Resource Management of Highly Configurable Tasks *

Jeffery P. Hansen
Institute for Complex Engineered Systems
Carnegie Mellon University
Pittsburgh, PA 15213
Email: *hansen@cmu.edu*

Sourav Ghosh
Dept. of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
Email: *sourav@cs.cmu.edu*

Ragunathan Rajkumar
Dept. of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
Email: *raj@ece.cmu.edu*

John Lehoczky
Dept. of Statistics
Carnegie Mellon University
Pittsburgh, PA 15213
Email: *jpl@stat.cmu.edu*

Abstract

In this paper we present an extension to our QoS optimization algorithm, Q-RAM[6][7], that can improve optimization time by several orders of magnitude when managing highly configurable tasks. A highly configurable task is one with a large number of QoS dimensions and/or a large number of quality levels on those dimensions. For example, an application that has ten QoS dimensions with ten quality levels each will have 10^{10} setpoints, or ways in which it can be configured. While the existing Q-RAM algorithm has been shown to be a very effective resource management tool, it must still explicitly perform computations on all of the setpoints for each task. For tasks with 10^{10} setpoints or more, this is clearly impractical. The key idea presented here is a new approximation algorithm for the convex majorant step in Q-RAM. By using this algorithm in a filtering step, the best performing subset of the setpoints can be quickly found without explicitly examining all of the setpoints. The idea is validated using a phased array radar system as an example application.

1 Introduction

Traditional QoS optimization algorithms assume that a collection of tasks compete for resources, with each task receiving some benefit from those resources. The goal is to allocate the resources to tasks in such a way as to optimize the total benefit received by all the tasks. This benefit is often called "utility"[5]. A larger QoS for a task generally requires a larger amount of resources and results in larger utility. Furthermore, the QoS-utility curve usually saturates at high QoS levels with further increases in QoS yielding smaller increases in utility [7].

Many applications have parameters that can be adjusted to tune the QoS and the resource demands of the application. Each parameter has a fixed number of possible values and we call each combination of parameter value settings a "setpoint". The basic QoS optimization goal is to choose a setpoint for each application so as to maximize the total benefit to the system. As the number of parameters increases, the number of setpoints increases

*This work was supported by the DoD Multidisciplinary University Research Initiative (MURI) program administered by the Office of Naval Research under Grant N00014-01-1-0576

dramatically. For example if we have an application that has 10 parameters with 10 possible settings for each parameter then there would be 10^{10} different setpoints. Clearly this is problematic for any optimization algorithm if each setpoint of each application must be explicitly examined.

In this paper we will show that in many cases it is possible to prune the setpoint space without explicitly examining most of the points in that space. The new algorithms presented here are extensions to the QoS-based Resource Allocation Model (Q-RAM) presented in [7][4][3]. Q-RAM is a QoS management framework that enables system developers and application developers to quantitatively define QoS, and to analytically plan and allocate resources. The system resources get distributed among multiple applications at different quality levels such that the net utility that accrues to the end-users is maximized.

The technique will be illustrated using a phased-array radar tracking application as an example. The radar system is modeled as a set consisting of a radar antenna, a shared power source, and an bank of processors to run signal and track processing algorithms. Resources modeled in the system include not only traditional resources such as CPU and radar bandwidth, but constraint-model resources such as energy utilization.

2 QoS Model

Let us assume a distributed system with a set of shared resources and a set of n independent tasks $T_1, \ldots, T_n$. Each task is assumed to have a set of parameters that can be changed to configure its QoS and resource demands. We will call these *operational dimensions*. Some operational dimensions may also be *QoS dimensions*. A QoS dimension is a single aspect of task quality that is of direct relevance to the user; for example, the frame rate of a videoconference application. Other operational dimensions may be transparent to the user with respect to quality, but may affect resource demands. For example, the selection of a video coding algorithm, or the path through the network. Still other operational dimensions may affect both quality and resource demand, but may not be of direct relevance to the user. An example is the dwell period of a radar tracking task. While a shorter dwell period will increase both tracking quality and resource demand, the dwell period itself is not directly of interest to the user. In such applications, the QoS dimensions may not be directly aligned with the operational dimensions. In fact, the task quality may even depend on factors in the environment in addition to the operational setting of a task. For example, in a radar tracking task the tracking quality can depend not only on the configuration of the task, but on factors such as the distance to the target as well as its speed and type.

Because of the possibile disparity between the QoS objectives of the user and the configuration options of the task we separately define an *operational space*, a *quality space* and an *environment space*. For a task T_i we define the operational space as:

$$V_i = V_{i1} \times \cdots \times V_{iN_i^V} \tag{1}$$

where V_{ij} is the jth operational dimension, we define the *quality space* as:

$$Q_i = Q_{i1} \times \cdots \times Q_{iN_i^Q} \tag{2}$$

where Q_{ij} is the jth QoS dimension, and we define the *environment space* as:

$$E_i = E_{i1} \times \cdots \times E_{iN_i^E} \tag{3}$$

where E_{ij} is the jth environment dimension. We also define a shared *resource space* as:

$$R = R_1 \times \cdots \times R_m \tag{4}$$

where R_k is the kth shared resource (or resource dimension).

For some types of tasks, the operational dimensions and quality dimensions may be equivalent and there may be no environment dimensions (e.g., a videoconferencing task), but in general we say that there is a *quality function*

$f_i : V_i \times E_i \to Q_i$ mapping each point in the cross product of the operational space and environment space to a point in the quality space. We assume that the operational dimensions are discrete having a fixed number of possible values for each dimensions. The QoS and environment dimensions may or may not be discrete.

Rather than concern ourselves with the semantics of the values that can be taken on by the operational dimensions, we assume each operational dimension can be characterized by an index value between 1 and $v_{ij}^{\max}$. For operational dimensions that affect application quality (e.g., frame-rate) we assume 1 to be the lowest quality with increasing index values representing increasing quality. For other types of operational dimensions (e.g., the video compression algorithm to use) the mapping between the index value and its semantics is arbitrary. We call each possible combination of index values $\{v_{i1}, \cdots, v_{iN_i^V}\}$ a *setpoint*. The number of setpoints for a task T_i is $\prod_{j=1}^{N_i^V} v_{ij}^{\max}$.

For each task, the user may specify a *utility function* $u_i : Q_i \to \Re$ mapping each point in the quality space with a real number which we call the *task utility*. Usually task utility is defined in terms of the weighted sum of a set of dimension-wise utility functions $u_{ij} : Q_{ij} \to \Re$. The *system utility* is then simply the sum of the individual task utilities. Note that for a given environmental condition e_i, it is straightforward to map a setpoint v_i in the operational space to a utility value using the quality function f and the utility function u. Using this mapping, a resource manager can only concern itself with selecting a setpoints in the operational space and need not directly consider points in the quality space.

In order for a task to operate at a particular setpoint v_i, it generally requires resources. We define a function $g_i : V_i \to R$ specifying the amount of resources required for a task to operate at each setpoint. We also define a quantity r_i^* we call *compound resource* in terms of a compound resource function $h : R \to \Re$ mapping a resource requirement of a task to a value representing the overall demand on the system for a particular set of resource requirements. A typical compound resource function[4] is:

$$h(R) = \sqrt{\sum_{k=0}^{m} (R_k P_k)^2} \tag{5}$$

where P is a *penalty vector* representing the relative scarcity of the resources. For brevity, we will also write $h(v_i)$ to represent the compound resource required of a task T_i operating at setpoint v_i.

An operational dimension V_{ij} of task T_i is said to be *monotonic* if for all points in the environment space an increase in the operational index value results in increasing utility and non-decreasing resource requirements across all resource dimensions. All other operational dimensions are said to be *non-monotonic*. Monotonic operational dimensions typically include direct QoS-like dimensions such as video frame-rate, as well as indirect QoS-like dimensions such as the dwell period in radar tracking.

2.1 QoS Optimization

We can now define the problem of QoS-based resource allocation as follows. For each task T_i in the set $T_1, \ldots, T_n$, assign a setpoint v_i such that the system utility is maximized and no resource utilization exceeds its maximum. Formally, we write this as:

$$\begin{array}{ll} \textbf{maximize:} & u(v_1, ..., v_n) \\ \textbf{subject to:} & \\ & \forall_{1 \le k \le n} \quad \sum_{i=1}^{n} r_{ik} \le r_k^{max} \\ & \forall_{1 \le k \le n, 1 \le i \le n} \quad r_{ik} = g_{ik}(v_i) \end{array}$$

While finding the optimal resource allocation is NP-hard, the Q-RAM algorithm[3][6] is able to find a near optimal solution to this problem with n tasks and l setpoints per task in $O(nl \log nl)$ time. Q-RAM uses a convex majorant

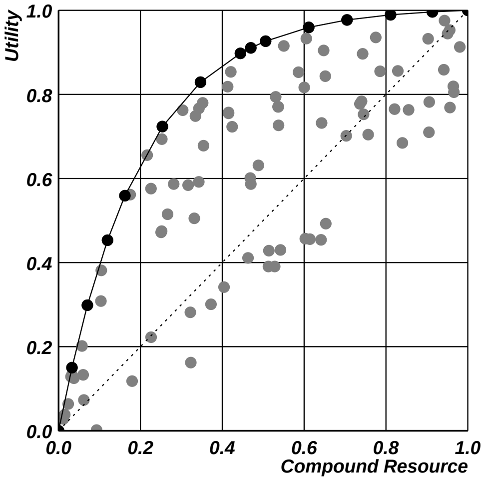


Figure 1. Convex Majorant Example

computation as a pre-processing step to reduce the number of setpoints that must be considered for a task. The convex majorant is computed in the space of compound resource versus utility where compound resource is a value representing the total resource demand of a task. In the context of Q-RAM, the convex majorant is the set of setpoints defining a curve under which all of the other setpoints lay. An example is shown in Figure 1. Setpoints not on the convex majorant need not be considered since they represents operating points for which an increase in utility can be achieved with no increase in resource requirements.

The steps of the basic Q-RAM algorithm[7][4][5][6] can be summarized as:

1. For each setpoint of each task compute the *compound resource* and utility values. The compound resource is measure of the impact on the system of a particular resource requirement vector.

2. Compute the convex majorant of the setpoints in the compound resources versus utility space and eliminate all setpoints not on the convex majorant.

3. Each task is initialized to the lowest utility setpoint.

4. Choose the task with the highest *marginal utility* between its current point and the next point up its convex majorant and make that the new current point for that task. The marginal utility is defined as the increase in utility divided by the increase in compound resource.

5. Repeat the previous step until all resources have been allocated.

The basic algorithm shown here works best when all operational dimensions are monotonic. Extensions to this algorithm for handling non-monotonic dimensions have also been presented in the literature[3]. These heuristics have proven to be very effective at quickly finding resource allocations that are near optimal for tasks with reasonable numbers of setpoints. However, for systems with highly configurable tasks (i.e., tasks with a "large" number of setpoints) the optimization time can increase dramatically due to the time required to do the convex majorant computation. Even though the number of setpoints remaining after the convex majorant step may be small, a direct computation requires that one first determine the compound resource and utility values for all of the setpoints. Since an application with d operational dimensions and p index values per dimension has a total of $l = p^d$ setpoints, the number of setpoints can quickly become unmanageable when there are a large number of operational dimensions.

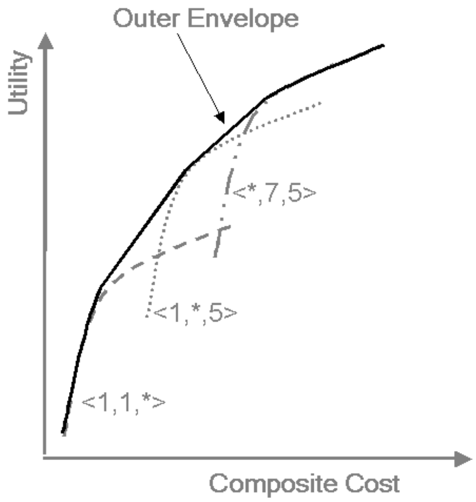


Figure 2. Fast Traversal

3 Convex-Majorant Approximation

The best known algorithm for computing the convex majorant of a set of l points is $O(l \log l)$. Even though this is a relatively benign complexity, when l is large it can still be prohibitively expensive. In this section, several new heuristics for approximating the convex majorant will be presented. These heuristics can be applied without explicitly performing computations on all of the setpoints. For simplicity, we will first assume all tasks have only monotonic operational dimensions. Later we discuss the general case in which some tasks have operational dimensions that are non-monotonic.

3.1 Slope-based Traversal (ST)

Let the *minimum* setpoint for a task T_i for which all operational dimensions are monotonic be defined as $v_i^{\min} = \{1, \ldots, 1\}$, and let the *maximum* setpoint be defined as $v_i^{\max} = \{v_{i1}^{\max}, \ldots, v_{iN_i^V}^{\max}\}$. Clearly all of the setpoints in the utility/compound resource space that lie below a "terminating" line from $(u(v_i^{\min}), h(v_i^{\min}))$ to $(u(v_i^{\max}), h(v_i^{\max}))$ as shown in Figure 1 can not be on the convex majorant. These points can be eliminated immediately without being passed on to the convex majorant step. We call this heuristic "slope-based traversal" (ST). While this heuristic can reduce the time to compute the convex majorant by a constant factor, it must still scan all of the setpoints to determine if they are above or below the terminating line.

3.2 Fast Traversal Methods

We now consider a set of fast traversal heuristics that do not require computations on all of the setpoints. Again, for now we will assume that all operational dimensions are monotonic. The key observation is that when observing the actual convex majorant generated using all of the setpoints for typical tasks, the convex majorant tends to be comprised of runs of setpoints which vary in only one dimension at a time with occasional jumps between runs of points. This suggests we can use local search to follow the setpoints up the convex majorant. For all of these methods we know that $v_i^{\min}$ will always be the first point on the convex majorant and $v_i^{\max}$ will always be the last. The different methods presented here differ primarily in the method used to perform the local search.

As an example, consider a task with three operational dimensions. If we consider the subset of the setpoints $< 1, 1, * >$ consiting of all the setpoints for which dimensions 1 and 2 have index value 1, these points will tend to form a line as shown in Figure 2. The convex majorant will tend to follow such a line until it switches to some other line, in this case $< 1, *, 5 >$ followed by $< *, 7, 5 >$.

While the fast traversal heuristics presented in this section are not guaranteed to find the exact convex majorant, in Section 5 we will show that these heuristics produce good approximations to the convex majorant and more importantly that the drop in system utility from using the approximations is negligible.

3.2.1 First-Order Fast Traversal (FOFT)

In *first-order forward traversal* (FOFT) we keep a current point v_i for each task T_i which we initialize to $v_i^{\min}$. We then compute the marginal utility for all the setpoints adjacent to v_i. A setpoint is adjacent if all of its index values except for one are identical, and the one that differs varies by only one (i.e., they have a Manhattan distance of one). We, in fact, need only consider positive index value changes. We then choose the point that has the highest marginal utility, add it to the convex majorant and make that point the current point. Formally, if v_i is the current point we choose the next current point $v_i' = v_i + \xi_j$ where j maximizes the marginal utility:

$$\frac{u(v_i + \xi_j) - u(v_i)}{h(v_i + \xi_j) - h(v_i)} \tag{6}$$

and where ξ_j is a vector that is zero everywhere except in dimension j where it is equal to 1. We repeat this step until we reach $v_i^{\max}$. After we have generated this set of points, the resulting curve may not be a convex majorant so we perform a final convex majorant operation.

The number of setpoints generated before the final convex majorant step will be the Manhattan distance between $v_i^{\min}$ and $v_i^{\max}$ which is: $\sum_{j=1}^{N_i^V}(v_{ij}^{\max} - v_{ij}^{\min})$. Ignoring boundary conditions, at each point we consider N_i^V possible next setpoints. This means that when we have d dimensions and k index levels per dimensions then the complexity of this algorithm is $O(kd^2)$.

3.2.2 Higher Order Fast Traversal Methods

We can generalize the FOFT algorithm to an m-step p-order Fast Traversal algorithm as follows. Just as in the FOFT heuristic, initialize the current point v_i to $v_i^{\min}$. Then choose the next point $v_i' = v_i + z$ where $z \in G_m^p$ such that the marginal utility is maximized and G_m^p is defined as:

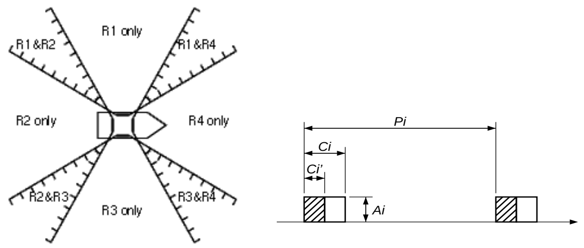
$$G_m^1 = \bigcup_{1 \le j \le N^V, 1 \le k \le m} \{k\xi_j\} \tag{7}$$

$$G_m^p = \{x + y : x \in G_m^1, y \in G_m^{p-1}, x \bullet y \equiv 0\} \cup G_m^{p-1} \tag{8}$$

That is, we look at the all next setpoints that can be reached from the current setpoint by increasing up to p dimensions up to m steps. For the FOFT algorithm described above we would use G_1^1. As with FOFT, we need to perform a final convex majorant operation on the raw points generated by this heuristic. In this paper, in addition to FOFT we will only consider 2-Step First-Order Fast Traversal (G_2^1) which we will call 2-FOFT, and 1-step, Second-Order Fast Traversal (G_1^2) which we will call SOFT.

3.3 Non-Monotonic Dimensions

The fast traversal algorithms described above assume that all of the operational dimensions are monotonic. Unlike monotonic dimensions, non-monotonic operational dimensions generally do not have structure that can be



(a) Radar Layout (b) Dwell Parameters

Figure 3. Radar Model

easily exploited. For this reason, if some of the operational dimensions are non-monotonic, we simply apply the fast traversal algorithms above to the subset that is monotonic for all combinations of index values of the non-monotonic dimensions. We then form the union of all these results and apply a convex majorant. In the worst case that a task has only non-monotonic dimensions this simply reduces to a full convex majorant operation. However, most tasks of interest tend to have more monotonic dimensions and furthermore tasks in which most or all of the dimensions are non-monotonic are not well suited to Q-RAM optimization anyway.

4 Radar Tracking Problem

To validate our Fast Traversal heuristics, we will use radar tracking in a phased array radar system as an example. Unlike a traditional fixed antenna radar, a phased array radar has an electronically controllable beam. This allows different targets to be tracked at different update rates and with different amounts of radar energy depending on their importance. In an actual radar, there are different types tasks including search tasks for acquiring targets and confirmation tasks to determine if some radar return really is a target and if so to determine its type. Other radar problems can include track merging when we determine that two tracks are actually the same target and track resolution when we must decide between multiple candidate tracks for a target. Since this paper is primarily concerned with the resource management issues, we will ignore most of this complexity and concentrate primarily on tracking tasks of known targets.

We assume a ship-board radar system comprised of 4 radar antennas oriented at $90°$ to each other as shown in Figure 3(a). We also assume that each antenna is capable of tracking targets over a $120°$ arc. This means that there are regions of the sky that are capable of being tracked by only one radar antenna, as well as regions that can be tracked by two antennas. The antennas are assumed to share a pool of 128 processors used for tracking and signal processing algorithms and a common power source to supply energy to the antennas.

A single instance of sending a radar signal and receiving the echo for a particular tracking task T_i is called a *dwell* (Figure 3(b)). A dwell consists of multiple short pulses transmitted and received over a period. The duration of this dwell is called the **dwell time** (C_i). The time between two successive dwells is called the **dwell period** (P_i). For simplicity, we model all short transmitting pulses in the form of a single long pulse sent over a **transmit time** (C_i'). Data from the radar return is then passed to a tracking algorithm (Π_i) that processes the data and computes beam steering commands for the next radar dwell.

In our model of the radar tracking problem we consider the operational dimensions shown in Table 1. The dimension X_i specifies the radar antenna to which the track is assigned. Note that due to the orientation of the radars, each target can be tracked by only one or at most two of the radars depending in its position. The tracking algorithm Π_i is one of three commonly used tracking algorithms. The period P_i and dwell time C_i are as described

Dimension	Values
Radar (X_i)	1,2,3,4
Algorithm (Π_i)	Kalman, Least Sqr., $\alpha\beta\gamma$
Period (P_i)	120ms, 180ms, 240ms, ..., 720ms
Dwell Time (C_i)	0.6ms, 1.2ms, 1.8ms, ..., 30ms
Power (A_i)	$0.1d^4$, $0.2d^4$, $0.3d^4$, $\ldots$, $a_m d^4$

Table 1. Operational Dimensions

above, and the power A_i is the transmit power in Watts. Note that since required power increases as the fourth power of distance, we specify the actual power levels as a function of the distance d (in miles) to the target. We also use an environmentally dependent maximum power coefficient of a_m. The value of a_m is chosen at the point where increasing power has little additional benefit for a particular target. Typically it is no larger than about 1.5. This gives us up to $2 \times 3 \times 11 \times 50 \times 15 = 49,500$ setpoints for targets that can be tracked by two radars and 24,750 setpoints for targets that can be tracked by only one radar.

The dimensions P_i, C_i and A_i are monotonic operational dimensions in that increases in the value on these dimensions result in increased tracking quality and increased resource demand. The radar selection dimension X_i is clearly a non-monotonic dimension since it is merely choosing from among a set of resources. While selection of a tracking algorithm (Π_i) affects tracking quality, the best algorithm to use also depends on environmental properties of the targets. For this reason we consider this to be a non-monotonic dimension.

None of the dimensions from Table 1 are of direct relevance to the user in assessing the tracking quality. For this reason we model a tracking error QoS dimension to assess tracking quality. The tracking error is a function of the operational dimensions as well as a number of environmental dimensions. We consider distance, acceleration, noise (e.g., due to electronic counter-measures), speed and bearing from the ship as environmental properties of the targets. Details of the track error computation are given in Appendix A.

The radar problem resource space consists of the following resource dimensions: radar bandwidth, radar power and computing resources. Radar resources consumption is modeled by a utilization value U_k for each radar k given by:

$$U_k = \sum_{i \in \mathcal{R}_k} \frac{C_i}{P_i} \tag{9}$$

where $\mathcal{R}_k$ is the set of tracks that are mapped to radar k. We assume that for each radar there is a utilization bound $U_k^{\max}$ such that U_k can not exceed this bound. While $U_k^{\max}$ can not exceed 1 (corresponding to 100% utilization) in some cases we may need a value less than 1 to model limitations of the dwell scheduler. For the purposes of this paper we will assume that $U_k^{\max} = 1$.

In addition to utilization, radars typically have limitations on how much power can be dissipated as well. Too much power dissipation risks damaging the radar antenna. There are typically both short-term and long-term constraints, but for simplicity in this paper we will only model the long-term constraints. We define the average power for a radar k as:

$$\mathcal{P}_k = \sum_{i \in \mathcal{R}_j} \frac{C_i' A_i}{P_i} \tag{10}$$

This power must be less than some specified bound $\mathcal{P}_k^{\max}$ for each radar.

In addition to the radar resource, each tracking task requires computing resources to process the radar data, and predict the next location of the target. The computing resources required depend on the tracking algorithm Π_i used, and the period P_i. We assume that the required CPU is of the form C_{Π_i}/P_i where C_{Π_i} is the coefficient representing the computational cost of algorithm Π_i. For simplicity we treat the bank of 128 processors as a single processing resource (QoS optimization with processor mapping is addressed in [3]). We then define the CPU utilization as:

$$U_C = \sum_i C_{\Pi_i} / P_i \tag{11}$$

and define $U_C^{\max}$ as the resource bound for CPU.

5 Experimental Results

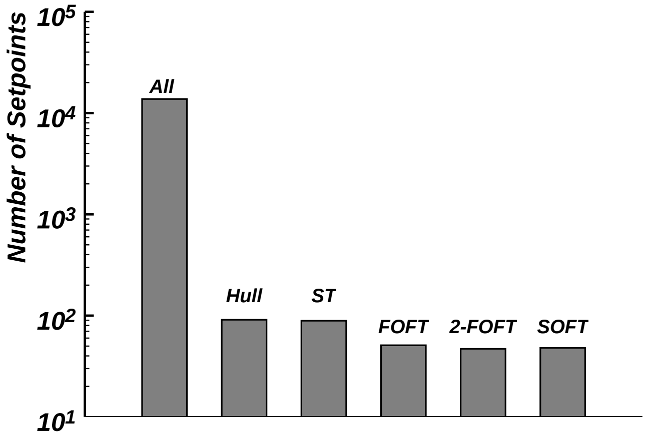


Figure 4. Average Number of Set-points

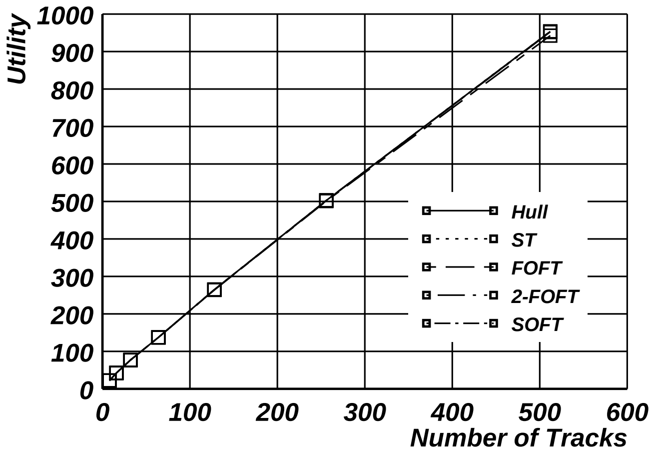


Figure 5. Utility Variation with Basic-QRAM

In order to validate the fast traversal methods, a series of experiments were performed using randomly generated radar tracking scenarios. A 2.0GHz Pentium IV with 256MB of memory was used for the experiments. The number of tracking tasks was varied between 8 and 512 with results averaged over fifty runs for each scenario size. The track parameters were generated randomly from the environmental dimensions shown in Table 2 with the mapping to utility values being performed as described in Appendix A.

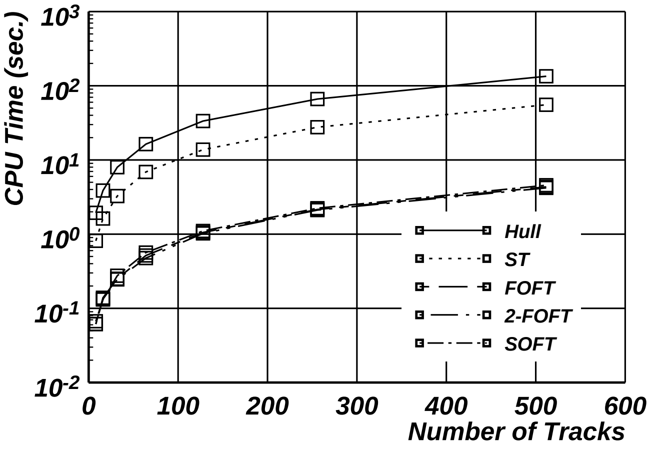


Figure 6. Q-RAM Execution Time

For the purpose of these experiments, the fast traversal algorithms were implemented as a pre-processing step to the Q-RAM algorithm. We used a compound resource expression:

$$r^* = \sqrt{(r_R/r_R^{\max})^2 + (r_P/r_P^{\max})^2 + (r_C/r_C^{\max})^2}$$

to evaluate the resource cost of each setpoint for each task. r_R, r_P and r_C are the total amount of radar bandwidth, power and computational resource required by each task irrespective of the particular radar and cpu to which it was mapped. $r_R^{\max}$, $r_P^{\max}$ and $r_C^{\max}$ are the total resource capacities for radar, power and cpu.

The bar-graph in Figure 4 shows the average number of setpoints per task averaged for all of the generated tasks both before the convex majorant generation (labeled "All"), after the convex majorant generation (labeled "Majorant"), and after each of the of the convex majorant generation heuristics. The tasks had an average of 13,794 setpoints which was reduced to 91 setpoints after the convex majorant operation. As expected, the ST algorithm did not significantly affect the number of setpoints. Note that the heuristics FOFT, 2-FOFT and SOFT produced approximate convex majorant curves that have slightly fewer points than the actual convex majorant.

While the fact that the heuristic convex majorants have fewer setpoints than the actual convex majorant imply that some desirable setpoints have been eliminated, experiments show that the actual impact on system utility is negligible. Figure 5 shows the system utility as a function of the number of tasks. Each point on the curves is the average over the fifty runs. As can be seen all of the performance curves using the heuristic convex majorants with Q-RAM are nearly coincident with the performance curve generated using the full convex majorant. This shows that even though the convex majorant curves generated by the heuristics are not exact, the effect on optimization quality are very small.

Figure 6 shows a plot of the overall execution time including convex majorant generation time and solution generation time for Q-RAM using each of the convex majorant techniques. The top curve is for a convex majorant generated using a full convex majorant algorithm, and the doted line underneath is for the slope-based traversal technique. The slope-based traversal technique results in approximately a factor of two speedup over the full convex majorant algorithm. The execution time curves for Q-RAM using the fast traversal heuristics are nearly identical to each other and much less for Q-RAM using the full convex majorant or the slope-based traversal method. For example, with 512 tracking tasks, the execution time using the full convex majorant algorithm is 134 seconds, but is only 4.3 seconds using the FOFT heuristic.

6 Conclusion

In this paper we have presented a set of heuristics for rapidly generating an approximation to the convex majorant of a set of task operating points by exploiting structure in the tasks' QoS space. Of the three fast traversal heuristics FOFT, 2-FOFT and SOFT, it was found that all performed nearly identically and required nearly an identical amount of cpu time to execute. It was shown that for a set of tasks with approximately 14,000 different configurations, the time required to find a near-optimal configuration for all of the tasks was reduced by a factor of over 30. It was also shown that the quality of the solution was negligibly affected by the use of the heuristics. It is anticipated that this performance gap would be even larger when presented tasks with even larger numbers of possible configurations.

A Tracking Error Computation

Parameter	Value(s)
Type	helicopter, jet, missile
Distance	0.1-10 mi.
Acceleration	$0.001g$-$5g$
Noise	1-8 levels
Speed	60-160 kts. (helicopter) 160-960 kts. (jet) 800-3200 kts. (missile)
Bearing	0°-360°

Table 2. Environmental Dimensions

Constant Name	Values
Speed (K_s)	1.25×10^{-6}
Distance (K_d)	2×10^{-6}
Acceleration (K_a)	Kalman : 5×10^{-4} Least Squares : 5×10^{-5} $\alpha\beta\gamma : 1 \times 10^{6}$
Noise (K_n)	Kalman : 5×10^{-4} Least Squares : 5×10^{-5} $\alpha\beta\gamma : 1 \times 10^{-6}$
Power (K_p)	7×10^{-2}
Dwell (K_c)	3×10^{-2}
Intercept (K_r)	5
Type (K_t)	(helicopter) 1.2 (jet) 1.4 (missile) 1.6

Table 3. Constants in Error Equation

Since we are primarily focused on evaluating the resource management algorithm, we assume a simple model of tracking error. We make no effort to create a precise model of actual radar tracking error but merely to capture the key features to the extent necessary to evaluate the fast traversal heuristics. We make the following assumptions about tracking error:

- Increases in target speed (v_i), target range (r_i), target acceleration (a_i) or generated target noise (n_i) result in increased tracking difficulty.
- The power (A_i) required to obtain similar tracking quality increases as the 4^{th} power of the range (r_i).
- Tracking can be improved by using shorter dwell periods (P_i) and/or longer dwell times (C_i). More distant target will require longer dwell times due to speed-of-light limitations.
- The *Kalman* algorithm works best for noisy but non-maneuvering (low acceleration) targets. The $\alpha\beta\gamma$ algorithm works best for maneuvering (high acceleration) but non-noisy targets. The *Least Squares* algorithm is a compromise with moderate performance against maneuvering and noise.

Based on the above assumptions, we assume the following error estimation equation:

$$\epsilon_i = P_i \Bigg(K_s v_i + K_d r_i + K_a a_i P_i + K_n n_i (\frac{1 + K_p r_i^4}{A_i}) + \frac{1 + K_c r_i}{C_i} \Bigg) \tag{12}$$

Where the coefficients are those shown in Table 3. Targets generated for the experiments have environmental dimension values generated at random from the values shown in Table 2. Three types of targets are generated: helicopters, fighter-jets, and missiles. We use a function of the form:

$$U(\epsilon_i) = w_i(1 - e^{-\gamma/\epsilon_i}) \tag{13}$$

where γ is a constant to map the tracking error into a utility value and w_i is a weight function of the form:

$$w_i = K_t(\frac{v_i}{r_i + K_r}). \tag{14}$$

providing an estimate of the importance of a particular target. The K_r term represents the importance based on the target type, and the right-most term represents the time-to-intercept (i.e., the time that would be required for a target to reach the ship if flying directly toward it).

References

[1] R. Baugh. *Computer Control of Modern Radars.* RCA M&SR-Moorestown Library,1973.

[2] D.R. Billetter. *Multifunction Array Radar.* Artech House, 1989.

[3] S. Ghosh, R. Rajkumar, J. Hansen, J. Lehoczky. *Scalable Resource Allocation for Multi-Processor QoS Optimization.* International Conference on Distributed Computing Systems, May 2003.

[4] 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 1998.

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

[6] 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.

[7] 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.

[8] Jeffery P. Hansen, John Lehoczky and Ragunathan Rajkumar Optimization of Quality of Service in Dynamic Systems In *Proceedings of the 9th International Workshop on Parallel and Distributed Real-Time Systems* April 2001.