

SHARP Control: Controlled Shared Cache Management in Chip Multiprocessors*

Shekhar Srikantaiah, Mahmut Kandemir
Department of CSE
The Pennsylvania State University
University Park, PA 16802, USA
{srikanta, kandemir}@cse.psu.edu

Qian Wang
Department of MNE
The Pennsylvania State University
University Park, PA 16802, USA
quw6@psu.edu

ABSTRACT

Shared resources in a chip multiprocessors (CMPs) pose unique challenges to the seamless adoption of CMPs in virtualization environments and high performance computing systems. While sharing resources like on-chip last level cache is generally beneficial due to increased resource utilization, lack of control over management of these resources can lead to loss of determinism, faded performance isolation, and an overall lack of the notion of Quality of Service (QoS) provided to individual applications. This has direct ramifications on adhering to service level agreements in environments involving consolidation of multiple heterogeneous workloads. Although providing QoS in presence of shared resources has been addressed in the literature, it has been commonly observed that reservation of resources for QoS leads to under-utilization of resources.

This paper proposes the use of formal control theory for dynamically partitioning the shared last level cache in CMPs by optimizing the last level cache space utilization among multiple concurrently executing applications with well defined service level objectives. The advantage of using formal feedback control lies in the theoretical guarantee we can provide about maximizing the utilization of the cache space in a fair manner. Using feedback control, we demonstrate that our fair speedup improvement scheme regulates cache allocation to applications dynamically such that we achieve a high fair speedup (global performance fairness metric). We also propose an adaptive, feedback control based cache partitioning scheme that achieves service differentiation among various applications with minimal impact on the fair speedup. Extensive simulations using a full system simulator with accurate timing models and a set of diverse multiprogrammed workloads show that our fair speedup im-

*This research is supported in part by NSF grants CNS #0720645, CCF #0811687, CCF #0702519, CNS #0202007 and CNS #0509251, a grant from Microsoft Corporation and support from the Gigascale Systems Research Focus Center, one of the five research centers funded under SRC's Focus Center Research Program.

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MICRO'09, December 12–16, 2009, New York, NY, USA.
Copyright 2009 ACM 978-1-60558-798-1/09/12 ...\$10.00.

provement scheme achieves 21.9% improvement on the fair speedup metric across various benchmarks and our service differentiation scheme achieves well regulated service differentiation.

Categories and Subject Descriptors

B.3.2 [Memory Structures]: Design Styles – Cache memory; C.4 [Performance of Systems]: Design Studies

General Terms

Management, Design, Performance, Experimentation

1. INTRODUCTION

Chip multiprocessors (CMPs) are being accepted as the microprocessor architecture of choice and have received strong impetus from almost all leading chip manufactures. Shared resources in a CMP environment pose unique challenges to the seamless adoption of CMPs in both virtualization based environments and high performance computing systems. While sharing resources (like on-chip last level cache) is generally beneficial due to increased resource utilization, lack of control over management of these resources can lead to loss of determinism, faded performance isolation, and an overall lack of the notion of QoS provided to individual applications [16]. This has direct ramifications on adhering to service level agreements in environments involving consolidation of multiple heterogeneous workloads. Partitioning shared resources in CMPs among multiple concurrently executing applications in a controlled manner is arguably an effective solution to this problem in an era where virtualization, service oriented computing and server consolidation are gaining significance.

One of the most important shared resources that directly influences the performance of applications in CMPs is the shared last level cache [5, 14, 15, 9, 7, 17]. Shared last level cache management has been reasonably well studied in recent times [29, 32, 6, 28, 20, 18]. However, many prior schemes [15, 9, 17, 10] that provide QoS for CMPs have identified that the objective of providing QoS by means of static/dynamic resource reservations often conflicts with the goal of achieving good cache space utilization, which has been the primary reason for adapting a shared cache configuration in the first place. By relying on heuristics for management of shared cache space, it is not possible to provide guarantees about the optimization objectives, such as maximizing the cache space utilization. It is also not possible to reason about the “settling time” or the time

it takes to achieve this objective in a stable state since a heuristic may never converge towards an optimization objective. Consequently, the results with the use of such heuristics may be sub-optimal or, under special circumstances (depending on application and platform characteristics like working-set size, resource pool size etc.), poorer than the base case of equal share or no partitioning. To alleviate the above mentioned problems, we propose to adopt *feedback control theory* for shared last level cache management in CMPs and demonstrate this using two shared cache partitioning schemes. Control theory [8, 12] is a powerful tool that can provide us such guarantees and precise reasoning about system parameters like cache space utilization. The use of control theory also has certain secondary advantages such as resilience to minor inaccuracies in the modeling of the system, quick adaptive response to changes in dynamic behavior of the workloads by rejecting disturbances due to modulations in resource landscape, and ease of specifying high level objectives.

Apart from achieving the best utilization possible, an important global performance objective for such a shared resource partitioning scheme could be to achieve a high “fair speedup”. The Fair Speedup metric (FS) of the workload using a partitioning scheme is defined as the harmonic mean of per application speedup using the scheme with respect to the baseline equal resource share case (equal number of ways to all applications; similar to those used in [6] and [28]). Fair speedup achieved by a scheme can be expressed as $FS(\text{scheme}) = N / \sum_{i=1}^N \frac{IPC_{app_i}(\text{base})}{IPC_{app_i}(\text{scheme})}$, where N is the number of applications in the workload, i.e., the set of applications that execute together. FS is an indicator of the overall improvement in instructions per cycle (IPC) gained across the applications. It is also a metric of fairness. Another important consideration in partitioning shared resources is providing *service differentiation* to each of the consolidated applications. Not all applications consolidated on the same platform may be of equal importance. The difference in priority could be based purely on the service level agreement or could be based on the relative throughput requirements of each of the consolidated applications. We realize these two objectives (maximizing cache utilization and improving global performance metrics) in a framework that provides best effort QoS using the SHARed Resource Partitioning (SHARP) control architecture. We believe that the SHARP control architecture proposed in this paper is the first step towards precise control of resource partitioning for CMPs. Our proposed control system consists of three major parts: (i) Per-application controllers that determine the cache space allocation required to meet the specified performance targets by the applications; (ii) A *Pre-Actuation Negotiator* (PAN) in order to gracefully handle scenarios where the total demanded cache capacity by the application controllers is greater than the available resources according to a flexible policy; and (iii) A SHARP controller that increases the reference performance targets to the application controllers in a fair manner in order to cater to the scenarios of under-utilization of cache capacity. We also implement two policies to illustrate the importance of PAN and the SHARP controller in our architecture. Specifically, we make the following contributions:

- We develop an empirical model to quantify, compare and predict the performance achievable by an application,

given a certain number of ways of the cache, using a *way partitioning* scheme for shared caches. We further validate this model using a diverse range of applications.

- We propose a novel controller for the per-application controller called *Reinforced Oscillation Resistant* (ROR) controller for each application that helps to alleviate the problem of oscillations in resource allocation as compared to a design using the state-of-the-art Proportional-Integral-Derivative (PID) controllers. The ROR controller is a customized controller that has two fundamental properties of a model predictive controller [4], namely, an internal dynamic model of the process and a history of past control decisions.

- We demonstrate two policies that can be implemented in the SHARP control architecture in the PAN and the SHARP controller: (i) A *fair speedup improvement* scheme that regulates cache allocation to applications dynamically such that we achieve an improved fair speedup (IPC). We argue that metrics like IPC and cache miss rate, which were previously considered “inconvertible” to available resource capacity, may be considered for target specification in a QoS system with the use of simple models by adopting control based techniques to provide such guarantees. It is also important to note that we are proposing a best-effort QoS assurance to every application that is part of the workload and not an admission control policy. However, predictions based on our performance model may also be used to develop a powerful admission control policy. (ii) A policy that achieves *service differentiation* among various applications with minimal impact on the fair speedup achieved. We also provide a sketch of the proof for the guarantee we provide on maximizing the cache space utilization.

Extensive simulations using a full system simulator with accurate timing models using 2-core and 8-core CMP configurations and a set of diverse multiprogrammed workloads show that our fair speedup improvement policy, with our ROR controller, improves the Fair Speedup metric by 21.9% on an average. We also compare our scheme to two previously proposed heuristic schemes for shared cache partitioning. The significant improvements in fair speedup metric achieved highlights the importance of formal techniques and accurate performance modeling in achieving improved performance results apart from the theoretical guarantees (on resource utilization) provided by such formal schemes in shared resource partitioning.

2. BACKGROUND

Dynamic adaptation to changing workload and environment conditions has several analogies to many physical world problems, where feedback control theory has been successfully applied. Consider, for example, water temperature control for a shower. People measure the water temperature using their hands and then tune the hot/cold knob in terms of the feedback error between the measured temperature and a desired reference temperature, i.e., to turn towards hot when the temperature is too low and to turn towards cold when the temperature is too high. A control-theoretic approach to system management requires: (i) Identification of the criteria for optimization (e.g., minimizing the tracking error between actual temperature and the reference temperature, and the cost for water being heated), which is referred to as “cost function”; (ii) Determination of the parameters to control (e.g., the angular displacement of the hot/cold knob for a shower), called “control inputs”; (iii) Monitoring of certain

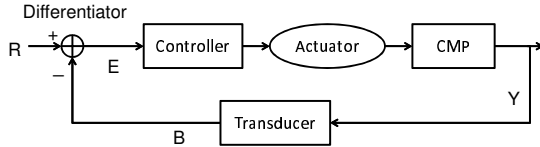


Figure 1: Illustration of a simple feedback control loop for CMP management.

metrics to observe the effect of the control (e.g., the current water temperature), called the “output variables”; and (iv) A model to understand how much effect would a change in the control parameters have on the output variables (e.g., a differential (or difference) equation that characterizes the dynamic relation between the displacement of the hot/code knob and the water temperature), which is called the “dynamic system model”.

The principles and design methodologies from control theory can be used to design systems that maintain desirable performance and provable stability by automatically adapting to changes in the environment. By measuring the operation of a system, comparing it to a reference or target value, and adjusting available control variables, a feedback control system can respond properly even if its dynamic behavior is not exactly known.

Figure 1 illustrates a feedback control loop that can be used for the CMP resource management problem. To associate this generic feedback loop with our CMP resource management problem, let us consider a particular scenario, which is not the specific problem addressed in this paper but given here for purpose of illustration. In this scenario, our reference input, R , is the maximum allowable degradation of an application with respect to the unlimited resource case (e.g., we want the maximum degradation due to limited resources to be bounded by $x\%$ and would like to determine the amount of resources to allocate for achieving this). The output signal, Y , can be CPI (cycles per instruction), which can be obtained using hardware performance counters. Noting that R and Y are of different types of metrics, the job of the transducer is to convert Y to a metric (B) which can be directly compared to R . The difference between R and B , denoted using E , is fed to the controller which decides the new resource management (in this case, allocation) policy. This policy is implemented by the actuator by exercising the necessary hardware knobs and/or invoking certain OS services. Depending on the magnitude of E , the controller may instruct the actuator component to take the appropriate course of action, e.g., allocate more resources to satisfy the goal. Note that designing these components may not be very easy. For example, the transducer in this scenario also involves modeling the performance under the unlimited resource case. Note also that in this approach the controller makes the “policy” and actuator enforces that policy using a “mechanism” such as resource allocator. In general, depending on the specified goal, these three components (controller, transducer, and actuator) and the feedback loop that connects them can be very complex, and in fact, we can have multiple feedback loops controlling one another.

3. CONTROL ARCHITECTURE

A high-level view of our proposed control architecture (the SHARP control architecture) is shown in Figure 2. There are three major components of this architecture, as

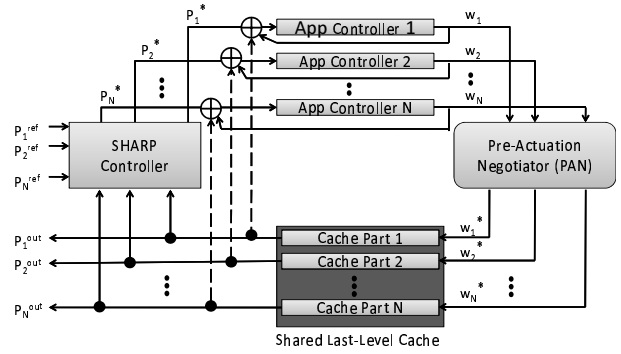


Figure 2: SHARP Control architecture.

seen from this figure: (i) SHARP Controller; (ii) Application Controllers (AppC); and (iii) Pre-Actuation Negotiator (PAN). The two types of controllers (SHARP controller and Application controllers) are organized in a hierarchical fashion. The SHARP controller along with the PAN is responsible for tracking the system objectives (like maximizing the cache space utilization and improving the fair speedup metric or a specific service differentiation objective). In order to achieve these objectives, the SHARP controller tunes the reference inputs (constant IPC targets provided to SHARP controller, $\{P_1^{ref}, P_2^{ref}, \dots, P_N^{ref}\}$) and determines new targets ($\{P_1^*, P_2^*, \dots, P_N^*\}$) for each of the AppC controllers to maximize the utilization of the cache space whenever it is under-utilized. Note that, $P_i^* \geq P_i^{ref}$ for all i . Each application of a workload has its own AppC controller that determines the ideal resource share (as number of ways, w_i in the way partitioned shared last level cache) that it needs to track the desired input, P_i^* , determined by the SHARP controller. The AppC controllers try to track the revised targets P_i^* and not P_i^{ref} . The process of activating the partition $\{w_1, w_2, \dots, w_N\}$ determined by the AppC controllers on the shared last level cache in control terminology is referred to as *actuation*. But, it is important to observe that each of the AppC controllers *independently* determines the cache share (the number of cache ways) that it needs and the sum of all these shares ($\sum_{i=1}^N w_i$) may be more than the total available ways, W in the cache. In order to handle such a situation, we have introduced the Pre-Actuation Negotiator (PAN) to negotiate among each of the demanded shares in a “fair” manner and determine a partition $\{w_1^*, w_2^*, \dots, w_N^*\}$ that is in turn activated on the resource to produce the response output performance metrics ($\{P_1^{out}, P_2^{out}, \dots, P_N^{out}\}$). The output performance metrics are again used by the SHARP controller and the AppC controllers as input for the next interval of time.

The shared last level cache is thus controlled by a hierarchical organization of controllers. The knob for tuning the performance of each application is the number of cache ways allocated to it in the way-partitioned shared last-level cache. This architecture supports multiple goals like fair speedup improvement or service differentiation. Note that, the goals are automatically transitioned into *best-effort solutions* in case the reference input is over-specified. The delegation of the responsibility of regulating the AppC control inputs to the SHARP controller and the responsibility of revising the AppC output to the PAN is a key feature of the SHARP control design that helped to substantially reduce the complexity of the architecture from a functionally equivalent

monolithic architecture. Apart from reducing the complexity of each of the components, the presence of the SHARP controller and the PAN in SHARP control architecture also provides the AppC controllers the liberty of requesting any degree of resource share independent of the other controllers. The impact of other AppC controller decisions is further considered by PAN that negotiates a feasible partition in a “fair” manner. By suitably designing the AppC controllers in this “liberal domain”, the SHARP control architecture also helps to mitigate the possibility of frequent oscillations of high magnitude in the allocations, a common problem in many feedback control systems [12].

4. APPC CONTROLLER

The design of AppC controllers in the SHARP control architecture is a significant part of the design, as the AppC controllers influence the resource (shared last level cache) partitioning to the greatest degree. The SHARP controller and the PAN are only reconciliatory in nature as they influence the partitioning decision only when the partition determined by AppC controllers is infeasible. Each AppC controller $AppC_i$ has to determine at every time interval t , the number of ways $w_i(t)$ necessary to achieve a target performance (like IPC) of $P_i^*(t)$. The other inputs that are available for the controller to make this decision are the performance $P_i^{out}(t-1)$ achieved during time interval $t-1$ with a partition of size $w_i(t-1)$.

4.1 PID Controller

We designed the AppC controller using a Proportional-Integral-Derivative controller (PID controller). PID controller is the most dominant form of classical control methodology in use today [3]. The advantages of the PID control are that the design is very intuitive and often directly related to practical design specifications. A proportional-integral-derivative (PID) feedback control law for the AppC controller takes the following form:

$$w_i(t) = w_i(t-1) + K_P \cdot e_i(t) + K_I \cdot \int_0^t e_i(\tau) d\tau + K_D \cdot \frac{d(e_i(t))}{dt}, \quad (1)$$

where $w_i(t)$ is the number of allocated cache ways to application i in interval t , the design parameters K_P , K_I , and K_D are obtained using complex formal methodologies (such as Bode plots or through the application of stability criteria, such as Nyquist stability [8]). In general, the classical PID control cannot handle the multiple-input multiple-output (MIMO) systems directly; instead, it has to translate the problem to multiple single-input single-output (SISO) loops as is the case in our AppC controllers.

4.2 Reinforced Oscillation Resistant Controller

Although the PID controller is reasonably accurate in tracking the desired inputs (P_i^*), in doing so, the eventual resource allocations, $\{w_1^*, w_2^*, \dots, w_N^*\}$, may produce significant amount of oscillations. There are several other disadvantages of using PID controllers:

- The process of setting the design parameters, K_P , K_I , and K_D in Eq. (1) is painstaking to say the least. Moreover, these design parameters are also sensitive to the applications constituting the workload.

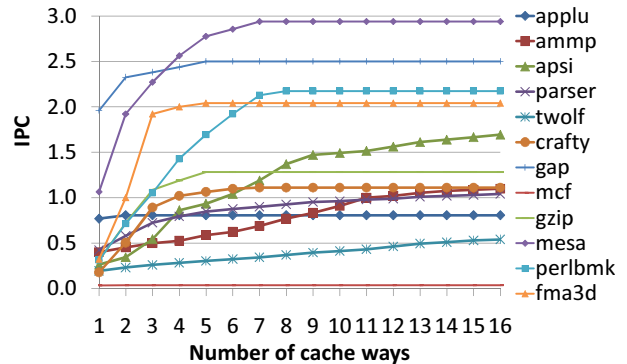


Figure 3: Measured IPC vs. the number of allocated cache ways in a 16-way set associative cache.

- The spikes caused due to oscillations in cache space allocation of one AppC controller can adversely affect the decisions made by the Pre-Actuation Negotiator (PAN), leading to reduced benefits in terms of the overall speedups across the applications of the workload.

- Frequent changes in allocated cache space blur the notions of “controllability” and “observability” as a change in the cache allocation may not manifest as a change in the output (IPC) within the window of observation, before the allocation changes again.

- Frequent changes in the cache allocation (either a positive or negative spike in cache allocation) may also deceive the SHARP controller as it encourages the SHARP controller to redistribute either too less cache space (in case of positive spikes) or too much (in case of negative spikes), effectively reducing cache utilization.

Most of the disadvantages listed above can be traced to be due to two fundamental characteristics of the PID controllers. Firstly, the design parameters, K_P , K_I , and K_D (in Eq. (1)) are constants for the controller and hence do not take into account any dynamic, time varying characteristics of the applications executing on the CMP with the partitioned cache. A more fundamental reason for this is that the PID controller is not affected by its history (previous control decisions) and is only dependent on the current state of the system (the observed IPC as a result of the current cache allocation irrespective of previous allocation). Updating the system identification model by incorporating additional data from time to time would help improve the modeling accuracy and thus the performance of the resulting control. Secondly, the PID controller is a generic controller and does not utilize any domain specific knowledge about the process dynamics. PID controller treats the system as a blackbox, where, it knows only about the change in output for a given change in input. It performs well for linear behaviors of the system. Given the high degree of non-linearity in cache behavior, it is important to have an explicit model exposed to the controller. Simple understanding of the process dynamics (a performance model for the partitioned cache in our case) can take us a long way in improving the performance of the controller.

An important characteristic of control-theoretic feedback loops [8, 12] is that they can deal with poor knowledge of the system, changes in the system or the workloads, and other disturbances. This resilience to modeling/performance estimation of systems is a crucial factor that has led to the suc-

cess of control systems in managing complex and dynamic runtime systems. We exploit this flexibility to develop a simple, yet powerful empirical cache performance model applicable to a way-partitioning scheme for shared caches [32]. Our model estimates the performance of an application in terms of instructions per cycle (IPC) as a function of the number of cache-ways allocated to it in a way-partitioned shared cache. Based on this model, we propose a customized AppC controller (similar in design principles to a model predictive controller [4]) that exploits the insights derived from the performance model to alleviate the problem of variations in cache space allocation by tracking the history of control decisions made. We call this history and model reinforced controller as Reinforced Oscillation Resistant (ROR) controller.

4.3 Cache Performance Modeling

Partitioned shared cache lends itself to accurate and better modeling as cache behavior of each application can be modeled independently in the absence of inter-thread interferences. Figure 3 plots the variations in the measured instructions-per-cycle (IPC) of various applications on our baseline processor configuration (details in Section 8) by varying the number of cache ways allocated to the application in a 16-way associative shared cache. The generic shape of the curves clearly shows the non-linear dependence of the performance of applications measured in IPC on the number of cache ways allocated to it in the way-partitioned shared cache. The saturating non-linear behavior of the IPC of each benchmark (φ_i) observed in the plot can be modeled as a function of the number of cache ways allocated to it (ω_i) in the following general form:

$$\varphi_i = \varphi_i^\infty \cdot (1 - e^{-\alpha_i \cdot \omega_i}), \quad (2)$$

where φ_i^∞ is the IPC of the application i when it is allocated maximum number of cache ways (theoretically, $\omega_i = \infty$) and α_i is a parameter that roughly determines the utility of each additional cache way that may be allocated to application i . Note that, the model given in Eq. (2) very well captures the saturating behavior of the IPC increase with the increase in the number of cache ways and saturates at a value of φ_i^∞ . It is also important to emphasize that SHARP control architecture can be extended to handle non-convexity in application’s response to increasing cache by increasing the number of parameters in our model (by having a vector of parameters for each convex curve). More complex models of cache performance have been studied [31], but the complexity of such models is prohibitive considering that control theory already provides resilience to minor inaccuracies in the models.

In order to validate this model, we plot the measured instructions per cycle (IPC) and the predicted IPC using our simple cache performance model given by Eq. (2) by varying the number of allocated cache ways in a 16-way set associative way cache. The plots in Figure 4 show the comparison of the measured values against the predicted values along with the model parameters when benchmarks are executed with training inputs¹. We can see from the plots that this model in fact tracks the measured values with acceptable

¹We use training inputs for validating the model, but the parameters of the model are obtained dynamically and benchmarks are run with reference inputs when used in the controller.

precision. The simple, yet accurate model has another important characteristic that makes it a good fit for a history based controller; the model can be updated frequently as and when more samples of φ_i and w_i are known in a very simple manner using least mean square methods. This helps in accounting accurately for the phase behavior typically observed in applications. Note that the cache model parameters (both φ_i^∞ and α_i) are learned online and updated with continuing-allocations to adjust to phase-behavior of applications. Frequently updating the model also helps us to account for other cache performance behaviors like elimination of capacity misses (when the working set fits in the allocated ways) apart from inherently capturing the decrease in conflict-misses as the number of allocated cache ways increases. We rely on this history-reinforced model to design the ROR controller. Given that the parameter being controlled in the ROR controller at time t is $w_i(t)$ (the number of cache ways requested by the AppC controller), we can write the general form of the control law for the ROR controller as:

$$\forall i, w_i(t) = w_i(t-1) + \Delta w_i(t), \quad (3)$$

where $\Delta w_i(t)$ is the correction computed for time t , given the history of control moves made by the controller and the current state of the model (current values of φ_i^∞ and α_i) for application i . The value of $\Delta w_i(t)$ can be determined using the history of control moves as the difference between $\psi_i(t)$, the predicted value of $w_i(t)$ by the cache performance model, and the moving average $m_i(t-1)$, of the previous values of w_i determined using the same control law. Note that, the moving average is defined recursively over the values of w_i computed by the ROR controller over the previous intervals of time as:

$$m_i(t-1) = \beta m_i(t-2) + (1-\beta)w_i(t-1). \quad (4)$$

The older values of average ways (m_i) are exponentially attenuated with a factor β , where $0 < \beta < 1$. A higher value of β will increase the window size over which w_i is averaged.

From Eq. (2), we can determine the value of $\psi_i(t)$ as predicted by the cache performance model by using simple algebraic manipulations as:

$$\psi_i(t) = ceil \left(-\frac{\log \left(1 - \left(\frac{P_i^*(t)}{\varphi_i^\infty} \right) \right)}{\alpha_i} \right), \quad (5)$$

where $P_i^*(t)$ is the target IPC given as input to the ROR controller. Having determined all the individual components of the control law for the ROR controller, we can now describe the control law (by consolidating Eqs. (3), (4), and (5)) as:

$$\forall i, w_i(t) = w_i(t-1) + \psi_i(t) - m_i(t-1). \quad (6)$$

It is important to observe that φ_i^∞ should always be greater than $P_i^*(t)$. If this is not the case, the reference IPC of $P_i^*(t)$ can not be achieved and the system is said to lose its property of being controllable. In the ROR controller, it is easy to detect such a situation and also correct if necessary by using the minimum of the two values. It is clear that, we could also have a flexible admission control policy that accepts all workloads but either rejects or degrades those which over-specify target performance. Note that, ROR controller is one particular design for the general AppC controllers. We are conservatively approximating the number of cache ways required to be the ceiling of the predicted number of cache

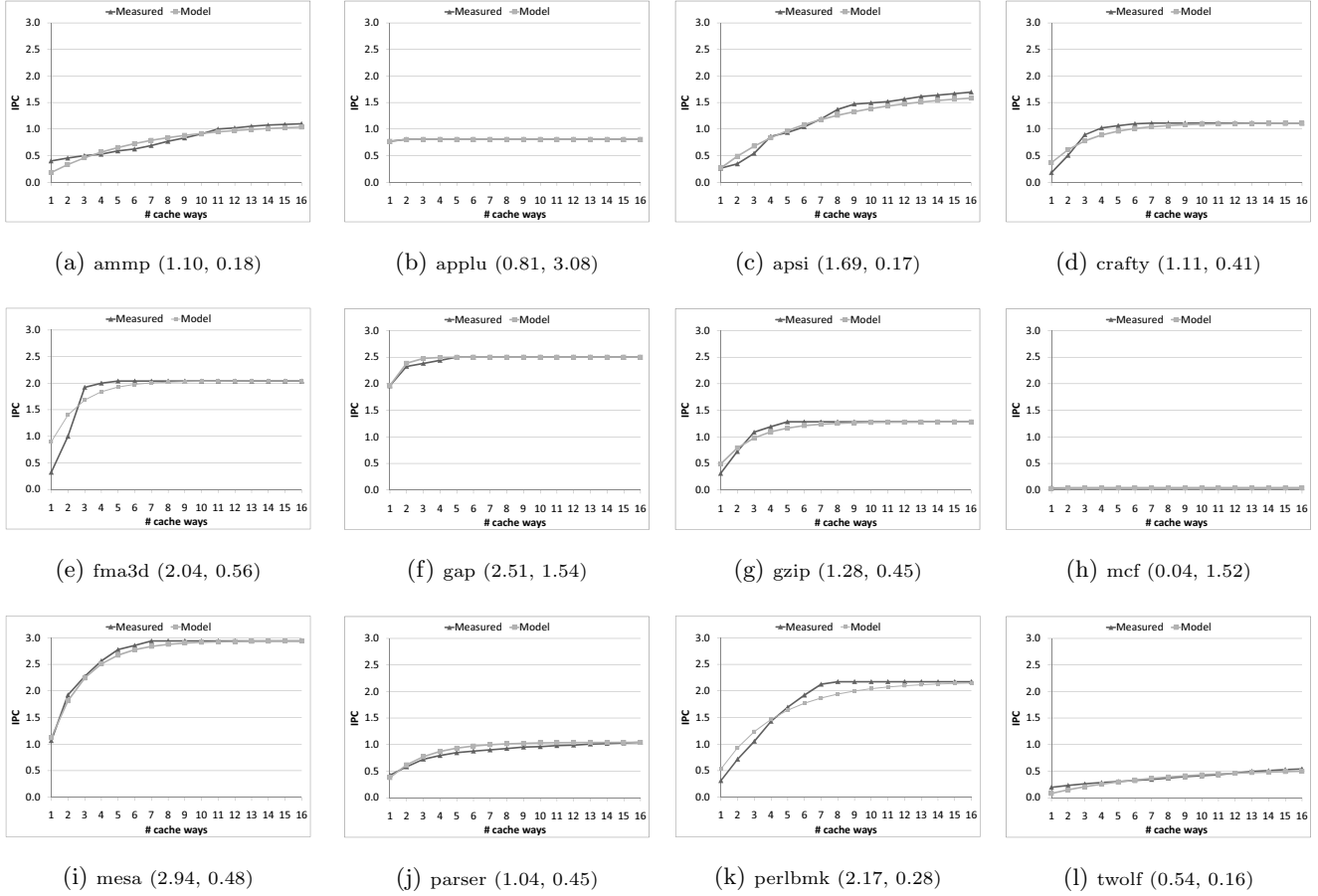


Figure 4: Measured instructions per cycle (IPC) vs. predicted IPC using our simple cache performance model for various benchmarks with varying number of allocated cache ways in a 16 way set associative cache. Only one line is visible in (b) and (h) due to a perfect match between the measured values and the values predicted by the model. The numbers indicated with each benchmark represent $(\varphi_i^\infty, \alpha_i)$ pair when benchmarks are run with training inputs.)

ways. We can take this liberty of conservative estimation as there is no interdependence between two different AppC controllers. As explained earlier, in general, $\sum_{i=0}^N w_i(t)$ may be greater than W , the total number of ways available in the shared last level cache. The values of $w_i(t)$ computed by the AppC controllers are then fed to PAN in order to determine a feasible partition.

5. PRE-ACTUATION NEGOTIATOR

The Pre-Actuation Negotiator (PAN) is used to derive a feasible partition, $\{w_1^*, w_2^*, \dots, w_N^*\}$ in a way-partitioned shared cache with W ways, based on the demands $\{w_1, w_2, \dots, w_N\}$ from individual AppC controllers, when $\sum_{i=0}^N w_i > W$. By separating the AppC controllers and the PAN, we have the flexibility to implement various policies in the PAN for deriving a feasible partition. In this work, we implemented two sample policies, as explained below.

5.1 Fair Speedup Improvement (FSI)

A total of $spillw = \sum_{i=0}^N (w_i) - W$ number of ways have to be recovered from the workload in order to determine a feasible partition. In doing so, the idea behind fair speedup improvement (FSI) policy in PAN is to keep the *fair speedup*

metric (explained later) of the workload as high as possible. The treatment given by the FSI policy to each application is proportional to w_i , the number of ways determined to be required by the AppC controller for that application. Therefore, w_i^* is determined as:

$$w_i^* = \text{floor} \left(w_i \left(1 - \frac{spillw}{\sum_{i=0}^N w_i} \right) \right). \quad (7)$$

We determine w_i^* as the floor, as we cannot exceed the total limit of W in order to obtain a feasible partition. If there are any excess ways that remained due to this conservative estimation, they can be redistributed in the PAN for best utilization of the cache or we could wait until the SHARP controller reacts to this condition in the next interval of time.

5.2 Service Differentiation (SD)

In many practical scenarios, it is important to provide service differentiation as opposed to fairness. For this purpose, we also implemented and experimented with a policy called service differentiation (SD) that provides service differentiation in case of the demand for cache space being higher than the available space. A simple, yet effective service differentiation scheme, in recovering $spillw = \sum_{i=0}^N (w_i) - W$

ways, recovers as many ways as possible from the lower priority applications (until they are left with no more than a specified minimum number of ways) before recovering any from the higher priority applications. Note that, the recovery of ways from lower priority applications still happens in a manner proportional to the requested number of ways by the AppC controller of that application as in the FSI policy explained above. An important feature of the SD policy is that it enforces service differentiation only when necessary and does not impact the execution of lower priority applications if it is possible to satisfy both. Also, given reasonable target specifications, the long term targets of low priority applications are also achieved, albeit not guaranteed.

6. SHARP CONTROLLER

While PAN handles the case where the sum of requested ways by the AppC controllers is higher than the available number of ways in the cache, the opposite scenario of the sum of requested number of ways for achieving the target IPCs being less than the total available cache ways in the cache is handled by the SHARP controller in order to increase the utilization of the cache. Note that, power consumption is also an important metric in the management of shared last level caches in CMPs. So, we could consider turning off the excess ways if the cache is appropriately banked for reducing the power consumption, while achieving the targeted IPCs. However, in this work, our focus is to achieve an improved fair speedup (performance) of the workload. In order to achieve this goal, it is desirable to keep the cache fully utilized.

Therefore, whenever the cache remains under-utilized, the reference IPCs for the AppC controllers $\{P_1^*, P_2^*, \dots, P_N^*\}$ are recomputed by the SHARP controller in a fair way, in case the FSI policy is used in the PAN so that the additional $W - \sum_{i=0}^N w_i$ ways are distributed fairly across applications as:

$$P_i^*(t) = P_i^{ref} \frac{W}{\sum_{j=0}^N \left(\frac{w_j^*(t-1)}{P_j^{out}(t-1)} P_j^{ref} \right)}. \quad (8)$$

In case of the SD policy being used in the PAN, the additional cache ways are redistributed based on the weights associated with the applications (weights, $\{\Theta_1, \Theta_2, \dots, \Theta_N\}$ may be computed as inverses of application priorities) as:

$$P_i^*(t) = P_i^{ref} \frac{W \sum_{j=0}^N \Theta_j}{\sum_{j=0}^N \left(\frac{w_j^*(t-1)}{P_j^{out}(t-1)} \cdot P_j^{ref} \cdot \Theta_j \right)}. \quad (9)$$

Note that the initial targets inputs, $\{P_1^{ref}, P_2^{ref}, \dots, P_N^{ref}\}$, to the SHARP controller are constants. We can prove that recomputing IPC reference inputs as in Eq. (8) minimizes $|W - \sum_{i=0}^N w_i|$, if we model the IPC of each application as a time varying function of the number of cache ways allocated to it.

7. GUARANTEEING OPTIMAL CACHE SPACE UTILIZATION

We claimed that the process of recomputing performance targets in the SHARP controller in order to redistribute unutilized cache ways happens in a fair manner such that it minimizes the deviation of actual allocations from the

demands of the AppC controllers. We would like to substantiate the claim here by providing a sketch of the proof. We need to show that the recomputation of target IPCs according to Eq. (8) minimizes $|W - \sum_{i=0}^N w_i|$, where W is the number of available cache ways in the shared last level cache and w_i is the number of ways allocated to application i , so that the cache space utilization is optimal. Consider, without loss of generality, the IPC of application i , $P_i(t)$, being modeled as a time-varying function of the number of cache ways allocated ($w_i(t)$) to it:

$$w_i(t) = \tau_i \times P_i(t), \quad (10)$$

where τ_i is a time varying co-efficient. Note that, this simple time-varying nature of the model captures local behavior of the cache at any instant of time and in general, provides a close piecewise linear estimate of the overall performance model presented in Section 4.3. During the recomputation, we need the newly computed target, $P_i^*(t+1)$ to be proportional to the original reference, P_i^{ref} . Therefore, we have:

$$P_i^*(t+1) = P_i^{ref} \times K_i(t), \quad (11)$$

where K_i is a constant for each application, but may be time varying. Assuming that $w_i^\dagger(t+1)$ is the number of cache ways that would be allocated ideally due to the recomputed value of the target IPC, in order to maximize the utilization and $w_i^*(t)$ is the number of ways actually allocated by the PAN to each application i and the corresponding actual IPC obtained was $P_j^{out}(t)$, we have the following relationships from Eq. (10):

$$w_i^\dagger(t+1) = \tau_i \times P_i^*(t+1), \quad w_i^*(t) = \tau_i \times P_j^{out}(t) \quad (12)$$

Clearly, $|W - \sum_{i=0}^N w_i|$ is minimized after the targets are recomputed when $\sum_{i=0}^N w_i = W$. From Eq. (12), this condition is true when:

$$\sum_{i=0}^N \tau_i \times P_i^*(t+1) = W \quad (13)$$

From Eq. (11) and Eq. (13), we have:

$$\begin{aligned} \sum_{i=0}^N \left(\tau_i \times P_i^{ref} \times K_i(t) \right) &= W \\ \Rightarrow K_i(t) \times \sum_{i=0}^N \left(\tau_i \times P_i^{ref} \right) &= W \\ \Rightarrow K_i(t) &= \frac{W}{\sum_{i=0}^N \left(\tau_i \times P_i^{ref} \right)} \end{aligned} \quad (14)$$

By substituting for the value of $K_i(t)$ from Eq. (14) in Eq. (11), we obtain:

$$P_i^*(t+1) = P_i^{ref} \times \frac{W}{\sum_{i=0}^N \left(\tau_i \times P_i^{ref} \right)}. \quad (15)$$

Since we have $\tau_i = \frac{w_i^*(t)}{P_j^{out}(t)}$ from Eq. (12), by substituting this in Eq. (15), we obtain:

$$P_i^*(t+1) = P_i^{ref} \frac{W}{\sum_{i=0}^N \left(\frac{w_i^*(t)}{P_i^{out}(t)} P_i^{ref} \right)}. \quad (16)$$

Substituting $t - 1$ for t and changing the index of summation from i to j , we obtain:

$$P_i^*(t) = P_i^{ref} \frac{W}{\sum_{j=0}^N \left(\frac{w_j^*(t-1)}{P_j^{out}(t-1)} P_j^{ref} \right)} \quad (17)$$

This is identical to the logic used in the SHARP controller (Eq. (8)). Similarly, by modifying Eq. (12) to account for the allocated cache ways to be biased by weights Θ_i , we can also prove the same result for the logic used in the SHARP controller in case of the service differentiation policy.

8. EXPERIMENTAL SETUP

Base System Configuration. We evaluate our SHARP control architecture on 2-core and 8-core CMPs with 4 way issue superscalar processors. The baseline configuration of our target CMP is as shown in Table 1. We simulate the complete system using Simics full-system simulator [24]. We have augmented Simics with accurate timing models similar to RUBY module of GEMS [25].

Processor Model	4 way issue superscalar
Private L1 I & D-Caches	Direct mapped, 32KB, 64 bytes block size, 3 cycle access latency
Shared L2 Cache	16-way set associative, 4MB, 64 bytes block size, 15 cycle access latency
Memory	4GB, 200 cycle off-chip access latency
Enforcement Interval	10 Million cycles

Table 1: Baseline configuration.

Type	Mix	Applications	Ref-IPC	CCP
Type I	Mix 1	<i>apsi, parser</i>	{1.0, 1.0}	16 ways
	Mix 2	<i>crafty, ammp</i>	{1.0, 1.0}	15 ways
	Mix 3	<i>perlbmk, mesa</i>	{2.0, 2.8}	13 ways
Type II	Mix 4	<i>mcf, gzip</i>	{0.04, 1.2}	5 ways
	Mix 5	<i>gap, applu</i>	{2.5, 0.8}	6 ways
	Mix 6	<i>twolf, fma3d</i>	{0.3, 2.0}	7 ways

Table 2: Mixes of applications considered for our multiprogrammed workload (CCP - Combined Cache Pressure).

Benchmarks. We have used twelve representative programs from the SPEC benchmark suite [13] which includes both integer and floating point benchmarks with diverse computational and memory access characteristics. We have chosen these twelve benchmarks so as to represent a large spectrum of IPCs ranging from a mere 0.04 (mcf) to 2.94 (mesa). We also have selected the reference IPCs to ensure that the cache demand of workloads is spread over a wide spectrum. We constructed six workloads from the selected twelve and divide the workloads into Type-I and Type-II based on the combined cache pressure measured as the sum of number of cache ways they need to achieve the specified target when they are executed independently. Type-I workloads have a high cache pressure (13-16 ways) and Type-II workloads have a low cache pressure (5-7 ways). All the benchmark programs use the reference input set. We warm up caches for approximately 1 billion cycles. The six different mixes of multiprogrammed workloads, the reference IPC targets used with them and the combined cache pressure of the workloads are shown in Table 2. Later in Section 9.5, we use multiprogrammed workloads of eight applications each, for evaluating the our control architecture on 8-core CMPs.

Platform. We implemented the proposed control architecture as a module in Solaris 10. We bind the individual applications to processors so that we can assign partitions to applications. We use the *pset.create()* and *pset.assign()* system calls in Solaris to dynamically create processor sets and assign processors to processor sets (effectively creating processor partitions) and the system call *pset.bind()* to assign processes to these partitions. We have implemented Simics modules to collect processor performance information like IPC values. Our OS interface for enforcement of cache partitioning is similar to that in [29] and a detailed discussion about timing and area overheads of the necessary hardware implementation is given in [29]. Once determined and enforced, a cache partition remains for a duration of enforcement interval (default value of 10 Million cycles).

Parameters. SHARP control architecture requires the selection of some key parameters (of the controller and the controlled system) that is crucial for obtaining desired results. We describe how we envision each of these parameters to be selected below:

- **Reference-IPCs:** We envision a system where references IPCs are set based on negotiation between the administrator and the OS similar to service-level-agreements. As a guideline to specify reference IPCs, we can use reference IPCs of a specified percentage degradation from that derived when the application is executed independently on the infrastructure. We used 10% to 50% degradation to demonstrate the versatility of our approach.

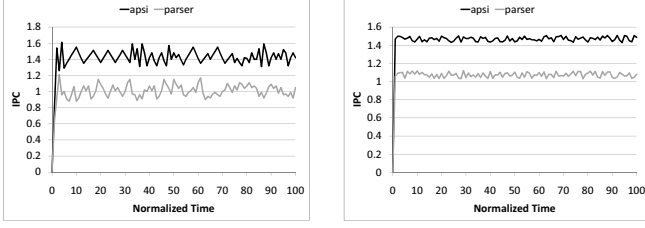
- **Cache model parameters:** We learn the cache model parameters (psi and alpha) by estimating them using statistical regression to find the curve-fitting the model with minimum mean-squared-error. The attenuation factor (beta) was experimentally determined to be best accounting for the attenuation of the history. We could also use a per-application attenuation factor, but this was not found to be making a significant difference.

- **PID Controller design parameters:** PID controller coefficients have been tuned using iterative feedback tuning [22] based on the response that maximizes the average fair speedup across the workloads.

- **Enforcement interval:** Enforcement interval need not vary with applications as long as it is small enough to capture the effects of phase changes.

- **SD-policy weights:** Weights (priorities) can be set based on different classes of service. This is an input parameter and is not determined by the system.

Overheads. The overheads involved in computations in all the controllers put together is insignificant compared to the enforcement interval (roughly about 15000 cycles or 0.15% overhead). Note that we are performing full system simulation and our results discussed in Section 9 includes all these overheads. Apart from the components and parameters discussed above, we also conducted experiments for studying the sensitivity of the controller to the enforcement interval. The default enforcement interval is 10M cycles. While intervals lesser than 10 Million cycles lead to a smoother response, the overheads are higher leading to 1.4% reduction in improvements in the fair speedup metric averaged across various mixes for an interval of 1M cycles. Enforcement interval of 100 Million cycles (default interval is 10M cycles) results in a 4.6% reduction in improvements in the fair speedup metric (using the FSI policy and the ROR controller), when averaged across various mixes due



(a) PID Controller ($K_P = K_I = 0.8, K_D = 0.6$) (b) ROR Controller ($\beta = 0.6$)

Figure 5: Adaptation of instructions per cycle (IPC) with time for applications of Mix-1 with the PID and ROR controllers using the FSI policy.

to increased oscillations in the resultant cache allocations.

9. EXPERIMENTAL EVALUATION

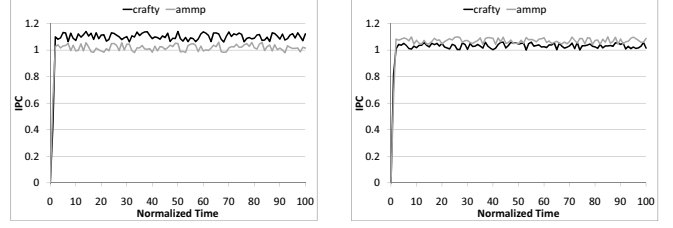
9.1 Evaluation of the FSI policy

One of the most important objectives of our control architecture is to guarantee that the achieved performance of each application is greater than or equal to the reference inputs (performance guarantee) and to improve the fair speedup metric with this constraint. Figure 5 plots the IPCs achieved with the two benchmarks of mix-1 (apsi and parser) with time (normalized over a window of 1 Billion cycles) on a 2-core CMP. Note that, mix-1 has a high cache pressure (applications with high cache demand), as shown in Table 2. Therefore, the PAN is stressed more often than the SHARP controller.

It is observed that initially, the SHARP controller quickly adapts the target IPCs to achievable targets with the FSI policy in place. Although the target IPCs are set to 1.0 for each of the benchmarks in the mix, the SHARP control architecture, assisted by the constructive interactions between the PAN and the SHARP controller, achieves average IPCs of 1.45 and 1.04 for the two applications (with the ROR controller). We would like to re-emphasize here that the PAN and the SHARP controller are performance oriented and strive to increase performance by keeping the cache utilization as high as possible. A power centric scheme could be integrated with our architecture by suitably modifying the SHARP controller, which is part of our future work.

Figure 5(a) and Figure 5(b) plot the variations in the IPC achieved using the PID controller and the ROR controller, respectively, for mix-1. We can see that both the PID controller (with design parameters $K_P = K_I = 0.8, K_D = 0.6$) and the ROR controller (with the default value of parameter $\beta = 0.6$) achieve the required performance guarantees. Overall, the ROR controller fares better by consistently achieving higher IPC values. This smooth (and consistently higher) response of the ROR controller is also responsible for achieving a higher fair speedup, as will be shown later.

As discussed earlier, one of the most important characteristics of the history based ROR controller is the smooth response *i.e.*, lesser oscillations or variations in the cache allocations with time. Figure 6(a) and Figure 6(b) plot the variations in the cache allocations with time produced by the PID controller and the ROR controller, respectively. We can



(a) SD policy ($\Theta_{crafty} = 2, \Theta_{ammpp} = 1$) (b) SD policy ($\Theta_{crafty} = 1, \Theta_{ammpp} = 2$)

Figure 7: Adaptation of IPC with time for applications of Mix-2 with the SD policy using the ROR controllers.

easily see that the allocations in case of the PID controller are highly fluctuating as compared to the allocations produced by the ROR controller. For measuring this variation, we use a metric called the variation index (VIN) defined as, is given by $\frac{\sum_{t=1}^N |w(t) - w(t-1)|}{N}$, for an application which is allocated $w(t)$ number of ways at time (t) over a time range $[0..N]$. A high VIN indicates a high degree of variation in the cache allocations to the application with time. In fact, the VIN for cache allocations in case of the PID controller for apsi and parser are 1.99 and 2.12, respectively, as against values of 0.73 and 0.77, respectively, with the ROR controller.

Owing to the lower variations in the allocations produced by the ROR controller, it also fares better across different mixes on the FS metric. Figure 6(c) plots the FS metric achieved with the PID controller and the ROR controller for the different mixes of applications. We observe an average improvement of 15.7% over the equal share case in case of the PID controller. The ROR controller achieves an average improvement of 21.9% on the FS metric over the equal share case. We can also see from Figure 6(c) that the improvements in the FS metric are higher in case of Type-II applications. This is due to the low cache pressure of Type-II applications for achieving the target IPCs.

9.2 Evaluation of the SD policy

Unlike the FSI policy which handles all applications in a fair manner, the SD policy associates weights (Θ_i , computed from application priorities) with each application of the workload. The weights influence both the partitioning determined for a given reference input (by the PAN) and the target IPCs determined by the SHARP controller. Figure 7 plots the adaptation of IPCs as observed in two scenarios with the SD policy for applications of Mix-2 (on 2-core CMP). Figure 7(a) plots the scenario where the benchmark crafty is given a higher priority than ammp. Figure 7(b) plots the other scenario where the benchmark ammp is given a higher priority than crafty. As seen from the figures, the SD policy successfully tracks the reference inputs. It can also be seen that the application with higher weight derives better IPC in general, although the difference is more striking in the case where crafty is given a higher priority. This can be explained by observing that the IPC of crafty approaches its target IPC with fewer number of cache ways than ammp, thereby leading to relatively higher estimates for $P_i^*(t)$ computed by the SHARP controller.

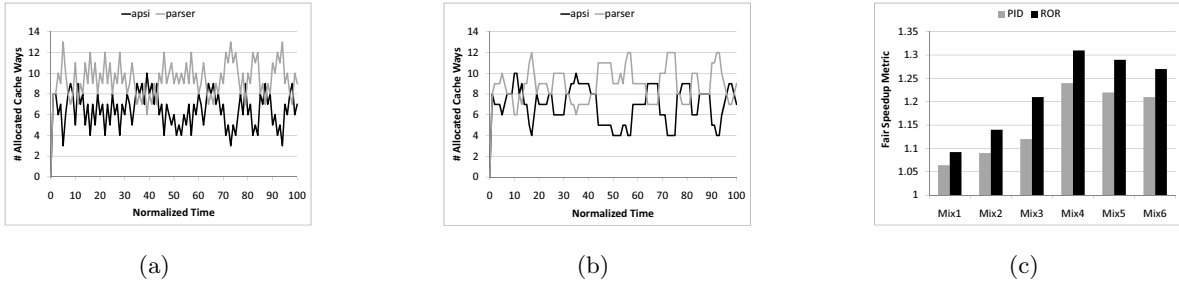


Figure 6: (a) Number of cache ways allocated with time for applications of Mix-1 with the PID ($K_P = K_I = 0.8, K_D = 0.6$), (b) Number of cache ways allocated with time for applications of Mix-1 with the ROR controllers ($\beta = 0.6$). (c) Fair speedup metrics achieved with various mixes (see Table. 2) with the PID and ROR controllers.

9.3 Significance of the SHARP Controller

The role of the SHARP controller is crucial in implementing the service level objectives in terms of the policies. It is important to note that the SHARP controller would not be necessary if the only objective of our SHARP control architecture was to track specified input parameters like IPC. However, our objective is not limited to providing a performance guarantee in terms of IPC, but extends to achieving a higher FS metric. The role of the SHARP controller is crucial in distributing the unutilized cache ways when the demanded by applications is lesser than the available number of ways in the cache. In order to study the role of the SHARP controller in our architecture, we measured the adaptation of IPC in a system with no SHARP controller for a mix of applications (Mix-1). The result is plotted in Figure 8(a). It is easy to see that the inputs (IPC values of 1.0 for each application) are in fact tracked even with a system without the SHARP controller, albeit with an average fair speedup of 0.92 and an average cache utilization of 87.2%. In contrast, in presence of SHARP controller we observed an FS value of 1.09 and cache utilization of 99.3% with the achieved IPC's as high as 1.45 and 1.04 respectively for apsi and parser respectively (see Figure 5(b)).

9.4 Sensitivity to Reference IPC

The adaptive nature of SHARP control architecture enables it to be resilient (less sensitive) to the specified reference IPCs that serve as a seed input to the system. This is because the reference IPCs to the SHARP controller do not directly serve as inputs to the AppC controllers but only serve as a seed to target IPCs computed repetitively in a feedback loop by the SHARP controller (using Eq. (8) or Eq. (9)). However, this does not mean that the performance of the SHARP control architecture is independent of the specified reference inputs. More specifically, we can see from Eq. (8) and Eq. (9) that the IPC value computed by the SHARP controller is sensitive to the proportions of the reference IPCs, *i.e.*, as long as the proportions of the specified IPCs remain the same the SHARP control architecture quickly adapts to the best possible IPC (according to the PAN policy). Figure 8(b) plots the response of our SHARP control architecture to reference IPCs of (0.5, 0.5) to apsi and parser (default reference IPCs of 1.0 and 1.0 respectively). Since the proportions of reference IPCs remains the same, the achieved performance is also very similar (compare Figure 5(b) and Figure 8(b)). Apsi and parser achieve average IPCs of 1.42 and 1.01 respectively. In con-

trast, when the proportions of the specified reference IPCs are biased towards one application, the observed behavior is similar to prioritizing that application. Again, this can be explained from Eq. (8) and Eq. (9). Figure 8(c) plots the response of our architecture to specified reference IPCs of 1.0 and 0.5 respectively to apsi and parser (proportion biased towards apsi). In this case, apsi achieves an average IPC of 1.49 while parser achieves an average IPC of 0.88, indicating the prioritization of apsi over parser. However, note that the SHARP control architecture is resilient enough to keep the performance of parser fairly high as compared to the specified reference IPC. If a disproportionate bias is used to prioritize one application over the other, the FSI policy starts approaching the SD policy.

9.5 Scalability and Significance of PAN

The PAN is an integral part of the SHARP control architecture. Unlike the SHARP controller, it is necessary to have PAN in order to determine feasible partitions when the system is subjected to workloads with high cache demand. The behavior of PAN also determines the scalability of our approach to cache intensive workloads. The advantage of the control theory will become more pronounced when workloads as well as CMPs are more complex. In order to measure the scalability of the SHARP control architecture to workloads with larger number of applications and processor cores, we experimented with two workload mixes of 8 applications executing on 8 cores (formed by combining mixes 1, 2, 4, and 5). The results are plotted in Figure 8(d). It can be seen that, in spite of using highly cache intensive workloads (with a combined cache pressure of 42 ways in a 32 way set associative cache), the SHARP control architecture is able to track the targets. We can observe from Figure 8(d) that our SHARP control architecture can effectively scale to larger CMPs without compromising on its ability to provide guarantees on tracking as long as the target specifications are feasible. It is important to note that over-specification of targets (even of one application) is only going to hurt all the applications in a fair manner in case of the FSI policy and in a differentiated manner based on the weights in case of the SD policy.

10. RELATED WORK

Researchers have explored CMP cache partitioning heuristics that attempt to alleviate inter-thread conflicts at the shared last level cache [29, 32, 20, 28, 6, 17, 14, 15]. Rafique et al [29] proposed an OS scheme that implements a variety

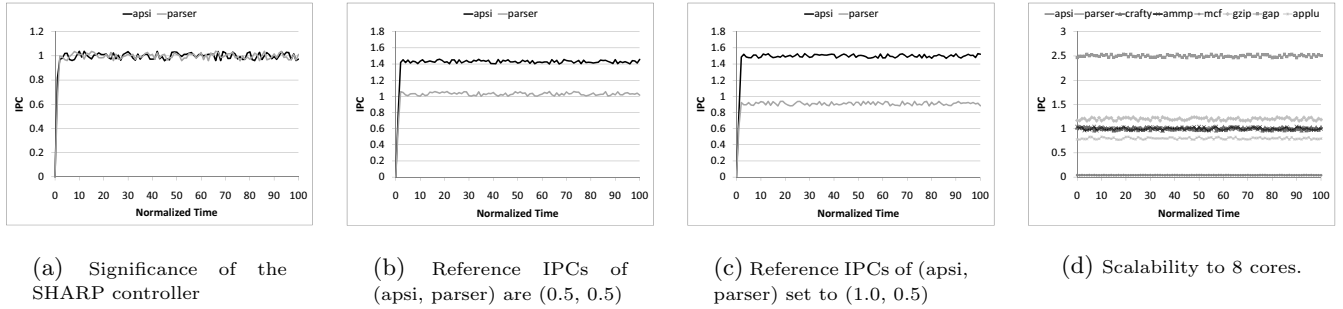


Figure 8: Results of the sensitivity analysis experiments. The 8 applications in (d) are chosen from Mix-1, Mix-2, Mix-4 and Mix-5. Note that the distinction between four applications (apsi, parser, crafty, and ammp) may not be clear as they all have a target IPC of 1.0.

of policies including those for fair cache sharing and achieving differentiated instruction throughput. In comparison, we use feedback control theory and therefore can provide guarantees about adapting to varying workloads.

Chang and Sohi [6] have proposed cooperative cache partitioning, wherein they use multiple time sharing partitions to resolve cache contention. Qureshi and Patt [28] proposed a utility based cache partitioning scheme where the share received by an application was proportional to the utility rather than its demand. Figure 9 plots the fair speedups achieved with various mixes on our default configuration using these two schemes (denoted CoopPart and Utility) in comparison with that obtained using the two controllers proposed in this paper (PID and ROR) using our SHARP control architecture. As seen from the figure, the ROR controller consistently outperforms the heuristic schemes. The PID controller also outperforms the heuristic schemes in the mixes with high combined cache pressure (mix-1, mix-2 and mix-3) while the heuristic schemes have comparable performance to the PID control in case of mixes with low combined cache pressure (specifically mix4 and mix5). It is important to note that the real benefit of using a control theoretic approach lies in its strong formal methods and the resultant deterministic behavior rather than in providing significant improvements over the other heuristic proposals. We believe that by relying on mature formal methods to provide theoretical guarantees about execution characteristics (see Section 4), we have taken the first step to providing a solution that can scale to handle the multi-resource / multi-objective optimization problems. In contrast, most previous works in this area relied on ad-hoc techniques.

Fair Queuing Memory System [26] proposed by Nesbit et al uses basic network fair queuing principles in order to provide QoS and fairness to threads competing for SDRAM memory system bandwidth. This was further extended by proposing virtual private machine (VPM) [27] as an overall framework that extends across CMP subsystems. Providing QoS in shared caches in CMPs has also been widely studied in the recent past [10, 17, 14, 15]. [15] and [10] examine the feasibility of providing QoS in CMPs through prototypes that allow experimentation with QoS-aware execution environments and architectural resources. Iyer et al [17] have recently proposed a QoS-enabled memory architecture. This memory architecture enables dynamic memory resource management for high priority applications based on guidance from the operating environment.

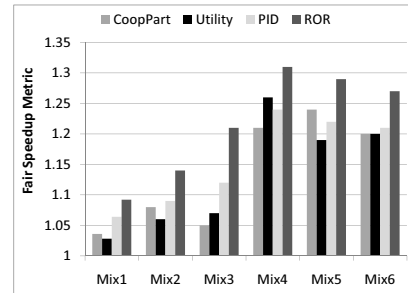


Figure 9: Comparison of fair speedup metric obtained by different controllers (PID and ROR) of SHARP control with state-of-the-art heuristic schemes (co-operative cache partitioning and utility based cache partitioning).

Most control-theoretic approaches for server applications used system-identification based linear models and classical PID control [2, 11]. Researchers have also applied PID control to performance management of Internet Web servers [1] and real-time scheduling [34]. There have been various applications of the PID control to the management of storage and database servers [23] and storage caches [21]. Ko et al [21] discuss a QoS architecture for a shared storage proxy cache to provide long-term hit rate assurances to competing QoS classes. Prior research has also considered using feedback control in the context of hardware and software directed power and temperature management [30, 33, 19].

11. CONCLUSIONS AND FUTURE WORK

Lack of control over management of shared resources can lead to lack of QoS. This paper discussed the benefits of formal control theory for partitioning the shared last level cache in CMPs among multiple concurrently executing applications. We presented the SHARP control architecture that controls partitioning of shared on-chip cache (L2) capacity with different objectives such as improving fair speedup or providing service differentiation and presented experimental results targeting 2-core and 8-core systems. Our fair speedup improvement scheme achieves 21.9% improvement on the fair speedup metric and the service differentiation scheme achieves well regulated service differentiation. We also showed that using our control architecture maximizes cache space utilization. Our future work includes extending the SHARP control architecture to include power aware poli-

cies and an integrated operating system directed processor-cache partitioning system using feedback control theory.

12. REFERENCES

- [1] T. F. Abdelzaher, K. G. Shin, and N. Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Trans. Parallel Distrib. Syst.*, 13(1):80–96, 2002.
- [2] T. F. Abdelzaher, et al. Feedback performance control in software services. *IEEE Control Systems Magazine*, 23(3), 2003.
- [3] K. Astrom and T. Hagglund. The future of PID control. *Control Engineering Practice*, 9(11):1163–1175, 2001.
- [4] E. F. Camacho and C. Bordons. *Model Predictive Control*. Springer, Berlin, 2004.
- [5] D. Chandra, et al. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proc. of the International Symposium on High-Performance Computer Architecture*, 2005.
- [6] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proc. of the International Conference on Supercomputing*, 2007.
- [7] A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proc. of the International Conference on Parallel Architecture and Compilation Techniques*, 2007.
- [8] G. F. Franklin, M. L. Workman, and D. Powell. *Digital Control of Dynamic Systems*. Addison-Wesley Longman Publishing Co., Inc., 1997.
- [9] F. Guo, et al. From chaos to QoS: case studies in cmp resource management. *SIGARCH Comput. Archit. News*, 35(1):21–30, 2007.
- [10] F. Guo, et al. A framework for providing quality of service in chip multi-processors. In *Proceedings of the International Symposium on Microarchitecture*, 2007.
- [11] J. Hellerstein. Challenges in control engineering of computer systems. In *Proceedings of the American Control Conference*, 2004.
- [12] J. L. Hellerstein, et al. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.
- [13] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer*, 33(7):28–35, 2000.
- [14] L. R. Hsu, et al. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, 2006.
- [15] R. Iyer. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *Proc. of the International Conference on Supercomputing*, 2004.
- [16] R. Iyer, et al. Datacenter-on-chip architectures: Tera-scale opportunities and challenges. *Intel Technology Journal*, 11(03), 2007.
- [17] R. Iyer, et al. QoS policies and architecture for cache/memory in CMP platforms. *SIGMETRICS Perform. Eval. Rev.*, 35(1):25–36, 2007.
- [18] A. Jaleel, et al. Adaptive insertion policies for managing shared caches. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2008.
- [19] R. Joseph, D. Brooks, and M. Martonosi. Control techniques to eliminate voltage emergencies in high performance processors. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2003.
- [20] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, 2004.
- [21] B.-J. Ko, et al. Scalable service differentiation in a shared storage cache. In *Proceedings of the International Conference on Distributed Computing Systems*, 2003.
- [22] O. Lequin et al. Iterative feedback tuning of PID parameters: comparison with classical tuning rules. *Control Engineering Practice*, 11(9), 2003.
- [23] C. Lu, G. A. Alvarez, and J. Wilkes. Aqueduct: Online data migration with performance guarantees. In *Proceedings of the Conference on File and Storage Technologies*, 2002.
- [24] P. S. Magnusson, et al. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.
- [25] M. M. K. Martin, et al. Multifacet’s general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4), 2005.
- [26] K. J. Nesbit, et al. Fair queuing memory systems. In *Proceedings of the International Symposium on Microarchitecture*, 2006.
- [27] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. *SIGARCH Comput. Archit. News*, 35(2):57–68, 2007.
- [28] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. of the International Symposium on Microarchitecture*, 2006.
- [29] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven CMP cache management. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, 2006.
- [30] R. Raghavendra, et al. No “power” struggles: coordinated multi-level power management for the data center. In *Proceedings of the International conference on Architectural Support for Programming Languages and Operating Systems*, pages 48–59, 2008.
- [31] X. Shi, et al. CMP cache performance projection: accessibility vs. capacity. *SIGARCH Comput. Archit. News*, 35(1):13–20, 2007.
- [32] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *J. Supercomput.*, 28(1):7–26, 2004.
- [33] Q. Wu, et al. Formal control techniques for power-performance management. *IEEE Micro*, 25(5):52–62, 2005.
- [34] R. Zhang, et al. Controlware: A middleware architecture for feedback control of software performance. In *Proceedings of the International Conference on Distributed Computing Systems*, 2002.