A Scalable Solution to the Multi-Resource QoS Problem

Chen Lee, John Lehoczky, Dan Siewiorek
Ragunathan Rajkumar and Jeff Hansen

May 1999
CMU-CS-99-144

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

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 [6]. In this paper, 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 provably NP-hard problem. We show that dynamic programming and mixed integer programming compute optimal solutions to this problem but exhibits very high running times. 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 that is less than 5% away from the optimal solution but which is more than two orders of magnitude faster. 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.

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.

Keywords: QoS management and optimization, QoS index and utility model, QoS tradeoff, resource tradeoff, QoS based resource allocation, multi-media, scalability

1 Introduction

1.1 Motivation

Consider a video-conferencing application. The audio streams in this application have multiple QoS considerations: 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 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.

1.2 Our Approach

The QoS architecture [6] we consider consists of a QoS specification interface, a quality trade-off 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. The QoS trade-off model allows applications and users to assign (utility) values to different levels of service that a system can provide. Finally, a QoS resource manager, taking QoS operating parameters of arriving applications as its inputs, makes resource allocations to these applications so as to maximize the global utility derived by these systems. A set of profiles that map QoS requirements to resource usage is used by the QoS resource manager during this allocation step.

In [6], we presented a QoS (Quality of Service) management framework that enabled system and application developers to quantitatively measure QoS, and to analytically plan and allocate resources. 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.

Using this QoS architecture as the foundation, we studied in [6] 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 presented two near optimal algorithms to solve this problem. The first yielded a resource allocation which was within a known bounded distance from the optimal solution, and the second yielded an allocation whose distance from the optimal solution can be explicitly controlled by a QoS manager.

In this paper, we consider the more complex problem of apportioning *multiple* finite resources to satisfy the QoS needs of multiple applications along multiple QoS dimensions. Each application in this context must satisfy requirements along multiple QoS dimensions and also requires the use of multiple resources. We provide a proof that an optimal solution to this problem is NP-hard. We then present and compare three solutions to this problem. These solutions comprise of a dynamic programming solution, a mixed integer programming solution and an approximation based on a *local search* technique.

1.3 Related Work

It must be noted that utility functions have been used by economists for several decades in attempts to model human behavior. Value-function scheduling was first applied by Jensen et al. [4] in the context of real-time systems. Liu et al. have used a similar notion in their use of "imprecise" computations [7]. They 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. Liu et al., as part of the Open Systems project at the University of Illinois at Urbana-Champaign, have also been studying an end-to-end QoS model that allows a stream spanning multiple nodes to have the same (or appropriately transformed) QoS parameters. Seto et al. [15] studied the problem of the optimal allocation of CPU cycles to feedback control applications, whose control quality improves in concave fashion with higher frequencies of operation. The computation time per instance of an application also remained constant in their model. In our model, the allocation decision is made with respect to the utilization available on a single resource, and not with respect to either the computation time or the period. Our model also deals with multiple resources and multiple QoS dimensions.

Prior to [6], the work reported in [12, 13] primarily dealt with *continuous* QoS dimensions, and assumed that the utility gained by improvements along a QoS dimension were always representable by concave functions. In [6], both of these assumptions were relaxed by supporting discrete QoS operating points, and making no assumptions about the concavity of the utility functions. In this paper, we relax the assumption of allocating a single resource made in [6].

1.4 Organization of the Paper

The rest of the paper is organized as follows. In Section 2, we describe the QoS model and formulate the QoS optimization problem being solved. In Section 3, we present an alternative formulation of the problem both to demonstrate its computational complexity and to serve as a basis for an algorithm to be presented in Section 4. In Section 4, we present three optimization algorithms for solving the multi-resource QoS problem. In Section 5, we present a detailed performance evaluation of the three algorithms. We show that our approximation solution based on a local search technique yields high quality results (better than 95% of the optimal result) at speeds that are orders of magnitude faster. Finally, we draw some conclusions and discuss future work in Section 6.

2 System Modeling

In this section, we provide a complete overview of the entities and metrics we use in this paper, including the Quality of Service and resource allocation model that we employ.

2.1 Tasks and System Resources

We consider a system with multiple resources that services n independent applications denoted by $T_1, \ldots, T_n$. There are m distinct shared system resources which are allocated across the n applications. We let R_i denote the set of possible allocation choices for the i^{th} shared resource. This set of possible allocations could be modeled either as a discrete set (e.g. the bandwidth required to support periodic transmission of fixed sized packets at one of several sampling frequencies) or as a continuous variable (e.g. processor cycles required to ensure completion of a task's worst case computation requirements). Each of the shared resources has a maximum quantity or size denoted by $r^{\max} = (r_1^{\max}, \ldots, r_m^{\max})$. We also denote the set of possible resource allocation choices by $R = R_1 \times \cdots \times R_m$.

2.2 Application QoS Requirements

Each application has its own quality-of-service (QoS) requirements, and it contends with many other applications for system resources. We let $Q_{i1}, Q_{i2}, \ldots, Q_{id_i}$ be the quality-of-service dimensions associated with task T_i. Each Q_{ij} is a finite set of quality choices for the i^{th} task's j^{th} quality-of-service dimension, and we define the set of possible quality vectors by $Q_i = Q_{i1} \times \cdots \times Q_{id_i}$.

Using the video-conferencing application as an example, the following is a sample list of quality dimensions (and their dimensional spaces) that might be associated with any particular application. The list is given to concretely illustrate quality dimensions that might be considered and is not intended to be exhaustive.

- Cryptographic Security (encryption key-length) : 0(off), 56, 64, 128
- Data Delivery Reliability, which could be
  - maximum packet loss : as a percentage of all packets
  - expected packet loss : as a percentage of all packets
  - packet loss occurrence : as a per packet probability of loss
- 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-frame-rate cartoon or 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 — end-to-end delay(ms): ..., 100, 75, 50, 25, ...
    (Note that we list these in worst-to-best order, not numerically increasing.)

2.3 Application and User Profiles

We associate with each T_i an *Application Profile* and a *User Profile.* An Application Profile comes from an application designer, while a User Profile is a means for a user to provide user-specific quality requirements associated with a particular session. A user can either instantiate the attributes of the default application profile, by selecting one of many templates supplied with the application profile, or the user can supply her own utility values to quantify the quality derived from any particular choice of a QoS setting.

An *Application Profile* has two components: a *QoS Profile* and a *Resource Profile.* A *QoS Profile* for task T_i consists of the following components for each of the quality dimensions:

[1]The choices listed here come from [3] [14]. Other standards, such as MPEG [9] [5] [14] can be used as well.

- Quality Space — Q_i (e.g. possible picture formats, possible audio sampling rates, etc.)
- Quality Index — for the jth component of Q_i we define a bijective function, f_{ij} which maps the elements of Q_{ij} into integer valued labels, i.e.

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

  The labels must preserve the quality ordering, i.e., q_1 is "better than" q_2, if and only if $f_{ij}(q_1) > f_{ij}(q_2)$.
- Dimension-wise Quality Utility — $u_{ij} : Q_{ij} \rightarrow \mathbb{R}$, a numerical assessment of the utility achieve by setting $q_{ij} \in Q_{ij}$ for application T_i on quality dimension j.
- *Application Utility* — a utility or quantitative QoS measure associated with application T_i

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

  The function u_i could, for example, be defined as a weighted sum of u_{ij}

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

  We require that u_i be non-decreasing in all d_i arguments and that it be non-negative.

A *Resource Profile* for T_i defines a relation between R and Q_i, $r \models_i q$. This relation is a list of all resource allocation combinations that can be used to achieve each quality point q. It is important to recognize that there may be many choices of r which lead to the same value of q. For example, one might compress files to reduce network bandwidth requirements, while using more CPU processing cycles to perform the compression/decompression operations. Alternatively, one could avoid compression which would require more network bandwidth, but would reduce the processing cycles needed. It is important to note that the two approaches result in different system resource requirements but identical user quality. On the other hand, a given set of system resource allocations can be used differently by different applications and hence can result in different quality levels. For example, CPU processing capacity could be used to give medium levels of audio and video quality, or the same capacity could be used for high audio quality and low video quality. Consequently, we can only define a relation between Q_i and R, not a function.

2.4 Application QoS Constraints

We allow each application to specify minimum QoS requirements for each of its relevant QoS dimensions:

$$q_i^{\min} = (q_{i1}^{\min}, q_{i2}^{\min}, \ldots, q_{id_i}^{\min}).$$

When the minimum requirements cannot be satisfied, the user of task T_i might choose not to run T_i at all. Alternatively, we could assume that the system is designed for a fixed set of users, and each user will be entitled to at least its minimum quality levels, even if that means that the quality levels of other users must be reduced to achieve this. In this paper, we assume that there are sufficient resources to ensure that the minimum QoS constraints can be satisfied for all applications.

2.5 Application Utilities

QoS is usually multi-dimensional, and its measure can be either objective or subjective (user or session dependent). A user might want to make some quality tradeoffs, especially when resources (such as processing power or network bandwidth) can change dynamically and an application's resource allocation might be reduced. For example, a user (or task T_i) might generally have a desired quality level. However, she may be able to accept lower quality in certain dimensions if there are insufficient resources to obtain the desired quality levels. It is, therefore, to the user's advantage for a system to provide an interface that allows that user to make implicit or explicit quality tradeoffs.

Quality Index. Certain quality dimensions, such as frame rate or end-to-end delay have easily defined utility functions while others, such as picture format, are often expressed in non-numeric, non-uniform, or non-increasing order and require a mapping from the quality space to a numeric quantitative space. We introduce the notion of $QualityIndex$ to map quality levels to indices.

We now illustrate the concept and the use of the Quality Index in the context of our previous video-conferencing application. Consider task T_i, which could be a video conferencing system. T_i's quality dimensions, quality space and Quality Index might be represented by the following:

Picture format: Assume it uses the H263 [3] 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. 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).

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, $Q_{i7} = \{1, 2, \dots, 5\}$ maps high number to low indices.

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

Dimension-wise and Application Utilities. Quality points in the multi-dimensional case generally do not have a complete ordering. The individual dimensions, however, do. Recall that the application utility u_i for T_i is defined in terms of the value that accrues when T_i achieves a certain quality, i.e. $u_i : Q_i \to \mathbb{R}$. As discussed above, when many quality dimensions are involved, it is often very difficult for a user to express his/her quality preferences. We therefore provide the user with the capability to specify dimension-wise quality utilities. As a result, the application utility can then be defined as a weighted sum of dimension-wise utility.

Given the Quality Index, a dimension-wise utility can be defined and the application utility can be defined from it.

System Utilities. For the overall system, with multiple applications each of which possibly requires multiple resources, we define a system utility function

$$u : Q_1 \times \cdots \times Q_n \to \mathbb{R},$$

which could be defined in a variety of ways such as:

- A (weighted) sum of *Application Utilities*

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

  for *differential services,* where u_i is non-decreasing, and $0 \le w_i \le 1$ could be the relative importance or priority[2] of T_i, or

- $u = u^*$, where

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

  for *"fair" sharing.*

Problem Formulation. For a given set of tasks $T_1, \cdots, T_n$, our problem 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 optimization problem:

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

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

In the next section, we will formulate the combinatorial problem in a different manner, so as to precisely show the complexity of the QoS optimization problems. We will also also present an algorithm from that perspective in Section 4.2.

3 Optimization Problem Complexity

In the previous section, we defined a general QoS optimization problem involving multiple resources (`MR`) and multiple QoS dimensions (`MD`). The general problem is, therefore, denoted by `MRMD`. It is useful to identify three special cases of this problem in which either the number of resources is restricted to a single resource (`SR`) or there is a single QoS dimension (`SD`) or both. Algorithms for `SRSD`, `SRMD` and `MRSD` problems have been discussed in [12, 13]. In Section 2, we discussed QoS dimensions and indicated that those could be either continuous (e.g. processor cycles to ensure a task's worst case computation time) or discrete (e.g. different video formats). In this paper, we focus on the optimization problem only for the case of discrete QoS settings. In this formulation, we can give an explicit enumeration of all possible QoS operating points for each task. Using this discrete formulation, the `MRMD` optimization problem defined in the previous section can be restated 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 require $\rho_{ijk} \models_i \kappa_{ij}$, that is the resource vector ρ_{ijk} must provide QoS levels κ_{ij} to application T_i.

Let $x_{ijk} = 1$ if task T_i is assigned quality point κ_{ij} and resource consumption ρ_{ijk}, and $x_{ijk} = 0$ otherwise. Hence, if task T_i is accepted for processing by the system, then exactly one of the indicator variables x_{ijk} equals 1, while the others are 0. If T_i is not accepted, then all are 0. Using this notation, the optimization problem can be stated as:

$$\begin{aligned}
&\text{maximize} && \sum_{i=1}^{n}\sum_{j=1}^{|Q_i|}\sum_{k=1}^{N_{ij}} x_{ijk}u_i(\kappa_{ij}) \\
&\text{subject to} && \sum_{i=1}^{n}\sum_{j=1}^{|Q_i|}\sum_{k=1}^{N_{ij}} x_{ijk}\rho_{ijk\ell} \le r_\ell^{\max}, \quad && \ell = 1, \ldots, m, \\
& && \sum_{j=1}^{|Q_i|}\sum_{k=1}^{N_{ij}} x_{ijk} \le 1\ , && i = 1, \ldots, n, \\
& && x_{ijk} \in \{0, 1\}, && i = 1, \ldots, n,\ j = 1, \ldots, |Q_i|\ ,\ k = 1, \ldots, N_{ij}.
\end{aligned} \tag{2}$$

Note, that $\rho_{ijk\ell}$ is just the ℓth coordinate of the vector ρ_{ijk}. Note that the possible QoS levels for application T_i (κ_{ij}), their utility ($u_i(\kappa_{ij})$), resource requirements using ρ_{ijk} ($\rho_{ijk\ell}, 1 \le \ell \le m$) and total resource availability ($r_\ell^{\max}$) are given constants. The variables to be selected to optimize total system utility are the x_{ijk}. Consequently, all the instances of our problem (`SRSD`, `SRMD`, `MRSD`, `MRMD`) can be viewed as special cases of the general (mixed) Integer Programming or Nonlinear Programming problems.

We now consider the complexity of the `SRSD`, `SRMD`, `MRSD`, `MRMD` problems.

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

Proof Since SRSD is a special case of the other `SRMD`, `MRSD` and `MRMD`, it is sufficient to show that `SRSD` is NP-hard.

For `SRSD`, we have $m = N_{ij} = 1$ and thus $k = \ell = 1$. Consequently, Problem (2) becomes

$$\begin{aligned}
&\text{maximize} && \sum_{i=1}^{n}\sum_{j=1}^{|Q_i|} x_{ij1} u_i(\kappa_{ij}) \\
&\text{subject to} && \sum_{i=1}^{n}\sum_{j=1}^{|Q_i|} x_{ij1}\rho_{ij11} \le r_1^{\max}, \\
& && \sum_{j=1}^{|Q_i|} x_{ij1} \le 1, && i = 1, \ldots, n, \\
& && x_{ij1} \in \{0, 1\}, && i = 1, \ldots, n,\ j = 1, \ldots, |Q_i|\,.
\end{aligned} \tag{3}$$

The 0–1 Knapsack Problem is known to be NP-hard [8]. 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

$$\begin{aligned}
&\text{maximize} && \sum_{i=1}^{n} p_i x_i \\
&\text{subject to} && \sum_{i=1}^{n} w_i x_i \le c \\
& && x_i \in \{0, 1\}, \quad i = 1, \ldots, n,
\end{aligned} \tag{4}$$

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

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

which gives the 0–1 Knapsack Problem's x_i represented by x_{i11} in the SRSD problem.□

The observation that the `SRSD` (hence `MRMD`) problem is NP-Hard indicates that systems with a large number of tasks cannot be optimized in real-time. Nevertheless, this does not preclude the possibility that special versions of the problem can be optimally solved, or that fast algorithms that give near optimal solutions cannot be found. In the next section, we report on algorithms that offer near-optimal solutions for `MRMD` problems of substantial size.

4 `MRMD` Algorithms

In this section, we develop several algorithms which will be used to find near-optimal solutions to the `MRMD` QoS optimization problem. In view of the multi-dimensional and potentially subjective nature of QoS, there may be no complete ordering among quality-of-service points, even for individual tasks. Moreover, as discussed earlier, in some instances, different combinations of resources can be used to obtain one multi-dimensional quality point. Consequently, there may be no function that can be defined to map the resources allocated to an application to its achieved utility. To deal with this problem, a structural composition is required for the algorithms that calls for a mapping from resource to utility. Specifically, an *R-U* (Resource to Utility) function/graph will be constructed for each task through the *QoS Profile* and *Resource Profile* defined in Section 2.

Recall that given a particular resource allocation to a task, one could use those resources to improve QoS in some dimensions and reduce it in others, and these different allocations would lead to different utility values. Still, the most valued QoS point for each resource value can be chosen. We then define a function $g_i : R \rightarrow \mathbb{R}$, such that

$$g_i(r) = \max\{ u_i(q) \mid r \models_i q \} \tag{5}$$

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

$$h_i(r) = \{ q \in Q_i \mid u_i(q) = g_i(r) \wedge r \models_i q \}. \tag{6}$$

An *R-U* graph can then be generated for each task, each of which would be a multi-dimensional step function.

4.1 Finding the Optimal Solution for `MRMD`

The solution method and algorithm described in this section can be viewed as an extension of the dynamic programming algorithm described in [6]. 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

$$v(i, p_1, p_2) = \max_{\substack{p'_1 \in \{0,\ldots,p_1\} \\ p'_2 \in \{0,\ldots,p_2\}}} \{ g_i(p'_1, p'_2) + v(i-1, p_1 - p'_1, p_2 - p'_2) \} \tag{7}$$

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 (7), 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$(n, P_1, P_2, C_1, \ldots, C_n)$

```
for p1 = 0 to P1 do            // Initialization
  for p2 = 0 to P2 do
    v(0, p1, p2) := 0
    r(0, p1, p2) := 0
    qos(0, p1, p2) := nil

for i = 1 to n do              // Dynamic programming
  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 (rij ≰ (p1, p2)) then
          continue
        else
          u := uij + v(i − 1, p1 − rij1, p2 − rij2)
        if (u > u*) then
          u* := u
          r* := rij
          j* := j
      v(i, p1, p2) := u*
      r(i, p1, p2) := r*
      qos(i, p1, p2) := qos(i − 1, p1 − rij1, p2 − rij2) concat [hi(rij*)]

(p1, p2) := r^max              // Unwind and retrieve allocation results
for i = n downto 1 do
  resource(i) := r(i, p1, p2)
  utility(i) := v(i, p1, p2)
  (p1, p2) := (p1, p2) − resource(i)
return v(n, P1, P2), qos(n, P1, P2), resource(1), ...,resource(n), utility(1), ...,utility(n)
```

Upon the return of the **mrmd** algorithm, $qos(n, P_1, P_2)$ 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 multiple 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 measurement of other heuristic and approximation schemes.

4.2 Integer Programming

Using the problem formulation given in Equation 2 of Section 3, Integer Programming algorithms can also be applied. For efficiency reasons, we use the CPLEX [2] `MIP` callable library which employs a branch-and-bound algorithm. In the branch-and-bound method, a series of 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 section. Still, its applicability for practical but large `MRMD` problems is yet to be determined.

4.3 Approximation Algorithm for `MRMD`

In this section, we shall define an algorithm that yields near-optimal results but can execute at potentially much higher speeds than the optimal algorithms using dynamic programming or mixed integer programming. We shall use an algorithm that uses a *localsearch* 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 [6] 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:

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

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

```
u* := 0
p := initial_penalty (C_1, ..., C_n, r^max)
repeat := true
while repeat do
   repeat := false
   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*) > ε) then
      repeat := true
      u* := u
      for i = 1 to n do
         u*[i] := u[i]
         r*[i] := r[i]
      p := adjust_penalty (p, C_1, ..., C_n, r^c, r^max)
for i = 1 to n do
   q[i] := h_i(r*[i])              // see Equation (6)
return u*, q[1], ..., q[n], r*[1], ..., r*[i]
```

Note that the procedure **convex_hull_frontier** works on the compound resource portion of each element in C_{ic}. By setting ϵ 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** with zero refinement; 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*[3] (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 $O(nL)$, and Steps 27 through 35, 36 through 38 require $O(nL)$. The total running time of the algorithm is, therefore, $O(nL \log L) + O(nL \log n) + O(nL) + O(nL) = O(nL \log nL)$.

In the next section, we apply the algorithms **amrmd1** and `MRMD` to a range of test problems to evaluate their effectiveness.

5 Performance Evaluation

In this section, we present a detailed performance evaluation of the dynamic programming, integer programming and `amrmd1` algorithms discussed in the previous section.

5.1 The Nature of the Experiments

The experiments we conducted were as follows. For each task set, we generated given number of task profiles, each with the following properties:

- The number of QoS options was given.
- The resource usage for the QoS options were generated randomly, but consistently, i.e., more resouce would not lead to lower quality.
- The utility associated with each QoS options was likewise generated randomly, but consistently.

The three `MRMD` algorithms were then run on this task set 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 repeat 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. This provides us with a good degree of cross-validation of correctness with respect to our implementations of our schemes.

5.2 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 run times can be rather large.

Figure 1 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 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[4]. The number of tasks to which these two resources must be allocated is

[3] Overmars & Leeuwen's [10] algorithm, the quickhull [11] or Graham-Scan [1].

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

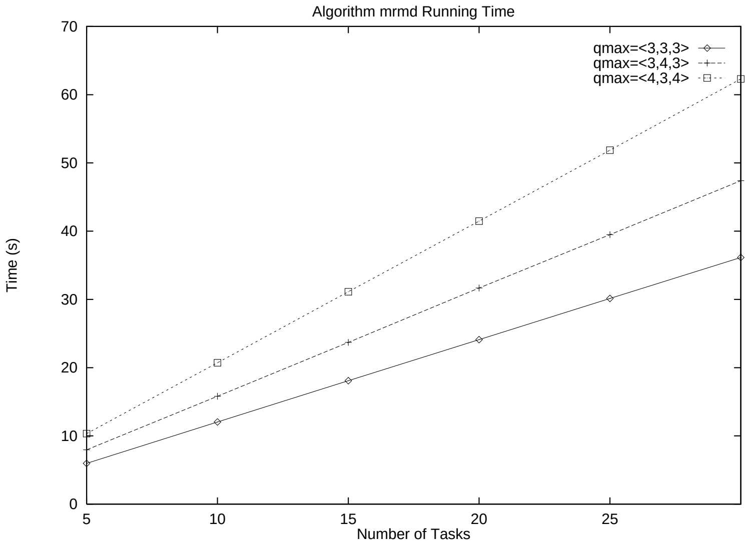

Figure 1: The Run times of Algorithm `mrmd` with $r^{\max} = \langle 180, 100\rangle$

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 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 4.1.

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 to make online decision-making in real-time systems is highly questionable.

5.3 Performance of the Integer Programming Scheme

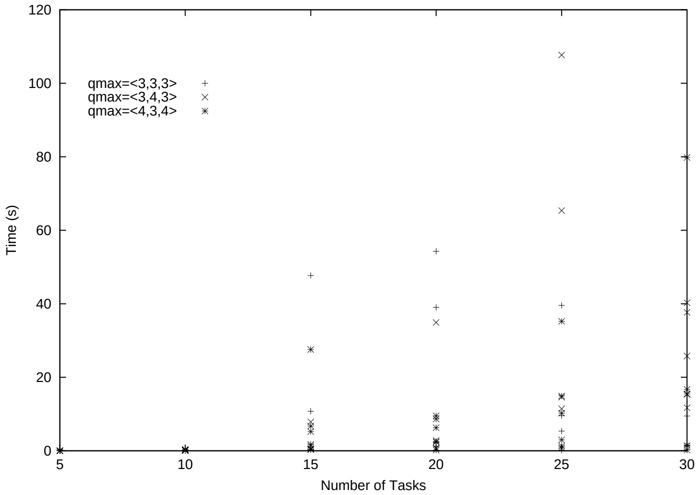

Figure 2: Running Times for Computing the Optimal Solution using Mixed Integer Programming

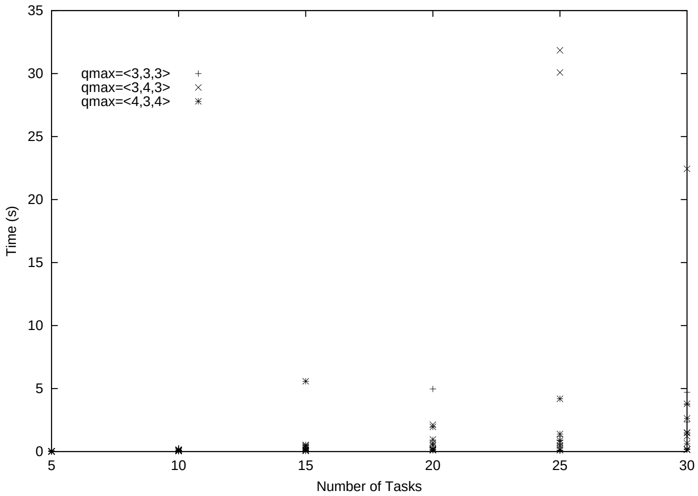

Figure 3: Running Times for Computing Solutions using Mixed Integer Programming and a Specified Maximum Deviation from the Optimal Solution

The CPU times consumed by the mixed integer programming package `CPLEX` on three-dimension, two-resource problems are shown in Figure 2. An option called `SOS` (Special Ordered Set) files was used to group sets of related variables and set weights on members of the sets. The graph plots the running times to find an optimal solution for each of the five problems 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 search space.

Optimality Thresholds. In order to reduce the running times while still maintaining high-quality results, an optimality threshold can be specified. The optimality threshold indicates that the solution will be within a fixed bound of the optimal solution. The running times for the same problem set with an optimality threshold of 5% is shown in Figure 3. By applying this threshold, the worst-case running time was reduced to 31.85 *secs* versus 107.69s 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 4. All of the solutions in our problem set were more than 96.95% of the optimal solution.

Running Time Thresholds. If a strict upper bound on the solution time is required, a time-out can also be set. When the time limit for a problem has expired, the currently available best solution is returned. The solution quality for a 3-second timeout is shown in Figure 5. Even with a 3 second 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.

5.4 Performance Evaluation of `amrmd1`

We now evaluate the performance of the `amrmd1` algorithm. Figures 6 and 7 correspond to the same set of tasks used to plot Figure 1 (i.e. $r^{\max} = \langle 180, 100 \rangle$, $n = \{5, 10, 15, 20, 25\}$).

Figure 6 plots the ratio of the solution quality obtained by `amrmd1` to the optimal solution obtained by the `mrmd` dynamic programming algorithm. Two conclusions are of immediate interest.

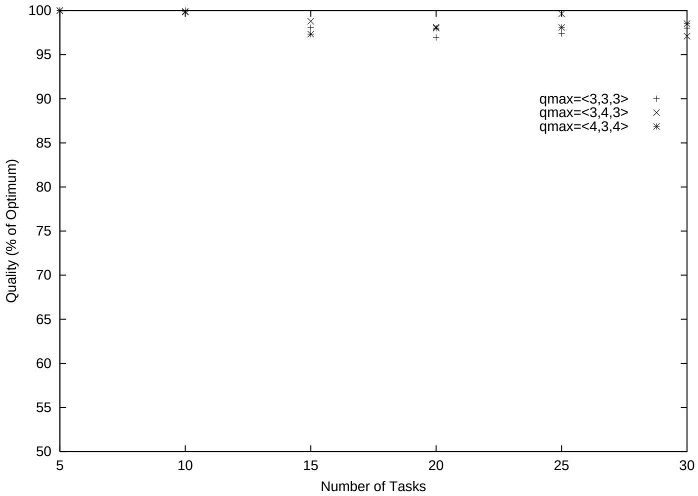

Figure 4: Solution Quality using Mixed Integer Programming and a Specified Maximum Deviation from the Optimal Solution

The first is from Figure 6 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 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 8 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 log_2 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 9 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, 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 single *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 is the number of tasks which are considered for allocation.

5.5 Comparative Evaluation of `amrmd1` & Integer-Programming

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

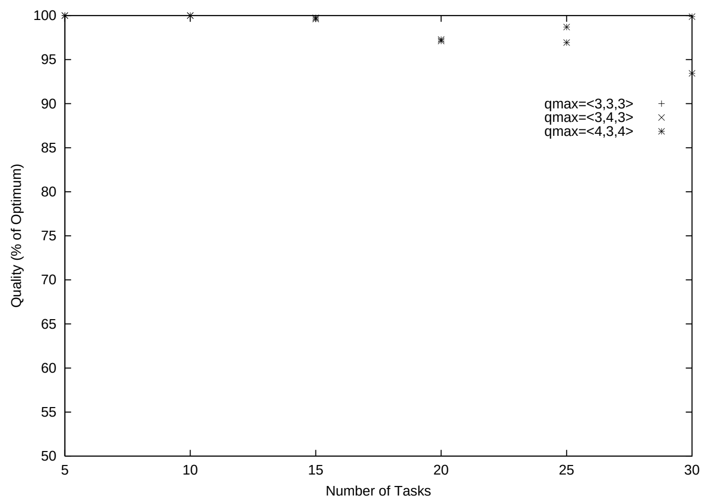

Figure 5: Solution Quality with Timeouts in Mixed Integer Programming

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.

6 Concluding Remarks

Real-time and multimedia systems must deal with a host of QoS dimensions including audio fidelity, video frame-rate, sampling rate, algorithmic precision, and end-to-end latency. Distributed applications such as video-conferencing, internet telephony and air traffic control also require the use of multiple resources at end-hosts and on intermediate network links. In this paper, we have studied the general problem of apportioning *multiple* finite resources to satisfy the QoS needs of multiple applications along multiple QoS dimensions. Each application derives some utility as resources are allocated to it and its QoS requirements can be satisfied to a greater or lesser degree. Our objective was to maximize the utility derived by all the applications in the system. This problem is shown to be NP-hard. We have then presented, evaluated and compare three strategies to solve this problem. Two traditional approaches, dynamic programming and mixed integer programming, are used to compute optimal solutions to this problem but we show that their running times are rather high (as might be expected). An adaptation of the mixed integer programming problem, however, yields near-optimal results with (potentially) significant lower running times. Finally, we present and evaluate an approximation algorithm based on a *local search* technique which combines multiple resources into a single compound pseudo-resource. This scheme yields a solution quality that is less than 5% away from the optimal solution but is shown to run more than two orders of magnitude faster. In addition, the use of the "compound resource" allows this technique to be very scalable and robust as the number of resources required by each application increases.

This work can progress along several future directions. First, we are currently implementing this optimization model in the Amaranth test-bed at Carnegie Mellon. Secondly, the current model has assumed that one pre-defined set of resources is used by each application (perhaps in different quantities based on the implementation chosen). This, for example, corresponds to the use of a static route between two nodes on a network. This assumption needs to be relaxed in the future.

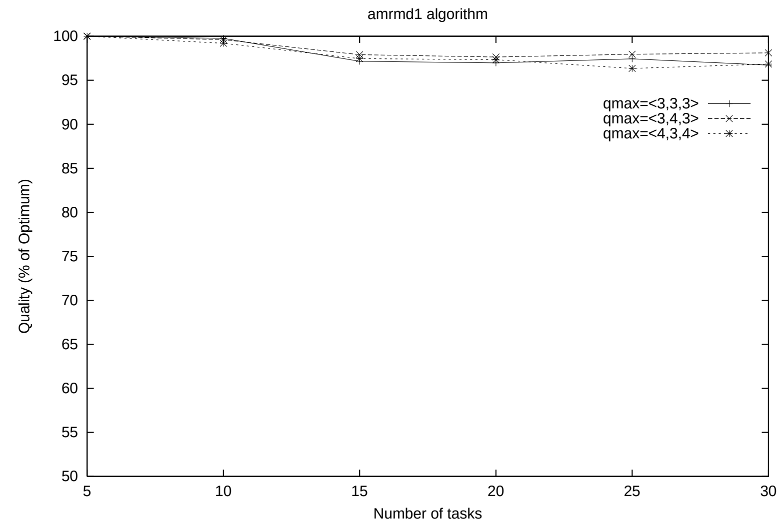

Figure 6: Solution Quality obtained by `amrmd1` with $r^{\max} = \langle 180, 100 \rangle$

7 Acknowledgement

The authors would like to thank Professor John Hooker at the Graduate School of Industrial Administration of CMU for providing invaluable advice on this work.

A Sample Task Profiles and Solution Results

A.1 Sample Task Profiles

```
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>]
      ...
      [<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> 0.171055 [<5,7>,<7,3>]
<1,1,2> 0.353209 [<6,8>,<9,3>]
<1,1,3> 0.535371 [<7,9>,<11,3>]
<1,1,4> 0.535371 [<9,11>,<14,4>]
```

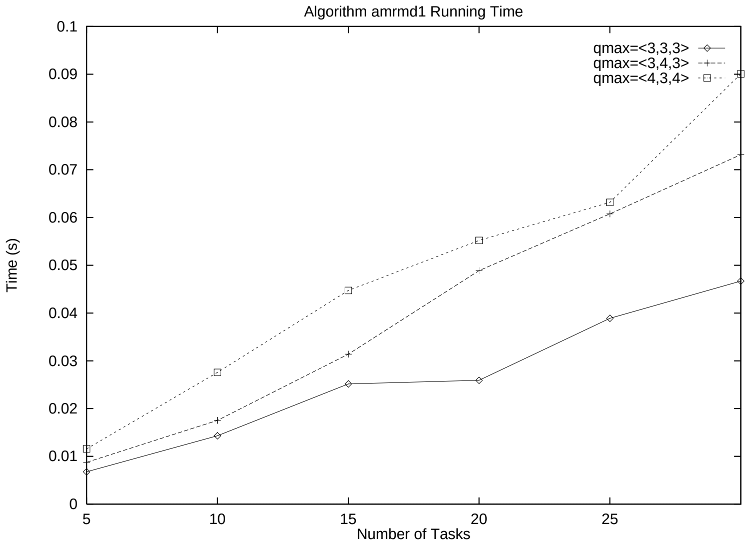

Figure 7: Running Times of Algorithm `amrmd1` with $r^{\max} = \langle 180, 100 \rangle$

```
...
<4,3,1> 0.635684 [<12,17>,<14,6>]
<4,3,2> 0.817838 [<14,20>,<17,7>]
<4,3,3> 1 [<18,23>,<21,7>]
<4,3,4> 1 [<22,27>,<26,8>]
```

A.2 Sample Results

Due to length limitations, we list only one assigned quality point for each task (multiple quality points which consume the same amount of resources exist for some tasks).

The following example consists of 15 tasks, an $r^{\max}$ of $\langle 180, 100 \rangle$ and a $q^{\max}$ of $\langle 4, 3, 4 \rangle$.

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

Algorithm mrmd:
Task 0 : (<12,7>,0.8979)
```

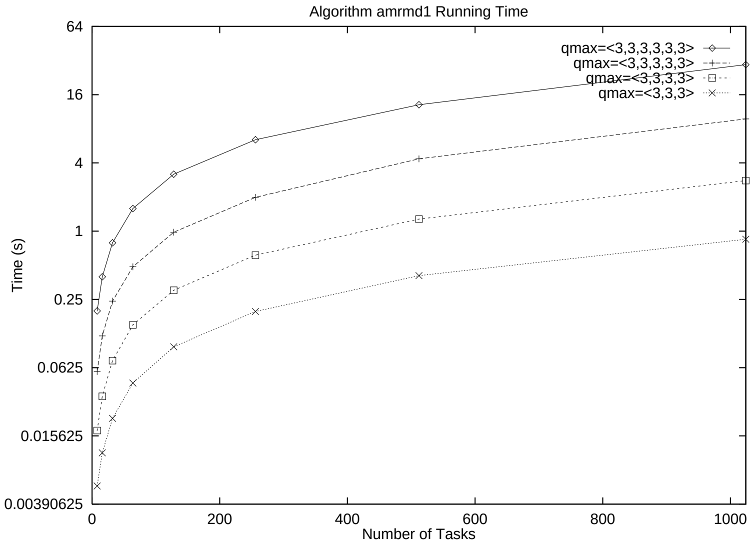

Figure 8: Running Times of Algorithm `amrmd1` with the number of resources $(m) = 3$ and varying the number of QoS Dimensions

```
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>)
```

References

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

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

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

[4] 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.

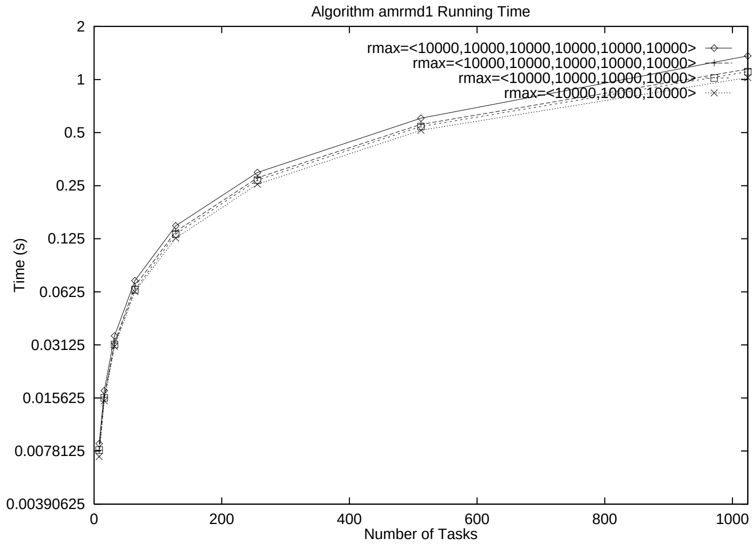

Figure 9: Running Times of Algorithm `amrmd1` with the number of QoS dimensions (d) = 3 and varying the number of resources

[5] D. Le Gall. MPEG: A Video Compression Standard for Multimedia Applications. *CACM*, 34(4):46–58, April 1991.

[6] 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.

[7] 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.

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

[9] J. Mitchell, D. Le Gall, and C. Fogg. *MPEG Video Compression Standard*. Chapman & Hall, 1996.

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

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

[12] 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.

[13] 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.

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

[15] D. Seto, J. Lehoczky, L. Sha, and K. Shin. On Task Schedulability in Real-Time Control Systems. In *IEEE Real-Time System Symposium*, December 1996.