ELECTRICAL ENGINEERING

A MODEL INTEGRATED FRAMEWORK FOR DESIGNING AND OPTIMIZATION OF SELF-MANAGING COMPUTING SYSTEMS

JIA BAI

Thesis under the direction of Professor Sherif Abdelwahed

This thesis addresses the problem of managing computing systems using an integration of model-based control techniques and efficient AI search strategies. The proposed control approach uses the system model to forecast all future system behavior up to a certain horizon and then searches for the best path for the system based on a given utility function. In practical computing systems, however, the large number of control (tuning) options directly affects the computational overhead of the control module which executes in the background at run-time, and ultimately slows down the overall system. To handle this problem, several search algorithms are introduced to improve the controller's performance.

This thesis also presents a model integrated framework, referred to as the Automatic Control Modeling Environment (ACME), to facilitate the use of control-based technology for self-management in computation systems. Control-theoretic concepts like above have been investigated and applied successfully to automate the management of computation systems of the

control technology. ACME is a domain-specific graphical modeling environment with automated synthesis tools. The framework allows domain engineers to develop models for general computation systems and to capture their performance requirements and operational constraints. The framework can automatically generates executable codes for the controllers based on the given system model and specifications.

A case study of an online processor power management is used to demonstrate the effectiveness of the new search techniques for the model-based control approach as well as the application of the ACME.

Approved _____ Date _____

A MODEL INTEGRATED FRAMEWORK FOR DESIGNING AND OPTIMIZATION OF SELF-MANAGING COMPUTING SYSTEMS

By

Jia Bai

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

August, 2008

Nashville, Tennessee

Approved:

Professor Gabor Karsai

Professor Sherif Abdelwahed

ACKNOWLEDGEMENTS

The work contained in this thesis could not have been accomplished without the help and support of numerous individuals. First and foremost, my advisor, Professor Sherif Abdelwahed, has been an invaluable resource. I offer Professor Abdelwahed special thanks for not only giving me sage advice, but for leading me to the way of scientific research.

I would also like to thank Professor Gabor Karsai, our institutes engineer Di Yao, and my fellow graduate students, Furui Wang, Tripti Saxena, Liang Dai, Abhishek Dubey, Aparna Barve and Jonathan Wellons. Many technical revisions were made through our discussions, and I genuinely appreciate your friendship. Thanks to those all who have kept my spirits high while I was completing this research.

Most importantly, I would like to thank my parents for their love and support from the other side of the Pacific ocean. All my successes are due to the opportunities you have provided me, and I am forever grateful.

This thesis was supported in part through a grant from the NSF SOD program, contract number: CNS-0804230

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

There has been an exponential increase in the complexity of computer systems in recent times. These systems often host information technology applications vital to transportation, online banking and shopping, military command and control, among others. In addition to the increasing complexity, such systems also need to satisfy stringent performance requirements, such as service delay bounded by a relatively small constant. Moreover, these systems operate in highly dynamic environments, where the workload to be processed may be time varying and hardware or software components may fail or degrade during system operations. In order to achieve the performance requirements while operating in such dynamic environments, numerous performance-related parameters must be continuously monitored, and if needed, optimized to respond rapidly to time-varying operating conditions.

The advent of corporate computer systems, where a combination of different technologies like networks is used has made the conventional, manual management of computing systems very difficult, time-consuming, and error-prone. There is an increasing need for these systems to possess autonomic or self managing execution environment, thus, requiring minimal human intervention. In such an autonomic managing environment, the systems will receive high-level objectives from human administrators [49] and maintain the specified requirements by adaptively tuning key operating parameters [35].

Control-theoretic strategies have been recently applied successfully to the design and verification of various adaptive resource management schemes in computation systems. This approach offers some important advantages over rule-based policies for performance management in that a generic control framework can address a variety of problems, such as power management, resource allocation and provisioning, by using the same basic control concepts. If system dynamics are precisely modeled and the changing environmental parameters are accurately estimated, the appropriate run-time control algorithms can be effectively developed to realize system self-regulation and achieve desired QoS objectives. Moreover, the feasibility or stability of the proposed control scheme with respect to the performance metrics can be verified prior to actual deployment. Examples of control-based resource management strategies include task scheduling [62, 86], QoS guarantees in web servers [61], resource allocation control [38, 77], network flow control [48], and power management [9].

A generic model-based control has also been designed [3, 43, 44] to address self-managing problems in computing systems. A switching hybrid-system model, previously introduced in [2], is adopted to capture the dynamics of systems having a finite control-input set. Using this system model, a limited look-ahead control (LLC) technique is developed where control actions are obtained by optimizing system behavior, as forecast by a mathematical model, for the specified performance criteria over a limited look-ahead prediction horizon. Both the control objectives and operating constraints are represented explicitly in the multi-objective optimization problem and solved for each time instant. The optimization problem is to optimize a

multi-variable objective function specifying the trade-offs between achieving the desired requirements and the corresponding cost incurred in terms of resource usage. For example, a controller may be required to meet a certain response time for a time-varying workload while minimizing system power consumption. This method can be applied to a variety of performance management problems, from systems with simple dynamics to more complex systems exhibiting non-linear behavior or ones with long delay or dead times. This method can also accommodate changes to the behavioral model itself caused by resource failures and/or parameter changes in time-varying systems.

The LLC approach to system control incurs a number of interesting challenges out of which we focus on two main challenges: the first challenge is the computational complexity of executing the model-predictive task online. At each time step, the controller needs to explore a limited lookahead region of the system state-space to select the best control action. If the search space is very large and the search algorithm is not time and space efficient, it will take too much time to achieve the best control action. Apart from this, the domain engineers who develop distributed real-time and embedded systems may not have the background to apply and implement the LLC method, so the seconde challenge is the complexity of the LLC approach itself.

To address the second challenge, in [70], a domain-specific modeling framework, referred to as the Dynamic QoS Modeling Environment (DQME), has been developed to achieve end-to-end Quality of Service (QoS) management in computing systems using the LLC approach. They proposed the use of model-based techniques to raise the abstraction of control theoretic

techniques and make them available to domain engineers. Also a middleware QoS-control architecture called ControlWare is proposed in [93] to facilitate the use of linear feedback control theory. It is motivated by the needs of performance-assured Internet services operating in highly uncertain environments or when accurate system load and resource models are not available. While the DQME and the middleware QoS-control architecture are applicable for certain systems, the limited lookahead control or the feedback control may not be applicable for other computation systems.

In this thesis, we address the complexity of the LLC algorithm, by considering several efficient search methods that can significantly reduce the size of the search space. The implemented search algorithms are shown to improve the controller performance by reducing its overhead while maintaining the system at or close to the optimal point.

We then present a generic model-based design framework that facilitates the design of general control-based adaptation components for a general class of computational systems. The framework is developed based on the LLC approach, and extended to more general techniques. It includes a control library from which a controller can be selected and parameterized for a given system and operation settings. The framework allows the user to develop formal models capturing relevant aspects of the system behavior as well as its performance specifications. Further, we have developed a model interpreter that auto-generates executable codes from the models for an appropriate control module. This framework is referred to as the Automatic Control Modeling Environment (ACME). The framework is based on the Generic Modeling Environment (GME) [52], a meta-programmable toolkit, which

allows for easy creation of domain specific modeling languages and environments. The ACME framework allows the design and specification of general control-based QoS adaptation policies.

The new search algorithms and the ACME framework is demonstrated on a case study of a processor capable of dynamic voltage scaling (DVS) where operating frequencies can be chosen from a finite set. We use the LLC approach in the ACME framework. The management problem is to maximize the processor utilization, which is a trade-off between processing speed and power consumption, under a time-varying workload.

Chapter 2 of the thesis presents a brief background of control applications. Chapter 3 reviews the LLC concepts. Chapter 4 describes in detail the different search approaches. Chapter 5 introduces the key concepts of the ACME language. Chapter 6 describes the implementation of a DVS-capable processor case study and provides test results of performances. Finally conclusion and future research are discussed in the final chapter.

Throughout this thesis, certain typographical conventions are used to distinguish technical terms. Table I.1 summarizes the conventions.

Table I.1: Table of Notation

Symbol	**Explanation**
k	time step/index/instance
U	control input set
$\|y\|$	size of a set y
X	system state space
$\hat{y}$	estimation value of a variable y
$\underline{\mathrm{y}}(k, m)$	the set of m previously observed vectors $y(k)$, ..., $y(k-m)$
ϕ	training model for estimators
$J()$	cost function
$D(y, y^*)$	distance map defining how close y is to y^*
$\|\|.\|\|$	proper norm
$Cost(y)$	accumulative distance cost from root to node y in tree structures
T	sampling time
R	robustness measure
$\sigma(y)$	standard deviation of a variable y
$O(.)$	asymptotic upper bound
$g(n)$	cost of reaching the node n from the root in a tree structure
$h(n)$	heuristic cost of node n in a tree structure
$\bar{y}$	the mean of a variable y
λ	arrival rate of environment inputs
y_{max}	maximum value of a variable y

CHAPTER II

BACKGROUND

This chapter provides background on control theory and relates its underlying concepts to the performance management problem.

While performance measurement is the process of assessing progress toward achieving predetermined goals, performance management is building on that process by adding the relevant communication and action on the progress achieved against these predetermined goals [18], to meet or exceed end-users' or business' expectations.

Performance can be thought of as actual results vs desired results. Any discrepancy, where actual is less than desired, corresponds to a performance improvement zone. Performance management and improvement can be considered as a three-stage cycle:

1. Goals and objectives are established in *performance planning*
2. A manager intervenes to give feedback and adjust performance in *performance coaching*
3. Individual performance is formally documented and feedback is delivered in *performance appraisal*

So the performance problem arises whenever there is a gap between desired results and actual results; performance improvement is any effort targeted at closing the gap. In this problem a controller is used as an analytic engine

to provide robust performance guarantees by manipulating tunable inputs to obtain the desired effect on the output of the system. From this point of view, the performance management is similar to the control approach to some extent. Control theory has well-established techniques which can be used to verify the design itself a priori by analyzing key system properties, such as its stability, convergence speed, safety, and liveness [72].

As control theory has been developing for several decades, various control strategies have been investigated to adapt to different application scenarios and specific requirements. Classical feedback control is one of the most basic and widely applied control techniques, due to its simplicity and effectiveness. More sophisticated control techniques such as adaptive control [85], fuzzy control [68], stochastic control [37], and optimal control [23] have their own features and applications to achieve performance management of physical systems.

In the following the basic idea of the general control approaches are introduced, several different control strategies are further discussed, and several application examples regarding the performance management of computing systems are presented.

General View of Control Approaches

Control theory was originally developed for physical process control. Although there are a variety of control techniques, they share the essential system elements introduced below:

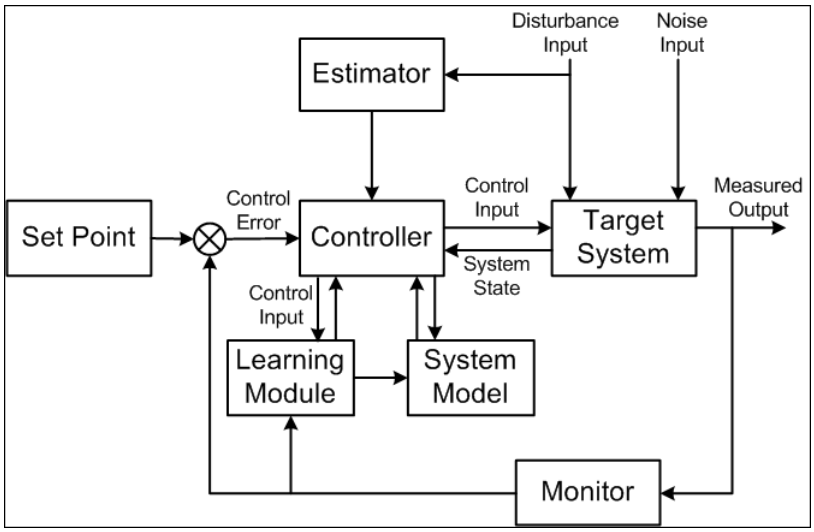

Figure II.1: Block diagram of a control system

- **Set point** is the desired value of the system response (or transformations of them) that a system aims to reach. Reaching a set point value in the system's response may have temporal requirements attached to it, such as a system response time less than 3 seconds. Sometimes, the set point is referred to as reference input or desired output.

- **Control error** is the difference between the set point and the measured output.

- **Control input** is a set of accessible parameters that affect the behavior of the target system and can be adjusted dynamically. For instance, the operating frequencies of the CPU affect the power consumption and operation speed; or the number of data transfer threads in a server affects the response time and server utilization.

- **Controller** determines the setting of the control input needed to achieve the reference input. The controller computes values of the control input based on the information it receives.
- **Disturbance input** is any externally caused change that affects the way in which the control input influences the measured output (e.g., work load of a web server).
- **Measured output** is a measurable characteristic of the target system such as CPU utilization and response time.
- **Target system** is the computing system to be controlled.
- **Monitor** transforms the measured output to state variables so that it can be used by the controller or the learning module.
- **Estimator** is a function of the observable sample data used to estimate unknown parameters.
- **System state** is the intermediate variable that is necessary to define the relationship between control inputs, measurements, and performance variables. The temperature of a processor and available memory space of a buffer are examples.
- **Learning Module** extracts information from data automatically by computational and statistical methods.
- **System model** is a dynamic model in the model-based control describing the mathematical relationship between the input and the output of a system, usually with differential or difference equations.

The foregoing is best understood in the context of a specific system [29]. Consider a cluster of three Apache Web Servers. The Administrator may want these systems to run at no greater than 66% utilization so that if any one of them fails, the other two can immediately absorb the entire load. Here, the measured output is CPU utilization. The control input is the maximum number of connections that the server permits as specified by the MaxClients parameter. This parameter can be manipulated to adjust CPU utilization. Examples of disturbances are changes in arrival rates and shifts in the type of requests (e.g., from static to dynamic pages).

Control Categories

This section provides a brief review of different control approaches applied to performance management problems.

PID Feedback Control

A proportional-integral-derivative control (PID controller) is a generic control loop feedback mechanism widely used in control systems. A PID controller attempts to correct the error between the measured output and the desired set point by calculating and then outputting a corrective action, or a control input that can adjust the process accordingly. The Proportional value determines the reaction to the current error, the Integral determines the reaction based on the sum of recent errors and the Derivative determines the reaction to the rate at which the error has been changing. The weighted sum of these three actions is used to adjust the control input. A PID controller

can be changed to a PI, PD, P or I controller in the absence of the respective control actions.

The feedback mechanism causes system performance to exhibit a self-correcting, self-stabilizing behavior, and since it is also robust, system convergence is observed even in the presence of modeling inaccuracies, inherent system nonlinearities, and variations of system parameters over time. However, it essentially is a reactive approach. The feedback loop operates by responding to measured deviations from the desired performance, i.e., exerting corrective action that attempts to reduce the deviation to zero. Unfortunately, a feedback controller, which measures the current delay, will remain oblivious to the impending overload until it occurs. This delay in response is particularly significant since the server response time is a moving average that is slow in response to changes.

In the server application, [5] uses a digital approximation of a linear continuous PI controller to measure server utilization. Paper [6] provides QoS guarantees in web server resource management also by applying a PI controller. For real time scheduling problems, [57] presents a feedback control real-time scheduling (FCS) framework with arriving-time QoS control, and [58, 59] propose a feedback control to guarantee low deadline miss-ratio in unpredictable environments, but the latter integrates PID control with an EDF scheduler and was applied to a practical case study in [80]. In service management, [32, 60] provide relative delay guarantees for different services classes on web servers, and [73] applies a saturated integral controller to the evaluation of controllers used for service level management of software system. Moreover, [81] designs a feedback-based controller to allocate CPU to

threads based on proportion and period, [66] casts cache resource allocation as a feedback-based controller design problem, [17] describes a mechanism for scalable feedback control of multi-cast continuous media streams to establish and maintain video conferences of reasonable quality even across congested connections in the Internet, and [40] applies feedback control theory to analyze a congestion control algorithm on IP routers.

Feed-Forward Control

Feed-forward control [13] sets the actuator directly based on the predicted behavior so that the system can react to disturbance before it takes effect. Further, it can be used to keep the system in the neighborhood of an operating point around which it can be linearized. Feedforward control requires a model that predicts the effect of system inputs on its performance. Several theoretic foundations can be used for such prediction, including real-time scheduling theory and queueing theory. Since the feedforward controller keeps the system around the operating point, a linearized small-deviation model becomes sufficient for the purposes of feedback control. Moreover, the feedback controller eliminates the need for accuracy and corrects the steady-state error from the estimation in feedforward models.

In [76], queuing theory is applied to predict the future queuing delay based on observation of request arrival rate and estimation of service rate, for which the queuing delay in the steady state can be computed with a simple formula [51]. In [39], this predictor is improved to respond to sudden and transient workload changes. The impact on the latency of future requests

is estimated through heuristic flow-level approximation. Similar techniques and control structures are also applied in [64] for relative delay guarantees in web servers. In [24], resource allocations required to meet certain service level objectives are computed based on the predicted workloads using on-line measurements of the request arrival process, service demand distribution and queue length. The approach in [4] uses feedforward control to keep the system in the neighborhood of an operating point around which it can be linearized, in order to accommodate software non-linearities. [89] directly sets the actuation level for the next control interval based on the target value for the output metric and the predicted value for a related variable. The possibility of predicting future resource demands through resource utilization metrics such as CPU consumption is studied.

Adaptive Control

Adaptive control involves modifying the control law used by a controller to cope with the fact that the parameters of the system being controlled are slowly time-varying or uncertain. For example, as a network server works, the workload will vary from time to time according to changing demands; we therefore need a control law that adapts itself to such changing conditions. The adaptive control is precisely concerned with control law changes.

An adaptive pole placement control applies to QoS-aware web caching, providing proportional differentiation on average hit rate of different content classes [65]. The controller parameters which are updated online, are based on a linear approximation, and the controller is dynamically fed a

model to fine-tune its function. In [55], a Queueing-Model-Based Adaptive Control, formed by an online parameter estimator and a control law from the known parameter case, is proposed to handle modeling inaccuracies and load disturbances. System model parameters are estimated by a Recursive Least-Squares estimator. An adaptive control of resource containers on shared servers is presented by [54]. The indirect self-tuning adaptive controller estimates the dynamic model from online input-output measurements and computes controller parameters from the current estimated model, using recursive least-squares method and pole placement. In another paper, [46], performance for storage access is ensured using an a direct self-tuning adaptive controller. The controller considers the system as a "black box", and the system model is inferred from a monitor. The adaptive control methodology is also applied for a class of nonlinear systems [31] and used to develop an intelligent fault-tolerant control system [30], where spatially localized models are used as online approximators to learn the unknown dynamics of the system, and the applied adaptive laws are localized. A MIMO adaptive optimal controller coupled with a nonlinear optimizer is described in [45] to maximize the utility of a computer service shared by multiple customers, and in [47] to provide non-intrusive performance differentiation in computing systems. A greedy ranking heuristic is used to get an approximate solution to the optimizer. The adaptive control is also applied to adjust the timer values of temporal event correlation rules based on propagation delays to reduce missed and false alarms [36]. The algorithm is based on a technique from statistical hypothesis testing using non-parametric statistics.

Fuzzy Control

A fuzzy control system is a control system based on fuzzy logic [90], with qualitative decision-making specification. In fuzzy logic, the membership of y in a set A has a degree value in a continuous interval between 0 and 1. Fuzzy sets are defined by membership functions that map set elements into the interval $[0, 1]$. One of the most important applications of fuzzy logic is the design of fuzzy rule-based systems. These systems use IF-THEN rules (fuzzy rules) whose antecedents and consequents use fuzzy-logic statements to represent the knowledge or control strategies of the system. A fuzzy model is a qualitative model constructed from a set of fuzzy rules to describe the relationship between system input and output [82].

A fuzzy system model is used in [88] to characterize the relationship between application workload and resource demand from its input-output data without requiring a-priori knowledge about the system. The main objective is to reduce resource consumption while achieving application performance targets. Paper [56] explores approaches to online optimization of configuration parameters of the Apache web server by employing hill climbing based on fuzzy control.

Model Predictive Control

Model predictive control [22] is a widely applied methodology, which uses a model to predict the system's behavior over a finite future horizon and

chooses the control action that optimizes a cost function subject to constraints. This approach was used in [63] to control CPU utilization in distributed real-time systems. It requires an online solution to a constrained optimization problem, and thus requires a significant overhead in real-time systems. The proposed approach is also limited to time invariant linear system models that are known a priori. An event-driven model predictive controller using detected events and remaining resource information as state variables was discussed in [83] to optimally control the system (sensors, transmission, storage) in real-time.

Stochastic Control

Stochastic control is a branch of control theory that aims at predicting and minimizing the effect of the random deviations of a dynamic system. Such deviations occur when random noise and disturbance processes are present in the system, and force it to deviate from its prescribed course. Difference mechanisms have been established to reduce uncertainty and to optimize control performances.

A number of applications of stochastic control to computing systems are studied in literature. [14] applies the stochastic control theory to resource allocation under uncertainty. [91] and [92] use an approach based on a continuous-time formulation and stochastic control theory to obtain optimal solutions for transmitting deadline-constrained data over time-varying channels with the objective of minimizing the total transmission energy expenditure. Standard suboptimal stochastic control methods were also used

online in conjunction with an approximation of the cost-to-go in a task assignment problem for a fleet of UAVs in a surveillance/search mission [71].

Optimal Control

Optimal control theory, a generalization of the calculus of variations, is a mathematical optimization method for deriving control policies. Optimal control deals with the problem of finding a control law for a given system to achieve a certain optimality criterion. An optimal control is a set of differential equations describing the paths of the control variables that minimize the functional cost.

An optimal control policy is implemented in a command and control center for military air operations [84]. The policy is threshold-based to minimize the average reconfiguration time when the center experiences failures. It is also applied to plan recommended maneuvers in advanced driver assistance systems by properly formulating the risk functional [15]. The suggestions given are related to the overall risk level as well as the whole vehicle dynamics and represented the most effective control input for the case. Paper [26] then proposes a dynamic optimal control-model-based queueing theory, to guarantee the schedulability of soft real-time systems and the quality of service of incoming tasks and to improve the system throughput. To compensate for delays in communication networks, optimal controllers with the performance cost criteria are designed [53, 87]. [87] presents an optimal stochastic control methodology for networked control systems. It derives the optimal time-stamp linear quadratic gaussian controller with quadratic cost by using

dynamic programming. [7, 25, 74] use H_∞ control for IP routers management. [74] uses a frequency domain solution to synthesize the controller, [25] adopts a simple approximation for the time delay and uses the state-space solution, while [7] obtains the linear matrix inequality constraint.

CHAPTER III

LLC APPROACH DESIGN

This chapter describes the switching hybrid system model and introduces key online control concepts.

Hybrid System Model

The control approach discussed here targets a special class of hybrid systems having a finite control-input set. The following discrete-time state-space equation describes the continuous dynamics of such a system.

$$x(k+1) = f(x(k), u(k), w(k)) \tag{III.1}$$

where with k as the time index, $u(k) \subset U \subset \mathbb{R}^m$ denotes the control inputs, and $x(k) \subset X \subset \mathbb{R}^n$ and $w(k) \subset \Omega \subset \mathbb{R}^r$ are sampled forms of the continuous system state and environment parameters at time k, respectively. The system state space, the input set and an know compact set are denoted by X, U, Ω, respectively. The input set U is assumed to be finite, and the state space X is considered compact and continuous, and referred to as the operating domain of the system. While 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. Many computing systems have a limited finite (quantized) set

of control inputs and, therefore, their dynamics can be adequately captured using the above model.

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 [33, 12, 11], and the key workloads 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 [28] and Kalman filters [20]. 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 w.

$$\hat{w}(k) = \phi(\underline{\mathrm{w}}(k-1, m), p(k)) \tag{III.2}$$

where $\hat{w}(k)$ denotes the estimated value, $\underline{\mathrm{w}}(k-1, m)$ is the set of m previously observed environment vectors $w(k-1), \ldots, w(k-m-1)$, and $p(k) \in \mathbb{R}^p$ denotes the relevant estimation parameters, for instance, the covariance matrix in the Kalman filter. These parameters are typically obtained by training the model ϕ using test data representative of actual values observed in the field. We assume ϕ to be differentiable over every argument. For simplicity and without loss of generality, we also assume the estimation parameters in

the forecasting model to be constant or time invariant, i.e., ϕ is not periodically re-tuned, and assume $m = 1$. Therefore, $\hat{w}(k) = \phi(w(k-1), e(k))$ where $e(k) \in E$ is a bounded random variable reflecting the effect of the estimation error.

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

$$x(k+1) = f(x(k), u(k), w(k)) = f(x(k), u(k), \phi(w(k-1), e(k))) \quad \text{(III.3)}$$

In the above equation, $e(k)$ is the (only) uncertain parameter of the model.

QoS Specifications

In general, computing systems are required to achieve specific QoS objectives while satisfying certain operating constraints. In most real-life systems, QoS specifications may be classified in two categories.

- *set-point specification* requires that the key operating parameters must be maintained at some specified level or follow a given pattern (or trajectory); examples include system utilization levels, response times, etc. 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.

- *performance specification* is involved where relevant measures such as power consumption and mode switching, etc., must be optimized.

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. Such performance measures may also take into account the cost of the control inputs themselves and their change.

To summarize, the primary objective of the controller is to drive the computing system to the desired state x^* in "reasonable" time using an admissible trajectory. The controller may also be required to achieve a secondary objective of minimizing the transient-cost function $J'(x, u)$ as the system moves towards x^*. Then, the overall performance measure can be represented by an overall function $J(x, u)$ where the control objective is to minimize J at every time instance k, and typically uses a norm in which these variables are added together with different weights reflecting their contribution to the overall system utility.

Controller Design

Fig. III.1 shows the overall framework of a generic online controller. Relevant parameters of the operating environment, such as workload arrival patterns, etc., are estimated and used by the system model to forecast future behavior over a look-ahead horizon. The controller optimizes the forecast behavior as per the specified QoS requirements by selecting the best control inputs to apply to the system[3]. The lookahead controller can be simply

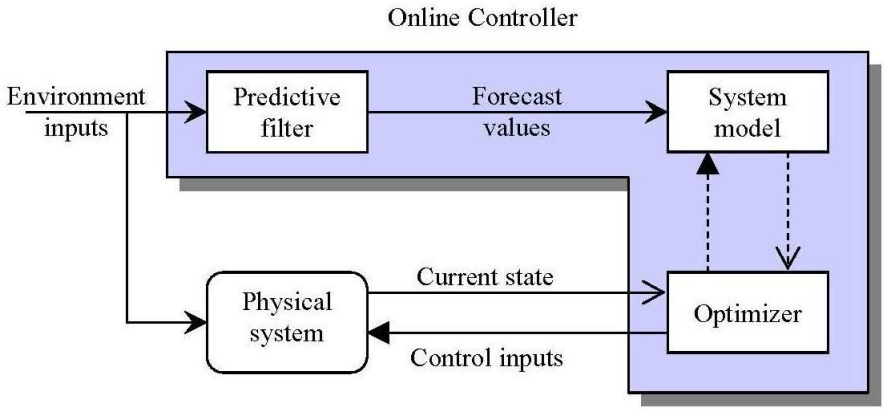

Figure III.1: Conceptual Structure of the Online Controller

considered as an agent that applies a sequence of actions to achieve a certain QoS objective. In particular, it constructs a set of future states from the current state up to a specified prediction horizon N. The controller then selects the trajectory within this horizon minimizing the cost function while satisfying both the state and input constraints. The input leading to this trajectory is chosen as the next control action. The process is repeated at each time step. The key ideas behind the controller are as follows:

- Future system states, in terms of $\hat{x}(k+j)$, for a predetermined prediction horizon of $j = 1 \dots N$ steps are estimated during each sampling instant k using the corresponding behavioral model. These predictions depend on know values (past inputs and outputs) up to the sampling instant k, and on the future control signals $u(k+j), j = 0 \dots N-1$, which are inputs to the system that must be calculated.

- A sequence of control signals $u(k+j)$ resulting in the desired system behavior is obtained for each step of the prediction horizon by optimizing the QoS-related specification.

- The control signal $u^*(k)$ corresponding to the first control input in the above sequence is applied as input to the system during time k while the other inputs are rejected. During the next sampling instant, the system state $x(k+1)$ is known and the above steps are repeated again. Note that the observed state $x(k+1)$ may be different from those predicted by the controller at time k.

A basic control specification in such system 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. As shown in Fig. III.2, in the LLC approach, the next control action is selected based on a distance map defining how close the current state is to the desired set point. This map may be defined for each state $x \in \mathbb{R}^n$ as $D(x) = ||x - x^*||$, where $||.||$ is a proper norm for n. For a performance specification, the control input optimizing a given utility function $J(x)$ is selected. This function assigns to each system state, a cost associated with reaching and maintaining that state.

Control Algorithm

Table III.1 shows the online control algorithm that aims to satisfy a given performance specification for the underlying system. At each time instant k,

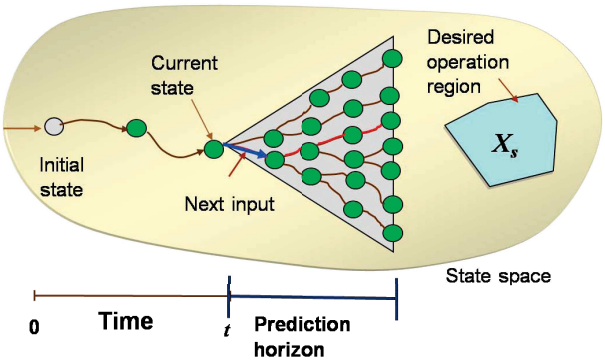

Figure III.2: The limited lookahead control approach

it accepts the current operating state $x(k)$ and returns the best control input $u^*(k)$ to apply. Starting from this state, the controller constructs in breadth-first fashion, a tree of all possible future states up to the specified prediction depth. Given an $x(k)$, we first estimate the relevant parameters of the operating environment, and generate the next set of reachable system states by applying all control inputs from the set U. The cost function corresponding to each estimated state is then computed. Once the prediction horizon is fully explored, a unique sequence of reachable states $\hat{x}(k+1), \ldots, \hat{x}(k+N)$ is applied to input $u^*(k)$ along the path to $\hat{x}(k+N)$ is applied to the system while the rest are discarded. The above control action is repeated each sampling step.

In a computation system where control inputs are chosen from discrete values, the LLC algorithm exhaustively evaluates all possible operating states within the prediction horizon to determine the best control input. Therefore, the size of the search tree grows exponentially with the number of inputs; if $|U|$ denotes the size of the input set, and N the prediction depth, the number of explored states is given by $\sum_{j=1}^{N} |U|^j$. This is not a major concern for

Table III.1: The LLC Algorithm

```
1   OLC(x(k)) /* x(k) := current state measurement */
2   s_k := x(k); Cost(x(k)) = 0
3   for all k within prediction horizon of depth N do
4     Forecast environment parameters for time k + 1
5     s_{k+1} := φ
6     for all x ∈ s_k do
7       for all u ∈ U do
8         x̂ = Φ(x, u) /* Estimate state at time k + 1 */
9         Cost(x̂) = Cost(x) + J(x̂)
10          s_{k+1} := s_{k+1} ∪ x̂
11      end for
12    end for
13    k := k + 1
14  end for
15  Find x_min ∈ s_N having minimum Cost(x)
16  u*(k) := initial input leading from x(k) to x_min
17  return u*(k)
```

systems with few control options. However, with a large control-input set, the corresponding control overhead may be excessive for real-time performance.

Characterizing LLC Performance

The goal of the LLC scheme is to optimize the system 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, system performance depends on several controller-related factors and the operating environment. One of these factors, the environment input, is not controllable, and therefore, must be neutralized with respect to the relevant performance measures. On the other hand, there are several controllable factors parameters, including

- *Prediction Horizon*: 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.

- *Control Set*: Increasing the number of control inputs improves controller accuracy and robustness with respect to environment inputs. In the case of a set-point specification, increasing the control set leads to a smaller containable region. The distribution of values within the control set can also have a major effect on control performance. In most cases, regularly quantized values for each control input leads to better performance than an irregular set.

- *Sampling Time*: In general, reducing the sampling time increases the accuracy and robustness of the controller.

The prediction horizon N can be tuned by the designer, and is only limited by the computational overhead. However, the size of the control set $|U|$ and the sampling time T are typically adjustable only within a limited range as they depend on the physical characteristics the underlying system. The above factors directly influence controller performance, characterized via the following quantitative measures.

- *Utility*: This characterizes the average cost incurred by the controlled system. The system utility is normalized with respect to the average

values of the environment inputs to reduce the effect of this (uncontrollable) factor. This performance measure can be improved by increasing the prediction horizon (up to a certain extent) and the number of control inputs, or by reducing the controller sampling time.

- *Robustness*: This characterizes the runtime variability in system utility, in response to the corresponding variability in the environment inputs. Here, we define control robustness as the standard deviation observed in the system utility against the standard deviation observed in the environment inputs, or $R = \sigma(J)/\sigma(w)$.

- *Computational Overhead*: This factor quantifies the execution-time requirement of the controller, which depends directly on prediction horizon, size of the control set, and the sampling time.

Increasing controller utility and robustness conflicts directly with reducing its computational overhead. Therefore, trade offs are necessary to achieve the desired controller performance; for example, by appropriately tuning the controller using values from (N, U, T) and synthetic environment inputs.

CHAPTER IV

ENHANCED SEARCH TECHNIQUES

As shown in the previous chapter, the search process is responsible for the exponential growth of the control algorithm. To enhance the efficiency of the control algorithm, we investigate several efficient search algorithms in the following sections that can be directly applied to the LLC approach.

Uniform-cost Search

Uniform-cost search [75] is a tree search algorithm used for traversing or searching a weighted tree, tree structure, or graph. As shown in Table IV.1, it begins at the root node, but instead of always expanding the shallowest node like breadth-first search, the uniform-cost search continues by visiting the next node with the least *Cost* - the accumulative path cost from the root to the current node. Nodes are visited in this manner until the goal state is reached. The uniform-cost search is complete and optimal if the cost of each step is greater than or equal to some small positive constant ϵ [75]. But when all path costs of the uniform-cost search are positive and identical, it changes back to breadth-first search.

The space complexity of the uniform-cost search is the number of nodes with *Cost*s smaller than or equal to the cost of the optimal solution, plus the ones extended by those nodes. The time complexity is the time needed to process the nodes. Formally, if C^* is the cost of the optimal solution and

Table IV.1: Uniform-cost search algorithm

```
1   Initialize Let Q = S /*S := start node*/
2   while Q is not empty
3     pull Q1, /*Q1 := first element in Q*/
4     if Q1 is a goal
5       report success and quit
6     else
7       childnodes = expand(Q1)
8       <eliminate childnodes which represent loops>
9       put remaining childnodes in Q
10       delete Q1
11       sort Q according to Cost /*Cost := pathcost(S to node)*/
12    end if
13   continue while
```

it is assumed that every path cost is at least ϵ, the algorithm's complexity is $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, instead of $O(b^d)$ in breadth-first search.

We implement the uniform-cost search for the LLC approach following the pseudo code in Table IV.1. Typically, the search algorithm involves expanding nodes by adding all unexpanded neighboring nodes that are connected by directed paths to a priority queue. In the queue, each node is associated with its *Cost*, and the least-*Cost* node is given highest priority, so that the queue is sorted in an ascending order. The node at the head of the queue is subsequently popped and expanded, appending the next set of connected nodes with their *Cost*s to the queue.

The completeness and optimality of the uniform-cost search can be guaranteed by setting even-exponent terms in the utility function of the *Cost* to make all the path costs positive. The utility function at time k can be

designed by the following form:

$$J(k) = \beta_1 y_1^2(k) + \beta_2 y_2^2(k) + \cdots + \beta_m y_m^2(k)$$

when there are m components the utility function tries to optimize, $y_i(k), i \in m$ represents a component at time k, and β_i is the user-specified weight denoting the relative importance of $y_i(k)$. We can also use even exponents instead of squared ones as shown in the equation according to application specifications. Moreover, in the control framework, usually different values are assigned to the components of the utility function, so the path costs will rarely be identical. The two conditions above provide promising supports for applying uniform-cost search in the control algorithm. But as uniform-cost search is guided by path costs rather than depths, sometimes its complexities cannot easily be characterized and its worst-case time and space complexities can be much greater than those of a breadth-first search.

A* Search

We have only considered the path costs from the starting node to the current node, but not the possible costs from the current node to the goal node in the tree structure. A* search, one of the most widely-known form of best-first search, 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:

$$f(n) = g(n) + h(n)$$

Then the algorithm complexity is determined by both $g(n)$ and $h(n)$.

Note that uniform-cost search is a special case of the A* search when the heuristic $h(n)$ is constant, so the A* search algorithm is similar to the uniform-cost search in Table IV.1 except that the *Cost* is given by pathcost(S to node) + h(node) instead. A* is complete in the sense that it will always find a solution if there is one. However, its optimality depends on if $h(n)$ is an admissible heuristic, or never overestimates the cost to reach the goal. Formally, for all paths y, z where z is a successor of y,

$$g(y) + h(y) \leq g(z) + h(z)$$

A* is also optimally efficient for any heuristic h; no algorithm employing the same heuristic will expand fewer nodes than A*, except when there are several partial solutions where h exactly predicts the cost of the optimal path. Therefore, the performance of the heuristic search depends on the quality of the heuristic function. If the heuristic is accurate, we will quickly reach the goal node. Good heuristics may be constructed by relaxing the problem definition, by pre-computing solution costs for sub-problems in a pattern database, or by learning from experience with the problem class.

To apply the A* search to the control algorithm, we can compose the uniform-cost search with a heuristic function. Since computing the heuristics is always time consuming, a heuristic-cost table computed before run time is used for the control implementation. In the previous control framework, a system is always subject to environment inputs, has its own system states, and manipulates a finite number of control inputs to the system, all

of which are key characteristic behaviors of the control system. Based on the underlying utility function, we can define a 3-dimensional heuristic table $heuristic(w, x, k)$. In this table, $w \subset \Omega$ denotes the environment input, $x \subset X$ represents the system state, and k is the step distance from current node to the goal node. Note that w and x refer to their respective groups of elements. If there are several environment inputs and they are related to each other, we can use just one to represent all the others; but if some of them are independent, we can either increase the dimension of the heuristic table, or only choose the more significant ones. More system states can be treated in the same way as the environment inputs. Then a cell c at position $heuristic(w, x, k)$ stores the estimated smallest accumulated cost value of a node with a system state of x, environment input of w, and step distance of k. The accumulated cost is the total cost from the node c to the goal node in the search tree.

Before computing the final heuristic table, several issues need to be specified.

1. w and x may not be integers. But according to the requirement of the heuristic table, w and k are matrix indexes, so we must ensure that they are integers before accessing the table, by rounding them down or up, or by mapping them to integral indices.

2. The ranges of w and x may be large. For example, when w is from 0 to 10000, it is not practical to generate a table of 10001 cells considering space limitation. Instead, we can select certain data points $0, 50, 100, \cdots$, and map them to the table indices $0, 1, 2, \cdots$.

3. Admissible heuristic should be satisfied to guarantee the optimality of the A* search. Thus data should always be underestimated by using a value equal to, or smaller than the real value whenever necessary. For instance, for a workload $w = 346$, if we only have data points at multiples of 50, w will be rounded down to 300.

4. An assumption of w is made. We need to iterate k steps to calculate the heuristic cost, but we do not know what the next value of w will be. To solve this problem, we define the difference of the environment inputs between two adjacent simulation steps as Δw. Assume that Δw is bounded, and is relatively small compared with the maximum value of w. Then a new w can be estimated by decreasing the last environment input by Δw if smaller environment input causes less cost; or increasing it by Δw if larger one causes less cost. This will help prevent overestimating the path costs.

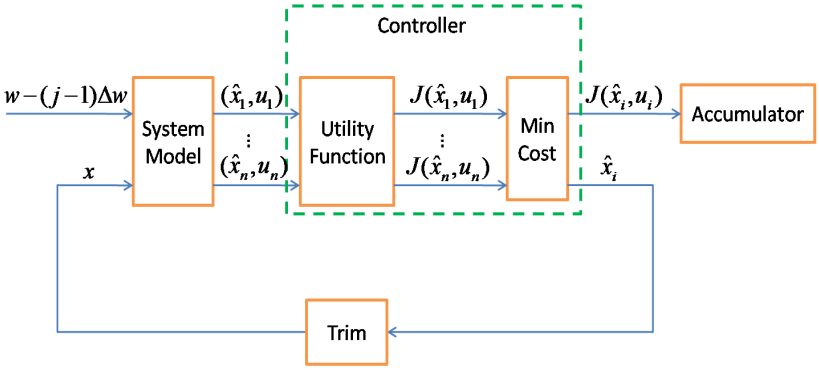

Figure IV.1: Generation of the heuristic table

Fig. IV.1 shows the steps to compute the heuristic table. For each combination of w, x and k, w and x are initially sent to the system model to calculate $\hat{x}_i$, the next system state corresponding to the control input $u_i \in U$. The j is initialized as 1, and will be increased by 1 after each loop. Assume the control set has $|U|$ control inputs, all the $|U|$ 2-tuples $(\hat{x}_i, u_i)$ are sent to the utility function to obtain a cost $J(\hat{x}_i, u_i)$. The smallest cost is then added to the accumulator(initialized to 0), while the system state $\hat{x}_i$ is trimmed, e.g. rounded to an integer. $\hat{x}$ and $w - \Delta w$ are further used as inputs for the next iteration. The computation will iterate k times, and the final value of the accumulator is filled to the cell $\mathit{heuristic}(w, x, k)$. Each cell of the heuristic table is calculated this way.

The calculated heuristic table will be used once a node is extended. After mapping environment input, system state and step distance of the current node to corresponding indices w, x and k in the table, we can get $\mathit{heuristic}(w, x, k)$ as the heuristic $h(n)$.

Assume that there are n_x, n_w, and n_u elements of system state, environment input and control input respectively in the heuristic table. According to the calculation of the heuristic table, for each element of x and each element of w, all the control inputs u_i will be tried for k iterations. Therefore the time complexity of calculating the heuristic table is $O(n_x * n_w * n_u * k)$. But as the heuristic table is calculated offline before system execution, the time cost is not a significant problem. The space complexity of the heuristic table is $O(n_x * n_w * k)$.

The complexity of the A* search is also $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, as it is based on the uniform-cost search and adds heuristics by just looking them up in the

heuristic table. In addition, since ϵ becomes the smallest underestimated cost from the root to the goal here, it should be larger than that of the uniform-cost search, and therefore the A* search will be faster than the uniform-cost search.

Pruning Algorithm

A search space is a structure built with all available information for finding the most suitable areas. However, sometimes the given set of data may be irrelevant, erroneous or unnecessary. Therefore pruning the search space is necessary. Pruning is a process of making the search space smaller by removing selected subspaces. Ignored portions of the space are no longer considered because the algorithm knows based on already collected data (e.g. through sampling) that these subspaces do not contain the searched object, and the pruning will therefore not affect the final choice [75].

In the search tree of the control algorithm, the system states of some nodes turn out to be the same. Moreover, from the definition of the control algorithm, nodes at the same depth will receive identical environment input. So if the nodes with same system states are at the same depth, their future evolutions will be the same. In this case, only the one with the smallest cost needs to be kept for further extension, while all the others having the same system states can be pruned together with their subtrees. If the successors of the kept node in the pruning process are invalid, then the successors of the deleted nodes will be invalid as well as they share the same future. Thus the

above pruning approach is complete. In addition, the 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.

In the implementation of the pruning, because we only compare each node with the one right before it within the same level, just one extra node needs to be stored for the comparison. Since the pruning is always combined with some other search algorithms, the complexity of the combined search depends on the complexity of the other search algorithms. However, as the pruning will largely reduce search space, especially when nodes with similar system states have close costs, pruning will decrease the search complexity.

Greedy Algorithm

A greedy algorithm is any algorithm that follows the problem solving metaheuristic of making the locally optimal choice at each stage with the hope that this choice will lead to a global optimum [27]. The algorithm will generally not find all the solution or the best solution, but a feasible one, because it usually does not operate exhaustively on all the data. It may make commitments to certain choices too early which prevent it from finding the best overall solution. Nevertheless, it is useful for a wide range of problems, particularly when overhead reduction is essential. In many practical situations, this approach can lead to good approximations of the optimum.

Beam search [16] can be viewed as a greedy algorithm. For a beam search of width k, the search only keeps track of the k best candidates at each step,

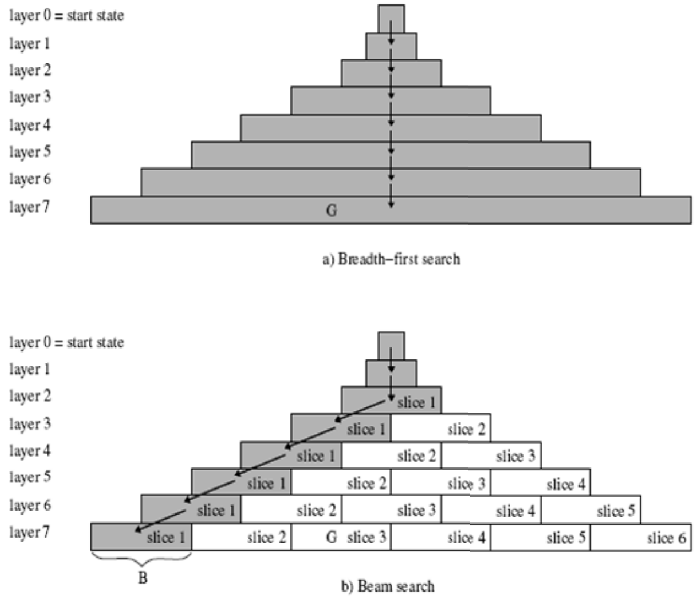

Figure IV.2: Visualization of the beam search

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 B states, where B is called the beam width [34]. 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 (from left to right in the figure), splits them into slides of at most B states each, and then extends the beam by storing the first slides only. Beam search terminates when it generates a goal state or runs out of memory. Therefore the beam search reduces the memory

consumption of breadth-first search from exponential to linear, as illustrated by the shaded areas in Fig. IV.2.

In our application of the beam search, we further define a vector by assigning the number of the best candidates for each level. Then we can change the beam width as well as the shape of the beam search according to system specifications. As one of the greedy algorithms, beam search has a serious drawback – it is incomplete, so it does not guarantee an optimal solution. However, the speed of the search and the possibility that the search obtains a solution close to the optimal one can be enhanced by changing the beam width. The search complexity will also depend on the values of the beam width. When a system has a relatively loose performance requirement but requires short and strict timing, the beam search may be a good choice.

CHAPTER V

ACME DEVELOPMENT

ACME Overview

Effective self management requires the ability to monitor and tune system variables that affect various QoS related parameters. Those parameters are often inter-dependent, i.e. modifications made on one may affect others. Also, operational constraints such as resource limitations and safety margins impose additional requirements on the system. The inter-dependencies and constraints need to be effectively captured for a self-management design. In addition, future variations in the system components and structure need to be considered as well to guarantee the system performance. Control-based techniques have proven to be effective in addressing the above requirements for self-management design and in addition they can provide performance guarantees under given operating conditions. However, the adoption of such techniques remains limited due to lack of tools and libraries that facilitate the control-based design for design engineers.

To address this problem, we propose in this thesis the ACME framework. The ACME is control-theory oriented framework aimed at providing effective self management for computation systems. Fig. V.1 shows the development process of the ACME. Structurally, the ACME is composed of three main aspects. One is the architecture structure, in which high level components

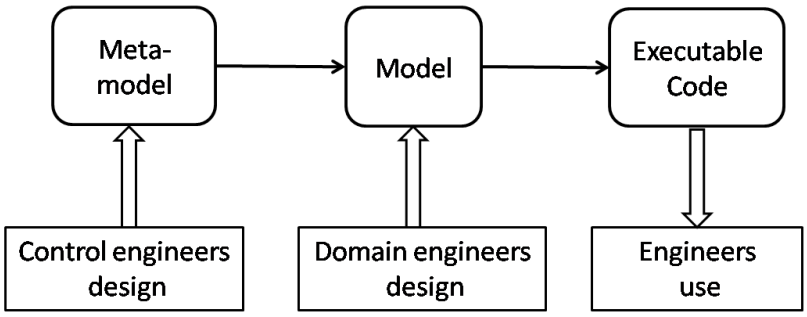

Figure V.1: ACME design process

and their interconnections are defined. Another is the data collection structure, which is responsible for collecting system measurements corresponding to the model variables. The third one is the system dynamics structure used for capturing the system model, specifications and operation constraints, as well as providing modules for estimating system future variations and tuning system variables with respect to operational variations and constraints.

Although LabView and MatLab have control toolkits with similar interface to ACME, the ACME can generate various executable codes, like C++, XML, Python, or even MatLab codes upon requests based on the graphical models. More importantly, control engineers can easily modify the modeling structure and specifications when necessary by updating meta-models.

The following subsections describe the semantic intent of the key modeling components in ACME. In this thesis, to enhance readability, the following font-based notations are adopted: "components" used for the main components of the meta-model, "`connections`" used for the connections between the components, "**visible**" used for the visibility aspects of models, and "*attribute*" used for the component attributes.

Architecture

The architecture in the ACME captures the main structure of the whole system. It contains any of the components in the self-management design, as well as the connections between the underlying ports of these components. From the architecture level of view, the designer can construct the high-level components of the system and define the connections betweens them. The details of these components are encapsulated in the underlying substructures, which have their own internal descriptions.

Data Collection

The data collection entity contains all the system variables. In practical systems, some of the system variables can be measured directly while others cannot. In some situations, system variables that cannot be measured can still be calculated based on the measured variables using observers. In other applications, future values of certain system variables need to be estimated. ACME distributes the data collection tasks to three different entity models as follows.

First, a Sensor model reads in all the measurable data, which include environment inputs, observable system states, and system outputs. Latency, bandwidth, and CPU utilization are examples of observable system states for some class of systems. Second, to calculate the system states that cannot be observed directly, an Observer model collects all the related variables and computes the system states by association equations. Third, an Estimator model uses the latest and historical sample data to estimate future system

variables. An example of implemented estimators in ACME is the autoregressive moving average (ARMA) estimator. In general, the user can choose estimators that best fit the system configuration from an estimator library in ACME.

System Dynamics and Adaptation

In the ACME framework, the system dynamics is a schematic description that captures the known or inferred behavioral properties of a computational system. The system dynamics is used for the design and verification of the self-managing structures.

The system adaptation specification represents the configuration of a controller module chosen from the control library available in the ACME. For example, the LLC controller can be selected as the system adaptation module, and can be configured by identifying the look ahead horizon, the possible control input set, and a utility function that characterizes each point in the QoS space with a utility value (or cost). The LLC utilizes these specifications to manage the system at run-time by optimizing the underlying system utility within the constraints posed by certain operational requirements.

ACME Meta-Models

This section introduces the ACME meta-models corresponding to the basic aspects of a self-management design specification. The aim of this modeling approach is to capture the system design in a modular component-based form that can be easily accessible to the system designer. For example, the

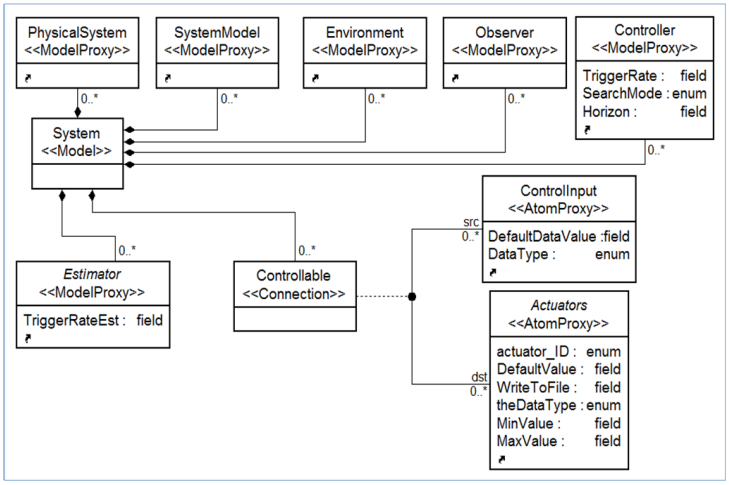

Figure V.2: meta-model of the architecture modeling

Estimator model discussed in the previous section can be added to the architecture as a high-level component, parameterized, and connected to other model blocks in the architecture through their available ports. In the following subsection we presents the ACME meta-model, which is expressed with a stereotyped UML class-diagram notation. The stereotypes including <<Model>> , <<Atom>>, <<Connection>>, etc., express the binding of the abstract syntax to the concrete syntax implemented by the GME environment. Details of the concrete syntactic constructs supported by the GME environment are presented in [1, 52]. The sub-languages that constitute the ACME language are addressed below.

Architecture Models

The architecture stereotyped as a folder, contains a System model that collects all necessary parts of a system, each of which encapsulates its local

components. In a distributed system, such as a web-server, a system involves multiple subsystems, each of which has independent local controllers with different performance requirements; also, a global controller addressing system-wide performance requirements will be constructed for the system, managing the interaction between the local controllers.

This model expresses the general structure of the overall system. Fig. V.2 shows the meta-model of the architecture modeling sub-language. Note that the meta-model figures only show the main models, while other models are diminished in gray for simplification. 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. So PhysicalSystem, SystemModel, Environment, Observer, Controller, and Estimator are all key components which can be contained in the System.

The connections in the architecture define data transportation between models. As shown in Fig. V.2, 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". Similarly, signals in the Environment models can be sent to the Estimator models by `SensorToEst` connection, to the Observer models by `Measurement` connection, or to SystemModel by `SensorConn` connection; estimated variables can be sent from the Estimator models to the SystemModel through `EstSignalOut` connection; ports of system states in different blocks can be connected to each other by `SystemStateConn` connection, as can of control inputs with `ControlInputConn` connection.

Data Collection Models

All basic data types used in the meta-model like the ControlInput are first defined in a component paradigm. SystemState, SystemOutput, and ControlInput are basic types of variables for control systems. ControlInput and SystemState represent the control inputs and system states respectively. SystemState and ControlInput can be used in the Observer, SystemModel and Controller models, while SystemOutput is used in the Observer only. Composite data types can be defined and modified only in the component paradigm, since data used in all the other places are proxies of the data in the component. The following models are used to get the values of the data proxies. The data types are often defined with their attributes, some examples of the attributes are name, type, IP address, and speed in a configured network system. In Fig. V.2, ControlInput has two attributes *DefaultValue* and *DataType* in the lower half of the class rectangle.

Environment Model

The operation plants involved in certain environment always interact with the environment. The Environment model then represents the operation environment. In real time applications, the Environment only contains Sensor models to measure relevant environment variables from the real environment; in the simulation application, environment is simulated and environment variables are generated by the methods defined in a data generation library. For

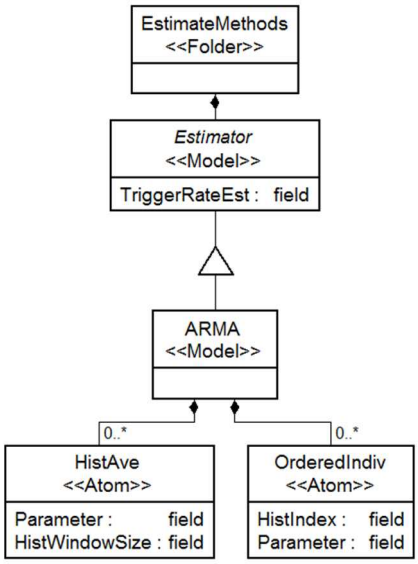

Figure V.3: meta-model of the ARMA estimator

example, in the library, model Reader reads in data from local files, and Generator model can generate uniform distributed numbers. The generated data are then sent to other components via Sensor models.

Estimator Model

The Estimator model can be selected from an estimator library, where different estimators like ARMA filters and Kalman filters are included. For example, we use an ARMA filter to estimate the environment parameter such as future data arrival rate $\hat{\lambda}(k+1)$. Given the arrival rate $\lambda(k)$ at time k and the mean $\bar{\lambda}$ of past observations over a specified window size of m, the estimate rate for $k+1$ is:

$$\hat{\lambda}(k+1) = (1 - \sum_{i=0}^{m-1} \beta_i)\bar{\lambda} + \sum_{i=0}^{m-1} \beta_i \lambda(k-i), i \in [0, m]$$

where the gain β determines how the estimator tracks variations in the observed arrival rate. The ACME uses two kinds of models to represent the ARMA filter. The HistAve model specifies $\bar{\lambda}$, and its attribute *HistWindowSize* defines m. The OrderedIndiv model specifies the $\lambda(k-i)$, and its attribute *HistIndex* defines i (e.g. a *HistIndex* of 1 represents the $(k-1)th$ observed data). Both models have *Parameter* attributes defining the gains $(1-\sum_{i=0}^{m-1}\beta_i)$ and β_i respectively.

Observer Model

The Observer model calculates unobservable system states using measurable variables and parameters if the underlying functions are available. All the needed variables like SystemOutput, Variable, ControlInput, and SystemState are read in the Observer to the Function models to calculate the unknown values. Finally, SystemStates hold the computed data and assign them to other models.

Controller Model

The Controller model specifies the parameters of the controller design, and Fig. V.4 shows the meta-model of the LLC controller, which has an attribute *Horizon* specifying the prediction horizon of the LLC. It contains Utility, ControlInputSet, and SetPoint models. The Utility has three important attributes: *Constraints* includes the constraints the system need to follow, *UtilityFunction* is to write the utility function, and *Operation* decides

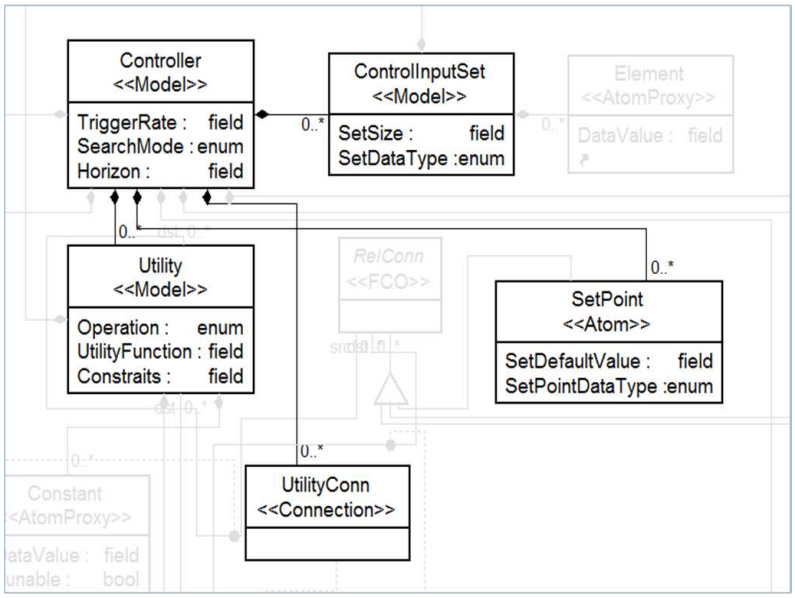

Figure V.4: LLC meta-model

whether to "minimize" or "maximize" the utility function. The ControlInput-Set contains all the available control inputs for the system. SetPoint is the target value that the automatic control system aims to reach. ControlInput, SystemState and SetPoint can be sent to the Utility by `UtilityConn`. Users can then use the LLC by setting the above values of the models without knowing the implementation details.

System Dynamics Model

The system dynamics specifies the behavioral characteristics of a computation system. The ACME has three types of models for the system dynamics: SystemModel, PhysicalSystem_sim and PhysicalSystem. In System-Model and PhysicalSystem_sim, the behavioral characteristics are expressed by hybrid automata or mathematical functions, through which system states

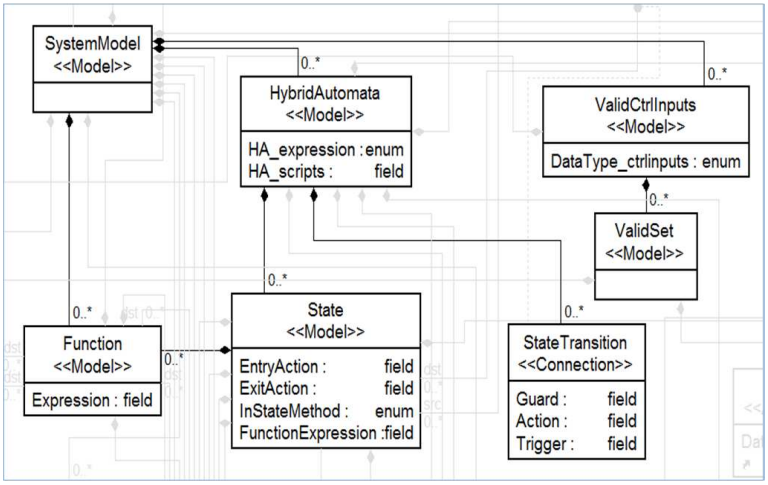

Figure V.5: meta-model of the System Dynamics

are updated. The general forms of HybridAutomata notation and Function notation are defined in the meta-model. In PhysicalSystem, the behavioral characteristics are the physical system states measured by the Sensor models.

The key models of the SystemModel as shown in Fig. V.5 are HybridAutomata, Function, and ValidCtrlInputs. The HybridAutomata has State models, including one InitialState in each HybridAutomata, and `StateTransition` connections between them. State has attributes *EntryAction*, *ExitAction*, and *FunctionExpression*; `Transition` has attributes *Action*, *Trigger*, and *Guard*. The transitions can be addressed in the attribute *HA_scripts* of the HybridAutomata, or modeled inside the HybridAutomata by choosing from the *HA_expression* attribute, "Using scripts" or "Embed HA inside". The HybridAutomata model also has two aspects: **FSMAspect** and **DataFlowAspect**. In the **FSMAspect** state transitions are visible, while the **DataFlowAspect** demonstrates how data flow into States. The Function model has an *Expression* attribute that captures mathematical relations. 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 like 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.

PhysicalSystem_sim model is used to simulate the behaviors of physical systems. Similar to the SystemModel, PhysicalSystem_sim has HybridAutomata and Function. It also has Actuator and Sensor models corresponding to the same elements as in the real physical system.

The PhysicalSystem, working in a real-time application mode, contains Actuator and Sensor models. Sensor receives system states from, and Actuator sends control inputs selected by Controller to physical plants. Both models have two main attributes: *sampling rate* and *accuracy*. System dynamics can also be included if the system can be analytically modeled.

ACME Interpreter

Interpreters are model translators designed to work with all models created using the domain-specific GME. The translated models then can be used as sources for analyzing programs [1]. We use a framework named Builder Object Network version 2.0 (BON2) to access the ACME components and the relationships between them. The BON2 generates the basic files of the interpreter, and our work consists of writing the crucial portion of the interpreter code. First, the interpreter navigates the object network and traverses

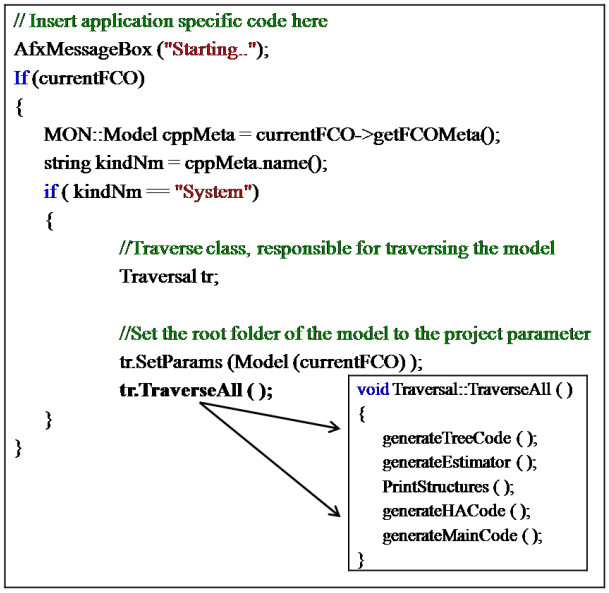

Figure V.6: Navigating the object network

all the models. If a System exists, the traversal will start using TraversalAll() in the `Component::invokeEx()` function, and the TraverseAll() function will generate necessary files successively as in Fig. V.6, when each individual component is queried by accessing its properties, attributes, meta-information, or associations. For instance, the LLC controller code identifies the Controller by the model property, reads the *Horizon* attribute from the Controller, and obtains the associated system states and control inputs. The generated scripts are ready to run for execution.

Following are the descriptions for the sub functions of the TraverseAll() function.

- *generateTreeCode():* In generateTreeCode(), two functions generateTreeHeader() and generateTreeSource() are included, which generate a

Tree.h and a Tree.cpp separately as a library providing tree structures. The tree structure is to help do some computing.

- *generateEstimator():* Same as generateTreeCode(), function generateEstimator() is also to generate a library of an estimator to help work on prediction. This function will generate an "Estimator.h" file. Libraries are mostly independent of user's applications, so they do not require much information from GME models.

- *PrintStructures():* Next function PrintStructures() is to print a structs.h file with a structure containing the current simulation time, system states and control inputs. Figure 3 shows the main body of the PrintStructures(). It traverses the PhysicalSystem_sim model in the System model, and collects data it needs. Note that we use several 2-dimension arrays here. Actually three arrays are defined in the Traversal class: systemstate[ ][ ], ctrlinputs[ ][ ], and Env_var[ ][ ]. Currently, systemstate[i][0], ctrlinputs[i][0], or Env_var[i][0] stores the date type, while systemstate[i][1], ctrlinputs[i][1], or Env_var[i][1] keeps the name.

- *generateHACode():* The fourth function generateHACode() is responsible for generating HA_PSi.h and HA_PSi.cpp files or HA_SMi.h and HA_SMi.cpp, where i=0, 1, 2, ... As there may be several hybrid automatons in the physical system model or system state model, i is to distinguish them from one another. Now the transition functions are simply embedded as scripts in the GME model. We just need to read the scripts are put them into the transition function. A library

class of hybrid automaton can be added, thus the GME model can be interpreted directly instead of asking for scripts from users.

- *generateMainCode():* The last one, also the most important one, the generateMainCode() function also interprets using two sub functions: generateMainHeader() and generateMainSource(), one to write .h file and the other for .cpp file.

  The generateMainHeader() function has two sub functions as well.

  First, it calls PrintInitialParam() to output all the important definitions of variables, constants, and global functions. It needs to traverse all the models in the main System model to get all the useful values. The generateMainHeader() calls the PrintClassHead() next to write definitions of PhysicalSystem class, SystemModel class, and Controller class into the .h file. Figure 4 shows the skeleton of the PrintClassHead(). Note that in the full code, there is a _maincpp variable appears quite often in this function, which is to collect data for the .cpp file. To work out those classes, the function traverses the PhysicalSystem model, the SystemModel model, and the Controller model separately.

  Second, a function GetUtilityPrecedence() in PrintClassHead() is to get all the utility functions. In the Controller block, there can be several Utility models with their own functions on each of them. Each Utility model has one output or no output, but it can have several inputs. All the Utility models have one output except the last one. So the GetUtilityPrecedence() will find the last Utility model first, then backtrack all its inputs. If the root of an input is also a Utility model,

GetUtilityPrecedence() is called again inside the function itself. After calling another GetUtilityPrecedence() function, the first GetUtilityPrecedence() outputs its utility function to the .cpp file. Otherwise, the input is a variable used by the Utility model. We just need to read the variable into the .cpp file, followed by the utility function. Figure 5 shows how this function works. There is another function named GetFunctionPrecedence(), very similar to the GetUtilityPrecedence(). The difference is that Function block always has one output to the SystemState. Thus, we start from one of those atoms to backtrack all the functions. The GetFunctionPrecedence() is called whenever Function model appears.

The generateMainSource() function is simple. We just output the collected data _maincpp to the .cpp file.

CHAPTER VI

CASE STUDY

To demonstrate the efficiency of the search algorithms and the applicability of the ACME modeling, we did a case study concerning power management(PM) of a DVS-capable processor operating under a time-varying workload [3].

Problem Formulation

Power consumption has become an important design constraint for densely packed processor clusters due to electricity costs and heat dissipation issues [69]. 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 [8] and Pentium M processors [41] offer a limited number of discrete frequency settings, eight and six, respectively. We first apply the LLC approach to manage the power consumed by such a processor under a time-varying workload comprised of 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.

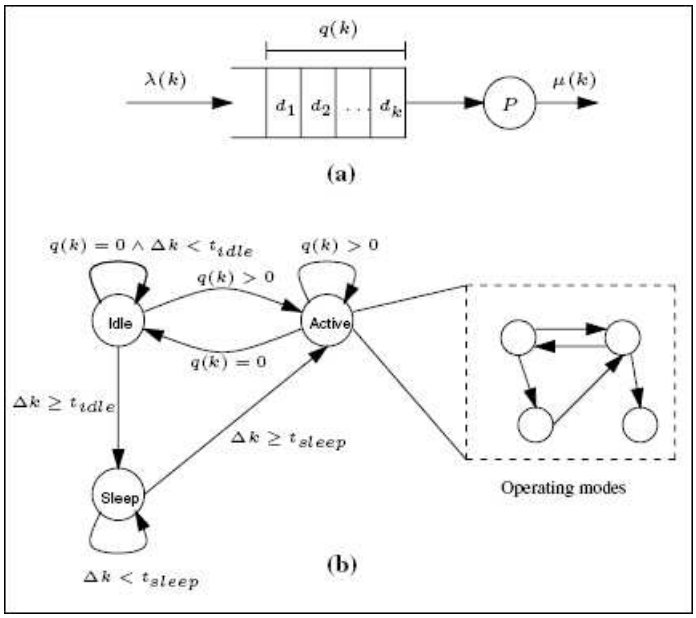

Figure VI.1: (a) A queueing model of the processor and (b) a hybrid automaton representation of processor operating modes

Processor Model

Fig. VI.1(a) shows a simple queueing model for 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 the incoming data stream d_i, where Λ and Υ are bounded, and $q(k)$ is the queue size at time k [42]. We do not assume a prior arrival-rate distribution for d_i and P does not idle if the queue contains data items; queue utilization is given by $\rho(k) = q(k)/q_{max}$ where q_{max} is the maximum queue size.

Processor P may be treated as a switching hybrid system and its operation is represented using the hybrid automaton model in Fig. VI.1(b) [10]. Transitions between operating modes may be triggered by events or the passage of time. For example, when the queue is empty, P is idled to save power;

when new events arrive, P is switched back to the active state with little time overhead. If the processor stays idle beyond some threshold duration, it is placed in the sleep state for a specified time period. In this state, however, P does not register external events, and consequently, they are simply dropped. The processor transitions back to the active state at the end of the sleep period. We assume P can be operated at multiple frequencies. Therefore, its active state is a bounded collection of discrete sub-states, each with a specific frequency setting f_i. In this state, power consumption can be minimized by scaling f_i appropriately. Power consumption relates quadratically to supply voltage which can be reduced at lower frequencies [21]. Consequently, energy savings can be quite significant. We denote the time required to process d_i while operating at the maximum operating frequency f_{max} by c_i. Then the corresponding processing time while operating at some instantaneous frequency $f(k) \in f_i$ is $c_i/\alpha(k)$ where $\alpha(k) = f(k)/f_{max}$ is the appropriate scaling factor. The energy consumed by P while operating at $f(k)$ is given by $\alpha^2(k)$ [78] and this simple energy model has been shown to provide reasonably accurate estimates [67].

This section develops a controller to address P' s power consumption in the active state. It can, however, be readily integrated with techniques such as predictive shutdown [79] to affect the other mode transitions in Fig. VI.1(b).

Model Dynamics

The following equations describe the dynamics of the processor in the active state:

$$\hat{q}(k+1) = max\{q(k) + (\hat{\lambda}(k+1) - \alpha(k+1)/c(k+1)) \times T, 0\} \quad \text{(VI.1)}$$

$$\hat{D}(k+1) = max\{\hat{q}(k+1) - q_{max}, 0\} \quad \text{(VI.2)}$$

$$\hat{q}(k+1) = \hat{q}(k+1) - \hat{D}(k+1) \quad \text{(VI.3)}$$

$$\hat{r}(k+1) = (1 + \hat{q}(k+1)) \times \hat{c}(k+1)/\alpha(k+1) \quad \text{(VI.4)}$$

$$\hat{\rho}(k+1) = \hat{q}(k+1)/q_{max} \quad \text{(VI.5)}$$

$$\hat{E}(k+1) = \alpha(k+1)^2 \quad \text{(VI.6)}$$

Given the observed queue length dynamic $q(k)$ at time instant k, Equation VI.1 estimates its length at time $k+1$ where $\hat{\lambda}(k+1)$ and $\hat{c}(k+1)$ denote the estimated data arrival rate and execution time, respectively. When the queue is full, the estimated dropped requests are represented by $\hat{qd}(k+1)$. The average response time of requests arriving during the time interval $[k, k+1]$ is estimated as $\hat{r}(k+1)$ in Equation VI.4, and $\alpha(k+1) = f(k+1)/f_{max}$ is the scaling factor; the execution time is obtained with respect to the maximum processor frequency f_{max}. The sampling time of the controller is denoted by T. Equation VI.5 estimates the corresponding queue utilization while Equation VI.6 gives the energy consumed by the processor.

Returning to Equation VI.1, good estimators of future system inputs and outputs are crucial to model accuracy. Here, we use an ARMA filter to estimate the environment parameter: the future data arrival rate $\hat{\lambda}(k{+}1)$ [19]. Given the arrival rate $\lambda(k)$ at time k and the mean $\bar{\lambda}$ of past observations over a specified history window, the estimated rate for $k+1$ is:

$$\hat{\lambda}(k+1) = \beta\bar{\lambda} + (1-\beta)\lambda(k) \tag{VI.7}$$

where the gain β determines how the estimator tracks variations in the observed arrival rate; a low β biases the estimator towards the current observation while larger values favor past history. Rather than statically fix β, an adaptive estimator described in [50] can be used. It tracks large arrival-rate (execution time) changes quickly while remaining robust against small variations. When the estimated values match the observed ones, those estimates are given more weight with a higher β. If, however, the estimator does not accurately match the observed values, β is decreased to improve convergence.

Control Problem

During any time interval k, the controller on processor P must select the proper frequency settings to operate P as close as possible to the desired performance criterion. Let J be the cost function to be optimized at time k. Therefore, J is determined by the queue utilization $\rho(k)$, the dropped requests D_k, the achieved response time $r(k)$ and the corresponding energy consumption $E(k)$. Lower $\rho(k)$ values are desirable since the processing delay incurred by a newly arrived data item is inversely proportional to $1-\rho(k)$,

and we do not expect any requests to be dropped. The OLC algorithm is suitably modified to minimize the following cost function to obtain the required operating frequency $f(k)$:

$$\hat{J}(k+1) = \beta_1\hat{\rho}(k+1)^2 + \beta_2\hat{D}(k+1)^2 + \beta_3(\hat{r}(k+1) - r^*)^2 + \beta_4\hat{E}(k+1)^2 \quad \text{(VI.8)}$$

where $\beta_i, i = 1, 2, 3, 4$ are user-specified weights denoting the relative importance of $\hat{\rho}(k), \hat{D}(k), \hat{r}(k)$ and $\hat{E}(k)$ respectively, and r^* denotes the desired average response time.

Performance Evaluation

We evaluated the performances of the search algorithms and the ACME using above models under a time-varying workload.

Advanced Search

We assume the scaled possible operating frequencies of the processor are 0.2564, 0.3479, 0.4349, 0.5219, 0.650, 0.7829 and 1.0000. The execution time of the workload is 0.0367ms, and the sampling period of the controller is 30ms. The weights in the utility function are all set to 1. A series of experiments are carried out to evaluate the effect of different search strategies on the controller's performance in Matlab, SIMULINK.

The uniform-cost search follows the pseudo code and employs a priority queue. All the extended nodes are added to the 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.

The A* search extends the uniform-cost search with the heuristic table $heuristic(w, x, k)$. In the PM case, workload to the system λ is the environment input, which ranges in [0, 1000]. Assume that $\Delta\lambda$ is no larger than 50, to simplify the table, the indexes $w = 0, 1, 2, \cdots, 20$ are mapped to workload $0, 50, 100, \cdots, 1000$. Moreover, the PM has three system states: queue level, dropped signal and response time, but as the last two are dependent on the first one, only queue size q is represented for x. In addition, the queue level can be decimal, but the table can not use non-integer index, so only integers between 0 and the maximum queue level have values in the table. Finally, a $21 \times 51 \times (N-1)$ heuristic table $heuristic(\lambda, q, N-1)$ is built, where N is the prediction horizon. We use $N-1$ instead of N because leaf nodes do not extend anymore and no heuristic is needed for the Nth level of the tree.

The pruning is currently combined 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 the same queue level. The pruning search then decides which subtree should be removed from the current tree structure.

Fig. VI.2, VI.3 summarize results for four prediction horizons N using a synthetic workload. They demonstrate the number of nodes extended and

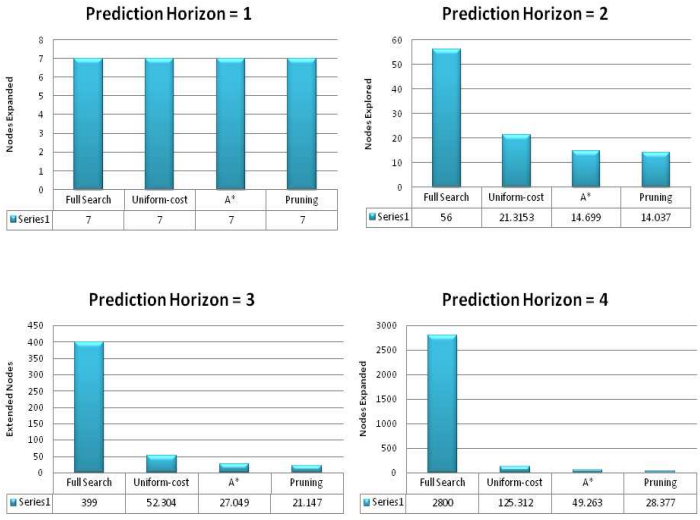

Figure VI.2: Comparing the node extended for different search strategies

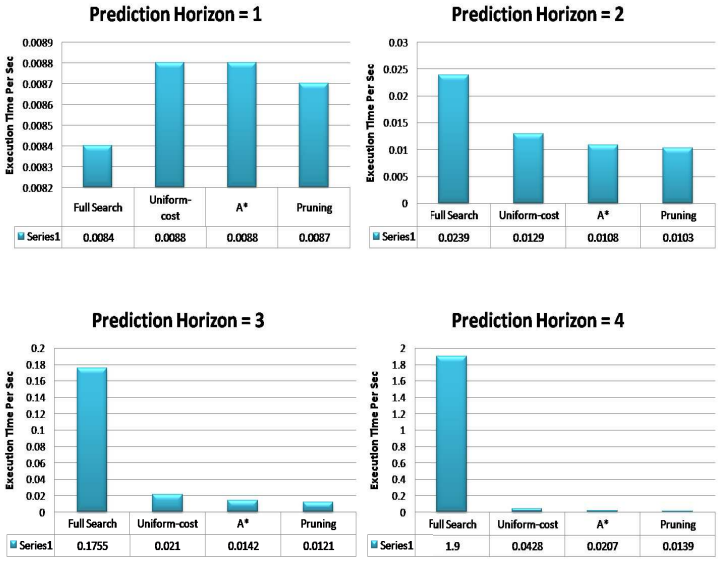

Figure VI.3: Comparing the time spent for different search strategies

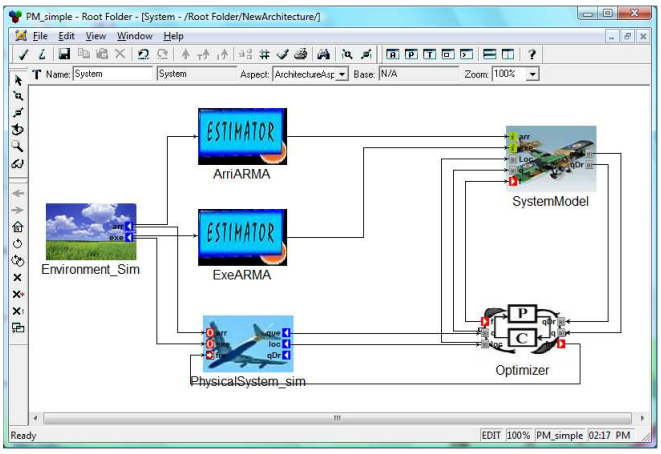

Figure VI.4: An ACME system implementation

time spent by the controller per sampling time step (workload arrivals span 9000 simulation time) for the first three methods. The larger N is, the better the new searches perform. In addition, all the control inputs obtained by the first three search methods are identical. In this special case, the pruning almost reaches the optimal solution, that is, only extending nodes on the path to the goal.

ACME Model

The generic control framework is fully developed using the ACME tool. We build the PM application using the models generated by the ACME meta-models. Fig. VI.4 is a screen shot of the implemented application, which is the architecture of the system. As the case study is in the simulation mode, we use Environment_Sim and PhysicalSystem_sim models instead of Environment and PhysicalSystem. In each simulation step, two environment variables are generated in Environment_Sim. One is the request arrival rate obtained from

a local file using the Reader 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 filters and sent to the SystemModel to forecast two system states, queue level and dropped requests, over the *Horizon* of the Controller Optimizer. The queue is 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, PhysicalSystem_sim updates the system states using the selected control input and new environment variables.

In each simulation step, the Optimizer reads the current queue level, and sends it together with all the frequencies in the ControlInputSet to the SystemModel. The SystemModel will calculate all the next possible queue sizes q_i and dropped requests D_i corresponding to the ith frequency f_i. The set q_i, D_i, f_i are compared with their SetPoint and computed in the Utility model.

Each time the SystemModel receives new data, including current queue size and all the possible frequencies, it will check the validity of the processing frequencies for the queue size in the ValidCtrlInputs model as shown in Fig. VI.5. If the frequency is valid, it will be sent to the Functions of the SystemModel together with the queue size to compute the next possible queue size, which is then sent back to the Optimizer for further operation. Otherwise, it will be discarded.

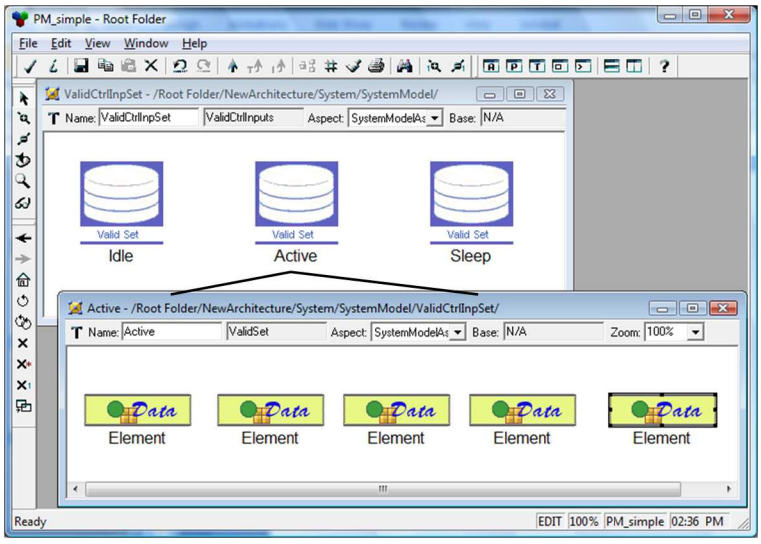

Figure VI.5: Valid control input set modeling

Performance Analysis. We tested code generated by the interpreter. The performance of the power management system is evaluated using a synthetic workload file and Fig. VI.6 shows the results of one simulation run. The processor can operate between [200, 600] Mhz with 25 Mhz increments, and the *Horizon* of the Optimizer was set to 2. For simplicity, we use the utility function of

$$J(k+1) = 0.45*(0.01*q(k))^2+0.65*(0.01*f(k))^2+1.0*(0.1*D(k))^2 \quad \text{(VI.9)}$$

The request arrival rates exhibit cyclical variations characteristic of most HTTP and e-commerce workloads [12]. From the frequency responses, we can see that the controller tracks the arrival rates well. The increase in the dropped requests is due to a sustained high request arrival rate, when the controller already operates with its maximum frequency.

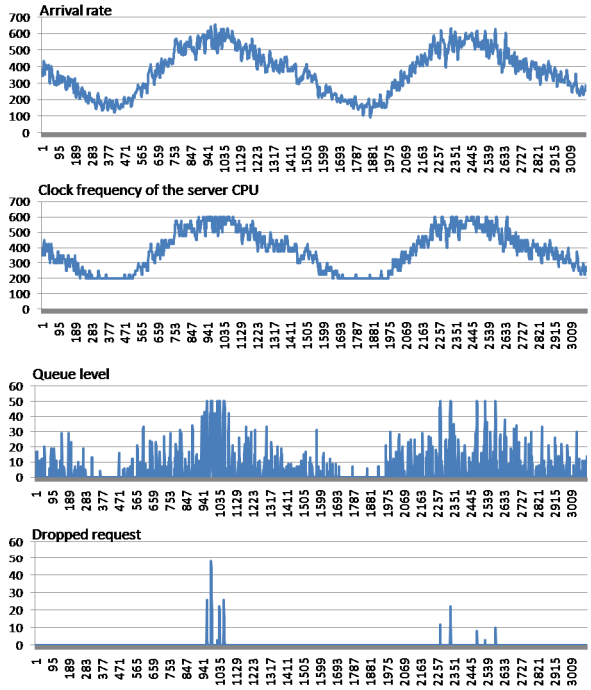

Figure VI.6: Performance of power management system

We compare the simulation results above with a similar system using a constant frequency 400Mhz for 10000 simulation steps, where the first 200 data are discarded considering the system adaptation. As shown in Table VI.1, the LLC drops only 1.5% of the requests dropped by the constant control, while spending 73.3% of the power spent by the constant control.

Table VI.1: Comparison with systems without control

	With Control		No Control		With Control/No Control
	max	average	max	average	average
Queue level	50	3.6	50	14.2	25.4 %
Dropped request	107	0.4	310	27.2	1.5 %
Frequency(Mhz)	600	342.5	400	400	85.6 %
Power cost(MJ)	360000	117306	160000	160000	73.3 %
Robustness	0.05		0.71		7.0 %

Moreover, if the frequency in the uncontrolled model is decreased, more requests will be dropped as the processing speed of the server is slower; while an increase of the constant frequency will make the system consume more power, because the frequency of 400Mhz is already greater than the average frequency 342.5Mhz of the LLC system. In addition, the robustness of the constant control is much worse than that of the LLC.

In summary, the new search strategies have shown great potential for improving the performance of the LLC approach. They try to decrease the time and space complexity by reducing the search space. 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. Overall, the new search strategies can be successfully applied to different situations, providing flexibility and useful advantages for the LLC approach.

The ACME framework abstracts the control theories and their related components into graphical and simple models. It can provide great convenience for end users. The users can not only put the models together, they can also select the models and values of variables they consider most appropriate for their systems. On the other hand, it may be not easy for developers to maintain the framework, because every time an adjustment is made to their meta-models, they need to change the interpreter code together with the models. Further, by allowing the users to select their own

model, it should better provide guidance for the selection of the models and variable values. Although the ACME is not a complete toolkit, it can be a direction of embedding control theories to computational systems, providing easy access for engineers who are not familiar with control techniques.

CHAPTER VII

CONCLUSION AND FUTURE RESEARCH

We have presented a limited lookahead control framework to design self-optimizing computing systems. In the approach, control actions governing system operation were obtained by optimizing its behavior, as forecast by a mathematical model, over a finite time horizon. We investigated the application of efficient AI search algorithms to improve the performance of the LLC framework. Furthermore, we presented a model integrated framework ACME to facilitate the design of self-managed computation systems. The proposed framework can accommodate a variety of model-based control strategies. Modules supporting the control structure such as estimators can be added and parameterized based on the user-defined system model and its specification. The framework provides supports of automatic synthesis of the managing controller modules based on a given system model, constraints and specification.

To demonstrate the performance of the proposed search algorithms, we implemented the limited lookahead controller in a case study of efficiently managing power consumption in a DVS-capable processor under a time-varying workload. Our results indicated that the search algorithms can largely decrease the memory usage and greatly speed up the system execution. The case study is also successfully developed in the ACME framework.

Although the simulation results of the case study are satisfying, the case study may be too simple to address hidden problems. In future work, we will

test more complex applications in order to correct and improve the current approaches.

BIBLIOGRAPHY

[1] *GME 5 Users Manual(v5.0)*, 2005. WebSite: http://www.isis.vanderbilt.edu/Projects/gme /GMEUMan.pdf.

[2] S. Abdelwahed, G. Karsai, and G. Biswas. Online safety control of a class of hybrid systems. *Decision and Control, 2002, Proceedings of the 41st IEEE Conference on*, 2:1988–1990 vol.2, Dec. 2002.

[3] Sherif Abdelwahed, Nagarajan Kandasamy, and Sandeep Neema. Online control for self-management in computing systems. In *10th IEEE Real-Time and Embedded Technology and Applications Symposium(RTAS'04)*, Toronto, Canada, May 2004.

[4] T. Abdelzaher, Ying Lu, Ronghua Zhang, and D. Henriksson. Practical application of control theory to web services. *American Control Conference, 2004. Proceedings of the 2004*, 3:1992–1997 vol.3, 30 June-2 July 2004.

[5] T. F. Abdelzaher, K. G. Shin, and N. Bhatti. Performance guarantees for web server end-systems: a control-theoretical approach. In *Parallel and Distributed Systems, IEEE Transactions on*, volume 13, pages 80–96, Jan 2002.

[6] T.F. Abdelzaher and N. Bhatti. Web server qos management by adaptive content delivery. In *Quality of Service, 1999. IWQoS '99. 1999 Seventh International Workshop on*, number 6375742 in 0-7803-5671-3, pages 216–225, 1999.

[7] F. Abdollahi and K. Khorasani. A robust dynamic routing strategy based on h control. *Control & Automation, 2007. MED '07. Mediterranean Conference on*, pages 1–6, 27-29 June 2007.

[8] Advanced Micro Devices Corp. *Mobile AMD-K6-2+ Processor Data Sheet*, publication 23446 edition, June 2000.

[9] Andrea Alimonda, Andrea Acquaviva, Salvatore Carta, and Alessandro Pisano. A control theoretic approach to run-time energy optimization of pipelined processing in mpsocs. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 876–877,

3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[10] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theor. Comput. Sci.*, 138(1):3–34, 1995.

[11] M. Arlitt and T. Jin. Workload characterization of the 1998 world cup web site. Technical report hpl-99-35r1, Hewlett-Packard Labs, Septermber 1999.

[12] Martin F. Arlitt and Carey L. Williamson. Web server workload characterization: the search for invariants. *SIGMETRICS Perform. Eval. Rev.*, 24(1):126–137, 1996.

[13] K. Astrom and T. Hagglund. *PID Controllers: Theory, Design, and Tuning*. Instrument Society of America, 2nd edition, 1995.

[14] Y. Bar-Shalom, R. Larson, and M. Grossberg. Application of stochastic control theory to resource allocation under uncertainty. *Automatic Control, IEEE Transactions on*, 19(1):1–7, Feb 1974.

[15] E. Bertolazzi, F. Biral, and M. Da Lio. Future advanced driver assistance systems based on optimal control: the influence of "risk functions" on overall system behavior and on prediction of dangerous situations. *Intelligent Vehicles Symposium, 2004 IEEE*, pages 386–391, 14-17 June 2004.

[16] R. Bisiani. *Encyclopedia of Artificial Intelligence*, pages 56–58. Beam search. Wiley & Sons, 1987.

[17] JC Bolot, T Turletti, and I Wakeman. Scalable feedback control for multicast video distribution in the internet. In *Proceedings of the conference on Communications architectures, protocols and applications*, pages 58–67, 1994.

[18] M. Bourne, M. Franco, and J. Wilkes. *Measuring Business Excellence*, volume 7, pages 15–21. Emerald Group Publishing Limited, 2003.

[19] G.P. Box, G.M. Jenkins, and G.C. Reinsel. *Time Series Analysis: Forecasting and Control*. Prentice-Hall, Upper Saddle River, New Jersey, 3 edition, 1994.

[20] K. Brammer and G. Siffling. *Kalman-Bucy Filters*. Norwood MA: Artec House, 1989.

[21] T.D. Burd and R.W. Brodersen. Energy efficient cmos microprocessor design. *System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on*, 1:288–297 vol.1, 3-6 Jan 1995.

[22] E.F. Camacho and C. Bordons. *Model Predictive Control, Advanced Textbooks in Control and Signal Processing*. Springer-Verlag, 2004.

[23] Tianyou Chai. A hybrid intelligent optimal control method for the whole production line and applications. *Integration Technology, 2007. ICIT '07. IEEE International Conference on*, pages nil14–nil15, 20-24 March 2007.

[24] A. Chandra, W. Gong, and P. Shenoy. Dynamic resource allocation for shared data centers using online measurements. 11th IEEE International Workshop on Quality of Service, June 2003.

[25] Qiang Chen and O.W.W. Yang. Design of aqm controller for ip routers based on h/sub /spl infin// s/u msp. *Communications, 2005. ICC 2005. 2005 IEEE International Conference on*, 1:340–344 Vol. 1, 16-20 May 2005.

[26] Xudong Chen, qingxin Zhu, Yong Liao, Ping Kuang, and Guangze Xiong. Dynamic optimal control for aperiodic soft real-time systems. *Communications, Circuits and Systems Proceedings, 2006 International Conference on*, 4:2796–2800, June 2006.

[27] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, 2nd edition, 2001.

[28] S.A. DeLurgio. *Forecasting Principles and Applications*. McGraw-Hill, 1998.

[29] Yixin Diao, Joseph L. Hellerstein, Sujay Parekh, Rean Griffith, Gail Kaiser, and Dan Phung. Self-managing systems: A control theory foundation. *ecbs*, 00:441–448, 2005.

[30] Yixin Diao and K.M. Passino. Stable fault-tolerant adaptive fuzzy/neural control for a turbine engine. *Control Systems Technology, IEEE Transactions on*, 9(3):494–509, May 2001.

[31] Yixin Diao and K.M. Passino. Adaptive neural/fuzzy control for interpolated nonlinear systems. *Fuzzy Systems, IEEE Transactions on*, 10(5):583–595, Oct 2002.

[32] C. Dovrolis, D. Stiliadis, and P. Ramanathan. Proportional differentiated services: Delay differentiation and packet scheduling. *ACM SIGCOMM Computer Communication Review*, 29(4):109–120, Oct. 1999.

[33] D. Menasce et al. In search of invariants for e-business workloads. In *Proc. ACM Conf. Electronic Commerce*, pages 56–65, 2000.

[34] D. Furcy and S. Koenig. Limited discrepancy beam search. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2005.

[35] A. G. Ganek and T. A. Corbi. The dawn of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, 2003.

[36] R. Griffith, J. Hellerstein, G. Kaiser, and Yixin Diao. Dynamic adaptation of temporal event correlation for qos management in distributed systems. *Quality of Service, 2006. IWQoS 2006. 14th IEEE International Workshop on*, pages 290–294, June 2006.

[37] Ning Gui, Chaoxin Wu, Songqiao Chen, and Jianxin Wang. A stable stateless fair bandwidth allocation algorithm using stochastic control. *Communications, Circuits and Systems Proceedings, 2006 International Conference on*, 3:1722–1726, 25-28 June 2006.

[38] F. Harada, T. Ushio, and Y. Nakamoto. Adaptive resource allocation control for fair qos management. *Transactions on Computers*, 56(3):344–357, March 2007.

[39] D. Henriksson, Y. Lu, and T. Abdelzaher. Improved prediction for web server delay control. *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, pages 61–68, 30 June-2 July 2004.

[40] C.V. Hollot, V. Misra, D. Towsley, and Wei-Bo Gong. A control theoretic analysis of red. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3 of *0-7803-7016-3*, pages 1510–1519, 2001.

[41] Intel Corp. *Enhanced Intel SpeedStep Tecnology for the Intel Pentium M Processor*, 2004.

[42] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, New York, 1991.

[43] N. Kandasamy and S. Abdelwahed. Designing self-managing distributed systems via online predictive control. Tech. report isis-03-404, Vanderbilt University, 2003.

[44] N. Kandasamy, S. Abdelwahed, and J.P. Hayes. Self-optimization in computer systems via on-line control: application to power management. *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 54–61, 17-18 May 2004.

[45] M. Karlsson. Maximizing the utility of a computer service using adaptive optimal control. *Networking, Sensing and Control, 2006. ICNSC '06. Proceedings of the 2006 IEEE International Conference on*, pages 89–94, 23-25 April 2006.

[46] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: performance isolation and differentiation for storage systems. *Quality of Service, 2004. IWQOS 2004. Twelfth IEEE International Workshop on*, pages 67–74, 7-9 June 2004.

[47] M. Karlsson, Xiaoyun Zhu, and C. Karamanolis. An adaptive optimal controller for non-intrusive performance differentiation in computing services. *Control and Automation, 2005. ICCA '05. International Conference on*, 2:709–714 Vol. 2, 26-29 June 2005.

[48] P.F. Kelly, A.K. Maulloo, and D.K.H. Tan. Rate control for communication networks: Shadow prices, proportional fairness and stability. *The Journal of the Operational Research Society*, 49(3):237–252, Mar. 1998.

[49] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Jan 2003.

[50] Minkyong Kim and Brian Noble. Mobile network estimation. In *Proceedings of the Seventh Annual International Conference on Mobile Computing and Networking*, pages 298–309, July 2001.

[51] L. Kleinrock. *Queueing Systems Theory*, volume 1. John Wiely & Sons, January 1975.

[52] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The generic modeling environment. In *WISP'*, Budapest, Hungary, May 24-25 2001.

[53] Bo Lincoln and Bo Bernhardsson. Optimal control over networks with long random delays. 2000.

[54] X. Liu, X. Zhu, S. Singhal, and M. Arlitt. Adaptive entitlement control of resource containers on shared servers. *Integrated Network Management, 2005. IM 2005. 2005 9th IFIP/IEEE International Symposium on*, pages 163–176, 15-19 May 2005.

[55] Xue Liu, Jin Heo, Lui Sha, and Xiaoyun Zhu. Adaptive control of multi-tiered web applications using queueing predictor. *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*, pages 106–114, 2006.

[56] Xue Liu, Lui Sha, Yixin Diao, steven Froehlich, Joseph L. Hellerstein, and Sujay Parekh. *Quality of Service - IWQoS 2003*, chapter Online Response Time Optimization of Apache Web Server, page 153. Springer Berlin / Heidelberg, 2003.

[57] C. Lu, J. Stankovic, G. Tao, and S. Son. Feedback control real-time scheduling: Framework, modeling, and algorithms. *J. Real-Time Syst.*, 23(1-2):85–126, July/September 2002.

[58] C. Lu, J. A. Stankovic, T. F. Abdelzaher, G. Tao, S. H. Son, and M. Marley. Performance specifications and metrics for adaptive real-time systems. In *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*, pages 13–23, Orlando, FL, USA, 2000.

[59] C Lu, JA Stankovic, G Tao, and SH Son. Design and evaluation of a feedback control edf scheduling algorithm. In *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, 0-7695-0475-2, pages 56–67, 1999.

[60] Chenyang Lu, Tarek F. Abdelzaher, John A. Stankovic, and Sang H. Son. A feedback control approach for guaranteeing relative delays in web servers. In *Real-Time Technology and Applications Symposium, 2001. Proceedings. Seventh IEEE*, pages 51–62, 2001.

[61] Chenyang Lu, Ying Lu, T.F. Abdelzaher, J.A. Stankovic, and Sang Hyuk Son. Feedback control architecture and design methodology for service delay guarantees in web servers. *Transactions on Parallel and Distributed Systems*, 17(9):1014–1027, Sept. 2006.

[62] Chenyang Lu, John A. Stankovic, Sang H. Son, and Gang Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms*. *Real-Time Syst.*, 2006.

[63] Chenyang Lu, Xiaorui Wang, and Xenofon Koutsoukos. Feedback utilization control in distributed real-time systems with end-to-end tasks. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):550–561, 2005.

[64] Y. Lu, T. Abdelzaher, Chenyang Lu, Lui Sha, and Xue Liu. Feedback control with queueing-theoretic prediction for relative delay guarantees in web servers. *Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. The 9th IEEE*, pages 208–217, 27-30 May 2003.

[65] Ying Lu, T. Abdelzaher, Chenyang Lu, and Gang Tao. An adaptive control framework for qos guarantees and its application to differentiated caching. *Quality of Service, 2002. Tenth IEEE International Workshop on*, pages 23–32, 2002.

[66] Ying Lu, A. Saxena, and T.F. Abdelzaher. Differentiated caching services; a control-theoretical approach. In *Distributed Computing Systems, 2001. 21st International Conference on.*, 0-7695-1077-9, pages 615–622, Apr. 2001.

[67] Zhijian Lu, Jason Hein, Marty Humphrey, Mircea Stan, John Lach, and Kevin Skadron. Control-theoretic dynamic frequency and voltage scaling for multimedia workloads. In *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 156–163, New York, NY, USA, 2002. ACM.

[68] A.K. Moharana, K. Panigrahi, B.K. Panigrahi, and P.K. Dash. Vsc based hvdc system for passive network with fuzzy controller. *Power Electronics, Drives and Energy Systems, 2006. PEDES '06. International Conference on*, pages 1–4, 12-15 Dec. 2006.

[69] T. Mudge. Power: a first-class architectural design constraint. *Computer*, 34(4):52–58, Apr 2001.

[70] Sujata Mujumdar, Nagabhushan Mahadevan, Sandeep Neema, and Sherif Abdelwahed. A model-based design framework to achieve end-to-end qos management. In *ACM-SE 43: Proceedings of the 43rd annual Southeast regional conference*, pages 176–181, New York, NY, USA, 2005. ACM.

[71] J. Le Ny, M. Dahleh, and E. Feron. Multi-agent task assignment in the bandit framework. *Decision and Control, 2006 45th IEEE Conference on*, pages 5281–5286, 13-15 Dec. 2006.

[72] K. Ogata. *Modern Control Engineering*. Prentice Hall, Englewood Cliffs, NJ, 1997.

[73] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance management. *J. Real-Time Syst.*, 23(1-2):127–141, July/September 2002.

[74] P.-F. Quet and H. Ozbay. On the design of aqm supporting tcp flows using robust control theory. *Automatic Control, IEEE Transactions on*, 49(6):1031–1036, June 2004.

[75] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, NJ, 2nd edition, 2003.

[76] Lui Sha, Xue Liu, Ying Lu, and T. Abdelzaher. Queueing model based network server performance control. *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, pages 81–90, 2002.

[77] A. Shukla, A. Ghosh, and A. Joshi. State feedback control of multilevel inverters for dstatcom applications. *Power Delivery, IEEE Transactions on*, 22(4):2409–2418, Oct. 2007.

[78] A. Sinha and A.P. Chandrakasan. Energy efficient real-time scheduling [microprocessors]. *Computer Aided Design, 2001. ICCAD 2001. IEEE/ACM International Conference on*, pages 458–463, 2001.

[79] M.B. Srivastava, A.P. Chandrakasan, and R.W. Brodersen. Predictive system shutdown and other architectural techniques for energy efficient programmable computation. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 4(1):42–55, Mar 1996.

[80] JA Stankovic, C Lu, SH Son, and G Tao. The case for feedback control real-time scheduling. In *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on*, 0-7695-0240-7, pages 11–20, 1999.

[81] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the third symposium on Operating systems design and implementation*, 1-880446-39-1, pages 145–158. USENIX Association, Berkeley, CA, USA, 1999.

[82] M. Sugeno and T. Yasukawa. A fuzzy-logic-based approach to qualitative modeling. *Fuzzy Systems, IEEE Transactions on*, 1(1):7–, Feb 1993.

[83] A. Talukder, R. Bhatt, T. Sheikh, R. Pidva, L. Chandramouli, and S. Monacos. Dynamic control and power management algorithm for continuous wireless monitoring in sensor networks. *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 498–505, 16-18 Nov. 2004.

[84] S. Thavamani. Control of c2 unit using arena modeling and simulation. *Simulation Conference, 2006. WSC 06. Proceedings of the Winter*, pages 1316–1323, 3-6 Dec. 2006.

[85] Wanqing Tu, Cormac J. Sreenan, and Weijia Ji. Worst-case delay control in multigroup overlay networks. *Transactions on Parallel and Distributed Systems*, 18(10):1407–1419, Oct. 2007.

[86] Xiaorui Wang, Yingming Chen, Chenyang Lu, and Xenofon Koutsoukos. On controllability and feasibility of utilization control in distributed real-time systems. *Real-Time Systems, 2007. ECRTS '07. 19th Euromicro Conference on*, pages 103–112, 4-6 July 2007.

[87] Linbo Xie, Weiyi Zhao, and Zhicheng Ji. Lqg control of networked control system with long time delays using $\delta-$operator. *Intelligent Systems Design and Applications, 2006. ISDA '06. Sixth International Conference on*, 2:183–187, Oct. 2006.

[88] Jing Xu, Ming Zhao, Jose Fortes, Robert Carpenter, and Mazin Yousif. On the use of fuzzy modeling in virtualized data center management. *Autonomic Computing, 2007. ICAC '07. Fourth International Conference on*, pages 25–25, 11-15 June 2007.

[89] Wei Xu, Xiaoyun Zhu, S. Singhal, and Zhikui Wang. Predictive control for dynamic resource allocation in enterprise data centers. *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*, pages 115–126.

[90] L. A. Zadeh. "fuzzy sets". *Information and Control*, 8:338–353, 1965.

[91] M. Zafer and E. Modiano. Minimum energy transmission over a wireless fading channel with packet deadlines. *Decision and Control, 2007 46th IEEE Conference on*, pages 1148–1155, 12-14 Dec. 2007.

[92] M. Zafer and E. Modiano. Delay-constrained energy efficient data transmission over a wireless fading channel. *Information Theory and Applications Workshop, 2007*, pages 289–298, Jan. 29 2007-Feb. 2 2007.

[93] Ronghua Zhang, Chenyang Lu, T.F. Abdelzaher, and J.A. Stankovic. Controlware: a middleware architecture for feedback control of software performance. *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 301–310, 2002.