt}}$.

Suppose that convex hull frontier segments (ordered and stored in *s_list*) are consecutively used (with corresponding quality upgrade and added utility) until the first segment, s, is found that requires more resource than residual resource capacity r^c to realize the extra utility at the end of the segment s (remember that the convex hull segments are imaginary linear relaxation of the real utility functions).

Let the two end points of the critical segment s be (r_{si}, u_{si}) and (r_{si+1}, u_{si+1}) in C'_i. Based on the Kuhn-Tucker condition and the Dantzig[9] upbound (combined referred to as the *Optimality Condition*), we have

$$U_{\text{opt}} \quad \leq \quad U_{\text{asrmd1}} + (r^c - r_{si}) \frac{u_{si+1} - u_{si}}{r_{si+1} - r_{si}}$$

[1]Overmars & Leeuwen's [42] algorithm, or simply the quickhull [45] or Graham-Scan [7] when C_i are not pre-sorted.

$$
\begin{aligned}
&< U_{\text{asrmd1}} + (r_{si+1} - r_{si}) \frac{u_{si+1} - u_{si}}{r_{si+1} - r_{si}} \\
&= U_{\text{asrmd1}} + u_{si+1} - u_{si}
\end{aligned}
$$

and we know that $u_{si+1} - u_{si} \leq \chi$, therefore $U_{\text{opt}} - U_{\text{asrmd1}} <^2 \chi$. $\square$

Remark: To give a feel for how tight the bound is from below, examine two cases (see Figure 5.1) when the results are suboptimal. The reason for the first case is

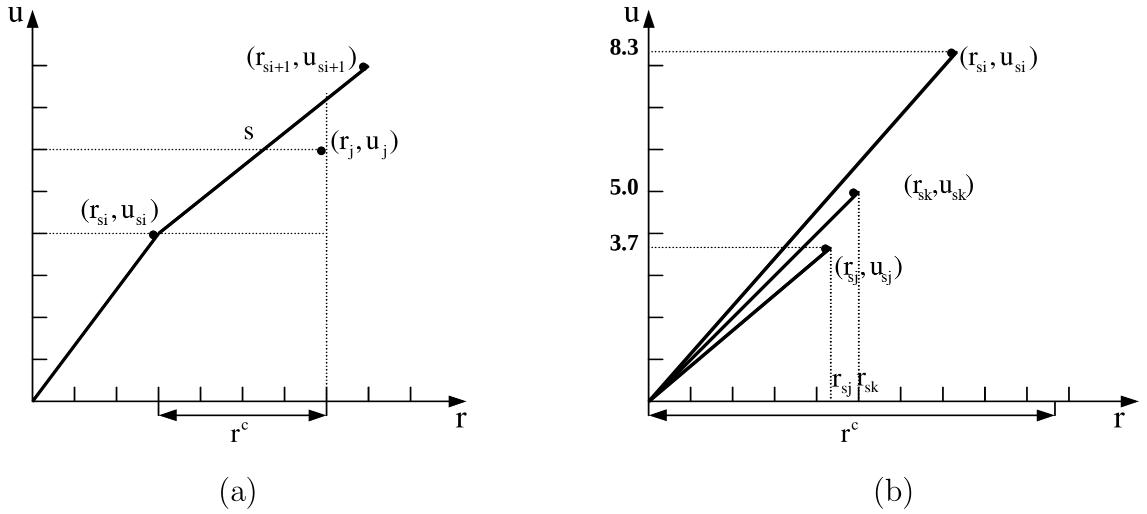

Figure 5.1: Suboptimal Cases

sub-optimality is due to the convex hull approximation error (where one or more intermediate utility points are bridged and removed when we construct g_i° (or C_i'), from g_i (or C_i); the reason for the second case is the consequence of the greedy heuristic (no costly backtracking after optimal condition is violated) near the end of the `asrmd1` optimization process.

Case 1. When interior (intermediate) points are bridged over and dominated by the critical convex hull segment s (see Figure 5.1(a)).

Let the inferior point bridged over by s with the largest utility be (r_j, u_j), where $j > si$ in the original C_i list of T_i. Further assume that $r_j - r_{si} \leq r^c$, and there are no more elements left in s_list. Then when `asrmd1` stops and reports the achieved utility of U_{asrmd1}, which excludes (r_{si+1}, u_{si+1}), the optimal is in fact $U_{\text{opt}} = U_{\text{asrmd1}} + (u_j - u_{si})$. Since $u_j - u_{si} < \chi$, $U_{\text{asrmd1}} < U_{\text{opt}} - \chi$.

[2] Except in the degenerate case where $\chi = 0$, and $U_{\text{opt}} - U_{\text{asrmd1}} = \chi = U_{\text{opt}} = U_{\text{asrmd1}} = 0$.

Case 2. When $r^c > r_{si}$, and there are (r_{sj}, u_{sj}) and (r_{sk}, u_{sk}) [3] in s_list, where $sj, sk > si$, and their slopes are lower than that of $\binom{u_{si}}{r_{si}}$, but $(r^c - r_{si}) < r_{sj}$, $(r^c - r_{si}) < r_{sk}$, $r_{sj} + r_{sk} \leq r^c$, and $(u_{sj} + u_{sk}) > u_{si}$ (Figure 5.1(b)). By the Dantzig bound, the U_{asrmd1} would be well within χ as well.

Although `asrmd1` is a polynomial approximation algorithm with a describable and potentially small error bound from the optimal result, the bound is not controllable. Section 5.3 presents another polynomial scheme with a controllable error bound.

5.3 A Polynomial Scheme with Controllable Bound

The algorithm `asrmd2` to be described will give an approximate quality and resource allocation which is guaranteed to have a maximum relative error, ε, where $0 < \varepsilon < 1$ is a user-specified value. A relative error of ε means that the utility U_{asrmd2} found by the algorithm satisfies

$$(1 - \varepsilon)U_{\text{opt}} \leq U_{\text{asrmd2}} \leq U_{\text{opt}}$$

where U_{opt} is the optimal utility.

Before presenting `asrmd2`, let us define some data structures and operations to be used in the algorithm. All utility function g_i's discontinuity points are listed in increasing u-order as

$$C_i = \left\langle \begin{pmatrix} u_{i1} \\ r_{i1} \end{pmatrix}, \ldots, \begin{pmatrix} u_{ik_i} \\ r_{ik_i} \end{pmatrix} \right\rangle$$

where $\binom{0}{0}$ is the first element, and referred to as r-u-pair lists. We also define the following addition operation for r-u-pair lists and r-u-pair elements.

$$\left\langle \begin{pmatrix} u_1 \\ r_1 \end{pmatrix}, \ldots, \begin{pmatrix} u_k \\ r_k \end{pmatrix} \right\rangle + \begin{pmatrix} u \\ r \end{pmatrix} = \left\langle \begin{pmatrix} u_1 + u \\ r_1 + r \end{pmatrix}, \ldots, \begin{pmatrix} u_k + u \\ r_k + r \end{pmatrix} \right\rangle$$

Note, that this operation produces a new r-u-list that is sorted non-decreasingly in u-value. From now on such sorting will be assumed.

Let A and B be r-u-pair lists. The procedure *combine_and_merge* will combine A and B into a single r-u-pair list.

[3] Or a single element with higher utility value than u_{si} given $r_{si} + r^c$. This case is not shown in Figure 5.1

combine_and_merge(A, B)

1. **foreach** $b_i \in B$
2. $\quad A_i := A + b_i$ /* A_i *is now increasing in* u*-value.* */
3. $C :=$ **merge**$(A_1, \ldots, A_k)$
4. **return** C

where $k = |B|$, and A_i, $1 \leq i \leq k$, are intermediate r-u-pair lists.

Steps 1 and 2 takes $O(|A|\,|B|)$, step 3 takes $O(|A|\,|B| \log |B|)$ if we do it through divide-and-conquer and merge lists in pairs recursively. So *combine_and_merge* is $O(|A|\,|B| \log |B|)$.

The procedure *resource_sieve* trims those r-u-pair elements of list $L = \left\langle \binom{u_{i1}}{r_{i1}}, \ldots, \binom{u_{in}}{r_{in}} \right\rangle$ which do not satisfy $r < r^{\max}$; and those elements that are inefficient. By inefficient we mean: for each element $\binom{u_i}{r_i}$ and element $\binom{u_{i+1}}{r_{i+1}}$ from L, if $r_{i+1} \leq r_i$ (and $u_i \leq u_{i+1}$ since elements are sorted) then $\binom{u_i}{r_i}$ is inefficient and should be removed from L. Intuitively, we only want to keep those choices that use less resource while achieving the same or higher utility. The procedure takes $O(|L|)$.

resource_sieve$(L, r^{\max})$

$i := 1$
while $i < |L|$ **do**
$\quad$**if** $r_{i+1} > r^{\max}$ **then**
$\quad\quad$Remove $\binom{u_{i+1}}{r_{i+1}}$ from L
$\quad$**else**
$\quad\quad$**while** $i \geq 1$ **and** $r_{i+1} \leq r_i$ **do**
$\quad\quad\quad$Remove $\binom{u_i}{r_i}$ from L
$\quad\quad\quad$$i := i - 1$
$\quad\quad$$i := i + 1$
if $r_i > r^{\max}$ **then**
$\quad$Remove $\binom{u_i}{r_i}$ from L
return L.

Procedure *representative_list* trims the r-u-pair list further in $O(|L|)$ by removing elements that are too close to other element in terms of u-value. That is, for each adjacent $\binom{u_i}{r_i}$ and $\binom{u_{i+1}}{r_{i+1}}$ from L, if $(u_{i+1} - u_i)/u_{i+1} \leq \delta$, then $\binom{u_{i+1}}{r_{i+1}}$ can be presented by $\binom{u_i}{r_i}$ with a discrepancy of at most δ w.r.t. the u-value of $\binom{u_{i+1}}{r_{i+1}}$, and therefore $\binom{u_{i+1}}{r_{i+1}}$ can be removed from L.

representative_list(L, δ)

$L' := \left\langle \binom{u_1}{r_1} \right\rangle$
$u^* := u_1$
for i = 2 **to** $|L| - 1$ **do**
    if $(u^* < u_i(1-\delta))$ **then**
        append $\binom{u_i}{r_i}$ to L'
        $u^* := u_i$
return L'

Given the above procedures, the bounded approximation scheme can be constructed as follows. For the sake of simplicity of the complexity analysis to follow, we introduce some intermediate lists L_{ia}, L_{ib} and L_i.

asrmd2$(C_1, ..., C_n, \varepsilon)$

1. $L_0 := \left\langle \binom{0}{0} \right\rangle$
2. $\delta := \varepsilon / n$
3. **for** $i = 1$ **to** n **do**
4. $L_{ia} :=$ **combine_and_merge**(L_{i-1}, C_i)
5. $L_{ib} :=$ **resource_sieve**$(L_{ia}, r^{\max})$
6. $L_i :=$ **representative_list**(L_{ib}, δ)
7. let $\binom{u}{r}$ be the element with the largest utility value in L_n
8. **return** $\binom{u}{r}$

Without *resource_sieve* and *representative_list* the length of the list obtained at step 4 in `asrmd2` could increase exponentially. We will show that with those steps, the length of L_i will be bounded by $\left\lfloor \frac{n \ln(u_{\text{up}}/u_{\text{low}})}{\varepsilon} + 2 \right\rfloor$, where u_{up} and u_{low} are easily determined from C_i.

Lemma 1 *Given two sorted r-u-pair lists A and B,* combine_and_merge *generates a sorted r-u-pair list which contains all the possible combinations of a choice element from A and a choice element from B.*

Proof Since A is sorted, each A_i in step 2 of *combine_and_merge* maintains its order. Moreover A_i contains all new combinations of choices that can be generated by selecting one choice from A and the other as b_i from B. Therefore after the loop at step 1 of *combine_and_merge* finishes, all possible combinations of one element chosen from A and one element chosen from B are stored in A_i, where $1 \leq i \leq |B|$. The merge at step 3 will therefore generate a single combined sorted list. $\square$

Theorem 2 *The approximation of* `asrmd2` *is within a bound of* ε *w.r.t. the optimal.*

Proof If we were not to have the trimming operations *resource_sieve* and *representative_list* in steps 5 and 6 (denote such lists generated without trimming by L_i°), we could prove, based on Lemma 1, by induction on i that *combine_and_merge* at step 4 would list all the possible r-u-pair combinations for i tasks. It would then lead us to an optimal solution at the expense of exponential time complexity in general, since the length of L_i° would grow exponentially.

With trimming that removes from L_i every element that is greater (in terms of r-value) than $r^{\max}$ in step 5, and the trimming in step 6, the property that every remain element in L_i is a member of the complete solution space is maintained. Therefore, the r-u-pair returned in step 7 is indeed one valid allocation scheme. It remains to show that the u-value of the returned pair is not smaller than $1 - \varepsilon$ times an optimal solution.

Since *resource_sieve* at step 5 only throws away invalid elements that violate the *resource constraint,* or those that for sure cannot contribute toward the optimal solution, any error will only be caused by *representative_list.* So it remains to be shown that the relative error caused by *representative_list* is bounded.

When L_i is trimmed by *representative_list,* a relative error of at most δ (or ε/n) is introduced between the representative values remaining in the list and the values before the trimming. By induction on i, it can be shown that for every element $\binom{u^\circ}{r^\circ}$ in L_i° with $r^\circ \leq r^{\max}$, there is an $\binom{u}{r}$ in L_i such that

$$(1 - \varepsilon/n)^i u^\circ \leq u \leq u^\circ.$$

If $\binom{U_{\text{opt}}}{r^*} \in L_i^\circ$ denotes an optimal solution to the SRMD problem, then there is an $\binom{u}{r} \in L_i$ such that

$$(1 - \varepsilon/n)^n U_{\text{opt}} \leq u \leq U_{\text{opt}}$$

The $\binom{u}{r}$ with the largest u is the value returned by *representative_list* and $u = U_{\text{asrmd2}}$. The value of $(1-\varepsilon/n)^n$ increases with n, as it can be shown that

$$\frac{d}{dx}\left(1-\frac{\varepsilon}{x}\right)^x > 0 \quad \text{for } x \geq 1,$$

so that $n > 1$ implies $1-\varepsilon < (1-\varepsilon/n)^n$, and therefore

$$(1-\varepsilon)U_{\text{opt}} \leq U_{\text{asrmd2}} \leq U_{\text{opt}}$$

That is, the result returned by *representative_list* has a maximum relative error of less than ε. □

We will show that the algorithm is of polynomial time complexity. Begin by investigating L_i in *representative_list.* After trimming, successive elements $\binom{u_i}{r_i}$ and $\binom{u_{i+1}}{r_{i+1}}$ of L_i must satisfy $u_i < u_{i+1}(1-\delta)$, that is

$$\frac{u_{i+1}}{u_i} > \frac{1}{1-\delta} \; .$$

as illustrate in Figure 5.2.

Figure 5.2: Successive Elements in L_i After *representative_list*

Let $f = 1/(1-\delta)$ and $K = \lfloor \log_f(u_{\text{up}}/u_{\text{low}}) + 2 \rfloor$, where $u_{\text{up}} > 0$ is the u in step 7 of `asrmd2` and $u_{\text{low}} > 0$ is the smallest utility value, among all tasks, other than 0.

Lemma 2 *There are at most K elements in each L_i of step 6 of* `asrmd2`*.*

Proof Not counting the first element (whose u-value is zero), *representative_list* at step 6 removes elements that differ in u-value from each other by a factor of less than f. Therefore, the number of elements in L_i will be at most

$$\begin{aligned} 1 + \max\{\, k \geq 0 \mid f^k u_{\text{low}} \leq u_{\text{up}} \,\} &= 1 + \lfloor \log_f(u_{\text{up}}/u_{\text{low}}) + 1 \rfloor \\ &= \lfloor \log_f(u_{\text{up}}/u_{\text{low}}) + 2 \rfloor \\ &= K. \end{aligned}$$

□

Theorem 3 *`asrmd2` is a polynomial approximation for SRMD.*

Proof Since steps 4 through 6 in `asrmd2` are all polynomial in the lengths of the lists they handle, and since step 6 by Lemma 2 reduces the number of elements to less than K, it remains to be shown that the number of elements after steps 4 and 5 are bounded.

For step 5 this is trivial since it reduces the number of elements. For step 4, the number of elements grows by a factor of $|C_i|$, so the number of elements after step 4 is bounded by $KC^{\max}$ where

$$C^{\max} = \max_{i=1,\ldots,n} |C_i|$$

The total number of steps in `asrmd2` therefore is bounded by

$$\begin{aligned} cnKC^{\max} &= cnC^{\max} \left\lfloor \log_f(u_{\mathrm{up}}/u_{\mathrm{low}}) + 2 \right\rfloor \\ &= cnC^{\max} \left\lfloor \log_{1/(1-\varepsilon/n)}(u_{\mathrm{up}}/u_{\mathrm{low}}) + 2 \right\rfloor \\ &\leq cnC^{\max} \left\lfloor \frac{n \ln(u_{\mathrm{up}}/u_{\mathrm{low}})}{\varepsilon} + 2 \right\rfloor \end{aligned}$$

for some constant $c > 0$. So the running of `asrmd2` can be obtained as

$$\begin{aligned} O(nL_i L \log L) &= O(nKL \log L) \\ &= O(n^2 \ln \frac{u_{\mathrm{up}}}{u_{\mathrm{low}}} \frac{1}{\varepsilon} L \log L) \\ &= O(n^2 \ln \frac{nu_{\max}}{u_{\min}} L \log L \frac{1}{\varepsilon}) \\ &= O(n^2 L \ln n \log L/\varepsilon) \\ &= O(n^2 L \log n \log L/\varepsilon) \end{aligned}$$

where $u_{\max}$ is the maximum utility and $u_{\min}$ is the minimum utility in the system. Therefore it is polynomial in time in terms of the input n, L and $1/\varepsilon$. And it is clear that the algorithm is polynomial in space as well. □

The analysis of `asrmd2` is, in part, modelled after [15].

5.4 An Optimization Example

Here we present a simple example to illustrate the `asrmd1` algorithm. Figure 5.3 depicts a set of simplified task profiles after the resource-utility structural composition is done (see definition 4.4 and 4.5). In this case, there are eight tasks, each with twenty different quality levels specified, and a total available resource level of 100.

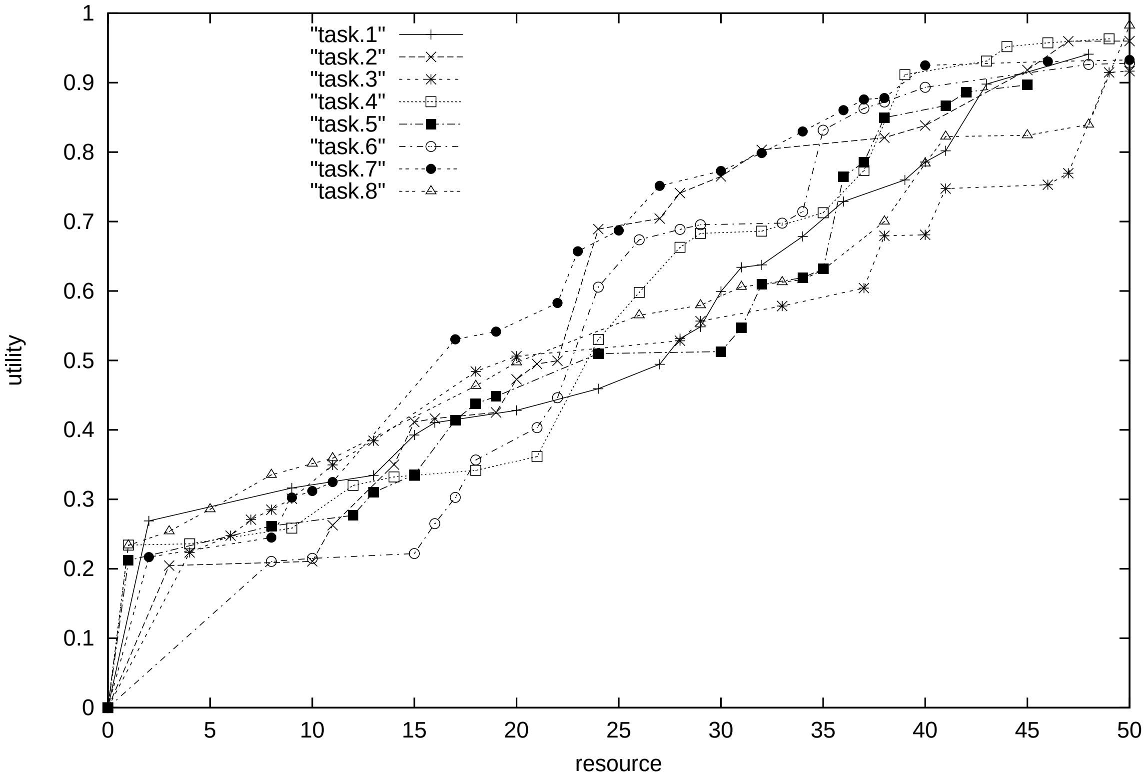

Figure 5.3: Task Profiles: num_tasks=8, $r^{\max}$=100, quality_levels=20

Figure 5.4 plots the approximation data points for each task after the **convex_hull_frontier** procedure is called in `asrmd`. Note the drastic reduction in the number of points. Table 5.1 shows the resource allocation result of both algorithm `asrmd1` and `srmd`, which happens to be exactly the same.

5.5 Chapter Summary

This chapter focused on solving the SRMD (Single Resource, Multiple QoS Dimension) problem. A dynamic programming scheme that obtains the optimal solution is first presented. An approximation algorithm is then presented. This computes the convex hull frontier on the processed scatter graph, and uses a greedy method to

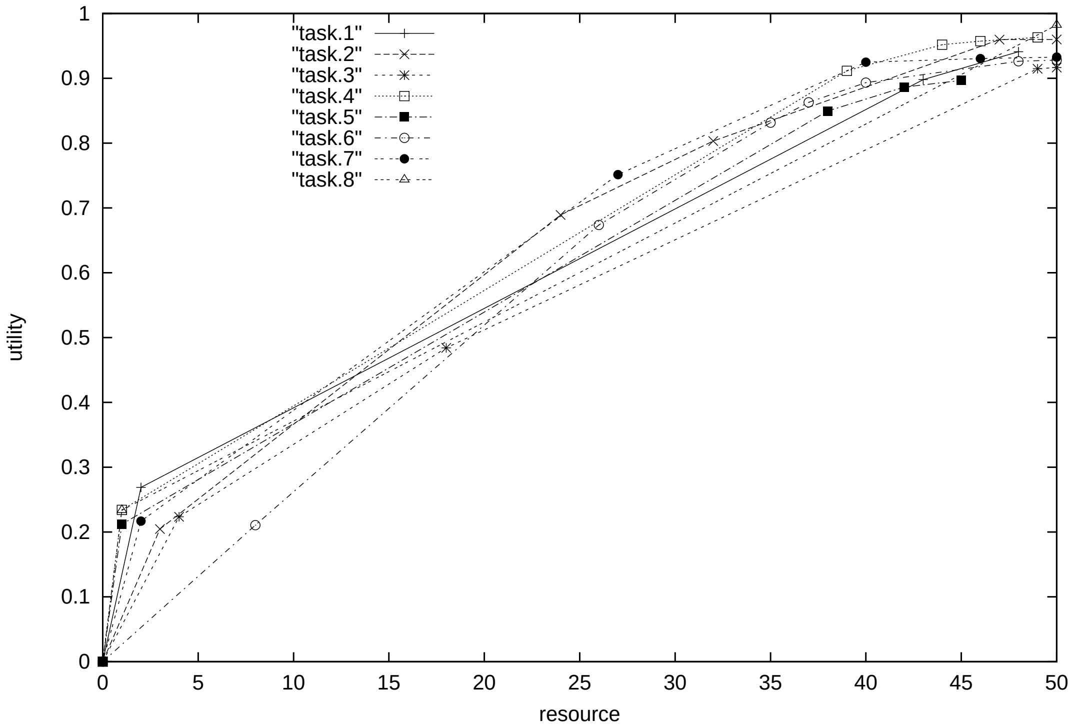

Figure 5.4: Convex Hull Frontier Approximation

traverse the highest utility-per-resource slope at any given allocation step. This algorithm has the property that the distance of this solution from the optimal solution is bounded and that the bound can be easily determined from a set of task profiles. An example of this computationally efficient algorithm is also given to illustrate its key steps. Then, a polynomial approximation scheme that allows the specification of a maximum distance from the optimal solution is presented. This algorithm works by keeping a set of evenly interspersed partial solutions.

We will study the practical performance of these algorithms in Chapter 7.

task_id	asrmd1		srmd	
	resource	utility	resource	utility
1	2	0.2689	2	0.2689
2	24	0.6891	24	0.6891
3	18	0.4842	18	0.4842
4	1	0.2342	1	0.2342
5	1	0.2121	1	0.2121
6	26	0.6738	26	0.6738
7	27	0.7513	27	0.7513
8	1	0.2337	1	0.2337
total	100	3.547	100	3.547

Table 5.1: Example resource allocations of `asrmd1` and `srmd`

Chapter 6

MRMD Algorithms

The problem of maximizing system utility by allocating a *single* finite resource to satisfy discrete Quality of Service (QoS) requirements of multiple applications along multiple QoS dimensions was studied in Chapter 5 and [27]. In this chapter, we consider the more complex problem of apportioning *multiple* finite resources to satisfy the QoS needs of multiple applications along multiple QoS dimensions. In other words, each application, such as video-conferencing, needs multiple resources to satisfy its QoS requirements. We evaluate and compare three strategies to solve this class of problem. We show that dynamic programming and mixed integer programming can be used to compute optimal solutions to this problem but exhibit high complexity. We then adapt the mixed integer programming problem to yield near-optimal results with smaller running times. Finally, we present an approximation algorithm based on a *local search* technique with very low complexity. Perhaps more significantly, the local search technique turns out to be very scalable and robust as the number of resources required by each application increases.

6.1 An Exact Solution Scheme

The solution method and algorithm described in this section can be viewed as an extension of the dynamic programming algorithm described in [27]. The scenario we use to illustrate the algorithm is a two-resource ($m = 2$) case, but the scheme and results described below extend readily to higher dimensions.

The challenge here is to extend the tabular or regular dynamic programming scheme to the case of multiple resources. As in the single resource case, each allocation is in units of size $r_1^{\max}/P_1$ and $r_2^{\max}/P_2$. These represent the smallest possible allocation of each resource type, and $P_i,\ i = 1, 2$ determine the total number of these resource bundles. When $P_1 = P_2 = 100$, for instance, this would mean that allocation is given as an integer percentage of the total resource available.

For the two-resource case, the structure of an optimal solution to the problem can be characterized as follows.

Denote by $v(i, p_1, p_2)$ the maximum utility achievable when only the first i tasks are considered with $r_1^{\max} p_1/P_1$ units of resource R_1 and $r_2^{\max} p_2/P_2$ units of resource R_2 available for allocation. Define the value of an optimal solution recursively in terms of the optimal solutions to subproblems as

$$v(i, p_1, p_2) = \max_{\substack{p'_1 \in \{0,\ldots,p_1\} \\ p'_2 \in \{0,\ldots,p_2\}}} \{g_i(p'_1, p'_2) + v(i-1, p_1 - p'_1, p_2 - p'_2)\} \tag{6.1}$$

In analogy with the single resource case, $v(n, P_1, P_2)$ will be the maximum utility achievable given n tasks and $r^{\max}$ of resources. The set of interesting p'_1 and p'_2 values are the discontinuity points of g_i.

We shall use the following notation in our algorithm. Let

$$C_i = \left\langle \begin{pmatrix} u_{i1} \\ r_{i1} \end{pmatrix}, \ldots, \begin{pmatrix} u_{ik_i} \\ r_{ik_i} \end{pmatrix} \right\rangle$$

list the discontinuity points of g_i, the utility function associated with T_i in increasing u-order. Let $r(i, p_1, p_2)$ contain the corresponding resource allocations that yield $v(i, p_1, p_2)$. Let $qos(i, p_1, p_2)$ be the list of QoS allocations choices for tasks T_1 through T_i that result in $v(i, p_1, p_2)$.

Using the above notation and based on Equation (6.1), an exact algorithm can be constructed for the MRMD problem with discrete resource bundle allocations. As an illustrative example, the following formalizes this algorithm for $m = 2$ and general n assuming that resources have been divided into $P_i,\ i = 1, 2$ bundles.

```
mrmd(n, P1, P2, C1, ..., Cn)
                /* Initialization */
for p1 = 0 to P1 do
    for p2 = 0 to P2 do
        v(0, p1, p2) := 0
        r(0, p1, p2) := 0
        qos(0, p1, p2) := nil
                /* Dynamic programming */
for i = 1 to n do g
    for p1 = 0 to P1 do
        for p2 = 0 to P2 do
            u* := 0
            r* := 0
            j* := 0
            for j = 1 to |Ci| do
                if (r_ij ≰ (p1, p2)) then
                    continue
                u := u_ij + v(i − 1, p1 − r_ij1, p2 − r_ij2)
                if (u > u*) then
                    u* := u
                    r* := r_ij
                    j* := j
            v(i, p1, p2) := u*
            r(i, p1, p2) := r*
            qos(i, p1, p2) := qos(i − 1, p1 − r_ij1, p2 − r_ij2) concat [h_i(r_ij*)]
                /* Unwind and retrieve allocation results */
(p1, p2) := r^max
for i = n downto 1 do
    resource(i) := r(i, p1, p2)
    u(i) := v(i, p1, p2)
    (p1, p2) := (p1, p2) − resource(i)
return v(n, P1, P2), qos(n, P1, P2), resource(1), ..., resource(n),
        u(1), u(2) − u(1), ..., u(n) − u(n − 1)
```

Upon the return of the algorithm `mrmd`, $qos(n, P_1, P_2)$ will contain the QoS values assigned to T_1 through T_n, $utility(i)$ contains the corresponding utility accrued for T_i, and $resource(i)$ gives the resource allocation for T_i. Notice that the resource part in each element of the C_i list above is a vector, and therefore they do not necessarily increase in the resource component.

Let $L = \max_{i=1}^{n} |C_i|$. The computational complexity of the algorithm is then given by $O(nLP_1P_2)$, or $O(nP_1^2P_2^2)$, which is pseudo-polynomial as in the `SRMD` case.

The above algorithm extends in straightforward fashion to m resources with computational complexity $O(nP_1^2 \cdots P_m^2)$, where m is the number of different resources available for allocation. Due to its pseudo-polynomial complexity, we expect that it will have limited use for large-sized on-line systems. However, it can be used for off-line and solution quality evaluation of other heuristic and approximation schemes.

6.2 Integer Programming

Using the problem formulation given in Equation 4.1 of Section 4.2, Integer Programming (IP) algorithms can also be applied. For efficiency reasons, we use the CPLEX [10] `MIP` callable library which employs a branch-and-bound algorithm. In the branch-and-bound method, a series of linear programming (LP) subproblems is solved. A tree of subproblems is built, where each subproblem is a node of the tree. The root node is the LP relaxation of the original IP problem.

To improve the performance of the integer programming with branch-and-bound approach, one can use task priorities and gradients of the dimension-wise quality utility functions as heuristics for developing an integer solution at the root node and for selecting the branching node, the variable and direction. By setting the optimality tolerance (such as the gap between the best result and utility of the best node remaining) or setting limits on time, nodes, memory, etc., one can also obtain fast approximately optimal results.

One drawback of the branch-and-bound technique for solving integer programming problems is that the solution process can continue long after the optimal solution has been found, while the tree is exhaustively searched in an effort to guarantee that the current feasible integer solution is indeed optimal. As we know, the branch-and-bound tree may be as large as 2^n nodes, where n equals the number of binary variables. A problem containing only 30 variables could produce a tree having over one billion nodes.

We shall provide a performance evaluation of this scheme in the next chapter. Its applicability for practical but large MRMD problems is yet to be determined.

6.3 An Approximation Scheme

In this section, we shall define an algorithm that yields near-optimal results but can execute at much higher speeds than the optimal algorithms using dynamic programming or mixed integer programming. We shall use an algorithm that uses a *local search* technique. Recall that n denotes the number of tasks and m denotes the number of resources. Let

$$C_i = \left\langle \begin{pmatrix} u_{i1} \\ r_{i1} \end{pmatrix}, \ldots, \begin{pmatrix} u_{ik_i} \\ r_{ik_i} \end{pmatrix} \right\rangle$$

represent the discrete set of utility-resource pairs for task T_i. Note that in contrast with the SRMD algorithms presented in Chapter 5 and [27] where each $r_{ij}, 1 \leq j \leq k_i$ was a scalar, the resource components, r_{ij}, in C_i are vectors.

To handle the multi-dimensional resource case, it is useful to define a penalty vector to "price" each resource combination. Specifically, let $p = (p_1, \cdots, p_m)$, where $p_i \in [1, \infty)$ be the penalty factor, and $r_p = (r_1 \cdot p_1, \cdots, r_m \cdot p_m)$ be the penalized resource vector. It is useful to define a scalar metric for each penalized resource vector. This metric is denoted r^*. A variety of metrics could be used. For example, r^* can be defined as:

$$r^* = \|r_p\| = \sqrt{(r_{p_1})^2 + \cdots + (r_{p_m})^2}$$

The notion of this virtual or compound resource can be thought of as either the length operator in a space scaled by p, or as similar to aggregate resource concept described in [58].

Once we have defined r^*, we augment C_i by adding this component to obtain:

$$C_{ic} = \left\langle \begin{pmatrix} u_{i1} \\ r_{i1} \\ r^*_{i1} \end{pmatrix}, \ldots, \begin{pmatrix} u_{ik_i} \\ r_{ik_i} \\ r^*_{ik_i} \end{pmatrix} \right\rangle.$$

We now define the algorithm `amrmd1`. In this algorithm, r^c denotes the current remaining resource capacity after some of the available resources have been allocated. $s\_list[i].t$, $s\_list[i].r$, $s\_list[i].u$ contain task ids, their associated r-values and u-values of the corresponding tasks, and r$[i]$ gives the resources currently allocated to T_i.

amrmd1$(n, C_1, \ldots, C_n, r^{\max}, \epsilon)$

```
u* := 0
p := initial_penalty (C_1, ..., C_n, r^max)
repeat := true; count := 3
while repeat and count > 0 do
  repeat := false; count := count − 1
  for i = 1 to n do
    C_ic := compound_resource (C_i, p)
  for i = 1 to n do
    C'_ic := convex_hull_frontier (C_ic)
    r[i] := 0                    // vector assignment
    u[i] := 0
    stop[i] := 0
  s_list= merge(C'_1c, ..., C'_nc)
  r^c := r^max
  for j = 1 to |s_list| do
    i := s_list[j].t
    if (stop[i]) then
      break
    β := s_list[j].r − r[i]      // vector subtraction
    if (β ≤ r^c) then
      r^c := r^c − β
      r[i] := s_list[j].r
      u[i] := s_list[j].u        // update allocation of T_i
    else
      stop[i] := 1
  u := 0
  for i = 1 to n do
```

      $u := u + u^*[i]$
   if $((u - u^*) > \epsilon)$ **then**
      $repeat := true$
      $u^* := u$
      for $i = 1$ **to** n **do**
         $u^*[i] := u[i]$
         $r^*[i] := r[i]$
      $p :=$ **adjust_penalty** $(p, r^c, r^{\max})$
for $i = 1$ **to** n **do**
   $q[i] := h_i(r^*[i])$ // see Equation (4.5)
return $u^*, q[1], \ldots, q[n], r^*[1], \ldots, r^*[i]$

Note that the procedure *convex_hull_frontier* works on the virtual resource portion of each element in C_{ic}.

The procedure *intitial_penalty* calculates the intial penalty vector by looking at the resource usage pattern of all task profiles. Resource dimensions in higher demand are assigned higher penalties.

initial_penalty$(n, C_1, \ldots, C_n, r^{\max})$

$m := \dim r^{\max}$
$rs := 0$
for $i = 1$ **to** n **do**
   for $j = 1$ **to** $|C_i|$ **do**
      $rs := rs + C_i[j].r$ /* *vector addition* */
for $k = 1$ **to** m **do**
   $p_k := rs_k / r_k^{\max} + 1$
return p

The procedure *adjust_penalty* updates the penalty vector using information about the residual resources from the previous iteration.

adjust_penalty$(p, r^c, r^{\max})$

$m := \dim r^{\max}$
for $k = 1$ **to** m **do**

3. $\quad p_k := (p_k * r_k^{\max})/(r_k^c + r_k^{\max}) + 1$
4. **return** p

Many different formulas could have been used here. The key to understanding the one used above is write the formula as

$$\frac{r_k^{\max}}{r_k^c + r_k^{\max}} * p_k + 1$$

where the scale factor applied to p_k is a number between 1/2 and 1. The factor will be 1/2 when the resource is completely unused and grows to 1 when the resource is used up.

By setting ϵ in `amrmd1` to different values, along with the heuristic result from procedure *initial_penalty* and *adjust_penalty*, we can control the solution refinement steps. The asymptotic computational complexity of `amrmd1` can be obtained as follows. Let $L = \max_{i=1}^{n} |C_i|$. The procedure *initial_penalty* takes $O(nL)$ operations. After the procedure *convex_hull_frontier*[1] (which requires $O(nL \log L)$ operations) a convex hull frontier with non-increasing slope segments is obtained for each task. The segments are merged at step 13 using a divide-and-conquer approach with $\log_2 n$ levels, with each level requiring nL comparisons. Merging thus requires $O(nL \log n)$ operations. Steps 15 through 25 require $O(|s\_list|) = O(nL)$. The *adjust_penalty* procedure requires $O(m)$, and steps 27 through 35, 36 through 38 require $O(nL)$. The total running time of the algorithm is, therefore, $O(nL \log L) + O(nL \log n) + O(nL) + O(m) = O(nL \log nL + m)$.

6.4 Related Work

Our MRMD problem can be recast as a multidimensional multiple-choice knapsack problem (MMKP) described in [38]. The authors of [38] describe a heuristic algorithm using the Lagrange multiplier technique for solving MMKP with a complexity of $O(m(n-g)^2 + mn)$ in their notation. It corresponds to $O(m(nL-n)^2 + mnL)$ or $O(mn^2L^2)$ in our notation, as their m, n and g correspond to our m, nL and n respectively.

[1] Overmars & Leeuwen's [42] algorithm, the quickhull [45] or Graham-Scan [7].

The author in [22] treats the MRSD problem (without resource tradeoff) using a heuristic algorithm based on [58]. It is basically a greedy scheme making each step based on current resource consumption. The algorithm also has a complexity of $O(mn^2L^2)$.

Our algorithm described in Section 6.3 is based on *convex_hull_frontier* approximation combined with a search scheme that can be viewed as an extended local search.

Comparing the approximation algorithms in [38] and [22] to ours, we notice that our algorithm has a significantly lower asymptotic complexity, $O(nL \log nL + m)$ versus $O(mn^2L^2)$. As we shall see in Chapter 7, this has been achieved without sacrificing solution quality. As a consequence, we have been able to work with much larger workloads.

6.5 Chapter Summary

This chapter studied solutions to the MRMD (Multiple Resource Multiple QoS Dimensions) problem. As in Chapter 5, *dynamic programming* is used to obtain the optimal solution to this problem. Only the case for two resources is presented, but the same methodology can be used in higher dimensions.

Next, *integer programming* is used to solve the problem by representing each resource trade-off, each QoS tradeoff and the admission of each task as an integer (boolean, in fact) variable. If the integer programming formulation is allowed to run to completion, it will also yield the optimal solution. However, since that can be computationally intensive, it may be terminated when a execution-time-bound and/or an error-bound is reached. In this case, only an approximate solution will result.

Finally, a local search technique is used to combine the various resources needed by a task into a single virtual resource. Then, an optimization technique similar to the convex hull frontier technique from the previous chapter is performed on this virtual resource. This search technique, therefore, is computationally very efficient.

In the following chapter, we will study the practical performance of these algorithms.

Chapter 7

Performance Evaluation

The ideal way to evaluate the performance of our QoS management optimization system would be to subject it to acutal loads from large portfolios of real applications. Such an approach would not work given the current scarcity of QoS-aware applications. Therefore, we have chosen to evaluate the effectiveness of our optimization algorithms by creating a synthetic, but broad, collection of task profiles that are well beyond the limits that a few real applications can provide. Note that the task profiles that we have used for evaluation were not created completely arbitrarily, but chosen to cover the spectrum into which real applications would fall or likely exhibit.

7.1 Experimental Design — Task Profile Characterization

Assume now that the number of tasks, which we will subject our system to, is n. Assume further, that the maximum available resource, $r^{\max}$, is known. With this knowledge, the construction of a test case consists of constructing n task profiles. These task are created independently to represent different applications.

The set of possible task profiles is infinite, we cannot exhaustively test them all. We will therefore work with selected representative cases. We have chosen to select the representative cases randomly. However, care must be taken to ensure that the produced task profiles really are representative of what applications exhibit and that they are rational. By rational we mean that

- the application utility function is non-negative and non-decreasing as a function of quality, and
- quality is non-decreasing with respect to resources, that is $r \models_i q$ respects the partial orderings of R and Q as defined in Section 2.4.

It is relatively simple to arrange for the former condition to be satisfied, but the latter requires much more attention.

In the following two sections, we will describe in detail how we create the two parts of a task profile, and how we preserve the two rationality properties above.

7.1.1 QoS Profile

Recall that a QoS profile consists of

- Quality indices — Q_{ij}, $1 \leq j \leq d_i$.
- Quality space — $Q_i = Q_{i1} \times \cdots \times Q_{id_i}$.
- Application utility $u_i : Q_i \to \mathbb{R}$ which we define as a weighted sum of dimensional utility functions, $u_{ij} : Q_{ij} \to \mathbb{R}$:

$$u_i(q_i) = \sum_{j=1}^{d_i} w_{ij} u_{ij}(q_{ij})$$

Assume that the largest point $q_i^{\text{sysmax}} \in Q_i$ has been given. This point uniquely identifies the quality indices and the quality space. This leaves us with the application utility.

The weights are generated in the following way. Given d_i, the number of quality dimensions of T_i, we will generate d_i real numbers $w'_{i1}, \ldots, w'_{id_i}$, each of them in the range of $[0, 1]$. Each weight can then be obtained as

$$w_{ij} = \frac{w'_{ij}}{\sum_j w'_{ij}}, \qquad j = 1, \ldots, d_i$$

and these weights will sum up to 1.

The requirement that u_i be non-decreasing in all d_i arguments and that it be non-negative can be satisfied by enforcing similar requirements for the dimensional utility functions.

The dimensional utilities of each task are generated using the methods outlined in Chapter 3, that is using the same QoS specification interface structure that we envision the end user would use. Specifically, we create a pool of typical utility function shapes as in Figure 3.3 in Section 3.2. In other words, we have a pool of parameterized function classes — templates that when properly instantiated will yield a dimensional utility function. Example classes include, but are not limited to:

Affine utility with saturation and minimum, as is illustrated in Figure 7.1. The most general is on the right, where utility remains zero until a certain minimum quality, p_0, has been reached. At that point, utility p_2 is gained, and utility increases linearly with quality until it saturates at p_1. This class of utility function describes

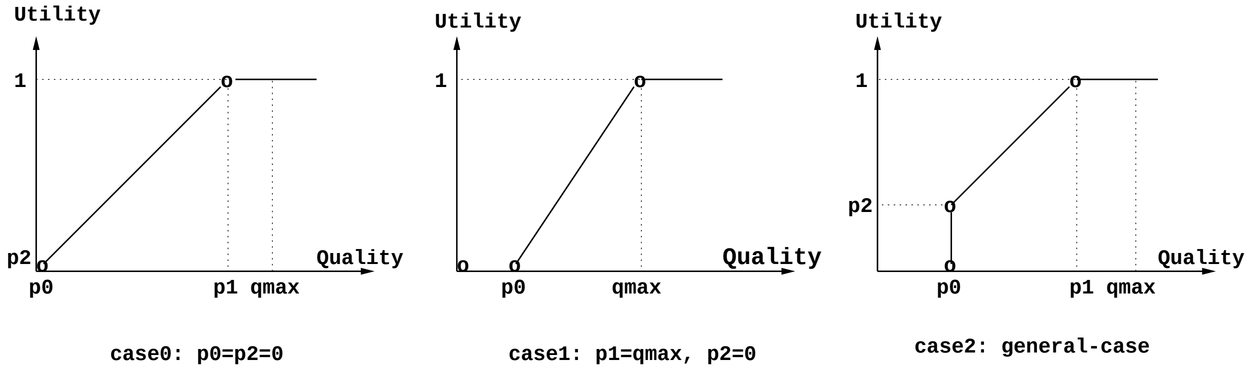

Figure 7.1: Affine Function Class

a "the more, the better" type of quality, such as the one for cryptographic security level in the example of Section 3.4.

Step-function utility a series of quality and utility pairs, as illustrated in Figure 7.2. This class of utility function also describes "the more, the better" type of quality but in a manner well-suited for dimensions (the original dimensions, not the indices) that are of non-numeric nature and dimensions for which the size of the index set is relatively small.

Exponential Decay where the difference between 100% and the achieved utility decreases exponentially, as illustrated in Figure 7.3. This type of utility we imagine

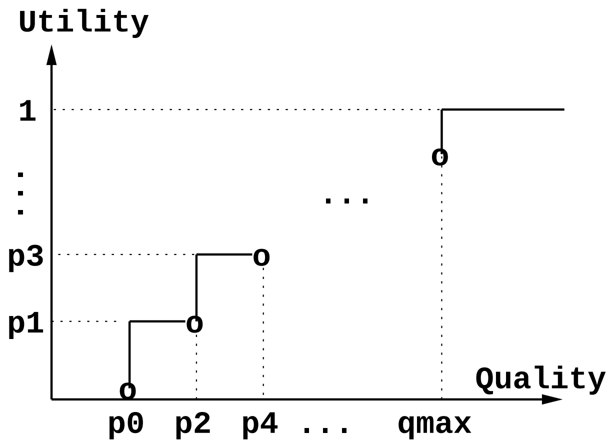

Figure 7.2: Step Function Class

being used for quality dimensions with a diminishing-returns behavior, such as frame rate. The utility gain between 5 fps and 10 fps is much larger than between 25 fps and 30 fps.

7.1.1.1 Function Class Creation

For the **Affine Function Class**: we can define a general function

$$u(q) = \begin{cases} 0 & \text{if } q < p_0 \\ \dfrac{1-p_2}{p_1-p_0} * (q-p_0) + p_2 & \text{if } p_0 \leq q \leq p_1 \\ 1 & \text{if } q > p_1 \end{cases}$$

where p_0 and p_1 are quality indices or zero for which $0 \leq p_0 < p_1 \leq q^{\max}$, and $p_2 \in [0,1]$. The dimensional utility reaches 100% at quality index point p_1.

Note that case 0 and case 1 in Figure 7.1 are two degenerate cases of affine function classes, in which $p_0 = p_2 = 0$ for case 0 whereas $p_1 = q^{\max}$ and $p_2 = 0$ for case 1.
For the **Step Function Class**: we have

$$u(q) = \begin{cases} 0 & \text{if } q < p_0 \\ p_{2i+1} & \text{if } p_{2i} \leq q < p_{2i+2}, \qquad i = 0, 1, \ldots \\ 1 & \text{if } q \geq q^{\max} \end{cases}$$

where p_{2i} are quality indices or zero, and $p_{2i+1} \in [0,1]$, both strictly increasing with respect to their subscripts.

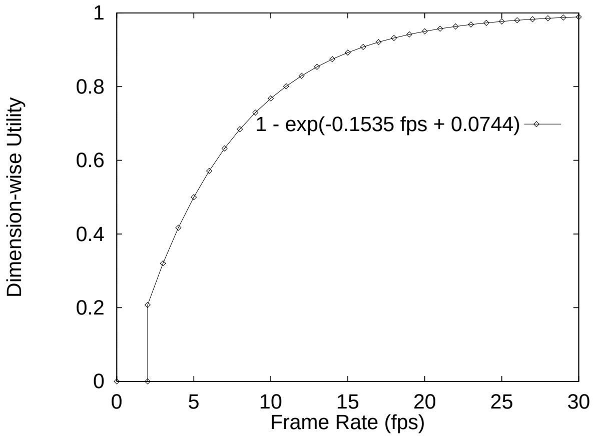

Figure 7.3: Exponential Decay Function Class

For **Exponential Decay**: we have

$$u(q) = \begin{cases} 0 & \text{if } q < q^{\min} \\ 1 - e^{p_0 * q + p_1} & \text{if } q \geq q^{\min} \end{cases}$$

where p_0 and p_1 are real numbers (not necessarily positive). To make $u(q)$ non-decreasing, we need to have $p_0 \leq 0$. Further, in order to have $0 \leq u(q) \leq 1$, we need to ensure that $e^{p_0 * q + p_1} \leq 1$, therefore $p_0 * q + p_1 \leq 0$. That is, $p_1 \leq -(p_0 * q)$ for all $q \in [1..q^{\text{sysmax}}]$, which is equivalent to requiring $p_1 \leq -(p_0 * q^{\min})$, that is $p_1 \leq -p_0$.

Now we can describe the algorithm that creates the function classes for QoS profiles.

$\mathbf{F}(j, q_{ij}^{\max})$ /* *generate function for QoS dimension j with $Q_{ij} = \{1, \ldots, q_{ij}^{max}\}$* */

$F.class := random\_choice$ (Affine, Step, ..., Exponential_Decay)
switch (F.class)
case Affine:
  $subtype := random\_int\ (0, 2)$ /* *favor the 3 cases equally* */
  switch (*subtype*) /* *set the corresponding p_0, p_1 and p_2* */
  case 0:
    $F.p_0 := 0$
    $F.p_1 := random\_int\ (0, q_{ij}^{\max})$
    $F.p_2 := 0$
    break

```
    case 1:
        F.p_0 := random_int (1, q_ij^max)
        F.p_1 := q_ij^max
        F.p_2 := 0
        break
    case 2:
        a := random_int (1, q_ij^max)
        repeat
            b := random_int (1, q_ij^max)
        until a ≠ b
        F.p_0 := min(a, b)
        F.p_1 := max(a, b)
        F.p_2 := random_real (0, 1)
    break
case STEP:
    if q_ij^max = 1 then                                /* binary function */
        F.num_param := 1
        F.p_0 := 1
    else
        num_steps := random_int (0, q_ij^max − 1)
        F.num_param := 2 ∗ num_step + 1
        L_1 := [ ]
        repeat
            i := random_int (1, q_ij^max)
            if i ∉ L_1 then
                L_1 := L_1 concat [i]
        until |L_1| = num_step + 1
        L_1 := sort (L_1)
        L_2 := [ ]
        repeat
            r := random_real (0, 1)
            if r ∉ L_2 then
                L_2 := L_2 concat [r]
```

```
        until |L_2| = num_step
        L_2 := sort (L_2)
        for i = 0 to num_step - 1 do
            F.p_{2i} := L_1[i]
            F.p_{2i+1} := L_2[i]
        F.p_{2*num_step} := L_1[num_step]
    break
case ... :                                   /* other function classes */
    ⋮
    ⋮
case EXPONENTIAL_DECAY:
    F.num_param := 2
    F.p_0 := random_real (-1, 0)
    repeat
        F.p_1 := random_real (0, 1)
    until F.p_1 <= -F.p_0
    break
return F
```

Given a utility function F which was instantiated from some class, the QoS dimension j and the quality index $q_{ij} \in Q_{ij}$, we can now calculate the dimensional utility value of T_i on q_{ij}.

F::fv(q_{ij})

```
switch (F.class)
case AFFINE:
    if q_ij < F.p_0 then
        rfv := 0
    else if q_ij <= F.p_1 then
        rfv := (1 - F.p_2)/(F.p_1 - F.p_0) * (q_ij - F.p_0) + F.p_2
    else
        rfv := 1
    break
case STEP:
```

```
    if q_ij < F.p_0 then
        rfv := 0
    else
        rfv := 1
        for i = 2 to F.num_param step 2
            if q_ij < F.p_i then
                rfv := F.p_{i-1}
                break
    break
case ... :                                        /* other function classes */
    ⋮
    ⋮
case EXPONENTIAL_DECAY:
    rfv := 1 − e^{F.p_0 * q_ij + F.p_1}
    break
return rfv
```

7.1.2 Resource Profile

A resource profile is much more difficult to create than QoS Profile. In the context of the resource profiles of our QoS management model, a resource profile is representative and rational means that we must ensure:

1. Resource usage is non-negative;
2. Resource profiles exhibit resource trade-off; and
3. Higher resource consumption will not result in lower quality.

Enforcing the first property is trivial; the second, as will be shown shortly, will be satisfied by the construction scheme that we use; the last property, however, is initially challenging, but we will prove that it follows from the second property when combined with some reasonable assumptions.

Before we devise the resource profile generation algorithm to satisfy these conditions, let us introduce some symbols and notations that will help to formulate and validate the steps taken to fulfil the properties imposed on the typical resource profiles.

We shall assume the resource tradeoff comes from the use of a set of schemes programmed within an application. Let N_{T_i} represent the number of resource tradeoff schemes for a T_i. Denote by $f_{ij} : Q_i \rightarrow R$, where $j = 1, \ldots, N_{T_i}$, the resource usage function in effect when task T_i is using scheme j. Define $f_{ijk} = \pi_k \circ f_{ij}$ the resource usages on m individual resources respectively using trade-off scheme j, where $j = 1, \ldots, N_{T_i}$, $k = 1, \ldots, m$, and π_k is the projection of the resource space into its kth dimension.

For practical reasons, we will assume $N_{T_i} = 2$ from now on. That is, there will be two ways of trading off resources to realize a task's quality space. In other words, for any given $q \in Q_i$, we will have two data points, (q, r) and (q, r'), that describe the usage of m resources to realize each quality point. Therefore the resource profile $r \models_i q$ of task T_i could be described as:

$$\begin{aligned} r \models_i q &= \{(q, f_{i1}(q)) \mid q \in Q_i\} \cup \{(q, f_{i2}(q)) \mid q \in Q_i\} \\ &= \bigcup_{q \in Q_i} \{(q, f_{i1}(q)), (q, f_{i2}(q))\} \\ &= \bigcup_{q \in Q_i} \Big\{\Big(q, \langle f_{i11}(q), \ldots, f_{i1m}(q)\rangle\Big), \Big(q, \langle f_{i21}(q), \ldots, f_{i2m}(q)\rangle\Big)\Big\} \end{aligned}$$

With these notations, we can now formalize the three properties above as conditions on the individual functions f_{ij} or $\pi_k \circ f_{ij}$:

$$\forall q, j : f_{ij}(q) \geq 0 \tag{7.1}$$

$$\forall q \in Q_i, \exists s, t : (f_{i1s}(q) > f_{i2s}(q)) \wedge (f_{i1t}(q) < f_{i2t}(q)) \tag{7.2}$$

$$\forall q_s, q_t \in Q_i : \forall j_1, j_2 : (f_{ij_1}(q_s) > f_{ij_2}(q_t)) \rightarrow \neg(q_s < q_t) \tag{7.3}$$

We call a resource profile $r \models_i q$ **rational** if it satisfies condition (7.3). Note that many of the above comparisons ("$\geq$", "$>$" and "$<$") in (7.1) through (7.3) are between vectors, not simple numbers. For example, $\neg(q_s < q_t)$ means that either q_s and q_t are not comparable, or q_s is bigger than or equal to q_t. That is, $\neg(q_s < q_t)$ is not the same as $q_s \geq q_t$. The comparisons on f_{ijk} in condition (7.2), on the other hand, are scalar.

Theorem 4 *$r \models_i q$ is rational if f_{ijk} are non-decreasing and satisfy condition (7.2).*

Proof Let q_s and q_t be given. We will consider three cases:

Case 1: If the two quality points q_s and q_t are not comparable, then $q_s < q_t$ is false, so $\neg(q_s < q_t)$ is true and condition (7.3) is satisfied.

Case 2: If the two quality points q_s and q_t are equal, then condition (7.3) is trivially satisfied.

Case 3: Assume that two quality points q_s and q_t are comparable, but not equal. Without loss of generality let us assume that $q_s < q_t$. We need to prove that $\neg(f_{ij_1}(q_s) \geq f_{ij_2}(q_t))$, i.e. either (a) $f_{ij_1}(q_s)$ and $f_{ij_2}(q_t)$ are not comparable or (b) $f_{ij_1}(q_s) < f_{ij_2}(q_t)$. We will prove this by contradiction. Assume that $f_{ij_1}(q_s) > f_{ij_2}(q_t)$, where $j_1, j_2 \in \{1, 2\}$.

Since f_{ijk} are non-decreasing functions,

$$\forall i, j, k, q_s, q_t : (q_s \leq q_t) \rightarrow (f_{ijk}(q_s) \leq f_{ijk}(q_t))$$

which is the same as

$$\forall i, j, q_s, q_t : (q_s \leq q_t) \rightarrow (\forall k : f_{ijk}(q_s) \leq f_{ijk}(q_t))$$

that is,

$$\forall i, j, q_s, q_t : (q_s \leq q_t) \rightarrow (f_{ij}(q_s) \leq f_{ij}(q_t))$$

Therefore we have $f_{ij_2}(q_s) \leq f_{ij_2}(q_t)$. Together with our assumption, $f_{ij_1}(q_s) > f_{ij_2}(q_t)$, this means that $f_{ij_1}(q_s) > f_{ij_2}(q_s)$.

From this, we first conclude that $j_1 \neq j_2$ and that one of them thus is 1 while the other is 2. Then, from condition (7.2), we conclude that $f_{i1}(q_s)$ and $f_{i2}(q_s)$ are non-comparable — a contradiction to $f_{ij_1}(q_s) > f_{ij_2}(q_t)$. Our assumption must therefore be false, and the rational condition (7.3) for $r \models_i q$ holds. □

Given the above proposition, we are ready to construct the algorithm for proper resource profile generation.

Let $FC = \{F_1, F_2, ...\}$ be the function class set with a finite number of elements, Nc_i be the number of coefficient for F_i. Let $CE_{ij}, j = 1, \ldots, Nc_i$ be the domain set for the jth coefficient of F_i, CE_i be the set of CE_{ij}, and CE be the set of CE_i, and f_i be the function with F_i being fully instantiated with its coefficient(s) chosen from CE.

We omit the description of the creation of each function class in FC here. The process is similar to the function class generation in QoS profile except that the different kind of function shapes are created, for example, exponential instead of exponential decay, since when quality increasing the rate of resource consumption often is not decreasing.

```
resource_profile(FC, CE, m)
                  /* generate vector function f_ij (consists of scalar functions f_ijk) */
for j = 1 to 2 do {
  for k = 1 to m do {
    randomly pick a function class, say F_l, from FC
    for p = 1 to Nc_l do
      randomly pick the pth coefficient from CE_lp
    let f_ijk be the function with F_l fully instantiated
    while ¬ non_decreasing (f_ijk) do {
      p := random_int (1, Nc_l)
      replace the pth coefficent of f_ijk with a new value
    }
  }
}
repeat {                 /* make sure that the f_ij exhibits resource tradeoff */
  done_tradeoff_all := true
  foreach q ∈ Q_i do {
    done_tradeoff_one := false
    s := 1
    while ¬done_tradeoff_one and s ≤ m − 1 do {
      t := s + 1
      while ¬done_tradeoff_one and t ≤ m do {
        if (f_i1s(q) > f_i2s(q) and f_i1t(q) < f_i2t(q)) or
              (f_i1s(q) < f_i2s(q) and f_i1t(q) > f_i2t(q))
          done_tradeoff_one := true
        t := t + 1
      }
      s := s + 1
    }
```

if ¬*done_tradeoff_one* **then** {
  done_tradeoff_all := false
  break /* *foreach loop* */
}
} /* *foreach loop* */
if ¬*done_tradeoff_all* **then**
  repeat { /* *might need to regen functions, not just coeff* */
    j := *random_int* $(1, 2)$
    k := *random_int* $(1, m)$
    p := *random_int* $(1, Nc_l)$
    replace f_{ijk}'s pth coefficient with a new value
  } **until** *non_decreasing* (f_{ijk})
} **until** *done_tradeoff_all*

7.1.3 Sample Task Profile

Below is the task profile for T_{11}, a randomly picked sample. T_{11} has three quality dimensions in concern, and its $q^{\min}$ and $q^{\max}$ is $\langle 0, 0, 0\rangle$ and $\langle 4, 3, 4\rangle$ respectively. Stanza form is the system internal representation of a task profile, whereas each line in the vanilla form gives the detaildescription on the utility and resource consumption of each quality point. For example,

```
<1,1,1> 0.507014 [<9,5>,<5,15>]
```

shows that quality level $\langle 1, 1, 1\rangle$ brings a utility of 0.507014. Moreover, the quality can be realized in either 9 units of resource 1 and 5 units of resource 2, or 5 units of resource 1 and 15 units of resource 2. Note that there is resource tradeoff between the two resources.

```
Task Profile in stanza form:

{TASK_Profile: tid = 11
  qmin = <0,0,0> qmax = <4,3,4>
  {Application_Profile:
    [QoS_Profile: 3
      [QoS_Profile_Dim: 0.3031 4 <0.7697,0.8849,1,1>]
```

```
  [QoS_Profile_Dim: 0.3745 3 <0,1,1>]
  [QoS_Profile_Dim: 0.3224 4 <0.849,0.849,0.849,1>]
]
[Resource_Profile: <4,3,4>
  [<9,5>,<5,15>]
  [<13,5>,<6,17>]
  [<17,6>,<7,19>]
  [<22,6>,<8,21>]
  [<11,5>,<5,16>]
  [<14,6>,<6,18>]
  [<19,6>,<7,20>]
  [<26,7>,<8,22>]
  [<12,6>,<5,17>]
  [<17,6>,<6,19>]
  [<22,7>,<7,21>]
  [<29,8>,<8,23>]
  [<12,6>,<5,16>]
  [<16,6>,<6,18>]
  [<21,7>,<7,20>]
  [<28,8>,<8,22>]
  [<14,6>,<5,17>]
  [<18,7>,<6,19>]
  [<24,8>,<7,21>]
  [<32,9>,<8,23>]
  [<16,7>,<5,18>]
  [<21,8>,<6,20>]
  [<27,9>,<7,22>]
  [<36,9>,<8,24>]
  [<15,7>,<6,17>]
  [<20,8>,<7,19>]
  [<26,9>,<8,21>]
  [<35,10>,<9,23>]
  [<17,8>,<6,18>]
  [<23,9>,<7,20>]
  [<30,10>,<8,22>]
```

```
      [<40,11>,<9,24>]
      [<19,9>,<6,19>]
      [<26,10>,<7,21>]
      [<34,11>,<8,23>]
      [<45,12>,<9,25>]
      [<18,9>,<6,18>]
      [<25,10>,<7,20>]
      [<33,11>,<8,22>]
      [<43,12>,<9,24>]
      [<21,10>,<6,19>]
      [<28,11>,<7,21>]
      [<37,12>,<8,23>]
      [<49,13>,<9,25>]
      [<24,11>,<6,20>]
      [<32,12>,<7,22>]
      [<42,13>,<8,24>]
      [<56,14>,<9,26>]
    ]
  }
}

Translated Task Profile in vanilla form:

<1,1,1> 0.507014 [<9,5>,<5,15>]
<1,1,2> 0.507014 [<13,5>,<6,17>]
<1,1,3> 0.507014 [<17,6>,<7,19>]
<1,1,4> 0.555696 [<22,6>,<8,21>]
<1,2,1> 0.881514 [<11,5>,<5,16>]
<1,2,2> 0.881514 [<14,6>,<6,18>]
<1,2,3> 0.881514 [<19,6>,<7,20>]
<1,2,4> 0.930196 [<26,7>,<8,22>]
<1,3,1> 0.881514 [<12,6>,<5,17>]
<1,3,2> 0.881514 [<17,6>,<6,19>]
<1,3,3> 0.881514 [<22,7>,<7,21>]
<1,3,4> 0.930196 [<29,8>,<8,23>]
```

```
<2,1,1> 0.541931 [<12,6>,<5,16>]
<2,1,2> 0.541931 [<16,6>,<6,18>]
<2,1,3> 0.541931 [<21,7>,<7,20>]
<2,1,4> 0.590613 [<28,8>,<8,22>]
<2,2,1> 0.916431 [<14,6>,<5,17>]
<2,2,2> 0.916431 [<18,7>,<6,19>]
<2,2,3> 0.916431 [<24,8>,<7,21>]
<2,2,4> 0.965113 [<32,9>,<8,23>]
<2,3,1> 0.916431 [<16,7>,<5,18>]
<2,3,2> 0.916431 [<21,8>,<6,20>]
<2,3,3> 0.916431 [<27,9>,<7,22>]
<2,3,4> 0.965113 [<36,9>,<8,24>]
<3,1,1> 0.576818 [<15,7>,<6,17>]
<3,1,2> 0.576818 [<20,8>,<7,19>]
<3,1,3> 0.576818 [<26,9>,<8,21>]
<3,1,4> 0.6255 [<35,10>,<9,23>]
<3,2,1> 0.951318 [<17,8>,<6,18>]
<3,2,2> 0.951318 [<23,9>,<7,20>]
<3,2,3> 0.951318 [<30,10>,<8,22>]
<3,2,4> 1 [<40,11>,<9,24>]
<3,3,1> 0.951318 [<19,9>,<6,19>]
<3,3,2> 0.951318 [<26,10>,<7,21>]
<3,3,3> 0.951318 [<34,11>,<8,23>]
<3,3,4> 1 [<45,12>,<9,25>]
<4,1,1> 0.576818 [<18,9>,<6,18>]
<4,1,2> 0.576818 [<25,10>,<7,20>]
<4,1,3> 0.576818 [<33,11>,<8,22>]
<4,1,4> 0.6255 [<43,12>,<9,24>]
<4,2,1> 0.951318 [<21,10>,<6,19>]
<4,2,2> 0.951318 [<28,11>,<7,21>]
<4,2,3> 0.951318 [<37,12>,<8,23>]
<4,2,4> 1 [<49,13>,<9,25>]
<4,3,1> 0.951318 [<24,11>,<6,20>]
<4,3,2> 0.951318 [<32,12>,<7,22>]
<4,3,3> 0.951318 [<42,13>,<8,24>]
```

```
<4,3,4> 1 [<56,14>,<9,26>]
```

7.2 Practical Performance Evaluation of SRMD Algorithms

In Chapter 5, we presented the theoretical behavior of the SRMD algorithms. We will now examine their practical performance. We compare actual computation cost in terms of running time, and solution quality with respect to optimum.

In our experiments, each algorithm is fed the same series of workloads. The parameters for each workload consists of:

- Number of tasks (ranging from 8 to 1024).
- Number of quality levels (ranging from 8 to 128)
- Total available system resources (ranging from 100 to 1000000 units)

Note that the number of quality levels is specified in terms of utility value, which is less than or equal to the number of quality points. The point with the highest utility is taken when the same resource allocation supports multiple quality points.

7.2.1 Comparative Evaluation of `asrmd1` and `srmd`

We note that all experiments were conducted on a 300 MHz Pentium machine with 192 MB running RedHat Linux.

We now present a series of experiments conducted to compare the run-time efficiency and solution quality of `asrmd1` relative to the optimal `srmd` algorithm. Recall that the three main variables among the parameters are:

- Number of tasks (num_tasks: ranging from 8 to 1024).
- Number of quality levels (quality_levels: ranging from 8 to 128).
- Total available system resources (rmax: ranging from 100 to 1000000 units).

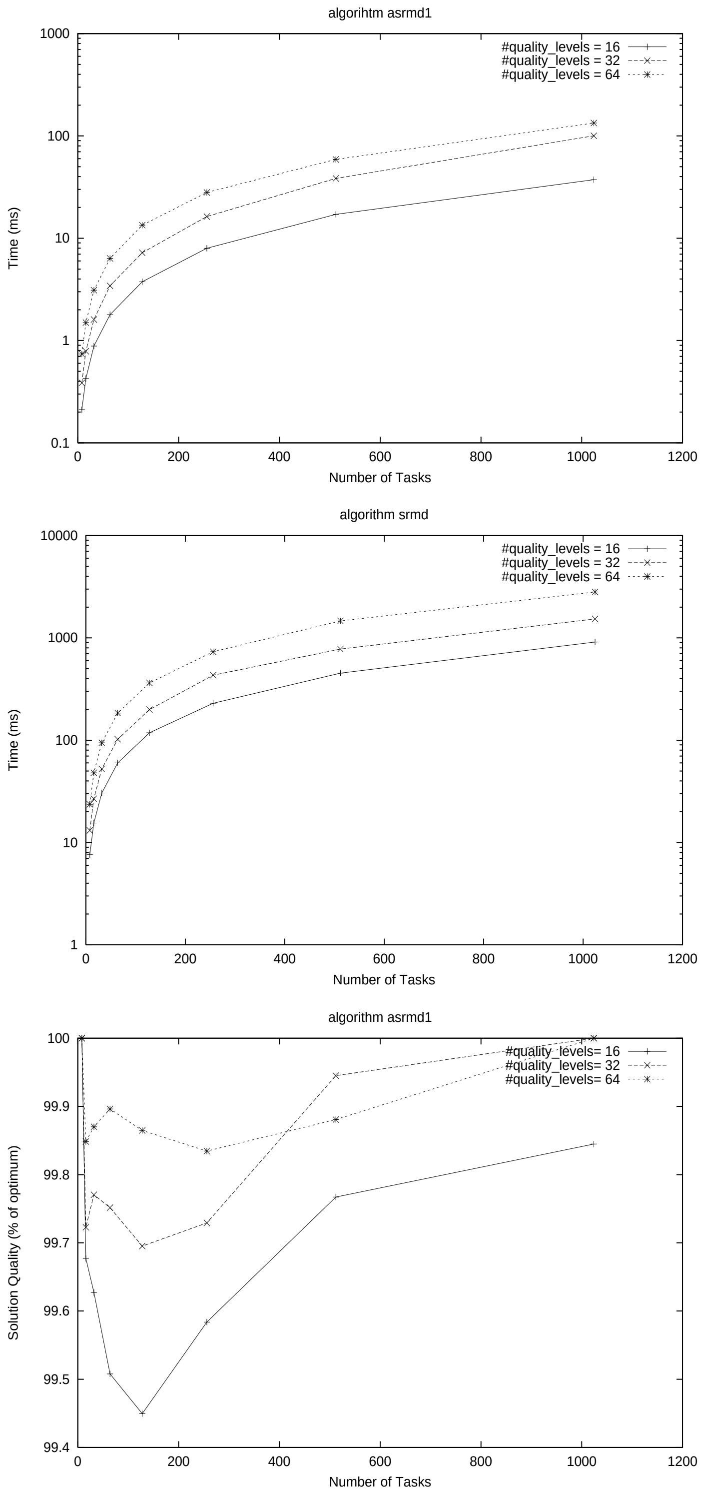

Figure 7.4: Run Time and Solution Quality: `asrmd1` vs `srmd`, with $r^{\max}$=800

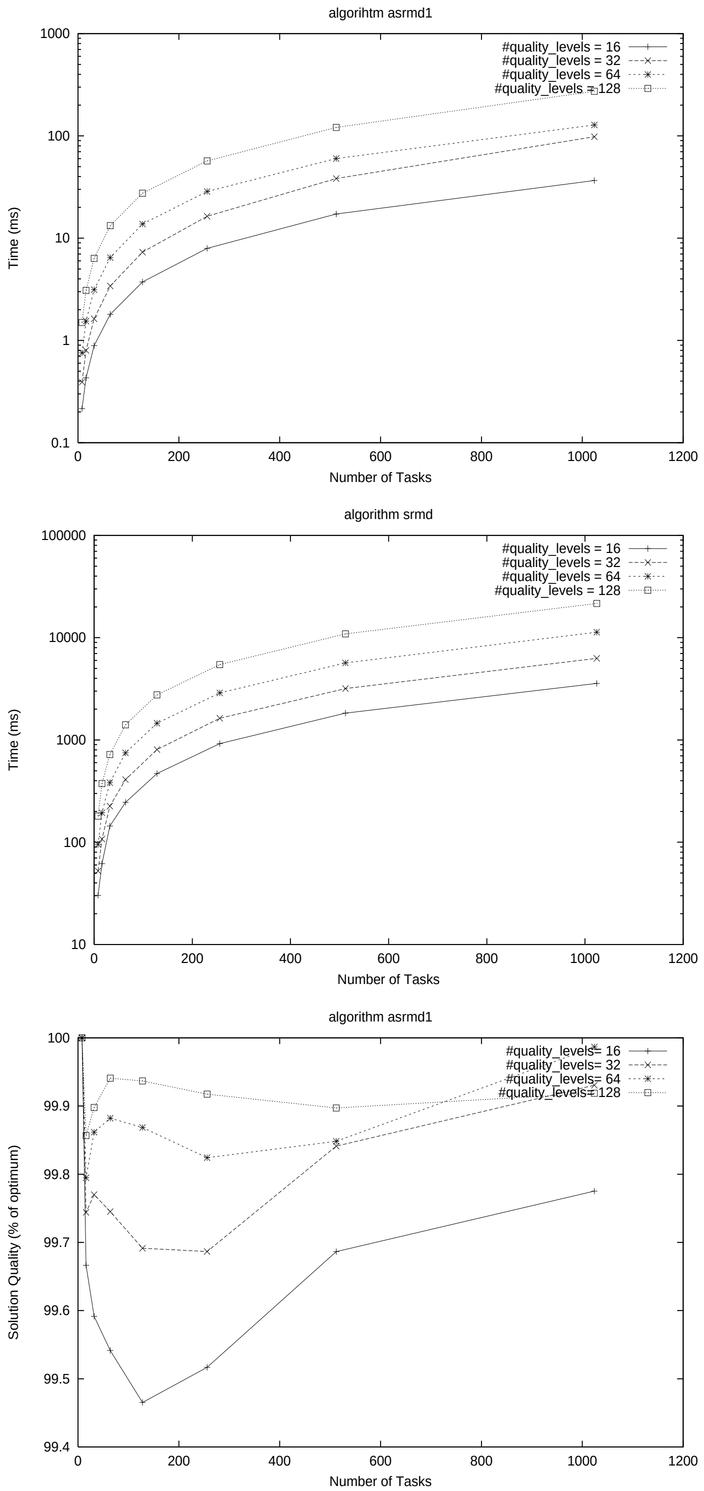

Figure 7.5: Run Time and Solution Quality: `asrmd1` vs `srmd`, with $r^{\max}$=3200

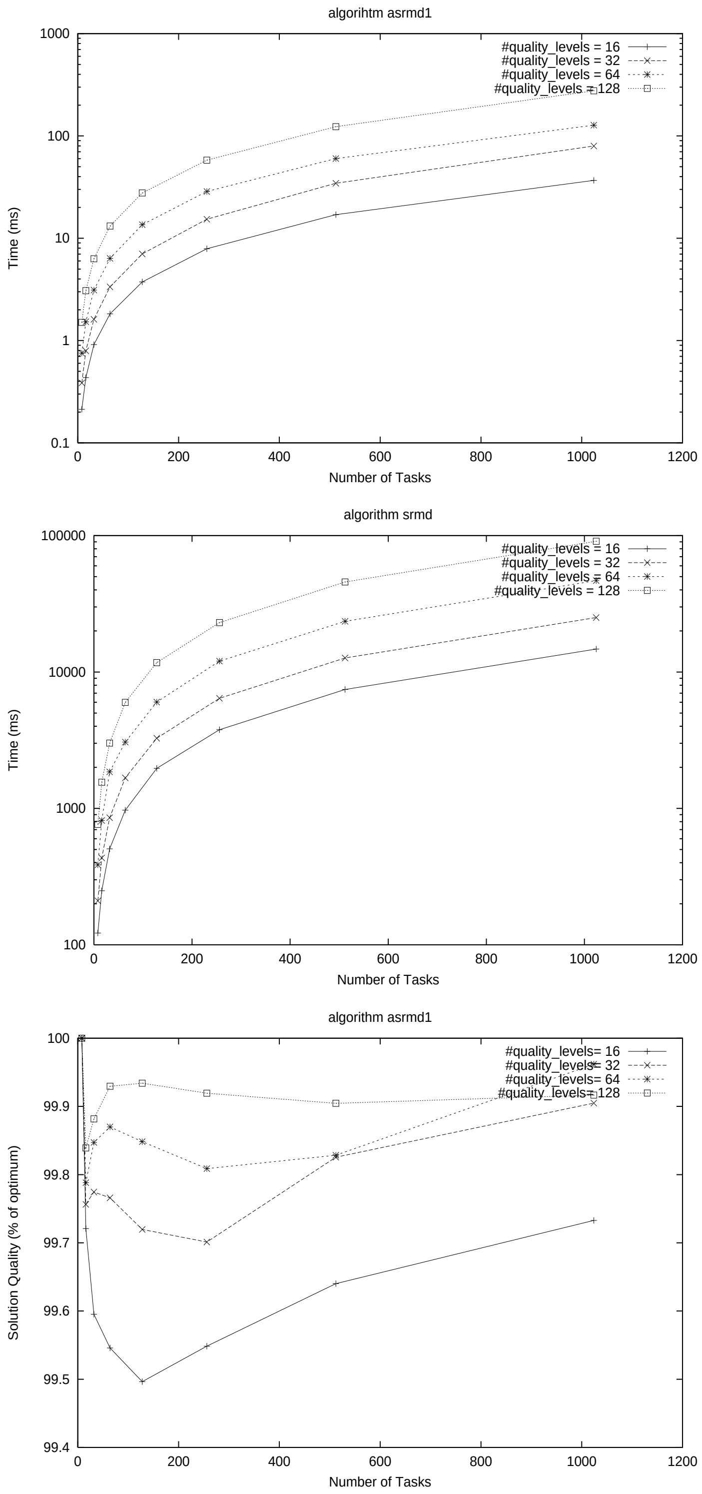

Figure 7.6: Run Time and Solution Quality: `asrmd1` vs `srmd`, with $r^{\max}$=12800

Alternatively, we could think of $r^{\max}$ in terms of the precision of the resource allocation for the `srmd` algorithm. When $r^{\max} \geq 100$, `srmd` can give fractional resource allocations. For example, $r^{\max} = 10000$ corresponds to a precision of one-hundredth of total capacity or bandwidth. While `asrmd1` and `asrmd2` handle non-integral resource allocation without any added computational complexity, the computation time of `srmd` increases as the granularity of resource allocation increases.

Figures 7.4 through 7.6 present the run-times of algorithms `asrmd1` and `srmd`, and the solution quality obtained by algorithm `asrmd1` relative to `srmd`, which represents the optimal solution. Each figure presents three graphs each. The first and third graphs plot the run-times (in millisecond) for algorithms `asrmd1` and `srmd` respectively as the number of tasks in the system is increased. The second graph plots the solution quality of algorithm `asrmd1` relative to the optimal solution obtained by `srmd`. Figure 7.4 assumes that $r^{\max}$=800 and 16 QoS options per task. Similarly, Figures 7.5 and 7.6 respectively assume $r^{\max} = 3200$, 16/32/64 QoS levels and $r^{\max} = 12800$, 16/32/64/128 QoS levels. Each workload in these experiments was repeated 100 times, each with *num_task* number of different tasks with random QoS options and generated such that we could examine the solution quality of approximation algorithms in a broad range of scenarios.

A couple of behaviors can be easily observed in each of the graphs. The run-times increase as the number of tasks increases (e.g. see Figure 7.4, Figure 7.5, and Figure 7.6). The run-times also increase as the number of QoS options per task increases. But notice that the run-times increase as the $r^{\max}$ increases (e.g. compare Figures 7.4.[bc] and 7.5.[bc]) only for algorithm `srmd`, whereas `asrmd1` has a running time that is independent of $r^{\max}$.

Returning to points of interest, notice how algorithm `asrmd1` consistently runs about an order of magnitude faster than the exact algorithm `srmd` in Figures 7.4 through 7.6. The difference approaches two orders of magnitude when the granularity of resource allocation is finer in Figures 7.5 and 7.6. Notice further that the average solution quality for algorithm `asrmd1` in the second graph of each figure stays above 99% for most cases. Since the plotted values are the averages over 100 runs, the worst case obviously is lower. However, in general, it is easy to conclude that the approximation algorithm `asrmd1` exhibits excellent behavior in achieving near-optimal results within a small fraction of time needed to find the optimal solution.

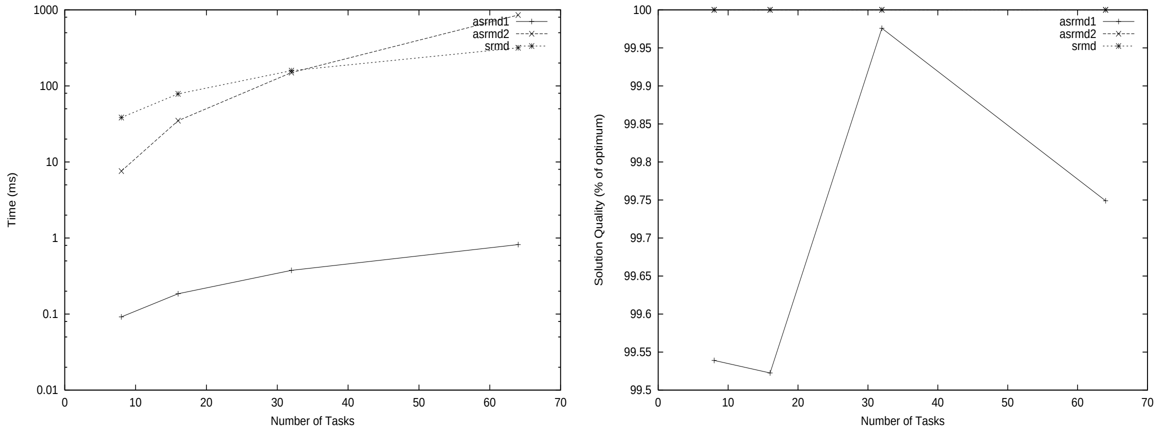

Figure 7.7: Run Time and Solution Quality: `asrmd1` vs `asrmd2` vs `srmd` with $r^{\max}$=10000, $\varepsilon = 0.01$, and num_quality_levels = 16

7.2.2 Comparative Evaluation of `asrmd1` and `asrmd2`

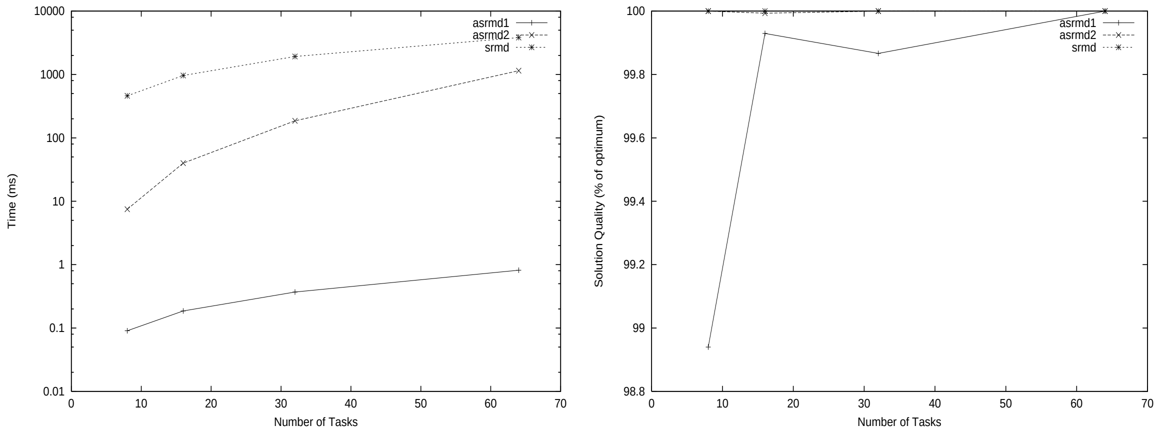

Figure 7.8: Run Time and Solution Quality: `asrmd1` vs `asrmd2` vs `srmd` with $r^{\max}$=100000, $\varepsilon = 0.01$, and num_quality_levels = 16

We conducted a second series of experiments to compare the relative performances of algorithms `asrmd1` and `asrmd2`. In these set of experiments, we fixed the number of QoS options per task to be 16 in each run, and ε was chosen to be a constant 0.01 (i.e. the desired quality obtained by `asrmd2` must be within 1% of the optimal solution). The run-times and solution qualities of the two approximation algorithms

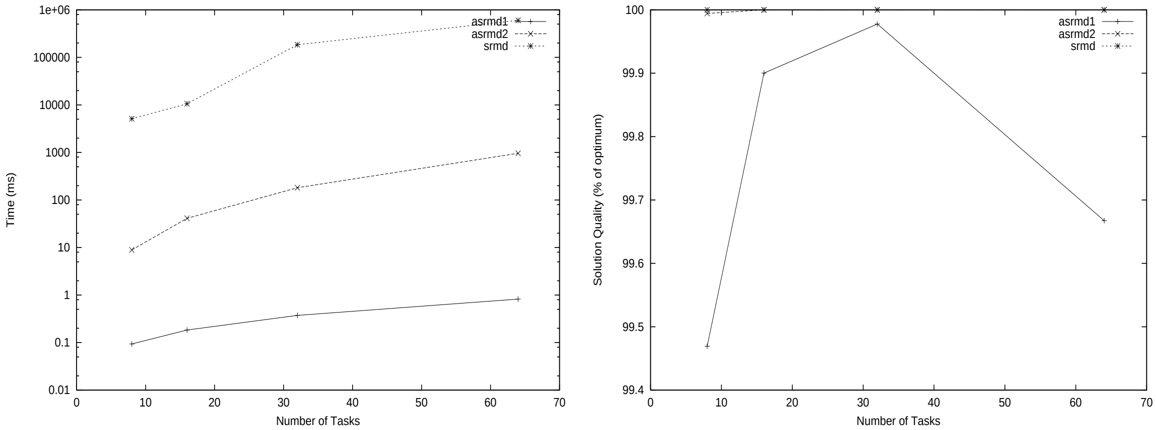

Figure 7.9: Run Time and Solution Quality: `asrmd1` vs `asrmd2` vs `srmd` with $r^{\max}$=1000000, $\varepsilon = 0.01$, and num_quality_levels = 16

along with the optimal `srmd` algorithm were measured with $r^{\max} = 1000$, 100000 and 1000000. The resulting graphs are plotted in Figures 7.7 through 7.9 respectively. The first of the two graphs in each figure plots the run-times of the three algorithms as the number of tasks is increased.[1] The second graph shows each algorithm's relative solution quality compared to the optimal solution.

As discussed earlier, algorithm `asrmd2` is very promising from a theoretical point of view: it always delivers a guaranteed solution quality in polynomial time. Unfortunately, its actual running time is up to two orders of magnitude more than that for `asrmd1` (e.g. see Figure 7.8.a). The solution quality graphs plot the solution quality of algorithms `asrmd1` and `asrmd2`. They show that `asrmd1` is mostly within 1% of the optimal solution while `asrmd2`, which must always be within 1%, on the average yields a solution very close to the optimal solution. However, we believe that the difference between the two run-times is relatively high, particularly when the solution quality obtained by `asrmd1` is very good.

Based on the above two sets of experiments, we conclude that the `asrmd1` algorithm using the convex hull frontier approach yields the largest benefit for the limited computational time that it consumes.

It is practically useful to note that in absolute terms, even with 128 tasks and 128

[1] To keep run-times feasible, the maximum number of tasks tested had to be significantly dropped.

quality levels per task, `asrmd1` yields a near-optimal result in about 20ms. The result is also within 0.5% of the optimal solution on the average. The absolute time spent, 20ms, is an amount of time that can be used in practice in real-time systems to make near-optimal online QoS-based allocation.

7.3 Practical Performance Evaluation of MRMD Algorithms

In this section, we present a detailed performance evaluation of the dynamic programming (`mrmd`), integer programming (`IP`), and the fast approximation algorithm (`amrmd1`) discussed in Chapter 6.

The workloads for experiments described here were created using the principles and algorithms in Section 7.1.

The three MRMD algorithms were then run on these workloads for a given number of available units on each resource. The running times and total utility obtained for each algorithm were noted. This was repeated for several task sets and we computed the average performance across these repeated experiments. Finally, for larger sized problems, the running times for dynamic programming and integer programming proved to be impractical (hours or days in some cases) and we evaluated only the near-optimal algorithm `amrmd1`.

It must be added here that the optimal results obtained by the integer programming scheme and the dynamic programming algorithm matched. Furthermore, as expected, the approximative algorithms always delivered results that were bounded by the optimal solutions. This provides us with a good degree of cross-validation of correctness with respect to our implementations of our schemes.

7.3.1 Performance of the Dynamic Programming Scheme

We first present the results of the evaluation of the dynamic programming scheme. As mentioned earlier, dynamic programming yields the optimal resource allocation to the various tasks but its running time can be rather large.

Figure 7.10 plots the CPU time consumed by `mrmd` (the dynamic programming algorithm) when there are two resources and $r^{\max} = \langle 180, 100 \rangle$. In other words, the

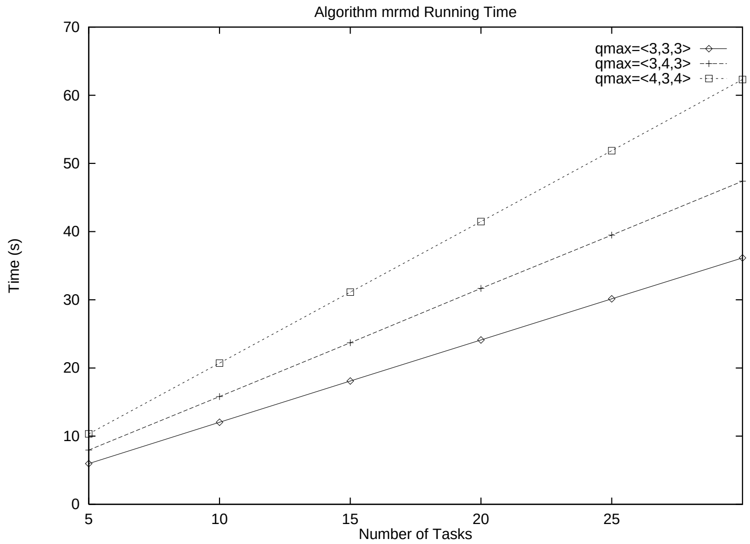

Figure 7.10: The Run Times of Algorithm `mrmd` with $r^{\max} = \langle 180, 100 \rangle$

number of units of resource 1 is 180, and the number of units of resource 2 is 100. By assumption, each resource can only be allocated in integer units[2]. The number of tasks to which these two resources must be allocated is plotted along the x-axis. The CPU time consumed by the dynamic programming scheme is plotted along the y-axis and is in terms of seconds. Three lines are plotted corresponding to different QoS options available to each task. For example, the top-most line corresponds to a QoS maximum of $\langle 4, 3, 4 \rangle$ (i.e. there are three QoS dimensions, each having 4, 3 and 4 discrete options respectively).

As it can be seen, the consumed time increases linearly with the number of tasks, and the slope increases as the number of QoS options to be considered increases. These results are consistent with the pseudo-polynomial complexity of the dynamic programming scheme discussed in Section 6.1. Note also that the running times do not fluctuate, that is they are data independent and very predictable.

It must be noted that, in absolute terms, `mrmd` consumes several tens of seconds for a problem of modest size in terms of the number of tasks. As a result, its applicability in making online decisions in real-time systems is highly questionable with current hardware.

[2]Higher the total number of units, finer is the granularity of the resource allocation.

7.3.2 Performance of the Integer Programming Scheme

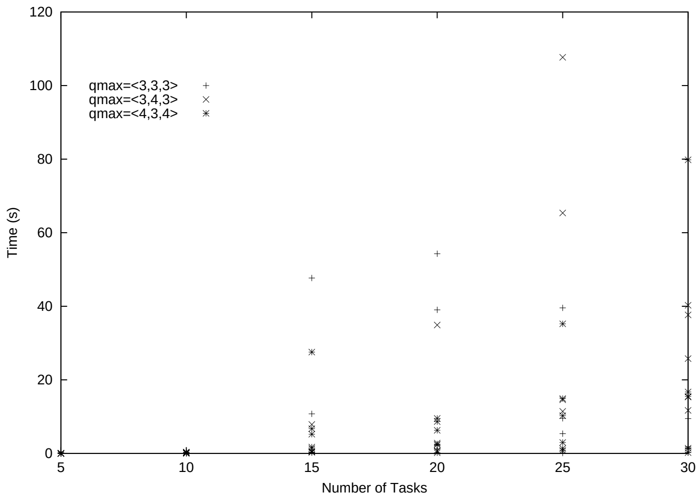

Figure 7.11: Running Times for Computing the Optimal Solution using Mixed Integer Programming

The CPU time consumed by the mixed integer programming package `CPLEX` on three-dimension, two-resource problems are shown in Figure 7.11. The graph plots the running times to find an optimal solution for each of the five runs of different sizes. The results are shown as a scatter-plot rather than as an average of running times due to their high degree of variability. For example, among the five problems with 15 tasks having a QoS maximum of $\langle 4, 3, 4 \rangle$, the running times were 0.59, 0.69, 2.43, 2.79 and 34.91 seconds. This indicates that subtle differences in the specific utility and resource values of set-points can drastically increase the size of the traversed search space. Note that the distribution of the running times has a heavy tail and certainly not normal. Therefore we omit plotting the traditional two-sigma intervals.

Optimality Bounds In order to reduce the running times while still maintaining high-quality results, an upper bound for the deviation from optimality can be specified. A value of 5% of this bound, for example, instructs the algorithm to teminate as soon as it reaches a solution that is within 5% of optimum. The running times for the same problem set as earlier with an optimality bound of 5% is shown in Figure 7.12. By applying this bound, the worst-case running time was reduced to 31.85 seconds

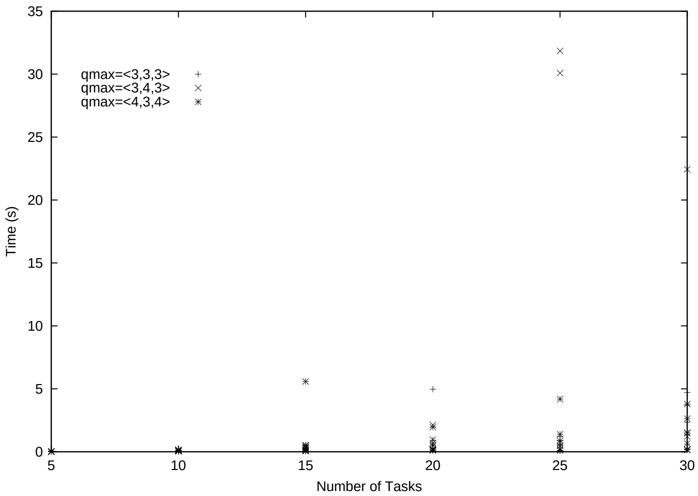

Figure 7.12: Running Times for Computing Solutions using Mixed Integer Programming and a Specified Maximum Deviation from the Optimal Solution

versus 107.69 seconds for finding the optimal solution while at the same time maintaining results which are very close to the optimal solution. The actual quality of the results measured as a fraction of the optimal result is shown in Figure 7.13. All of the solutions in our problem set were more than 96.95% of the optimal solution.

Running-Time Limits If a strict upper bound on the solution time is required, a time limit can also be set. When the time limit for a problem is reached, the best available solution at that time is returned. The solution quality for a 3-second timeout is shown in Figure 7.14. Even with this timeout, all of the sample problems completed with solutions that are at least 93.43% of the optimal. This demonstrates that reasonable sized problems can be solved using integer programming techniques when a timeout is used.

7.3.3 Performance of Local Search Scheme `amrmd1`

We now evaluate the performance of the `amrmd1` algorithm. Figures 7.15 and 7.16 correspond to the same set of tasks used to plot Figure 7.10 (i.e. $r^{\max} = \langle 180, 100 \rangle$, $n = \{5, 10, 15, 20, 25\}$).

Figure 7.15 plots the ratio of the solution quality obtained by `amrmd1` to the opti-

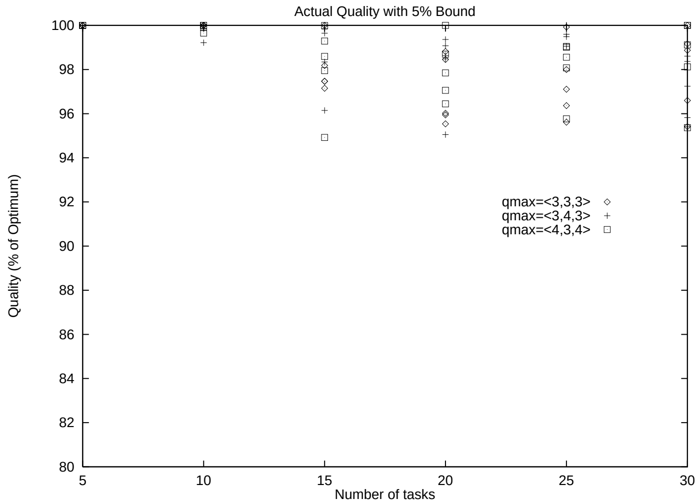

Figure 7.13: Solution Quality Using Mixed Integer Programming and a Specified Maximum Deviation from the Optimal Solution

mal solution obtained by the `mrmd` dynamic programming algorithm. Two conclusions are of immediate interest. The first is from Figure 7.15 and shows that `amrmd1` obtains more than 96% of the maximum quality obtained by the dynamic programming algorithm. The second conclusion is from Figure 7.16 which shows that the solutions can be obtained in the order of tens of milliseconds (instead of tens of seconds for `mrmd`). Hence, in brief, `amrmd1` obtains better than 96% of the quality obtained by `mrmd` but does so three orders of magnitude faster.

We then used `amrmd1` to solve much larger problems (where `mrmd` and mixed integer programming would take too long to be practical). Figure 7.17 plots the scalability of `amrmd1` with respect to the number of tasks and the size of each task's quality space. We used $r^{\max} = \langle 10000, 10000, 10000 \rangle$, $n = 8$, 16, 32, 64, 128, 256, 512, 1024, and the number of QoS dimensions ranged from 1 through 6. The run times plotted along the y-axis are in logarithmic scale. As can be seen, acceptable running times are obtained for up to 100 tasks. The running times scale with both the number of tasks and the number of QoS dimensions.

Finally, Figure 7.18 plots the scalability of `amrmd1` with respect to the number of tasks and number of resources. We now use $q^{\max}$ of each task to be $\langle 3, 3, 3 \rangle$, $n = 8$, 16, 32, 64, 128, 256, 512, 1024. The number of resources ranges from 1 through 6,

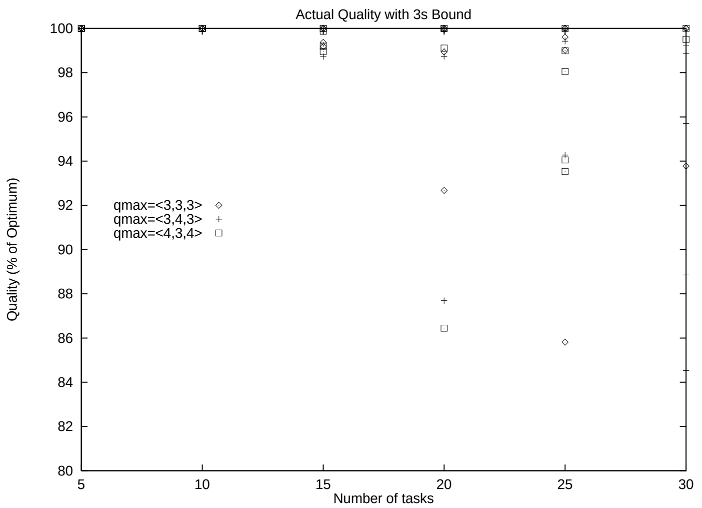

Figure 7.14: Solution Quality with Timeouts in Mixed Integer Programming

where each resource has a very large number of 100000 units. As can be seen, the run times do not change much at all as the number of resources increases. The primary reason is that `amrmd1` uses a compound resource that combines multiple resources into a single virtual resource to be allocated. Hence, it scales well and is robust with any increase in the number of resources. The primary determinant of run times in this case are the number of tasks and quality space which are considered for allocation.

7.3.4 Comparative Evaluation of `amrmd1` and `IP`

The unpredictable run times and the lack of scalability to large problems clearly make pure integer programming methods unsuitable for use in on-line admission control. Even with approximation techniques, such as setting a timeout, high quality results cannot be achieved within a reasonable amount of time. By contrast, the `amrmd1` algorithm obtained solution quality of better than 96% of optimal with a worst-case execution time of only 90ms on the 30 task example compared to solution qualities of 93% of optimal using integer programming with a 3 second timeout. In addition, `amrmd1` also uses far less memory than integer programming which uses substantial amounts of memory as it searches the solution space. The combination of the faster running times and lower memory consumption make `amrmd1` far more suitable for on-line admission control.

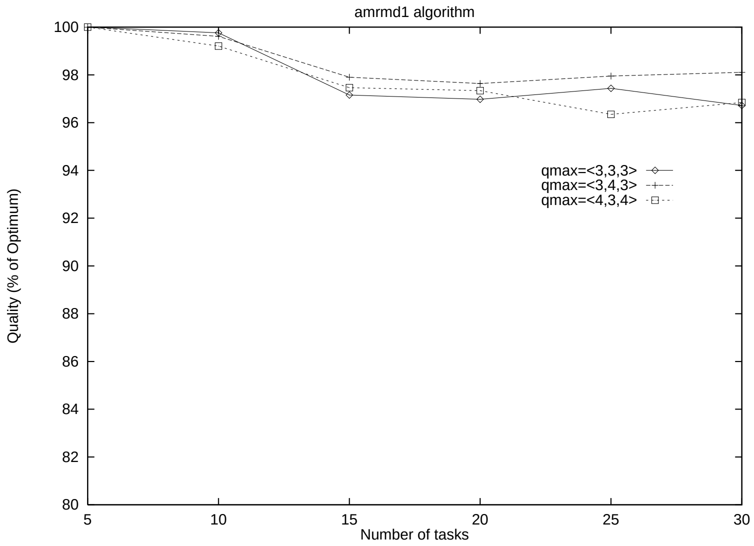

Figure 7.15: Solution Quality obtained by `amrmd1` with $r^{\max} = \langle 180, 100\rangle$

7.3.5 Sample Results

The following example consists of 15 tasks, an $r^{\max}$ of $\langle 180, 100\rangle$ and a $q^{\max}$ of $\langle 4, 3, 4\rangle$. Note that we only show a single assigned quality point for each task even though there might be multiple quality points which consume the same amount of resources.

```
Algorithm amrmd1:
Task 0 : (qid=33,q=<3,3,1>,r=<12,7>,u=0.897889)
Task 1 : (qid=21,q=<2,3,1>,r=<13,3>,u=0.986799)
Task 2 : (qid=20,q=<2,2,4>,r=<10,9>,u=0.635217)
Task 3 : (qid=0,q=<0,0,0>,r=<0,0>,u=0)
Task 4 : (qid=46,q=<4,3,2>,r=<26,8>,u=1)
Task 5 : (qid=33,q=<3,3,1>,r=<7,12>,u=0.476604)
Task 6 : (qid=8,q=<1,2,4>,r=<11,12>,u=0.97287)
Task 7 : (qid=24,q=<2,3,4>,r=<18,11>,u=0.950321)
Task 8 : (qid=9,q=<1,3,1>,r=<18,0>,u=0.904968)
Task 9 : (qid=43,q=<4,2,3>,r=<9,5>,u=0.891014)
Task 10 : (qid=10,q=<1,3,2>,r=<12,2>,u=0.963892)
Task 11 : (qid=5,q=<1,2,1>,r=<11,5>,u=0.881512)
Task 12 : (qid=0,q=<0,0,0>,r=<0,0>,u=0)
Task 13 : (qid=37,q=<4,1,1>,r=<16,14>,u=0.717887)
Task 14 : (qid=16,q=<2,1,4>,r=<17,10>,u=0.907425)
```

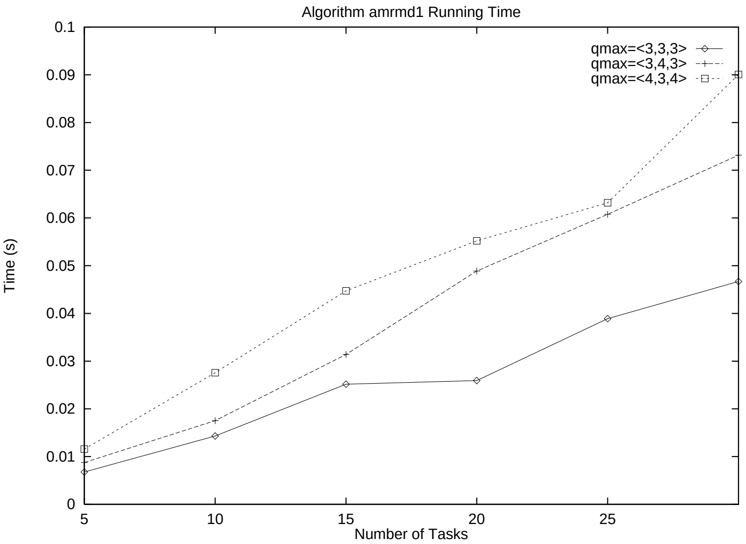

Figure 7.16: Running Times of `amrmd1` with $r^{\max} = \langle 180, 100 \rangle$

```
Time: 39439us
Total: (11.1864,<180,98>)

Algorithm mrmd:
Task 0 : (<12,7>,0.8979)
Task 1 : (<13,3>,0.9868)
Task 2 : (<7,5>,0.5198)
Task 3 : (<0,0>,0)
Task 4 : (<24,7>,0.9452)
Task 5 : (<7,12>,0.4766)
Task 6 : (<21,4>,0.9457)
Task 7 : (<14,11>,0.8911)
Task 8 : (<18,0>,0.905)
Task 9 : (<9,5>,0.891)
Task 10 : (<12,2>,0.9639)
Task 11 : (<11,5>,0.8815)
Task 12 : (<18,8>,0.4959)
Task 13 : (<13,14>,0.6667)
Task 14 : (<0,17>,0.9074)
Time: 31137433us
Total: (11.37, <179,100>)
```

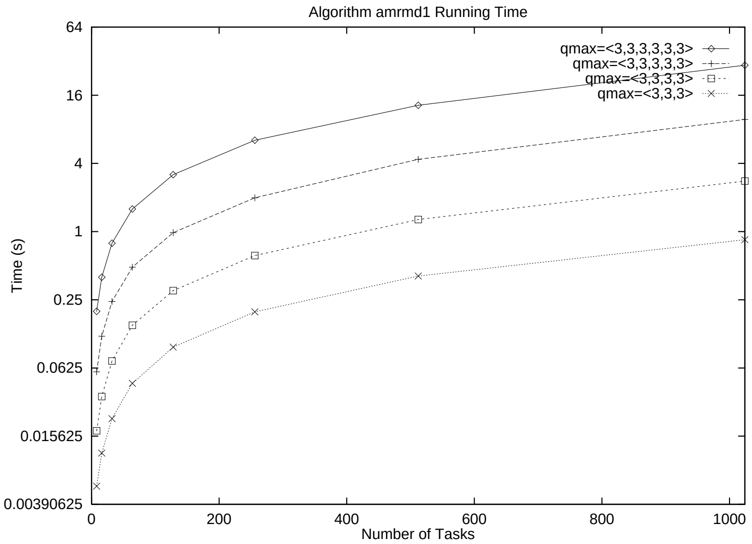

Figure 7.17: Running Times of `amrmd1` with the number of resources $(m) = 3$ and varying the number of QoS Dimensions.

7.4 Chapter Summary

Using extensive simulation studies, this chapter evaluated the performance of the algorithms to solve the SRMD and MRMD problems from the previous chapters. The performance of an algorithm is measured both in terms of its solution quality and its run-time. A good algorithm must yield both a high solution quality and a low run-time.

The workload for the simulations were generated using the principles for user-interfaces described in Chapter 3. Care was taken to ensure that the workloads were rational and that they exhibited tradeoffs.

For the SRMD algorithms, detailed evaluations of the run-times of the three algorithms and their solution qualities shows that the first near-optimal algorithm using the convex hulls performs very close to the optimal solution. It also has very practical run-times that it can even be used on-line. For the MRMD case, as might be expected, the running times are rather high for the dynamic programming and mixed integer programming. The adaptation of the mixed integer programming problem, however, yields near-optimal results with (potentially) significant lower running times. Finally, the approximation algorithm based on a local search technique yields a solu-

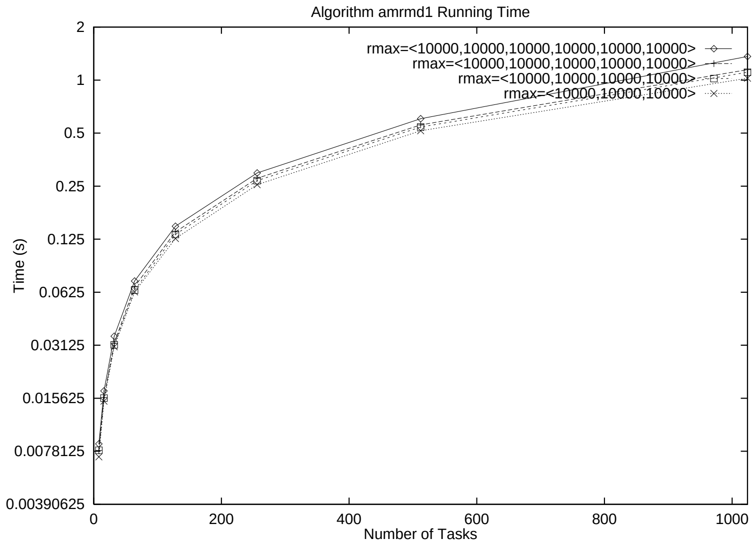

Figure 7.18: Running Times of `amrmd1` with the number of QoS dimensions $(d) = 3$ and varying the number of resources

tion quality that is less than 4% away from the optimal solution but runs more than two orders of magnitude faster. In addition, the use of the "virtual resource" allows this technique to be very scalable and robust as the number of resources required by each application increases.

Chapter 8

Conclusion and Future Work

This dissertation has made some contributions and opened many avenues for future work on the management of quality of service.

8.1 Contributions of the Thesis

The main contributions of this thesis include:

An Analytical Framework for Multidimensional QoS Management

By introducing the abstraction of quality index, which maps qualities to indices in a uniform way, and by the mathematical modeling of QoS tradeoff and resource tradeoff, we transformed the multi-dimension QoS management problem into a combinatorial optimization problem which ultimately enabled us to measure QoS quantitatively, and to analytically plan and allocate resources. We proved that the QoS management problem is NP-hard.

Our QoS management scheme goes beyond the basic QoS scheme of delivering service in a prioritized fashion. Instead, our system allows applications and users to assign values (utilities) to different levels of service that a system can provide. The QoS management optimization module can then explore fully the QoS tradeoffs and resource tradeoffs to make resource allocations to these applications so as to maximize the global utility derived by these systems.

The global objective can be the overall appreciation of the applications in the system, with or without priority enforcement, or the profit margin. The enhanced prioritized allocation can prevent greedy users from cheating the system and amassing resources when accounting is not in place for the services. We also demonstrated that the problem instance associated with each of those objectives can all be solved with our approximation or heuristic schemes without modification of the algorithms.

QoS and Resource Tradeoff

In coping with the shortage of QoS support from an end-user point of view, we proposed a management scheme that empowers the end users to give guidance on the qualities they care about and the tradeoffs they are willing to make under potential resource constraints.

We investigated the important issues of QoS tradeoff and resource tradeoff and demonstrated how they can be used in accommodating user requirements of adaptive nature.

The notion of application utility is used to quantify the relative merits of various QoS levels or points. QoS tradeoffs can therefore be made based on application utilities.

The general resource-quality relation $r \models_i q$ for each task is a description of the application's resource usages at different levels of quality. The same quality level can be satisfied in several ways by making tradeoffs among resources.

QoS Specification Interface

The QoS Specification Interface is semantically rich both in terms of expressiveness and customizability. We proposed and presented some user-interface mechanisms to facilitate such rich specification acquisition, such as utility templates, saturation point, satisfaction knee points, and conditional requirement etc., that users can use to interact easily with the system and to communicate their sophisticated request to the optimisation module efficiently.

Synthetic Rational Task Profile Generation

We developed a systematic way of generating synthetic task profiles that exhibit all the properties we expect from real task profiles, in particular QoS tradeoff, resource tradeoff, and that quality is non-decreasing with respect to resources. Moreover, the characterization of task profiles conforms to user interface methodology described in this thesis.

Fast Approximation Algorithms

We have developed a series of optimization algorithms that tackle the QoS management problem.

The first set of algorithms treats the problem of maximizing system utility by allocating a *single* finite resource to satisfy the QoS requirements of *multiple* applications along *multiple* QoS dimensions. We developed two near-optimal algorithms to solve this problem. The first yields an allocation within a known distance from the optimal solution, and the second yields an allocation whose distance bound from the optimal solution can be *explicitly controlled* by the QoS manager.

The second set of algorithms deals with apportioning *multiple* finite resources to satisfy the QoS needs of *multiple* applications along *multiple* QoS dimensions. We evaluated and compared three strategies. First, dynamic programming and mixed integer programming compute optimal solutions to this problem but exhibit very large running times. We then adapted the mixed integer programming problem to yield near-optimal results with faster running times. Finally, we present an approximation algorithm based on a *local search* technique that is less than a few percent (less than 4% in average in the experiments we conducted) from the optimal solution but which is more than two orders of magnitude faster than the optimal scheme of dynamic programming. Perhaps more significantly, the local search technique turns out to be very scalable and robust as the number of resources under management increases.

8.2 Future Research Directions

It would be interesting to conduct experiments where the integer programming package is combined with an approximation algorithm. This, for example, could be done by giving the near-optimal result of the approximation algorithm as a starting point to the integer programming scheme. This might improve the already very good solution quality of an approximation algorithm with moderate use of extra CPU time. In addition, parallel algorithms can be developed to speed up the computation process.

In this thesis, most of the effort has been spent on the mathematical modeling of QoS management that facilitates QoS tradeoff and resource tradeoff, developing practical algorithms for optimization, and benchmarking workload that characterizes or represents QoS tradeoff and resource tradeoff. The next important step is to subject the system to actual loads from real QoS-aware applications. This means that existing applications need to be modified or new QoS adaptive applications need to be developed.

Our QoS management optimization system is essentially centralized within each domain. When several such domains are connected and thus share resources, a distributed framework has to be in place to facilitate global QoS management.

This thesis as well as [30, 27, 28] focus on the QoS management with discrete quality dimensions, whereas [46] and [47] focus on continuous quality dimensions. It would be useful to combine the two into a unified system.

8.3 Concluding Remarks

We envision an environment where many real-time and non-real-time applications each with multiple QoS dimensions co-exist in a system with a finite set of resources. During loaded periods, the system may not have sufficient resources to deliver the maximum quality possible to every application along each of its QoS dimensions. Hence, decisions must be made by the underlying resource manager to apportion available resources to these applications such that a global objective is maximized. The system can be used to continuously monitor and adjust clients' level of service in light of the dynamically changing operational environment of clients and resources.

Our QoS specification allows applications and users to put values on the different levels of service that the system can provide. When "value" is taken literally, this means that our model is able to facilitate market-efficient resource distribution. Such a system has considerable potential, especially in solving bandwidth problems of the increasingly crowded Internet.

Appendix A

Symbols and Notations

A.1 Symbols in General Use

The following symbols are used throughout the thesis:

Symbol	*Brief Explanation*	*Page*
f_{ij}	mapping between actual quality and quality indices	10
n	number of tasks	13
m	number of resources	13
d_i	number of quality dimensions of task T_i	13
$T_1, \ldots, T_n$	tasks in the system	13
$R_1, \ldots, R_m$	resources for management	13
$Q_{i1}, \ldots, Q_{id_i}$	dimensional quality space for T_i	13
R	resource space	13
$r^{\max}$	maximum amount of resource for allocation — a vector	13
$\models_i$	resource-quality relation, or resource profile for T_i	13
$Q_1, \ldots, Q_n$	quality space for task T_1, …, T_n	14
u_i	application utility for T_i	14
$\mathbb{R}$	real numbers	14
$q_i^{\min}$	minimum quality for T_i — a vector	15
$q_i^{\max}$	maximum quality for T_i — a vector	15

Symbol	*Brief Explanation*	*Page*
u	system utility in general	16
u_w	system utility — weighted sum of application utilities	16
u^*	system utility — minimum utility accrued for a task	16
w_i	weight for T_i	16
u_{ij}	dimensional utility of T_i	29
w_{ij}	dimensional utility weight for T_i	30
g_i	best-utility function	41
h_i	best-utility quality selector	41
π_k	projection of the resource space into its kth dimension	77

A.2 Symbols Used for NP-hard Proof

Symbol	*Brief Explanation*	*Page*
$\kappa_{i1}, \ldots, \kappa_{i\|Q_i\|}$	an enumeration of the quality space	36
N_{ij}	number of resource usage choices	36
$\rho_{ij1}, \ldots, \rho_{ijN_{ij}}$	an enumeration of the resource usage choices	36
x_{ijk}	binary variable, $x_{ijk} = 1$ if task T_i has been given quality point κ_{ij} and resource consumption ρ_{ijk}, and $x_{ijk} = 0$ otherwise	36
n	number of knapsack items	37
c	knapsack capacity	37
p_i	the profit of adding item i	37
w_i	the weight of item i	37
x_i	indicator variable for knapsack item i	38

A.3 Symbols and Notations Used in the Algorithms

Symbol	*Brief Explanation*	*Page*
v	dynamic programming recursive function	43
P	number of allocation units in total	43
p	units of resource allocated	43
$\binom{u}{r}$	r-u-pair	44
C_i	r-u-pair list; list of function g_i's discontinuity points	44
g_i°	convex hull frontier of g_i	45
C_i'	list of function g_i°'s discontinuity points	47
L	maximum length of C_i	45
U_{opt}	optimal utility	47
U_{asrmd1}	utility obtained through algorithm `asmrd1`	47
δ_i	maximum utility difference between adjacent discontinuity points of C_i'	47
χ	largest δ_i	47
U_{asrmd2}	utility obtained through algorithm `asmrd2`	49
ε	error bound relative to optimal result	49

Bibliography

[1] E. Aarts and J. Lenstra, editors. *Local Search in Combinatorial Opitmization*. John Wiley & Sons, 1997.

[2] E. Atkins, T. Abdelzaher, and K. Shin. QoS Negotiation in Real-Time Systems and Its Application to Automated Flight Control. In *Proceedings of the IEEE Real-time Technology and Applications Symposium*, June 1997.

[3] J. Bolliger and T. Gross. A Framework-Based Approach to the Development of Network-Aware Applications. In *IEEE Trans. Software Engineering (Special Issue on Mobility and Network-Aware Computing)*, volume 24, pages 376–390, May 1998.

[4] A. Campbell, G. Coulson, and D. Hutchison. A Quality of Service Architecture. *Computer Communication Review*, 24(2):6–27, April 1994.

[5] P. Chandra, A. Fisher, C. Kosak, and P. Steenkiste. Network Support for Application-Oriented Quality of Service. In *Sixth IEEE/IFIP International Workshop on Quality of Service*, May 1998.

[6] D. Clark, S. Shenker, and L. Zhang. Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism. In *Proceedings of the SIGCOMM '92 Symposium on Communications Architectures and Protocols*, pages 14–26, October 1992.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press / McGraw-Hill, 1990.

[8] G. Coulson, A. Campbell, and P. Robin. Design of A QoS Controlled ATM Based Communication System in Chorus. In *IEEE Journal of Selected Areas in*

Communications (JSAC), Special Issues on ATM LANs: Implementation and Experiences with Emerging Technology, May 1995.

[9] B. Dantzig. Discrete Variable Extremum Problems. In *Operations Research*, volume 5, pages 266–277, 1957.

[10] CPLEX Division. *Using the CPLEX Callable Library.* ILOG Inc., 1997.

[11] S. Floyd and V. Jacobson. Link-sharing and resource management models for packet networks. In *IEEE/ACM Transactions on Networking*, volume 3, pages 365–386, August 1995.

[12] J. M. Hyman, A. A. Lazar, and G. Pacifici. Real-Time Scheduling with Quality of Service Constraints. *IEEE Journal on Selected Areas in Communications*, 9(7), September 1991.

[13] T. Ibaraki. Enumerative Approaches to Combinatorial Optimization — Part 1. *Annals of Operations Research*, 10, 1987.

[14] T. Ibaraki. Enumerative Approaches to Combinatorial Optimization — Part 2. *Annals of Operations Research*, 11, 1987.

[15] O. Ibarra and C. Kim. Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems. *Journal of ACM*, 22:463–468, 1975.

[16] ITU. ITU-T Recommendation H.263 - Video Coding for Low Bit Rate Communication, July 1995.

[17] K. Jeffay, D. Stone, and F. Smith. Kernel support for live digital audio and video. In *Proceedings of the Second International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 10–21, November 1991.

[18] E. D. Jensen, C. D. Locke, and H. Tokuda. A Time-Driven Scheduling Model for Real-Time Operating Systems. In *Proceedings of the 6th IEEE Real-Time Systems Symposium*, December 1985.

[19] M. Jones, P. Leach, R. Draves, and J. Barrera. Modular Real-Time Resource Management in the Rialto Operating System. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, May 1995.

[20] A. Kalavade and P. Moghè. Asap – a framework for evaluating run-time schedulers in multimedia end-systems. In *Proc. of ACM Multimedia Conference*, September 1998.

[21] K. Kawachiya and H. Tokuda. A Negotiation-Based Resource Management Framework for Dynamic QoS Control. In *Proceedings Real-Time Mach Workshop '97*, August 1997.

[22] S. Khan. *Quality Adaptation in a Multisession Multimedia System: Model, Algorithms and Architecture*. PhD thesis, University of Victoria, 1998.

[23] AT&T Labs. GeoPlex — The Enhanced Network Infrastructure For 21st Century Services. AT&T White Paper, May 1997.

[24] A. Lazar, L. Ngoh, and A. Sahai. Multimedia Networking Abstraction with Quality of Services Guarantees. In *Proc. SPIE Conference on Multimedia Computing and Networking*, February 1995.

[25] C. Lee. A Translucent QoS Architecture. In *Proceedings of the RT-Mach Workshop'97*, August 1997.

[26] C. Lee, Y. Katsuhiko, R. Rajkumar, and C. Mercer. Predictable Communication Protocol Processing in Real-Time Mach. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, June 1996.

[27] C. Lee, J. Lehoczky, R. Rajkumar, and D. Siewiorek. On quality of service optimization with discrete qos options. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*. IEEE, June 1998.

[28] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hansen. A scalable solution to the multi-resource qos problem. *To appear in Proceedings of the IEEE Real-Time Systems Symposium*, 1999.

[29] C. Lee, R. Rajkumar, and C. Mercer. Experience with Processor Reservation and Dynamic QoS in Real-Time Mach. In *Proceedings of the Multimedia Japan ́96*, March 1996.

[30] C. Lee and D. Siewiorek. An Approach for Quality of Service Management. Technical Report CMU-CS-98-165, Computer Science Department, Carnegie Mellon University, October 1998.

[31] J. Liu, K. Lin, R. Bettati, D. Hull, and A. Yu. *Use of Imprecise Computation to Enhance Dependability of Real-Time Systems*. Kluwer Academic Publishers, 1994.

[32] C. D. Locke. *Best-effort Decision Making for Real-Time Scheduling*. PhD thesis, CMU, 1986.

[33] Q. Ma and P. Steenkiste. Quality of Service Routing for Traffic with Performance Guarantees. In *IFIP International Workshop on Quality of Service*, May 1997.

[34] S. Martello and P. Toth. *Knapsack Problems — Algorithms and Computer Implementations*. John Wiley & Sons Ltd., 1990.

[35] S. McCanne and V. Jacobson. vic: A Flexible Framework for Packet Video. In *Proc. ACM Multimedia'95*, November 1995.

[36] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS)*, pages 90–99, May 1994.

[37] P. Moghè and A. Kalavade. Terminal qos of adaptive applications. *Bell Labs Technical Journal*, Spring Issue, 1998.

[38] M. Moser, P. Jokanovic, and N. Shiratori. An algorithm for the multidimensional multiple-choice knapsack problem. *IEJCE Trans. Fundamentals*, E80-A(3), March 1997.

[39] K. Nahrstedt. Network Service Customization: End-point Perspective. Technical Report MS-CIS-93-100, University of Pennsylvania, December 1993.

[40] T. Nakajima and H. Tezuka. A Continuous Media Application supporting Dynamic QOS Control on Real-Time Mach. In *Proceedings of the Second ACM International Conference on Multimedia*, pages 289–297, October 1994.

[41] B. Noble, M. Satyanarayanana, D. Narayanan, J. Tilton, J. Flinn, and K. Walker. Agile Application-Aware Adaptation for Mobility. *Proceedings of the 16th ACM Symposium on Operating System Principles*, oct 1997.

[42] M. Overmars and J. Leeuwen. Maintenance of Configurations in the Plan. In *Journal of computer and System Sciences*, volume 23, pages 166–204, 1981.

[43] A. L. Peressini, R. E. Sullivan, and Jr. J. J. Uhl. *Convex Programming and the Karish-Kuhn-Tucker conditions*, chapter 5. Springer-Verlag, 1980.

[44] D. Pisinger. *Algorithms for Knapsack Problems*. PhD thesis, University of Copenhagen, Dept. of Computer Science, February 1995.

[45] F. Preparata and M. Shamos. Computational Geometry : An Introduction. In *Texts and Monographs in Computer Science*. Springer-Verlag, 1985.

[46] R. Rajkumar, C. Lee, J. P. Lehoczky, and D. P. Siewiorek. A QoS-based Resource Allocation Model. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1997.

[47] R. Rajkumar, C. Lee, J. P. Lehoczky, and D. P. Siewiorek. Practical Solutions for QoS-based Resource Allocation Problems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1998.

[48] R.Gopalakrishnan and G. Parulkar. A Framework for QoS Guarantees for Multimedia Applications within an Endsystem. In *Swiss German Computer Society Conference*, September 1995.

[49] T. Saaty. Multicriteria Decision Making - The Analytic Hierarchy Process. Technical report, University of Pittsburgh, RWS Publications, 1992.

[50] S. Sahni. Approximation algorithms for the 0-1 knapsack problem. In *Journal of ACM*, volume 23, pages 555–565, 1975.

[51] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann Publishers, Inc., 1996.

[52] H. Schulzrinne, S. Casner, R. Frederic, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 1889, 1996.

[53] M. Shankar, M. Miguel, and J. Liu. An end-to-end qos management architechure. In *Porcceding of the Fifth Real-Time Technology and Applications Symposium*. IEEE, June 1999.

[54] P. Steenkiste, A. Fisher, and H. Zhang. Resource Management in Application-aware Networks. In *Workshop on the Integration of IP and ATM*, November 1997.

[55] I. Stoica, H. Zhang, and E. Ng. A Hierarchical Fair Service Curve Algorithm for Link-Sharing, Real-Time and Priority Service. In *Proceedings of SIGCOMM'97*, 1997.

[56] H. Tokuda and T. Kitayama. Dynamic QOS Control based on Real-Time Threads. In *Proceedings of the Fourth International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 113–122, November 1993.

[57] H. Tokuda, Y. Tobe, S. T.-C. Chou, and J. M. F. Moura. Continuous Media Communication with Dynamic QOS Control Using ARTS with an FDDI Network. In *Proceedings of the SIGCOMM '92 Symposium on Communications Architectures and Protocols*, pages 88–98. ACM, October 1992.

[58] Y. Toyoda. A simplified algorithm for obtaining approximate solution to zero-one programming problems. *Management Science*, 21, 1975.

[59] Carnegie Mellon University. Amaranth Project. Amaranth White Paper, December 1996.

[60] C. Volg, L. Wolf, R. Herrtwich, and H. Wittig. HeiRAT – Quality of Service Management for Distributed Multimedia Systems. In *Multimedia Systems Journal*, November 1995.

[61] H. Zhang and D. Ferrari. Improving Utilization for Deterministic Service in Multimedia Communication. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS)*, pages 295–304, May 1994.

[62] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource ReSerVation Protocol. *IEEE Network*, pages 8–18, September 1993.