Self-Optimizing Memory Controllers: A Reinforcement Learning Approach

Engin İpek[1,2] Onur Mutlu[2] José F. Martínez[1] Rich Caruana[1]

[1]Cornell University, Ithaca, NY 14850 USA

[2] Microsoft Research, Redmond, WA 98052 USA

ABSTRACT

Efficiently utilizing off-chip DRAM bandwidth is a critical issue in designing cost-effective, high-performance chip multiprocessors (CMPs). Conventional memory controllers deliver relatively low performance in part because they often employ fixed, rigid access scheduling policies designed for average-case application behavior. As a result, they cannot learn and optimize the long-term performance impact of their scheduling decisions, and cannot adapt their scheduling policies to dynamic workload behavior.

We propose a new, self-optimizing memory controller design that operates using the principles of reinforcement learning (RL) to overcome these limitations. Our RL-based memory controller observes the system state and estimates the long-term performance impact of each action it can take. In this way, the controller *learns* to optimize its scheduling policy on the fly to maximize long-term performance. Our results show that an RL-based memory controller improves the performance of a set of parallel applications run on a 4-core CMP by 19% on average (up to 33%), and it improves DRAM bandwidth utilization by 22% compared to a state-of-the-art controller.

1. INTRODUCTION

Chip Multiprocessors (CMPs) are attractive alternatives to monolithic cores due to their power, performance, and complexity advantages. Current industry projections indicate that scaling CMPs to higher numbers of cores will be 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, CMPs could deliver as much as twice the number of cores and the available on-chip cache space every 18 months. Unfortunately, the benefits of Moore's Law are unavailable to conventional packaging technologies, and consequently, both the speed and the number of signaling pins grow at a much slower rate (pin count increases by roughly 10% each year) [18]. As a result, off-chip bandwidth may soon present a serious impediment to CMP scalability [40].

Yet providing adequate peak bandwidth is only part of the problem. In practice, delivering a large fraction of this theoretical peak to real-life workloads demands a memory controller that can effectively utilize the off-chip interface. DRAM scheduling is a complex problem, requiring a delicate balance between circumventing access scheduling constraints, 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 [27, 37]. The end result is either a significant performance hit, or an over-provisioned (and therefore expensive) memory system [13, 22, 23].

Figure 1 shows an example of potential performance loss due to access scheduling constraints with an existing DRAM controller. The leftmost graph shows the sustained DRAM bandwidth of an example parallel application (SCALPARC [20]) with both a realistic, contemporary controller design (using the FR-FCFS scheduling policy [37, 49]), and an optimistic (and unrealizable) design that is able to sustain 100% of the controller's peak bandwidth, provided enough demand.[1] (A detailed description of the architecture employed can be found in Section 4.) With an optimistic scheduler, SCALPARC utilizes over 85% of the memory bandwidth effectively. When running with the realistic scheduler, however, the application's data bus utilization falls to 46%. The middle plot in Figure 1 shows the impact of this scheduling inefficiency on average L2 load miss penalty: while 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% performance loss compared to the optimistic scheduler (rightmost plot).

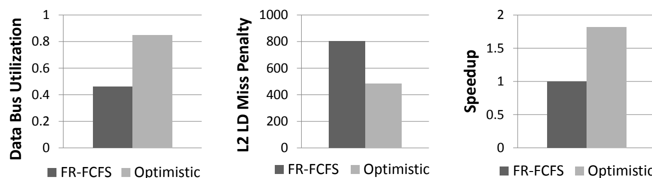

Figure 1: Bandwidth, latency, and execution time comparison of a state-of-the-art memory controller (employing the FR-FCFS scheduling policy) vs. an optimistic memory controller that can sustain 100% of the peak bandwidth (provided enough demand) for the SCALPARC application.

A potential limiting factor to higher performance in current memory controller designs is that they are largely ad hoc. Typically, a human expert chooses a few attributes that are likely to be relevant to optimizing a particular performance target (e.g., the average waiting time of a request in FR-FCFS) based on prior experience. With these in mind, the expert devises a (fixed) scheduling policy incorporating these attributes, and evaluates such a policy in a simulation model. The resulting controller usually lacks two important functionalities: First, it cannot anticipate the long-term consequences of its scheduling decisions (i.e., it cannot do *long-term planning*). Second, it cannot generalize and use the experience obtained through scheduling decisions made in the past to act successfully in new system states (i.e., it cannot *learn*). As we will show, this rigidity and lack of adaptivity can manifest itself as severe performance degradation in many applications.

This paper proposes the use of machine learning technology in designing a self-optimizing, adaptive memory controller capable of planning, learning, and continuously adapting to changing workload demands. We formulate memory access scheduling as a *reinforcement learning* problem [42]. Reinforcement learning (RL) is a field of machine learning that studies how autonomous agents situated in stochastic environments can learn optimal control policies through interaction with their environment. RL provides a general framework for high-performance, self-optimizing controller design.

Key idea: *We propose to design the memory controller as an RL agent whose goal is to learn automatically an optimal memory scheduling policy via interaction with the rest of the system.*

An RL-based memory controller takes as input parts of the system state and considers the long-term performance impact of each action it can take. The controller's job is to (1) associate system states and actions with long-term reward values, (2) take the action (i.e., schedule the command) that is estimated to pro-

[1]We accomplish this by lifting all timing constraints except DRAM data bus conflicts (i.e., CAS-to-CAS delays), and then servicing requests on a first-come-first-serve basis in our simulation environment. CAS (t_{CL}) and Write (t_{WL}) latencies, as well as bus contention, are included in the memory access latency.

1063-6897/08 $25.00 © 2008 IEEE
DOI 10.1109/ISCA.2008.21

IEEE computer society

vide the highest long-term reward (i.e., performance) value at a given system state, and (3) continuously update long-term reward values associated with state-action pairs, based on feedback from the system, in order to adapt to changes in workloads and memory reference streams. In contrast to conventional memory controllers, an RL-based memory controller:

- Anticipates the long-term consequences of its scheduling decisions, and continuously optimizes its scheduling policy based on this anticipation.
- Utilizes experience learned in previous system states to make good scheduling decisions in new, previously unobserved states.
- Adapts to dynamically changing workload demands and memory reference streams.

An RL-based design approach allows the hardware designer to focus on *what* performance target the controller should accomplish and *what* system variables *might* be useful to ultimately derive a good scheduling policy, rather than devising a fixed policy that describes *exactly how* the controller should accomplish that target. This not only eliminates much of the human design effort involved in traditional controller design, but also (as our evaluation shows) yields higher-performing controllers.

We evaluate our self-optimizing memory controller using a variety of parallel applications from the SPEC OpenMP [4], NAS OpenMP [5], Nu-MineBench [33], and SPLASH-2 [46] benchmark suites. On a 4-core CMP with a single-channel DDR2-800 memory subsystem (6.4GB/s peak bandwidth in our setup), the RL-based memory controller improves performance by 19% on average (up to 33%) over a state-of-the-art FR-FCFS scheduler. This effectively cuts in half the performance gap between the single-channel configuration and a more expensive dual-channel DDR2-800 subsystem with twice the peak bandwidth. When applied to the dual-channel subsystem, the RL-based scheduler delivers an additional 14% performance improvement on average. Overall, our results show that self-optimizing memory controllers can help utilize the available memory bandwidth in a CMP more efficiently.

2. BACKGROUND AND MOTIVATION

We briefly review the operation of the memory controller in modern DRAM systems to motivate the need for intelligent DRAM schedulers, and provide background on reinforcement learning as applicable to DRAM scheduling. Detailed descriptions are beyond the scope of this paper. Interested readers can find more detailed descriptions in [11, 12, 37] on DRAM systems and in [6, 28, 42] on reinforcement learning.

2.1 Memory Controllers: Why are They Difficult to Optimize?

Modern DRAM systems consist of dual in-line memory modules (DIMMs), which are composed of multiple DRAM chips put together to obtain a wide data interface. Each DRAM chip is organized as multiple independent memory banks. Each bank is a two-dimensional array organized as *rows* × *columns*.

Only a single row can be accessed in each bank at any given time. Each bank contains a row buffer that stores the row that can be accessed. To access a location in a DRAM bank, the memory controller must first make sure that the row is in the row buffer. An *activate* command brings the row whose address is indicated by the address bus from the memory array into the row buffer. Once the row is in the row buffer, the controller can issue *read* or *write* commands to access a column whose address is indicated by the address bus. Each *read* or *write* command transfers multiple columns of data, specified by a programmable *burst length* parameter. To access a different row, the controller must first issue a *precharge* command so that the data in the row buffer is written back to the memory array. After the precharge, the controller can issue an *activate* command to open the new row it needs to access.

The memory controller accepts cache misses and write-back requests from the processor(s) and buffers them in a *memory transaction queue*. The controller's function is to satisfy such requests by issuing appropriate DRAM commands while preserving the integrity of the DRAM chips. To do so, it tracks the state of each DRAM bank (including the row buffer), each DRAM bus, and each memory request. The memory controller's task is complicated due to two major reasons.

First, the controller needs to obey all DRAM timing constraints to provide correct functionality. DRAM chips have a large number of timing constraints that specify when a command can be legally issued. There are two kinds of timing constraints: local (per-bank) and global (across banks due to shared resources between banks). An example local constraint is the *activate to read/write delay*, t_{RCD}, which specifies the minimum amount of time the controller needs to wait to issue a read/write command to a bank after issuing an activate command to that bank. An example global constraint is the *write to read delay*, t_{WTR}, which specifies the minimum amount of time that needs to pass to issue a read command to any bank after issuing a write command to any bank. State-of-the-art DDR2 SDRAM chips often have a large number of timing constraints that must be obeyed when scheduling commands (e.g., over 50 timing constraints in [26]).

Second, the controller must intelligently prioritize DRAM commands from different memory requests to optimize system performance. Different orderings and interleavings of DRAM commands result in different levels of DRAM throughput and latency [37]. Finding a good schedule is not an easy task as scheduling decisions have long-term consequences: not only can issuing a DRAM command prevent servicing other requests in subsequent DRAM cycles (due to timing constraints), but also the interleaving of requests from different cores (and therefore the future contents of the memory reference stream) is heavily influenced by which core's blocking requests get serviced next (possibly unblocking the instruction window or the fetch unit of the requesting core, allowing it to generate new memory requests). Moreover, the ultimate benefit provided by a scheduling decision is heavily influenced by the future behavior of the processors, which certainly is not under the scheduler's control.

Current memory controllers use relatively simple policies to schedule DRAM accesses. Rixner et al. [37] show that none of the fixed policies studied provide the best performance for all workloads and under all circumstances. However, the FR-FCFS (first-ready first-come first-serve) policy [36, 37] provides the best average performance. Among all ready commands, FR-FCFS prioritizes (1) column (CAS) commands (i.e., read or write commands) over row (RAS) commands (i.e., activate and precharge) in order to maximize the number of accesses to open rows, and (2) older commands over younger commands. Hence, FR-FCFS gives the highest priority to the oldest ready column command in the transaction queue. The goal of the FR-FCFS policy is to maximize DRAM throughput and minimize average request latency. However, FR-FCFS does not consider the long-term performance impact of either (1) prioritizing a column command over a row command, or (2) prioritizing an older command over a younger command.

2.2 Reinforcement Learning and Its Applicability to DRAM Scheduling

Machine learning is the study of computer programs and algorithms that learn about their environment and improve automatically with experience. Within this larger framework, *Reinforcement Learning* (RL), sometimes called "learning from interaction," studies how autonomous agents situated in stochastic environments can learn to maximize the cumulative sum of a numerical reward signal received over their lifetimes through interaction with their environment [6, 42].

Figure 2(a) depicts the agent-environment interface that defines the operation of RL agents. The agent interacts with its environment over a discrete set of time steps. At each step, the agent senses the current *state* of its environment, and executes an *action*. This results in a change in the state of the environment (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 2(b) 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 can include the number of reads, writes, or load misses in the transaction queue, the criticality of each request (e.g., based on the requester's relative in-

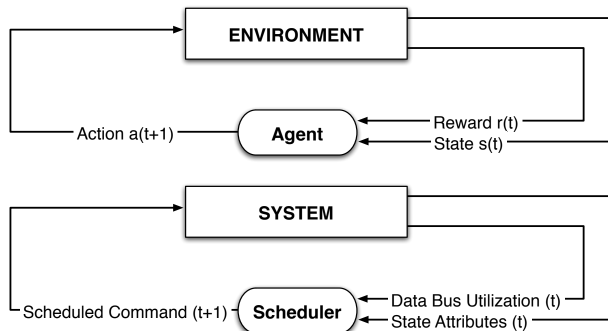

Figure 2: **(a) Intelligent agent based on reinforcement learning principles; (b) DRAM scheduler as an RL-agent**

struction sequence number), whether a given request would hit in the row buffer if serviced next, or the total number of reads and writes pending for each row and bank. The actions available to the agent cover the legal DRAM commands that can be issued in the following DRAM cycle (precharge, activate, read, or write). Note that the set of available actions can change from one time step to the next depending on the system state; in particular, the agent may have no actions available in states where timing constraints prevent issuing any commands. Finally, rewards can be designed in several different ways depending on optimization goals; for instance, to have the agent learn how to maximize data bus utilization, the agent can be given a reward of one each time it issues a command that utilizes the data bus (i.e., a *read* or *write* command), and a reward of zero at all other times.

Three major challenges facing an RL agent are:

Temporal credit assignment. The agent needs to learn how to assign credit and blame to past actions for each observed immediate reward. In some cases, a seemingly desirable action that yields high immediate reward may drive the system towards undesirable, stagnant states that offer no rewards; at other times, executing an action with no immediate reward may be 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 where DRAM bandwidth is underutilized, 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 the future consequences of its actions and act accordingly to maximize its long-term cumulative payoffs.

Exploration vs. exploitation. The agent needs to explore its environment sufficiently (and collect training data) before it can learn a high-performance control policy, but it also needs to 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 to explore its environment. Furthermore, the agent needs to continue exploring its environment and improving its policy (life-long learning) to accommodate changes in its environment (e.g., due to phase changes or context switches).

Generalization. Because the size of the state space is exponential in the number of attributes considered, the agent's environment may be represented by an overwhelming number of possible states. In such cases, it is exceedingly improbable for the agent 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.

In the rest of this section, we describe briefly the formal mechanisms behind reinforcement learning, and we illustrate its potential to successfully characterize and optimize memory access scheduling.

2.2.1 Markov Decision Processes

In reinforcement learning, the agent's environment is described by an abstraction called a Markov Decision Process (MDP). Formally, an MDP consists of a set S of system states, a set A of actions, a transition probability distribution $T = P(s_{t+1} = s'|s_t = s,\ a_t = a)$ that assigns the probability of each possible next state s' for each state s and action a, and a reward function $R = E[r_{t+1}|s_t = s, a_t = a, s_{t+1} = s']$ that defines the expected value of the next reward received when action a is executed in state s, followed by a transition to state s'. Together, the transition probability distribution and reward function completely describe the dynamics of the environment. Figure 3 shows the transition graph of an example non-deterministic MDP. The environment described by the MDP has two states S0 and S1, and there are two actions A0 and A1 available to the agent in both states. Executing action A0 in state S0 causes the agent to remain in state S0 and receive a reward of 3 with a probability of 0.1, and causes a transition to state S1 and a reward of 15 with a probability of 0.9. The goal of the agent is to learn an effective policy π that maps states to actions to maximize its cumulative long-term rewards. Note that the next state is non-deterministic from the viewpoint of the agent because the next state is not solely a function of the action but rather a function of both the action and the environment (i.e. the system's behavior). For example, in DRAM scheduling, the next state of the system depends partly on the command scheduled by the DRAM scheduler, and partly on the system's behavior in that cycle, which is not under the scheduler's control (but may be part of a function that can be learned).

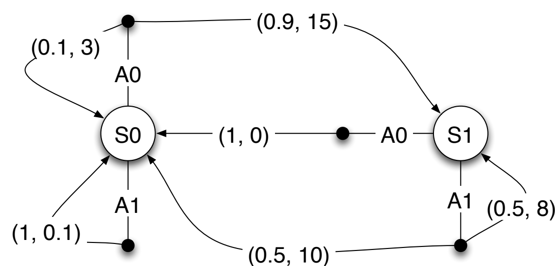

Figure 3: **Example of a non-deterministic Markov Decision Process (MDP).**

Cumulative Rewards: Memory access scheduling is most naturally formulated as an infinite-horizon (i.e., continuing) task, where the agent (scheduler) constantly navigates the MDP by scheduling DRAM commands, accumulating rewards along the MDP edges as specified by the MDP's reward function. One issue that comes up in an infinite-horizon MDP is the convergence of the cumulative rewards. Since the agent's lifetime is infinite (at least for all practical purposes), all policies would lead to infinite reward over the agent's lifetime, leading to an ill-defined objective function for optimization. Instead, in infinite-horizon problems, optimizing for a discounted cumulative reward function at each time step t is more appropriate:

$$E[\sum_{i=0}^{\infty} \gamma^t \cdot r_{t+i}] \qquad (1)$$

Here, r_t is the immediate reward observed at time step t, and γ is a discount rate parameter between zero and one ($0 \leq \gamma < 1$). Discounting causes the sum of future rewards to act as a geometric series, and guarantees convergence [6]. Intuitively, γ can be seen as a knob that controls how important future rewards are compared to immediate rewards. Larger values of γ lead to agents with greater foresight and better planning capabilities, but require longer training times. As γ is reduced, the agent becomes increasingly myopic; in the extreme case where γ is set to zero, the agent learns only to choose actions that maximize the immediate rewards it receives from its environment.

2.2.2 Rewarding Control Actions: Q-values

An elegant way of addressing the temporal credit assignment problem is through the notion of Q-values [28, 42], which form the basis of many reinforcement learning algorithms. In an infinite-horizon non-deterministic MDP, the Q-value of a state-action pair (s,a) under policy π (denoted by $Q_{\pi(s,a)}$) represents the expected value of the cumulative discounted future reward that is obtained when action a is executed in state s, and policy π is followed thereafter. In the context of DRAM scheduling, a Q-value describes the long-term value of scheduling a command in a given system state. If a Q-value for the optimal policy π*

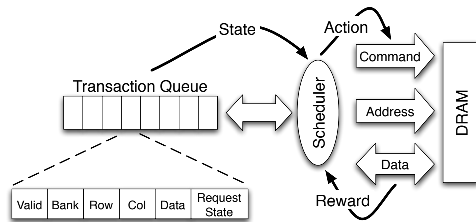

Figure 4: **High-level overview of an RL-based scheduler.**

is learned and stored for each state-action pair, then, at a given state the agent can simply choose the action with the largest Q-value in order to maximize its cumulative long-term rewards. In other words, determining the Q-values associated with the optimal policy π^* is equivalent to deriving π^* itself. In subsequent sections, we explain how an RL-based DRAM scheduler can learn to estimate the Q-values (i.e., long-term cumulative reward values) associated with its scheduling decisions.

3. RL-BASED DRAM SCHEDULERS: STRUCTURE, OPERATION, AND IMPLEMENTATION

We first provide an overview of the structure of our RL-based controller along with the rewards, actions, and system state that are available to it (Section 3.1). We then describe how the controller operates, makes scheduling decisions, and learns better scheduling policies (Section 3.2), along with its implementation (Section 3.3). Our discussion explains the operation of our RL-based command scheduler when optimizing DRAM data bus utilization. Intuitively, optimizing data bus utilization makes sense because of the high degree of correlation between effective data bus utilization and high system performance The performance and data bus utilization figures in our evaluation (Section 5) support this choice.

3.1 Overview of the RL-Based Memory Controller

Figure 4 shows an overview of the proposed RL-based memory controller. Each DRAM cycle, the scheduler (agent) examines valid transaction queue entries, each one of which requires a *Precharge*, *Activate*, *Read*, or *Write* command to be scheduled next. The scheduler's goal is 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 subsequently 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.

3.1.1 Formulation of the RL-Based DRAM Scheduler

As we explained in Section 2.2.1, DRAM command scheduling is naturally formulated as an infinite-horizon discounted MDP, where the scheduler issues DRAM commands to transition from one system state to the next, collecting rewards along the MDP edges based on the data bus utilization it achieves. To complete this formulation, we need to specify an appropriate reward structure, as well as a set of states and actions that the scheduler can use to reason about its scheduling decisions.

Reward structure. To learn to maximize long-term data bus utilization, the scheduler is given an immediate reward of 1 each time it schedules a *Read* or *Write* command (which are the only commands that lead to data bus utilization), and a reward of 0 at all other times.[2] Note that providing an immediate reward of 0 for *Precharge* and *Activate* commands does not mean that the scheduler will not issue those commands when there is a legal command with an immediate reward value of 1. The scheduler learns to maximize *long-term rewards* (i.e., the long-term value of data bus utilization). Hence, if scheduling a command that provides an immediate reward of 0 ultimately brings about the highest cumulative reward by enabling better data bus utilization, the scheduler will learn to pick that command (as opposed to another one with an immediate reward value of 1).

[2]This includes DRAM cycles in which *Precharge* and *Activate* commands are issued, as well as cycles during which timing constraints prevent the scheduler from issuing any commands.

States. For each candidate command, there are six attributes[3] of the system state that the scheduler considers, all of which are locally available in the controller's transaction queue. These six attributes are:

1. Number of reads (load/store misses) in the transaction queue.[4]
2. Number of writes (writebacks) in the transaction queue.
3. Number of reads in the transaction queue that are load misses.
4. If the command is related to a load miss by core C in the transaction queue, the load's order in C's dynamic instruction stream relative to other loads by C with requests in the transaction queue. (This is determined by comparing sequence numbers, which are assigned dynamically at rename time, similarly to Alpha 21264's inum [21], and piggybacked in the request to the controller.)[5]
5. Number of writes in the transaction queue waiting for the row referenced by the command under consideration.[6]
6. Number of load misses in the transaction queue waiting for the row referenced by the command under consideration which have the oldest sequence number among all load misses in the transaction queue from their respective cores.

The first two attributes are intended to help the RL controller learn how to optimize the balance of reads and writes in the transaction queue. For example, the controller might learn to reduce write buffer stalls by balancing the rate at which writes and reads are serviced. The third attribute can allow the controller to detect states that lead to low levels of request concurrency in the transaction queue (i.e., states where many cores are blocked due to a high number of load misses), and to avoid such situations in advance by prioritizing load misses (possibly at the expense of other inefficiencies due to timing constrains). The fourth attribute can facilitate learning how to prioritize among load misses. The fifth attribute might help the controller learn how to amortize write-to-read delays (i.e., t_{WTR}) by satisfying writes in bursts. The sixth attribute is intended to approximate the number of critical (i.e., core-blocking) requests that are likely to block the instruction windows of their cores; opening a row with many critical requests can improve performance by unblocking multiple cores and allowing them to make forward progress.

Note that, with an integrated memory controller (which is the industry trend as seen in IBM POWER5 [17, 39], Sun Niagara [22], AMD Athlon/Opteron [1], and Intel Nehalem [3]), it is relatively easy to communicate sequence numbers and whether a request is due to a load or store miss from the processor to the controller. If the memory controller is off chip, extra pins would be needed to accomplish this communication. Our design assumes an integrated on-chip memory controller.

Actions. There are up to six different actions available to the scheduler from each state. These are: (1) issue a precharge command, (2) issue an activate command, (3) issue a write command, (4) issue a read command to satisfy a load miss, (5) issue a read command to satisfy a store miss, and (6) issue a NOP. The NOP action is used to update the scheduler's internal Q-value estimates in cases where timing or resource constraints prevent the scheduler from issuing any legal commands, so that the scheduler can learn to associate actions leading to such states with low long-term cumulative rewards. Distinguishing between reads that satisfy load and store misses allows the scheduler to learn to prioritize load misses when it helps performance.

3.2 Our RL-based DRAM Command Scheduling Algorithm

Algorithm 1 shows our RL-based scheduling algorithm. In its simplest form, the scheduler operates on a table that records Q-

[3]We selected these six attributes from a set of 226 candidates through an automated *feature selection* process (see Section 3.4). It is possible to use more state attributes at the expense of increased hardware complexity in the controller.

[4]For scheduling purposes, here we treat instruction misses as load misses.

[5]Sequence numbers are generated using a counter that is incremented by one each time a new instruction is renamed. No special actions are taken upon counter overflow, and the counter need not be checkpointed across branches: from the controller's point of view, such inaccuracies in the sequence numbers constitute a small amount of "noise" which the RL scheduler can readily accommodate. We assign the minimum sequence number to all instruction misses.

[6]Naturally, precharge commands reference the row currently in the row buffer, *not* the row that will eventually be activated to satisfy the corresponding request.

values for all possible state-action pairs. Initially, all table entries are optimistically reset[7] to the highest possible Q-value ($\frac{1}{1-\gamma}$). Each DRAM cycle, the scheduler observes the transaction queue to find all DRAM commands that can be legally issued without violating timing constraints (line 6). Occasionally, the scheduler issues a random legal command to explore its environment and to adapt its policy to dynamic changes (lines 9-10). We describe this random exploration in detail in Section 3.2.2. Normally, the scheduler picks the command with the highest Q-value for scheduling (lines 11-12). To do so, for each candidate command, the scheduler estimates the corresponding Q-value by accessing its internal Q-value table. The scheduler selects the command with the highest Q-value for scheduling in the next cycle (line 12).

Note that command selection takes a full DRAM clock cycle (often equivalent to multiple CPU cycles–ten in our experimental setup). The selected command is issued at the clock edge (i.e., at the beginning of the next cycle). After issuing the command selected in the previous cycle (line 6), the scheduler records the reward of the action it took (line 7). In a given cycle, the Q-value associated with the previous cycle's state-action pair is updated as shown in lines 15 and 18. The update of the Q-value is affected by the immediate reward, along with the Q-values of the previous and current state-action pairs. As this is critical to learning a high-performance policy, we next describe the Q-value update in detail.

3.2.1 Learning Q-values: Solving the Temporal Credit Assignment Problem

To learn the Q-values, the scheduler continuously updates its estimates based on the state transitions and rewards it experiences as it issues commands. Specifically, after taking action a_{prev} in state s_{prev}, the scheduler observes an immediate reward r, transitions to state $s_{current}$, and executes action $a_{current}$ (i.e., schedules the selected command *cmd*). The Q-value associated with executing a_{prev} in state s_{prev} is then updated according to an update rule known as the SARSA update [42]) (line 18):

$$Q(s_{prev}, a_{prev}) \leftarrow (1-\alpha)Q(s_{prev}, a_{prev}) + \alpha[r + \gamma Q(s_{current}, a_{current})] \quad (2)$$

Here, α is a learning rate parameter that facilitates convergence to the true Q-values in the presence of noisy or stochastic rewards and state transitions [42]. Recall that γ is a discount rate parameter for future rewards as explained in Section 2.2.1. In our implementation of an RL-based memory controller, we empirically observe that $\alpha = 0.1$ and $\gamma = 0.95$ work quite well. The quantity $r + \gamma Q(s_{current}, a_{current})$ intuitively represents the sum of the immediate reward obtained by executing action a_{prev} in state s_{prev}, plus the discounted sum of all future rewards when the current policy is followed from that point on. Hence, the update can be interpreted as taking a sample estimate of the true Q-value $Q^{\pi}(s_{prev}, a_{prev}) = r + \gamma Q^{\pi}(s_{current}, a_{current})$ of the current policy π, and then moving the estimated Q-value towards this sample by a small step size α. For non-deterministic MDPs with stationary reward and transition probability distributions, SARSA is guaranteed to find the optimal scheduling policy with probability 1 in the limit where each table entry is visited infinitely often [6].

3.2.2 Balancing Exploration vs. Exploitation

The table-based RL algorithm we have discussed in Section 3.2 depends critically on the assumption that the scheduler has a non-zero probability of visiting each table entry; if the scheduler never chooses to schedule certain commands in a given state, it would be unable to learn the associated Q-values. Even if the scheduler has already learned an optimal policy, changes in the dynamic behavior of the environment (e.g., due to context switches or phase changes) could render the already-learned policy obsolete. To avoid these problems, the scheduler must continuously explore its environment throughout its lifetime, while at the same time utilizing the best policy it has found at each point in time.

To strike a balance between exploration and exploitation, we implement a simple yet effective exploration mechanism known as ϵ-greedy action selection [42]. Each DRAM cycle, the scheduler randomizes its scheduling decision by picking a random (but legal) command with a small probability ϵ (line 7 in Algorithm 1). In our implementation, we set ϵ to 0.05. This guarantees that the scheduler continues to try different actions in each state, while following the best policy it has found the majority of the time.

3.2.3 Generalization: Enabling Across-State Learning while Reducing the Number of Q-values

A practical problem with RL-based controllers is that the number of Q-values that need to be maintained grows exponentially with the number of attributes used in state representation. Consequently, a naive implementation that keeps a table entry for every possible Q-value is infeasible beyond a very small number of attributes.[8] Not only do the storage requirements of the Q-values grow to impractical sizes with the number of states, but also building hardware that implements the derived policy while simultaneously meeting latency and power requirements becomes much more difficult.

Coarse-grain vs. fine-grain quantization: One way of overcoming this limitation is to quantize the state space into a small number of cells. By aggregating all states within each cell and representing them by a single Q-value (Figure 5(a) and (b)), dramatic reductions in storage requirements can be accomplished. However, this quantization approach requires a compromise between resolution and generalization that is hard to optimize statically. On the one hand, a fine-grain quantization (Figure 5(a)) may result in too many cells and make it hard to generalize from scheduling decisions executed in similar, past system states. On the other hand, a coarse-grain quantization (Figure 5(b)) with large cells may not offer enough resolution to accurately represent the Q-values over the state space.

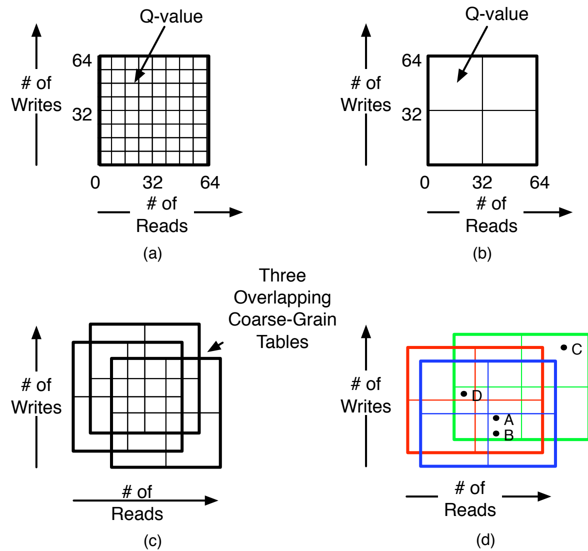

Figure 5: **(a) Fine-grain quantization, (b) Coarse-grain quantization, (c) CMAC using three overlapping coarse-grain tables (for adaptive resolution), (d) CMAC example. Each table entry represents one Q value. Each table index is a function of the state attributes. This example uses only two state attributes to simplify the figures.**

CMAC: Balancing Generalization and Resolution A computationally efficient way of balancing generalization and resolution at runtime is a learning model known as *CMAC* [41]. When integrated with an RL-based memory controller, the CMAC replaces the scheduler's internal table holding all possible Q-values (Section 3.2). A CMAC consists of multiple coarse-grain Q-value tables, each of which is shifted by a random (but constant) amount with respect to one another, as shown in Fig-

[7] Optimistic initialization encourages high levels of exploration in the early stages of the execution [42].

[8] The number of state-action pairs is $O(NumberOfStates * NumberOfActions) = O(TransactionQueueEntries^{NumberOfAttributes} * NumberOfActions)$. This would lead to over 1 billion Q-values in our case.

Algorithm 1 RL-Based DRAM Scheduling Algorithm

```
1: Initialize all Q-values to 1/(1−γ)
2: Initialize C ← get_legal_command_set()  //Get all legal commands from the transaction queue
3: Initialize cmd ← select_random_command(C)  //Select a random legal command from the set of all legal commands
4: Initialize Q_prev ← get_Q-value_from_table(SystemState, cmd)  //Get the Q-value associated with the current state and command

5: for all DRAM cycles
6:    issue_command(cmd)  //Issue command selected in the previous cycle
7:    r ← observe_reward()  //Collect immediate reward associated with the issued command
8:    C ← get_legal_command_set()  //Get all legal commands from the transaction queue
9:    if (rand() < ε) then
10:      cmd ← select_random_command(C)  //With ε probability, select a random legal command to explore (Section 3.2.2)
11:   else
12:      cmd ← select_command_with_max_Q-value(C)  //Otherwise, select the legal command with the highest Q value
13:   end if
14:   Q_selected ← get_Q-value_from_table(SystemState, cmd)  //Get Q-value associated with the current state and selected command
15:   update_Q-value(Q_prev, r, Q_selected)  //SARSA update (Section 3.2.1)
16:   Q_prev ← Q_selected  //Record the selected Q-value for use in the Q-value update of next cycle
17: end for

   Function update_Q-value(Q_prev,r,Q_selected)
18: Q_prev ← (1 − α)Q_prev + α(r + γQ_selected)  //Update Q-value of previous state action pair in the table
```

ure 5(c). Shifting is done by simply adding to each index a fixed number, set randomly at design time (as shown in Figure 6(c)).

For a given state, a Q-value estimate is obtained by accessing all of the coarse-grain tables with the appropriate index, reading the value in that location, and summing the values read from all tables. Each table entry is updated (trained) based on the SARSA update explained in Section 3.2.1. Because the tables overlap, nearby points in the state space tend to share the same entry in most tables, whereas distant points will occupy different entries in different tables. As a result, a SARSA update to a given point in the state space will heavily influence the Q-value estimates of nearby points, whereas distant points will not be influenced as heavily; the magnitude of this change is commensurate with the number of tables in which the two points in space share the same entry.

Figure 5(d) shows an example that illustrates this point. A two-dimensional state space is quantized with three different coarse-grain tables, each one of which consists of four entries.[9] Points A and B are close-by in the state space, and consequently, they share the same entry (i.e., index) in all three tables. Hence, the model will provide the same Q-value estimate for both of these points, and an update to A will be reflected fully in the Q-value estimate for B. On the other hand, point D is further away from A; in particular, A and D share only a single entry in one of the three tables. Consequently, the model will be able to represent these two points with different Q-values, but an update to A will nevertheless influence D's Q-value through the value stored in the single entry they share. In this way, a CMAC model provides the ability to generalize between relatively distant points, while also retaining the capacity to distinguish points that are relatively close-by.

Hashing: Another important technique used in CMAC-learning of Q-values is hashing. The index of each CMAC table is calculated by passing the "state attributes" through a hash function (as we will show in Figure 6(b)). Through hashing, dramatic savings in storage requirements are possible because the storage requirements are no longer exponential in the number of state attributes considered. Similarly to destructive aliasing in branch or value prediction, hashing can result in collisions and thus lead to interference across Q-values. In practice this turns out not to be a major problem because (a) while the full MDP is far larger than the storage capacity of the hash-based CMAC tables, the scheduler tends to navigate a small, localized region of the state space at any point in time (which reduces the likelihood of interference), and (b) RL algorithms readily accommodate stochastic rewards, and such interference can be viewed as a small source of noise in the scheduler's Q-value estimates.

3.2.4 Ensuring Correct Operation

Since the scheduler's decisions are restricted to picking among the set of legal commands each cycle, it is not possible for the scheduler to compromise the integrity of the DRAM chips and corrupt data by violating any timing or resource constraints. Nevertheless, care must be taken to ensure that the system is guaranteed to make forward progress regardless of the scheduler's decisions.

Specifically, an RL-based memory controller implements three provisions to ensure forward progress at all times. First, as explained in Section 3.1.1, the scheduler is not permitted to select NOPs when other legal commands are available. Second, the scheduler is only allowed to activate rows due to pending requests in the transaction queue (i.e., the scheduler cannot choose to activate an arbitrary row with no pending requests). Consequently, any new row that is brought into a row buffer is guaranteed to have at least one pending request. Finally, the scheduler is not allowed to precharge a newly activated row until it issues a read or write command to it. As a result, the scheduler cannot prevent progress by constantly precharging and activating rows without issuing any intervening *Read* or *Write* commands.

A second correctness issue for existing out-of-order command schedulers as well as our RL-based scheduler is the possibility of starvation. Consider, for instance, the case of a core waiting on a spinlock. Although unlikely in practice, it is nevertheless possible for this core to generate a continuous stream of DRAM accesses while spinning on the lock variable. Consider now a situation in which the lock owner also has to satisfy a DRAM access before it can release the lock. If the scheduler indefinitely keeps satisfying the continuous stream of requests originating from the spin loop, the owner cannot perform the lock release and starvation ensues. Like existing schedulers, an RL-based command scheduler solves this problem by implementing a timeout policy: any request that has been pending for a fixed (but large - in our case 10,000) number of cycles is completed in its entirety before other commands can be considered for scheduling.

Finally, as DRAM refresh is an operation that is essential to correctness, we do not allow the RL controller to dictate the refresh schedule. Instead, at the end of a refresh interval, the RL scheduler is disabled, the appropriate rows are refreshed, and then control is returned to the RL scheduler.

3.3 Hardware Implementation

Figure 6(a) shows an overview of the scheduler's Q-value estimation pipeline. While DRAM commands are issued every DRAM cycle, the on-chip memory controller's five-stage pipeline itself is clocked at the processor core frequency. We describe below the operation of a pipeline that can calculate and compare two Q-values every processor clock cycle. The width of the pipeline can be tuned for different types of systems depending on the DRAM and processor clock speeds, and the size of the transaction queue.

In the first pipe stage, the scheduler gets the set of legal commands along with the current state attributes from the transaction queue,[10] and generates a state-action pair for each can-

[9] Note that the CMAC tables are actually implemented as SRAM arrays. In Figure 5, we show them as two-dimensional tables to illustrate the concept of adaptive resolution more clearly. Figure 6 shows the implementation of the CMAC SRAM arrays and how they are indexed.

[10] The state attributes are obtained in the previous DRAM cycle and stored in the transaction queue.

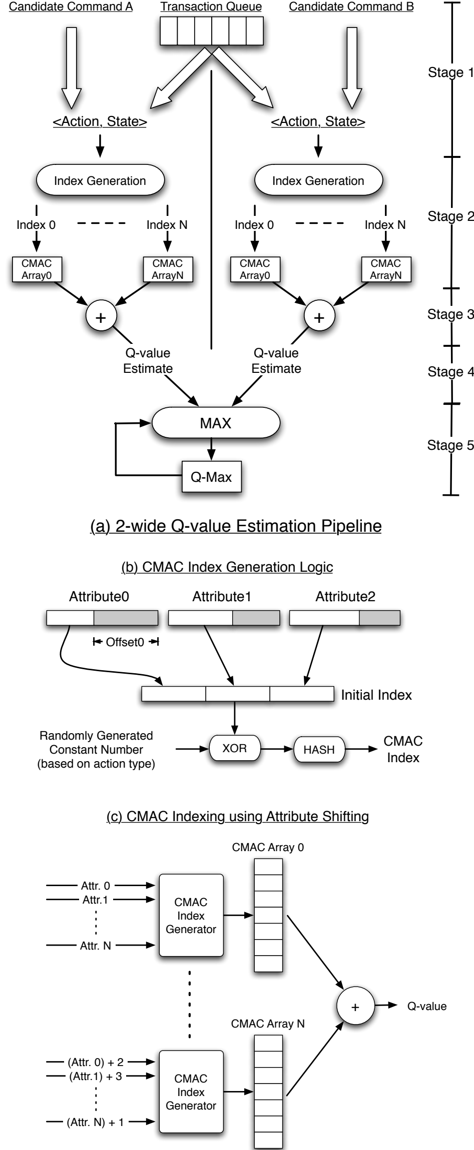

Figure 6: **(a) RL-based scheduler's Q-value estimation pipeline, (b) CMAC array index generator, (c) CMAC array indexing**

didate command (using the states and actions described in Section 3.1.1). This information is used to access the scheduler's CMAC arrays in the second pipeline stage. To index the CMAC arrays, the high order bits of each state attribute are concatenated. For each state attribute, the granularity of quantization determines how many low-order bits are skipped. This initial index is then XOR-ed with a randomly-generated constant number depending on the type of action to avoid generalization across Q-values corresponding to different types of commands. The final CMAC index is formed by hashing this index to reduce storage requirements. The hardware used for generating the index of a CMAC array is shown in Figure 6(b). Each CMAC array is indexed differently using shifted attributes to obtain adaptive resolution, as shown in Figure 6(c).

In the third and fourth pipe stages, the Q-value of each candidate command is computed by adding the estimates from all of the corresponding CMAC arrays. In a two-wide pipe, there are two sets of CMAC arrays, one for each Q-value. In the final stage, the maximum Q-value found so far is compared to the two Q-values leaving the pipeline, and the maximum Q-value found so far is updated along with the corresponding candidate command.

For a 4GHz chip attached to a DDR2-800 system (typical of today's high-end desktop and workstation systems), the scheduler's pipe can be clocked ten times per DRAM cycle. With a throughput of two Q-values per cycle, and four initial cycles to fill the pipe, an RL-based scheduler can consider up to 12 commands for scheduling every DRAM clock. For the applications we studied, we have empirically observed that the number of legal commands in a DRAM cycle very rarely exceeds this value in a 64-entry transaction queue, resulting in negligible performance loss (less than 0.3% on average) compared to a scheduler that can consider up to 64 commands. Nevertheless, the number of considered candidates can be increased by increasing the pipe width if needed.

Hardware Overhead: The hardware overhead of the RL-based scheduler consists primarily of three portions: 1) logic required to compute state attributes; 2) logic required to estimate and update Q-values; and 3) SRAM arrays required to store Q-values. The logic for computing state attributes consists of counters that are updated every DRAM cycle. Q-values are stored in 16-bit fixed point format. We have already described the logic for estimating them. Q-value updates are made using fixed point arithmetic, according to the SARSA update described in section 3.2.1. To perform this update, we use a single, pipelined 16-bit fixed-point multiplier. Note that the predictor update is not on the critical path of making scheduling decisions, and can be performed in multiple DRAM cycles (i.e., many tens of processor cycles) without significantly affecting the accuracy of Q-values. In our experiments, we maintain a total of 8192 distinct Q-values across all CMAC arrays in each of the two command pipes, resulting in 32kB of on-chip storage.[11]

3.4 Design Effort

RL-based memory controllers are attractive from a design complexity standpoint because they alleviate the hardware designer's burden of scheduling policy design. The hardware designer can treat the RL-based controller as a *black box*, which, given a performance target, relevant state attributes, and a set of actions to choose from, automatically learns to map system states to actions to optimize a long-term performance target. Thus, the architect's job at design time is centered on (1) selecting a reward structure that reflects the desired optimization target, and (2) selecting a set of relevant state attributes and actions to be used by the controller.

In practice, there may be hundreds of available attributes that might be used as inputs to the controller. Typically, however, training RL-based controllers on a carefully selected subset of the available attributes yields better-performing schedulers. *Feature selection* is the automated process of finding the best subset of state attributes to use as inputs to the controller. There has been substantial prior work on how to perform (near-) optimal feature selection [9]. We conduct a thorough review of system attributes and identify 226 candidates, based on our intuition of whether they could possibly be useful to guide memory scheduling. Then we use feature selection to systematically trim them down to six attributes. At a high level, the *forward stepwise feature selection* algorithm we use can be seen as a greedy search (via simulation experiments) through the attribute space to find the best set of state attributes.

Specifically, given N candidate attributes, forward selection initially evaluates N RL-based schedulers in simulation, each using a different one of these N candidates. Once the best single-attribute scheduler is determined, forward selection evaluates $N-1$ two-attribute schedulers, each of which uses the best attribute found earlier, plus one of the remaining $N-1$ candidate attributes. This process is repeated N times by incorporating at each turn the attribute that yields the best-performing RL-based scheduler when combined with the attributes selected so far. When this greedy search terminates (i.e, after N iterations), the best set of attributes found among all sets that were evaluated are selected as inputs to be used by the RL-based controller at runtime.

[11] Each CMAC array is an SRAM array of 256 Q-values with 1 read port and 1 write port. There are a total of 32 CMAC arrays in each of the two ways of the two-wide command pipe.

Processor Parameters	
Frequency	4.0 GHz
Number of cores	4
Number of SMT Contexts	2 per core
Fetch/issue/commit width	4/4/4
Int/FP/Ld/St/Br Units	2/2/2/2/2
Int/FP Multipliers	1/1
Int/FP issue queue size	32/32 entries
ROB (reorder buffer) entries	96
Int/FP registers	96 / 96
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	512 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

Table 1: **Core Parameters.**

Shared L2 Cache Subsystem	
Shared L2 Cache	4MB, 64B block, 8-way
L2 MSHR entries	64
L2 round-trip latency	32 cycles (uncontended)
Write buffer	64 entries
DDR2-800 SDRAM Subsystem [26]	
Transaction Queue	64 entries
Peak Data Rate	6.4GB/s
DRAM bus frequency	400 MHz
Number of Channels	1, 2, 4
DIMM Configuration	Single rank
Number of Chips	4 DRAM chips per rank
Number of Banks	4 per DRAM chip
Row Buffer Size	2KB
Address Mapping	Page Interleaving
Row Policy	Open Page
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 2: **Shared L2 and DRAM subsystem parameters.**

4. EXPERIMENTAL SETUP

We evaluate the performance of our RL-based memory controller by comparing it to three different schedulers: (1) Rixner et al.'s FR-FCFS scheduling policy, which was shown to be the best-performing policy on average [36, 37], (2) a conventional in-order memory controller [37], and (3) an optimistic (i.e., ideally efficient) scheduler that can sustain 100% of the peak DRAM throughput if there is enough demand (this optimistic scheduler was described in Section 1). We simulate nine memory-intensive parallel applications with eight threads, running on a CMP with four two-way simultaneously multithreaded (SMT) cores, 4MB of L2 cache, and a DDR2-800 memory system. Table 1 shows the microarchitectural parameters of the processor cores we model, using a heavily modified version of the SESC simulation environment [35]. Our CMP model is loosely based on Intel's Nehalem processor [3], which integrates four 2-way SMT cores with an on-chip memory controller. Table 2 shows the parameters of the shared L2 cache and the SDRAM memory subsystem modeled after Micron's DDR2-800 SDRAM [26].

Our parallel workloads represent a mix of scalable scientific applications (three applications from the SPLASH-2 suite [46], three applications from the SPEC OpenMP suite [4], and two parallel NAS benchmarks [5]), and a parallelized data mining application SCALPARC from Nu-MineBench [33].[12] The input sets we use are listed in Table 3. All applications are compiled using the gcc and Fortran compilers at the O3 optimization level. Each application is simulated to completion. We select state attributes by conducting feature selection experiments (Section 3.4) on the four fastest applications to simulate (MG, FFT, RADIX, SWIM). We observe no qualitative differences between final performance results obtained on these applications and the remaining ones that were not used in feature selection.

[12]These are the only applications from the SPEC OpenMP, NAS OpenMP, and Nu-MineBench suites that our simulation infrastructure currently supports. In the case of SPLASH-2, we chose these applications as they are the only ones that result in an average transaction queue occupancy of 1 in the baseline, providing an opportunity for DRAM scheduling.

Benchmark	Description	Problem size
Data Mining		
SCALPARC	Decision Tree	125k pts., 32 attributes
NAS OpenMP		
MG	Multigrid Solver	Class A
CG	Conjugate Gradient	Class A
SPEC OpenMP		
SWIM-OMP	Shallow water model	MinneSpec-Large
EQUAKE-OMP	Earthquake model	MinneSpec-Large
ART-OMP	Self-Organizing Map	MinneSpec-Large
Splash-2		
OCEAN	Ocean movements	514×514 ocean
FFT	Fast Fourier transform	1M points
RADIX	Integer radix sort	2M integers

Table 3: **Simulated applications and their input sizes.**

5. EVALUATION

Figure 7 compares the performance of in-order, FR-FCFS, and RL-based scheduling policies along with the performance of an *optimistic* (and unrealizable) controller, which serves as an upper bound (described in Section 1). The data is normalized to the performance of FR-FCFS. On average, the RL-based memory controller (RL) improves performance by 19%, thereby achieving 27% of the performance improvement that can be provided by making DRAM scheduling ideally efficient. RL's performance improvement is greater than 5% for all applications, with a maximum of 33% for SCALPARC and a minimum of 7% for FFT. Note that the conventional in-order controller significantly underperforms the baseline FR-FCFS controller, in line with previous research results [37].

Figure 8 provides insight into the performance improvement obtained using our RL-based controller by showing the DRAM data bus utilization (i.e., sustained DRAM bandwidth). The average bus utilization of the optimistic controller is 80%, which means that aplications can utilize 80% of the 6.4GB/s peak DRAM bandwidth if the memory controller can provide it. In-order and FR-FCFS controllers can sustain only 26% and 46% of the peak DRAM bandwidth.[13] On average, RL improves DRAM data bus utilization from 46% to 56%. All applications experience increases in data bus utilization, which is strongly correlated with the performance improvement we showed in Figure 7. The two applications that see the highest speedups with RL, SCALPARC (33%) and MG (30%) also see the largest increases in bus utilization. Note that there is still a large gap between RL's and the optimistic controller's bus utilization. This is due to two major reasons: (1) RL requires time to adapt its policy to changes in workload behavior whereas the optimistic controller does not suffer from such a scheduling inefficiency, (2) RL cannot overcome timing constraints and bank conflicts whereas the optimistic controller can *always* sustain the peak bandwidth if the application demands it. Hence, the optimistic controller is an unachievable upper bound, but provides a good measure of the application's DRAM bandwidth demand.

5.1 Performance Analysis

An RL-based memory controller enjoys several advantages over a conventional memory controller design. In this section, we identify important sources of RL's performance potential, and conduct experiments to provide further insight into the bandwidth and performance improvements reported earlier.

5.1.1 Queue Occupancy and Load Miss Penalty

Figure 9 shows the average transaction queue occupancy and the average L2 load miss penalty under FR-FCFS and RL-based command schedulers. Recall that the RL-based scheduler's performance target is to optimize data bus utilization in the long

[13]Even though it might seem low, the data bus utilization in our FR-FCFS baseline is consistent with what is reported in previous research [37] and by DRAM manufacturers [27]. Rixner [37] reported an average utilization of approximately 35% for a different set of applications. Due to the difficulty of DRAM scheduling, Micron considers an average data bus utilization of 45% to be "high" [27].

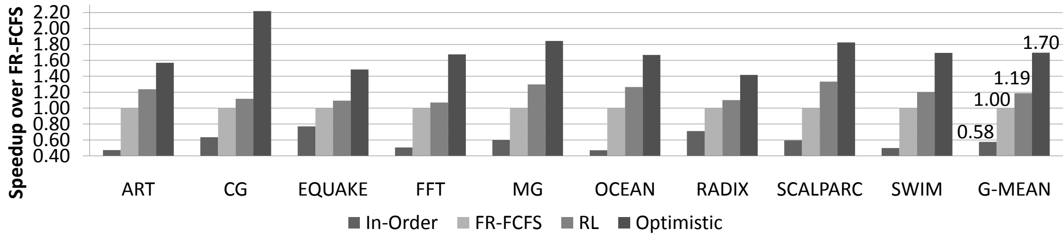

Figure 7: Performance comparison of in-order, FR-FCFS, RL-based, and optimistic memory controllers

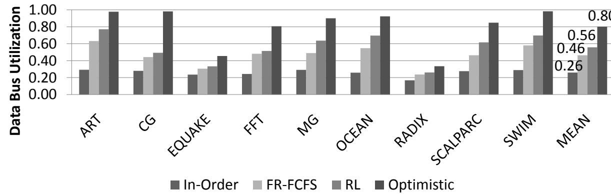

Figure 8: DRAM data bus utilization (sustained DRAM bandwidth) of in-order, FR-FCFS, RL-based, and optimistic memory controllers

term. To improve data bus utilization, a scheduler needs to exploit row buffer locality and maximize bank-level parallelism, both of which can be better exploited with higher transaction queue occupancy. The plot at the top of Figure 9 shows that the RL-based scheduler is indeed able to keep the transaction queue much busier than the baseline FR-FCFS system; on average, the RL-based scheduler has 28 requests active in its transaction queue, while FR-FCFS has only 10. A higher transaction queue occupancy is possible if cores supply requests to the memory controller at a faster rate. A core can issue its memory requests faster if its instruction window is blocked less due to L2 load misses, i.e., if the average L2 load miss penalty is lower. As shown in the plot at the bottom of Figure 9, the RL-based command scheduler achieves an average L2 load miss penalty of 562 cycles, while FR-FCFS sustains an 824-cycle penalty. Hence, by learning to optimize data bus utilization, the RL-based scheduler reduces average L2 load miss latency and improves execution time.

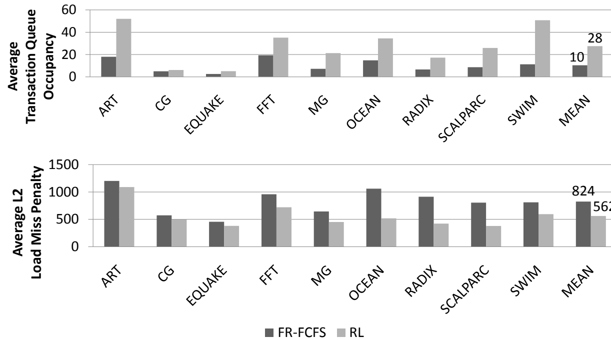

Figure 9: Average transaction queue occupancy (top) and average L2 load miss penalty (bottom) under FR-FCFS and RL-based schedulers

5.1.2 Performance Impact of Selected Attributes

Some of the state attributes used in our RL-based memory controller (Section 3.1.1) require additional information not utilized by the baseline FR-FCFS scheduler (whether a request is a read or a write, whether a read is due to a load or a store miss, and the relative order among load misses based on their sequence numbers). To understand how much of the speedups can be credited to this additional information alone, we create a family of scheduling policies, each of which extends FR-FCFS by incorporating the extra state information used by the RL controller into its scheduling decisions.

Specifically, we define three new preference relationships prioritizing (1) reads over writes, (2) load misses over store misses, and (3) more critical load misses over less critical ones, based on sequence numbers (Section 3.1.1). We combine these with the two preference relationships already utilized in FR-FCFS (prioritizing CAS commands over RAS commands, and older requests over younger requests) to derive a family of scheduling policies, each of which corresponds to a different order in which preference relationships are applied to prioritize legal commands. We exhaustively evaluate all possible policies within this family, and report the performance of the best policy we find.[14] (We have also evaluted an augmented version of this policy that issues writes in bursts to amortize write-to-read delays (t_{WTR}), but found its performance to be inferior.) Figure 10 shows the results.

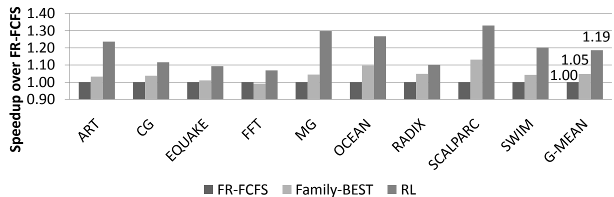

Figure 10: Performance comparison between RL-based scheduler and an extended FR-FCFS based scheduler using the same state information

Incorporating the additional state information into FR-FCFS results in modest performance improvements, with an average of 5%. Compared to FR-FCFS and its derivatives, an RL-based scheduler enjoys two key advantages. First, an RL-based memory controller encoding its policy in a CMAC array exhibits much higher *representational power* than a conventional controller: the number of possible policies that can be expressed by a CMAC array are typically much larger than the number of policies that can be expressed via preference relationships. Hence, many policies that can be formulated by our RL-based controller are fundamentally obscured to conventional controllers based on less expressive representations. Second, online learning allows an RL-based controller to adapt its scheduling policy to changes in workload demands (e.g., due to phase changes) and memory reference streams at runtime, while FR-FCFS and its derivatives are fixed scheduling policies that cannot change their mapping of system states to actions. Consequently, an RL-based memory controller achieves considerably higher speedups than FR-FCFS derivatives utilizing the same state information.

5.1.3 Performance Impact of Runtime Adaptation

To gauge the importance of runtime adaptation to our RL-based memory controller, we make performance comparisons against a static policy found by an offline version of our CMAC-based RL algorithm. We train this offline RL algorithm on training data collected from all of our benchmarks. Once training is over, we install the learned policy in our simulation environment by hard-coding the final Q-value estimates in the controller's CMAC array, and evaluate the performance of this hard-coded policy on all of our benchmarks. Figure 11 shows the results.

On average, offline RL provides a speedup of 8% over FR-FCFS, and significantly underperforms its online, adaptive version. This is due to two main reasons. First, online RL can accommodate changes in workload demands by adapting its control policy at runtime, whereas offline RL calculates a fixed scheduling policy that cannot adequately cater to the different needs

[14]The best policy we found among FR-FCFS derivatives prioritizes (1) CAS commands over RAS commands, (2) reads over writes (new to FR-FCFS), (3) load misses over store misses (new), (4) more critical load misses over less critical ones, based on sequence numbers (new), and (5) older requests over younger ones.

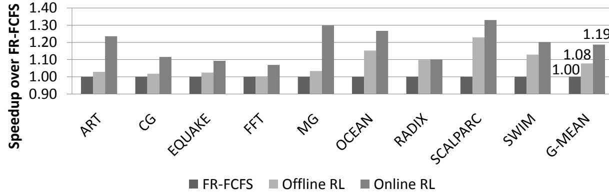

Figure 11: Performance comparison against a static, RL-based scheduler trained offline.

of all phases within a program. Second, offline RL algorithms can have difficulty finding high-performance control policies for non-stationary environments where the state transition probability distribution characterizing the dynamics of the environment (Section 2) changes over time. In contrast, due to its ability to sample training data from the actual distribution at runtime, online RL more readily accommodates non-stationarity. Overall, these results suggest that online learning is essential to our RL-based memory controller.

5.1.4 Sensitivity to Controller Parameters

The plot on the left side of Figure 12 shows speedups with respect to FR-FCFS for different values of the discount rate γ. Recall that γ is effectively a knob that controls how important future rewards are compared to immediate rewards. In the extreme case where γ is set to zero, the scheduler learns to maximize only its immediate rewards, and suffers a significant performance loss compared to schedulers with better planning capabilities (achieved by setting γ to larger values). At the opposite extreme, setting γ to one causes convergence problems in Q-values: the resulting scheduler fails to converge to a high-performance control policy, and achieves relatively poor performance compared to more balanced schedulers. The optimum value of γ is achieved at 0.95 for this set of benchmarks, at which point the scheduler converges to a high-performance control policy.

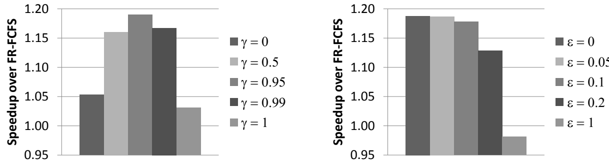

Figure 12: Performance sensitivity to discount rate γ (left) and exploration parameter ϵ (right)

The plot on the right side of Figure 12 shows speedups with respect to FR-FCFS for different values of the exploration parameter ϵ. For this set of benchmarks, lower values of ϵ work better, and setting ϵ to zero does not affect performance adversely. Note, however, that maintaining exploration could still be necessary to accommodate drastic changes in the scheduler's environment, e.g., due to context switches. As expected, large amounts of random exploration lead to severe performance degradation.

5.2 Scaling to Multiple Memory Controllers

A common way of increasing peak DRAM bandwidth for larger-scale CMP platforms is to integrate multiple memory controllers on chip [22], where each controller serves a different set of physical addresses through an independent DRAM channel. In our case, this necessitates the integration of multiple autonomous schedulers (i.e., agents), each of which optimizes its own long-term cumulative performance target. One question that arises in such a distributed RL setting with multiple schedulers is whether Q-values can successfully converge in the presence of potential interactions between the schedulers (mainly by unblocking cores and thereby affecting the level of parallelism/occupancy in each other's transaction queues), and whether there is a need for explicit coordination among the controllers.

To answer these questions, we experiment with two larger-scale CMP configurations: an 8-core CMP with two memory controllers, and a 16-core CMP with four memory controllers. Each memory controller is attached to an independent DDR2 channel with 6.4GB/s of peak bandwidth, for aggregate bandwidth values of 12.8GB/s and 25.6GB/s in the case of 8- and 16-core systems, respectively. Aside from the number of cores and independent DDR2 channels, we also scale the number of L2 MSHR entries. We keep all other microarchitectural parameters unchanged. Figure 13 shows the performance of all applications on our 8- and 16-core baselines, normalized to the 4-core baseline performance.

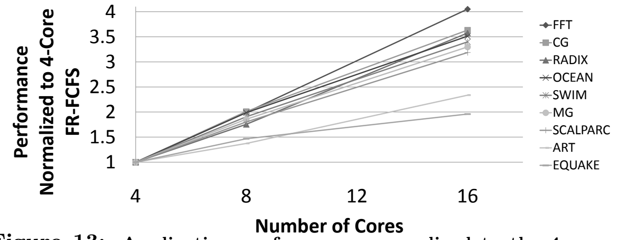

Figure 13: Application performance normalized to the 4-core FR-FCFS baseline.

In all three systems, we observe that Q-values converge successfully. To check the need for explicit coordination among schedulers, we repeat the feature selection process (Section 3.4) with additional attributes summarizing the states of other memory controllers to each controller. We find that such state attributes are not selected as they do not provide significant performance improvements. These results lead us to conclude that any potential interactions among the controllers are second-order effects, and that explicit coordination mechanisms among RL-based DRAM schedulers are not needed. Figure 14 shows the results.

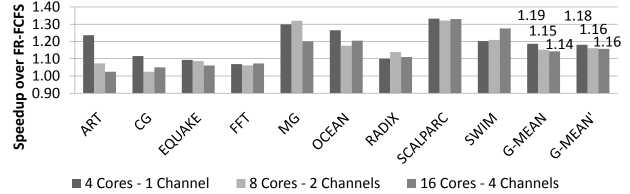

Figure 14: RL's speedup over FR-FCFS in 4-, 8-, and 16-core systems with 1, 2, and 4 memory controllers, respectively.

On average, RL-based memory controllers improve performance by 15% and 14% over FR-FCFS in systems with two and four independent channels. Across applications, there is no clear trend indicating a deterioration of performance potential as the number of controllers is increased from one to four. ART is an outlier, where speedups over FR-FCFS drop from 24% to 2% as the system is scaled from four to sixteen cores. We found that this application's data set size is too small to scale beyond four cores effectively, leading to heavy load imbalance in systems with 8 and 16 cores, and artificially reducing the performance potential for DRAM scheduling by restricting the number of in-flight requests in the transaction queue. In all three CMP configurations, the average speedups over all remaining applications are within 2% of one another (denoted as G-MEAN' in Figure 14), suggesting that our RL-based memory controller can be readily incorporated in a system with multiple memory controllers (i.e., a multi-agent configuration) without further modifications.

5.3 Bandwidth Efficiency Analysis

Figure 15 compares the performance of FR-FCFS and RL-based schedulers in 4-core systems with 6.4GB/s and 12.8GB/s peak DRAM bandwidth. The latter is a dual-channel DDR2-800 SDRAM sub-system with two integrated memory controllers. Results are normalized to the performance of the 6.4GB/s FR-FCFS baseline.

The RL-based 6.4GB/s system delivers half of the speedups that can be achieved by a more expensive, 12.8GB/s configuration with FR-FCFS scheduling. Single-channel RL-based and dual-channel FR-FCFS-based systems achieve speedups of 19% and 39% over a single-channel FR-FCFS-based system, respectively. While the dual-channel FR-FCFS system offers higher peak throughput, the RL-based controller uses its single 6.4GB/s channel more efficiently. Hence, by improving DRAM utilization, a self-optimizing memory controller can deliver 50% of the speedup of an over-provisioned system, at a lower system cost.

Figure 15 also shows that the RL-based scheduler can utilize a dual-channel, 12.8GB/s interface much more effectively

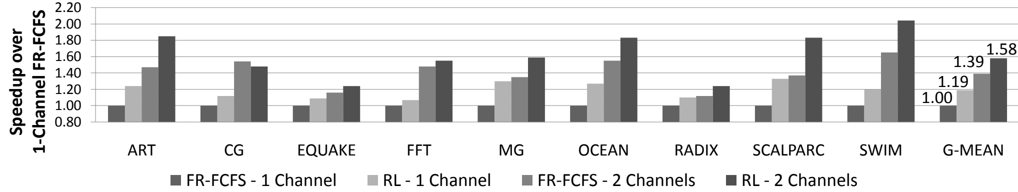

Figure 15: Performance comparison of FR-FCFS and RL-based memory controllers on systems with 6.4GB/s and 12.8GB/s peak DRAM bandwidth

than the FR-FCFS scheduler. Consequently, the RL-based dual-channel system outperforms the other three configurations by significant margins. In ART and SCALPARC, for instance, a dual-channel system with an RL-based scheduler outperforms the baseline single-channel system by 85% and 83%, respectively. With an FR-FCFS scheduler, however, the same system achieves speedups of only 47% and 37% over the single-channel baseline. Thus, RL-based memory controllers are also an attractive design option for more expensive (and higher performance) CMP memory systems.

5.4 Comparison to QoS-Aware Schedulers

Most memory controller proposals [17, 25, 30, 36, 37, 38, 48, 49], including ours, aim at increasing overall system performance by delivering higher DRAM throughput and/or lower access latency. With the proliferation of CMPs, however, there is growing interest in hardware platforms that can provide performance isolation across applications in a multiprogrammed environment. In particular, several QoS-aware memory controller proposals have been published lately [29, 31, 34].

Providing QoS guarantees in a multiprogrammed environment using an RL design is beyond the scope of our paper and is left for future work. Nevertheless, in the context of executing a parallel application, it is conceivable that a QoS-aware scheduler could help performance in a variety of ways, for example by reducing starvation. To understand whether the speedups delivered by our RL-based memory controller could also be achieved by simply addressing any potential fairness problems across threads, we evaluate Nesbit et al.'s Fair Queueing scheduler [31]. Figure 16 shows the results.

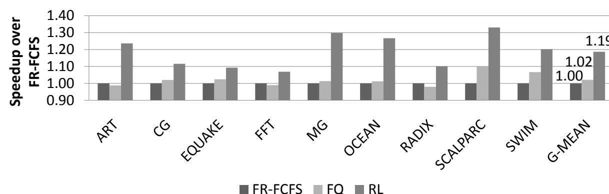

Figure 16: Performance comparison of Fair Queueing and RL-based memory controllers

The fair queueing scheduler provides significant speedups over the baseline FR-FCFS scheduler for two applications (SCALPARC and SWIM); the average speedup across all nine applications is 2%. While the RL-scheduler is designed specifically to optimize long-term system performance via runtime interaction with the system, the fair queueing scheduler addresses only one potential source of performance loss (load imbalance) as a by-product of improving fairness.

6. RELATED WORK

6.1 DRAM Scheduling

None of the previous work in DRAM scheduling provides a scheduler that can learn the long-term impact of its scheduling decisions and plan accordingly to optimize a given performance metric. To our knowledge, this paper is the first to propose such a scheduler along with a rigorous methodology to designing self-optimizing DRAM controllers.

Hur and Lin [17] propose a history-based memory scheduler that adapts to the mix of read and write requests from the processor. They target inefficiencies arising in a lower-performing memory controller organization than our baseline. Their scheduler chooses among multiple hard-coded scheduling policies based on the history of read and write requests it receives. Unlike an RL-based controller, their scheme (1) cannot learn a new scheduling policy that is not already implemented by the hardware designer, (2) cannot learn the long-term performance impact of its scheduling decisions, (3) does not consider the scheduling of row commands (i.e., precharge and activate), and therefore does not completely address the trade-offs involved in DRAM command scheduling. We have adapted their scheduler to our baseline controller organization, but found its performance impact to be negligible (less than 0.3% improvement over FR-FCFS on average) in our more aggressive setup.

Rixner et al. [36, 37] examine various DRAM command scheduling policies and propose the FR-FCFS policy. Hong et al. [16] and McKee et al. [25] describe policies that reorder accesses from different streams in stream-based computations. Zhu and Zhang [48] present modifications to existing scheduling policies to improve system throughput in simultaneous multithreading processors. QoS-aware memory controllers were recently proposed [29, 31, 34] to provide fair access to threads sharing the DRAM system. Shao and Davis [38] describe an access reordering based burst scheduling policy to utilize the DRAM burst mode more effectively. Natarajan et al. [30] examine the impact of different policy decisions in the memory controller in a multiprocessor environment and find that better policies that more efficiently utilize DRAM bandwidth could provide the same performance as doubling the DRAM bandwidth. Ahn et al. [2] study the effect of DRAM scheduling policies and DRAM organization on the performance of data-parallel memory systems. All these previously proposed policies are fixed, rigid, non-adaptive, and unable to learn the long-term effects of their scheduling decisions on performance. In contrast to these, an RL-based controller learns and employs new and adaptive scheduling policies based on system and workload behavior.

Other work has proposed techniques for intelligent address remapping [8], eliminating bank conflicts [44, 47], and refresh scheduling [15] in DRAM controllers. These approaches are orthogonal to our proposal.

6.2 Applications of RL in Computer Systems

Reinforcement learning [6, 28, 42] has been successfully applied to a broad class of problems, including autonomous navigation and flight, helicopter control, pole balancing, board games with non-deterministic outcomes (e.g., backgammon), elevator scheduling, dynamic channel assignment in cellular networks [32], processor and memory allocation in data centers [43, 45], routing in ad-hoc networks [10], and the control of pricing decisions in shopbots. Its application to microarchitecture has been limited.

Tesauro et al. [43] explore a reinforcement learning approach to make autonomic resource allocation decisions in data centers. They focus on assigning processors and memory to applications. The proposed algorithm delivers significant speedups over a variety of existing queueing-theoretic allocation policies.

McGovern and Moss [24] apply reinforcement learning and Monte Carlo roll-outs to code scheduling, finding that the learned scheduling policies outperform commercial compilers when compiling for the Alpha 21064 microprocessor.

Other machine learning techniques have been used in the context of branch prediction. Calder et al. [7] use neural networks and decision trees to predict conditional branch directions statically, using static program features. Jimenez and Lin [19] propose the perceptron branch predictor, which outperforms table-based prediction schemes by significant margins. Emer and Gloy [14] propose the use of genetic programming for automatic derivation of hardware predictors at design time. They do not consider the derivation of on-line architectural control policies. Genetic programming does not utilize the properties of the underlying Markov Decision Processes, and may require a very large

number of repeated evaluations on a given set of benchmarks before delivering a high-performance policy. In contrast, reinforcement learning methods can learn and generalize from individual decisions, explicitly balance exploration and exploitation, and thus are suitable for on-chip learning at run-time.

7. CONCLUSIONS

We have presented a new approach to designing memory controllers that operate using the principles of reinforcement learning (RL). An RL-based, self-optimizing memory controller continuously and automatically adapts its DRAM command scheduling policy based on its interaction with the system to optimize performance. As a result, it can utilize DRAM bandwidth more efficiently than a traditional controller that employs a fixed scheduling policy. Our approach also reduces the human design effort for the memory controller because the hardware designer does not need to devise a scheduling policy that works well under all circumstances.

On a 4-core CMP with a single-channel DDR2-800 memory subsystem (6.4GB/s peak bandwidth in our setup), the RL-based memory controller improves the performance of a set of parallel applications by 19% on average (up to 33%) and DRAM bandwidth utilization by 22% on average, over a state-of-the-art FR-FCFS scheduler. This improvement effectively cuts in half the performance gap between the single-channel configuration and a more expensive dual-channel DDR2-800 subsystem with twice peak bandwidth. When applied to the dual-channel subsystem, the RL-based scheduler delivers an additional 14% performance on average (up to 34%). We conclude that RL-based self-optimizing memory controllers provide a promising way to efficiently utilize the DRAM memory bandwidth available in a CMP.

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.

8. REFERENCES

[1] Advanced Micro Devices, Inc. *AMD Athlon$^{(TM)}$ XP Processor Model 10 Data Sheet*, Feb. 2003.

[2] J. H. Ahn, M. Erez, and W. J. Dally. The design space of data-parallel memory systems. In *SC*, 2006.

[3] Anandtech. Intel Developer Forum 2007. http://www.anandtech.com/cpuchipsets/intel/showdoc.aspx?i=3102.

[4] V. Aslot and R. Eigenmann. Quantitative performance analysis of the SPEC OMPM2001 benchmarks. *Scientific Programming*, 11(2):105–124, 2003.

[5] D. H. Bailey et al. NAS parallel benchmarks. Technical report, NASA Ames Research Center, March 1994. Tech. Rep. RNR-94-007.

[6] D. Bertsekas. *Neuro Dynamic Programming*. Athena Scientific, Belmont, MA, 1996.

[7] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence-based static branch prediction using machine learning. *ACM TOPLAS*, 19(1), 1997.

[8] J. Carter et al. Impulse: Building a smarter memory controller. In *HPCA-5*, 1999.

[9] R. Caruana and D. Freitag. Greedy attribute selection. In *International Conference on Machine Learning*, pages 28–36, New Brunswick, NJ, July 1994.

[10] Y. Chang, T. Hoe, and L. Kaelbling. Mobilized ad-hoc networks: A reinforcement learning approach. In *International Conference on Autonomic Computing*, 2004.

[11] V. Cuppu, B. Jacob, B. T. Davis, and T. Mudge. A performance comparison of contemporary DRAM architectures. In *ISCA-26*, 1999.

[12] B. T. Davis. *Modern DRAM Architectures*. Ph.D. dissertation, Dept. of EECS, University of Michigan, Nov. 2000.

[13] T. Dunigan, M. R. Fahey, J. White, and P. Worley. Early evaluation of the Cray X1. In *SC*, 2003.

[14] J. Emer and N. Gloy. A language for describing predictors and its application to automatic synthesis. In *ISCA-24*, 1997.

[15] M. Ghosh and H.-H. S. Lee. Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3D Die-Stacked DRAMs. In *MICRO-40*, 2007.

[16] S. I. Hong, S. A. McKee, M. H. Salinas, R. H.Klenke, J. H. Aylor, and W. A. Wulf. Access order and effective bandwidth for streams on a direct Rambus memory. In *HPCA-5*, 1999.

[17] I. Hur and C. Lin. Adaptive history-based memory schedulers. In *MICRO-37*, 2004.

[18] ITRS. *International Technology Roadmap for Semiconductors: 2005 Edition, Assembly and packaging*. http://www.itrs.net/Links/2005ITRS/AP2005.pdf.

[19] D. A. Jimenez and C. Lin. Dynamic branch prediction with perceptrons. In *HPCA-7*, 2001.

[20] M. Joshi, G. Karypis, and V. Kumar. ScalParC: A new scalable and efficient parallel classification algorithm for mining large datasets. In *IPPS*, 1998.

[21] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 9(2):24–36, Mar. 1999.

[22] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[23] J. D. McCalpin. *Sustainable Memory Bandwidth in Current High Performance Computers.* http://home.austin.rr.com/mccalpin/papers/bandwidth/.

[24] A. McGovern and E. Moss. Scheduling straight line code using reinforcement learning and rollouts. In *Advances in Neural Information Processing Systems*, 1999.

[25] S. A. McKee et al. Dynamic access ordering for streamed computations. *IEEE Transactions on Computers*, 49(11):1255–1271, Nov. 2000.

[26] Micron. *512Mb DDR2 SDRAM Component Data Sheet: MT47H128M4B6-25*, March 2006. http://download.micron.com/pdf/datasheets/dram/ddr2/512MbDDR2.pdf.

[27] Micron. *Technical Note TN-47-04: Calculating Memory System Power for DDR2*, June 2006. http://download.micron.com/pdf/technotes/ddr2/TN4704.pdf.

[28] T. Mitchell. *Machine Learning*. McGraw-Hill, Boston, MA, 1997.

[29] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO-40*, 2007.

[30] C. Natarajan, B. Christenson, and F. Briggs. A study of performance impact of memory controller features in multi-processor server environment. In *WMPI*, 2004.

[31] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *MICRO-39*, 2006.

[32] J. Nie and S. Haykin. A Q-learning-based dynamic channel assignment technique for mobile communication systems. In *IEE Transactions on Vehicular Technology*, Sept. 1999.

[33] J. Pisharath, Y. Liu, W. Liao, A. Choudhary, G. Memik, and J. Parhi. NU-MineBench 2.0. Technical report, Northwestern University, August 2005. Tech. Rep. CUCIS-2005-08-01.

[34] N. Rafique, W.-T. Lim, and M. Thottethodi. Effective management of DRAM bandwidth in multicore processors. In *PACT*, 2007.

[35] J. Renau, B. Fraguela, 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.

[36] S. Rixner. Memory controller optimizations for web servers. In *MICRO-37*, 2004.

[37] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA-27*, 2000.

[38] J. Shao and B. T. Davis. A burst scheduling access reordering mechanism. In *HPCA-13*, 2007.

[39] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5):505–521, 2005.

[40] L. Spracklen and S. G. Abraham. Chip multithreading: Opportunities and challenges. In *HPCA-11*, 2005.

[41] R. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Neural Information Processing Systems Conference*, Denver, CO, June 1996.

[42] R. Sutton and A. Barto. *Reinforcement Learning*. MIT Press, Cambridge, MA, 1998.

[43] G. Tesauro et al. Online resource allocation using decompositional reinforcement learning. In *Conference for the American Association for Artificial Intelligence*, July 2005.

[44] M. Valero et al. Increasing the number of strides for conflict-free vector access. In *ISCA-19*, 1992.

[45] D. Vengerov and N. Iakovlev. A reinforcement learning framework for dynamic resource allocation: First results. In *ICAC*, 2005.

[46] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA-22*, 1995.

[47] Z. Zhang et al. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *MICRO-33*, 2000.

[48] Z. Zhu and Z. Zhang. A performance comparison of DRAM memory system optimizations for SMT processors. In *HPCA-11*, 2005.

[49] W. K. Zuravleff and T. Robinson. Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order. United States Patent #5,630,096, May 1995.