Leveraging Resource Prediction for Anticipatory Dynamic Configuration

Vahe Poladian, David Garlan, Mary Shaw, Bradley Schmerl, João Sousa[†]
School of Computer Science, Carnegie Mellon University
Information and Software Engineering, George Mason University[†]
{vahe.poladian, garlan, mary.shaw, schmerl}@ cs.cmu.edu, jpsousa@gmu.edu

Abstract

Self-adapting systems based on multiple concurrent applications must decide how to allocate scarce resources to applications and how to set the quality parameters of each application to best satisfy the user. Past work has made those decisions with analytic models that used current resource availability information: they react to recent changes in resource availability as they occur, rather than anticipating future availability. These reactive techniques may model each local decision optimally, but the accumulation of decisions over time nearly always becomes less than optimal.

In this paper, we propose an approach to self-adaptation, called anticipatory configuration that leverages predictions of future resource availability to improve utility for the user over the duration of the task. The approach solves the following technical challenges: (1) how to express resource availability prediction, (2) how to combine prediction from multiple sources, and (3) how to leverage predictions continuously while improving utility to the user. Our experiments show that when certain adaptation operations are costly, anticipatory configuration provides better utility to the user than reactive configuration, while being comparable in resource demand.

1. INTRODUCTION

Recent self-adaptive systems improve quality of service despite resource shortage by using models of user preferences, historical profiles of application resource intensity, and estimates of current resource availability. Such systems partially automate various system decisions, such as which suite of applications to select for a task and how to allocate scarce resources among concurrent applications with the objective of best satisfying the individual preferences of the user. These systems may also guide the adaptation of resource-aware applications, when more than one dimension of quality is of concern to the user, perhaps using a set of preference functions explicitly specified by the user for a given task and context.

A common shortcoming in the behavior over time of such self-adaptive systems arises from their purely reactive adaptation policies. When dealing with changes in the operating environment, e.g., changes in resource availability, these systems make decisions based only on recent data, often resulting in sub-optimal decisions over time. For example,

- Aura ([4][12][15]) achieves dynamic behavior by performing re-configurations, which are costly in terms of both resource usage and user disruption. Multiple costly re-configurations in response to several changes add up to a globally suboptimal utility to the user over time.
- Q-RAM [8] admits tasks and allocates resources among them based on their utility to the system and resource demand intensity. Running tasks have a priority over new ones. If available resource levels are not sufficient, the system will not admit new tasks, even though there might be a resource allocation using both running and new tasks that improves utility.

As these examples demonstrate, making decisions without considering future changes is myopic and prone to be suboptimal over long term.

Recent results in resource prediction offer an alternative to reactive adaptation. For example, [13] and [14] have analyzed a significant number of traces and have concluded that in many cases, network traffic has good predictability. Using relatively inexpensive linear time-series models, predictions of network traffic can be done in near real time for a meaningful future horizon, e.g., a few dozen seconds, or even minutes. Other sources of predictive information are also available, e.g., administrator-announced network, CPU and service outages, recurring patterns of resource usage that have weekly or daily periods, remaining battery level, and models of battery drainage.

In this paper, we propose an anticipatory approach to self-adaptation, which combines the benefits of resource prediction research into an existing framework of dynamic configuration. Specifically, we present an enhanced analytical model of configuration that builds upon existing reactive models of resource allocation and takes advantage of resource predictions from multiple sources. We also present a set of resource allocation algorithms that leverage predictive information to the benefit of the user. We demonstrate that these new algorithms improve upon the earlier reactive algorithms while remaining feasible for real time evaluation.

In this work three main results address key challenges in engineering a system for anticipatory configuration. First, we define a mathematical notation for expressing uncertainty in predictors that is consistent with resource prediction literature. Next, we define a calculus for combining multiple predictors to provide aggregate predictions based on multiple sources and types of predictive information. Third, we present an efficient on-line algorithm of anticipatory configuration that leverages predictive information.

The rest of the paper is structured as follows. Section 2 surveys related work and highlights the novelty of this work. Section 3 defines important terms, enumerates the requirements for anticipatory configuration, and presents our approach, including a notation for predictors and combining calculus. In Section 4, we describe the algorithms for anticipatory configuration and analyze their theoretical running times. Section 5 presents the results of runtime experiments comparing several approaches to configuration, including

anticipatory and reactive. The evaluation of our approach and enumeration of software engineering benefits in Section 6. We summarize the conclusions in Section 7.

2. RELATED WORK

Self-adaptive systems such as Q-RAM [8], Aura [12][15], and Nemesis [10] incorporate models of user preferences, application behavior, and resource availability in order to optimize some measure of user satisfaction. Q-RAM is a general framework for quality of service and resource management, while Aura is a configuration (self-adaptation) infrastructure for ubiquitous computing. Both of these systems implement a centralized resource arbiter that makes resource allocation decisions. Nemesis is an experimental operating system that has a centralized resource accounting component, but uses a decentralized congestion pricing based approach to optimize the allocation of resources among competing concurrent applications.

Unlike the previous three systems that focus on policy of adaptation, Odyssey [11] and Puppeteer [7], among others, primarily focus on adaptation mechanisms. They both implement application and operating system mechanisms for multi-fidelity, resource-aware execution. Odyssey emphasizes *agile* mechanisms for handling both transient and persistent surges and drops in resources,

All of the above systems are purely reactive in adaptation policies and mechanisms; *none* of them considers predictions of future resource availability in decision making.

NWS [16] and RPS [1] are tools for gathering and analyzing available resource information. [3] presents a comprehensive overview of linear time series models used in prediction. Using tools such as NWS and RPS, studies in [13][14][17] have demonstrated that: (1) resources have good predictability and (2) when resources are predictable, relatively inexpensive linear models with autoregressive (AR) components work as well as other more complex predictors. It is conjectured, e.g. in [13], that resource prediction can be done online, using software running on routers.

Anticipatory configuration is similar to online stochastic combinatorial optimization (OSCU) problems such as package routing and vehicle dispatch ([2][6]). While the problem domains are different, each dynamic configuration algorithm has an analog in OSCU. The Reactive, Perfect, and Expectation algorithms in dynamic configuration are respectively called Local, Offline, and Expectation in OSCU.

3. APPROACH

We now introduce the problem of anticipatory dynamic configuration, discuss the specific technical challenges that we addressed in this work, and describe our approach. Because anticipatory configuration builds on an earlier model of reactive configuration, we briefly review the pertinent details of that model first.

3.1 Terminology

Following [15], we define the set of computational devices, applications, and resources available to a user in a location as the *environment*. Applications and devices provide *services*, which are abstract descriptions of the application capabilities, identified by service type, e.g. “play video”, “edit text”, “browse web”. A specific application on a specific device is called a *supplier*. Users carry out *tasks* to work on their everyday projects, e.g., plan a vacation, produce a report, or review a video clip. A task specifies the use of one or more simultaneous services for the *duration* of the *activation* of the task. Each task might be activated several times, possibly in different locations.

Applications use computational *resources* (such as CPU cycles, network bandwidth, disc, memory, and battery energy) to provide service to the user. In many environments resources are scarce and can change over time. Some applications are resource and fidelity aware, able to provide lower level of service in one or more *quality of service* (QoS) *dimensions* while consuming fewer resources. Lower quality service allows the user to make progress on his task, although his satisfaction from the task might be lower.

A suite of applications that can satisfy a task is called a *supplier assignment*. There might be multiple *candidate* assignments for a task in an environment, because each service in the task might be satisfied by alternative available applications. A resource *allocation* is a set of resource vectors, one per supplier in an assignment. Each of these vectors specifies the maximum amount of a resource that the application should consume. A QoS *set-point* is a vector of QoS levels that the application should meet. A *configuration* is a triple of supplier assignment, resource allocation and QoS set-points.

The problem of configuration is to find a configuration that maximizes user’s *utility*. Utility depends on the suppliers in the assignment as well as QoS set-points.

In the earlier model of configuration [12], utility is an instantaneous measure of a user’s satisfaction. That model works *reactively*, by considering only snapshots of current resource availability in the resource allocation and configuration selection. As resource availability changes, the reactive model performs reconfiguration, changing the previous configuration if there is gain in instantaneous utility. Thus the solution in the reactive model maximizes instantaneous utility in a series of locally optimal decisions.

In contrast, an *anticipatory* model of configuration considers future resource availability predictions, and chooses *sequences* of configurations over the duration of the task, and maximizes the expected value of utility *accrued* over the duration of the task.

3.2 Challenges

The principal goal in this work is to improve the quality of service to the user in an existing framework of dynamic configuration by leveraging resource predictions. To do so, we must address the following three requirements:

R1. *Define a measure of accrued utility that captures the temporal dimension of anticipatory configuration.*

To make globally optimal decisions, the utility function of the user needs to be enhanced. The enhanced notion of utility should: (1) incorporate the temporal dimension of the

anticipatory configuration and guide globally optimal decision-making, (2) represent a user's satisfaction with service quality over a period of time, while capturing the relevant attributes of a task, as before, and (3) allow for comparison of anticipatory and reactive configuration models.

R2. *Express and combine predictive information about future resource availability from multiple predictors.*

One part of the challenge here is to express predictive information in a way that is consistent with existing prediction literature. Second part of the challenge is to aggregate predictions from multiple sources.

R3. *Design efficient on-line algorithms for anticipatory configuration that improve expected utility for the user.*

The new algorithms for anticipatory configuration must make online decisions under uncertainty. Such algorithms must balance the runtime resource overhead and latency with optimal decision making. These algorithms should demonstrate improvement over those in the reactive model under reasonable assumptions of predictor accuracy.

3.3 Utility

3.3.1 Utility in the Reactive Model

Utility is a measure of user satisfaction with respect to the running state of the systems. In the model of reactive configuration, the system is concerned with *instantaneous utility* (IU), which has three parts: affinity for applications, preference for quality of service, and penalty for switching. The first part in the instantaneous utility allows the user to express his preference for specific applications. For example, the user might specify that among video players he strongly prefers Windows Media Player, but might also be happy with QuickTime or RealOne Player by giving scores to each of these choices. Furthermore, he might also accept any other video player, but score them below either QuickTime or RealOne.

The second part in the utility is collection of preference functions and weights that allow the user to express a desired level of service in each QoS dimension as well as trade-offs among different dimensions. Using a preference function for each QoS dimension, the user specifies how much he values improvement or deterioration of service along that dimension. Using a scalar weight, the user specifies how important that dimension is relative to others.

The third part in the utility allows the user to specify penalties for disruptive changes. This is to discourage the system from switching currently running applications, unless the gain in utility is sufficiently large. For each service in the task, switching of applications is penalized by a scalar amount.

The instantaneous utility is combination of the three parts. Appendix A has the formal expression of IU.

3.3.2 Utility in the Anticipatory Model

In the model of anticipatory configuration the objective of the system is to maximize the *accrued* utility. We use a discrete time model by dividing the duration of the task into T equal windows, and index each using variable t, $0 \le t < T$. Let *Seq* denote a sequence of *T-1* configurations, one per each window in the duration of the task: $Seq = \{Seq_0, Seq_1,\ldots,Seq_{T-1}\}$, where each Seq_s is a configuration chosen to run during period s. The accrued utility (AU) of the sequence *Seq* is defined as:

$$AU(Seq) = IU(Seq_0, \phi) + \sum_{s=1}^{T-1} IU(Seq_s, Seq_{s-1})$$

where in the expression of the instantaneous utility we include both the current and previous configuration. In other words, the accrued utility over a time period is the sum of instantaneous utilities during that time period.

3.4 Supplier (Application) Profiles

Applications use resources to provide service. Typically, providing a better level of service requires the use of more resources. Using historical profiling [9], it is possible to find an application's resource requirement for each level of quality of service. An *application profile* is an enumeration of resource and QoS vector pairs, where the resource vector is the required level of resources for providing the level of service specified by the QoS vector.

In the anticipatory configuration model we continue to assume that application profiles are static, i.e. they are computed using offline profiling, don't change over time, and are sufficiently accurate.

3.5 Resource Availability Predictions

3.5.1 Resource Availability in the Reactive Model

In the reactive models of configuration, only the current level of resource availability is modeled.

The anticipatory model explicitly incorporates predictions of future resource availability. Next we discuss the details of resource prediction.

3.5.2 Resource Prediction

Ideally, a prediction for the available level of a resource is a probability density function for a future time of prediction s and the current time, t. For each possible level r of the resource, the function predicts the probability that the resource will be at that level at time t. Thus, the available level of resource R at time s is a random variable, R_s. To capture the fact that the prediction is made at time t, we use the following notation: $R_{s|t}$, which is the conditioning of the random variable R_s based on the information available at time t.

A *generalized predictor* for resource R at time t, $0 \le t < T$, is a set of probability density functions, one for each s, $s \ge t$, of the random variable $R_{s|t}$.

In practice, a predictor might not provide the complete distribution of the resource for all future times s. For example, a prediction from one source might be that with 100 percent probability, the available resource level can not exceed a certain threshold. Another source might predict a surge or drop in the resources around a specific time.

3.5.3 Types of predictors

We define three types of resource predictors: linear recent history, relative move, and bounding predictors.

A *linear recent history predictor* is any predictor that uses recent history and a linear time-series model to predict

the next value in the series of resource availability. This predictor is motivated by existing literature [3]. We consider autoregressive (AR) models of low orders. Moving Average (MA) and auto-regressive moving average (ARMA) models can be easily handled in a similar manner by the anticipatory configuration algorithms.

Formally, an autoregressive linear recent history predictor of order p for resource R is an equation of the form:

$$R_{t+1|t} = \varphi_1 r_t + \varphi_2 r_{t-1} + \ldots + \varphi_p r_{t-p+1} + Z_{t+1},$$

where r_i are the previous p observations of the resource (the small letters indicate that these numbers are not random), φ_i are parameters of the model and are known at prediction time, and Z_{t+1} is a normal random variable with mean 0 and variance σ, $Z_{t+1} \sim N(0, \sigma)$.

Notice that the prediction we have is only one step ahead. However, we can easily express $R_{t+2|t}$ using $R_{t+1|t}$, the previous $p-1$ observations, and Z_{t+2}, an additional normal random variable which is independent of Z_{t+1}.

There might be opportunities for prediction that are not captured by a linear predictor. For example, by observing resource demand changes (surges and drops) and correlating these with calendar information, it might be possible to predict such changes and their length in the future.

The second, *relative move predictor* predicts step-up or step-down changes in resource availability. Formally, a relative move predictor is a set of tuples $\langle s, M \rangle$, where s is the time of prediction and M is the possibly random magnitude of the predicted move. For the purposes of this work, we will assume that M is normally distributed, $M \sim N(\mu,\sigma)$. Formally, if *rm* is a relative move predictor, then $rm = \{\langle s,M \rangle\}$.

The third, *bounding* predictor specifies the maximum and minimum possible level of resource availability for a union of time intervals. A bounding predictor is motivated by the availability of various sources of information, such as the maximum bandwidth specification of a DSL line, signal strength and type of WiFi network. In case of CPU, the maximum available level is available from hardware specification and from the power saving settings.

3.5.4 Predictor Calculus

We now define a calculus for combining multiple predictors into an aggregate prediction. Let L denote the set of all linear predictors, RM denote the set of relative move predictors, B denote the set of bounding predictors. We define operations on predictors as follows.

Boosting of two linear predictors: if $l1$ and $l2$ are predictors in L, then $l3 = l1 \times l2$ is a linear predictor. The term *boosting* refers to the machine learning technique that allows improved prediction or classification by combining multiple predictors or classifiers. Simple averaging is boosting, although a good booster should reduce the prediction error by finding correlations among the predictors.

Concatenation of two relative move predictors: if $rm1$ and $rm2$ are predictors in RM, then $rm3 = rm1 \cdot rm2$ is also a relative move predictor. If $rm1$ and $rm2$ have conflicting predictions, i.e. one of the predictions in $rm1$ is for the same time period as another prediction in $rm2$, we can simply combine those two predictions by adding the random moves. Otherwise, the predictions are combined by taking the union of the two sets of predictions.

Addition of a relative move predictor to a linear predictor: if $rm1$ is in RM and $l1$ is in L, then $gp1 = l1 + rm1$ is a generalized predictor that combines the relative moves predicted by $rm1$ into the linear prediction of $l1$.

Bounding of a generalized predictor by a bounding predictor: if $gp1$ is a generalized predictor and $b1$ is a bounding predictor in B, then $gp2 = gp1 \,||\, b1$ is a generalized predictor that is the bounding of $gp1$. Bounding limits the support of any probability density function to the interval specified by $b1$.

Intuitively, a linear predictor finds short-term correlations in the recent history of resource availability. A relative move predictor finds periodic patterns of resource increases or decreases that are not reflected in the recent history. The effect of a relative move is in addition to the prediction of a linear predictor (hence justifying the operation of addition). A bounding predictor limits the range of resource availability.

Let's see the predictor calculus in action by the way of a simple example. Suppose $l1$ and $l2$ are linear predictors, $rm1$ and $rm2$ are relative move predictors and $b1$ is a bounding predictor. Then $l1 \times l2 + rm1 \cdot rm2 \,||\, b1 = \{(l1 \times l2) + (rm1 \cdot rm2)\} \,||\, b1$. In other words, we first apply all boosting operations to obtain one linear predictor. Next we apply all the concatenation operations until one relative move predictor remains. Then we perform addition with the only resulting linear predictor and the only resulting relative move predictor. After that, we apply as many bounding operations as there are bounding predictors.

Figure 1 shows resource predictions using 3 curves. The middle curve is the expected value (mean) of the predicted level of the resource. The top curve is one standard deviation above the mean and the bottom curve is one standard deviation below the mean. This predictor was obtained by the addition of a relative move predictor (*rm1*) to a linear predictor (*l1*).

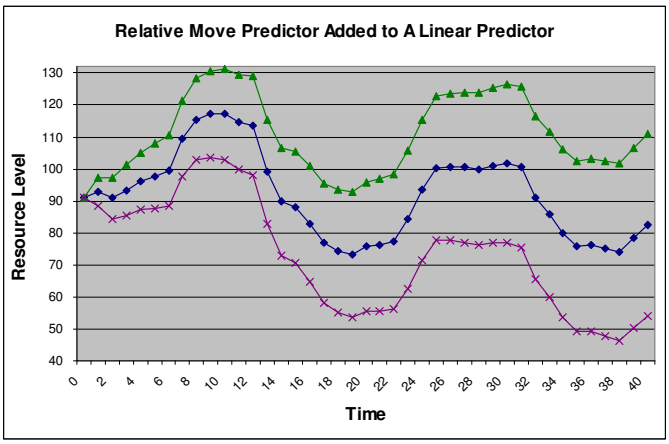

Figure 1: The mean and one standard deviation band of the sum of linear and relative move predictors. The Y axis is in abstract units of resource.

3.6 The Formal Optimization Problem

Informally, the optimization problem at hand is one of choosing a sequence of configurations over the duration of the task, such that the expected value of accrued utility is maximized given the aggregate knowledge of all resource predictors. There are two constraints to the optimization problem: (1) the level of quality of service of each supplier is bound by the supplier's historical QoS vs. resource profile, and (2) the sum of resource demands of the suppliers in the running configuration in each time period can not exceed the actual resource supply.

Let *Set(Seq)* be the set of all possible configuration sequences. Let $QoSProf_{\text{supp}}$ denote the QoS vs. resource profile for the supplier identified by *supp*. Let rr_{sls} denote the actual resource availability vector for each time period s, $0 \le s<T$. Let *Seq* denote a sequence of configurations: $Seq = \{Seq_0, Seq_1,\ldots,Seq_{T-1}\}$, where each Seq_s specifies a supplier assignment. Then the objective of anticipatory configuration is to maximize:

$$\arg\max_{Seq \in Set(Seq)} E\left[IU(Seq_0, \phi) + \sum_{s=1}^{T-1} IU(Seq_s, Seq_{s-1}) \right]$$

given the knowledge of all the resource predictors and subject to the following constraint for each s, $0 \le s<T$:

$$\sum_{Supp \in Seq_S} QoSprof_{Supp}(f_{Supp}) \le \left\| rr_{sls} \right\|$$

The inequality in the above constraint is understood for each resource dimension separately and must hold for *any* combination of quality set-point choices (denoted by of f_{Supp}) among the suppliers of Seq_s.

4. ALGORITHMS AND ANALYSIS

An algorithm for anticipatory configuration should maximize the expected accrued utility over the duration of the task. At each time step, an algorithm decides which assignment of the suppliers to choose as well as how to allocate resources among them. First we describe some prerequisite computation. Next, we describe two different algorithms for anticipatory configuration.

4.1 Prerequisite Algorithms

4.1.1 Resource sieve and resource scenarios

For tractability purposes, we discretize the available levels of resources. For each resource, we enumerate the possible available levels of that resource. Then we consider the Cartesian combinations of the available levels of all the resources. We call this the resource sieve. And we call each point in the sieve a resource scenario.

For example, if the environment includes one laptop, and one of the resources under consideration is the CPU of the laptop, then the possible available levels of the CPU can be anywhere from 0 to 100 percent of the maximum, in 4 percent increments.

Another resource of interest, downstream TCP bandwidth, might have 13 levels: 100, 150, 200, 250, 300, 350, 400, 500, 600, 700, 800, 900, 1000 (all in kbps). These levels are based on the available suppliers and their resource intensity for various levels of quality of service.

With two resources, CPU and bandwidth, as described above, the resource sieve will have 26 * 13 = 338 resource *scenarios*. We generate resource scenarios using an ad-hoc enumeration and call this algorithm GenResourceSieve.

4.1.2 Supplier assignments

For each service in a task, there may be multiple available suppliers in the environment that can satisfy that service. For example, for a "play Video" service, on a typical Windows system there might be as many as 4 applications to choose from: Windows Media Player, RealOne Player, Apple QuickTime, iTunes. The same applies for browsers, e-mail readers, text editors, compilers, etc.

Recall that a supplier assignment is an assignment of one application for each service in the task. A task can have multiple supplier assignments. We generate all possible supplier assignments using ad-hoc enumeration and call this algorithm GenSuppAssignments.

We now present three lemmas that help in computing instantaneous utility and designing the algorithms:

Lemma 1: affinity for applications depends only on the choice of a supplier assignment and does not depend on the level of available resources in the past, present, or future.

Lemma 2: the switching penalty depends only on the supplier assignments in the previous and in the current time period.

Lemma 3: QoS utility depends only on the current assignment of suppliers and current resource availability.

These lemmas follow directly from the definition of instantaneous utility. Armed with these three lemmas, we perform the following computations.

Penalty(SA1, SA2): for each pair of candidate supplier assignments, SA1 and SA2, compute the penalty in the hypothetical situation when switching from SA1 to SA2 by complete enumeration of all pairs and store the result in a two-dimensional array.

SuppPrefScore(SA): for each candidate supplier assignment, SA, compute the portion of instantaneous utility that is due to application preference and store the results in a one-dimensional array.

OptInstQoS(SA, RS): for each supplier assignment, SA, and each resource scenario, RS, compute the optimal resource allocation among the suppliers in SA when the current level of resources is given by RS. Also determine the QoS set-points for each supplier and the resulting utility. The mrmd algorithm in [8] solves the resource allocation problem *all* resource scenarios at once, using dynamic programming. The running time of the algorithm is proportionate to the size of the resource sieve.

4.2 Algorithms For Anticipatory Configuration

The first algorithm, OptAccUtilPerfect, maximizes actual accrued utility in the case when resource predictions are known precisely. The second, OptAccUtilExpectation, ex-

plicitly deals with uncertainty in resource predictions and works by maximizing the expected accrued utility.

4.2.1 Algorithm for No-Uncertainty Prediction

In the simplest case of anticipatory configuration, we assume that the resource predictors are error-free. Under this unrealistic assumption, the predictions are exact resource paths into the future for the duration of the task and as time goes by, this proves to be true. The benefits of considering this case are two-fold: (1) we would like to find out if anticipatory configuration is ever better than reactive configuration and (2) we create a shared routine that can be invoked in hind-sight for comparison purposes.

Prediction in this case is a vector of snapshots: *RS(s)*, where $0 \leq s < T$ and each *RS(s)* is a resource scenario.

We find a sequence of configurations that maximize the accrued utility using dynamic programming. Let's define PartMaxAU(j,s) to be the maximum partial accrued utility possible if the task were to run starting from time period *s* till time period *T* and if supplier assignment with index *j* were chosen to run at time *s*. To demonstrate how the dynamic programming algorithm would work, we show the terminal condition and the recursive rule:

- PartMaxAU(j,T-1) = OptInstQoS(j,RS(T-1) + SuppPrefScore(j), because *T-1* is the last time period.
- PartMaxAU(j,s) = $\max_k$ { OptInstQoS(j, RS(s)) + SuppPrefScore(j) + Penalty(j,k) + PartMaxAU(k,s+1)}.

The first three terms in the last sum are pre-computed and together add up to the instantaneous utility possible when choosing to run supplier assignment *j* in period *s*. The fourth term is the future partial accrued utility from time period *s+1* till the end of the task if supplier assignment with index *k* is chosen next. In addition to recording the maximum utility, we also record the value of *k* for which that maximum is achieved.

The dynamic programming algorithm will start from time period *T-1* to compute PartMaxAU(j,T-1) and work backwards in time in a simple loop to compute PartMaxAU(j,s). The maximum of PartMaxAU(j,0) over all possible supplier assignments *j* will be the maximum possible accrued utility for the task. The sequence of supplier assignments is also recorded. We call this algorithm OptAccUtilPerfect.

Next we analyze the runtime complexity of the algorithm. To help in this analysis, we define the following variables:

- *nServices*, the number of services in the task,
- *nAltSupp*, the typical number of alternative suppliers for each service type,
- *nResources*, the number of resources,
- *nResPoints*, the number of different resource points for a typical resource,
- *nRSieve*, the size of the resource sieve or the number of different resource scenarios. This number is O(*nResPoints*^ *nResources*).
- *nSA*, the number supplier assignments. This number is O(*nAltSupp*^ *nServices*).
- *nQoS*, the number of maximum different possible QoS points among all suppliers,
- *T,* the duration of the task or the number of time periods.

Here are the running times of the prerequisite algorithms:

- GenSuppAssignments is O(*nSA*nServices*),
- GenResourceSieve is O(*nRSieve*),
- Penalty is O(*nSA*nSA*),
- SuppPrefScore is O(*nSA*),
- OpUtil is O(*nQoS*nRSieve*nSA*),
- OptAccUtilPerfect is O(*nSA*nSA*T*).

After adding the running times of the above algorithms, the following two terms dominate all the others:

- O(*nQoS*nRSieve*nSA*) + O(*nSA*nSA*T*).

We conclude that the running time is pseudo-polynomial with respect to the inputs of the problem. We argue that this algorithm is feasible for online computation.

```
<double, int> OptAccUtilFullSearch(int t,int s,
              Vector RPath (0..s),
              Vector Seq(-1..s-1),
              Vector RPred(s+1..T-1) ){
  <RSim[],Prob[]> = SimRScenario(s+1,
             RPath (0..s), RPred(s+1..T-1));
  for each a in SA {
    Seq(1..s+1) = Seq(1..s), SA[a];
    double instUtil = OptInstQoS(a, RPath[s]) +
       SuppPrefScore(a) + Penalty(a,Seq(s-1));
    double expUtilFuture = 0;
    if ( s < (T-1) )  // termination condition
      for each rs in RSim {
        RPath(0..s,s+1) = RPath(0..s),RSim[rs];
        <U,next> = OptAccUtilFullSearch (t,
              s+1, RPath(0..s+1),
              Seq(-1..s), RPred(s+2..T-1)) ;
        expUtilFuture += Prob[rs] * U;
      }
     // find + record max of expUtilFuture
  }
  return <maxExpAccU, bestSA>;
}
AnticipatoryDynConfig()
{
  Obtain RHistory;
  Initialize RPath = RHistory, Seq, RPred;
    for s = 0 to T-1  {
    <util, a> = OptAccUtilFullSearch (s, s,
              RPath(0..s), Seq(-1..s-1),
              RPred(s+1..T-1);
    ExecuteConfig(a, RHistory(0..s), prevSA);
    Update RHistory, RPath;
    UpdatePred(RPred, RPath[s+1]);
    prevSA = a;
  }
}
```

Figure 2: The Full Search Anticipatory Algorithm.

4.2.2 Algorithm for Uncertainty in Prediction

In practice, resource predictors are noisy, and an anticipatory algorithm must handle the noise in the predictors.

Figure 2 shows a generic algorithm that maximizes the expected value of accrued utility when predictors are noisy.

The algorithm, OptAccUtilFullSearch, simulates likely paths of resources and computes the expected future accrued utility along these paths for each candidate supplier assignment. *t* is the current time, *s* is the time of simulation and is continually rolled forward, *RPath* contains actual resource availability history up to time *t* and simulated resource history up to time *s*, *Seq* contains supplier assignments that are

selected to run at each time, *RPred* contains the predictor objects conditional on information up to time *s*.

The algorithm first simulates possible resource states and their probabilities for the next time period. Next it cycles through all candidate supplier assignments and calculates the expected future accrued utility in the hypothetical case when a particular supplier assignment is selected to run in the current time period. The expected future utility is calculated in the inner loop by iterating through the simulated resource states and computing the probability weighted average of future expected utility from each state. The instantaneous utility from running that supplier in the current period is added to the expected future accrued utility.

The AnticipatoryDynConfig routine is the entry point of configuration. This routine is responsible for "rolling" time forward, injecting the system with actual resource availability information, executing (starting / stopping) the necessary applications and setting their runtime state.

The algorithm computes the maximum expected accrued utility as defined in section 3.6. However, because of the liberal use of recursion, the running time of the algorithm is exponential in *T*. Indeed, the algorithm does a complete scan of all possible simulated resource paths, with a branching factor proportionate to *nSA*nRSim*, the latter being the number of simulated resource state at each recursive step. Even for small problem sizes the running time of the algorithm can quickly escalate.

We considered several possibilities to overcome this problem. By memoizing computation results and using dynamic programming, we can reduce the running time, but will need exponential storage space. We can also partially memorize computation results, rearrange the two loops, and reduce the branching factor of the recursion, but this will not reduce the running time below exponential. We also considered Monte Carlo simulation.

We discovered that reducing the depth of the recursive search does not affect the optimality of the algorithm dramatically, but reduces the running time to less than exponential. We modify the Full Search algorithm, by limiting the depth of complete simulations to a configurable parameter. For the remainder of the search, we obtain an approximation of future expected accrued utility using the OptAccUtilPerfect algorithm over the *expected* path of the resources. We call this modified algorithm OptAccUtilExpectation.

5. EXPERIMENTS

We implement the algorithms from Section 4 and the reactive model. We experimentally compare the algorithms along two metrics: (1) the actual accrued utility to the user over the duration of the task and (2) runtime efficiency.

5.1 Experimental Setup

The basis for our experiments is a task of a newspaper movie critic, who watches clips and writes reviews. This task has 2 services: *video playing* and *browsing*. The user simultaneously watches streaming clips using a video player and searches for information using a browser. Both the browser and the server support levels of data compression and the server can provide content at varying levels of fidelity (e.g., text-only, images, multi-media). There are three alternative applications for video playing and three alternative applications for browsing, generating 8 alternative supplier assignments (*nSA*).

5.1.1 Input Data

The model requires three inputs: (1) user preferences, (2) application profiles, and (3) resource availability predictions. We profile 6 applications: 3 video players and 3 browsers. As an experimental platform, we use a relatively old IBM T30 laptop that can operate in power-saving mode and limit the available CPU level to a percentage of the maximum. For user preferences, we use synthetic data.

In this case study we consider 2 resources: CPU and bandwidth. For CPU availability predictions, we use only relative move and bounding predictors, because the user is in complete control of his hardware and there are no external demands on the CPU to require the use of a linear predictor. Variations in CPU availability come from a number of sources: (1) whether the laptop is plugged into an outlet or not, (2) planned background tasks (e.g., virus checker or backup). For bandwidth prediction, it is appropriate to use a linear predictor. We combine an autoregressive predictor of order 5 with several relative move predictors (so a bandwidth prediction very much looks like the graph in Figure 1).

We create a resource sieve of about 350 points (*nRSieve*) and consider a task of duration 25 (*T*). Based on these numbers, OptAccUtilFullSearch is not feasible to run online.

5.2 Algorithm Comparisons

5.2.1 Comparisons in Utility to User

We now address the following questions. Under what conditions is anticipatory configuration better than reactive configuration? How do we quantify the improvement?

These factors influence whether anticipatory configuration can outperform reactive configuration:

- The magnitude of switching (re-configuration) costs relative to other components of instantaneous utility,
- Similarity of the profiles of suppliers.

If there are no switching costs, or those costs are very small, then several locally optimal decisions provably add up to a globally optimal decision. In that case, reactive configuration is temporally globally optimal.

With respect to the second factor, consider the extreme case when all the suppliers providing the same service have identical application profiles, i.e. their resource requirements for each QoS level are identical. Mathematically, the problem reduces to having only one supplier assignment, and there is never a need to switch suppliers to capture better utility when resources change.

We have observed that suppliers offering the same service can have vastly different resource requirements. Applications from large commercial vendors tend to be feature-rich and resource-intensive, while open-source applications tend to be more efficient. Also, different vendors offer different runtime options in their applications, contributing to the diversity of application resource profiles. Thus, even

with static user preferences, the ranking of the supplier assignments changes under different resource scenarios.

Assuming the above factors are satisfied, there are two reasons why reactive configuration might under-perform anticipatory configuration. First is thrashing and is likely to occur when switching penalties are small (but not insignificant). Oscillating resource paths will force reactive configuration to change supplier assignments and pay switching penalty frequently. The second reason is lack of amortization of switching costs over multiple time windows and is likely to occur when penalties are relatively high. Reactive configuration might not find it optimal to switch a configuration when resources change, although doing so will pay in the long term, if the resources persist at that level.

To investigate these questions, we performed experiments of 1000 trials that compared the actual accrued utility achieved by 4 different algorithms under the same resource conditions: OptAccUtilPerfect, OptAccUtilExpectation, Reactive, and Random. Random randomly selects a supplier assignment in the beginning of the task and commits to that assignment throughout the task. As resources change, Random can change the resource allocation to maximize QoS portion of IU. Obviously, Random does not incur penalties since it can't switch the suppliers.

We calculated the average accrued utility achieved over 1000 trials for each algorithm. Next, we computed GainReactive as the difference between the average utility of Reactive and Random. Similarly, we computed GainAnticipatory as the difference between the averages of OptAccUtilExpectation and Reactive. We computed Handicap as the difference between OptAccUtilPerfect and OptAccUtilExpectation. Since OptAccUtilPerfect is run after the resource availability is known for the entire duration, it calculates the maximum possible accrued utility for that resource path.

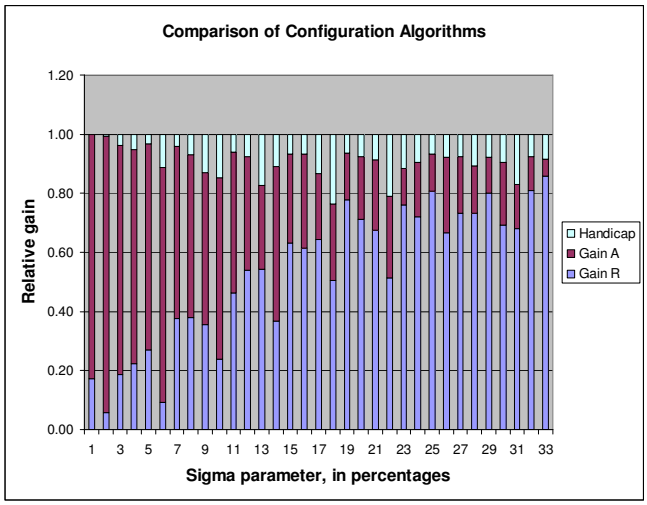

Figure 3: Comparison of the relative gain measures. The stacks, from lower to upper, are: GainReactive, GainAnticipatory, and Handicap.

We varied the *sigma* parameter of the linear predictor from 1% to 33% of the mean (This parameter determines the range of the one-standard deviation band around the predicted mean). Figure 3 shows the resulting graph.

From the graph, we conclude that:

- With accurate prediction, GainAnticipatory exceeds the GainReactive by a factor of 3-4. As prediction accuracy falls, GainAnticipatory disappears,
- With accurate prediction, OptAccUtilExpectation performs very close to OptAccUtilPerfect,.

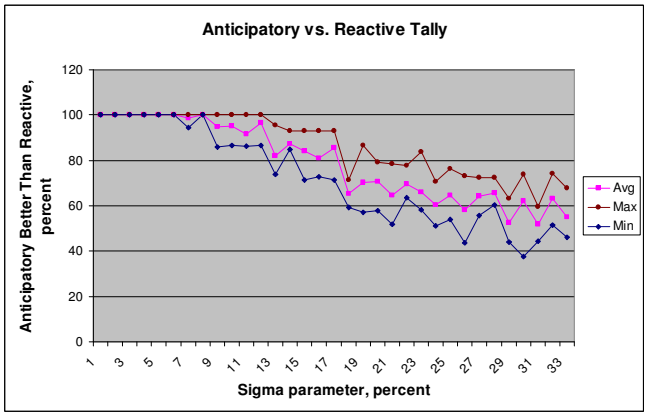

Figure 4: The tally of the percentage of times Anticipatory is better than Reactive. We show the average, maximum, and minimum of 4 experiments.

We also tallied the number of beat OptAccUtilExpectation exceeded Reactive. The results are in Figure 4.

We investigated the effect that the relative magnitude of the switching penalty has on the algorithms. We confirmed that when penalties are very small or very large, there is no substantial difference in the utility of the Anticipatory (Perfect or Expectation) and Reactive algorithms. This is consistent with the observation made in the beginning of this subsection.

5.2.2 Runtime Efficiency

To evaluate the runtime efficiency of the anticipatory algorithms, we measure the resource (CPU) demand and latency of one configuration decision.

We measure the time it takes to make one configuration decision using the standard C clock() function. For OptAccUtilExpectation, we vary the depth of recursive simulation from 2 to 6 (experiments for section 5.2.1 used value of 3). We use two different CPU speed settings: maximum and slowest possible. The results in Table 1 show that the latency of one configuration decision is very small.

Table 1: running times of one invocation of the algorithm, OptActUtilExpectation, in milliseconds. The row labels show the simulation depth, and the column labels show CPU speed. CPU is 100 percent utilized.

	2	3	4	5	6
Max	0.50	1.95	7.65	24.20	72.50
Slow	1.71	6.15	23.30	72.00	201.50

The latencies of both OptAccUtilPerfect and Reactive algorithms are in the nanoseconds and have the same order of magnitude. For comparison, OptAccUtilExpectation with simulation depth of 2 is about 40 times slower than either of the former two. Since the configuration framework has other overheads (communication, application control), the additional time required by the anticipatory algorithm is not significant.

Anticipatory configuration requires one invocation of the algorithm during each time period. The resource demand of the algorithm depends on the size of the time window. This size depends on a number predictor and application parameters, but is on the order of dozens of seconds. Thus, we conclude that the OptAccUtilExpectation algorithm can run in near-real-time with very little CPU demand.

6. EVALUATION

We now refer to the requirements set earlier in the paper and discuss how our work has addressed those. Next, we enumerate the engineering benefits of our work.

6.1 Addressing the Requirements

R1. *Define a measure of accrued utility.* We have introduced a discrete time model and defined accrued utility as the sum of instantaneous over the duration of the task. Accrued utility represents user's satisfaction with the running state of the task over the entire duration, is backwards compatible to the reactive model of configuration and allows comparisons between algorithms, satisfying this requirement.

R2. *Express and combine prediction into the model.* We have formalized three representative predictors and defined a predictor calculus. The linear recent history predictor is grounded by resource prediction research and factors in uncertainty of predictions. The other two predictor types are motivated by other sources of available information.

R3. *Design optimal and efficient algorithms for anticipatory configuration.* We have designed and experimented with two anticipatory algorithms. OptAccUtilPerfect provides a benchmark for maximum possible accrued utility when resource predictions are exact, while OptAccUtilExpectation explicitly handles uncertainty in predictors. With reasonable accuracy of predictors, we have demonstrated that OptAccUtilExpectation nearly always performs better than Reactive and the gain in utility is substantial.

We have also demonstrated that OptAccUtilExpectation is fast and resource efficient, and can be used online on a resource constrained platform.

6.2 Engineering Benefits

We have presented the design and partial implementation of a self-adaptive system that leverages resource predictions for user preference-driven application configuration and resource allocation. We argue that our solution addresses a number of important engineering concerns: (1) combining multiple sources of predictive information, (2) optimal decision making under uncertainty, and (3) runtime efficiency.

Our approach defines three predictor types and a calculus for combining information from multiple sources into a single generalized prediction. The model allows expressing near and long term predictions about resource availability. To the best of our knowledge, previous adaptive systems have not considered multiple sources of information.

The second benefit of our approach is demonstration of improved decision making under uncertainty. Presently, many systems choose to ignore uncertainty: they operate by waiting and then reacting. We have argued that when predictors are sufficiently accurate, system decision making can be improved despite uncertainty.

Third, our approach addresses runtime efficiency, an important concern for self-adaptive systems. We have demonstrated that the anticipatory configuration system can make decisions on-line, while consuming very little resources. The result holds even for resource constrained platforms, making the approach usable in pervasive and mobile environments.

6.3 Limitations

Our approach has limitations:

- while linear recent history predictors are empirically grounded, the other two predictor types are merely plausible. There are no studies that demonstrate the statistical validity of relative move or bounding predictors,
- the results we have demonstrated are sensitive to the choice of preference functions. In particular, for anticipatory configuration to have an edge over the reactive version, the penalty term in instantaneous utility must satisfy certain bounds,
- we have not modeled resources with intertemporal substitution such as battery. This is the subject of future work.

7. CONCLUSION

In this paper we have proposed to leverage research in resource availability prediction to improve the decision making of self-adaptive systems that allocate resources among concurrent fidelity-aware applications. We have demonstrated that anticipatory decision making based on predictions of future resource availability improves over an earlier implemented reactive configuration system, while being resource-efficient.

Our approach presents several engineering benefits, including formal treatment of predictors and analysis of improved utility as a function of predictor accuracy. We conjecture that other self-* systems (e.g., Rainbow [5]) can potentially benefit from predictive information. This paper has provided a blue-print for integrating predictive information in such systems.

8. ACKNOWLEDGMENTS

This work was funded in part by the National Science Foundation under grants CCR-0205266, CCF-0438929, CNS-0613823, by DARPA grant N66001-99-2-8918, and by ETRI Institute in Korea. Authors would like to thank professors Anthony Brockwell and Mahadev Satyanarayanan of Carnegie Mellon and Peter Dinda of Northwestern University for their help in this work.

9. References

[1] P. Dinda. Design, Implementation, and Performance of an Extensible Toolkit for Resource Prediction In Distributed Systems. *IEEE Transactions on Parralel and Dist Syst (TPDS),* 17:2, February 2006.

[2] R. Bent and P. Van Hentenryck. Regrets Only! Online Stochastic Optimization under Time Constraints. *Proc 19th National Conf on Artificial Intelligence (AAAI)*, 2004.

[3] P. Dinda, D. O'Hallaron. Host Load Prediction Using Linear Models. *Cluster Computing*, 3:4, 2000.

[4] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste. Project Aura: Towards Distraction-Free Pervasive Computing. *IEEE Pervasive Computing,* Volume 21, Number 2, April-June, 2002.

[5] D. Garlan, S. Cheng, et al. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *IEEE Computer*, 37:10, 2004.

[6] P. Hentenryck, et al. Online Stochastic Optimization Under Time Constraints. Working paper, last accessed in February 2007 at http://www.cs.brown.edu/people/pvh/aor5.pdf.

[7] E. de Lara, D. S. Wallach, W. Zwaenepoel. Puppeteer: Component-based Adaptation for Mobile Computing. *Proc. USENIX Symp on Internet Technologies and Systems (USITS)*, 2001.

[8] C. Lee. On Quality of Service Management. *PhD Thesis, Carnegie Mellon University Technical Report CMU-CS-99-165,* 1999.

[9] D. Narayanan, J. Flinn, M. Satyanarayanan. Using History to Improve Mobile Application Adaptation. *Proc. 3rd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, 2000.

[10] R. Neugebauer and D. McAuley. Congestion Prices as Feedback Signals: An Approach to QoS Management. *Proc. ACM SIGOPS European Workshop*, 2000.

[11] B. Noble, et al. Agile Application-Aware Adaptation for Mobility. *Proc. ACM Symp on Operating Systems Principles (SOSP)*, 1997.

[12] V. Poladian, et al. Dynamic Configuration of Resource-Aware Services. *Proc IEEE Intl Conf on Software Engineering (ICSE)*, 2004.

[13] Y. Qiao, J. Skicewicz, P. Dinda. An Empirical Study of the Multiscale Predictability of Network Traffic. *Proc Intl Symp on High Perf Dist Computing (HPDC)*, 2004.

[14] A. Sang and S. Li. Predictability analysis of network traffic. *Proc. of INFOCOM*, 2000.

[15] J.P. Sousa, D. Garlan. The Aura Software Architecture: an Infrastructure for Ubiquitous Computing. *Carnegie Mellon Technical Report, CMU-CS-03-183*, 2003.

[16] R. Wolski, et al. The network weather service: A distributed resource performance forecasting system. *J. of Future Generation Computing Systems*, 1999.

[17] R. Wolski, et al. Predicting the CPU availability of time-shared Unix systems. *Proc Intl Symp High Perf Dist Computing (HPDC)*, 1999.

10. APPENDIX A

In this appendix we reproduce the formal definition of instantaneous utility originally published in [12]. The text is copied nearly verbatim.

10.1.1 QoS Preferences

QoS preferences specify the utility function associated with each QoS dimension. The names of the QoS dimensions are also part of the shared vocabulary. The utility of service *svc* as a function of the quality of service is given by:

$$U_{QoS}(svc) \triangleq \prod_{d \in QoS\ \dim(svc)} F_d^{c_d}$$

where for each QoS dimension d of service *svc*, $F_d : dom(d) \rightarrow (0,1]$ is a function that takes a value in the domain of d, and the weight $c_d \in [0,1]$ reflects how much the user cares about QoS dimension d. As an example, *video playing* has a QoS dimension of *frame update rate*. The function $F_{frameRate}$ gives utility for various frame rates, and $c_{framerate}$ specifies the weight of frame rate.

Weighted product specifies an "AND" semantics when combining QoS dimensions. A utility value of zero in one dimension indicates that the user is not interested in the configuration even if the quality of other dimensions is high.

10.1.2 Supplier Preferences And Switching Penalty

To evaluate the assignment of specific suppliers, we employ a supplier preference function, which is a discreet function that assigns a score to a supplier, based on its type. Also, we account for the *cost of switching* from one supplier to another at run time.

Precisely, the utility of the supplier assignment for a set a of requested services is:

$$U_{Supp}(a) \triangleq \prod_{svc \in a} h_{svc}^{x_{svc}} \cdot F_{svc}^{c_{svc}}$$

where for each service *svc* in the set a, F_{svc} *: Supp(svc) -> (0,1]* is a function that appraises the choice of a supplier for service *svc*; and the weight $c_{svc} \in [0,1]$ reflects how much the user cares about the supplier assignment for that service.

The term $h_{svc}^{x_{svc}}$ above (10.1.2) expresses a change penalty as follows: h_{svc} indicates the user's tolerance for a change in supplier assignment: a value close to 1 means that the user is fine with a change, the closer the value is to zero, the less happy the user will be. The exponent x_{svc} indicates whether the change penalty should be considered (x_{svc}=1 if the supplier for s is being exchanged by virtue of dynamic change in the environment) or not (x_{svc}=0 if the supplier is being newly added or replaced at the user's request).

10.1.3 Instantaneous Utility

Overall utility is the product of the QoS preference, supplier preference, and change penalty. Let *a'* be the previous assignment of suppliers and a be the current. Then the instantaneous utility is:

$$IU(a,a') = \prod_{svc \in a} h_{svc}^{x_{svc}} \cdot F_{svc}^{c_{svc}} \left(\prod_{d \in QoS\ \dim(svc)} F_d^{c_d} \right)$$