

# Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach

**Ramazan Bitirgen**  
Computer Systems Laboratory  
Cornell University  
Ithaca, NY 14853 USA  
bitirgen@csl.cornell.edu

**Engin İpek**  
Microsoft Research  
Redmond, WA 98052 USA  
ipek@microsoft.com

**José F. Martínez**  
Computer Systems Laboratory  
Cornell University  
Ithaca, NY 14853 USA  
martinez@cornell.edu

**Abstract**—Efficient sharing of system resources is critical to obtaining high utilization and enforcing system-level performance objectives on chip multiprocessors (CMPs). Although several proposals that address the management of a single microarchitectural resource have been published in the literature, coordinated management of multiple interacting resources on CMPs remains an open problem.

We propose a framework that manages multiple shared CMP resources in a coordinated fashion to enforce higher-level performance objectives. We formulate global resource allocation as a machine learning problem. At runtime, our resource management scheme monitors the execution of each application, and learns a predictive model of system performance as a function of allocation decisions. By learning each application’s performance response to different resource distributions, our approach makes it possible to anticipate the system-level performance impact of allocation decisions at runtime with little runtime overhead. As a result, it becomes possible to make reliable comparisons among different points in a vast and dynamically changing allocation space, allowing us to adapt our allocation decisions as applications undergo phase changes.

Our evaluation concludes that a coordinated approach to managing multiple interacting resources is key to delivering high performance in multiprogrammed workloads, but this is possible only if accompanied by efficient search mechanisms. We also show that it is possible to build a single mechanism that consistently delivers high performance under various important performance metrics.

## I. INTRODUCTION

As industry rides the transistor density growth in chip multiprocessors (CMPs) by providing more and more processor cores, these will exert increasing levels of pressure on shared system resources. Efficient resource sharing becomes critical to obtaining high utilization and enforcing system-level performance objectives on CMP platforms.

Unrestricted sharing of microarchitectural resources can lead to destructive interference. Although several proposals that address the management of a single microarchitectural resource have been published in the literature, proposals to manage multiple interacting resources effectively on CMPs are much more scarce.

Consider, for example, the case of a CMP where the on-chip L2 cache space, off-chip bandwidth, and the chip’s power budget are shared among applications, and the usage of each resource is regulated via an independent Quality of Service (QoS) knob. It is easy to see that, if the allocation of a resource changes, the application’s demands on the other resources also change. For example, increasing an application’s allocated cache space may cause its working set to fit in the cache, and can dramatically reduce its off-chip bandwidth demand

(which could in turn be allocated to other applications with higher demand). Similarly, increasing an application’s power budget could cause it to run at a higher frequency, and to demand more bandwidth. Hence, the optimal allocation of one resource type depends in part on the allocated amounts of other resources, requiring a coordinated resource management scheme for optimal performance.

Figure 1 shows an example of performance loss due to uncoordinated resource management in a CMP incorporating three QoS knobs for regulating the system’s shared cache, off-chip bandwidth, and power budget. A four-application, desktop-style multiprogrammed workload is executed on a quad-core CMP with an associated DDR2-800 memory subsystem. Performance is measured in terms of weighted speedup (ideal weighted speedup here is 4, which corresponds to all four applications executing as if they had all the resources to themselves). Configurations that dynamically allocate one or more of the resources in an uncoordinated fashion (*Cache*, *BW*, *Power*, and combinations of them) are compared to a static, fair-share allocation of the resources (*Fair-Share*), as well as an unmanaged sharing scenario (*Unmanaged*), where all resources are fully accessible by all applications at all times. (Section IV gives the details of our experimental setup, the resource allocation mechanisms borrowed from the literature, and the target performance metrics.)

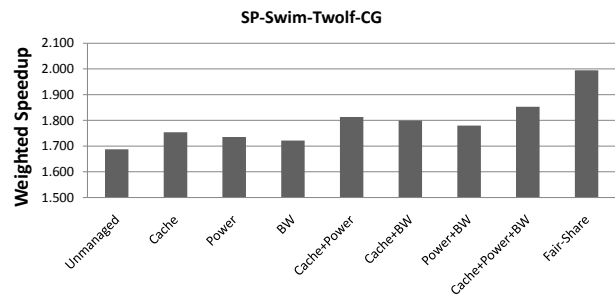


Fig. 1. Weighted Speedup of different resource management schemes.

In this workload, unmanaged resource sharing delivers considerable slowdowns, even when compared to a rigid, static resource distribution among the cores (*Fair-Share*). By introducing dynamic resource management, one would hope to exceed the performance of *Fair-Share*. But the figure shows that managing the resources in an uncoordinated fashion

is still inferior to simple static partitioning, indicating that resource interactions render individual adaptive management policies largely ineffective. (Section V-A shows results for other workloads. The trends are largely the same.)

In this paper, we address these limitations by proposing a resource allocation framework that manages multiple shared CMP resources in a coordinated fashion to enforce higher-level performance objectives. We formulate global resource allocation as a machine learning problem. At runtime, our resource management scheme monitors the execution of each application, and learns a predictive model of system performance as a function of allocation decisions. By learning each application's performance response to different resource distributions, our approach makes it possible to anticipate the system-level performance impact of allocation decisions at runtime. As a result, it becomes possible to make reliable comparisons among different points in a vast and dynamically changing allocation space, allowing us to adapt our allocation decisions as applications undergo phase changes.

## II. BACKGROUND

Software applications generally have different hardware resource requirements, and exhibit varying sensitivity to resource allocation. Often times, this also applies to phases within the same application. Extracting the highest performance out of multiple applications that compete for shared hardware resources on a CMP necessitates a mechanism by which such shared resources are (periodically) partitioned in a productive manner. Three critical resources in virtually any CMP design are (a) shared cache(s), (b) the chip's power budget, and (c) off-chip memory bandwidth. Most existing work in architectural support for managing these resources focuses only on one resource; very few proposals attempt to address allocation of several of these resources in a coordinated fashion. In this section, we discuss some of that work.

### A. Cache Partitioning

Liu et al. [14] propose a profile-based static partitioning technique that tries to straddle shared and private L2 caching. A hardware table records which L2 banks each core can access. The goal is to maximize the overall throughput. They show that, when sharing across threads is scarce, a mostly-private configuration is preferable; otherwise, a mostly-shared configuration tends to perform better.

Suh et al. [25] propose a dynamic partitioning technique that distributes the L2 at the granularity of cache ways to improve system throughput. Their allocation policy relies on a greedy heuristic that assigns cache ways to the application that derives the highest additional benefit from the assignment. Later, Qureshi et al. [21] propose utility based cache partitioning (UCP), which improves upon Suh et al. [25]'s allocation policy by estimating the marginal utility of assigning additional cache ways to an application more accurately. We also partition the L2 cache at the granularity of cache ways, but use a global, ML-based resource distribution framework to arrive at the allocation decisions.

Yeh and Reinman [27] distribute a NUCA L2 cache in a CMP at the granularity of independent cache clusters. Their goal is to provide quality-of-service (QoS) guarantees to each core while maximizing overall system performance. Their allocation policy statically assigns a portion of the cache to each application to deliver QoS, and partitions the remaining cache space to optimize aggregate system performance.

Kim et al. [12] propose a partitioning technique to equalize the slowdowns of equal-priority applications at runtime, and Iyer [11] proposes the CQoS framework to address QoS in shared caches. Rafique et al. [22] propose a hardware-based cache management mechanism that enables the OS to enforce quotas on shared caches to provide performance differentiation and fairness across applications.

### B. Bandwidth Management

Rixner et al. [24] perform a design space study of various DRAM scheduling policies, finding the FR-FCFS scheduling policy to provide the best performance on average. Among all DRAM commands that are ready to issue, FR-FCFS prioritizes (1) ready commands over commands that are not ready, and (2) older commands over younger ones. Later, Nesbit et al. [19] demonstrate that FR-FCFS can lead to QoS and fairness problems if used in the context of multiprogrammed workloads, and propose the Fair Queueing Memory Scheduler (FQMS) to address these limitations. FQMS partitions a CMP's available off-chip DRAM bandwidth among applications by providing an OS-specified minimum service guarantee to each thread. Policies for allocating the off-chip bandwidth to meet system-level performance objectives are not explored, and allocation decisions are left to the OS. We use FQMS as a mechanism to enforce the bandwidth allocation decisions made by our framework.

Mutlu et al. [17] propose stall time fair memory scheduling (STFM), a technique that provides QoS to applications sharing the DRAM by equalizing the slowdowns that are observed by equal-priority threads. Later, the same authors improve upon STFM via batch scheduling [18], in which groups of outstanding DRAM requests from a given thread form the fundamental unit of scheduling. Neither of these works addresses the dynamic partitioning of off-chip bandwidth, and similar to FQMS, they leave allocation decisions to the OS.

### C. Power Management

Isci et al. [10] propose global power management techniques for throughput optimization, prioritization, and fairness in CMPs. Their highest-performing allocation policy (maxBIPs) optimizes throughput by estimating the performance impact of per-core DVFS actions based on an analytical model. We also rely on per-core DVFS to enforce power allocation decisions, but perform the decision-making using our global allocation framework.

### D. Coordinated Management

Choi and Yeung [5] propose a coordinated resource allocation scheme to distribute microarchitectural resources in an SMT processor (issue queue, ROB, and register file) among simultaneously executing applications, using hill-climbing to decide on resource distributions. This is done without the aid of any learning models, and thus search trials are measured in execution as opposed to queried. Compared to our proposal, this is likely to result in high overheads to navigate the search space. Moreover, responding to changes in workload demands takes time, since the search algorithm has to move from the current allocation to a new local optimum over a number of execution intervals. We port their scheme to our CMP resource allocation framework, and present a detailed performance comparison in our evaluation (Section V).

Vardhan et al. [26] propose a hierarchical resource management technique to minimize energy consumption in mobile

environments. The proposed scheme relies on profiling runs to predict the resource usage of real-time multimedia workloads, and uses this information to pick the lowest-energy system configuration that can meet the workloads' deadlines. In contrast, our proposal relies on ANNs to learn a predictive model of system performance at runtime, and does not require profiling.

### III. PROPOSED MECHANISM

Figure 2 shows an overview of our resource allocation framework, which comprises per-application hardware performance models, as well as a global resource manager. Shared system resources are periodically redistributed between applications at fixed decision-making intervals, allowing the global manager to respond to dynamic changes in workload behavior. Longer intervals amortize higher system reconfiguration overheads and enable more sophisticated (but also more costly) allocation algorithms, whereas shorter intervals permit faster reaction time to dynamic changes.

At the end of every interval, the global manager searches the space of possible resource allocations by repeatedly querying the application performance models. To do this, the manager presents each model a set of state attributes summarizing recent program behavior, plus another set of attributes indicating the allocated amount of each resource type. In turn, each performance model responds with a performance prediction for the next interval. The global manager then aggregates these predictions into a system-level performance prediction (e.g., by calculating the weighted speedup). This process is repeated for a fixed number of query-response iterations on different candidate resource distributions, after which the global manager installs the configuration estimated to yield the highest aggregate performance.

Successfully managing multiple interacting system resources in a CMP environment presents several challenges. The number of ways a system can be partitioned among different applications grows exponentially with the number of resources under control, leading to over one billion possible system configurations in our quad-core setup with three independent resources. Moreover, as a result of context switches and application phase behavior, workloads can exert drastically different demands on each resource at different points in time. Hence, optimizing system performance requires us to quickly determine high-performance points in a vast allocation space, as well as anticipate and respond to dynamically changing workload demands.

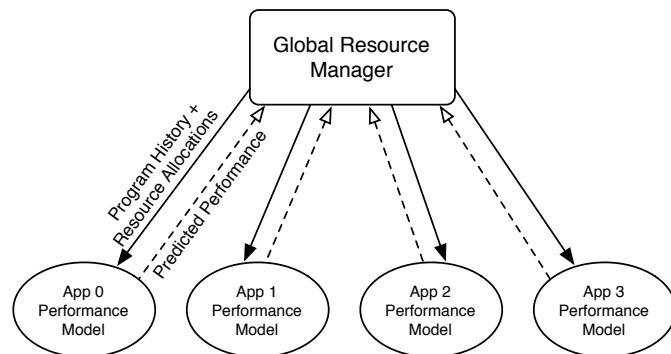


Fig. 2. Interaction of the global resource manager with each of the performance models (ANNs).

#### A. Predicting Performance

Comparing different allocation options requires the ability to predict each application's performance as a function of the resources allocated to it. In this section, we describe our performance prediction mechanism based on Artificial Neural Networks (ANNs).<sup>1</sup>

##### 1) Artificial Neural Networks

Artificial Neural Networks (ANNs) are machine learning models that automatically learn to approximate a target function (application performance in our case) based on a set of inputs. Figure 3 shows an example ANN consisting of 12 input units, four hidden units, and an output unit. In a fully-connected feed-forward ANN, an input unit passes the data presented to it to all hidden units via a set of weighted edges. Hidden units operate on this data to generate the inputs to the output unit, which in turn calculates ANN predictions. Hidden and output units form their results by first taking a weighted sum of their inputs<sup>2</sup> based on edge weights, and by passing this sum through a non-linear *activation function* (Figure 4). Increasing the number of hidden units in an ANN leads to better representational power and the ability to model more complex functions, but increases the amount of training data and time required to arrive at accurate models. ANNs represent one of the most powerful machine learning models for non-linear regression; their representational power is high enough to model multi-dimensional functions involving complex relationships among variables.

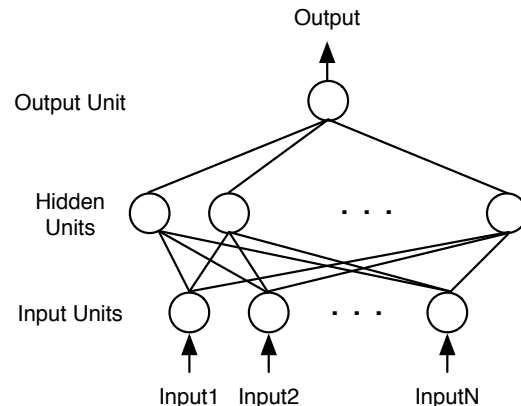


Fig. 3. Example of a feed-forward, fully-connected ANN with one hidden layer.

We propose to model each application's performance as a function of its allocated resources and recent behavior, and use an ensemble of ANNs to learn an approximation of this function (Figure 5). Input values summarizing past program behavior and indicating allocated resource amounts are presented at the input units, and performance predictions are obtained from the output units. The final performance prediction is formed by averaging the predictions of all ANNs in the ensemble, a technique that often increases accuracy over a single network, and allows us to assign confidence levels to ANN predictions.

<sup>1</sup>We also experimented with simple linear regression and locally weighted regression, finding both of them to yield higher error rates in predictions compared to ANNs.

<sup>2</sup>One of the input units, called the "bias unit", is hardwired to one.

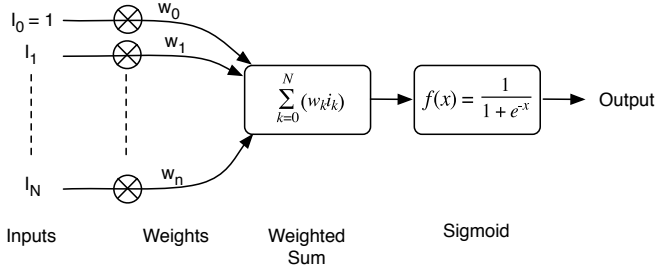


Fig. 4. Single Node of an ANN.

Each network takes as input the amount of L2 cache space, off-chip bandwidth, and power budget allocated to its application. In addition, networks are given nine attributes describing recent program behavior and current L2-cache state. These nine attributes are the number of (1) read hits, (2) read misses, (3) write hits, and (4) write misses in the L1 d-Cache over the last 20K instructions; the number of (5) read hits, (6) read misses, (7) write hits, and (8) write misses in the L1 d-Cache over the last 1.5M instructions; and (9) the fraction of cache ways allocated the modeled application that are dirty.

The first four attributes are intended to capture the program’s phase behavior in the recent past, whereas the next four attributes summarize program behavior over a longer time frame. Summarizing program execution at multiple granularities allows us to make accurate predictions for applications whose behaviors change at different speeds. Using L1 d-Cache metrics as inputs allows us to track the application’s demands on the memory system without relying on metrics that are affected by resource allocation decisions. The ninth attribute is intended to capture the amount of write-back traffic that the application may generate; an application typically generates more write-back traffic if it is allocated a larger number of dirty cache blocks.

## 2) Building the Models

To make accurate predictions, an ANN’s edge weights need to be adjusted to reflect the functional relationship between inputs (Section III-A1) and application performance. Edge weights are trained by performing successive passes over training examples. During training, each training example is presented at the input units, predictions are obtained from the output units, and the difference between actual and predicted performance is accumulated in a squared error term. After one full pass through the training set, the network’s weights are updated based on their current values and error. This process is repeated until the weights start to converge and the ANN starts to predict the training examples accurately.

One important problem that comes up when training an ANN is *overfitting*: if trained for long enough, an ANN can end up memorizing outliers in sample data, yielding an approximation with excellent accuracy on training examples, yet poor performance on new samples drawn from the same distribution. A common way of preventing overfitting is to hold aside part of the data as a test set to obtain an unbiased estimate of the ANN’s accuracy, and to stop training when the accuracy on these unbiased samples stops improving. Of course, holding aside portions of the training set for error estimation reduces the number of samples used for training, and can itself degrade model accuracy.

A common way of circumventing this limitation is *cross-validation*, a mechanism that permits the use of all available

data for training while also reducing the risk of overfitting. In cross validation, the data set is divided into  $N$  equal-sized folds, and an ensemble consisting of  $N$  ANN models is built. Each ANN in the ensemble is trained on  $N-1$  folds and tested on the remaining fold, and the test fold for each ANN differs from the other models. Once training is complete, application performance is predicted by averaging the predictions of all ANNs in the ensemble. As a result, all of the data is used for both training and testing purposes, and at the same time, none of the models is simultaneously tested and trained on the same data. In our evaluation, we use four-fold cross validation .

## 3) Dealing with Prediction Error

Inevitably, the ANNs will not always be able to predict code behavior with sufficient accuracy. Blindly supplying such predictions to the global resource manager is likely to lead to poor allocation decisions. (Preliminary experiments indeed showed that blind predictions were very damaging to our framework’s performance.) Instead, if these cases were identified and flagged properly, the global manager could make more selective use of the information available, potentially leading to better allocation decisions, and thus performance.

Naturally, calculating prediction error is not possible without the actual performance values, which are unavailable at the time predictions are made. Instead, we employ a mechanism based on local variance to estimate the prediction error. Specifically, we estimate the error of each application’s ANN ensemble as a function of the coefficient of variance (CoV), or ratio of standard deviation to the mean, of the predictions by each ANN in the ensemble. To determine the correlation between prediction error and CoV, we proceed as follows: We pick one quad-app workload for each of the 15 applications considered. We run the workload until the application under study has committed 800 million instructions (the specific resource allocation during this phase is irrelevant). Then we randomly distribute the three target resources, run for 500,000 cycles, collect one sample point, and repeat for a total of 300 sample points (150 million cycles). After that, we train our ensemble of ANNs using four 50-point folds, setting aside 100 sample points for testing. Then, we use the (trained) ANN ensemble to predict the performance in each of the 100 test points, and calculate the prediction error relative to the measured performance at that sample point, as well as the CoV (which does not depend on the actual error). Finally, we plot the CoV-error pairs, sorted by error.

Fig. 6 shows plots for four applications, representative of the plots for all 15 applications considered (Section IV). The plots show a strong linear relationship between prediction error and CoV in all cases. This is good news, because it means that we can in fact use CoV as a good estimation of the prediction error.

Empirically, we determine that a 9% prediction error cut-off point is a good trade-off. Notice, however, that because the linear coefficients vary across applications, calibration is needed in each case to determine which CoV value corresponds to that 9% prediction error cut-off point. In our experiments, we empirically observe that calibrating once per application, as described above, is sufficient in our framework to guide resource allocation throughout the execution of the workload and deliver significant gains (Section V).

## B. Searching for the Optimal Resource Allocation

Finding a resource allocation to optimize a system-level performance metric requires us to search the allocation space

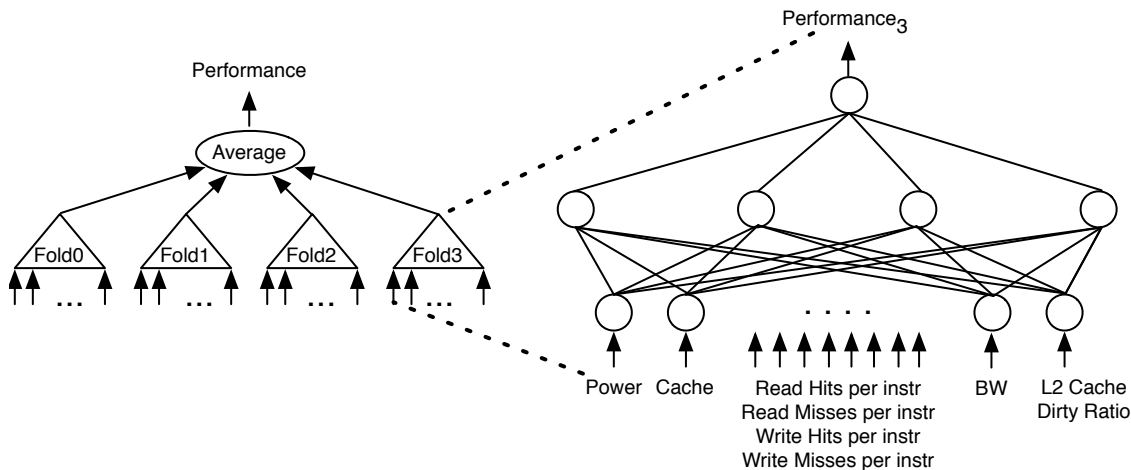


Fig. 5. Ensemble of ANNs (left) and the structure of a single ANN (right).

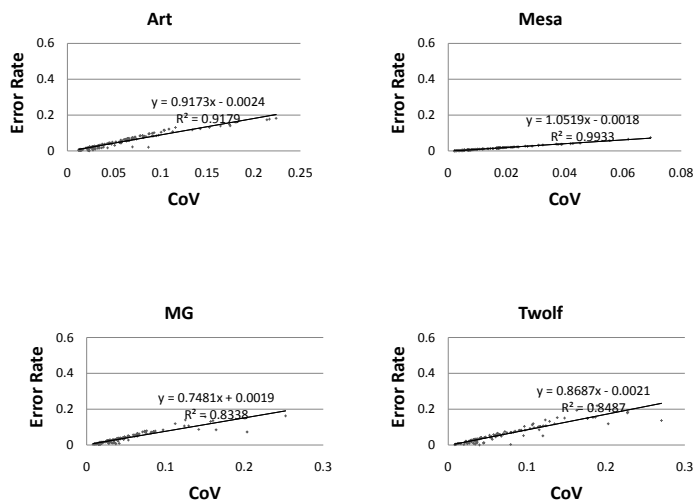


Fig. 6. Prediction error as a function of CoV on four applications.

to identify the optimal distribution. Ideally, we would like to conduct this search exhaustively on all possible combinations of resource allocations. In practice, however, even a quad-core allocation scenario with three resources can result in over one billion system configurations, rendering exhaustive search intractable. An alternative approach is to use a heuristic search algorithm to navigate a relatively small portion of the search space, and to install the allocation with the highest estimated performance found during this process. In this section, we explain how we conduct this search via a modified version of stochastic hill climbing.

### 1) Search Methodology

One important challenge that the global resource manager faces when searching the allocation space is the confidence level of the ANN models for a particular allocation query. A naive way of organizing the search would be to blindly query the ANN models for a fixed number of iterations, discarding any queries involving an application with a high error

estimate (section III-A3). Unfortunately, error rates for different applications are largely independent, and consequently, the probability of making one low-confidence prediction (or more) for a given interval increases exponentially with the degree of multiprogramming. For instance, for a quad-core workload where each application has low error rates on 90% of the points in the allocation space, this blind approach would discard roughly one third of all trials during search. Moreover, in cases where the performance of one application in the workload is consistently predicted with high error rates, resource distribution among the remaining three applications would become infeasible.

To address these limitations, we focus the attention of our search algorithm on specific subregions of the global allocation space. Each subregion restricts resource allocation optimization to a subset of the applications, and systematically gives the other applications their fair share. This results in a total of eleven possible subregions.<sup>3</sup>

At the beginning of each interval, the global resource manager queries the ANN models to obtain each application's baseline, fair-share performance. If the baseline performance prediction is estimated to be inaccurate for an application (high CoV), that application is automatically assigned its fair share, and subregions where this application is eligible to participate in global resource allocation are deemed illegal for the next interval. Next, the total number of trials that can be afforded during search are partitioned equally among all legal subregions, and the algorithm searches each subregion for the corresponding number of trials. In our evaluation, we find a total of 2,000 trials to produce the best tradeoff between search performance and overhead.

At each step of the search, the global manager tries a random resource allocation within the current subregion with a small probability  $\epsilon$ . Otherwise, the global manager successively calculates performance for every neighbor of the highest-performing point found so far. This is accomplished by querying all relevant ANNs, and aggregating their predictions in a single, system-level performance metric. If the performance prediction for any of the applications participating in

<sup>3</sup>Notice that the subregions are not disjoint; for example, if we define E and N to mean Eligible and Not Eligible for tuning of resources, respectively, it is immediate that  $NNNN \subseteq NNEE$ ,  $NNNN \subseteq NEEN$ , and  $NNEE \cup NEEN \subseteq NEEE$ .

global resource allocation yields a high error estimate, the point gets discarded and the search moves on to the next point. As soon as a point with higher estimated performance is found, the search shifts to the neighbors of this new estimated optimum. After the search terminates, the allocation with the highest estimated performance among all evaluated points is installed for the next interval.

## 2) Sampling Training Data

Sampling representative training data is essential to capturing program behavior and training accurate models. While ANNs generally perform well on program phases they have been trained on, execution of previously unobserved code regions not represented in the training set can render the ANNs obsolete. To avert such situations, the models need to be periodically re-trained on fresh training samples at runtime.

We sample points by distributing resources uniformly randomly over the allocation space, recording system configurations and performance results as training instances. In steady state, we keep a training set consisting of 300 points at all times. When a new point is sampled, one of the existing points is removed from the training set via random replacement. As a result, the training set contains points spread across a relatively long period of time, while always retaining the property that newer samples have a higher probability of inclusion in the training set than older ones.

Minimizing noise by reducing performance fluctuations due to cross-application interference can improve the quality of sampled points and dramatically improve model accuracy. One potential source of noise with our baseline QoS knobs is the Fair Queueing Memory Controller, which allows applications to temporarily exceed their allocated quota and use otherwise unutilized, excess system bandwidth. While this improves system throughput, it also makes an application's performance dependent on other applications' resource allocations and demands, which are unobservable to the application's ANN model. We address this problem by adding a *sampling mode* to the Fair Queueing DRAM scheduler. When in this mode, an application is prevented from exceeding its quota and utilizing excess system bandwidth.<sup>4</sup> While this inevitably results in lower performance during sampling intervals, it significantly improves the quality of the sampled points and results in higher accuracy.

Distributing resources randomly, as well as operating the memory controller in sampling mode, can lead to performance degradation when collecting training data. To minimize overhead, it is important to avoid excessive sampling. In our evaluation, we have empirically determined that sampling a new training instance on every fifth interval yields a good tradeoff between sampling overhead and model accuracy.

## C. Hardware Implementation

We now describe the on-chip hardware implementation of our proposal. Recall that each of the four applications is modeled using four ANNs, for a total of 16. Each ANN takes 12 inputs, which are fed to four hidden units, which in turn feed a single output unit. (Recall also that each ANN edge has an assigned weight.) We derived area and power estimations for the ANNs (described later in this section), and concluded that implementing all 16 ANNs on silicon was wasteful and,

<sup>4</sup>This is accomplished by preventing the issue of requests until their virtual finish time matches the global clock in the Fair Queueing Memory Controller.

using reasonable delay assumptions, probably unnecessary. Instead, we propose to implement a single hardware ANN, and multiplex edge weights on the fly to achieve 16 "virtual" ANNs. We now explain the hardware setup for searching and training.

**Search.** In our ANN implementation, each hidden (output) unit works as follows: 12 (4) inputs are pairwise multiplied with as many edge weights (52 multipliers total).<sup>5</sup> The results are then added together, and the sum is used to index a 50-entry-table-based quantized sigmoid function. Notice that both multiplication and addition can be pipelined easily, which allows us to supply a new set of inputs every cycle. To evaluate a query point in the hill-climbing algorithm, we feed the 16 virtual ANNs back to back, setting inputs and selecting weights appropriately in each cycle. Since we cannot compute the prediction until all 16 points have fully gone through, we choose to begin processing the next query point right away.<sup>6</sup> As a result, our hardware manages to complete an ANN query every cycle, or an entire query point in 16 cycles. Once all 16 virtual ANNs have responded, the global resource allocator computes the performance prediction, and then applies a hill-climbing step.

**Training.** Training the (virtual) ANNs requires some additional hardware, since it not only involves the querying but also updating the ANN edge weights through backpropagation. Fortunately, the update procedure can leverage the ANN's existing 16-bit fixed-point multipliers. The backpropagation hardware needs two parameters for training the neural nets: (1) learning rate and (2) momentum. These can either be fixed at design time or programmable by the OS. Our experiments with different learning rate and momentum pairs show that these parameters are robust—any reasonably chosen pair of values work well across most applications.

## 1) Area, Power, and Delay Figures

**Area.** The dominant circuit in the ANN hardware itself is the 16-bit fixed-point multipliers. Hickmann et al. [8] implement a 64-bit fixed-point multiplier, and estimates the total area (including the fixed-point multiplier, round logic, left shift, datapath and the interconnects) to be  $0.65\text{mm}^2$  at 110nm CMOS technology. We apply linear scaling down to 16 bits and 65nm, which yields an area of  $0.057\text{mm}^2$  for one of the ANN's multipliers. Since hidden and output units comprise 52 multipliers, the total area devoted to this function (which dominates the ANN) adds up to  $2.964\text{mm}^2$ , or about 1.5% of a  $200\text{mm}^2$  chip.

Our proposal also requires storage for the edge weights of all 16 virtual ANNs (224 16-bit numbers per weight set). Since there are three weight sets (one for the trained ANNs, one for the ANNs under construction, and a third set for the best weights found so far for early stopping), the total number of weights is 672, or about 1.3KB worth of storage. Outside the ANN, the global resource manager requires storage for 300 sample points in steady state. Each point contains 12 16-bit

<sup>5</sup>The bias unit does not require multiplication as it is hardcoded to one.

<sup>6</sup>Notice how this is roughly equivalent to inspecting two neighboring points simultaneously in the hill-climbing algorithm, which deviates slightly from its original behavior. This incurs a small probability that the first of the two points yield a more profitable resource allocation than the current one, in which case the hill-climbing algorithm moves immediately to it and "wastes" the second query point. In practice, this probability is small per se, and even smaller when stochasticity is factored in. Consequently, in our simulation infrastructure, we do not model this subtlety.

inputs and 1 16-bit output (IPC), thus requiring about 7.6KB of storage for all 300 sample points.

When adding the multiplier and storage overheads, together with the ANN’s adders and sigmoid storage, as well as the global resource manager’s other circuitry (e.g., four fixed-point dividers to calculate weighted speedups), the total area overhead of our proposal is probably on the order of 3% of the chip’s area.

**Power.** To approximate the power consumption by the fixed-point multiplier circuits in the ANN hardware, we assume the power density of IBM Power6’s FPU [6], which at 100% utilization and nominal voltage and frequency values (1.1 V and 4 GHz) is 0.56 W/mm<sup>2</sup>. In our ANN hardware, the aggregate power consumption of all fixed-point multipliers at full utilization and nominal voltage and frequency works out to be about 1.7 W. In our experiments, we assume the power consumption of our proposed hardware to be 3 W. Specifically, the power budget for all configurations except ours is 60 W, whereas ours is limited to 57W.

The power consumption of the cores and the shared L2 cache can be monitored by isolating the Vdd lines feeding each of the cores and the shared cache, and monitoring the current draw. The power dissipation of the L2 can be distributed among cores by using a set of hit and miss counters (one per core) to track each core’s usage, and by charging each core the appropriate fraction of the total L2 cache power.

**Delay.** Our goal is to be able to process 2,000 query points during one stochastic hill-climbing search cycle while staying within 5% of an interval (i.e., stay within 25,000 cycles).<sup>7</sup> As it turns out, this is a comfortable margin in our hardware setup for any set of reasonable delays. Specifically, in our experimental setup, we assume the following basic delays: In the ANN, four cycles for a 16-bit fixed-point multiplication; 3 cycles to add up to thirteen<sup>8</sup> 16-bit numbers using a Wallace tree; and one cycle to access the table-based sigmoid function.

## 2) Scalability of the Hardware Mechanism

Notice that, in our proposal, the number of (virtual) ANNs is proportional to the number of applications running on the CMP, and not the number of cores in the CMP. We conjecture that number of applications (whether sequential or parallel) running on a CMP is likely to increase at a significantly slower rate than the increase in the number of cores (which would tend to follow Moore’s law). Since our proposed mechanism targets desktop/workstation environments, we do not believe the scalability of our proposal is a concern in the near future.

## D. OS Interface

Our proposal requires a small number of modifications to the OS kernel to enable ANN models to survive across context switches. Making an ANN model persistent in this way can greatly improve model accuracy, by eliminating the need to train a new (and initially inaccurate) model each time an application is dispatched. To accomplish this, we simply make both the training set and the ANN weight values part of the process state.

While our ML-based runtime resource allocation mechanism can optimize for a variety of different performance

<sup>7</sup>Recall that execution does not stop while the stochastic hill-climbing search is taking place. This requires the search to be relatively fast, so that the winning configuration can be installed and exploited in a timely manner.

<sup>8</sup>The weight associated to the bias unit is also included in the addition.

targets, system-level optimization goals are ultimately determined by the OS. At runtime, the OS communicates the desired objective function to our hardware resource manager by annotating a control register. In the case of weighted metrics, it is also possible for the OS to set static weights for each application based on priorities. Finally, the OS can deliver minimum service guarantees to an application by statically allocating a portion of the resources to that application. In our experiments, every application is statically allocated a minimum of one L2 cache way, 1/8 of the available DRAM bandwidth, and a 5 W power budget.

## IV. EXPERIMENTAL SETUP

### A. Metrics

We evaluate the efficacy and versatility of our proposal using four different optimization targets: (1) sum of IPCs;<sup>9</sup> (2) weighted speedup; (3) harmonic mean of normalized IPCs; and (4) weighted sum of IPCs. We now describe these metrics and the rationale behind them; in the discussion,  $IPC_i^{Shared}$  and  $IPC_i^{Alone}$  denote application  $i$ ’s IPC when it is running with other applications, and when it is running alone, respectively:

**Sum of IPCs.** Aggregate system throughput is a relevant optimization target for a number of computing environments (e.g., web servers, databases). For cases where the workloads are homogeneous and/or the latency is largely irrelevant, optimizing the sum of IPCs maximizes system performance:

$$\sum_i IPC_i^{Shared}$$

**Weighted speedup.** In other environments, such as desktop-style computing platforms, workloads often include latency sensitive applications. In such cases, targeting aggregate IPC improvement can cause unwanted degradation in some applications’ performance, for example by speeding up inherently high-IPC programs. Instead, the weighted speedup metric aims to measure the overall reduction in execution time, by normalizing each application’s performance to its inherent IPC value, which is obtained when the program runs alone:

$$\sum_i \frac{IPC_i^{Shared}}{IPC_i^{Alone}}$$

**Harmonic mean of normalized IPCs.** This metric strives to strike a balance between system-throughput and fairness [15].

$$\frac{n}{\sum_i \frac{IPC_i^{Alone}}{IPC_i^{Shared}}}$$

**Weighted sum of IPCs.** Finally, in cases where the OS can specify a set of priorities for each application in the workload mix, these priorities can be used to weigh the contribution of each application to an overall performance metric. Note that weighted speedup is a special case of this more general optimization goal, where the individual weights are determined by the applications’ inherent IPC figures. In our evaluation, we pick a case where the OS assign weights one through four to the applications running on cores zero through three, respectively.

<sup>9</sup>The IPCs are measured according to a nominal frequency (4GHz) regardless of the actual frequency of each core at each point in time.

$$\sum_i (\text{weight}_i \cdot \text{IPC}_i^{\text{Shared}})$$

## B. Architecture

We evaluate our proposal using simulation models of quad-core CMPs with a DDR2-800 memory subsystem. The microarchitectural parameters of our baseline CMP are loosely based on Intel’s Core2Quad chips, and the parameters of the DRAM subsystem are obtained from Micron’s data sheets [16]. Tables I and II list the parameters of the modeled cores and memory system, respectively. Our experiments are based on a heavily modified version of the SESC [23] simulator, augmented for dynamic and leakage power calculation similar to [13].

We use Wattch [2] to model dynamic power consumption, and employ a methodology that accounts for the exponential dependence of leakage on temperature when calculating static power. To do this, we first perform a renormalization of power values [13] across Wattch and HotSpot [9]. Next, we use HotSpot to populate a table recording average temperature as a function of average power consumption.<sup>10</sup> At runtime, we recalculate temperature at the end of every interval by consulting this table with the last interval’s average power figure. The resulting temperature estimate is in turn used to approximate the ratio of static to dynamic power for the next interval [4].<sup>11</sup>

Our baseline CMP system includes three independent knobs for regulating the sharing of the on-chip cache space, memory bandwidth, and the chip’s power budget. We distribute a 60 W power budget among four applications via per-core DVFS. We estimate the power overhead of our proposal to be less than 3 W (Section III-C1), and conservatively deduct 3 W from our global power budget when running with the ANNs.

We partition the L2 cache space at the granularity of cache ways [21]. We statically allocate one way of our 4MB, 16-way L2 cache to each application, and distribute the remaining 12 cache ways in response to changing workload demands.

Finally, we use Nesbit et al.’s hardware support to provide performance isolation at the memory controller [20]. The distribution of the off-chip bandwidth is modulated by changing the service share of each application at the fair-queuing memory controller. At any point in time, each application is statically allocated 800MB/s of off-chip DRAM bandwidth, and the remaining 3.2GB/s is distributed based on dynamic changes.

## C. Workload Construction

We experiment with nine quad-core multiprogrammed workloads, incorporating applications from the SPEC 2000 [7] and NAS [1] suites. We use reference input set for the SPEC applications and C, W, W, C problem sizes for Cg, Mg, Lu, Sp and Ep, respectively. We classify applications into three categories based on their workload behavior. *CPU-bound* applications can fit their working set into the minimum possible allocated cache space, and are thus insensitive to

changes in their cache or off-chip bandwidth shares. *Memory-bound* applications have working sets that are larger than the maximum possible cache allocation, and consequently do not benefit significantly from increases in allocated cache space. Finally, *cache sensitive* applications’ working sets are larger than the minimum L2 cache allocation, but smaller than the full cache capacity; these workloads’ pressure on the off-chip interface depends on their cache and power allocations at any point in time. Table III shows a classification of the applications used in our evaluation. To construct the workloads, we pick applications randomly from the pool of already classified applications. For example, in one workload, CG represents the memory-bound application, whereas in another workload MG represents the same class of applications. Due to the limitations of our simulation infrastructure there are not any multiple instances of the same application within the same workload. In the plots, we denote each application in a workload with a single letter (Table III).

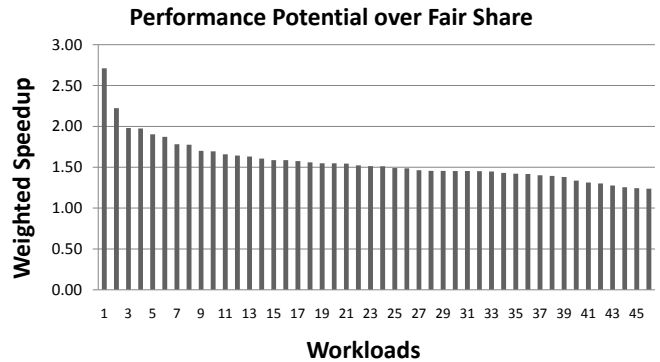


Fig. 7. Weighted speedup of fair-share vs. unmanaged resource allocation for 46 representative quad-application workloads.

We construct a total of 46 workloads, comprising all possible combinations of application types based on the classification above. We then measure the performance potential of each workload, by calculating the weighted speedup when running with a static, fair-share allocation of resources. Figure 7 shows the weighted speedups for the 46 workloads considered, in decreasing magnitude. Finally, we pick the nine workloads with the highest potential, and use them in our evaluation.

During simulation, we fast-forward each application by 800 million instructions, and warm-up the ANNs and other micro-architectural structures for another 200 million.<sup>12</sup> We then start our timing measurements, and continue recording each application’s performance until it hits its 1.2 billion committed-instruction count. When an application reaches this point in its execution, we stop measuring its performance, but nevertheless continue executing and engaging it in global resource allocation, to ensure that it continues to exert pressure on the resources shared with other applications in the workload mix.

<sup>12</sup>Because applications in a workload will generally exhibit different IPCs, high-IPC applications end up waiting at the 1-billion instruction mark for the slower ones. Notice that this may result in lower data noise for the slower applications, which are collecting samples to train their ANNs, potentially resulting in models that are artificially accurate. We carefully measured the error reduction in the ANN models due to this simulation approach, and determined it to be under 2%.

<sup>10</sup>This is accomplished by running 10 SPEC benchmarks at different voltage and frequency values, periodically measuring the average power consumption over 500K-cycle intervals, and using curve fitting to interpolate among the resulting data points.

<sup>11</sup>Early experiments revealed negligible difference between this and using HotSpot at runtime to calculate temperature.



TABLE I  
CORE PARAMETERS

Processor Parameters	
Frequency	0.9 GHz - 4.0 GHz (32 levels with 0.1 GHz increments)
Voltage	0.8 V at 0.9 GHz - 1.1 V at 4.0 GHz
Number of cores	4
Fetch/issue/commit width	4/4/4
Int/FP/Ld/St/Br Units	2/2/2/2
Int/FP Multipliers	2/2
Int/FP issue queue size	32/32 entries
ROB (reorder buffer) entries	96
Int/FP registers	80 / 80
Ld/St queue entries	24/24
Max. unresolved br.	24
Br. mispred. penalty	9 cycles min.
Br. predictor	Alpha 21264 (tournament)
RAS entries	32
BTB size	1024 entries, direct-mapped
iL1/dL1 size	32 kB
iL1/dL1 block size	32B/32B
iL1/dL1 round-trip latency	2/3 cycles (uncontended)
iL1/dL1 ports	1 / 2
iL1/dL1 MSHR entries	16/16
iL1/dL1 associativity	direct-mapped/4-way
Memory Disambiguation	Perfect
Coherence protocol	MESI
Consistency model	Release consistency

#### D. Configurations

We make performance comparisons against several previously proposed resource allocation schemes. We start with the simplest possible design, and progressively add higher levels of control on the sharing of system resources:

**Unmanaged.** This scheme leaves all shared resources unmanaged. Applications are free to access all of the on-chip L2-cache space, as well as the available off-chip bandwidth, and the chip’s power budget. Naturally, this scheme does not provide any level of performance isolation, and cannot adapt to different metrics or workloads.

**Isolated Cache Management (Cache).** This scheme implements utility-based cache partitioning as proposed by Qureshi and Patt [21], while leaving the off-chip bandwidth and power

TABLE II  
PARAMETERS OF THE SHARED L2 AND DRAM SUBSYSTEM

Shared L2 Cache Subsystem	
Shared L2 Cache	4MB, 64B block, 16-way
L2 MSHR entries	64
L2 round-trip latency	32 cycles (uncontended)
Write buffer	64 entries
DDR2-800 SDRAM Subsystem [16]	
Transaction Queue	64 entries
Peak Data Rate	6.4GB/s
DRAM bus frequency	400 MHz
Number of Channels	1
Number of Ranks	1
Number of Banks	4 per DRAM chip
Row Buffer Size	2KB
tRCD	5 DRAM cycles
tCL	5 DRAM cycles
tWL	4 DRAM cycles
tCCD	4 DRAM cycles
tWTR	3 DRAM cycles
tWR	6 DRAM cycles
tRTP	3 DRAM cycles
tRP	5 DRAM cycles
tRRD	3 DRAM cycles
tRAS	18 DRAM cycles
tRC	22 DRAM cycles
Burst Length	8

TABLE III

SIMULATED APPLICATIONS AND THEIR RESOURCE SENSITIVITY (P, C AND M STAND FOR PROCESSOR BOUND, CACHE SENSITIVE AND MEMORY BOUND, RESPECTIVELY)

Workload	Resource Sensitivity
Lu-Mgrid-Amm-Parser (LDAP)	C-C-C-C
Vpr-Art-Mcf-Ep (VRFE)	P-C-P-P
Mesa-Art-Mg-Twolf (MRGT)	P-C-M-C
Ep-Art-Swim-Mg (ERWG)	P-C-P-M
Wupwise-Art-Mcf-Mg (URFG)	P-C-P-M
Sp-Swim-Twolf-Cg (SWTC)	C-P-C-M
Mg-Art-Mcf-Cg (GRFC)	M-C-P-M
Twolf-Mg-Cg-Mgrid (TGCD)	C-M-M-C
Art-Cg-Lu-Sp (RCLS)	C-M-C-C

unmanaged. The goal is to distribute L2 cache ways to minimize the aggregate cache miss rate. While it provides a level of performance isolation along one of the dimensions, this mechanism still runs the danger of interference along the remaining two resources. Since the target optimization metric is not based on performance (but cache misses), this scheme is unable to tailor its allocation decisions to a particular metric.

**Isolated Power Management (Power).** This scheme implements the global power distribution policy proposed in [10] via per-core DVFS. The L2 cache and the off-chip bandwidth are left unmanaged, and interference is thus still possible.

**Isolated Bandwidth Management (BW).** This scheme implements the FQMS policy proposed by Nesbit et al. [19] to distribute the available off-chip bandwidth. The L2 cache and power are left unmanaged, and can thus result in interference. The off-chip bandwidth is partitioned equally among the four cores at all times.

**Uncoordinated Cache and Power Management (Cache+Power).** This scheme combines the isolated cache and power management schemes by using the corresponding performance isolation and allocation policies. Bandwidth is left unmanaged, and the two resources under control are managed in an uncoordinated fashion.

**Uncoordinated Cache and Bandwidth Management (Cache+BW).** This scheme combines the isolated cache and bandwidth management schemes while leaving power unmanaged. Interference is possible through the sharing of power, and the two resources under control are allocated without any coordination using their corresponding, proposed management mechanisms.

**Uncoordinated Power and Bandwidth Management (Power+BW).** Here, the isolated power and bandwidth management schemes are combined in an uncoordinated manner. Interference can occur through the L2 cache.

**Uncoordinated Cache, Power, and Bandwidth Management (Cache+Power+BW).** This policy merges all of the individual resource management schemes using their proposed control mechanisms, but does not provide any coordination among the individual knobs. As a result, while performance isolation is achieved, the allocation decisions tend to be sub-optimal.

**Continuous Stochastic Hill-Climbing (Coordinated-HC).** This scheme tries to port a coordinated resource allocation mechanism previously proposed in the context of SMT processors [5] to our CMP resource allocation problem. The original proposal aims to distribute microarchitectural resources in an SMT processor (issue queue, ROB, and register file) among simultaneously executing applications, using hill-climbing to

decide on resource distributions. This is done without the aid of any learning models, and thus search trials are measured in execution as opposed to queried. Compared to our proposal, this is likely to result in high overheads to navigate the search space. Moreover, responding to changes in workload demands takes time, since the search algorithm has to move from the current allocation to a new local optimum over a number of execution intervals. To cut down on the cost of the search, the proposed mechanism restricts its search to allocating all resources proportionately to each other. This makes sense in the context of resources whose demands tend to grow in proportion (e.g, register file and ROB entries), but in the context of our paper this may not be the case, and thus we also experimented with a variation of this algorithm that, like our proposal, does not impose a proportionate allocation across resources (at the expense of incurring longer searches). A third difference with our proposed mechanism is that their hill-climbing search is greedy (as opposed to stochastic). In our evaluation, we also conducted experiments using the two allocation variants described above and stochastic hill-climbing. In the end, the combination of non-proportional resource allocation and stochastic hill-climbing was systematically the best; thus, we report this configuration in our evaluation.

**Fair-Share.** This policy, to which we normalize all our results, divides each resource in equal shares across all applications. **Our Proposal (Coordinated-ANN).** This is our ANN-based proposal, where the ANN-based models of the applications' IPC response to resource allocation are used to guide a stochastic hill-climbing search.

## V. EVALUATION

### A. Performance Results

The first plot in Figure 8 shows the weighted speedup of each quad-workload, normalized to the Fair-Share configuration. In general, most configurations with uncoordinated management of resources yield performance inferior to plain Fair-Share. Surprisingly, in some instances, the uncoordinated combination of two or three resource management policies results in performance degradation with respect to applying only one of those policies. This puts into evidence the fact that not only are the effects of multiple, uncoordinated resource management policies not additive, they can actually incur destructive interference.

Indeed, Coordinated-HC generally outperforms its uncoordinated counterpart Cache+Power+BW; however, it is still inferior to Fair-Share in all cases. There are two main reasons for this: (1) While the scheme provides coordinated allocation, performing the search through direct execution leads to considerable overhead. (2) As the shape of the search space changes in response to application phases, Coordinated-HC needs to traverse a potentially expensive trajectory through the allocation space, moving from one local optimum to the next over a sequence of intervals. In cases where the new optimum is separated from the old one by a significant number of points, one of the applications may undergo a phase change before the algorithm gets a chance to converge to the new optimum.

In contrast, by estimating the shape of the allocation space in advance, and by navigating the space via queries that are essentially overhead-free, Coordinated-ANN can cope with both the exponential size and the dynamic nature of the global resource allocation problem, outperforming significantly both

Coordinated-HC and Fair-Share. Specifically, Coordinated-ANN (our proposal) delivers 14% average speedup over Fair-Share, and is consistently the best resource management scheme across all nine workloads.

These trends also carry over to the optimization of other performance targets. The second to fourth plots in Figure 8 show the harmonic mean of normalized IPCs, the sum of IPCs, and the weighted sum of IPCs metrics across the nine workloads, respectively. Similarly to the case of weighted speedup, performance isolation and coordinated management are critical to delivering high performance. Across all four metrics, Coordinated-ANN is the best-performing resource management scheme in every workload, whereas all other management schemes significantly underperform in one or more of the metrics.

Overall, these results suggest that a coordinated approach to multiple resource management is key to delivering high performance in multiprogrammed workloads. But this is possible only if accompanied by efficient search mechanisms such as our Coordinated-ANN proposal. The results also show that it is possible to build a single mechanism that consistently delivers high performance under various important performance metrics.

### B. Sensitivity to Confidence Threshold

Figures 9 and 10 show the effect of the CoV threshold on the weighted speedup and harmonic mean of IPCs metrics, respectively. Setting the CoV threshold to zero causes the confidence estimation mechanism to preclude resource allocation, and to effectively default to a fair-share configuration. As the threshold is increased, the global resource manager becomes less and less conservative in its allocation decisions, relying more on ANN predictions to manage resources. The optimal performance for both metrics is attained at a threshold of 9% in virtually every workload. As expected, higher threshold values lead to misguided allocation decisions, and thus lower performance.

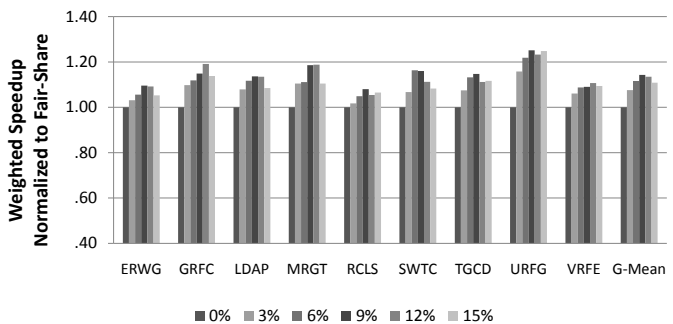


Fig. 9. Sensitivity of weighted speedup to confidence threshold. Results are normalized to Fair-Share.

### C. Confidence Estimation Mechanism

Figure 11 shows the fraction of the total execution time where the confidence estimation mechanism permits ANN predictions to guide resource allocation optimization for each application. The percentage of time an application engages in resource allocation varies depending on the other applications executing with it, indicating that resource isolation among applications is imperfect. On average, cache sensitive and memory-bound applications engage in optimized resource allocation more frequently than processor-bound applications.

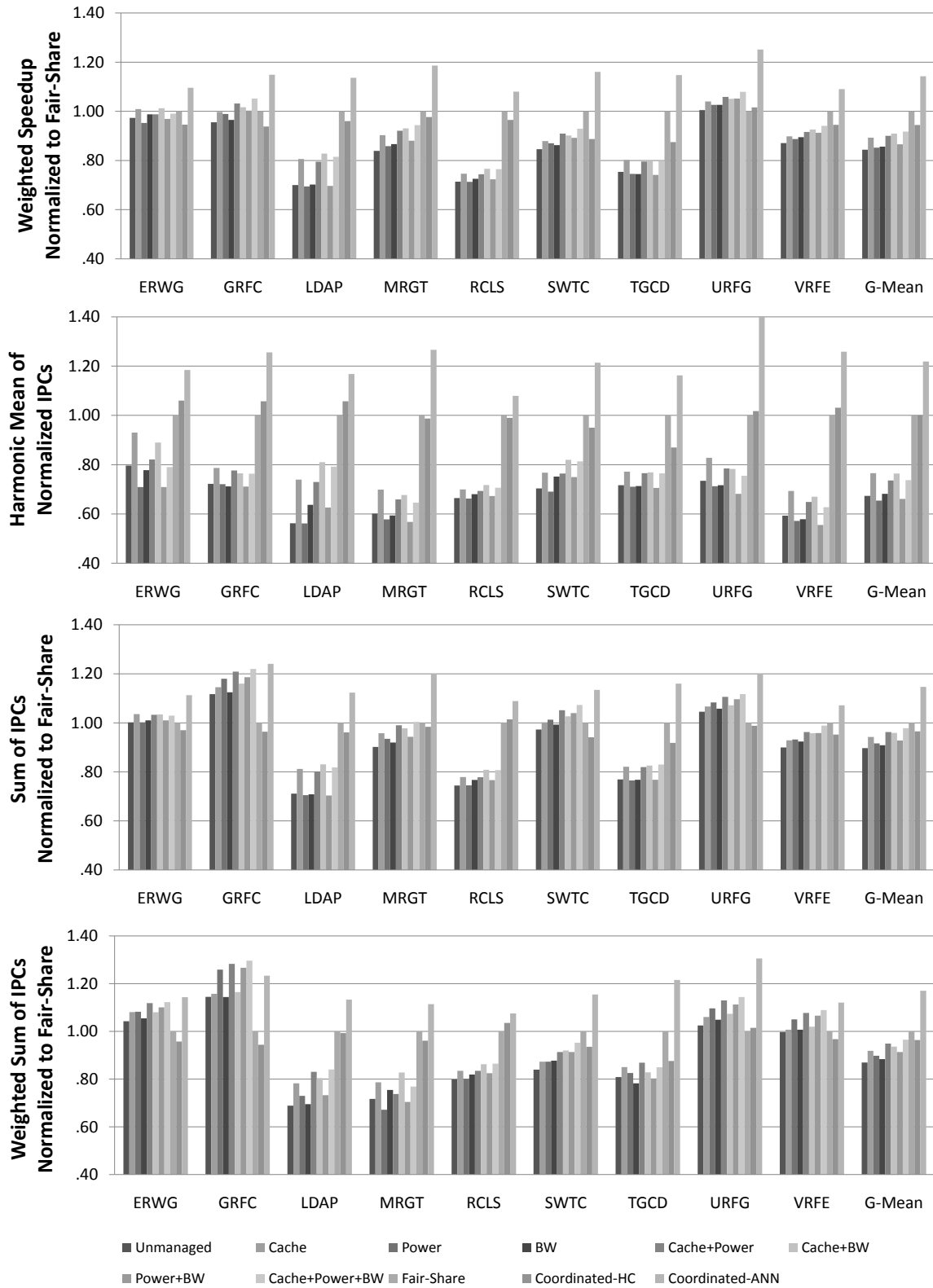


Fig. 8. Performance comparison using (top to bottom) weighted speedup, harmonic mean of normalized IPCs, sum of IPCs, and weighted sum of IPCs . Results are normalized to Fair-Share in all cases.

## REFERENCES

- [1] D. H. Bailey et al. NAS parallel benchmarks. Technical report, NASA Ames Research Center, March 1994. Tech. Rep. RNR-94-007.
- [2] D. Brooks, V. Tiwari, and M. Martonosi. Watch: A framework for architectural-level power analysis and optimizations. In *International Symposium on Computer Architecture*, pages 83–94, Vancouver, Canada, June 2000.
- [3] R. Caruana and D. Freitag. Greedy attribute selection. In *International Conference on Machine Learning*, pages 28–36, New Brunswick, NJ, July 1994.
- [4] P. Chaparro, J. González, and A. González. Thermal-effective clustered microarchitectures. In *Workshop on Temperature-Aware Computer Systems*, München, Germany, June 2004.
- [5] S. Choi and D. Yeung. Learning-based smt processor resource distribution via hill-climbing. In *ISCA-33*, 2006.
- [6] B. Curran, B. McCredie, L. Sigal, E. Schwarz, B. Fleischer, Y.-H. Chan, D. Webber, M. Vaden, and A. Goyal. 4ghz+ low-latency fixed-point and binary floating-point execution units for the power6 processor. In *ISSCC*, 2006.
- [7] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.
- [8] B. Hickmann, A. Krioukov, M. Schulte, and M. Erle. A parallel ieee p754 decimal floating-point multiplier. In *ICCD*, 2007.
- [9] W. Huang, M. R. Stan, K. Skadron, K. Sankaranarayanan, S. Ghosh, and S. Velusamy. Compact thermal modeling for temperature-aware design. In *IEEE/ACM Design Automation Conference*, pages 883–878, San Diego, CA, June 2004.
- [10] C. Isci, A. Buyuktosunoglu, C. Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *MICRO-39*, 2006.
- [11] R. Iyer. Cqos: a framework for enabling qos in shared caches of cmp platforms. In *ICS-18*, 2004.
- [12] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT-13*, 2004.
- [13] J. Li and J. F. Martínez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *International Symposium on High-Performance Computer Architecture*, pages 14–23, Austin, TX, Feb. 2006.
- [14] C. Liu, A. Sivasubramaniam, and M. Kandemir. Organizing the last line of defense before hitting the memory wall for cmps. In *HPCA-10*, 2004.
- [15] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in smt processors. In *International Symposium on Performance Analysis of Systems and Software*, Tucson, AZ, Nov. 2001.
- [16] Micron. *512Mb DDR2 SDRAM Component Data Sheet: MT47H128M4B6-25*, March 2006.
- [17] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO-40*, 2007.
- [18] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *ISCA-35*, 2008.
- [19] K. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queueing memory systems. In *MICRO-39*, 2006.
- [20] K. Nesbit, J. Laudon, and J. Smith. Virtual private caches. In *ISCA-34*, 2007.
- [21] M. Qureshi and Y. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO-39*, 2006.
- [22] N. Rafique, W. Lim, and M. Thottethodi. Architectural support for operating system-driven cmp cache management. In *PACT-15*, 2006.
- [23] J. Renau, B. Fraguera, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005. <http://sesc.sourceforge.net>.
- [24] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA-27*, 2000.
- [25] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA-8*, 2002.
- [26] V. Vardhan, D. G. Sachs, W. Yuan, A. F. Harris, S. V. Adve, D. L. Jones, R. H. Kravets, and K. Nahrstedt. Integrating fine-grained application adaptation with global adaptation for saving energy. In *International Workshop on Power-Aware Real-Time Computing*, Jersey City, NJ, Sept. 2005.
- [27] T. Y. Yeh and G. Reinman. Fast and fair: Data-stream quality of service. In *CASES.05*, 2005.

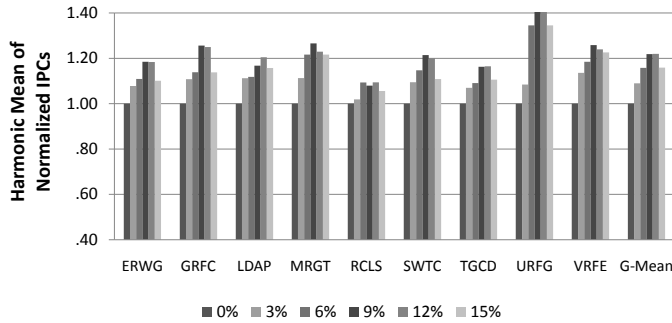


Fig. 10. Sensitivity of the harmonic mean of normalized IPCs to the confidence threshold. Results are normalized to Fair-Share.

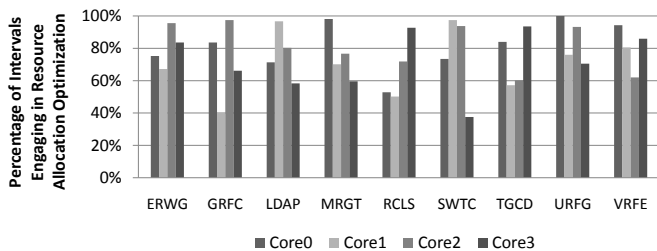


Fig. 11. Percentage of intervals where each application engages in resource allocation optimization (high confidence).

## VI. CONCLUSIONS

We have proposed a resource allocation framework that manages multiple shared CMP resources in a coordinated fashion to enforce higher-level performance objectives, by formulating the global resource allocation as a machine learning (ML) problem. By learning each application’s performance response to different resource distributions, our approach makes it possible to anticipate the system-level performance impact of allocation decisions at runtime. As a result, it becomes possible to make reliable comparisons among different points in a large and dynamically changing allocation space at runtime, allowing us to adapt our allocation decisions as applications undergo phase changes.

Our evaluation using quad-application workloads on simulation models of four-core CMP configurations shows significant, across-the-board gains for our proposed mechanism vs. a number of uncoordinated resource management schemes previously proposed, consistently over four different performance metrics.

We conclude that a coordinated approach to multiple resource management is key to delivering high performance in multiprogrammed workloads, but this is possible only if accompanied by efficient search mechanisms.

## ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for useful feedback. This work was supported in part by NSF awards CNS-0509404, CNS-0720773, and CAREER CCF-0545995, as well as gifts from Intel and Microsoft.