Dynamic Scheduling of Progressive Processing Plans

Shlomo Zilberstein[1] and **Abdel-Illah Mouaddib**[2] and **Andrew Arnt**[3]

Abstract. Progressive processing plans allow systems to tradeoff computational resources against the quality of service by specifying alternative ways in which to accomplish each step. When the structure of a plan is known in advance, it can be optimally scheduled by solving a corresponding Markov decision process. This paper extends this approach to dynamic scheduling of plans that can be constantly modified. We show how to construct an optimal meta-level controller for a single task and how to extend the solution to the case of multiple and dynamic tasks using the notion of an opportunity cost. Several fast approximation schemes for the opportunity cost are evaluated. The results provide an effective framework for managing computational resources in highly dynamic environments.

1 INTRODUCTION

This paper is concerned with dynamic scheduling of progressive processing task structures. In this framework, each task is mapped to a *progressive processing unit* (PRU) composed of a set of modules that can contribute to the quality of the result. The problem is to select at run-time the best subset of modules so as to maximize the quality of the result produced with limited computational resources.

While the general framework is extremely general, we focus in this paper on a particular information retrieval application. Information retrieval from a large collection involves uncertainty regarding the duration of the process and the quality of the result. In addition, there may be large variability in the number of requests that require a response at any given time. By taking a context dependent, dynamic approach to the problem we can significantly improve the average quality of service provided by such systems.

A typical search engine is composed of several information retrieval modules that perform such tasks as query formation, query optimization, query evaluation, precision improvement, recall improvement, clustering, and results visualization. For each one of these phases, there are a wide variety of techniques that have been developed in recent years [12]. Currently, search engines are built by choosing and integrating a fixed set of modules and techniques. The choices are made off-line by the designer of the system. This static approach excludes techniques that work well in special situations. In addition, current information retrieval systems are optimized for a particular load; they cannot respond dynamically to varying load, availability of computational resources, and to the specific characteristics of a given query.

The ability to dynamically adjust computational effort based on the availability of computational resources has been studied extensively by the AI community since the mid 1980's. These efforts have led to the development of a variety of techniques such as *anytime algorithms* [1, 13], *design-to-time* [2], *flexible computation* [5], *imprecise computation* [7], and *progressive reasoning* [8, 9].

In particular, the progressive processing approach offers a natural framework to describe the set of information retrieval techniques available to the system. Figure 1 shows a simple task structure whose input is a *query* composed of a list of keywords. The task structure has three processing *levels*. The first level includes three alternative techniques to improve the initial query: (a) scan the query using concept recognizers to identify company names, dates, locations, personal names, and so on; (b) examine the query for pairs of words that have high statistical likelihood of being related and enhance the query with that information; (c) perform part-of-speech analysis to identify noun phrases within the query. The second level includes two alternative techniques that can improve the query's recall ability by expanding it to include related words and phrases: (d) use of Local Context Analysis (LCA), a statistical method for expanding queries that relies upon in-context analysis of word co-occurrence; (e) use of InFinder, an association thesaurus that is faster than LCA and does not capture context as well. Finally, the third level performs the actual query evaluation and returns the results. Quality in this application is measured by the number of relevant documents within the top n documents retrieved (i.e., precision in the retrieved set).

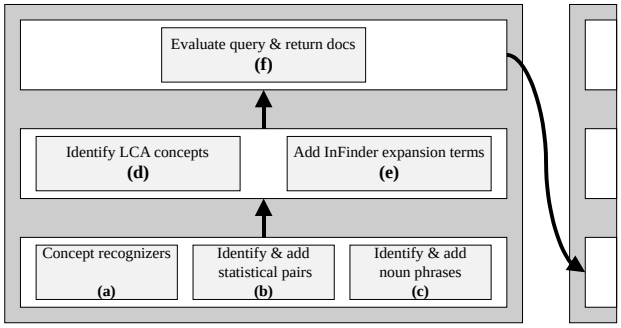

Figure 1. Illustration of a progressive processing task for an information retrieval search engine

This information retrieval application provides a good example of several fundamental issues:

1. Handling the *duration uncertainty* and *quality uncertainty* associated with each technique.
2. Handling the *dependency* of quality and duration on the quality of intermediate results.
3. Handling a rich task structure in which some levels include several *alternatives* or optional computational steps; optional steps can be skipped under time pressure, leading to direct evaluation of the

[1] Department of Computer Science, University of Massachusetts, Amherst, MA 01003, USA, email: zilberstein@cs.umass.edu
[2] CRIL-IUT de Lens-Université d'Artois, Rue de l'université, S. P. 16, 62307 Lens Cedex, France, email: mouaddib@cril.univ-artois.fr
[3] Department of Computer Science, University of Massachusetts, Amherst, MA 01003, USA, email: arnt@cs.umass.edu

input query.

4. Selecting the optimal set of retrieval techniques in a *dynamic* environment taking into account the entire set of queries waiting for execution.

The rest of this paper offers an efficient solution to the meta-level control problem. Section 2 gives a formal definition of the problem. We then solve the problem in two steps. In Section 3, we develop an optimal solution for a single PRU, ignoring the fact that additional tasks are waiting for processing. Section 4 shows how to handle multiple PRUs using the approach developed in Section 3 and summarizing the effect of the waiting requests using the notion of an opportunity cost. In section 5 we address the issue of reactive control in a highly dynamic environment by estimating the opportunity cost and pre-compiling the control policies. We conclude with a summary of the results and a brief description of related work.

2 THE META-LEVEL CONTROL PROBLEM

This section describes formally the problem of meta-level control of the progressive processing model. Each information retrieval request is mapped to a task structure described below.

Definition 1 *A* **progressive processing unit** *(PRU) is composed of a sequence of processing levels,* $(l_1, l_2, \ldots, l_L)$*. The first level receives the input query and the last one produces the result.*

Definition 2 *Each processing level,* l_i*, is composed of a set of* p_i **alternative modules**, $\{m_i^1, m_i^2, \ldots, m_i^{p_i}\}$.

Each module can perform the logical function of level l_i, but it has different computational characteristics defined below.

Definition 3 *The* **module descriptor**, $P_i^j((q', \delta)|q)$*, of module* m_i^j *is the probability distribution of output quality and duration for a* given *input quality.*

Note that q is a discrete variable representing quality and δ is a discrete variable representing duration. The module descriptor specifies the probability that module m_i^j takes δ time units and returns a result of quality q' when the quality of the previously executed module is q. Module descriptors are similar to *conditional performance profiles* of anytime algorithms [13]. They are constructed empirically by collecting performance data for a sample set of inputs.

When the search engine responds to a particular request, it receives an immediate reward defined as follows.

Definition 4 *A* **time-dependent utility function**, $U(q, t)$*, measures the utility of a solution of quality* q *if it is returned* t *time units after the arrival time of the request.*

We assume that there is a given constant T such that $\forall q, t > T : U(q, t) = 0$. That is, responding to a request more than T time units after its arrival has no value.

Suppose that a system maintains a set of information retrieval requests, W, with arrival times $\{a_1, a_2, \ldots, a_n\}$. The set of requests is updated dynamically as new requests arrive. The system processes the requests in a first-in-first-out order using a progressive processing unit to handle each request.

Given a set of requests, the module descriptors of all the components of the progressive processing unit, and a time-dependent utility function, we define the following control problem.

Definition 5 *The* **reactive control problem** *is the problem of selecting a set of alternative modules so as to maximize the expected utility over the set of information retrieval requests.*

The meta-level control is "reactive" in the sense that we assume that the module selection mechanism is very fast, largely based on offline analysis of the problem. The rest of the paper provides a solution to this problem.

3 OPTIMAL CONTROL OF A SINGLE PRU

We begin with the problem of meta-level control of a single progressive processing unit corresponding to a single task. This problem can be formulated as a simple Markov decision process (MDP) with states representing the current state of the computation. The state includes the current level of the PRU, the quality produced so far, and the elapsed time since the arrival of the request. The rewards are defined by the utility of the solution which depends on both quality and time. The possible actions are to *execute* a module of the next processing level or to *skip* that processing level. The transition model is defined by the descriptor of the module selected for execution. The rest of this section gives a formal definition of the MDP and the reactive controller produced by solving it.

3.1 State representation

The execution of a single progressive processing unit, u, can be seen as an MDP with a finite set of states $\mathcal{S} = \{[l_i, q, t] | l_i \in u\} \bigcup \{[\text{failure}, t]\}$ where $0 \leq i \leq L$ indicates the last executed (or skipped) level, $0 \leq q \leq 1$ is the quality produced by the last executed module, and $0 \leq t \leq T$ is the elapsed time since the arrival time, a_u, of the request. Note that quality is discretized and normalized to be in the range $[0..1]$. All the intermediate modules use a uniform representation of input and output (a "query" in our application). Note also that T is the maximum delay after which we consider the response to be useless. When the system is in state $[l_i, q, t]$, one module of the i-th level has been executed. (The first level is $i = 1$; $i = 0$ is used to indicate the fact that no level has been executed.) The states $[\text{failure}, t]$ represent termination at time t without any useful result. We distinguish between different failure states because failure can occur before the deadline leaving some remaining time for the execution of other requests in the queue.

3.2 Transition model

The initial state of the MDP is $[l_0, q_{init}, t]$, where t is the elapsed time since the arrival of the request ($t =$ current time $- a_u$) and q_{init} is the initial quality of the request (0 in our application). The initial state indicates that the system is ready to start executing a module of the first level of the PRU. The terminal states are all the states of the form $[l_L, q, t]$ or $[failure, t]$. The former set represents finishing execution of the last level and the latter set represents failure. Other states such as $[l_i, q_{max}, t]$ (reaching maximal intermediate quality) or $[l_i, q, T]$ (reaching the deadline before the execution of the last level) are not considered terminal states. A terminal state can be reached from state $[l_i, q, T]$ by executing a series of *skip* actions until a failure state is reached. Similarly *skip* actions take the automaton from state $[l_i, q_{max}, t]$ to the last level because no *execute* action can improve the intermediate quality.

In every nonterminal state the possible actions are: $\mathbf{E}_{i+1}^j$ (execute the j-th module of the next level) and **S** (skip the next level). To

complete the transition model, we need to specify the probabilistic outcome of these actions. Equations 1-4 define the transition probabilities for a given nonterminal state $[l_i, q, t]$.

The **S** action is deterministic. It skips the next level without affecting the quality or elapsed time. (It can be implemented as an additional "dummy" module whose execution takes no time and has no effect on quality.)

$$Pr([l_{i+1}, q, t] \mid [l_i, q, t], \mathbf{S}) = 1$$
$$\text{when } 0 \leq i < L-1 \tag{1}$$

Skipping the last level results in failure.

$$Pr([failure, t] \mid [l_{L-1}, q, t], \mathbf{S}) = 1 \tag{2}$$

The $\mathbf{E}^j_{i+1}$ action is probabilistic. Duration and quality uncertainties define the new state. Equation 3 determines the transitions following successful execution and Equation 4 determines the transition to the failure state when the deadline, T, is reached.

$$Pr([l_{i+1}, q', t+\delta] \mid [l_i, q, t], \mathbf{E}^j_{i+1}) =$$
$$P^j_{i+1}((q', \delta)|q) \quad \text{when } t+\delta \leq T \tag{3}$$

$$Pr([failure, T] \mid [l_i, q, t], \mathbf{E}^j_{i+1}) =$$
$$\sum_{q', \delta > T-t} P^j_{i+1}((q', \delta)|q) \tag{4}$$

3.3 Rewards and the value function

Rewards are determined by the given time-dependent utility function applied to the final result (produced by the last level of the PRU). The utility depends on the quality of the result and the elapsed time. Keep in mind that in our application the intermediate results are useless and therefore have no direct rewards associated with them. We now define a value function (expected reward-to-go) over all states. The value of terminal states is defined as follows.

$$V([l_L, q, t]) = R(q, t) = U(q, t) \tag{5}$$

$$V([failure, t]) = R(0, t) = U(0, t) \tag{6}$$

The value of nonterminal states of the MDP is defined as follows.

$$V([l_i, q, t]) =$$
$$\max_a \begin{cases} V([l_{i+1}, q, t]) & \text{If } a = \mathbf{S}, 0 \leq i < L-1 \\ V([failure, t]) & \text{If } a = \mathbf{S}, i = L-1 \\ EV([l_i, q, t] \mid \mathbf{E}^j_{i+1}) & \text{If } a = \mathbf{E}^j_{i+1}, 0 < j \leq p_i \end{cases} \tag{7}$$

Such that $EV([l_i, q, t] \mid \mathbf{E}^j_{i+1})$ =

$$\sum_{q', \delta > T-t} P^j_{i+1}((q', \delta)|q)\, V([failure, T]) +$$
$$\sum_{q', \delta \leq T-t} P^j_{i+1}((q', \delta)|q)\, V([l_{i+1}, q', t+\delta])$$

The value function is defined as maximum over all actions with the top expression representing the value of a skip action for any level l_i such that $0 \leq i \leq L-1$, the middle expression representing the value of a skip action for level l_{L-1}, and the bottom expression representing the value of an execute action.

This concludes the definition of an MDP. This MDP is a finite-horizon MDP with no cycles. It can be solved easily using standard dynamic programming algorithms or using search algorithms such as AO*.

Theorem 1 *Given* one *progressive processing unit u and a time-dependent utility function $U(q, t)$, the optimal policy for the corresponding MDP is an optimal reactive control for u.*

Proof: Because there is a one-to-one correspondence between the reactive control problem and the MDP (including the fact that the PRU transition model satisfies the Markov assumption), and because of the optimality of the resulting policy, we conclude that it provides optimal reactive control for the progressive processing problem. □

3.4 Choice of unit resolution

The number of states of the MDP we must solve to control a single PRU is bounded by the product of the number of levels L, the maximum number of alternative modules per level $\max_i p_i$, the number of discrete quality levels, and the maximum execution time. While the maximum execution time can be quite large, the time unit used for the purpose of meta-level control is an arbitrary system parameter. A small time unit leads to a more effective control at the expense of a larger state-space. The choice of a unit of quality has a similar effect. These units introduce a tradeoff between the size of the policy and its effectiveness. We evaluate this tradeoff below be measuring the policy size and construction time for different unit sizes. For the sake of simplicity, the same unit reduction factor, u, is used for both time and quality.

In this experiment, quality (q) and time (t) have the following ranges:

$$0 \leq q \leq 100$$
$$0 \leq t \leq T$$

where T, the failure state deadline, is between 300 and 1000. The unit resolution, u, defines the number of base level units grouped together into a larger unit size. The following table shows the number of discrete states per each level of the MDP for $T = 300$ and $u = 1, 5, 10, 20, 40, 80$.

Table 1. States per level as a function of unit resolution

u	# of t values	# of q values	states per level
1	301	101	30401
5	61	21	1281
10	31	11	341
20	16	6	96
40	8	3	24
80	5	2	10

The experiments were conducted with five randomly generated PRUs for each of the four types described in the table below. Type A is representative of the characteristics of an actual information retrieval application, while the others are used to test scalability.

For each PRU, the corresponding MDP was solved using the above six different unit values. For each resolution u, an optimal policy

Table 2. States per level as a function of unit resolution

PRU type	L	Modules per level	T
A	3	6	300
B	3	15	300
C	3	6	1000
D	3	15	1000

P_u was constructed. The value of the initial state, $[l_0, q_0, t_0]$, for $u = 1$ represents the precise initial state expected value. The policy P_k was used to select actions in a simulation starting from the initial state of the MDP. The simulation traced the precise state (with units size $u = 1$) while selecting actions based on the *approximate* policy. This simulation was repeated 1000 times for each u, recording the returned value (reward) of the initial state. Finally, we computed the relative error between the value achieved using P_k and the exact value for each type of PRUs, using five random cases of each type. The results are summarized in the following table.

Table 3. The effect of unit resolution on policy value and construction time

PRU Type	Exp Value	u	Avg Const Time(H:M:S)	Avg Value	Avg % Error
A	36.597	1	1:28:07.624	36.488	-0.297
		5	0:00:10.474	36.591	-0.017
		10	0:00:00.897	35.905	-1.891
		20	0:00:00.093	35.499	-3.001
		40	0:00:00.011	34.789	-4.942
		80	0:00:00.004	28.397	-22.406
B	14.254	1	3:27:20.762	14.301	0.329
		5	0:00:25.289	14.188	-0.469
		10	0:00:02.109	13.303	-6.676
		20	0:00:00.223	10.219	-28.310
		40	0:00:00.027	4.449	-68.786
		80	0:00:00.008	-10.279	-172.112
C	30.429	1	4:16:10.096	30.074	-1.167
		5	0:00:30.377	30.067	-1.189
		10	0:00:02.492	29.981	-1.472
		20	0:00:00.026	29.137	-4.245
		40	0:00:00.031	22.781	-25.135
		80	0:00:00.008	19.436	-36.127
D	21.378	1	13:18:36.698	21.464	0.401
		5	0:01:35.471	21.039	-1.585
		10	0:00:07.781	21.191	-0.876
		20	0:00:00.805	14.678	-31.338
		40	0:00:00.094	15.604	-27.007
		80	0:00:00.023	14.084	-34.118

Several important observations can be made based on the above table. First, it confirms the intuition that the value of a policy degrades gracefully as the unit size increases. But more importantly, the table shows that a unit size of 10 leads to a dramatic reduction in policy construction time with only a small relative error. For example, for type A PRUs, the time reduction is from more than 88 minutes to less than 1 second. The loss of value is less than 2%. These results confirms the applicability of the approach to realistic problems by adopting a good unit resolution.

4 OPTIMAL CONTROL OF MULTIPLE UNITS USING OPPORTUNITY COST

Suppose now that we need to schedule the execution of multiple PRUs. We assume that there are $n + 1$ requests whose arrival times are $a_0 \leq a_1 \leq \ldots \leq a_n$. One approach to construct an optimal schedule is to generalize the solution presented in the previous section. We can construct a larger MDP for the combined sequential decision problem including the entire set of $n + 1$ PRUs. To do that, each state must also include i, the request number, leading to a general state represented as $[i, l, q, t]$. Note that t is the elapsed time since the arrival of the *first* request.

This rather complex MDP is still a finite-horizon MDP with no loops. Moreover, the only possible transitions between different PRUs are from a terminal state of one PRU to an initial state of a succeeding PRU. Therefore, we can solve this MDP by computing an optimal policy for the *last* PRU for any starting time between 0 and $T + a_{n+1} - a_0$, then use the value of its initial states to compute an optimal policy for the previous PRU and so on.

Theorem 2 *Given a set, W, of progressive processing units and a time-dependent utility function $U(q, t)$, the optimal policy for the corresponding MDP is an optimal reactive control for W.*

This is an obvious generalization of Theorem 1. The complete proof, by induction on the number of PRUs, is omitted.

We now show how to reformulate the effect of the remaining n requests on the execution of the first task. This reformulation preserves the optimality of the solution, but it suggests a more efficient control structure developed in Section 5.

Definition 6 *Let $V_i^*(t) = V([i, l_0, q_0, t])$ denote the expected value of the optimal policy for the last $n - i + 1$ PRUs.*

To compute the optimal policy for the i-th PRU, we can simply use the following reward function.

$$R_i(q, t) = U(q, t + a_0 - a_i) + V([i + 1, l_0, q_0, t]) \quad (8)$$

In other words, the reward for responding to the i-th request is composed of the immediate reward (defined by the time-dependent utility function) and the reward-to-go (defined by the remaining PRUs). Alternatively, the reward can be represented as follows.

$$R_i(q, t) = U(q, t + a_0 - a_i) + V_{i+1}^*(t) \quad (9)$$

Therefore, the best policy for the first PRU can be calculated if we use the following reward function for final states:

$$R_0(q, t) = U(q, t) + V_1^*(t) \quad (10)$$

Definition 7 *Let $OC(t) = V_1^*(0) - V_1^*(t)$ be the* **opportunity cost** *at time t.*

The opportunity cost measures the loss of expected value due to delay in the starting point of executing the last n tasks (all the tasks except the first one).

Definition 8 *Let the* **OC-policy** *for the first PRU be the policy computed with the following reward function:*

$$R(q, t) = U(q, t) - OC(t)$$

The OC-policy is the policy computed by deducting from the actual reward for the first task the opportunity cost of its execution time.

Theorem 3 *Controlling the first PRU using the OC-policy is optimal.*

Proof: From the definition of $OC(t)$ we get:

$$V_1^*(t) = V_1^*(0) - OC(t) \tag{11}$$

To compute the optimal schedule we need to use the reward function defined in Equation 9 that can be rewritten as follows.

$$R_0(q,t) = U(q,t) + V_1^*(0) - OC(t) \tag{12}$$

But this reward function is the same as the one used to construct the OC-policy, except for the added constant $V_1^*(0)$. Because adding a constant to a reward function does not affect the policy, the conditions of Theorem 2 are met and the resulting policy is optimal. □

Theorem 3 suggests an optimal approach to scheduling the entire $n+1$ requests by first using an OC-policy for the first request that takes into account the opportunity cost of the remaining n requests. Then the OC-policy for the second request is used taking into account the opportunity cost of the remaining $n-1$ tasks and so on. To be able to implement this approach we need to have the control policies readily available. This issue is addressed in the following section.

5 REACTIVE CONTROL BASED ON ESTIMATED OPPORTUNITY COST

In the previous section, we presented an optimal solution to the control problem of multiple progressive processing units without accounting for its computational complexity. In particular, the opportunity cost must be computed and revised quickly each time a new request arrives. Once the opportunity cost is revised, a new policy for the current PRU must be constructed. Finding the exact opportunity cost requires the construction of an optimal policy for the entire set of tasks. In practice, this may slow down the operation of the information retrieval search engine.

In order to provide an effective reactive controller for dynamic progressive processing, it is necessary to:

1. use a fast approximation scheme to estimate the opportunity cost; and
2. use pre-compiled policies for different opportunity cost functions.

The rest of this section explains this method in more detail.

5.1 Estimating the Opportunity Cost

The opportunity cost is defined in terms of the function V_1^* which represents the value of an optimal policy for the remaining tasks in the queue. Thus, it can be estimated by approximating this function.

5.1.1 Naïve approximation

A naïve approach to approximating the cumulative value of the remaining tasks is to add the value of each task *without* taking into account the opportunity cost. In this calculation, the start time of each task is the *expected* end time of the previous one. The following set of equations summarizes this approximation scheme.

$$\begin{aligned}
&V_1^*(t) \simeq V([l_0, q_0, t + a_0 - a_1]) + V_2^*(t + \tau_1) \\
&\vdots \\
&V_i^*(t) \simeq V([l_0, q_0, t + a_0 - a_i]) + V_{i+1}^*(t + \textstyle\sum_{j=1}^{j=i} \tau_j) \\
&\vdots \\
&V_n^*(t) = V([l_0, q_0, t + a_0 - a_n])
\end{aligned} \tag{13}$$

where $V[l,q,t]$ is the value function defined in Section 3 for a single PRU. Therefore, V_1^* can be approximated as follows.

$$V_1^*(t) \simeq \sum_{i=1}^{i=n} V[l_0, q_0, t + a_0 + (\sum_{j<i} \tau_j) - a_i] \tag{14}$$

The expected duration of task i, τ_i, depends on the duration of the previous tasks. Let $\tau(d)$ be the expected duration of the optimal single-PRU policy when starting at time d (relative to the arrival time of the request). Then τ_i is computed using τ with the expected starting time of task i relative to *its* arrival time.

$$\begin{aligned}
&\tau_0 = 0 \\
&\tau_i = \tau(t + a_0 + (\textstyle\sum_{j<i} \tau_j) - a_i)
\end{aligned} \tag{15}$$

The function τ (expected duration) can be computed for any finite-horizon MDP once the optimal policy is available by simply using durations as rewards. The function can be computed once off-line, making it easy to revise the opportunity cost when a new request is added.

5.1.2 Learning an approximate opportunity cost function

Another approach is to estimate the opportunity cost using some features that characterize the remaining PRUs in the queue. Using a data set of pre-computed opportunity costs for many different queues, we can use these features to quickly approximate the opportunity cost for the current queue. The features used is our experiment are:

1. The total number of PRUs in the queue.
2. The average waiting time of a PRU in the queue.

We performed some experiments to determine the effectiveness of this approach. The dataset of queues was generated using a simple model of query arrival time (a random number between 0 and 3 requests arrive over a period of ten time units). The exact opportunity cost was computed at each of the time units for 100 randomly generated queues.

Two different estimation methods have been tested. The first was the k-Nearest-Neighbor algorithm, where we use a Euclidean distance metric on the features to determine the k closest data queues to the test queue. The estimated opportunity cost for each time unit is then the average of the opportunity costs over the k data queues for that time unit.

The second method was kernel regression, where we give all of the data queues a weight based on their feature-based Euclidean distance to the test queue. The weighing function for data queue i is as follows.

$$w_i = e^{\frac{-D(i,test)^2}{W}} \tag{16}$$

Where $D(x_i, test)$ is the distance between i and the test queue, and W is the kernel width. The estimated opportunity cost for time unit t is then the weighted average over all of the data queues.

$$est\_oc_t = \frac{\sum w_i oc_{t_i}}{\sum w_i} \tag{17}$$

To determine the effectiveness of each of these methods we used leave-one-out cross validation. In L-O-O CV, for each data queue i, we omit i from the data set and then use our approximation method to estimate the opportunity costs for i. We then compute the error e_i

between the estimated values and the actual values for i. Two different methods for computing e_i were tested.
Method 1:

$$e_i = \left(\sum_{t=0}^{10} \frac{|oc_{t_{actual}} - oc_{t_{est}}|}{oc_{t_{actual}}} \right) / 10 \quad (18)$$

Method 2:

$$e_i = \frac{\sum_{t=0}^{10} |oc_{t_{actual}} - oc_{t_{est}}|}{\sum_{t=0}^{10} oc_{t_{actual}}} \quad (19)$$

Method 1 averages together the fraction error from each time step, while Method 2 finds the error of the sum of all the opportunity costs. Then, the total cross validation error is the average of all of the e_i.

Figures 2 and 3 show that an acceptable error of less than 0.1 is achieved using 1 Nearest Neighbor or Kernel Regression with a very small width. An estimate with such a small error suits our needs.

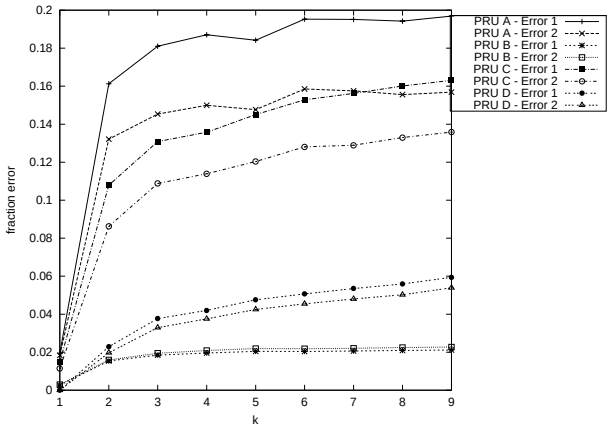

Figure 2. Leave one out cross validation error for k nearest neighbor

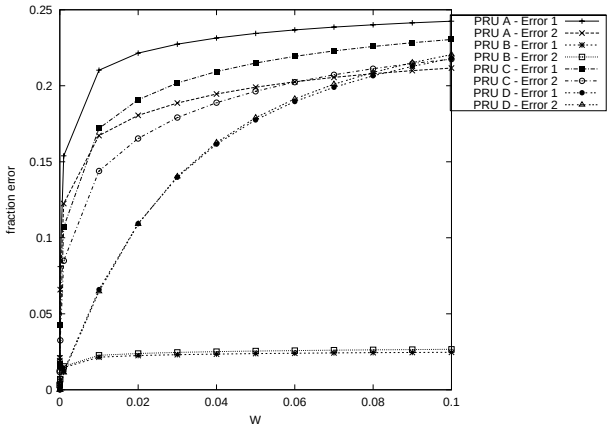

Figure 3. Leave one out cross validation error for kernel regression with width W

5.1.3 Comparison of opportunity cost approximation methods

We now compare the performance of the naïve method for opportunity cost approximation against the best feature-based function approximation technique, 1-Nearest-Neighbor, described above. Performance using no opportunity cost is given for comparison. This is based on an optimal policy for a single task, ignoring the entire queue of requests. Unlike the naïve method, in this case the the reward function does not take into account the reward-to-go.

To perform this comparison, 50 different PRU arrival queues were randomly generated for each of the four PRU types described in Table 2. They were generated by having a random number between 0 and 3 requests arrive at each time unit over a period of ten time units (using a unit size of 10 seconds). Note that all the PRUs in a given queue correspond to the same task structure. For each arrival queue, we estimated $OC(t)$ at each of the possible arrival times using both estimation methods. We then computed the actual opportunity cost. Table 4 gives the average relative error for each of the methods. As expected, 1NN generally outperforms the naïve method.

We also observed how often actions chosen by the estimated OC policy differed from those specified by the optimal policy. These values are also given in Table 4. We see that both of the estimation methods perform very well, with the 1NN method actually generating a policy identical to the optimal for PRUs of type A. Ignoring the opportunity cost leads to a large action error (up to 35%). It is interesting to note that in PRUs of type D, the action error is small for all three approaches. The large number of alternatives and high level of uncertainty about duration make the value of the second-best action closer to the value of the best action. Note also that in this case the naïve method provides the more accurate estimate of OC, but it also leads to larger action selection. A possible explanation is that while the estimate is more accurate in general, it is less accurate for some critical cases in which a small error makes a difference in action selection.

Table 4. Comparison of OC approximation methods

PRU type	OC Est Method	Est OC Error	Action Error
A	None	N/A	20.345
	Naïve	34.102	2.014
	1NN	7.178	0.0
B	None	N/A	18.422
	Naïve	10.550	5.586
	1NN	2.813	4.678
C	None	N/A	35.185
	Naïve	6.042	0.971
	1NN	2.668	0.233
D	None	N/A	1.453
	Naïve	1.142	1.302
	1NN	2.255	0.376

5.2 Pre-compiled control policies

To make the meta-level control truly reactive for large task structures, one may want to avoid computing a new policy (for a single PRU) each time the opportunity cost is revised. To avoid this, the space of opportunity cost can be divided into a small set of regions representing typical situations. For example, there could be just three regions that capture *low*, *medium*, and *high* loads. For each region, an optimal policy would be computed off-line and stored in a library. At run-time, the system will first estimate the opportunity cost and then use the appropriate pre-compiled policy from the library. These policies remain valid as long as the overall task structure and the utility function are fixed. Because the dependency of the control decisions

on the opportunity cost is monotonic (higher costs imply less time for execution), we anticipate that a small set of classes that correspond to *qualitatively* different action selection will be sufficient.

Another advantage of the use of pre-compiled policies is the ability to react quickly to dynamic changes. Control policies can be switched *during* the execution of a single request if the opportunity cost changes. This is possible because the policies share the same state space.

6 CONCLUSION

We present an innovative approach to meta-level control of progressive processing based on reformulating it as a Markov decision problem. It is shown that an optimal policy for a set of tasks can be constructed by controlling a single PRU, taking into account the opportunity cost of the remaining tasks. To apply this model to control the operation of an information retrieval search engine, a fast approximation of the opportunity cost is developed. Finally, a highly reactive controller is described that uses a library of pre-compiled control policies to operate in a dynamic environment.

A less complex model of progressive processing that relies on heuristic scheduling has been developed [9]. The task structure, however, is limited to a linear set of levels with one module per level and no quality uncertainty or quality dependency. The heuristic scheduler is fast, but it cannot solve the more complex task structure presented in this paper and it does not provide optimal control. Heuristic scheduling of computational tasks has also been studied by Garvey and Lesser [1993] for the *design-to-time* problem-solving framework. The latter framework represents explicitly non-local interactions between sub-tasks.

The progressive processing framework relates to a large body of work within the systems community on *imprecise computation* [7]. Each task in that model is decomposed into a *mandatory* subtask and an *optional* subtask. A variety of scheduling algorithms have been developed for imprecise computation under different assumptions about the optional part. Our model allows for a richer representation of quality and duration uncertainty and quality dependency. Unlike imprecise computation, the schedule constructed by the MDP scheduler is a *conditional schedule*; the selection of modules is conditioned on the *actual* execution time and outcome of previous modules.

The application of dynamic programming to solve meta-level control problems have been previously used by Hansen and Zilberstein [1996] to control interruptible anytime algorithms. Optimal monitoring of progressive processing tasks using a corresponding MDP has been studied by Mouaddib and Zilberstein [1998] with respect to a simpler task structure and without the notion of quality uncertainty and quality dependency.

The notion of *opportunity cost* is borrowed from economics. It has been used previously in meta-level reasoning by Russell and Wefald [1991]. Horvitz [1997] uses a similar notion to develop a model of *continual computation* in which idle time is used to solve anticipated future problems.

The use of pre-compiled control policies to construct a highly reactive real-time system has been studied by several researchers. For example, Greenwald and Dean [1998] show how a real-time avionics control system can use a library of schedules that cover all possible situations. Each schedule is conditioned on the state of the flight operation.

In collaboration with the Information Retrieval Center at UMass we are currently developing the stochastic module descriptors for the components of the search engine. By definition, IR tasks involve large collections and a substantial amount of test data allowing us to test the applicability and scalability of this resource-bounded reasoning technique.

ACKNOWLEDGMENTS

We thank James Allan and Victor Lavrenko for their contribution to the problem formulation and to the construction of the information retrieval testbed.

This work was supported in part by the National Science Foundation under grants No. IRI-9624992, IIS-9907331, and INT-9612092, by the GanymedeII Project of Plan Etat/Nord-Pas-De-Calais, and by IUT de Lens.

REFERENCES

[1] T. Dean and M. Boddy, An analysis of time-dependent planning, *Seventh National Conference on Artificial Intelligence*, 49–54, 1988.

[2] A. Garvey and V. Lesser, Design-to-time real-time scheduling, *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6):1491–1502, 1993.

[3] L. Greenwald and T. Dean, A conditional scheduling approach to designing real-time systems, *AI Planning systems*, 1229–1234, 1998.

[4] E.A. Hansen and S. Zilberstein, Monitoring the progress of anytime problem-solving, *Thirteenth National Conference on Artificial Intelligence*, 1229–1234, 1996.

[5] E. Horvitz, Reasoning under varying and uncertain resource constraints, *Seventh National Conference on Artificial Intelligence*, 111–116, 1988.

[6] E. Horvitz, Models of continual computation, *Fourteenth National Conference on Artificial Intelligence*, 286–293, 1997.

[7] J. Liu, K. Lin, W. Shih, A. Yu, J. Chung, and W. Zao, Algorithms for scheduling imprecise computations, *IEEE Transactions on Computers*, 24(5):58–68, 1991.

[8] A.-I. Mouaddib, *Contribution au raisonnement progressif et temps réel dans un univers multi-agents*, PhD thesis, University of Nancy I, (in French), 1993.

[9] A.-I. Mouaddib and S. Zilberstein, Handling duration uncertainty in meta-level control of progressive reasoning, *Fifteenth International Joint Conference on Artificial Intelligence*, 1201–1206, 1997.

[10] A.-I. Mouaddib and S. Zilberstein, Optimal scheduling of dynamic progressive processing, *Thirteenth Biennial European Conference on Artificial Intelligence*, 449–503, 1998.

[11] S. Russell and E. Wefald, *Do the Right Thing: Studies in Limited Rationality*, MIT Press, 1991.

[12] K.S. Jones and P. Willett (eds.), *Readings in Information Retrieval*, Morgan Kaufmann Publishers, 1997.

[13] S. Zilberstein and S. Russell, Optimal composition of real-time systems, *Artificial Intelligence* 82(1-2):181–213, 1996.