METE: Meeting End-to-End QoS in Multicores through System-Wide Resource Management

Akbar Sharifi, Shekhar Srikantaiah, Asit K. Mishra, Mahmut Kandemir and Chita R. Das
Department of CSE
The Pennsylvania State University
University Park, PA 16802, USA
{akbar, srikanta, amishra, kandemir, das}@cse.psu.edu

ABSTRACT

Management of shared resources in emerging multicores for achieving predictable performance has received considerable attention in recent times. In general, almost all these approaches attempt to guarantee a certain level of performance QoS (weighted IPC, harmonic speedup, etc) by managing a single shared resource or at most a couple of interacting resources. A fundamental shortcoming of these approaches is the lack of coordination between these shared resources to satisfy a system level QoS. This is undesirable because providing end-to-end QoS in future multicores is essential for supporting wide-spread adoption of these architectures in virtualized servers and cloud computing systems. An initial step towards such an end-to-end QoS support in multicores is to ensure that at least the major computational and memory resources on-chip are managed efficiently in a coordinated fashion.

In this paper, we propose METE, a platform for end-to-end on-chip resource management in multicore processors. Assuming that each application specifies its performance target/SLA, the main objective of METE is to dynamically provision sufficient on-chip resources to applications for achieving the specified targets. METE employs a feedback based system, designed as a Single-Input, Multiple-Output (SIMO) controller with an Auto-Regressive-Moving-Average (ARMA) model, to capture the behaviors of different applications. We evaluate a specific implementation of METE that manages cores, shared caches and off-chip bandwidth in an integrated manner on 8 and 16 core systems using a detailed full system simulator and workloads derived from the SPECOMP and SPECJBB multithreaded benchmarks. The collected results indicate that our proposed scheme is able to provision shared resources among co-runner applications dynamically over the course of execution, to provide end-to-end QoS and satisfy specified performance targets. Furthermore, the elegance of the control theory based multi-layer resource provisioning is in assuring QoS guarantees.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. ACM reserves the right to exceed permissions above in some cases. 

 SIGMETRICS’11, June 7–11, 2011, San Jose, California, USA. 
Copyright 2011 ACM 978-1-4503-0262-3/11/06 . . .$10.00.

Categories and Subject Descriptors
B.3.2 [Memory Structures]: Design Styles; C.4 [Performance of Systems]: Design Studies

General Terms
Design, Experimentation, Management, Performance

Keywords
Resource Management, Control Theory, End-to-End QoS

1. INTRODUCTION

Multicores are now ubiquitous [1, 2, 3, 4], owing to the benefits they bring over single core architectures including improved performance, lower power consumption and reduced design complexity. Several resources ranging from the cores themselves to multiple levels of on-chip caches and off-chip memory bandwidth are typically shared in a multicore processor. Prudent management of these shared resources for achieving predictable performance and optimizing energy efficiency is critical and thus, has received considerable attention in recent times. Most of this research has focussed around managing either the shared cache [5, 6, 7, 8, 9, 10, 11] or off-chip memory bandwidth [12, 13, 14, 15, 16] in isolation.

In general, almost all these approaches attempt to guarantee a certain level of quality-of-service (QoS) like weighted IPC, harmonic speedup, etc by managing a single shared resource or at most a couple of interacting resources. There are at least three fundamental shortcomings with such approaches: (i) Considering that system performance is heavily influenced by complex interactions among multiple resources [17, 18], attempting to optimize performance/guarantee QoS by managing a single resource is not only less effective, but may also be impossible; (ii) In most existing schemes, there is no feedback among mechanisms, trying to provide QoS by managing different resources, and this leads to an anarchy in resource management; and (iii) Working with low-level resources like cache or memory-bandwidth restricts the system-performance metrics that can be controlled to only low-level metrics like IPC, which are not easily comprehended by system administrators or application programmers.

On the other hand, providing end-to-end QoS in future multicores is essential for supporting wide-spread adoption of these new architectures in virtualized servers and cloud computing systems. An initial step towards such an end-to-end QoS support in multicores is to ensure that at least the major computational and memory resources on-chip are managed efficiently in a coordinated fashion. In this paper, we propose METE, a platform for end-to-end on-chip resource management in multicore processors. The proposed resource management scheme attempts to address all the three major constraints of existing techniques by (i) providing a multi-level resource provisioning mechanism for end-to-end QoS, (ii) developing
Figure 1: Three types of shared resources provisioned to two co-runner applications.

a control theoretic model for accurately tracking the system state, and (iii) by demonstrating the applicability of the model to system level parameters. While a few recent works [19, 20] have studied the advantages of using feedback control theory for resource management in multicore processors, to our knowledge, no prior study has taken a holistic multi-level control theory approach as proposed here.

Figure 1 shows the high-level view of the three resources managed by a specific implementation of METE (cores, on-chip shared caches and off-chip bandwidth) in an integrated manner among two applications. The main goal behind this management is to ensure that any performance target is tracked by provisioning resources in an end-to-end manner. Assuming that each application specifies a (potentially different) performance target, the main objective of METE is to dynamically provision sufficient on-chip resources to applications in order to achieve the specified targets, if it is possible to do so. Apart from low-level metrics like IPC (instructions per cycle) that have been used in the past, METE also accepts high-level (e.g., application specific) performance metrics like transactions per second for database applications. More importantly, the proposed system is flexible enough to handle different metrics for different applications at the same time.

METE employs a feedback based system designed as a Single-Input, Multiple-Output (SIMO) controller with an Auto-Regressive-Moving-Average (ARMA) model to capture the dynamic behaviors of different applications. Control theory [21, 22] is a powerful tool that offers several unique advantages over alternate schemes:

- Feedback control theory provides a robust strategy to track specified objectives over time. It can achieve this by modulating resource allocations in a coordinated fashion.
- A system equipped with a feedback controller can respond quickly to variations in the dynamic behaviors of running applications.
- A feedback controller can be designed to control multiple high-level metrics of interest at the same time under various constraints.
- Using control theory enables rejecting unexpected disturbances in the controlled system. For instance, in the context of multicore, a sudden change in the demands for a shared resource can be interpreted as an external disturbance in the system.

We evaluate METE on 8 and 16 core systems using a detailed full system simulator and workloads formed from the SPECCOMP and SPECJBB multithreaded applications. The collected results indicate that our proposed scheme is able to provision shared resources among co-runner applications dynamically over the course of execution, to provide end-to-end QoS and satisfy specified high-level performance targets.

The remainder of this paper is structured as follows. The next section discusses background on feedback control theory, and motivates our solution. Section 3 gives the mechanisms employed to partition each type of resource we target in this work across applications. Section 4 presents our proposed METE platform that enables QoS-aware multi-resource management. Our experimental evaluation is presented in Section 5. Section 6 discusses the related studies and finally, we conclude the paper in Section 7.

2. MOTIVATION AND BACKGROUND

2.1 Motivation

The allocated number of cores, amount of on-chip shared cache space and off-chip memory bandwidth are three major parameters that affect the performance of an application running on a multicore machine. However, the degrees to which each of these parameters influences the performance are not the same, both within an application’s execution (i.e., across its different phases) as well as across different applications. To illustrate this point, we performed a set of experiments that evaluates the impact of different shared resources on the performance of applications. Two applications, appl1 and swim from the SPEC benchmark suite [23], are used in these experiments. Unless otherwise mentioned, each of these applications is assigned 4 cores, 4 MB, 16 way associative shared on-chip cache space, and 6.4 Gb/s bandwidth. Then, each of the resource allocations, one at a time, is changed to study the impact the resource has on each application’s performance. Figures 2(a), (b) and (c) plot the performance (measured in IPC) as the amount of resources (cache ways, percentage bandwidth, and cores) allocated to the applications varies. In all experiments, each application has 8 threads. The detailed simulation parameters and system configuration used in this work are given later in Section 5.1.

Several observations can be made from these plots. First, the overall impact of varying the amount of each type of resource is significantly different from the others. For example, the number of allocated processing cores has much larger impact on the performance as compared to the amount of off-chip bandwidth allocated to the running applications. Therefore, while it is important to consider all resources in the system to provide end-to-end QoS, it is also crucial to consider the differences in their impacts on application performance. Second, for a given change in a resource, different applications react differently, based on the operating point. For example, for a bandwidth allocation change from 5% to 10%, swim receives a bigger boost in performance than appl1. That is, the slope of the performance vs. bandwidth allocation curve for swim is greater than that for appl1 in the 5-10% operating region. However, the corresponding slopes are similar in the 50-100% operating range. Therefore, it is important to consider the operating point of each resource for each application. Third, it is possible to track the same target performance for the same application by allocating different combinations of resources. For example, appl1 is able to achieve an IPC of 1.2 by using either 4 cache ways and 50% of peak bandwidth (Figure 2(a)), or 10% of the peak bandwidth and 16 cache ways (Figure 2(b)). On the other hand, when some of the resources are constrained, we may not have flexibility in resource allocations to track a specified target performance. For example, once swim is constrained to an allocation of 4 cores and an allocation of 50% bandwidth, it cannot achieve an IPC of 1.7 even with an allocation of 32 cache ways, while the same can be achieved by allocating 4 cores, 16 cache ways and 100% bandwidth. Therefore,

Note that, in a more bandwidth constrained system the significance of bandwidth may be reversed.
it is important to consider the allocations across all resources in unison (end-to-end), such that all performance targets can be met.

A recent work [19] elaborated on the advantages of using formal feedback control and modeling application performance to achieve performance QoS by partitioning only the shared cache (assuming that the other resource allocations do not change). The customized oscillation resistant controller proposed in that work is essentially a Single-Input, Single-Output feedback controller that can track the performance of an application by performing shared cache partitioning. As a preliminary study, we extended this design based on the same principles to design three controllers, one for each type of resource (core, cache and bandwidth). Each of the three controllers takes the same performance QoS target as input (corresponding to an IPC of 1.25) and seeks to satisfy this goal in isolation. The observed performance of one of our applications (applu) with such a system is shown in Figure 3. We can see that, the specified QoS target is often violated by this system. One of the major reasons for this behavior is the lack of coordination between the three controllers. Specifically, each of the three controllers decides the resource allocations in their layer without considering the impact that the other may have on the performance. This can often lead to conflicting decisions in different resource allocations [17, 18]. For example, at time $t = 5$, 5 cores, 16 cache ways and 80% of the off-chip memory bandwidth are allocated to the running application. Since the measured performance is higher than the target, the amount of allocated cache space and memory bandwidth are reduced to 7 and 60%, respectively. As a result, the measured IPC decreases to 1.2, which is lower than the performance goal. This simple experiment clearly motivates the need to have a coordinated end-to-end feedback controller that takes the specified QoS target as input, but simultaneously controls all the resources based on modeling application behavior.

2.2 Background

In this work, we employ Single-Input, Multiple-Output (SIMO) controller. Figure 4 illustrates a canonical feedback control loop with a SIMO controller. As an example for SIMO controller, consider a water pool (plant) that has to be maintained at a constant temperature by letting in both hot and cold water flows run into it (and correspondingly let equal volume of water out of the pool). Suppose that the rates of the flows can be controlled by a controller and the controller’s role is to adjust these rates in an automated fashion to keep the pool temperature at the desired value. The desired temperature is called Reference Input in the control theory terminology. Since in this case, the controller takes a desired pool temperature as a single input and controls it using multiple outputs (specifically two: hot and cold water), the controller is a SIMO controller. The controller functions by comparing the reference input to the current water temperature (System Output) and based on the obtained Error value, the rates of the hot and cold flows are modulated. The Transducer converts the System Output (pool temperature) to the same type as the Reference Input if they have different types and cannot be compared. This component aids in implementing a flexible system with different high-level target specifications by converting the system output into a comparable metric to the target specification. We will discuss the implementation of our transducer component in Section 5.5.

In METE, a separate SIMO controller is assigned for each application. At the end of each time interval (sampling period), the controller increases/decreases the control inputs (resource allocation), taking into account the variation observed in the measured IPC value (Observable System Output) over the last time interval. Note that the controller requires knowledge on the reaction of the application to the modulations in the resource allocations. A system model in control theory tries to capture this knowledge. Table 1 summarizes

Figure 2: The impacts of resource allocation on the performance of two running applications.

Figure 3: Behavior of applu (one of our applications) in a system consisting of three controllers (one for each resource). Oscillation occurs, since at time $t$ the cache layer controller increases/decreases the allocated cache space to meet the performance target unaware of the changes made by the off-chip bandwidth controller to the memory bandwidth allocation.

Figure 4: High level view of a feedback control system with a SIMO controller.
Table 1: Basic terms used in control theory and their descriptions in our problem domain.

<table>
<thead>
<tr>
<th>Term</th>
<th>Description in Our Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference Input</td>
<td>Desired IPC value for an application</td>
</tr>
<tr>
<td>Control Input</td>
<td>Resource allocation</td>
</tr>
<tr>
<td>System</td>
<td>multicore with running applications</td>
</tr>
<tr>
<td>System Output</td>
<td>Measured IPC value of an application</td>
</tr>
<tr>
<td>Controller</td>
<td>Application Controller</td>
</tr>
<tr>
<td>System Model</td>
<td>the employed ARMA model</td>
</tr>
</tbody>
</table>

the basic terms used in formal control theory and the corresponding descriptions in our problem domain. We want to emphasize that our scheme can work with any performance QoS. In most of our discussion however, we use IPC as our target metric.

If the values of the model parameters do not change over time, we refer to this type of model as static. In adaptive feedback control systems on the other hand, the system model is updated dynamically. In this work, we employ the latter type since behaviors of applications do not remain constant during the course of execution and each usually has multiple execution phases. Specifically, in METE, we employ an Auto-Regressive Moving Average (ARMA) [22] model. The ARMA model can approximate the behavior of a system with multiple inputs/outputs in a linear form and also can be updated dynamically, suitable for adaptive control system designs. We also employ a global controller (manager), called the Resource Broker, to handle cases where (i) requests by resource controllers (control knobs) are oversubscribed, and (ii) after satisfying all requests, there are still idle resources. In the following sub-sections, we study our proposed control design for end-to-end QoS management in multi-cores.

3. IMPLEMENTATION ISSUES

Figure 1 gives the high-level view of how the shared resources are partitioned between two applications. We envision implementing METE in the operating system (OS), with dynamic feedback control over the course of execution. The cache and bandwidth partitioning can also be handled by the OS with hardware support from the shared resources. In the following discussion, we explain the mechanisms METE employs to partition each type of resource across applications.

Core Partitioning: METE uses the psrset utility in Solaris to create a processor set containing one or more cores and run a particular application on it. Cores can be added or removed from that processor set over execution. Consequently, the OS provides a mechanism to adjust the number of cores allocated to each running application dynamically.

Cache Partitioning: In METE, we adopt the OS-level mechanism proposed in [24] to partition the shared cache space across co-runner applications or threads. In [24], the hardware part of the proposed scheme guarantees that the quotas specified by the OS are enforced in shared caches. In this scheme, an m-bit tag is associated with each cache block indicating which core that block belongs to. Also, each memory request contains an identifier indicating its cache block access domain.

Off-Chip Bandwidth Partitioning: In METE, the available memory bandwidth is partitioned across co-runner applications based on the priority of each sharer. This is similar to the fair queuing systems proposed in [25, 26]. For example, as can be seen in Figure 1, the memory controller serves the requests from the two applications in such a way that each of them receives its quota of the off-chip memory bandwidth based on its relative weight specified by the OS.

To implement this scheme, a service time (which is inversely related to its weight) is computed for each application request flow. The application with smaller service time (as compared to the memory controller’s virtual clock) is served at each time.

After explaining these actuators (control knobs), we next discuss the details of our control architecture.

4. CONTROL ARCHITECTURE

4.1 High Level View

Figure 5 illustrates the high level view of METE. Each Application Controller shown in Figure 5 is implemented in software. Assuming that each running application is assigned a specified IPC target (desired performance goal, \( ref_i \)), the application controller of application \( a_i \) determines the amount of resources, \( c_{ik} \), that need to be allocated for that application in order to satisfy the specified performance goal (\( IPC_{ik} \)) at the k-th execution epoch (Table 2 gives the notation we use in this paper).

As mentioned earlier, in our current implementation of METE, three types of resources are targeted and can be partitioned among co-runner applications. These resources include processing cores, shared L2 cache space, and off-chip memory bandwidth. The amount of resources in each type is limited, and consequently, at each epoch, if the total amount of resources requested by applications is less than or equal to available total, a successful resource allocation can be performed. Otherwise (if the three resource constraints shown below are not satisfied), a higher level module (called Resource Broker in this work) intervenes to modify the amount of requested resources determined by the application controllers \( u_{ik} \) and make the final resource partitioning decision \( u_{ik} \). Our resource constraints can be expressed as follows:

\[
\sum_{a_i \in A} c_{ik} \leq C, \quad \sum_{a_i \in A} w_{ik} \leq W, \quad \text{and} \quad \sum_{a_i \in A} m_{ik} \leq M.
\]

The above constraints ensure that the sum of the allocated cores,
cache ways and memory bandwidth cannot exceed the total available resources. Below, we first study our employed model and the feedback controller associated with it, and then present the details of the resource broker module.

4.2 System Model

As mentioned earlier, a feedback controller needs to know the impact of variations in control parameters on the measured system outputs. In other words, a system model is a mathematical function \( f \) that gives the system output for every feasible control parameters (i.e., \( y(k) = f[u(k)] \), where \( y \) and \( u \) are the output and input of the system, respectively). One way of obtaining the system model is to determine function \( f \) directly by mathematical analysis of how the system works. Unfortunately, this approach cannot be applied to most actual computing systems due to the complexity of the desired modeling function. Instead, a more practical approach would be estimating the system behavior through analysis of the sample measured data. In this method, the model is expressed as a parametric function of independent variables with a finite number of parameters. In this case, system model identification is the process of determining these parameters. Inputs with different values are fed to the system and the corresponding measured outputs are collected. Algorithms such as least square [22] can be employed in this step to determine the values of the model parameters.

Even though the behavior of most real systems is not linear, linear approximation can be used (and has been successfully used in prior works and actual control based systems) as a method to estimate non-linear characteristics of actual systems. In METE, we employ the ARMA model [27] to capture the effect of the control parameters (\( a_{ik} \)) on our system output (IPC\(_{ik}\)). The ARMA model is a linear regression that can be updated dynamically and is very suitable for adaptive control system designs. The equation below gives the mathematical representation of our employed model for each application:

\[
IPC_{ik}(k+1) = a_{ik} \times IPC_{ik}(k) + b_{ik}^{T} \times \Delta u_{k}(k), \tag{2}
\]

where \( a_{ik} \) and \( b_{ik}^{T} \) are the parameters of the model that are dynamically determined for each application. Note that \( IPC_{ik}(k) \) and \( \Delta u_{k}(k) \) correspond to the actual IPC of application \( a_{i} \) and the variations in the resource allocations at the \( k \)-th time interval, respectively. In this equation, \( b_{ik}^{T} = \begin{bmatrix} c & w & m \end{bmatrix} \), where the values of \( c, w \) and \( m \) capture the influence of the variations in the number of allocated cores, the number of cache ways and the reserved memory bandwidth on the application IPC value. As we later show in the experimental results section, the value of \( c \) for different applications is significantly larger than the value of \( w \), meaning that, if one core is added to the set of cores allocated to an application, the application’s IPC will improve much better as compared to the case in which one extra cache way is added to the previously reserved cache ways for that application. The values of our model parameters are determined in this work dynamically using the recursive least square algorithm [22]. In this algorithm, at each time interval, the model is updated based on the new operating point that has been obtained (i.e., by considering the performance impact of the last resource allocation). To evaluate the obtained model, the IPCs predicted by the model can be compared against the actual ones, while the inputs are taking different values. We present the results from our model evaluation in Section 5.

4.3 Application Controller

The State Space approach is a compact and convenient method to model, analyze and design a wide range of systems [21], especially systems with time-varying characteristics and multiple inputs/outputs. To use this approach, first, the system behavior has to be represented using a State Space Model. The following equations characterize the general form of a state space model (as illustrated in Figure 6):

\[
\begin{align*}
x(k + 1) &= A x(k) + B u(k) \tag{3} \\
y(k) &= C x(k), \tag{4}
\end{align*}
\]

where \( x \) is the state vector, \( u \) and \( y \) are, respectively, the input and the output vectors of the system, and \( A, B \) and \( C \) are the model parameters. The state vector \( x \) contains the state variables that reflect the current state of the system. Equation (3) is solved to predict the next state of the system based on the current state and input vector \( u \). The input vector contains the control input parameters. Equa-

Figure 5: High level view of our control architecture for \( n \) number of applications.

Figure 6: Input, output and state in a state space model.

Figure 7: Detailed view of an application controller.
in Figure 7. Our goal is to determine the values of \( K \) and \( N \) in such a way that the stability of the applications around the targets is ensured. Note that, \( K \) and \( N \) are two vectors with three elements. *pole placement and linear quadric regulation* [28] are two algorithms that can be employed to determine the values of \( K \) and \( N \). In the pole placement strategy, which is the method employed in this work, first, the values of \( K \) are determined in such a way that the poles of the system ensure the stability of the system. After that, the contents of \( N \) are obtained using the method outlined in [27].

The final resource demands (\( u_i(k) \)) of application \( i \) (\( a_i \)) are determined by adding the amount of decreasing/increasing values of the control parameters (\( \Delta u_i(k) \)) and the application resource allocation at the (k-1)th time interval (\( \hat{u}_i(k-1) \)), as given in Equation (6):

\[
\text{Equation (5)}, \ \text{the pole of each application controller's transfer function replacing the parameters in Equation (7) with the values given in Equation (8). By}\ 
\]

\[
u_i(k) = N \times \text{ref}_i - k \times IPC_i(k-1) + \hat{u}_i(k-1). \tag{6}
\]

4.4 Stability Guarantees

Stability is one of the most important properties of a feedback control based system [27]. If a system equipped with a feedback controller is stable, the measured output of the system converges to the desired target over time. For instance, the system with the output shown in Figure 3 is not stable, since the controller is not able to adjust the values of the system inputs in such a way that the system output converges to the desired target over time.

Different properties of a control system can be analyzed easily when the system is described in frequency domain (\( \z \)-domain). Considering Figure 6 and assuming \( U(z) \) and \( Y(z) \) are the \( \z \)-domain representations of \( u(k) \) and \( y(k) \) respectively, the value of \( u(k) \) at time \( k = k_0 \) is the coefficient of \( z^{-k_0} \) in \( U(z) \). In other words, the values of \( u(k) \) for different \( k \)s are encoded as the coefficients of \( \z \) terms in \( U(z) \). Further, the transfer function of the system is defined as \( G(z) = \frac{Y(z)}{U(z)} \) and indicates how an input \( U(z) \) is transformed into the output \( Y(z) \). In our multicore system, the transfer function can be determined by taking \( \z \)-transform of all terms in Equation 2 which describes the behavior of our system.

The *Stability Theorem* in the formal control theory states that a system represented by a transfer function \( G(z) \) is stable if and only if the roots (roots of the dominator polynomial) of \( G(z) \) are within the unit circle in the complex coordinate plane [27]. Therefore, in METE, to ensure the stability of the system, we first need to find the poles of each application controller and then determine the parameters of the controller in such a way that the poles are placed within the unit circle. As shown in [27], if a system is described by a state space model as in Equation (3), the poles of the system can be obtained solving the following equation in terms of \( z \):

\[
det[zI - (A - BK)] = 0, \tag{7}
\]

where \( A \) and \( B \) are the model parameters in Equation (3), and \( K \) is the coefficient vector in the feedback path (see Figure 7). By replacing the parameters in Equation (7) with the values given in Equation (5), the pole of each application controller’s transfer function would be:

\[
z = a_{ik} - c \times k_1 - w \times k_2 - m \times k_3, \tag{8}
\]

assuming that, in Equation (7), \( B \) is \( \begin{pmatrix} c \ w \ m \end{pmatrix} \) and \( K \) is a vector with three elements, \( k_1, k_2 \) and \( k_3 \). Consequently, to ensure stability of our system, the \( K \) values are determined such that \( |z| < 1 \) in Equation (8).

4.5 The Resource Broker

So far, we have discussed how an application controller determines the amount of different types of resources (processing cores, L2 cache ways, and off chip memory bandwidth) an application needs to satisfy its specified performance QoS (\( ref_i \)). However, since each of our application controllers operates independently, there can be cases where (i) the available resources are not sufficient to meet the requirements of all of the running applications or (ii) the available resources exceed the requirements of the entire workload. In METE, these cases are handled by the resource broker component.

**Lack of Resources:** Note that, resource contention may occur in any of the resource types (i.e., when one or more than one of the constraints in Expression (1) given earlier are not satisfied). In this case, the resource broker is responsible for performing a best effort allocation by considering the relative resource demands of the co-runner applications. As an example, suppose that \( Q1 \) and \( Q2 \) are the performance targets of two co-runner applications (\( a_1 \) and \( a_2 \)), and the available resources are not sufficient to satisfy both of the specified targets. Assume further that the amount of resources required to satisfy \( Q1 \) (determined by the application controller) is much larger than the resource demands of the other application to achieve \( Q2 \). In this situation, it would not be a fair policy to penalize both applications equally in an attempt to compensate for the lack of resources, since such policy may degrade the performance of the application with lower resource requirement significantly.

Without loss of generality, let us focus now on core allocation. Assuming that the total demands of applications is greater than the total available number of cores, one can observe that \( \delta \) cores have to be spilled, where \( \delta = \sum_{a \in A} c_{ik} - C \). The approach that we adopt in this work (i.e. our default policy) is to distribute \( \delta \) among the applications in proportion to their demands. The goal here is to distribute the penalty across applications in a fair manner. Consequently, in this case, the core demands \( c_{ik} \) are modified as:

\[
c_{ik} = \text{floor} \left( c_{ik} \left( 1 - \frac{\delta}{\sum_{a \in A} c_{ik}} \right) \right). \tag{9}
\]

A similar strategy can be adopted when the contention occurs in other types of resources as well (shared L2 cache ways and off chip memory bandwidth).

**Excess resources:** If sum of the resources required by co-runner applications is less than the available amount, one has different options. An energy-aware option would be turning off the excess resources (or placing them into a low-power operating mode) if the underlying hardware supports that. A performance-centric option (our default policy) on the other hand would proceed as follows. To extract additional performance from the available resources, we may be able to use the application priorities assigned to applications by system administrators. Assuming that each application in a workload has been assigned a priority/weight, \( (w_i) \) indicating its relative importance, \( \delta \) excess resources can then be distributed across applications based on Expression (10) given below:

\[
c_{ik} = \left( c_{ik} + \frac{\delta \times w_i}{\sum_{a \in A} w_i} \right), \quad \text{if } c_{ik} < 1 \rightarrow c_{ik} = 1. \tag{10}
\]

Note that, these priorities can be determined based on the values of vector \( b_{ik} \) in Equation (2). As mentioned earlier, these values capture the influence of the variations in the number of allocated cores, the number of cache ways and the reserved memory bandwidth on the IPC value of application \( i \). Consequently, if a change in the allocation of the contended resource has a larger impact on the performance of application \( i \), this application would receive more (excess) resources than the other applications to maximize performance benefits.
5. EXPERIMENTS

In this section, we present a detailed experimental evaluation of METE.

5.1 Benchmarks and Setup

We used the SPECOMP benchmark suite [23] to evaluate our control scheme. In addition, we also performed experiments with the SPECJBB benchmark [29] in order to evaluate the use of high-level QoS with METE. Our implementation and evaluations are carried out using SIMICS [30], which is a full system simulator that allows simulation of multi-processor systems [30]. Table 3 gives the baseline configuration used in our experimental evaluation.

Our default configuration contains 8 cores. Each core has a private L1 cache and the on-chip L2 cache is shared by all cores. In most of our experiments, we formed our workloads using applications from the SPECOMP benchmark suite. Each workload is composed of two SPECOMP applications. Table 4 shows different mixes (workloads) we consider in most of our experimental evaluation. The SPECOMP benchmark applications are multithreaded programs and the number of running threads can be specified before the execution starts. In our experiments, each of these programs is executed using 8 threads and, therefore, one to eight cores can be allocated to each application by our proposed control centric scheme (1 ≤ τi ≤ 8), since it is not beneficial from the performance and utilization perspectives to allocate more than eight cores to an application with eight running threads (note that, the number of threads does not change during execution when using METE). In other words, the number of threads of a running application indicates the maximum number of cores that can be allocated to it.

To run a workload that consists of two applications in our 8-core multicore machine, first, two processor sets (one core in each) are created using the psrset utility provided by the Solaris OS, and the minimum number of L2 cache ways (one way) and the minimum amount of memory bandwidth (5% of maximum available bandwidth) are allocated to each set. Each application in the workload is executed on one of the processor sets. At the end of each epoch, METE determines the resource allocations for the next time interval, and all three types of resources are partitioned among the co-runner applications based on that. If, for example, the new partitioning of the cores suggests an increase in the number of allocated cores to the first application by two, two cores are added to the processor set the first application belongs to. The default sampling period (epoch) to enforce our control decisions is set to 20 million cycles. In our sensitivity experiments, we study the impact of varying this default value.

Further, there are three other parameters that need to be specified in our evaluation: (i) Reference IPC: As in [19], the reference performance for each application can be specified as a percentage performance degradation with respect to the case when the application is executed independently on the multicore. Note that, this specification can be part of service-level agreement (SLA) between the system administrator and OS. We experimented with 10% to 40% degradations in our simulations for different applications. We also demonstrate the working of METE with a high-level metric (transactions per second) later in the paper. (ii) Model Parameters: The model parameters are determined initially by using regression analysis (for each application separately); they are updated during execution to track the variations in the execution phases of the co-runner applications. (iii) Enforcement Interval: This is one of the important parameters in control design that affect the transient response and overall stability of the system. In our simulations, its default value is selected to be large enough to capture the effects of phase changes. Although increasing the duration of the intervals can reduce the performance overheads incurred by METE, it may also result in system instability and slow reaction to the environmental variations. In any case, the results presented below include all performance overheads incurred by METE.

5.2 Model Validation

As discussed earlier, the model that we employ (as given in Equation (2)) captures the influence of the variations in control parameters (ui(k)) on the measured IPC values (IPCi(k + 1)). We determine the values of the model parameters using the least square algorithm for each application and these values (A and B) are shown in Table 5. To evaluate the accuracy of the obtained model, we compare the IPC values predicted by the model (IPCi(k + 1)) and the actual (measured) IPCs (IPCi(k + 1)), as the resource allocation varies. Figure 8 plots the predicted and actual IPC values for different applications as the layout of the available resource partitioning changes. It can be observed from these plots that the employed model tracks the actual system output with small errors.

The coefficient of determination (R2) and the Mean Absolute Percentage Error (MAPE) are two widely-used metrics to assess the accuracy of a model. MAPE indicates the average error of the model, and an R2 value which is close to 1 indicates high accuracy. These metrics are calculated as follows:

\[
R^2 = 1 - \frac{\sum (\hat{IPC}_i(k) - IPC_i(k))^2}{\sum (IPC_i(k) - IPC_i(avg))^2},
\]

\[
MAPE = \frac{1}{K} \sum_{k=1}^{K} \left| \frac{\hat{IPC}_i(k) - IPC_i(k)}{IPC_i(k)} \right|.
\]

Table 5 lists the obtained values of the above metrics and also the values of the model parameters for the SPECCOMP applications. We can see from the values presented in Table 5 that our employed ARMA model is a good approximation of applications’ behaviors.

5.3 Dynamics of METE

In this set of experimental results, we first show how different static partitioning schemes fail to satisfy the QoS targets of different applications when they execute on a multicore together. We then
Table 5: Model parameters and assessment.

<table>
<thead>
<tr>
<th>Applications</th>
<th>A</th>
<th>B</th>
<th>MAPE</th>
<th>$R^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>applu</td>
<td>1.02</td>
<td>0.41, 0.015, 0.077</td>
<td>0.08</td>
<td>0.91</td>
</tr>
<tr>
<td>apsi</td>
<td>1.02</td>
<td>0.28, 0.013, 0.067</td>
<td>0.09</td>
<td>0.89</td>
</tr>
<tr>
<td>art</td>
<td>0.98</td>
<td>0.33, 0.011, 0.083</td>
<td>0.06</td>
<td>0.96</td>
</tr>
<tr>
<td>gafort</td>
<td>1.01</td>
<td>0.32, 0.018, 0.094</td>
<td>0.11</td>
<td>0.86</td>
</tr>
<tr>
<td>galgel</td>
<td>1.06</td>
<td>0.34, 0.019, 0.074</td>
<td>0.01</td>
<td>0.93</td>
</tr>
<tr>
<td>mgrid</td>
<td>1.02</td>
<td>0.29, 0.014, 0.072</td>
<td>0.05</td>
<td>0.98</td>
</tr>
<tr>
<td>swim</td>
<td>1.01</td>
<td>0.33, 0.017, 0.083</td>
<td>0.06</td>
<td>0.96</td>
</tr>
<tr>
<td>wupwise</td>
<td>1.04</td>
<td>0.32, 0.016, 0.073</td>
<td>0.12</td>
<td>0.91</td>
</tr>
</tbody>
</table>

demonstrate that the same targets (QoS values) can be achieved by employing METE, which is able to dynamically track the specified performance targets (desired IPC targets) by partitioning the available resources (cores, L2 cache and off-chip bandwidth) dynamically during the course of execution. In addition, the performance of the running applications can be further enhanced beyond the specified targets by allocating the excess resources, as will be shown later in Section 5.6. We also study a case in which the QoS targets cannot be satisfied by partitioning the available resources due to the lack of resources.

Figures 9(a)-(c) plot the IPCs achieved by the applications in the mixes given in Table 4, when the multicore resources are statically partitioned between the co-runner applications in each mix (for each mix, the two bars denote the two applications, given in Table 4). In

Figure 9(a), each application receives an equal share of each type of resource, whereas in Figures 9(b) and (c), 75% of each type of resource is allocated to one of the applications in each mix; the remaining 25% is given to the other application. The IPC targets set for the applications in these mixes are shown as (horizontal) solid lines in Figure 9. Our main observation from Figures 9(a), (b) and (c) is that static resource partitioning schemes fail to satisfy some of the specified QoS targets. In comparison, METE takes the specified IPC targets as the reference inputs and decides how the resources we target need to be partitioned in each time interval. Figure 9(d) plots the achieved IPC values when METE is employed. In obtaining these results with METE, when not all the resources are used, we did not distribute them (the use of excess resources will be later discussed). As one can observe from these results, the target QoS values are satisfied successfully in this case for all workloads tested.

Figure 10 illustrates the dynamics of how METE tracks the IPC targets (shown in Figure 9) of different applications during execution. This tracking is achieved by varying the amount of resources allocated to each application dynamically. We observe from Figure 10 that the average maximum overshoot and settling time of different applications are 0.12 and 5 time epochs, respectively. Note
that the maximum overshoot and settling time are two important metrics for the evaluation of feedback control based systems. Based on these results, we can conclude that METE can track specified QoS targets reasonably well. It is important to mention that all types of resources allocated to the applications are dynamically modulated to achieve the results shown in Figure 10. The dynamic variations of these resource allocations are shown in Figures 11, 12 and 13 for caches, bandwidth and cores, respectively. Further, METE is also able to compensate for the changes in the applications’ behaviors over time. For instance, in Figure 10(f) the measured IPC of mgrid unexpectedly increases at time 11 due to the variation in the application’s behavior (i.e., a phase change). METE compensates for this variation and achieves the target QoS by changing the allocations of the available resources. Note that, in these tracking experiments, the goal is to meet minimum resource requirement of each running application to achieve the specified targets. Consequently, there exist excess resources that have not been allocated, as can be seen in Figures 11, 12, and 13.

As mentioned earlier, typically, the behaviors of the applications are not the same in different epochs and, as a result, we can observe that the controllers try to compensate for this by varying the resource allocations. Note also that the number of allocated cores has a significant impact on the measured IPC. Therefore, as can be observed from Figure 13, partitioning of the cores does not vary much once the IPC values get close enough to the specified targets.

5.4 QoS Sensitivity

The results presented so far were for specific QoS values highlighted in Figure 9. It is also important to study how METE behaves under different QoS values. Figure 14 plots the achieved IPC values by employing METE when different performance targets are specified for applu in mix1. In this experiment, we fixed the target IPC value of the other application (apsi) at 0.5. Our main observation is that, METE is successful in satisfying the QoS targets, even when the specified value is high. Only when the QoS target (IPC) is 1.8, METE fails to satisfy.

5.5 Using High-Level Performance Targets

The settling time is the time in which the system output get close enough to the targets (for instance, 5% of their values) So far we have assumed that the performance target accommodated by METE are in terms of IPCs. However, depending on the application domain, higher level performance metrics (like the number of transactions per second) may be specified to METE in a flexible manner. The only change necessary to incorporate any high-level metric is the inclusion of a transducer component that transforms an observable low-level metric like IPC to high-level metric like the number of transactions per second. Note that, such a conversion is necessary because the high-level metrics cannot be measured directly from hardware. However, it can be periodically input to the controllers by the OS or the application. The transducer itself can be designed to model the relationship between the high-level metric and the observable low-level metric. To demonstrate this, we use SPECJBB and use transactions per second as our high level QoS metric. First, in Figure 15(a), we show how our model captures the relationship between the specified high-level QoS and the IPC. We run this benchmark with applu on our multicore with the baseline configuration (given in Table 3). Figure 15(b) shows how METE can track the high-level performance target. Our observation from this plot is that METE can successfully track a high-level metric such as transactions per second. Further, Figure 15(c) shows that performance goals of both of the co-runner applications are satisfied in this case.

5.6 Evaluation of the Resource Broker

If the cumulative resource demands of applications are greater than what the available system resources can provide (i.e., Expression (1) is not satisfied), the resource broker is triggered and it arbitrates between the resource demands from the over-demanding or greedy applications. As a case study, we evaluated the efficacy of the resource broker when QoS target set by the application mixes is 1.8 IPC which is impossible to satisfy with the available resources. Figure 16 shows the result of this study. We evaluated two policies for the resource broker: (1) Policy 1 penalizes each application equally i.e. if the core demands from each application is 4 but there are only 6 cores available, then this policy penalizes each application by a core (2) Policy 2 (our default policy) penalizes each application in proportion to their demands, i.e., it penalizes each application in proportion to its demand. We observe that, using Pol-
In both cases we find that the individual application performance improves but using our default policy, we observe that the FS metric is 4.8% better when compared to the scheme of redistributing the resources that oblivious of application behavior.

5.7 Sensitivity Study

Our goal in this section is to study sensitivity of our scheme to the values of some of the important parameters. The duration of the sampling periods (time intervals) is one of the important design parameters of our control scheme. To study the sensitivity of METE to this parameter, we performed experiments with the time intervals of 5-million and 20-million cycles. Figure 19 shows how well the measured IPC values for different applications track the specified IPC targets (as in Figure 10) when the time intervals are 5-million and 20-million cycles long. The results show that METE is not very sensitive to the execution intervals.

Smaller execution intervals can result in faster responses to the dynamic variations in application behaviors, and therefore, the magnitude of fluctuations may get reduced. In comparison, reducing the duration of time intervals, will increase the control scheme computation and enforcement overheads. Further, the intervals may not be long enough to see the impact of varying resource allocations in the next epoch that leads to inaccurate control decisions.

We next investigate the behavior of our scheme when the work-
load size and core count are modified. Figure 18(a) plots both the IPC targets and the achieved IPC values when the workloads consist of 4 applications running together on 8 cores. We also tested METE on a multicore with 16 cores. As shown in Figure 18(b), the IPC targets are not achieved on the 16-core multicore when no partitioning scheme is used. By employing METE, these targets are satisfied through integrated partitioning of the available resources. As can be observed from these results, the specified targets are still achieved when we increase the number of running applications and cores and METE is scalable for more applications.

6. RELATED WORK

There has been extensive research on management and partitioning of the cache and off-chip memory bandwidth in multicores with the goal of improving the performance of hosted applications [5, 8, 9, 10, 11, 13, 14, 15]. Researchers have also explored various strategies to provide QoS in multicores [31, 32, 33, 34]. However, most prior studies have focused on the management of a single resource and have not considered multi-resource partitioning in multicores.

A multiple resource partitioning scheme called Symbiotic Resource Partitioning has been proposed in [18]. In this scheme, each of the shared cache space and off-chip memory bandwidth is partitioned dynamically based on the feedback from the partitioning of the other resource. The proposed scheme improves global performance metrics and does not consider individual QoS requirements that each application may have. Bitirgan et al. [17] have proposed a framework to manage multiple shared resources on a multicore dynamically to achieve higher level performance objectives by using an artificial neural network based global resource manager that searches the design space of resource allocations by repeatedly querying the performance models followed by selecting the best candidate. Searching the allocation space is, in general, expensive and requires an efficient search mechanism.

Additionally, researchers have applied formal control theory in various domains of computer systems [36, 27] such as software services and performance [37, 38] and power management [20, 39]. A formal control theory approach is proposed in [40] to allocate shared virtualized resources to host application in order to meet QoS requirements. In most of the prior schemes, the primary goal is to maximize the overall performance of a hosted workload. However, there may be applications that receive excess resources, whereas others might lack resources to achieve the specified targets. In contrast to this approach by prior works, in our work, the primary objective is to achieve the performance target of each individual application. Additionally, to our knowledge, ours is the first work that attempts to provide end-to-end QoS in multicore machines.

7. CONCLUDING REMARKS

In this paper, we propose a control theory centric scheme, called METE, to partition multiple shared resources in a multicore machine among concurrently-executing applications at runtime. In the current implementation of METE, we consider three types of shared resources: processing cores, shared cache space, and off-chip memory bandwidth. Assuming that each running application has a performance target to be satisfied, our main goal is to provide applications with sufficient resources to achieve the specified targets. For this, we propose a global resource broker for system wide resource management and a SIMO controller with an ARMA model for capturing the per-application demands. Our experimental results with various application workloads indicate that METE is able to partition the multi-level shared resources among co-running applications.
in most cases, such that the specified QoS targets such as IPC and throughput are satisfied. In summary, our results make a strong case for using a feedback control based resource management scheme like METE for multicore architectures.

8. REFERENCES