On the Application of Predictive Control Techniques for Adaptive Performance Management of Computing Systems

Sherif Abdelwahed, Senior Member, IEEE, Jia Bai, Rong Su, and Nagarajan Kandasamy Member, IEEE

Abstract—This paper addresses adaptive performance management of real-time computing systems. We consider a generic model-based predictive control approach that can be applied to a variety of computing applications in which the system performance must be tuned using a finite set of control inputs. The paper focuses on several key aspects affecting the application of this control technique to practical systems. In particular, we present techniques to enhance the speed of the control algorithm for real-time systems. Next we study the feasibility of the predictive control policy for a given system model and performance specification under uncertain operating conditions. The paper then introduces several measures to characterize the performance of the controller, and presents a generic tool for system modeling and automatic control synthesis. Finally, we present a case study involving a real-time computing system to demonstrate the applicability of the predictive control framework.

Index Terms—Self-management of computing systems, autonomic computing, model-based management techniques, power management of computing systems.

I. INTRODUCTION

COMPUTER systems hosting information technology applications vital to military command and control, commerce and banking, transportation, among others, must satisfy stringent performance or quality of service (QoS) requirements while operating in highly dynamic environments. For example, the workload to be processed may be time-varying, and hardware and software components may fail during system operation. To achieve QoS goals in such computing systems, numerous performance-related parameters must be continuously monitored, and if needed, optimized to respond rapidly to time-varying operating conditions. As these systems increase in size and complexity, manually tuning key operating parameters will become very difficult. Therefore, future systems must be largely autonomic or self-managing, requiring only high-level guidance from administrators, and maintain the specified QoS by adaptively tuning key operating parameters [1].

Advanced control and mathematical programming techniques offer a formal framework to design automated and efficient performance-control schemes for computing systems. If the system of interest is correctly modeled and the effects of its operating environment accurately estimated, control algorithms can be developed to achieve the desired performance objectives. The key advantages of using such a framework instead of ad hoc heuristics are: (1) various performance control problems of interest can be systematically posed within the same basic framework; (2) the management structure (control policy) can be automatically synthesized for a wide range of specifications, constraints, and operating conditions; and (3) the feasibility of the proposed algorithms with respect to the QoS goals can be verified prior to deployment.

The use of automatic control to achieve self-managing behavior in computing systems has attracted a great deal of interest in both academia and the computer industry. Research in real-time computing has focused on using PID control as the basis for performance management of single-CPU applications [2]. These reactive techniques observe the current application state and take corrective action to achieve a specified performance metric, and have been successfully applied to problems such as task scheduling [3]–[5], bandwidth allocation and QoS adaptation in web servers [6], load balancing in e-mail and file servers [7], [8], network flow control [9], [10], CPU power management [11], [12], as well as power management in virtualized servers [13]. Assuming a linear time-invariant system, an unconstrained state space, and a continuous input/output domain, a closed-loop feedback controller is designed under stability and sensitivity requirements.

Other researchers have studied the use of more advanced state-space and multi-input multi-output (MIMO) methods to manage computing systems [14]–[18]. These methods can take into account multi-objective cost functions and dynamic operating constraints while optimizing performance. In [17], for example, a MIMO controller manages the power consumed by a computer cluster by shifting power among the various servers with respect to their performance needs, thereby manipulating the cluster’s total power consumption to be lower than the specified power budget.

For more complex control problems, a pre-specified plan—
the feedback map—is inflexible and does not adapt well to constantly changing operating conditions (e.g., time-varying workload arrivals). Many computing systems also exhibit complex behavior comprising both continuous-flow and event-triggered dynamics. Moreover, the decision variables available to tune system behavior, i.e., the control-input set, is typically limited to a finite set of values. Since the underlying control set is finite, classical PID control techniques cannot be applied directly to such systems, and in most cases a closed expression for a feedback control map cannot be established. Finally, actions such as turning on servers have an associated dead time—the delay between a control input and the corresponding system response—which is usually on the order of minutes. This requires proactive control where control inputs must be provided in anticipation of future changes in operating conditions.

We consider in this paper, a predictive control approach to design self-managing computing systems. The underlying control policy selects actions by optimizing the system behavior, as forecast by a system model, for the specified QoS criteria over a limited prediction horizon. Both control objectives and operating constraints are represented explicitly in the optimization problem and solved for each time instant. This control method applies to various performance management problems, from those with simple linear dynamics to more complex ones, including systems with long delay or dead times, and those exhibiting non-linear and event-driven behavior. It can also accommodate changes to the behavioral model itself, caused by resource failures and/or parameter changes in time-varying systems. The proposed method is also amenable to feasibility analysis with respect to QoS guarantees.

We focus on the following aspects relevant to the application of the predictive control policy to practical computing systems.

- The number of possible control (or tuning) options directly affects the computational overhead of the controller—a major concern in real-time systems where the controller executes in the background at run time. We introduce several techniques to improve the performance of the control algorithm when the decision space is large.

- To characterize the accuracy of the proposed control scheme, we develop a feasibility test to provide an upper bound on the operating region of the controller and ensure bounded (or stable) system behavior. This test is applicable to situations in which the system is modeled as a hybrid system with a finite control set. The controller is termed feasible for a given set-point specification if it can drive the system in finite time from any initial state in a given operating region to a small neighborhood of the set-point and maintain the system there. We present a novel computational procedure based on nonlinear programming to automatically compute a minimal containable region in which each trajectory from inside cannot move out under the control policy. This work extends our earlier results [19] by considering the effects of stochastic environment inputs on feasibility analysis.

- We present a generic meta-modeling tool for building the system model, defining its requirement, and automatically synthesizing an appropriate predictive control structure.

- We develop a set of evaluation metrics to characterize the performance of the controller under uncertain operation conditions.

Finally, we demonstrate the applicability of predictive control using a case study of a real-time computing system. Assuming a processor capable of dynamic voltage scaling (DVS) where operating frequencies can be chosen from a finite set, we develop a controller to operate this processor under a time-varying workload while minimizing power consumption. Therefore, the controller must address the trade-offs between QoS requirements, given in terms of a desired response time for requests, and the corresponding power consumption.

The rest of this paper is organized as follows. Section II reviews the predictive control approach and presents its main components. Section III introduces techniques that reduce the computational complexity of the control algorithm. Section IV develops a technique to characterize controller feasibility and Section V discusses the various metrics used to characterize controller performance. Section VI presents a generic tool for system modeling and automatic control synthesis. Section VII describes the case study and we conclude the paper in Section VIII.

II. THE CONTROL-BASED PERFORMANCE MANAGEMENT STRUCTURE

Fig. 1 shows the key components of a self-managing computing system: 1) the system model and its QoS specification, 2) the environment-input forecaster, and 3) the controller.

A. Forecasting Environment Inputs

In computing systems operating in open and dynamic environments, the corresponding inputs to the controller are generated by external sources whose behavior typically cannot be controlled; for example, web-page requests made to a server by Internet clients. It has also been observed that most web and e-commerce workloads of interest show strong and pronounced time-of-day variations [20]–[22], and that key workload characteristics such as request arrival rates can change quite significantly and quickly—usually in the order of a few minutes. In most situations, however, such workload variations can be estimated effectively using well-known forecasting techniques such as the Box-Jenkins ARIMA modeling approach [23] and Kalman filters [24]. A forecasting model is typically obtained via analysis or simulation of relevant parameters of the underlying system environment and has the following form for a system input $\omega$:

$$\hat{\omega}(k) = \phi_k(\omega(k - 1,1)) \quad (1)$$

where $\hat{\omega}(k)$ denotes the estimated value and $\omega(k - 1,1)$ is the set of $r$ previously observed environment vectors $\{\omega(k - 1), \ldots, \omega(k - r - 1)\}$. The other estimation parameters—for instance, the covariance matrix in the Kalman filter—are assumed to be embedded in the model $\phi_k$. These parameters are typically obtained by training $\phi_k$ using test data representative of actual values observed in the field. We also assume that the estimation error is bounded and known with
a certain probability distribution. Therefore, we can write, \( \omega(k) = \phi_k(\omega(k-1, n)) + e(k) \) where \( e(k) \in E \) is a bounded random variable reflecting the effect of the estimation error.

From the implementation point of view, environment inputs are handled by a prediction module that continuously samples environment inputs and estimates their future values. The predictor module can be instantiated with different forecasting techniques and can be trained offline with representative input data. Signal and parameter estimators may also be added to extract information about, and build an accurate model of the operating environment.

### B. System Model, Performance Specification, and Constraints

As noted in the Introduction, the control approach proposed in this paper targets a general class of computing systems with finite control-input set. The following discrete-time equation describes the general dynamics of such a system:

\[
x(k+1) = f(x(k), u(k), \omega(k))
\]

where \( x(k) \in \mathbb{R}^n \) is the system state at time step \( k \), and \( u(k) \in U \subset \mathbb{R}^m \) and \( \omega(k) \in \Omega \subset \mathbb{R}^r \) denote the control inputs and environment parameters at time \( k \), respectively. The system model \( f \) captures the relationship between the observed system parameters, particularly those relevant to the QoS specifications, and the control inputs that adjust these parameters. This model could be in the form of difference equations for simple systems, and in the form of an approximation structure such as a neural network for more complex systems whose dynamics cannot be easily described from first principles.

Since the current value of \( \omega(k) \) cannot be measured until the next sampling instant, the system dynamics can only be captured using a model with uncertain parameters, as follows:

\[
\hat{x}(k+1) = f(x(k), u(k), \hat{\omega}(k))
\]

This estimated value of \( x \) is the one used by the controller to evaluate applicable control options. From the dynamic point of view, we can rewrite the system model as follows,

\[
x(k+1) = f(x(k), u(k), \phi_k(\omega(k-1, r)), e(k))
\]

In the above equation, \( e(k) \) is the only stochastic variable. In analyzing the feasibility of the predictive control approach, one needs to consider the value of \( e \) leading to the maximum deviation from the objective state. We will introduce an algorithm in Section IV to check the feasibility of the control approach for a bounded error domain \( E \).

**1) Performance Specifications:** Computing systems must achieve specific QoS goals while satisfying certain operating constraints. A basic control action in such systems is set-point regulation where key operating parameters must be maintained at a specified level or follow a certain trajectory. The controller, therefore, aims to drive the system to within a close neighborhood of the desired operating state \( x^* \in X \) in finite time and maintain the system there. A general form of such specification can be expressed using the cost function

\[
J(x, u) = \|x - x^*\|_P + \|u\|_Q + \|\Delta u\|_R
\]

where \( \| \cdot \|_A \) is a proper norm with weight \( A \). The above performance measure takes into account the cost of the control inputs themselves and their change. It is also possible to consider transient costs as part of the operating requirements, expressing the fact that certain trajectories towards the desired state are preferred over others in terms of their cost or utility to the system.

**2) Operating Constraints:** The system must also operate within strict constraints on both the system variables and control inputs. In general, such constraints can be expressed as a feasible domain for the composite space of a set of system variables, possibly including the control inputs themselves. Such operating constraints can be generally captured as as \( \psi(x) \leq 0 \) and \( U(x) \subseteq U \) where \( U(x) \) denotes the permissible input set in state \( x \) and \( \psi(x) \leq 0 \) defines reachable states.

To summarize, the primary objective of the controller is to drive the computing system to the desired state \( x^* \) while minimizing the control and transient costs in “reasonable” time using an admissible trajectory, defined by the constraints \( \psi(x) \leq 0 \) and \( U(x) \), and maintain it close to \( x^* \).

### C. Predictive Controller

The basic concept behind predictive control is to solve an optimization problem over a future time horizon, and then roll
this horizon forward at regular intervals, re-solving the control problem.

**Algorithm 1** The Predictive Control Algorithm: PControl(k)

```plaintext
input: \( x(k), \omega(k) \in \mathbb{R} \)
\( s_0 := x(k) \)
for \( i = 0 \) to \( N - 1 \) do
    \( \omega := \omega(k + i, r) \)
    \( s_{i+1} := \emptyset \)
    for all \( x \in s_i, u \in U(x) \) do
        \( \hat{x} := f(x, u, \omega) \)
        \( s_{i+1} := s_{i+1} \cup \{ \hat{x} \} \)
        Compute Cost(\( \hat{x} \)) based on \( J(\hat{x}, u) \).
    end for
end for
\( x_{\text{min}} := \arg\min \{ \text{Cost}(x) \mid x \in s_N \} \)
return \( u^*(k) := \text{initial input leading from } x(k) \text{ to } x_{\text{min}} \)
```

Algorithm 1 shows the details of the predictive control technique. At each time instant \( k \), it accepts the current operating state \( x(k) \), and starting from this state, the controller constructs a tree of all possible future states up to the specified prediction depth \( N \). The relevant parameters of the operating environment are first estimated. Then the next set of reachable system states (subject to both state and input constraints) are generated by applying all applicable control inputs from the set \( U(x) \). The total cost function corresponding to each estimated state is then computed based on the cost function \( J \). The state \( x_{\text{min}} \) with the minimum cost at the end of the tree is then selected and the first input leading to this state, \( u^*(k) \), is applied to the system while the rest are discarded. The above search is repeated each sampling step.

### III. STRATEGIES FOR COMPLEXITY REDUCTION

The optimization problem discussed in the previous section will show an exponential increase in worst-case complexity with an increasing number of control options and longer prediction horizons. If \( |U| \) denotes the size of the input set and \( N \) the prediction depth, then the number of explored states is \( O(|U|^N) \). This is not a major concern for systems having a small number of control inputs. However, with a large control-input set, the corresponding control overhead may be excessive for real-time performance. This section discusses several approaches that can help reduce the complexity of the control algorithm.

#### A. Enhanced Search Techniques

Since the control set is finite, the control problem is essentially a search problem over a tree structure. Many efficient algorithms have been developed within the AI community to deal with such search problems [25]. We now outline some algorithms that can be directly applied to the predictive control problem.

1. **A* Search:** This technique is one of the most widely-known form of best-first search and evaluates nodes by combining \( g(n) \), the cost to reach the node from the root, and \( h(n) \), the cost to get from the node to the goal. The algorithm complexity is determined by the choice of \( g(n) \) and \( h(n) \). A* is a complete search in the sense that it will always find the optimal solution if \( h(n) \) is an admissible heuristic, that is, it never overestimates the cost to reach the goal.

   To apply A* search to our control algorithm, we require an admissible heuristic function. Since computing the heuristic may increase the control overhead, a pre-computed heuristic-cost table is used. Based on the utility function \( J \), we can define a 3-dimensional heuristic table in which each cell \( \text{heuristic}(\omega, x, r) \) contains the estimated smallest accumulated cost value of a node with a system state of \( x \), environment input of \( \omega \), and step distance of \( r \). To enhance the search speed, \( \omega \) and \( x \) are translated into integer representatives. The requirement for admissible heuristic is met by using appropriate approximation; for instance, we will use a down-approximated valued for \( \omega \) to ensure that the cost is always overestimated. The estimation error is used to bound such approximation for consecutive environment inputs. Fig. 2 shows the steps needed to compute the heuristic table. For each combination \( (\omega, x, k) \), \( \omega \) and \( x \) are used to calculate \( \hat{x} \), the next system state corresponding to the control input \( u_i \in U(x) \). For the set of all \(|U(x)|\) we compute the node with the smallest cost and add that to the accumulator (initialized to 0). The corresponding next state and environment input are then used as arguments for the next iteration. The computation will iterate \( k \) times, and the end the accumulator will contain all the needed cells \( \text{heuristic}(\omega, x, k) \). These cells will then be used at run-time to get the heuristic value corresponding to a node. Note that this table is computed offline, and it is a hash table; so it does not affect the time complexity.

   The A* search can be converted to the simpler uniform-cost search by setting the heuristic function to a constant. In such a case, there is no need to compute the heuristic table. However, experiments, detailed in Section VII, show that the run-time performance of A* is typically better than that of the uniform search.

2. **Pruning Equivalent Nodes:** Pruning is a process of shrinking the search space by removing selected subspaces that do not contain the searched object. Pruning, in this case, does not affect the final choice. In the search tree of the control algorithm, similar system states may emerge. Since nodes at the same depth receive identical environment inputs, it follows that if nodes representing identical system states are at the same depth, their future evolutions will be the same. In this case, only the node with the smallest cost needs to be kept for further exploration, while the others can be pruned along
with their subtrees. Such pruning can be combined with other search methods by adding a step of checking and deleting the “equal” nodes in each level of a tree. Similar to A*, this search technique is complete.

3) Greedy Search: A greedy search algorithm makes the locally optimal choice at each stage with the hope that this choice will lead to the global optimum [26]. The algorithm will generally not find the best solution, but a feasible one, since it does not operate exhaustively on all the nodes. It may make commitments to certain choices too early during the search process, preventing it from finding the best solution. Nevertheless, it is useful for a wide range of problems, particularly when overhead reduction is essential. In many practical situations, this approach can also lead to a near-optimal solution.

Beam search [27] can be viewed as a generalized greedy algorithm. For a beam search of width \( k \), the search only keeps track of the \( k \) best candidates at each step, and generates descendants for each of them. The resulting set is then reduced again to the \( k \) best candidates. This process thus keeps track of the most promising alternatives to the current top-rated hypothesis and considers all of their successors at each step. Beam search uses breadth-first search to build its search tree but splits each level of the search tree into slices of at most \( k \) states, where \( k \) is called the beam width. The number of slices stored in memory is limited to one at each level. When beam search expands a level, it generates all successors of the states at the current level, sorts them in order of increasing values, splits them into slices of at most \( k \) states each, and then extends the beam by storing the first slices only. Beam search have linear time and space complexity. However, similar to greedy search, beam search is incomplete, so it does not guarantee the best solution.

B. Search Space Reduction

One simple approach to improving the efficiency of the search procedure is to identify and discard those branches that cannot improve progress towards the set point. Consider the system described in section II. Let \( \delta_{X'}(\delta_{X'}) \) be the maximal (minimal) single-step change to any state \( x \in X', X' \subseteq X \) under any input from \( U \) and environment input \( \omega \in \Omega \). Formally,

\[
\delta_{X'} = \max_{x \in X', u \in U, \omega \in \Omega} \| x - f(x, u, \omega) \|
\]

\[
\delta_{X'} = \min_{x \in X', u \in U, \omega \in \Omega} \| x - f(x, u, \omega) \|
\]

Computing \( \delta_{X'} \) and \( \delta_{X'} \) is simple if \( f \) is decomposable with respect to \( x \) and \( \omega \), i.e., if there exists two maps \( f' \) and \( f'' \) such that \( f(x, u, \omega) = f'(x, u) + f''(\omega, u) \). Given the finite nature of the set \( U \), the above optimization problem can be decoupled into a set of simpler single-variable optimization problems and can be efficiently solved for a wide class of systems using mathematical programming tools.

The parameters \( \delta_{X'} \) and \( \delta_{X'} \) can be used to prune the search tree by considering the prospects of each branch. We need to assume the following with respect to the domain \( X' \):

\[
(\forall x \in X')(\exists u \in U) \| f(x, u, w) - x^* \| \leq \| x - x^* \|
\]

The characterization of such a \( X' \) is discussed in more detail in Section IV. Now, without losing generality, consider \( x^* \) to be the origin. Let \( x \) be the current state and \( x_M^i \) be the \( i \)th state at some depth \( M < N \) within the search tree and let \( \text{Tree}(x_M^i) \) be the associated subtree within the search tree. Assuming \( \text{Tree}(x) \subseteq X' \), we have

\[
(\forall x' \in \text{Tree}(x_M^i)) (N - M)\delta_{X'} \geq \| x' \| - \| x_M^i \| \geq (N - M)\delta_{X'}
\]

Consequently, the search algorithm can safely stop further exploration from a given state \( x_M^i \) if there is no prospect of further reducing the distance to the set-point along any path starting from \( x_M^i \). That is, if there exists another explored state \( x'' \) satisfying

\[
\| x'' \| + (N - M)\delta_{X'} > \| x_M^i \| + (N - M)\delta_{X'}
\]

The above pruning technique is complete in the sense that it will provide the same result as the exhaustive search, assuming separable \( f \). If \( f \) is not separable then the pruning technique may not be complete. The efficiency of this approach clearly depends on the difference (\( \delta_{X'} - \delta_{X'} \)). Therefore, for a large operating space, it could be more efficient to partition this space into a set of smaller regions \( \{X'_i | i \in I \} \) and compute \( \delta_{X'_i}, \delta_{X'_i} \) for each region \( i \in I \). The value of \( \delta_{X'_i} \) and \( \delta_{X'_i} \) will then depend on the region \( X'_i \), and will generally be smaller than \( \delta_{X'}, \delta_{X'} \), thereby leading to better state pruning.

C. Approximation of the Control Input Domain

In certain situations, it is possible to approximate the control-input set as a continuous domain, particularly when the set of discrete inputs available to the controller is large. This allows the control problem to be solved using traditional optimization techniques [28]. Specifically, assuming the input domain \( U \) to be a bounded interval in \( \mathbb{R}^n \), the control problem can be represented as the following general-constraint optimization problem:

Minimize \( J(z(k + 1), u(k)) \)

Subject to \( z(k + 1) = h(z(k), u(k), e(k)), \psi(g(z(k), u(k))) \leq 0, u \in U, e \in E \)

In this formulation, \( z = [x \omega]^T \) is a vector comprising both the system state and the (estimated) environment inputs. The above optimization problem can be solved directly using either dynamic programming or the maximum principle techniques, which can be used to obtain a closed form of a feedback control function.

However, there are several limitations underlying the above-discussed technique. First, the objective function and the dynamic constraints must satisfy the necessary conditions for the existence of an optimal solution. Second, the transformation from the discrete to a continuous-input domain may result in inaccurate or suboptimal solutions, particularly if the original input set is small and non-uniform. There is also the problem of environment-input estimation errors that limit the accuracy of the solution for long lookahead horizons. Nevertheless, the continuous approximation of a discrete-input set can be effective for systems with a large and uniformly distributed (finite) input set where the effect of the environment input is relatively insignificant.
IV. FEASIBILITY ANALYSIS

Given the limited exploration nature of the online algorithm, it is important to obtain a measure of feasibility to determine if the online control can reach the desired region in a finite time. The online controller is said to be feasible for a given operation region $X$ and tolerance domain $D$ containing the set-point $x^*$ if it can drive the system from any initial state in $X$ to a neighborhood of the set-point that is contained in the tolerance domain and maintain the system within this neighborhood. Therefore the feasibility of the online control strategy can be tested, for a given $X$, $D$, and $x^*$, by computing a bounded neighborhood, $R$, of $x^*$ such that any trajectory inside it cannot move out under the given control policy. Clearly, the online control policy is feasible if $R \subseteq D$.

In this section we provide a brief overview of the feasibility check method based on nonlinear programming proposed in [29]. We also show how this method can be extended to systems with stochastic environment inputs as described in section II. First, consider a system without any stochastic environment-inputs as follows:

$$x(k + 1) = f(x(k), u(k))$$

where $u(k) \in U \subset \mathbb{R}^m$ and $f : \mathbb{R}^n \times U \to \mathbb{R}^n$ is differentiable over $x$ for each fixed $u \in U$. Considering only the set point specification, and without losing generality, we consider the origin as the set-point. Then, for each initial state $x_0 \in \mathbb{R}^n$ it is desirable to move the trajectory to a small closed neighborhood $D \subset \mathbb{R}^n$ of the origin, after a finite number of control steps, and then maintain the trajectory inside $D$. Here we will consider a one step lookahead control policy defined as follows: for each $x \in \mathbb{R}^n$ pick a $u^* \in U$ such that

$$u \in \underset{u \in U}{\arg\min} \|f(x, u)\|$$

where $\|\cdot\|$ means 2-norm. For each control action $u_i \in U$, let

$$W_i := \left\{ x \in \mathbb{R}^n \mid \|f(x, u_i)\| < \|x\| \right\}$$

That is, $W_i$ is the set of states at which the control action $u_i$ can bring the system closer to the origin. Let $Q := \bigcup_{i=1}^{n} W_i$. Intuitively, $Q$ is the set of states where there exists a control action that can bring the system closer to the origin (note that $Q$ is not a function of time). Let $\mathbb{R}^+$ be the set of all nonnegative real numbers. For an $r \in \mathbb{R}^+$ write $B(r)$ for the closed ball in $\mathbb{R}^n$ centered at the origin with the radius $r$, and $\partial B(r)$ for the boundary of $B(r)$, namely

$$\partial B(r) := \left\{ x \in B(r) \mid \|x\| = r \right\}$$

The operating region (state set) of a practical system is usually compact either for safety reasons or the saturation of physical components. Therefore, we will limit our analysis to a compact continuous operating region $X \subset \mathbb{R}^n$. Assume $X$ is defined as the solution set for a collection of known inequalities

$$\psi_i(x) \leq 0 \quad (i \in I)$$

where $I$ is a given index set and each $\psi_i$ is differentiable in $x$. Let $\overline{Q}$ be the complement of $Q$ with respect to $X$, i.e., $\overline{Q} := X - Q$. Since $Q$ is open and $X$ is compact, we get $\overline{Q}$ is compact. Let

$$r^* = \max_{x \in \overline{Q}} \|f(x, u^*)\| = \max_{x \in \overline{Q}} \min_{u \in U} \|f(x, u)\|$$

Since $\overline{Q} \subseteq X$ is compact, $f$ is differentiable (thus continuous) over $x$ for each fixed $u \in U$ and $U$ is finite, we have that $r^*$ is finite. By the definition of $\overline{Q}$,

$$\forall x \in \overline{Q} \forall u \in U \|x\| \leq \|f(x, u)\|$$

Particularly,

$$\forall x \in \overline{Q} \|x\| \leq \min_{u \in U} \|f(x, u)\| \leq \max_{x \in \overline{Q}} \min_{u \in U} \|f(x', u)\| = r^*$$

Thus, $\overline{Q} \subseteq B(r^*)$. But $B(r^*)$ may not be necessarily a subset of $X$. We have the following result.

**Proposition 4.1:** [29] If $B(r^*) \subseteq X$ then $B(r^*)$ is a containable region. Furthermore, if there exists another containable region $B(r)$ with $\overline{Q} \subseteq B(r)$, then $r^* \leq r$.

Computing $r^*$ a given above is a max-min problem, which can be converted to a non-linear programming (NLP) problem [30].

Now consider the system with stochastic input described in section II. With the assumption of bounded estimation error $e_\phi$ and bounded model error $e_f$, the system can be described by the model,

$$x(k + 1) = f(x(k), u(k), \omega(k), e_f(k))$$

$$\omega(k) = \phi(\omega(k-1), e_\phi(k))$$

Note that in the above equation we assume $\phi$ to be differentiable over every argument. For simplicity we assume the estimation parameters in the forecasting model to be constant or time invariant, i.e., $\phi$ is not periodically re-tuned. A reusable stable estimation model (such as Kalman filter) can usually lead to a better prediction. In many situations, such reusable models can be approximated by a time invariant model with a bounded error margin. In general the stability result in this section remains applicable as long as the model for the reusable estimator does not change the differentiability (within the interior of $Z \times E$) and continuity (within the closure of $Z \times E$) of function $h$ to be defined in the following paragraph.

We consider a simplified objective function $J$ given by $J = \|x - x^*\|.\overline{Q}$. The above problem can be converted to a simpler optimization problem using the state transformation, $z(k) = [x(k) \: \omega(k-1)]^T$. In this case the system is described by the following difference equation:

$$z(k + 1) = h(z(k), u(k), e(k))$$

where $z(k) \in Z \subset \mathbb{R}^{n+r}$ and $e(k) = [e_f(k) \: e_\phi(k)] = [e_1(k), \ldots, e_p(k)] \in \mathbb{R}^p$ is the composite error vector at time instant $k$ with each parameter $e_i(k)$ ($1 \leq i \leq p$) belonging to a finite closed interval $E_i = [e_i^l, e_i^u] \subset \mathbb{R}$. Suppose $Z = X \times \Omega$, which is the overall state set of the system composed of the operation region $X$ and the valid domain $\Omega$ of the environment input $\omega$, is defined as the solution set of a collection of known inequalities $\varphi_i(z) < 0$ ($i \in I$), where $I$ is a given index set and each $\varphi_i$ is differentiable over $z$. Since both $X$ and $\Omega$ are compact, the set $Z$ is also compact. The exact value of $e_i(k)$ at each instance $k$ is unknown. Let $E := \prod_{i=1}^{p} E_i$. The map $h : \mathbb{R}^{n+r} \times U \times \mathbb{R}^p \to \mathbb{R}^{n+r}$ is assumed differentiable on every interior point of $Z \times E$ and continuous on the closure of $Z \times E$ when $u$ is fixed. Again without loss of generality, suppose the origin is the desirable state. The control objective is to
compute \( \max_{u \in U} J_u(z, e, e_{u_1}, \ldots, e_{u_m}) = \|h(z, u, e)\|^2 \)

subject to \( e, e_{u_1}, \ldots, e_{u_m} \in E \)
\( (\forall u' \in U) \|h(z, u', e_{u'})\|^2 \geq \|z\|^2, \)
\( (\forall i \in I) \quad \varphi_i(z) \leq 0, \)

return \( r_* = \min_{u \in U} \{ J_u(z, e, e_{u_1}, \ldots, e_{u_m}) \} \)

Fig. 3. The non-linear programming algorithm for computing the radius of the containable region for a system with uncertain parameters.

minimize \( \|z\|^2 \). Then the one-step-lookahead control policy is defined as follows: for each \( z \in \mathbb{R}^{(n+r)} \) pick \( u^* \in U \) such that
\( u^* = \arg \min_{u \in U} \left[ \max_{e \in E} \|h(z, u, e)\| \right] \)

Similar to the previous development, for each \( u \in U \) let
\( W_u := \left\{ z \in Z \left| \max_{e \in E} \|h(z, u, e)\| < \|z\| \right. \right\} \)
and \( Q := U \cup W_u \). Let \( \overline{Q} \) be the complement of \( Q \) with respect to \( Z \), i.e. \( \overline{Q} := Z - Q \). Clearly \( \overline{Q} \) is compact. Let,
\[ r^* := \max_{z \in \overline{Q}} \min_{u \in U} \|h(z, u, e)\| \tag{4} \]

Since all related sets are compact and \( h \) is continuous, \( r^* \) is finite. Then we can have the following result extending Proposition 3.1.

**Proposition 4.2:** The closed ball \( B(r^*) \) is a containable region.

**Proof:** For each \( x \in B(r^*) \), if \( x \in Q \) then by definition of \( Q \) we have
\[ \|f(x, u^*, a)\| < \|x\| \leq r^* \]
If \( x \notin Q \) then \( x \in \overline{Q} \cap X \). By equation 4 we have
\[ \|f(x, u^*, a)\| \leq \min_{u \in U, a \in E} \max_{x' \in \overline{Q} \cap X} \|f(x', u', a')\| \leq \max_{x' \in \overline{Q} \cap X} \min_{u \in U, a \in E} \|f(x', u', a')\| = r^* \]
In either case we have \( f(x, u^*, a) \in B(r^*) \), as required. \( \square \)

Computing \( r^* \) as described above is rather complicated. To make it computationally simple, we compute an alternative value
\( r_* := \min_{u \in U} \max_{z \in \overline{Q}} \|h(z, u, e)\| \)

It is easy to see that \( r^* \leq r_* \). Clearly \( r_* \) is finite. Suppose \( U = \{u_1, u_2, \ldots, u_m\} \). Then computing \( r_* \) can be converted into the following nonlinear programming depicted in Figure 3, where the “subject to” conditions are used to specify that \( z \) takes a value from \( \overline{Q} = Z - Q \).

Note that \( \|\cdot\| \) is the 2-norm. As mentioned earlier, each \( \varphi_i \) is differentiable in \( z \). Also, for each fixed \( u \in U \), \( h(z, u, e) \) is differentiable in \( z \), so is \( \|h(z, u, e)\|^2 \). Thus the above NLP problem is well defined [31].

In some situations, the underlying system dynamics and constraints are simple enough to allow an analytic solution

1. Note that \( k \) is uncontrollable so, effectively, this reduces to minimizing \( \|x - x\|_Q \).

V. CHARACTERIZING CONTROLLER EFFECTIVENESS

The goal of the predictive control approach is to optimize the system performance based on a given utility function with respect to time-varying environment inputs. However, since the control set is finite and only a limited search is conducted, the controller can only achieve suboptimal performance. In general, the controlled system performance depends on several design related factors as well as the operating environment. The main design-related factors are:

- **Prediction horizon (\( T \))**: When future environment inputs are known in advance, or can be predicted perfectly, increasing the lookahead horizon will typically improve system performance. However, due to the stochastic nature of the environment inputs, the positive effects of increasing the prediction horizon on system utility will be countered by the gradual accumulation of prediction errors as the controller explores deeper into the horizon. The horizon after which any increase will not improve the controller performance, will be referred to as the limit horizon, can be estimated using simulation.

- **Control set (\( U \))**: In general, available control options in computing systems are finite. However, in certain situations, control options can be selected from a continuous domain, for example, the load share of a server. In such cases, a finite “discretized” approximation of the domain is used. The granularity of the approximated set is a design choice that will affect the performance of the controller in many aspects. In the case of a set-point specification, increasing the control set leads to a smaller containable region.

- **Sampling period (\( T \))**: The application of predictive control requires measurements sampling from the system as well as the environment inputs. The sampling period is typically constant and is chosen based on the frequency characteristics (speed) of the system and the operation environment, as well as available sensor technologies.
The controller effectiveness can be characterized by how close it can drive the system to the optimal region. It can also be characterized by the ability of controller to handle the operating environment variations as well as the computational overhead it requires as a part of the system. These aspects of the controller performance can be evaluated using the following measures.

1) Fitness: This measure characterizes the ability of the controller to achieve a suboptimal solution. For a given representative environment input trace, the controller fitness is defined as the ratio of the average utility achieved by the controller to the average optimal utility. We define the optimal utility as the one achieved over the whole operation cycle using a full knowledge of the environment inputs. We refer to the controller that can achieve optimal utility as the complete oracle controller. Clearly this controller is not realizable in practice and is used here as a reference for the ultimate performance level. The optimal utility can be computed offline for a given trace of environment inputs. Experiments shows that the fitness measure can be improved in most cases by increasing the prediction horizon (up to the limit horizon) and the number of control inputs, or by reducing the controller sampling time.

2) Robustness: This measure evaluates the variations in system utility, in response to variability in the environment inputs. Given the dependency of the controller on environment predictions, the controller performance can vary significantly depending on the prediction error. The robustness measure evaluates the degradation of the controller effectiveness with respect to increasing prediction error. For a given environment input trace, we define the associated controller robustness as the ratio of the standard deviation of the trace to the standard deviation of the system utility. Note that robustness is a relative measure. For example, if two controllers are used for the same workload trace and the robustness of one is 0.2 and the other is 0.4, it means that the the later controller is twice as effective in handling the variation in the environment input.

3) Computational Overhead: This performance factor targets relevant computational requirement of the controller at run-time. In general, these includes both memory and cpu utilization. However, dependent on the application, once factor can be more relevant than the other. For instance, in server management applications, cpu utilization is typically more important. The computational overhead of a predictive controller is directly dependent on prediction horizon, size of the control set, and the sampling time.

The above measures are applicable to various control technologies. They can be used to formulate a benchmark to evaluate the relative performance of different control approaches under the same operating conditions. For effective benchmarking, a suite of representative workload (environment inputs) will be used to compute the average performance value corresponding to each of the above measures.

VI. TOOL DEVELOPMENT

Since domain engineers may not have adequate expertise in system modeling and control design, we have developed a model-based design framework, allowing for the design of the predictive control structure discussed in Section II at a much higher level of abstraction. The Automatic Control Modeling Environment (ACME) has been developed using the Generic Modeling Environment (GME) [32], a meta-programmable toolset which allows for easy creation of domain-specific modeling languages and environments. The ACME meta-models capture the system design in a modular, component-based form easily accessible to the system designer. The meta-models generate models and specifications that can be used by domain engineers to synthesize the control structure. The synthesized structure can then automatically generate code for the controller implementation.

A. Overview of the ACME

The ACME comprises of three main models: 1) the architecture model, in which high-level system components and their interconnections are defined, 2) the data collection model, to obtain measurements from the physical system for the model variables, and 3) the system dynamics model, to capture the abstract system behavior, specifications, and operating constraints, as well as to estimate future variations in the environment inputs and tune system variables with respect to operational variations and constraints. We now describe the semantics of the key modeling concepts used in ACME.

The architecture meta-model shown in Fig. 1 captures the overall control structure at the highest level of abstraction, containing all the components needed for controller design as well as the connections between these components and the underlying connection ports. The UML notation for containment is a line connecting an object to its container, with a small black diamond on the “container” end of the line. The classes PhysicalSystem, SystemModel, Environment, Observer, and Controller are all key components contained in the main System class. These classes correspond to the “System”, “System Module”, “Environment Module”, “Observer”, and “Control Module” components, respectively, in Fig. 1. The details of each component are encapsulated within the underlying substructures which have their own internal descriptions. The connections in the architecture define data transportation between classes. As shown in Fig. 4, the System also contains a connection Controllable. The small black dot associates the connection with two endpoints ControlInput and Actuators, which act as ports of the high-level components, while the connection is directed from “src” to “dst”.

The data collection model contains the components required to compute and estimate the system variables required by the control structure. Some variables can be measured directly while others are unobservable. In some cases, however, unobservable system variables can be calculated based on records of measured variables using observers. For the predictive controller, future values of certain system variables must also be estimated. ACME distributes the data-collection task to three different models as follows.

In Fig. 4, sensors (embedded in the PhysicalSystem model) read in measurable data, including the environment inputs, observable system states, and the system output. Examples of measurable variables include response time, throughput, and CPU utilization. For unobservable data, the Observer model is used to collect related measurements and compute
an estimation of these variables. Third, an Estimator model uses a history of sampled data to estimate future environment variables. In our case, the Estimator realizes the forecasting model specified in Eq. 1.

The dynamics of the underlying computing system is captured by SystemModel which implements the discrete-time equation specified by Eq. 2. Finally, the basic control scheme detailed in Algorithm 1 can be parameterized within the Controller class by specifying the prediction horizon, the possible control input set, and a utility (cost) function. The predictive controller utilizes these specifications to manage the system at run-time by optimizing the underlying system utility within the constraints posed by operation requirements.

### B. The ACME Meta-Models

This section explains the conceptual aspects of the proposed performance control scheme by relating them to abstract syntactic elements of the ACME modeling language. The Estimator model discussed in the previous section is added to the overall system architecture as a high-level component, parameterized, and connected to other model blocks via their available ports. Our discussion is guided by meta-model screenshots, which are expressed with a stereotyped UML class-diagram notation. The stereotypes («Model», «Atom», «Connection», etc.) express the abstract syntax implemented by the GME environment. The details of these syntactic constructs supported by the GME environment have been explained in [32], [33]. The sub-languages that together constitute the ACME language have been elaborated below.

To enhance readability, the following font-based notations are adopted: “class” used for the classes of the meta-model, and “components” used for all the other components such as connections and attributes.

The basic data types used in the meta-model are first defined within a class paradigm, while data used in all the other places are proxies of the data in the paradigm. The following three classes are used to get the values for the data proxies. The Environment class represents the operating environment that the physical plant interacts with. The Estimator class can be selected from a library that includes different estimators like ARMA filters and Kalman filters. For example, if we use an ARMA filter to estimate the system input \( \tilde{\omega}(k) \) in Eq. 1, then, given \( r \) previously observed system inputs \( \omega(k-1), \omega(k-2), \ldots, \omega(k-r) \) and their mean \( \bar{\omega} \), the estimated value for \( k \) is

\[
\tilde{\omega}(k) = (1 - \sum_{i=1}^{r} \beta_i) \bar{\omega} + \sum_{i=1}^{r} \beta_i \omega(k-i) \quad (5)
\]

where the gain \( \beta \) determines how the estimator tracks the variations of the past observations and \( 0 \leq \sum_{i=1}^{r} \beta_i \leq 1 \). Similarly, the Observer class estimates unobservable system states using measurable variables and parameters if the underlying functions are available.

The Controller class shown in Fig. 5 parameterizes the predictive controller detailed in Algorithm 1. The meta-model allows the designer to choose from the three different search heuristics—A*, pruning, or greedy search—discussed in Section III-A using the SearchMode attribute and Horizon specifies \( N \), the length of the prediction horizon. The Utility class has three important attributes: Operation decides whether to minimize (maximize) the utility function, Constraints specifies the operating constraints the system must follow, and UtilityFunction specifies the utility function \( J \) in Eq. 3. The ControlInputSet class contains the available control inputs \( U \) for the system and SetPoint is the target value \( x^* \) that the controller aims to reach.

In Fig. 6, the SystemModel (and PhysicalSystem_sim) class realizes the state-evolution equation (Eq. 2) using one of various available techniques; for example, hybrid automata [34], [35], mathematical functions, or simple lookup tables. The general forms of HybridAutomata notation and Function notation are defined in the meta-model. For example, HybridAutomata contains State classes, including one InitialState...
in each HybridAutomata, and StateTransition connections between them. The state transitions can be addressed in the attribute HA_scripts of the HybridAutomata, or modeled inside it by choosing one of the HA_expression attributes, “Using scripts” or “Embed HA inside”. Both the Function-Expression attribute of the State class and the Expression attribute of the Function class capture mathematical relations, specifically the discrete-time equations of the system dynamics. The ValidCtrlInputs checks the validity of the control inputs sent by the controller corresponding to current system states. For example, if there are two States: Idle and Active, the ValidCtrlInputs should also have two ValidSets, IdleSet and ActiveSet correspondingly. Assume that the system is in the Idle State, then if a control input is not in the IdleSet, it is considered invalid; otherwise it is valid.

The PhysicalSystem_sim class is used to simulate the behavior of the computing system. Similar to SystemModel, PhysicalSystem_sim has HybridAutomata and Function. It also has Actuator and Sensor classes corresponding to the same elements as in real physical systems. Finally, the PhysicalSystem class, working in a real-time application mode, contains Actuator and Sensor classes. Sensor receives system states from the physical plant and Actuator sends control inputs selected by Controller to the plant.

VII. CASE STUDY: POWER MANAGEMENT

To demonstrate the predictive control approach and the underlying application aspects, we present a case study of power management of a DVS-capable processor operating under a time-varying workload. Power has become an important design constraint for densely packed processor clusters due to electricity costs and heat dissipation issues [36]. To tackle this problem, many modern processors allow their operating frequency and supply voltage to be dynamically scaled. For example, processors such as the AMD-K-2 [37] and Pentium M processors [38] offer a limited number of discrete frequency settings, eight and six, respectively. We apply the predictive control approach to manage the power consumed by such a processor under a time-varying workload comprising HTTP requests. Assuming a processor with multiple operating frequencies, the controller is required to achieve a specified response time for these requests while minimizing the corresponding power consumption.

1) System Model: We use a queuing model to capture the dynamics of a processor $P$ where $\lambda(k) \in \Lambda \subset \mathbb{R}$ and $\mu(k) \in \Upsilon \subset \mathbb{R}$ denote the arrival and processing rates, respectively, of incoming requests, where $\Lambda$ and $\Upsilon$ are bounded, and $q(k)$ denotes the queue size at time $k$. Each $P$ operates within a limited set of frequencies (control inputs) $U$. Therefore, if the time required to process a request while operating at the maximum frequency $u_{\text{max}}$ is $c(k)$, then the corresponding processing time while operating at some frequency $u(k) \in U$ is $c(k)/y(k)$ where $y(k) = u(k)/u_{\text{max}}$ is the scaling factor. The average response time achieved by $P$ during time step $k$ is denoted by $\tau(k)$; this includes both the waiting time in the queue and the processing time on $P$. We use the model proposed in [39] to estimate the average power consumed by $P$ while operating at $u(k)$ as $y^2(k)$. The following equations describe the dynamics of a processor:

$$q(k+1) = q(k) + \left(\hat{\lambda}(k) - \frac{y(k)}{c(k)}\right) \cdot T$$  \hspace{1cm} (6)

$$\hat{\tau}(k+1) = (1 + q(k)) \cdot \frac{c(k)}{y(k)}$$  \hspace{1cm} (7)

Given the observed queue length $q(k)$ on $P$, the estimated length $\hat{q}(k+1)$ for time $k+1$ is obtained using the predicted workload arrival and processing rates. The average response time of requests arriving during the time interval $[k, k+1)$ is estimated as $\hat{\tau}(k+1)$. The controller sampling time is denoted by $T$. Both $\hat{q}(k+1)$ and $\hat{\tau}(k+1)$ depend on the estimated processing time $c(k)$. The above equations adequately model the system dynamics when the processor is the bottleneck resource; for example, web servers where both the application and data can be fully cached in memory, thereby minimizing (or eliminating) hard disk accesses.

We use an ARIMA model [40] to estimate the arrival rate $\lambda$ at the processor. The state-space form of this model is implemented via a Kalman filter to provide load estimates to the controller. The processing-time estimate for time $k+1$ on $P$ is given by an exponentially-weighted moving-average model as $\hat{c}(k+1) = \pi \cdot c(k) + (1 - \pi) \cdot \overline{c}$ where $\pi$ is the smoothing constant.

2) The Control Problem: The overall controller schematic is shown in Fig. 7. Let $J$ be the cost function to be optimized at time $k$. Here, $J$ is determined by the achieved response time $\tau(k)$ and the corresponding power consumption. The estimated environment parameters $\hat{\omega}(k)$ include the arrival rate $\hat{\lambda}(k)$ and processing time $\hat{c}(k)$. Then, $J(\tau(k+1), u(k)) = \alpha_1 \cdot (\tau(k+1) - \tau^*)^2 + \alpha_2 \cdot (u(k)/u_{\text{max}})^2$ where $\alpha_1$ and $\alpha_2$ denote user-defined weights, and $\tau^*$ denotes the desired average response time.

The controller uses an exhaustive search strategy where a tree of all possible future states is generated from the current state up to the specified depth $N$. If $|U|$ denotes the size of the control-input set, then the number of explored states is $\sum_{q=1}^{N} |U|^q$. When both the prediction horizon and the number of control inputs are small, the control overhead is negligible, as confirmed by our simulations.

3) Experiment Settings: The performance of the controller is now evaluated using a representative web workload. We first describe how the workload is generated and then discuss the obtained results.

---

**Fig. 7.** The controlled server system.
We assume a processor with possible operating frequencies of 400, 600, 800, 1000, 1200, 1400, 1600, 1800, and 2000 MHz. As input to the processor, we generated a synthetic time-varying workload using trace files of HTTP requests made to one computer at an Internet service provider in the Washington DC area [21]. Fig. 8 (top) shows this synthetic workload where request arrivals are plotted at 30-second granularity. Next, we generated a virtual store comprising 10,000 objects, and the time needed to process an object request was randomly chosen from a uniform distribution between (25, 50) ms. The distribution of individual requests within the arrival sequence was determined using two key characteristics of most web workloads:

- **Popularity:** It has been observed that a few files are extremely popular while many others are rarely requested, and that the popularity distribution commonly follows Zipf’s law [21]. Therefore, we partitioned the virtual store in two—a “popular” set with 1000 objects receiving 90% of all requests, and a “rare” set containing the remaining objects in the store receiving only 10% of requests.

- **Temporal locality:** This is the likelihood that once an object is requested, it will be requested again in the near future. In many web workloads, temporal locality follows a lognormal distribution [41].

Parameters of the Kalman filter were first tuned using an initial portion of the workload—the first 50 time steps—and then used to forecast the remainder of the load during controller execution. Once trained, the filter provides effective estimates; the average error between the predicted and actual values is about 5% for the one-step-ahead estimate. Request processing times were estimated using an EWMA filter with a smoothing factor of $\pi = 0.1$. The sampling period of the controller was set to $T = 30$ secs—same as the prediction granularity.

4) **Performance Evaluation:** Fig. 8 summarizes the performance of the controller for a prediction horizon of $N = 3$ and a target response time of $\tau^* = 3$ seconds. The weights $\alpha_1$ and $\alpha_2$ were set to 10 and 1, respectively, in the cost function $J$. (Achieving the desired response time was given a higher priority than power consumption.) Fig. 8 shows the normalized frequency $u(k)/u_{\text{max}}$, as decided by the controller, and the achieved response time. Note that the controller does not achieve the desired QoS during very brief time periods since it cannot predict sudden (and short-term) spikes in the arrival rate. The frequent switching between operating frequencies occurs since the cost function $J(k)$ does not include a corresponding switching penalty. Though control actions in general systems typically incur some penalty, in the specific case of power management, this penalty is negligible, and therefore, was ignored in the cost function.

Another series of experiments was performed to evaluate the effect of different prediction horizons on controller performance. As before, the target response time was set to $\tau^* = 3$ seconds. Table I summarizes results for four values of the prediction horizon $N$ using the workload in Fig. 8, including the cost incurred by the controller per sampling time step (workload arrivals span 5000 time steps), robustness, and the execution time. Control performance, in terms of the cost incurred per step, improves with increasing the prediction horizon up to $N = 3$. For longer prediction horizons, one expects estimation errors to come into play. So, we conclude that for this workload, a prediction horizon of $N = 3$ gives best performance while incurring low execution-time overhead; for a sampling time of $T = 30$ seconds, the corresponding control overhead for $N = 3$ is only 0.5%. Also, we note that increasing the prediction horizon has marginal effect on the robustness factor.

Finally, in an uncertain operating environment, controller decisions cannot be shown to be optimal since the controller does not have perfect knowledge of future environment inputs, and control decisions are made using a limited prediction horizon. Therefore, we must be satisfied with good sub-optimal decisions. Our final series of tests were aimed at comparing the practical controller implementation against an “oracle” that has perfect knowledge of future workload arrivals. Here, we use the workload in Fig. 8 and set the average processing time per request to a constant value of 45 milliseconds. Table II shows that the performance of the oracle controller, again in terms of the average cost/step, improves monotonically with increasing prediction horizon, whereas that of a controller using predicted values for the request-arrival rate starts deteriorating slightly for $N = 3$ and $N = 4$. The improved performance of the oracle should be expected, considering that the error in the Kalman estimates, for the workload, starts at 5% and increases 1% for each prediction step thereafter.

5) **Impact of Efficient Search Techniques:** We applied the search techniques discussed in Section II. We first considered

<table>
<thead>
<tr>
<th>$N$</th>
<th>Cost/step</th>
<th>Robustness</th>
<th>Execution time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.11</td>
<td>81.12</td>
<td>0.003</td>
</tr>
<tr>
<td>2</td>
<td>0.12</td>
<td>80.5</td>
<td>0.01</td>
</tr>
<tr>
<td>3</td>
<td>0.12</td>
<td>80.3</td>
<td>0.15</td>
</tr>
<tr>
<td>4</td>
<td>0.13</td>
<td>80.3</td>
<td>1.8</td>
</tr>
</tbody>
</table>
the uniform-cost approach. In this approach, all the extended nodes are added to a priority queue, while each queue component is composed of information of the extended node, including its accumulative cost, its depth in the tree, its system states and the first control input along the path to the node. The queue is sorted according to the component costs in an ascending order.

We also considered the general A* search. The A* search extends the uniform-cost search with the heuristic table $heuristic(\omega, x, k)$. In this case, workload to the system is the environment input, which ranges in $[0, 1000]$. The environment input domain is approximated by a uniform set of discrete values $[0, 50, 100, \ldots, 1000]$. Similar approximation is used for the state variables. Accordingly, a $21 \times 51 \times N$ heuristic table is built.

We combined the pruning method with the uniform-cost search. Among all the systems states in the case study, only the queue level is used to calculate next system states, so we only compare the queue levels. As the queue level is bounded; it cannot be smaller than zero or larger than the maximum size, so it is likely to result in a same queue level. The pruning search then decides which subtree should be removed from the current tree structure.

Table III summarizes results for four prediction horizons $N$ using different search algorithms. The table shows the number of nodes extended and time spent by the controller per sampling time step for the first three methods. We see that for larger $N$ values, the efficient search techniques tend to performance significantly better than the comprehensive search. In this case study, the proposed efficient search strategies have shown great potential for improving the performance of the predictive controller, both in terms of time and space requirements. The first three strategies bound the search space to a smaller region, with the primary consideration of guaranteeing the optimality and completeness of the search, while the last strategy always cuts down a fixed amount from the search regions, providing a sub-optimal solution through a faster search. Still, the new strategies need extra time and space to process additional steps but these expenses are smaller than the time and space needed to explore the reduced search space.

6) ACM Model: We utilized the ACME environment to synthesize the control structure for the power management case study using the models generated by the ACME meta-models. For each simulation step, two environment variables are generated in the EnvironmentSim. One is the request arrival rate obtained from a local file using a DataReader model; the other is the execution time of the requests, set to 6.0ms in the Generator model. The future values of the variables are estimated by the ARMA filter in (5) and sent to the SystemModel to forecast two system states, queue level and dropped requests, over the given horizon of the controller Optimizer. The queue is updated by (6) and implemented as a buffer to store incoming requests with a limited size, so the dropped requests represent the signals dropped when the queue is full. By selecting the control input, the best CPU processing frequency, the Optimizer balances the forecast queue level, dropped requests, and the frequency. Finally, PhysicalSystemSim computes the new system states by equation 2 using the selected control input and new environment variables. To study the system, the domain engineers simply need to change the values of the attributes in the models, e.g., the prediction horizon of the controller, and the ACME will generate different data files for comparison and analysis.

7) Feasibility Analysis: For feasibility analysis, we only need to consider the queue size on the processor and the control input as the other two variables – response time and consumed power – are directly dependent on them. Here, we treat the arrival rate $\lambda(k) \in \Lambda$ and estimation error $e \in E$ as random inputs to the controller. Therefore, for a one-step lookahead controller, the optimization problem can be posed as

$$\text{Minimize} \quad (u'(k))^2 + \max_{\lambda(k) \in \Lambda, e \in E} (q'(k + 1))^2$$

Subject to $q'(k + 1) = q'(k) + (\lambda(k) + e - \beta \cdot u'(k)) \cdot T_i$

where $q'$ and $u'$ are appropriately scaled versions of the variables $q$ and $u$, respectively, and $Q', U'$ the associated domain set. The above minimization problem determines the optimal input $u^*(k)$ at each time instant $k$ when the current queue level $q'(k)$ is given. Thus, we have the following dynamic equation relating the predicted queue level $q'(k + 1)$ and the associated control input $u'(k)$:

$$\begin{bmatrix} q'(k + 1) \\ u'(k) \end{bmatrix} = \begin{bmatrix} f(q'(k), u^*(k), \lambda(k), e) \\ q'(k) + (\lambda(k) + e - \beta \cdot u^*(k)) \cdot T_i \end{bmatrix}$$

We now consider the containable region of the trajectory of $[q'(k + 1), u'(k)]^T$ ($k \in \mathbb{R}$) under the optimal control input $u^*(k)$ with the uncertain parameters $\lambda(k)$ and $e$. Applying the feasibility analysis results presented in Section 4, we can derive the containable region, where $[0, 0]^T$ is the desired point. Therefore, a state $[q'(k), u'(k-1)]^T$ is in the containable region iff $\forall u \in U'$

$$\left( \exists e \in E, \lambda \in \Lambda \mid ||f(q'(k), u, \lambda, e)|| \geq ||q'(k), u'(k-1)|| \right)$$

(8)

Since queue sizes are non-negative real values, we can provide an approximation of the containable region, defined by the following inequality:

$$\left( \forall u' \in U' \forall e \in E, \lambda \in \Lambda \mid q'(k + 1) \geq q'(k) \right)$$

(9)

From inequality (9) we obtain $\forall u' \in U'$,

$$\left( \exists e \in E, \lambda \in \Lambda \mid ||[q'(k + 1), u']^T|| \geq ||[q'(k), u'(k-1)]^T|| \geq ||[q'(k), u'(k-1)]^T|| \right)$$

(10)
Thus, by inequality (8) we know that \([q'(k), u'(k-1)]^T\) is in the containable region and (9) is a sufficient (though not necessary) condition for \([q'(k), u'(k-1)]^T\) to be in this region. In fact, the above stability condition is rather conservative in that the resulting solution set of (9) is independent of \(q'(k)\) due to the linear dynamics of \(q'(k+1)\). It turns out that, if the following inequality holds:

\[
    u_{\text{max}} \leq \max_{k \in \Lambda, e \in E} \frac{1}{\beta} (\lambda + e)
\]

then the containable region essentially contains all states \([q'(k+1), u'(k)]^T\). Thus, to make the problem of determining the minimum containable region nontrivial, we require that some states not be included within the region. Equivalently, we require that the minimum of the quantity \((u'(k))^2 + \max_{e \in E, \Lambda(k)} \Lambda q'(k+1))^2\) be reduced for at least some period. Then (11) must not hold, and it is easy to see that this condition ensures bounded controller behavior over any operating period. For example, in our case study, \(\beta = 0.53\) for an average processing time of 0.037 seconds per request (recall that the request processing time was chosen from a uniform distribution between 25, 50 milliseconds). We also have \(\max_k \lambda(k) = 732\) and the maximum estimation error is 70.8. Accordingly, \(u_{\text{max}}\) should be greater than 1514.7 MHz, which is clearly satisfied by the processor in question. In fact, based on the above condition, we can say that system performance is bounded for any arrival-rate pattern satisfying \(\max_k \lambda(k) < \beta u_{\text{max}} - \max(E)\) where \(u_{\text{max}} = 2000\) MHz.

Note that the above condition is highly conservative as it does not consider the dynamics of the environment input. A better estimate of the containable region can be obtained by considering the obvious sinusoidal pattern of the request arrival. However, the conservative approach discussed above is sufficient for this case study.

### VIII. CONCLUSIONS

We have addressed the problem of performance management of computing systems using a predictive control policy. Since, in most real-time computing systems, the execution time available for the controller is often limited by hard bounds, it is necessary to minimize the control overhead and consider the possibility that we may have to deal with suboptimal solutions. For control purposes, however, it is not critical to find the global optimum to ensure system stability; a feasible suboptimal solution will suffice. From the implementation point of view, it is important to characterize the controller performance with respect to design parameter and provided tools to facilitate the development and deployment of the control structure.

We described techniques to improve the controller performance, including efficient searching, reducing the search space, and approximating the control input domain. To characterize the accuracy of the predictive control approach, we also developed a feasibility test that provides an upper bound on its operating region. Finally, we applied the LLC scheme to a case study involving processor power management. Using realistic workload scenarios, we analyzed the impact of key controller parameters such as the prediction horizon on the system performance.

In future work, we plan to extend this approach to distributed multi-tier computing system and extend the tools and stability analysis techniques to handle these class of systems. For scalable control of large computing systems, one can extend the concepts developed in this paper within a decentralized framework wherein the overall performance-management problem is decomposed into smaller subproblems that individual controllers solve cooperatively. A second avenue of future research is to develop fault-adaptive control for computing systems wherein long-running software components managed by the controller can be treated as adaptive processes that exhibit slow behavioral changes over time as a result of both internal and external influences. This requires research into online model adaptation and refinement.

### REFERENCES
