Dynamic Multicore Resource Management: A Machine Learning Approach

A machine learning approach to multicore resource management produces self-optimizing on-chip hardware agents capable of learning, planning, and continuously adapting to changing workload demands. This results in more efficient and flexible management of critical hardware resources at runtime.

José F. Martínez
Cornell University

Engin İpek
University of Rochester

Over the last five years, multicore architectures have emerged as the primary mechanism to reap the benefits of Moore's Law in the billion-transistor era. If CMOS scaling continues to follow Moore's Law, multicore systems could deliver as many as twice the number of cores (and cache space) with every technology generation. Unfortunately, not all multicore resources scale as easily. For example, whereas these systems' off-chip bandwidth requirements will likely grow proportionally to the number of cores, the number of signaling pins in conventional packaging technologies grows by only 10 percent with each technology generation. Also, because of limited projected supply voltage scaling, the average switching power per transistor will likely decrease by only 40 percent per technology node.

Based on these trends and other factors, we predict that effective resource management in future multicore architectures will be most important to delivering cost-effective, high-performance products. This includes problems such as multiresource allocation (for example, cache, power budget, and memory bandwidth allocation), memory-scheduling optimization, cross-core load balancing, and instruction steering in clustered organizations. Alas, how these architectural mechanisms operate is traditionally fully specified by human experts at design time, and as a result they tend to be somewhat rigid and unsophisticated.

Machine learning is the study of computer programs and algorithms that learn about their environment and improve automatically with experience. Researchers and practitioners have successfully applied machine learning techniques and tools in a variety of contexts to tackle resource allocation, scheduling, load balancing, and design space exploration, among other problems (see the "Related work in machine learning" sidebar). In this article, we advocate dynamic multicore resource management based on machine learning techniques. Our position is that the expert's time is best invested in determining which objective function to optimize, which automatic learning technique best suits the problem, and which variables should drive the resulting self-optimizing mechanism at runtime. This approach thus contrasts with today's predominant approach of directly specifying at design time how the hardware should accomplish the desired goal. We illustrate this approach's

Published by the IEEE Computer Society

Related work in machine learning

Researchers and practitioners have successfully applied machine learning techniques and tools to important real-life problems, including not only resource allocation, but also scheduling, load balancing, and design space exploration.

Tesauro et al. explore a decompositional reinforcement learning approach to making autonomic resource-allocation decisions in data centers.[1] Vengerov proposes a reinforcement learning framework to perform adaptive job scheduling in soft real-time systems, in which resource-allocation decisions must be made concurrently with scheduling decisions.[2] Fedorova et al. use reinforcement learning to produce operating system scheduling policies that balance optimal performance, core assignment balance, and response time fairness in a heterogeneous multicore system.[3]

Vengerov et al. apply fuzzy rule bases and reinforcement learning to dynamic load balancing in distributed Web caching.[4] Platt et al. use approximate Bayesian inference to diagnose problems with Web servers.[5] Their proposed framework lets them analyze billions of network logs within a few hours. Ganapathi et al. leverage supervised learning and principal component analysis for query optimization in a large data warehouse.[6] Whiteson and Stone use Q-learning in network routing and job scheduling.[7]

Researchers have also used machine learning techniques in code generation and optimization. Calder et al. use neural networks and decision trees to predict conditional branch directions statically, using static program features.[8] McGovern and Moss apply reinforcement learning and Monte Carlo roll-outs to code scheduling.[9] They find that the learned scheduling policies outperform commercial compilers when generating code for the Alpha 21064 microprocessor. Coons et al. use reinforcement learning and genetic algorithms to learn instruction placement heuristics for distributed microarchitectures.[10] In this case, the derived policies match the performance of highly hand-tuned, specialized heuristics.

Few researchers have applied machine learning to microarchitecture (at runtime or otherwise). Jiménez and Lin propose a perceptron-based dynamic branch predictor, which significantly outperforms prior table-based prediction schemes.[11] Several years earlier, Emer and Gloy used genetic programming for automatic design-time derivation of hardware predictors.[12] İpek et al. use artificial neural networks to cull high-performing configurations from very large design spaces for both single-core and multicore microarchitectures.[13]

References

1. G. Tesauro et al., ''Online Resource Allocation Using Decompositional Reinforcement Learning,'' *Proc. Nat'l Conf. Artificial Intelligence* (AAAI 05), AAAI Press, 2005, pp. 287-299.
2. D. Vengerov, ''Adaptive Utility-based Scheduling in Resource-Constrained Systems,'' *Proc. Australian Joint Conf. Artificial Intelligence* (AI 05), LNCS 3809, Springer, 2005, pp. 477-488.
3. A. Fedorova, D. Vengerov, and D. Doucette, ''Operating System Scheduling on Heterogeneous Core Systems,'' *Proc. Operating System Support for Heterogeneous Multicore Architectures,* (OSHMA), 2007; http://research.sun.com/people/vengerov/OSHMA_2007.pdf.
4. D. Vengerov, H.R. Berenji, and A. Vengerov, ''Adaptive Coordination among Fuzzy Reinforcement Learning Agents Performing Distributed Dynamic Load Balancing,'' *Proc. IEEE Int'l Conf. Fuzzy Systems* (FUZZ-IEEE 02), IEEE Press, 2002, pp. 179-184.
5. J. Platt, E. Kiciman, and D. Maltz, ''Fast Variational Inference for Large-Scale Internet Diagnosis,'' *Proc. Advances in Neural Information Processing Systems Conf.* (NIPS 07), 2007; http://books.nips.cc/nips20.html.
6. A. Ganapathi et al., ''Predicting Multiple Performance Metrics for Queries: Better Decisions Enabled by Machine Learning,'' *Proc. Int'l Conf. Data Eng.* (ICDE 09), IEEE CS Press, 2009, pp. 592-603.
7. S. Whiteson and P. Stone, ''Adaptive Job Routing and Scheduling,'' *Eng. Applications of Artificial Intelligence,* vol. 17, no. 7, 2004, pp. 855-869.
8. B. Calder et al., ''Evidence-based Static Branch Prediction Using Machine Learning,'' *ACM Trans. Programming Languages and Systems* (TOPLAS), vol. 19, no. 1, 1997, pp. 188-222.
9. A. McGovern and E. Moss, ''Scheduling Straight Line Code Using Reinforcement Learning and Rollouts,'' *Proc. Advances in Neural Information Processing Systems Conf.* (NIPS 99), MIT Press, 1999, pp. 903-909.
10. K.E. Coons et al., ''Feature Selection and Policy Optimization for Distributed Instruction Placement Using Reinforcement Learning,'' *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 08), ACM Press, 2008, pp. 32-42.
11. D.A. Jiménez and C. Lin, ''Dynamic Branch Prediction with Perceptrons,'' *Proc. Int'l Symp. High-Performance Computer Architecture* (HPCA 01), IEEE CS Press, 2001, p. 197.
12. J. Emer and N. Gloy, ''A Language for Describing Predictors and Its Application to Automatic Synthesis,'' *Proc. Int'l Symp. Computer Architecture* (ISCA 97), ACM Press, 1997, pp. 304-314.
13. E. İpek et al., ''Efficiently Exploring Architectural Design Spaces via Predictive Modeling,'' *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 06), ACM Press, 2006, pp. 195-206.

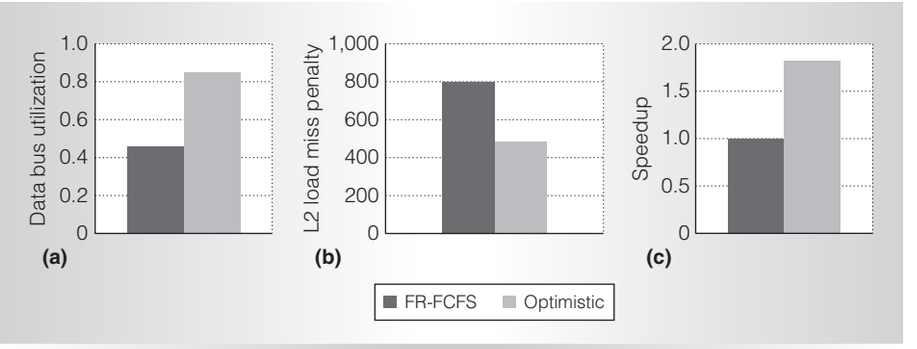

Figure 1. Performance comparison of a first ready, first come, first served (FR-FCFS) versus an optimistic memory controller that can sustain 100 percent of the peak bandwidth (provided enough demand) for the Scalparc application.

potential by describing two recent cases that have shown improvement significantly beyond the state of the art.

Case 1: DRAM scheduling

Providing workloads with adequate off-chip memory bandwidth is often critical to delivering high performance. Unfortunately, DRAM scheduling is a complex problem, requiring a delicate balance between circumventing many access-scheduling constraints (for example, more than 50 timing constraints in Micron's DDR2-800 products), prioritizing requests properly, and adapting to a dynamically changing memory reference stream. When confronted with this challenge, existing memory controllers tend to sustain only a small fraction of the peak bandwidth.[1] The end result is either a significant performance hit, or an overprovisioned (and therefore expensive) memory system.[2-4]

Figure 1 shows a representative example of potential performance loss due to access-scheduling constraints with an existing DRAM controller. Figure 1a shows the sustained DRAM bandwidth of an example parallel application (the Scalable Parallel Classifier [Scalparc][5]) executing on a detailed simulation model of a quadcore-based architecture with both a realistic, contemporary controller design (using the first ready, first come, first served [FR-FCFS] scheduling policy[1]), and an optimistic (and unrealizable) design that can sustain 100 percent of the controller's peak bandwidth, provided enough demand. With an optimistic scheduler, Scalparc uses more than 85 percent of the memory bandwidth effectively. With the realistic scheduler, however, the application's data bus utilization falls to 46 percent. Figure 1b shows this scheduling inefficiency's impact on average L2 load miss penalty. Whereas the optimistic scheduler attains an average load miss penalty of 482 cycles, the realistic scheduler experiences an 801-cycle average penalty. The end result is a 55 percent performance loss compared to the optimistic scheduler (Figure 1c).

In current memory controller designs, a human expert typically chooses a few attributes that will likely be relevant to optimizing a particular performance target (for example, a request's average waiting time), based on prior experience. The expert then devises a fixed, rigid scheduling policy incorporating these attributes, and evaluates it in a simulation model.

The resulting controller usually lacks two important functionalities:

- It cannot anticipate its scheduling decisions' long-term consequences (that is, it cannot do long-term planning).
- It cannot generalize and use the experience obtained through past scheduling decisions to act successfully in new system states (that is, it cannot learn).

DRAM scheduling based on reinforcement learning

Case 1 uses *reinforcement learning* (sometimes called *learning from interaction*).[6] Reinforcement learning studies how autonomous agents in stochastic environments can learn to maximize the cumulative sum of a numerical reward signal received over time through

interaction with their environment.[7,8] Figure 2a depicts the agent-environment interface that defines a reinforcement learning agent's operation. The agent interacts with its environment over a discrete set of time steps. At each step, the agent senses its environment's current state and executes an action. This results in a change in the environment's state (which the agent can sense in the next time step) and produces an immediate reward. The agent's goal is to maximize its long-term cumulative reward by learning an optimal policy that maps states to actions.

Figure 2b shows how a self-optimizing DRAM controller fits within this framework. The agent represents the DRAM command scheduler, while the environment comprises the rest of the system—cores, caches, buses, DRAM banks, and the scheduling queue are all part of the agent's environment. Each time step corresponds to a DRAM clock cycle, during which the agent can observe the system state and execute an action. (Relevant attributes for describing the environment's state are determined via an automated, offline feature selection process at design time.) The actions available to the agent cover the legal DRAM commands that can be issued in the subsequent DRAM cycle (precharge, activate, read, or write). Finally, to motivate the agent to maximize data bus utilization, the controller rewards the agent with +1 each time it issues a command that uses the data bus (that is, a read or write command) and with zero all other times.

Reinforcement learning agents face three major challenges. The first challenge is *temporal credit assignment*. The agent must learn how to assign credit and blame to past actions for each observed immediate reward. In some cases, a seemingly desirable action that yields a high immediate reward drives the system toward undesirable, stagnant states that offer no rewards. At other times, executing an action with no immediate reward is critical to reaching desirable future states. For example, a write command that fills an otherwise unused DRAM data bus cycle might result in several future cycles in which DRAM bandwidth is underused, whereas a precharge command that does not result in any immediate data bus utilization might facilitate better bandwidth utilization in future cycles. Hence, acting optimally requires planning: the agent must anticipate its actions' future consequences and act to maximize its long-term cumulative payoffs. To do this, a reinforcement learning-based memory controller records a *Q-value* indicating the expected long-term cumulative reward value of scheduling a command in a given system state. The scheduler iteratively approximates Q-values based on trial-and-error interaction with the system.

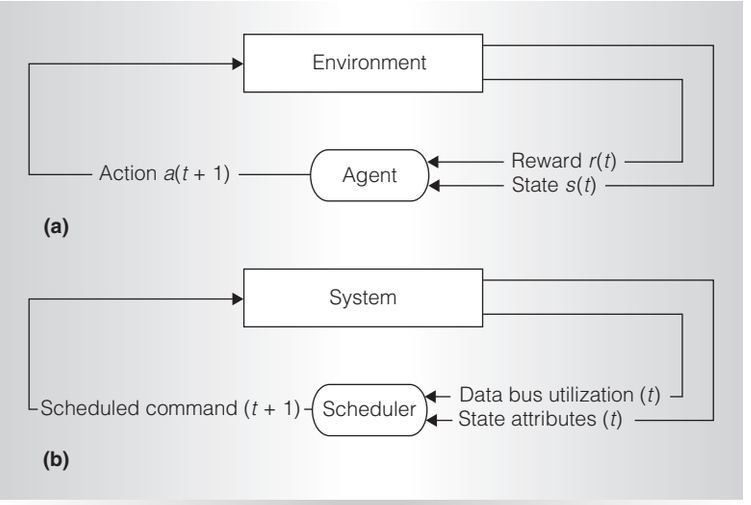

Figure 2. A reinforcement learning framework. An intelligent agent based on reinforcement learning principles reacts to changes in its environment by executing actions that change the environment's state (a). A DRAM scheduler as a reinforcement learning agent observes its environment in time steps corresponding to a DRAM clock cycle and executes actions accordingly (b).

Second, the agent must balance *exploration versus exploitation*. The agent must explore its environment sufficiently (and collect training data) before it can learn a high-performance control policy, but it must also exploit the best policy it has found at any point in time. Too little exploration of the environment can cause the agent to commit to suboptimal policies early on, whereas excessive exploration can result in long periods during which the agent executes suboptimal actions. Furthermore, the agent must continue exploring its environment and improving its policy (life-long learning) to accommodate changes in its environment (due, for example, to phase changes or

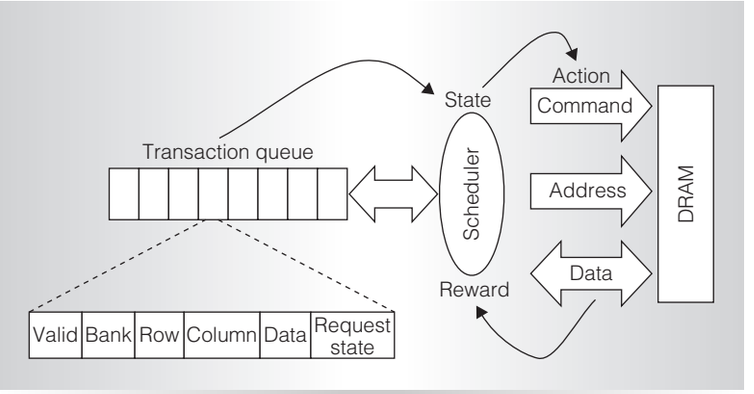

Figure 3. High-level overview of a reinforcement learning-based scheduler. In each DRAM cycle, the scheduler examines valid transaction queue entries, each requiring a *precharge*, *activate*, *read*, or *write* command to be scheduled next.

context switches). To balance exploration and exploitation, a reinforcement learning-based memory controller can implement a simple yet effective exploration mechanism known as ε-greedy action selection. Each DRAM cycle, the scheduler randomizes its scheduling decision by picking a random (but legal) command with a small probability ε.

The third challenge is *generalization*. Because the state spaces' size is exponential in the number of attributes considered, an overwhelming number of possible states can represent the agent's environment. In such cases, the agent is highly unlikely to experience the same state more than once over its lifetime. Consequently, the only way to learn a mapping from states to actions is to generalize and apply the experience gathered over previously encountered (but different) system states to act successfully in new states. To do this, a reinforcement learning-based memory controller records its Q-value estimates in a practical hardware data structure called a cerebellar model arithmetic computer (CMAC). A CMAC model facilitates generalization by effectively quantizing the state space, while also allowing for fast lookups in a realistic hardware setting.

Figure 3 gives an overview of the proposed reinforcement learning-based memory controller. In each DRAM cycle, the scheduler examines valid transaction queue entries, each requiring a *precharge*, *activate*, *read*, or *write* command to be scheduled next.

The scheduler aims to maximize DRAM utilization by choosing the legal command with the highest expected long-term performance benefit under the optimal policy. To do this, the scheduler first derives a state-action pair for each candidate command under the current system state, and uses this information to calculate the corresponding Q-values. The scheduler implements its control policy by scheduling the command with the highest Q-value each DRAM cycle.

Results

İpek et al. compare the proposed reinforcement learning-based memory controller to various other schedulers,[6] including the following:

- Rixner et al.'s FR-FCFS scheduling policy[1,9]
- a conventional in-order memory controller,[1] and
- an optimistic (that is, ideally efficient) scheduler that can sustain 100 percent of the peak DRAM throughput given enough demand.

The experimental setup uses nine memory-intensive parallel workloads, representing a mix of scalable scientific and data mining applications.

Figure 4 shows the results. On a four-core chip with a single-channel DDR2-800 memory subsystem (6.4 GBytes per second peak bandwidth), the reinforcement learning-based memory controller improves the performance of a set of parallel applications by 19 percent on average (up to 33 percent), and DRAM bandwidth utilization by 22 percent on average, over a FR-FCFS scheduler.

Additional experiments show that a traditional memory controller design cannot match the reinforcement learning-based memory controller's performance, even if the best attributes (as determined by the offline feature-selection procedure) are available. In addition, the proposed online reinforcement learning-based scheduling policy's ability to continuously adapt to changes in workload demands significantly outperforms a fixed policy designed offline using reinforcement learning through simulations, and then installed into the chip.

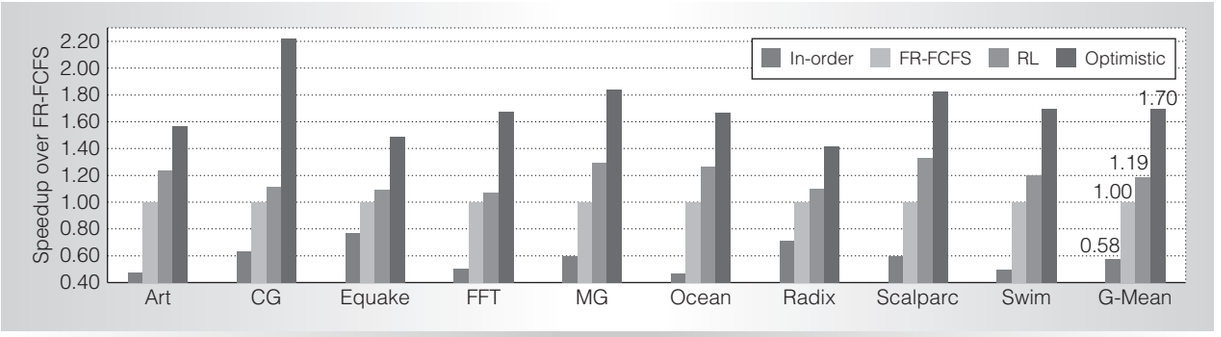

Figure 4. Performance comparison. The proposed reinforcement learning-based mechanism comes closest to the optimistic scheduler.

Case 2: Multiresource allocation

Although resources that are relatively independent of one another can be managed in isolation, many shared resources on multicore systems interact. Unrestricted sharing of microarchitectural resources can lead to destructive interference. Although several proposals that address the management of a single microarchitectural resource exist in the literature, proposals to manage multiple interacting resources on multicore chips at runtime are much scarcer.

Consider, for example, the case of a multicore system in which the on-chip L2 cache space, off-chip bandwidth, and the chip's power budget are shared among applications, and each resource's use is regulated via an independent QoS knob.[10,11] As the amount of one resource allocated to an application changes, the application's demands on the other resources also change. For example, increasing an application's allocated cache space can 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 5 shows an example of performance loss due to uncoordinated resource management in a multicore system incorporating three QoS knobs for regulating the system's shared cache, off-chip bandwidth, and power budget.[16] A four-application, desktop-style multiprogrammed workload is executed on a quadcore chip with an associated DDR2-800 memory subsystem. The figure compares configurations that allocate one or more of the resources in an uncoordinated fashion to a static, fair-share allocation of the resources, as well as an unmanaged sharing scenario in which all applications can access all resources at all times. In this workload, unmanaged resource sharing delivers considerable slowdowns even when compared to a rigid, static resource distribution among the cores (FairShare). Moreover, managing a subset of the resources in an uncoordinated fashion is inferior to static partitioning, indicating that resource interactions render individual adaptive management policies largely ineffective.

Multiresource allocation based on artificial neural networks

Case 2 addresses these limitations by proposing a resource-allocation framework that manages multiple shared multicore resources, in a coordinated fashion, to enforce higher-level performance objectives.[12] At runtime, and without prior knowledge of the workload's characteristics (for example, application profiles), a resource manager monitors each application's execution, and learns a predictive model of their responses to allocation decisions. These predictive models then make it possible to anticipate the system-level performance impact of allocation decisions. The resource manager uses this knowledge to determine what resource distribution to enforce at each point in time.

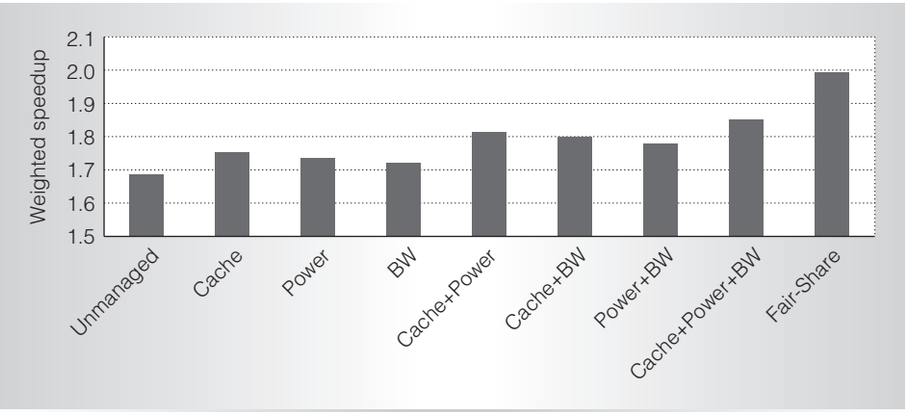

Figure 5. Weighted speedup of a quad-app workload under different resource-management schemes. Despite its simplicity, Fair-Share outperforms all other configurations.

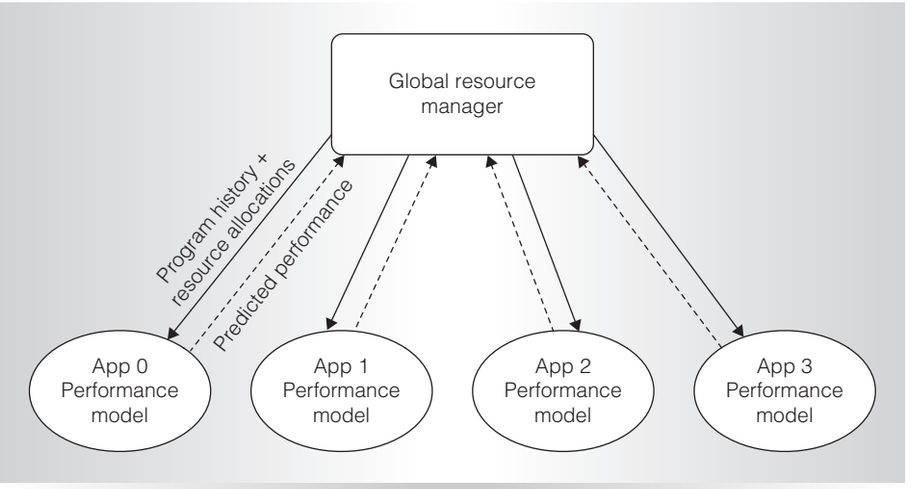

Figure 6. High-level view of the global resource manager's interaction with each application's performance model. The mechanism that we describe uses an ensemble of four artificial neural networks (ANNs) per performance model.

The predictive models are based on artificial neural networks. ANNs are machine learning models that automatically learn to approximate a target function (performance in our case) based on a set of inputs. 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 taking a weighted sum of their inputs based on edge weights, and passing this sum through a nonlinear activation function. 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 nonlinear regression; their representational power is high enough to model multidimensional functions involving complex relationships among variables.

Figure 6 gives an overview of the resource-allocation framework, which comprises dynamically trained, per-application hardware performance models and a global resource manager. The global resource manager

redistributes shared system resources among applications at fixed decision-making intervals, allowing it respond to dynamic changes in workload behavior.

Each application's performance is modeled as a function of its allocated resources and recent behavior, and an ensemble of ANNs is used to learn an approximation of this function. The global resource manager presents input values summarizing past program behavior and indicating allocated resource amounts at each network's input units, and obtains performance predictions from an output unit. The final performance prediction is obtained by averaging the predictions of all ANNs in the ensemble—a technique that often increases accuracy over a single network and lets us assign confidence levels to ANN predictions.

At the end of every interval, the global resource manager searches the space of possible resource allocations by repeatedly querying each application's performance model. The manager then aggregates these predictions into a system-level performance prediction (for example, 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 resource manager installs the configuration estimated to yield the highest aggregate performance. At no point in this iterative process does the global resource manager try candidate resource distributions live. The iterative querying of the predictive models lets the manager try many candidates in a short amount of time. It also helps avoid the negative impact on performance that live trials of subprime candidates would have.

Ideally, we would like to conduct this search exhaustively on all possible combinations of resource allocations. In practice, however, even a quadcore allocation scenario with three resources can result in more than one billion system configurations, rendering exhaustive search intractable. Instead, a heuristic search algorithm can navigate a relatively small portion of the search space, and install the allocation with the highest estimated performance found during this process. Specifically, the proposed solution uses a modified version of stochastic hill climbing that factors in the confidence level of the ANNs and focuses its trials on the most promising subregions of the global allocation space.

As a result of context switches and application phase behavior, workloads can exert drastically different demands on each resource at different points in time. To anticipate and respond to dynamically changing workload demands, the global resource manager keeps the predictive models trained by periodically (in 5 percent of the decision intervals) making a random allocation decision, and using the predicted and observed performance to retrain the models.

Results

Bitirgen et al. evaluate the proposed mechanism's efficacy and versatility using four optimization targets:[12]

- weighted speedups,
- sum of IPCs,
- harmonic mean of normalized IPCs, and
- weighted sum of IPCs.

Here we only show results for weighted speedups; however, additional experiments show that the trends carry over to the other optimization targets. The evaluation uses a simulation model of a quadcore chip with a DDR2-800 memory subsystem, running nine quadcore multiprogrammed workloads, incorporating applications with diverse characteristics from the SPEC 2000[13] and NASA Advanced Supercomputing suites.[14]

The evaluation compares the proposed ANN-based mechanism against 10 resource-sharing schemes:

- one design that leaves all resources unmanaged;
- one design that gives each application its fair share of all resources;
- seven schemes that manage either one, two, or all three resources in an uncoordinated fashion; and
- a port of a previously proposed, hill-climbing-based coordinated SMT resource-allocation framework[15] to this multicore context.

Figure 7 shows the weighted speedup of each quad-workload, normalized to the Fair-Share configuration. In general, most

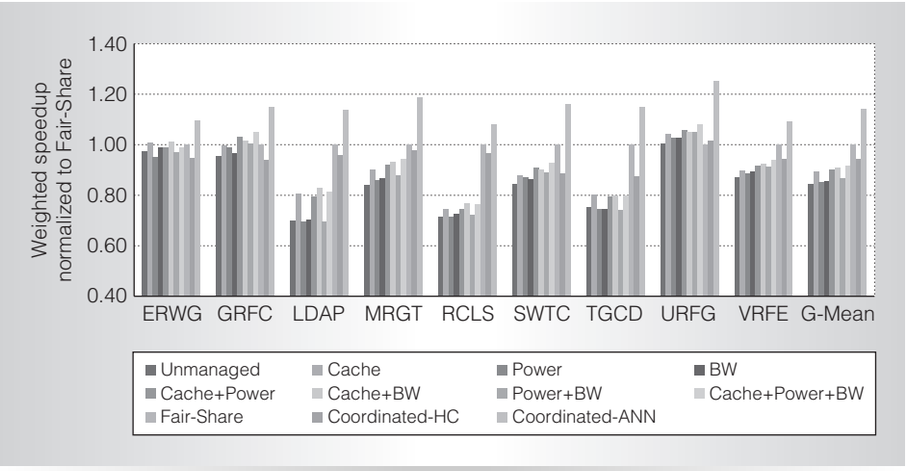

Figure 7. Performance comparison using weighted speedup. Results are normalized to Fair-Share. The proposed Coordinated-ANN scheme consistently outperforms the other configurations.

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 degrades performance with respect to applying only one of those policies. This suggests 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 usually still inferior to Fair-Share.

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 copes with both the exponential size and dynamic nature of the global resource-allocation problem, significantly outperforming both Coordinated-HC and Fair-Share. Specifically, Coordinated-ANN delivers 14 percent average speedup over Fair-Share, and is consistently the best resource-management scheme across all nine workloads.

Although it would be naïve to presume that either of our techniques can provide an optimum solution to the (daunting) problem of multicore resource management, we believe the two cases presented demonstrate that machine learning-based approaches have the potential to effect a significant leap in the performance and flexibility of future microarchitectures. MICRO

References

1. S. Rixner et al., ''Memory Access Scheduling,'' *Proc. Int'l Symp. Computer Architecture* (ISCA 00), IEEE CS Press, 2000, pp. 128-138.
2. T. Dunigan et al., ''Early Evaluation of the Cray X1,'' *Proc. Supercomputing* (SC 03), IEEE CS Press,2003, p. 18.
3. P. Kongetira et al., ''Niagara: A 32-Way Multithreaded SPARC Processor,'' *IEEE Micro,* vol. 25, no. 2, 2005, pp. 21-29.
4. J.D. McCalpin, ''Sustainable Memory Bandwidth in Current High Performance Computers,'' http://www.cs.virginia.edu/~mccalpin/papers/bandwidth/bandwidth.html.
5. M. Joshi et al., ''ScalParC: A New Scalable and Efficient Parallel Classification Algorithm for Mining Large Datasets,'' *Proc. Int'l Parallel Processing Symp.* (IPPS 98), IEEE CS Press, 1998, pp. 573.
6. E. İpek et al., ''Self-Optimizing Memory Controllers: A Reinforcement Learning Approach,'' *Proc. Int'l Symp. Computer Architecture* (ISCA 08), IEEE CS Press, 2008, pp. 39-50.
7. D. Bertsekas, *Neuro Dynamic Programming,* Athena Scientific, 1996.
8. R. Sutton and A. Barto, *Reinforcement Learning,* MIT Press, 1998.

9. S. Rixner, ''Memory Controller Optimizations for Web Servers,'' *Proc. Int'l Symp. Microarchitecture* (MICRO 04), IEEE CS Press, 2004, pp. 355-366.
10. C. Isci et al., ''An Analysis of Efficient Multi-core Global Power Management Policies: Maximizing Performance for a Given Power Budget,'' *Proc. Int'l Symp. Microarchitecture* (MICRO 06), IEEE CS Press, 2006, pp. 347-358.
11. M. Qureshi and Y. Patt, ''Utility-based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches,'' *Proc. Int'l Symp. Microarchitecture* (MICRO 06), IEEE CS Press, 2006, pp. 423-432.
12. R. Bitirgen, E. İpek, and J.F. Martínez, ''Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach,'' *Proc. Int'l Symp. Microarchitecture* (MICRO 08), IEEE CS Press, 2008, pp. 318-329.
13. J.L. Henning, ''SPEC CPU2000: Measuring CPU Performance in the New Millennium,'' *Computer,* vol. 33, no. 7, 2000, pp. 28-35.
14. D.H. Bailey et al., *NAS Parallel Benchmarks,* tech. report RNR-94-007, NASA Ames Research Center, 1994.
15. S. Choi and D. Yeung, ''Learning-based SMT Processor Resource Distribution via Hill-Climbing,'' *Proc. Int'l Symp. Computer Architecture* (ISCA 06), IEEE CS Press, 2006, pp. 239-251.
16. K.J. Nesbit et. al., "Fair Queueing Memory Systems," Proc. Int'l Symp. Microarchitecture (MICRO 06), IEEE CS Press, 2006, pp. 208–222.

José F. Martínez is an associate professor of electrical and computer engineering at Cornell University. His research interests include reconfigurable and self-optimizing architectures. Martínez has a PhD in computer science from the University of Illinois, Urbana-Champaign. He is a member of the ACM and the IEEE.

Engin İpek is an assistant professor of computer science and electrical and computer engineering at the University of Rochester. His research interests include self-optimizing architectures and low-power memory subsystems. İpek has a PhD in electrical and computer engineering from Cornell University. He is a member of the ACM and the IEEE.

Direct questions and comments to José F. Martínez at Cornell Univ., 336 Rhodes Hall, Ithaca, NY, 14850; martinez@cornell.edu.