Scalable Resource Allocation for Multi-Processor QoS Optimization

Sourav Ghosh* Ragunathan (Raj) Rajkumar† Jeffery Hansen‡ John Lehoczky§

Abstract

We present scalable QoS optimization algorithms for allocating resources to tasks in a multi-processor environment. Given a set of tasks, each of which is capable of running at one of several different QoS levels, the algorithms can select a QoS operating point, the number of replicas for fault-tolerance and the processors on which to run the replicas so as to maximize overall system QoS. The algorithms are extensions of techniques used by Q-RAM (QoS-based Resource Allocation Model) [5] and fix two deficiencies with a baseline algorithm. The first deficiency is that the existing algorithm is weak in making resource trade-off decisions as to which processor to map a task. The second is that it does not scale to very large numbers of resources. In this paper, we present two new algorithms which significantly enhance the ability of Q-RAM to make resource trade-off decisions. We also introduce a hierarchical decomposition scheme which enables QoS optimization techniques to be performed on large problems with thousands of resources and tens of thousands of tasks.

1 Introduction

1.1 Motivation

The quality of the outputs achieved in distributed real-time and embedded systems depends heavily on the amount of resources available to them. For example, in multimedia applications such as audio and video, higher network and CPU bandwidth produce better audio/video quality at higher resolutions and/or higher frame rates. And in feedback control and radar tracking systems, more resources allow a task to operate at higher sample rates and thus operate more smoothly or track targets more accurately.

In addition to the performance aspects of quality, applications need to be reliable or consistent in providing outputs as well. In other words, they should also be tolerant of faults in the system. The idea of fault-tolerance in the form of active or passive replication has been studied (see for example [6]) in great detail. Many fault-models

*Department of Electrical and Computer Engineering,Carnegie Mellon University, sourav@ece.cmu.edu

†Department of Electrical and Computer Engineering, Carnegie Mellon University, raj@ece.cmu.edu

‡Institute for Complex Enigeneering Systems, Carnegie Mellon University, hansen@cmu.edu

§Department of Statistics, Carnegie Mellon University, jpl@stat.cmu.edu

have been presented for determining the *necessity* of having a certain number of replicas of tasks [7], but not in conjunction with a QoS optimization framework. In our QoS framework, we treat fault-tolerance as an additional aspect of QoS. The degree of fault-tolerance can be equated to the degree of redundancy or replication offered to the applications by the system. Replication enables the application to provide reliable output even when one (or more) copies of the same application fail(s).

While there are existing algorithms for resource allocation with QoS optimization [5], and algorithms such as *bin-packing* and *load-balancing* [8] for mapping tasks with fixed resource requirements to fixed sets of resources, there are no existing algorithms which perform both of these two functions simultaneously. In this paper, we address this problem with new algorithms for combining QoS optimization with resource selection.

One other problem with existing QoS optimization algorithms is that they are not scalable to very large numbers of resources and tasks. We present a new hierarchical decomposition technique for solving these very large optimization problems. The hierarchical technique subdivides the problem into smaller similar subproblems, and then solves these similar subproblems individually, reusing the results of identical subproblems when applicable. The result is a two or more orders of magnitude reduction in execution time for large optimization problems.

1.2 Q-RAM: QoS-based Resource Allocation Model

The QoS architecture [1] we consider is composed 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 allows multiple QoS requirements to be specified. The trade-off model allows the applications and the users to assign quantitative values called "utilities" to the different levels of service. Finally, a QoS resource manager makes resource allocations to those applications so as to maximize the global utility as a weighted sum of the utilities of the individual applications.

In [1],[2], [4] and [3], we presented a QoS management framework that enabled system developers and application developers to quantitatively define QoS, and to analytically plan and allocate resources. The system resources are distributed among multiple applications of different quality levels such that the net utility that accumulates to the end-users is maximized.

Using this architecture, the problem of maximizing system utility by allocating a *single* finite resource and *multiple* finite resources to satisfy the QoS requirements has been studied in [1] and [5] respectively. It was also proved in [5] that an optimal solution for allocation of multiple resources is NP-hard.

In this paper, we consider a more refined problem of apportioning multiple resources with many *choices* on resource-types. We then present three solutions to the problem. All these solutions are the variants of the local search technique for *multiple-resource and multiple-dimension* scheme presented in [5].

1.3 Organization of the paper

The rest of the paper is organized as follows. In Section 2, we describe a QoS model and formulate the QoS optimization problem being solved. In Section 3, we first discuss the old Q-RAM multi-resource optimization algorithm and demonstrate how this solution fails to satisfy our requirements. In Section 4, we introduce our enhanced algorithms and compare their performance with the basic algorithm. In Sections 5 and 6, we introduce the theory underlying hierarchical resource optimization and provide simulation results demonstrating its effectiveness on solving very large problems. Finally, we draw some conclusions and discuss future work in Section 7.

2 System Model

In this section, we provide a complete overview of the resource allocation model that we employ, including a brief description of the entities and the metrics used in [5].

2.1 Tasks, System Resources and Quality of Service

We consider a distributed system with m shared resources $r_1, \ldots, r_m$. Resources can be of any type including CPU, memory, link bandwidth, or even radar bandwidth in the case of a radar tracking application. Each of the shared resources has a maximum quantity denoted by $r^{max} = (r_1^{max}, \ldots, r_m^{max})$.

The resources are shared by a set of n independent tasks $T_1, \ldots, T_n$. Each task is described by a set of QoS dimensions and a set of trade-offs. For example, a video conferencing application might have QoS dimensions related to security, performance (resolution, frame rate, etc.) and fault tolerance (e.g., redundant streams). Users derive *satisfaction* through these various qualities. The higher the quality along any dimension, the higher the satisfaction. We quantify this in the form of a parameter called *utility.* The value of this parameter along different quality dimension depends on the application as well as the user.

Trade-offs are used to model application options which affect the resource requirements of the application, but are not directly visible to the user. Examples of trade-offs might include selecting a processor from a set of processors to map an application, deciding between a high CPU/low bandwidth versus a low CPU/high bandwidth video codec and choosing a network path through which to route a flow. Generally there are no utility curves associated with trade-offs as the actual option selected is transparent to the user.

The *Application Profile* of a task defines the relation between the quality perceived by the user and the resources required to achieve that quality. It consists of a *QoS Profile* and a *Resource Profile.* The *QoS Profile* of a task defines the relationship between the QoS level of service of an application, and the perceived benefit to the user. It is defined in terms of the following components:

***quality space*:** The set of quality points $Q_i = Q_{i1} \times \cdots \times Q_{id_i}$ for task T_i where Q_{ij} is the j^{th} QoS dimension and d_i is the number of QoS dimensions.

***quality indices*:** An index in $\{1, 2, ..|Q_{ij}|\}$ enumerating the quality levels for dimension j of task T_i arranged in increasing quality for each dimension.

***dimension-wise utility*:** The mapping $u_{ij} : Q_{ij} \rightarrow \Re$ representing the utility achieved by assigning quality level q_{ij} to dimension Q_{ij}.

***application utility*:** The mapping $u_i : Q_i \rightarrow \Re$ representing the utility achieved by assigning quality level q_i to task T_i. Normally, this is simply the weighted sum of the dimension-wise utilities.

The *Resource Profile* of a task determines the relationship between the QoS level of service of an application and the resources required to achieve that level of service. It is defined in terms of the following components:

***option space*:** The set of trade offs $V_i = V_{i1} \times \cdots \times V_{ic_i}$ for task T_i where V_{ij} is the j^{th} trade off dimension and c_i is the number of trade-off dimensions.

***option indices*:** An index in $\{1, 2, ..|V_{ij}|\}$ enumerating the possible options for trade-off dimension j of task i.

***set-point space*:** The set $S_i = Q_i \times V_i$ of all possible operating points for a task T_i. We define a single point in the space a *set-point*.

***resource mapping*:** The function $g_i : S_i \rightarrow R$ mapping each set-point s_i to a resource requirement r_i.

For a task T_i, all QoS dimensions Q_{ij} must satisfy the condition $\forall_{k \in \{1,\ldots,m\}} \frac{\partial r_k}{\partial q_{ij}} \geq 0$. That is, an increase in any quality index value never results in the decrease in any resource requirement value. Likewise, all trade-off dimensions V_{ij}, must satisfy the conditions $\exists_{k \in \{1,\ldots,m\}} \frac{\partial r_k}{\partial v_{ij}} < 0$ and $\exists_{k' \in \{1,\ldots,m\}} \frac{\partial r_{k'}}{\partial v_{ij}} > 0$. That is, some option within a trade-off results in taking resources from one resource dimension and adding them to another resource dimension.

As an example, consider a video conference application with QoS and trade-off dimensions as shown in Table 1. There are two QoS dimensions, frame rate and resolution with weights 0.4 and 0.6. The weights represent the relative importance of the QoS dimensions from the user's perspective. For frame rate, there are three possible service levels at 10 frames/sec., 20 frames/sec. and 30 frames/sec. Associated with each of these service levels, there is a quality index with 1 for the lowest level of service, and 3 for the highest level of service. The user of the application has assigned utility values to each of these levels of service indicating the relative desirability of these service levels. Similar quality index and utility values are assigned for the various resolutions.

In addition to the QoS dimensions, there are also two trade-off dimensions. One of the trade-offs dimensions is the format, or codec, to use for the actual video data. In this example, we can use either the NV, the CELLB, or the H.261 video format. Since each of these formats does differing amounts of compression, some of them will consume substantial CPU while reducing the network bandwidth required, while others will minimize the use of CPU at the expense of network bandwidth. Each of the options is assigned an option index, but ordering does not matter. Furthermore, since the selected video format does not directly impact the user, there are no distinct

QoS Dimensions (weight)	Level of Service	Quality Index	Utility
Frame rate (0.4)	10 frames/sec	1	0.2
	20 frames/sec	2	0.6
	30 frames/sec	3	1.0
Resolution (0.6)	176x144	1	0.1
	352x288	2	0.8
	704x576	3	1.0
Trade-Off Dimensions	**Option**	**Option Index**	
Codec	NV	1	
	CELLB	2	
	H.261	3	
Path	Links B-C-D	1	
	Links B-E-F	2	

Table 1. Example of Quality and Trade-Off Space for an Application

utility values associated with any of the options. The other trade-off is the actual path through the network used to connect the two endpoints. In the first option, links B,C and D are used, while in the second option links B,E and F are used. Again, while the actual network path selected does not directly affect the user, it may impact on available system resources.

Using the concatenation of the quality and option indices (q_i, v_i), each possible set-point of the application can be assigned a unique vector. For example, the set-point $(3, 2, 3, 1)$ would represent the video conference application running at 30 frames/sec., 352x288 resolution, using the H.261 video codec, and routing the packets for the video flow along links B, C and D. The resource required for this set-point would be determined by applying the resource mapping function g_i. This function may be provided by the application developer or QoS engineer. The utility for a set-point is normally determined as the weighted sum of the dimension utilities. In this case, the utility for the set-point $(3, 2, 3, 1)$ would be $(0.4 \times 1.0) + (0.6 \times 0.8) = 0.88$.

The *system utility* for a set of tasks is the *combined* utilities of all the applications. Normally it is defined as the weighted sum of all application utilities for "*differential services*", but can also be defined as the minimum of the utilities among the applications for "*fair*" sharing.

Trade-off MRMD (Multi-Resource Multi-Dimension) Allocation Problem: For a set of of tasks $T_1, \ldots, T_n$, assign quality indices q_{ij} to each QoS dimension Q_{ij} and option indices v_{ij} to each trade-off V_{ij} such that the system utility is optimized and no resource utilization exceeds its maximum. Formally we write this as:

maximize $u(q_1, ..., q_n)$

subject to

$$q_i \geq q_i^{min} \qquad \text{(QoS Constraints)}$$

$$\sum_{i=1}^{n} r_{ik} \leq r_k^{max},\ k = 1, \ldots, m \qquad \text{(Resource Constraints)}$$

$$r_i = g(s_i),\ i = 1, \ldots, n. \qquad \text{(Mapping Constraints)}$$

where q_i^{min} is the minimum acceptable QoS for task T_i as specified by the user. Solving the optimization problem results in assigning specific set-points to each task. Note that a set-point s_i has a quality component q_i and a trade-off component v_i.

2.2 Replication and fault-tolerance

A task is made fault-tolerant through its replication. A replica will run on *different* resources relative to the original at the same quality level. The number of replicas that need to be run depends on the fault model of the system. Following a certain fault-model, we can represent fault-tolerance as another QoS dimension. The values along this dimension are the number of copies (replicas) of the task. The utilities achieved depend on the fault-model or the reliability of the system.

For example, consider a task i that has the following resource vector allocation choices (options): r_{i1},r_{i2} and r_{i3}. At the same level of quality, any of these resource choices can be allocated to the task. In order for the task to be fault-tolerant, more than one resource vector needs to be allocated. Thus, we can generate the QoS set-points in the following way:

Fault-tolerance Quality Index	Number of replicas	Resource Vectors
1	0	r_{i1},r_{i2}, r_{i3}
2	1	$(r_{i1}+r_{i2})$,$(r_{i1}+r_{i3})$,$(r_{i2}+r_{i3})$
3	2	$(r_{i1}+r_{i2}+r_{i3})$

For a task with N resource vector options, the fault-tolerant quality index of M can be attained in $\begin{pmatrix} N \\ M \end{pmatrix}$ combinations of resource vectors. This automatically limits the maximum number of replicas to the number of *independent* resource options.

3 MRMD Algorithms

In [5], it was demonstrated that the QoS optimization problem involving multiple resources (MR) and multiple QoS dimensions (MD) is NP-hard. An exact optimal solution to the problem based on dynamic programming and an approximation scheme based on the *local search* technique was then presented.

In this section, we first discuss the limitations of that scheme in addressing problems with resource trade-offs(which we call *resource options*) and then present our algorithms to address the problem.

3.1 Previous Approximate MRMD (`amrmd1`)Algorithm

The *local search* technique presented in [5] can be briefly described as follows. Recall that n denotes the number of tasks and m denotes the number of resources. Let

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

represent the discrete set of utility-resource pairs for task T_i.

We define a penalty vector for the resources (assuming we have m resources) $p = (p_1, ..., p_m)$,where $p \in [1, \infty)$ in order to "price" each resource combination. The values of an element in the vector is *directly* related to the overall demand of the corresponding resource.

Hence we define the scalar metric r^* as,

$r^* = \|r_p\| = \sqrt{(r_{p1})^2 + ... + (r_{pm})^2}$

where $r_p = (r_1.p_1, ..., r_m.p_m)$ be the penalized resource vector.

This r^* is used as a scalar metric to compare each resource combination. Using this, we augment C_i by adding r^*:

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

We call C_{ic} a compound resource vector. Now, we briefly describe the algorithm `amrmd1` (approximate multi-resource multi-dimension algorithm) from [5].

- Calculate the initial penalty vector.
- Repeat the following procedure for a specified number of (e.g. 3) times:
  - Generate the compound resource vector C_{ic}
  - Perform *convex hull frontier* optimization [5] on the compound resource
  - Create a list *s_list* by merging all C_{ic} of the tasks
  - Go through the entire *s_list* and enter/update the resource allocation of the tasks
  - After the procedure is finished, update the penalty vector from the usage of the individual resources
- Finalize resource allocations of the tasks

In short, this algorithm tries to compute the compound resource vector of a resource based on the relative demand or usage of that particular resource. Then, the procedure `convex_hull_frontier()` works on the compound resource portion of each element in C_{ic} and retains the points falling on the line of the highest slopes.

The slope is defined as the following equation:

$$slope_j = (C_{(ic)(j)}.u - C_{(ic)(j-1)}.u)/(C_{(ic)(j)}r^* - C_{(ic)(j-1)}r^*)$$

where r^* = compound resource, u = utility.

3.2 Drawbacks of the Amrmd1 Algorithm

One problem with the algorithm is that there can be many *co-located* points that have the same values of utilities but use different resource vectors (or resource combinations). The algorithm will arbitrarily choose only *one* out of those co-located points after the convex-hull procedure and eliminate others completely from consideration. This decision may not be right simply because during the course of the resource allocation process, one point may be infeasible while another co-located point with the same utility is feasible. This depends on the status of the current allocation of resources. As a result, `amrmd1` may stop allocating resources even though additional feasible allocations exist.

Secondly, the `amrmd1` algorithm statically computes the "penalty" vector. It is determined based on the demand on system resources and penalizes resources which are heavily loaded in favor of less loaded resources. This demand is determined by summing the resource requirements of all set-points of all the tasks. In a true sense, the computation of penalty vectors of the individual resources should reflect the real usage of the resources at any given point in time during resource allocation. In other words, the penalty vector may be computed dynamically each time a set-point gets admitted, based on the quality points that have already been admitted into the system so far.

Let us consider an example. We have a system consisting of 2 processors. Let the computational capacity on each node be 10 units. There are two tasks, each with the following QoS specifications:

Quality Index	Processing requirements	Resource Vectors	Utility
1	3	(3,0),(0,3)	0.3
2	5	(5,0),(0,5)	0.5
3	7	(7,0),(0,7)	0.7

The QoS specification indicates the resource requirements for a task on each processor. According to the above specification, each task has *two* options on resource requirements at each QoS level. The utility values are chosen as shown.

As we can be easily seen, the resource demand on the two nodes are completely balanced and hence the elements in the penalty vectors are completely identical. This produces a pair of co-located set-points at every QoS level.

From the diagram, it is rather obvious that an *optimal* utility maximizing solution is the following:

Task	QoS level	Resource Vector	Utility
0	3	(7,0)	0.7
1	3	(0,7)	0.7

In other words, each of these tasks must be allocated on *different* processors so that they can derive the maximum possible QoS. The system utility achieved is $0.7 + 0.7 = 1.4$

But since `amrmd1` does not keep co-located set-points nor does it compute the penalty vector dynamically, it may end up producing the following solution:

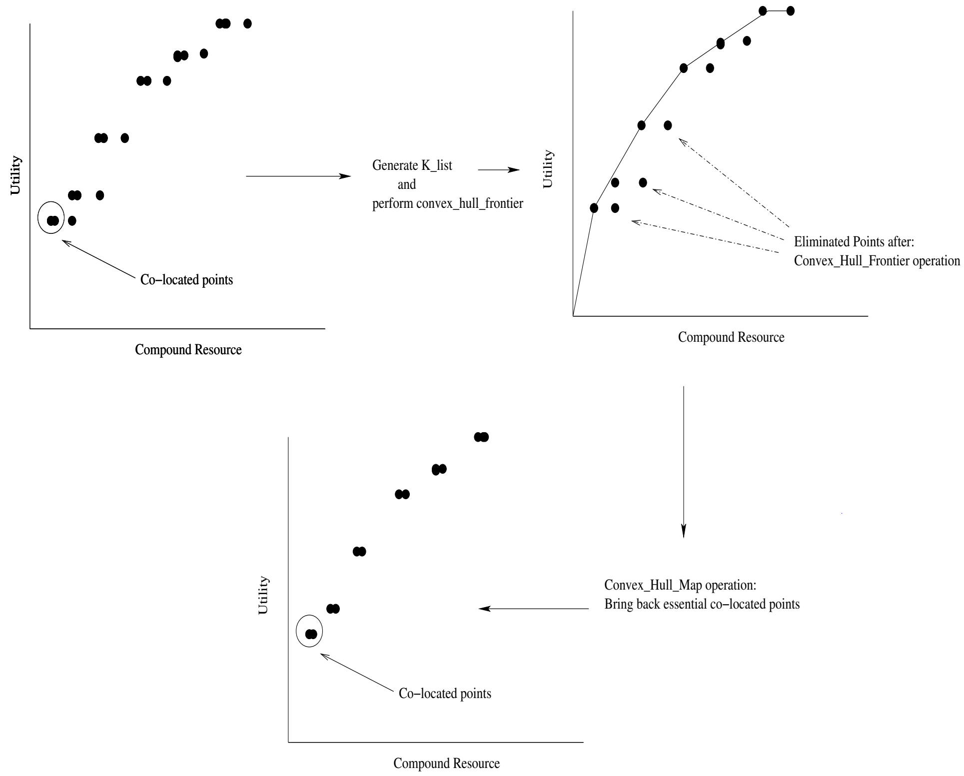

Figure 1. Amrmd_cm Algorithm

Task	QoS level	Resource Vector	Utility
0	3	(5,0)	0.5
1	3	(5,0)	0.5

This yields a total utility of $0.5 + 0.5 = 1.0$.

Based on the above observation, the original `amrmd1` algorithm can lead to sub-optimal solutions. In the next two sections, we will discuss two new algorithms that overcome this limitation in `amrmd1`.

3.3 Amrmd Algorithm for Resource Options

We now describe two new algorithms that address the limitations of the `amrmd` algorithm described earlier.

3.3.1 Amrmd Algorithm with Co-Mapping of Quality Points (`Amrmd_cm`)

This algorithm retains all the co-located set-points which yield the same utility value but with different resource vectors, and is otherwise similar to `amrmd1`. From the C_{ic} lists, we create another compound resource list K_i by including the elements from C_{ic} that only have distinct values of r^*. In other words, if two or more elements in C_{ic} have the same scalar value for r^* but different vector values for r, then they are co-mapped to a single element in K_i list.

Each element in K_i also stores the indices of the corresponding elements in C_{ic}.

Amrmd_cm then performs the convex_hull_frontier operation on this K_i list instead of the C_{ic} list and maintains the set of co-located points of same utility if they lie on the convex hull. It then follows the convex hull allocating resources and accruing system utility (as amrmd1 does). However, unlike amrmd1, it attempts to allocate one of the co-located points if a point is infeasible due to resource constraints.

The process of forming a K_i list and the corresponding retrieval of the *relevant* co-located points are illustrated in Figure 1. In step(1), co-located points are gathered. In step(2), the convex hull is determined. In step(3), only points (and their co-located points) along the convex hull are used for making resource allocation decisions.

The asymptotic computational complexity of amrmd_cm can be obtained as follows. Let $L = max_{i=1}^{n}|C_i|$. The procedure initial_penalty takes $O(nL)$ operations. After procedure create_Klist (requires $O(nL)$ operations), convex_hull_frontier (requires $O(nLlog(L))$ operations) and convex_hull_map (requires $O(nL)$ operations) respectively, a convex hull frontier with non-increasing slope segments is obtained for each task. The segments are merged by the divide-and-conquer approach. The merging requires $O(nLlogn)$ comparisons [5]. The steps for the QoS-point selection requires $O(nL)$ steps. The total running time of the algorithm is therefore, $O(nL) + O(nLlogL) + O(nL) + O(nLlogn) = O(nlLognL)$, which is the same as the amrmd1 algorithm. However, the presence of co-located points in amrmd_cm makes L somewhat higher than that of amrmd1.

Applying the algorithm to the example of Section 3.2, we obtain the optimal result with the utility of 1.4 using this algorithm.

3.3.2 Amrmd Algorithm with Dynamic Penalty Vector Computation (Amrmd_dp)

An alternative algorithm, amrmd_dp, dynamically computes the penalty vectors for the resources as it keeps allocating QoS-points for the tasks. It works as follows. It forms the C_i list for each task.

Without performing the *convex hull frontier*, it computes the marginal utility of the first point in C_i of each task, as the slope of the compound resource/utility curve. Marginal utility is defined as:

Marginal utility = gain in utility of the point under test with respect to the currently allocated point / increase in compound resource requirement = $\Delta U/\Delta r*$

It then selects the point of the highest marginal utility to be allocated next. If the allocation is successful, it updates the penalty vector. This assignment then requires an update in the compound resource parameters C_i of all tasks containing the remaining set-points. Thus, the points migrate from one location to another during the progress of the algorithm.

Then it repeats the same procedure until all points in the C_i lists of all the tasks are exhausted.

Figure 2 illustrates the process.

The asymptotic computational complexity of amrmd_dp is as follows. Assuming the previous parameters, the procedure initial_penalty takes $O(nL)$ operations. Within the loop, the procedure for updating compound resource takes $O(nL)$ operation, the procedure for selecting a point takes $O(nL)$ operations and the procedure for adjusting penalty takes $O(nL)$ operations. This loop can repeat nL times in the worst case. These steps yield the

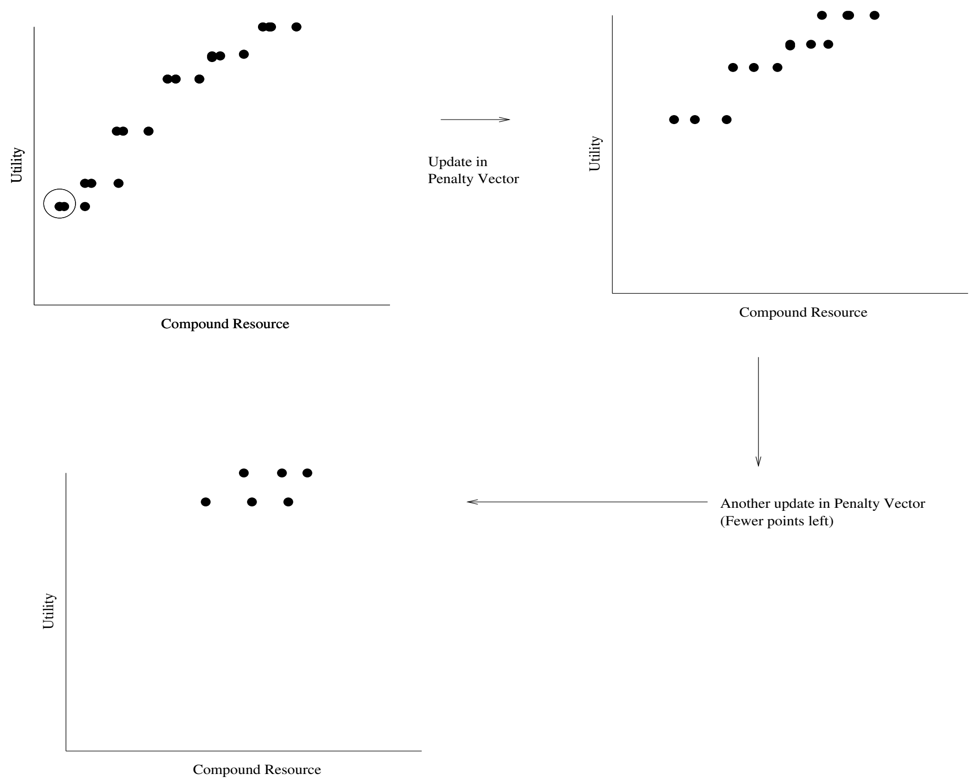

Figure 2. Amrmd_dp Algorithm

total complete running time as,

$O(nL) + nL(O(nL) + O(nL) + O(nL)) + O(nL) = O(n^2L^2)$

As can be seen, this algorithm has a higher degree of complexity than `amrmd1` and `amrmd_cm`. However, while discussing the experimental results, we shall see this does not always lead to a significant run-time cost.

If we apply `amrmd_dp` to the same example, we obtain the optimal solution in this case as well.

4 Performance Evaluation of Algorithms

In this section, we present a detailed performance evaluation of the algorithms `amrmd_cm` and `amrmd_dp`.

4.1 The Nature of the Experiments

For a given number of task profiles and resources or processors, our experiments focused on measuring the following performance metrics:

- The maximum number of tasks that can be admitted while satisfying minimum QoS requirements of all the admitted tasks.
- The utility obtained with the maximum number of admitted tasks when its minimum QoS is zero or non-zero.
- The execution time of the algorithms.

Number of QoS dimensions q	2
Length of each dimension	3
Utilities for QoS dimension ($u(q)$)	(0.5,0.7,0.8)
Weight for each QoS dimension	between (0.00,1.00)
Minimum resource for each task	between (1,3) units
Resource increment for higher QoS	between (1,2) units
Number of processors	5
Resource amount per processor	10 units

Table 2. Settings for the experiment for comparison optimal algorithm

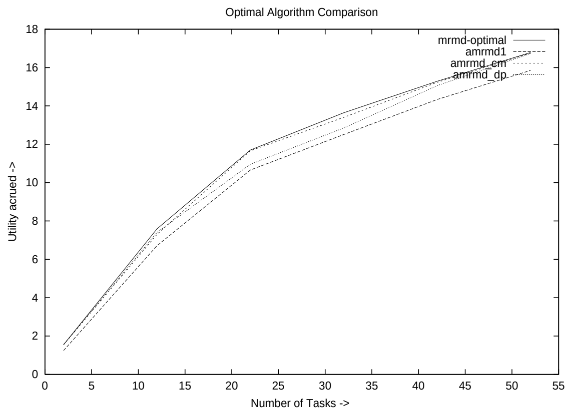

Figure 3. Utility variation with number of tasks (settings of Table 2

4.2 Comparison with the Optimal Exhaustive Search Algorithm

Our first experiment deals with the comparison among all the algorithms, including the optimal exhaustive search algorithm called `mrmd`. A small multi-processor system consisting of 5 processors is assumed for the sake of convenience of running the exponentially complex optimal algorithm. We present the specifications used in this set of experiments in Table 2.

Figure 3 shows the variation of utility as the number tasks is varied from 2 to 52. The result is averaged over 50 runs. It shows that `amrmd_cm` performed closest to the optimal `mrmd` scheme in terms of utility, `amrmd1` being the farthest.

Figure 4 plots the variation of the execution time of the algorithms. The results are plotted in log-scale as the optimal algorithm runs approximately $30,000$ times slower than `amrmd1`. The execution times for `amrmd_cm` is somewhat higher than those of `amrmd1` as expected (about 1.5 times slower). The algorithm `amrmd_dp` has quadratic complexity and thus runs slower than the other two algorithms.

The results show `amrmd_cm` yields utility values closest to that of the optimal algorithm with a somewhat higher execution time than `amrmd1`.

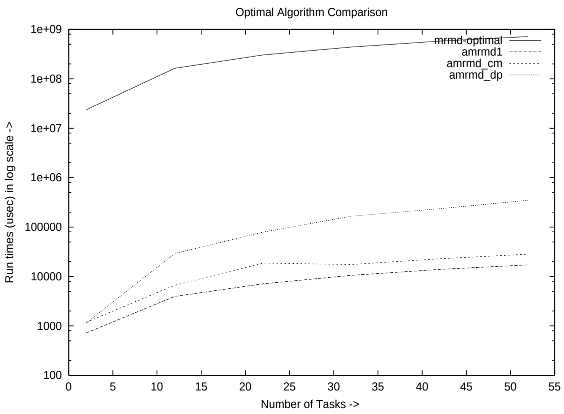

Figure 4. Run-time variation with number of tasks (settings of Table 2

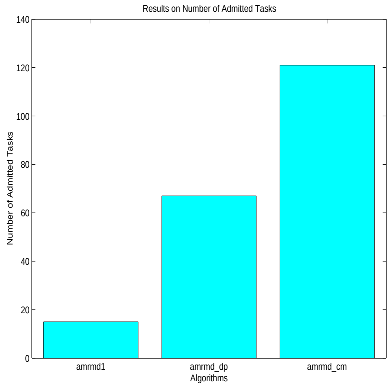

Figure 5. Number of tasks admitted on 20 processors (settings of Table 3)

4.3 Results on Multi-processor Resource Allocation

In this experiment we consider with a system with 20 processors according to the experimental set-up shown in Table 3. We compared these three algorithms in terms of the maximum number of tasks admitted into the system based on the utility maximization and the variation of utilities and execution times relative to the number of tasks. Each task has non-zero QoS requirements that must be satisfied for admission into the system.

The results for the maximum number of tasks that can be admitted under each algorithm, averaged over 100 randomly generated problems are shown in 5. As can be seen, `amrmd_cm` is able to admit 6 *times* more tasks than `amrmd1` and twice as many than `amrmd_dp`. From Figure 6, we observe a very high relative standard deviation of the performance of `amrmd1`, the least being that of `amrmd_cm`.

Next, we vary the number of tasks and determine the utility accrued under each algorithm. In this case, we assume all the tasks have a zero minimum QoS requirements and thus all of them can admitted. The results are plotted by averaging the results over 100 randomly generated problems (Figures 7 and 8). As expected, the results show higher utility for `amrmd_cm` over `amrmd1`.

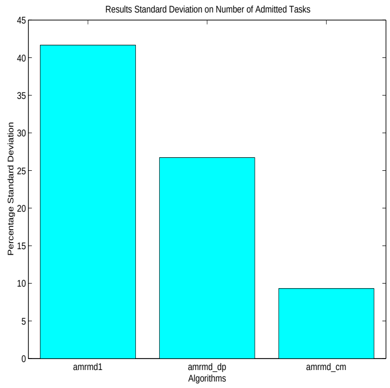

Figure 6. Percentage standard-deviation of number of admitted tasks on 20 processors (settings of Table 3)

Number of QoS dimensions (q)	2
Length of each dimension	3
Utilities for each quality dimension ($u(q)$)	(0.5,0.7,0.8)
Computational resource on each processor	100 units
Minimum resource for each task	between (1-25) units
Resource increment for higher QoS	between (1,10) units
Number of processors	20
Resource amount per processor	100 units

Table 3. Settings for the second experiment

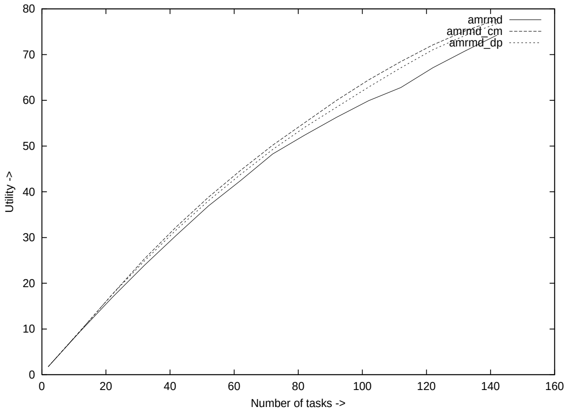

Figure 7. Utility variation with number of tasks (settings of Table 3)

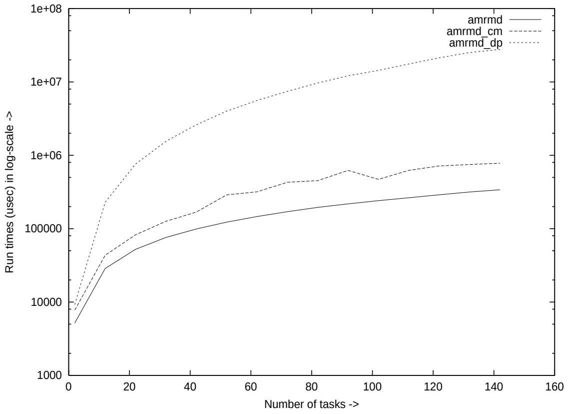

Figure 8. Run-time variation in log-scale with number of tasks (settings of Table 3)

Number of QoS dimensions	1
Number of copies for a task	1-2
Number of quality indices along the (FT) dimension(= number of copies)	2
Utilities along the dimension	(0.65 0.80)

Table 4. Settings for the experiment on Fault-tolerance

4.4 Results on Fault-tolerance

The notion of incorporating fault-tolerance in the QoS-based Resource allocation Model (Q-RAM) was explained in Section 2.2. We apply fault-tolerance by using replication on multiple processors. The higher the degree of replication, the higher the utility obtained. The tasks are assumed have fault-tolerance as the only QoS dimension in this case for the sake of illustration. Table 4 shows the experimental specifications.

The results shown in Figures 9 and 11 demonstrate that the `amrmd_cm` algorithm outperforms the other two with respect to the number of tasks it admits and the utility it achieves for a fixed number of tasks. The number of tasks it typically admits is 2 times to as high as 6 times that of `amrmd1`. Essentially, if tasks can be admitted with zero QoS (i.e., they are “rejected”), `amrmd1` can force a utility maximization fairly well. However, if all incoming tasks must be admitted, `amrmd1` performs poorly compared to `amrmd_cm`.

5 Large Number of Resources: Hierarchical Q-RAM

None of the algorithms we have presented so far scale well when there is a large number of resources. For example, in multi-dimensional radar systems, 64 or more processing nodes are common. Under these conditions, all three algorithms can consume large amount of computation time and memory, making them usable only for offline purposes. This scalability bottleneck arises because the factor L increases proportionately with the number of resources (same or different) in the system. This becomes inefficient as each task is allocated to only one resource unit (processor) out of many, unless fault-tolerance is required. Even under fault-tolerance, the number of resource

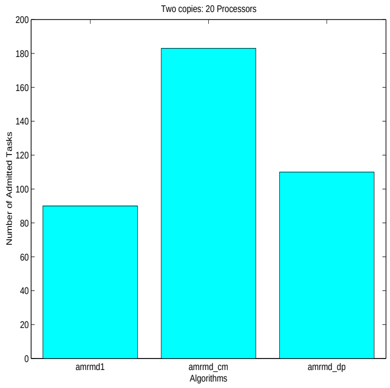

Figure 9. Number of tasks admitted on 20 processors under Fault-Tolerance (settings of Table 4)

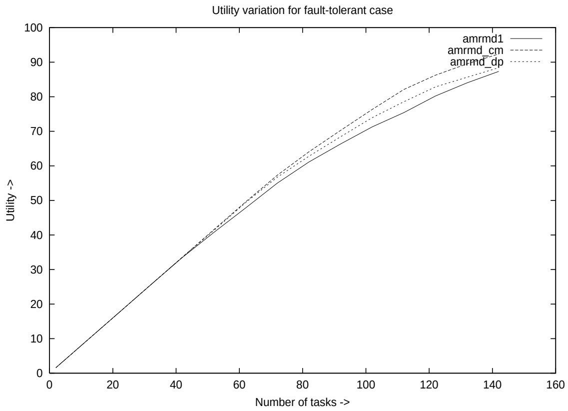

Figure 10. Utility variation with number of tasks under Fault-Tolerance(settings of Table 4)

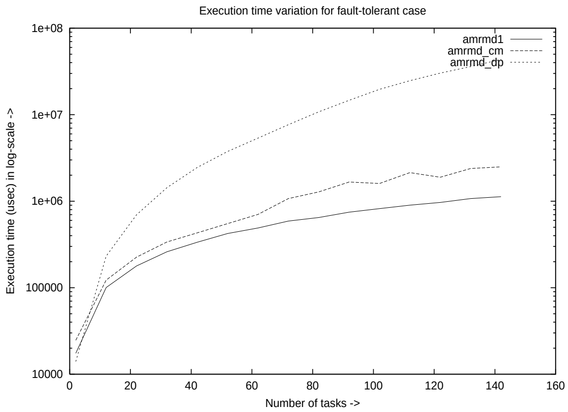

Figure 11. Run-time variation in log-scale with number of tasks under Fault-Tolerance (settings of Table 4)

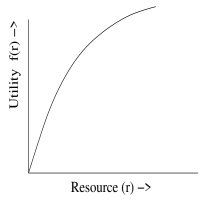

Figure 12. A typical utility function

units assigned is equal to the number replicas needed for a task. We thus propose a hierarchical Q-RAM approach in which we partition the problem into smaller sub-problems dealing with smaller number of resources.

Before we discuss the details of our approach, we state a theorem based on theorems of constrained extrema in Linear programming [13].

5.0.1 Theorem 1:

If a resource has to be distributed among identical tasks[1] *with* **continuous** *non-decreasing concave utility function, the total utility is maximized when each task is allocated an* **equal** *amount of resource (hence equal utility).*

Proof: The proof of this theorem follows directly from the theorems of constrained extrema [13]. Let us suppose we have to allocate a resource of amount R among n identical task with utility-resource function $f(r)$. The resources allocated to the tasks are $r_1, ..., r_n$. The allocation is optimal when the total utility is maximized. Hence we have the following problem:

$$Maximize : F(r_1, .., r_n) \equiv f(r_1) + .. + f(r_n) \tag{1}$$

$$Subject\ \ to : G(r_1, .., r_n) \equiv r_1 + .. + r_n = R \tag{2}$$

The *Lagrangian* of the above function can be constructed as:

$$L(r_1, .., r_n; \lambda) = f(r_1) + ... + f(r_n) + \lambda\{R - (r_1 + .. + r_n)\} \tag{3}$$

In order to determine the constraints of the critical point (i.e., optimal allocation of resources), we have the following equations (from theorems of constrained extrema):

$$\begin{gathered} \frac{\partial f(r_1)}{\partial r_1} = \lambda \\ \dots\dots\dots\dots \\ \frac{\partial f(r_n)}{\partial r_n} = \lambda \end{gathered} \tag{4}$$

From the above equations and the *monotonic increasing* curve of $f(x)$, we conclude:

$$\frac{\partial f(r_1)}{\partial r_1} = ... = \frac{\partial f(r_n)}{\partial r_n} = \lambda \tag{5}$$

[1] identical Task profile

$$\Rightarrow \quad r_1 = ... = r_n \tag{6}$$

Now, in order to prove the above allocation provides a maxima instead of a minima, we get the determinant of the Hessian matrix as,

$$H = \begin{vmatrix} 0 & -1 & .. & -1 \\ -1 & \frac{\partial^2 f(r_1)}{\partial r_1^2} & .. & 0 \\ \multicolumn{4}{c}{\dots\dots\dots\dots\dots\dots} \\ -1 & 0 & .. & \frac{\partial^2 f(r_n)}{\partial r_n^2} \end{vmatrix} = -(\frac{\partial^2 f(r_1)}{\partial r_1^2} + .. + \frac{\partial^2 f(r_n)}{\partial r_n^2}) > 0 \tag{7}$$

from the characteristic of the utility function.

Hence $r_1 = ... = r_n$ corresponds to the maximum value of total utility.

5.0.2 Corollary 1:

If a resource has to be allocated among various tasks with a fixed number of types with continuous utility functions, the maximum utility is obtained when the same resource amount is allocated to tasks of the same type.

This can be proved using the above theorem. Let us assume that we have k types of tasks with utility functions $f_1(x), .., f_k(x)$. The number of tasks for each types are $p_1, .., p_k$ respectively. Therefore, we have:

$$Maximize : F(r_1, .., r_n) \equiv \{f_1(r_{11}) + .. + f_1(r_{1p_1})\} + .. + \{f_k(r_{k1}) + .. + f_k(r_{kp_k})\} \tag{8}$$

$$Subject\ \ to : G(r_1, .., r_n) \equiv (r_{11} + .. + r_{1p_1}) .. + (r_{k1} + .. + r_{kp_k}) = R \tag{9}$$

Following the same procedure as before, we obtain:

$$\begin{aligned} &\tfrac{\partial f_1(r_{11})}{\partial r_{11}} = .. = \tfrac{\partial f_1(r_{1p_1})}{\partial r_{1p_1}} = \lambda \\ &\dots\dots\dots\dots\dots\dots\dots\dots \\ &\tfrac{\partial f_k(r_{k1})}{\partial r_{k1}} = .. = \tfrac{\partial f_k(r_{kp_k})}{\partial r_{kp_k}} = \lambda \end{aligned} \tag{10}$$

And therefore,

$$\begin{aligned} &r_{11} = ... = r_{1p1} \\ &\dots\dots\dots\dots \\ &r_{k1} = ... = r_{kp_k} \end{aligned} \tag{11}$$

The above proof also shows that the optimal result is obtained at a point where the slopes of the utility functions of all the tasks are *equal*.

5.0.3 Corollary 2: on Discrete Utility Function:

If a resource has to be distributed among identical tasks with **discrete** *utility functions, the total utility is maximized when the resources are distributed equally among as many tasks as possible until the remaining resource becomes too small to be usable.*

Proof: Consider Theorem 1 when the task utility functions are not continuous but discrete. If two identical tasks T_i and T_j have resource values r_i and r_j respectively, sum of their utility becomes: $f(r_i) + f(r_j)$. If we *assume* $r_i + r_j = 2r$ and $f(r)$ is a utility point of the task type, we have the gain in utility for this resource distribution over *equal* distribution as:

$$\delta f = f(r_i) - f(r) + f(r_j) - f(r) \tag{12}$$

From the nature of the convex utility curve (after `convex hull frontier`), since $r_i > r > r_j$, we have,

$$\frac{f(r_i) - f(r)}{r_i - r} < \frac{f(r) - f(r_j)}{r - r_j}$$

$$\Rightarrow f(r_i) - f(r) < f(r) - f(r_j) \Leftarrow r_i + r_j = 2r$$

$$\Rightarrow f(r_i) + f(r_j) < 2f(r)$$

$$\Rightarrow \delta f < 0 \tag{13}$$

Please note that (13) one may not be true if $r_i + r_j > 2r$. In other words, if extra resources can be allocated to one task or more task, that have to be *recursively equally distributed* to those tasks until no task can utilize the remaining resource.

5.0.4 Algorithm for Hierarchical `Amrmd`

Corollaries 1 and 2 guide us in designing a very efficient QoS allocation scheme when tasks fall into a finite set of categories. Consider the following algorithm,

- Cluster the resources/processors into a finite number of groups. Let us assume the number of groups to be p.
- Divide the tasks of each type into p identical groups. If the number of tasks of a type is not an integral multiple of p, keep the *remaining tasks* un-allocated.
- Each of the p identical groups of tasks is assigned to a distinct resource cluster, of which there 'p' clusters.
- Allocate the remaining tasks in the following way:
  - Compute `Convex Hull Frontier` on tasks assuming a single resource to be used for each task
  - Order the tasks with their *initial slope* from the *utility-resource* curve

Number of task types ($TaskType$ variable)	8 (0,1,2,3,4,5,6,7)
Utilities on QoS dimension in tasks	[0.5,0.7,0.8] or [0.4,0.6,0.65] or [0.6,0.7,0.75]
Minimum resource per task	$24 - 2TaskType$
Resource increment for higher *Quality*	$random(16 - 2TaskType, 20 - 2TaskType)$
Percentage of task of each type:	12.5% each on average
Number of processors	32
Number groups formed	$1, 2, 4, 8$ and 16

Table 5. Specifications for Hierarchical Q-RAM Experiments

- Choose a task based on the highest initial slope, and allocate it to the group *least* populated with the *same* task type.
- Run any of the `amrmd` algorithm of choice on each of the processor groups.
- Run the selected algorithm only *once* for multiple identical groups of tasks and apply the same result to all identical groups with identical composition.

6 Performance Evaluation of Hierarchical Q-RAM

In this section, we evaluate the scalability of hierarchical Q-RAM to perform resource allocation in multi-processor systems. In this case, we assumed the presence of *eight*[2] types of tasks in the system. The number of tasks belonging to each type is uniformly distributed. The specifications of the tasks are detailed in Table 5.

6.1 Multi-processor Resource Allocation

This experiment deals with tasks that have no Fault-tolerant QoS specifications. The parameter values are presented in Table 5. The 32 processors in the system are divided equally into $1, 2, 4, 8$ and 16 groups of processors. Hence, each group has $32, 16, 8, 4$ or 2 processors,with the grouping of 1 being equivalent to non-hierarchical `amrmd`. We measure the maximum number of tasks admitted under each of the 5 configurations (plot them on a bar-graph in Figure 13). The result is the average over 100 randomly generated problems.

The results of Figure 13 show that `hierarchical amrmd` under each grouping is able to admit nearly the same number of tasks as the non-hierarchical `amrmd`. The maximum drop in the number of admitted tasks was 5% (from 291 in the non-hierarchical version to 276 in the hierarchical version with 16 groups).

Next, we keep the number of tasks constant at 276 (the maximum that can be admitted by groups of 16 processors) and measure the execution time under each case. The result is averaged over 100 iterations and is presented in Figure 14. It shows a very sharp drop in execution time as the number of groups is increased. For example, the execution time non-hierarchical `amrmd` is 73 times that of groups of 16.

[2]A typical radar system may have eight priority levels of tasks.

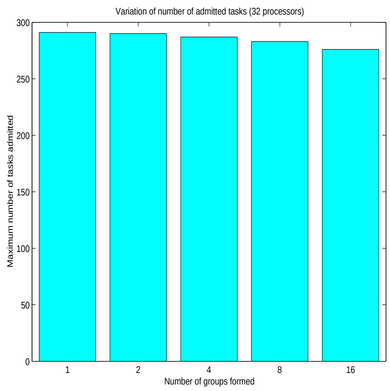

Figure 13. Number of tasks admitted on 32 processors (with/without grouping)

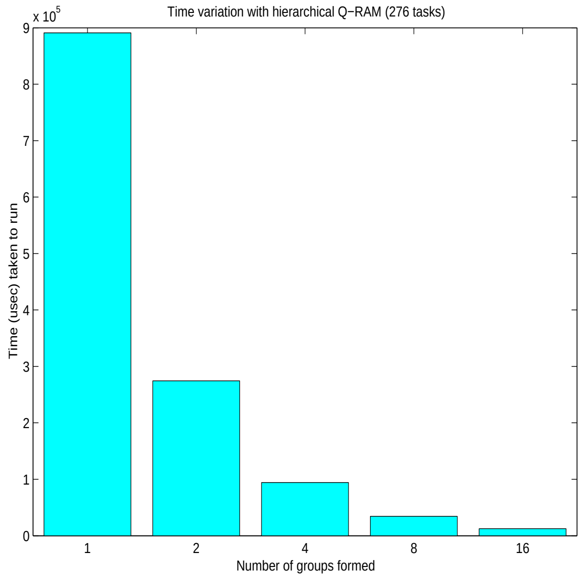

Figure 14. Run-time variation with grouping for 32 processors (total 276 tasks)

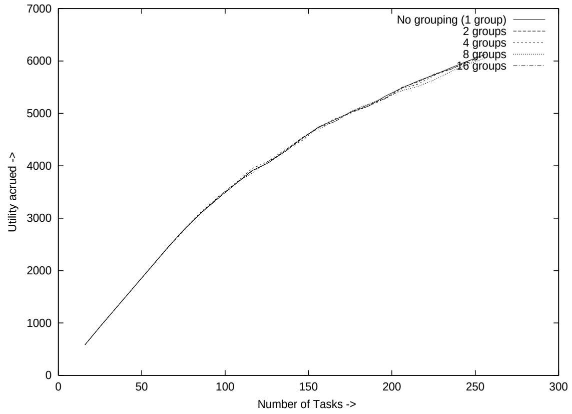

Figure 15. Utility plot with grouping for 32 processors (max 256 tasks)

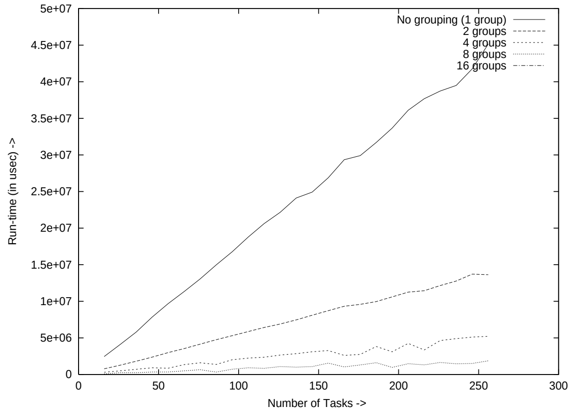

Figure 16. Run-time plot with grouping for 32 processors (max 256 tasks)

Next, we vary the number of input tasks from 16 to 256 and plot the utility and execution time of the algorithms against the number of tasks, as shown in Figures 15 and 16. The results show a negligible difference in utilities for a fixed number of tasks. It also shows a huge drop in execution time as the number of groups is increased from 1 to 16.

6.2 Fault-tolerance and Hierarchical `Amrmd`

In this experiment we consider tasks having fault-tolerance as the only QoS dimension. The specifications for the fault-tolerance are taken from Table 4 of the non-hierarchical case. Other configuration parameters are maintained same as in the previous experiments.

As observed from Figures 17 and 18, substantially fewer tasks are admitted with 16 groups. This is simply because each group of 16 has only 2 processors and the maximum number of replicas of the fault-tolerance is 2, thereby significantly reducing the trade-off options. On the other hand, we observe 37000X reduction in the execution time as the grouping is increased from 1 to 16 (Figures 18 and 20). In addition, all of the groupings produce a near identical utility curve when we vary the number of tasks from 16 to 76, as shown in Figure 19.

In summary, hierarchical Q-RAM obtains a very high system utility while reducing the execution time up to

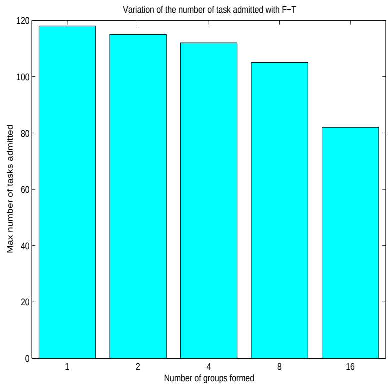

Figure 17. Number of tasks (under Fault-tolerance) admitted on 32 processors (with and without grouping)

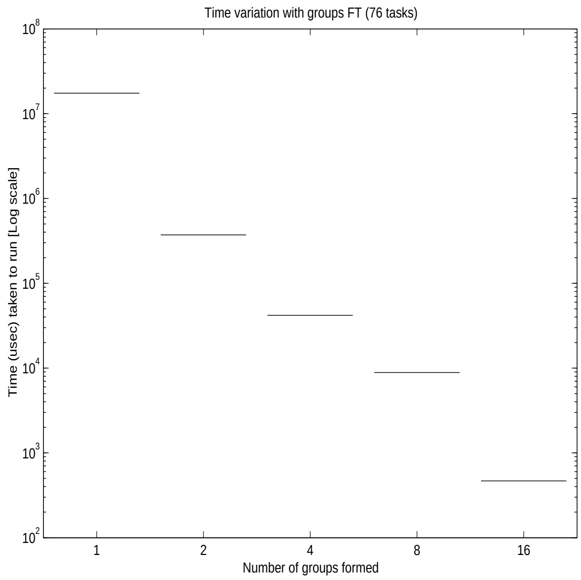

Figure 18. Run-time variation in log-scale with/without grouping (under Fault-tolerance) with on 32 processors and 76 tasks

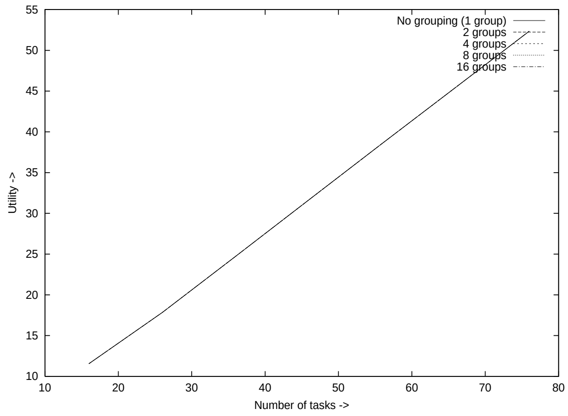

Figure 19. Utility plot with grouping for 32 processors under fault-tolerance (max 76 tasks)

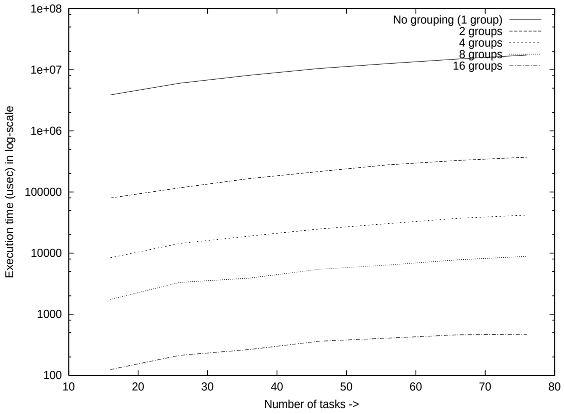

Figure 20. Run-time plot in log-scale with grouping for 32 processors under fault-tolerance(max 76 tasks)

two orders of magnitude or more.

7 Summary and Future Work

In this paper, we investigated extensions to Q-RAM to apply it to distributed multi-processor systems. We presented two schemes `amrmd_cm` and `amrmd_dp` for solving problems involving both resource allocation and resource selection. We showed that the `amrmd_cm` was able to achieve much larger global utility values when compared to the baseline algorithm with only a small increase in the execution time.

Next, we also presented a scheme for integrating fault-tolerance and Q-RAM-based optimization in a single framework. We showed that we can make trade-off decisions between performance and the number of replicas while selecting processors on which to run each of the replicas. We noted that providing fault-tolerance as a QoS dimension impacted both the basic Q-RAM algorithm and both of our enhanced algorithms in roughly the same way with respect to both execution time of the algorithms and the achievable global utility.

Finally, we showed that none of these algorithms scale well when allocating resources on large multi-processor systems. For example, a radar tracking system may consist of a bank of 64 or more processors for signal processing tasks. We then presented a hierarchical decomposition approach for applying our QoS optimization algorithms to such systems. We showed that this hierarchical approach significantly reduced the execution time for all of the algorithms. In particular, resource allocation problems involving fault tolerance as a QoS dimension become scalable as a result of our hierarchical approach.

In this paper, we focused on resource trade-off problems involving tasks that are mapped to one (or in the fault tolerance case, several) processors from a pool of processors. An extension to this is to consider problems involving non-independent resource trade-offs where a task may be required to choose one of several possible subsets of resources. An example of this is a network system when choosing a path through the network. In the path selection problem, we need resources for each link in the path, but the links we choose for a path are not arbitrary and depend on the network topology. We will address this problem in a future paper.

References

[1] C. Lee, J. Lehoczky, R. Rajkumar, and D. Siewiorek. On quality of service optimization with discreet qos options. In *Proceedings of the IEEE Real-time Technology and Applications Symposium.*, June 1999.

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

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

[4] C. Lee. On Quality of Service Management. *PhD Thesis*, Aug 1999.

[5] C. Lee and J. Lehoczky and D. Siewiorek and R. Rajkumar and J. Hansen. A Scalable Solution to the Multi-Resource QoS Problem. In *Proceedings of the IEEE Real-Time Systems Symposium*, Dec 1999. '

[6] P. Banerjee and J.A. Abraham. Fault-Secure Algorithms for MultiProcessor Systems. In *Proc. 11th International Symp. on Computer Architecture*, June 1984.

[7] P. Chevochot, I. Puaut. Scheduling Fault-Tolerant Distributed Hard real-Time Tasks Independently of the Replication Strategies. In *Proc. of the 6th International Conference on Real-Time Computing Systems and Applications*, Dec 1999

[8] B. Shirazi, A.R. Hurson, K.M. Kavi. Scheduling and Load Balancing in Parallel and Distributed Systems, ISBN 0-8186-6587-4, March 1995.

[9] Y. Oh, S.H. Son. Fixed-Priority Scheduling of Periodic Tasks on Multiprocessor Systems. In *Technical Report 95-16, Department of Computer Science, University of Virginia*, March 1995.

[10] B. Andersson, J. Jonsson. Fixed-Priority Preemptive Multiprocessor Scheduling: To Partition and not to Partition. In *Proc. of the Int'l Conf. on Real-Time Computing Systems and Applications*, Dec 2000.

[11] B. Natarajan, J. Cross, A. Gokhale, D.C. Schmidt, C. Andrews, S. Fernandez and C. Gill. Towards Dependable Real-time and Embedded CORBA Systems. In *Proceedings of the IEEE Workshop on Dependable Middleware-Based Systems*, June 2002.

[12] P. Bizzarri, A. Bondavalli, F. Di Giandomenico and F. Tarini. Planning the Execution of Task Groups in Real-Time Systems. In *Proc. 8th Euromicro Workshop on Real-Time Systems*, 1996.

[13] R.C. Walker. Introduction to Mathematical Programming, Prentice Hall.