Appears in Intl. Symp. on High Performance Computer Architecture (HPCA)
New Orleans, LA, Feb. 2012

MORSE: Multi-objective Reconfigurable Self-optimizing Memory Scheduler

Janani Mukundan **José F. Martínez**

Computer Systems Laboratory
Cornell University
Ithaca, NY, 14850 USA

http://m3.csl.cornell.edu/

We propose a systematic and general approach to designing self-optimizing memory schedulers that can target arbitrary figures of merit (e.g., performance, throughput, energy, fairness). Using our framework, we instantiate three memory schedulers that target three important metrics: performance and energy efficiency of parallel workloads, as well as throughput/fairness of multiprogrammed workloads. Our experiments show that the resulting hardware significantly outperforms the state of the art in all cases.

1 INTRODUCTION

Modern high-performance memory subsystems support a high degree of concurrency. This is primarily accomplished by increasing the number of independent channels and/or increasing the number of independent banks in a channel [7, 8, 9, 16, 34]. It is critical that the memory controller be able to produce a schedule that can leverage this potential concurrency, all while abiding by numerous strict timing constraints imposed by the DRAM.

Traditionally, computer architects have primarily optimized DRAM controllers for performance [8, 14, 24, 25, 28, 34]. This was appropriate, as the gap between the CPU and memory speed kept growing at the time.

DRAM energy consumption has been given due consideration only relatively recently [10, 15]. In current and upcoming multicore-based servers, DRAM accounts for a significant fraction of power consumption [21]. Therefore, apart from performance, power and energy are also becoming first-order issues while designing memory schedulers for multicore systems.

In addition to these issues, DRAM memory bandwidth is a critical shared resource in a multi-core system, and it is important to efficiently share the memory bandwidth among multiple threads running in a multicore environment, so as to not adversely affect system throughput and fairness.

Several scheduling algorithms have been proposed in the past to tackle the problems listed above. Most such proposals are relatively inflexible in two ways: (1) they have a limited ability to adapt to the environment and to improve automatically with experience; and (2) they each target a particular objective function.

İpek et al. [17] propose the use of reinforcement learning (RL) [32] to design high-performance *self-optimizing* memory schedulers. Reinforcement learning works by interacting with the environment and learning automatically with experience to pick the actions that maximize a desired long-term objective function. İpek et al. show that, when used to target performance, this approach can outperform existing ad hoc designs by a significant margin.

Still, İpek et al.'s methodology has a key limitation: they do not propose a generalizable way to target an objective function (performance in their case). Because it is intuitive that bus utilization and throughput (and ultimately performance) correlate strongly for memory-intensive applications, it was natural and entirely appropriate for them to take a completely ad hoc approach to designing the RL reward function, by trivially rewarding load/store commands over precharge and activate commands. Unfortunately, this approach becomes much more difficult in other important scenarios that target more sophisticated objective functions (e.g., metrics that combine performance, energy, and/or fairness).

This work builds upon İpek et al.'s RL-based framework. **We propose MORSE, a *systematic* and *general* mechanism to designing self-optimizing DRAM schedulers that can target *arbitrary* figures of merit**. We employ genetic algorithms to automatically calibrate the relative importance that the scheduler places on the different DRAM actions for a given environment and objective function (Section 3.1.2). We also employ a *multi-factor* variation of feature selection that takes into account first-order interactions among system attributes, which are used by the scheduler to sense the system's state at each point in time (Section 3.1.3). Importantly, the resulting hardware need not directly observe the objective function on the field: only during training at design time (using simulation models) does our framework require the objective function to be observable. This allows our framework to target relatively sophisticated figures of merit that would be generally hard to measure on the field (e.g., weighted speedup). Still, the hardware can be made to allow for on-the-field reconfiguration (Section 4).

Using this general approach, we rebuild İpek et al., and present quantitative evidence of significantly higher performance with respect to their original design (Section 6). We also use our general mechanism to design DRAM schedulers that can target energy efficiency (Section 7), as well as throughput/fairness of multiprogrammed workloads (Section 8). The designs prove significantly superior to the state of the art in each case.

2 BACKGROUND

2.1 Power-Aware DRAM Interfaces

A basic DRAM interface has one or more DRAM channels; each channel consists of one or more memory modules.

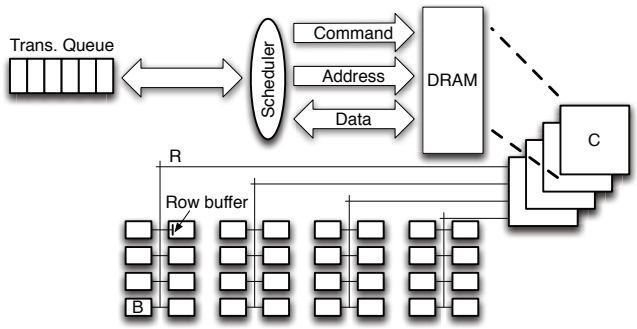

Figure 1. Basic DRAM Interface, with four independent channels (C), one quad ranked DIMM per channel (R), and eight internal banks (B) per rank.

Most modern DRAM systems make use of dual in-line memory modules (DIMM); each DIMM consists of one or more *ranks*–a set of DRAM devices that operate in lockstep. Each DRAM device contains a set of independent memory arrays called *banks*. A bank is made up of *rows* (a.k.a. *DRAM pages*), which are simply a set of storage cells that are acted upon in parallel. Figure 1 shows a DRAM system interface, with four independent DRAM channels (C). Each channel has one quad ranked DIMM (R), with eight internal banks (B) per rank.

Read and *Write* commands to a bank can only take place to locations within one row at a time, which must be first copied into the bank's *row buffer* (*Activate* command). Prior to accessing a different row, the one currently stored in the row buffer must be written back to its permanent location (*Precharge* command). Finally, since DRAM is non-persistent, rows need to be periodically read out and restored to maintain data integrity (*Refresh* command). To make matters more difficult, modern DRAM chips have a large number of timing constraints that must be obeyed when scheduling commands, which any memory scheduler must work around.

Current DDR3 SDRAMs have the capability of placing a rank in low-power mode. It may take as little as one DRAM cycle to place a DDR3 rank in low-power mode [2], however additional timing constraints must typically be met, depending on the DRAM command that is in progress. A rank in low-power mode must be powered back up before it can accept commands. Powering up a rank in DDR3 systems can take anywhere between 4-13 DRAM cycles [2]. Section 5.1 provides more details on the DDR3 DRAM interface that we model.

2.2 Basics of Reinforcement Learning

A reinforcement learning (RL) agent interacts with a probabilistic environment for the purpose of maximizing some notion of a long-term reward [32]. At each point in time, the agent does not necessarily pursue the action that offers the highest *immediate* reward; instead, the agent strives to take the action that provides the best *cumulative* reward over time. To learn how to do this, the agent needs to explore its environment carefully: Early exploitation (i.e., picking the action that seems most profitable in the long term at each point in time based on acquired knowledge) may result in an agent stuck with low-performing policies, while too much exploration (i.e., trying different actions) may cause the agent to take a long time to settle on an optimal policy. Moreover, the agent must never stop exploring completely if it is to adapt its policy to changes in the environment (e.g., program phases).

A basic RL model consists of: (1) a set of states that sufficiently describes the environment and the problem being solved; (2) a set of actions that the RL agent can perform; and (3) a reward function that assigns credit for performing an action in a state and moving to another state. In the context of DRAM scheduling, the RL agent is the memory scheduler, the pending requests and the state of the CPU and memory subsystem constitute the environment, and the legal DRAM commands at each point in time are the actions that the RL agent can perform [17]. The set of states and the reward function need to be determined depending on the long-term goal that needs to be achieved. At every time step: (1) the memory scheduler observes the state of the environment; (2) among the actions available for all the pending requests,[1] the memory scheduler chooses the one action that will maximize the cumulative reward; and (3) the memory controller performs that action, which results in a state change.

The agent needs to learn how to assign credit and blame for the actions it takes. A common way of learning to assign credit is through a technique called Q-learning. Formally, the Q-value of a state-action pair (s,a) while executing a policy π, $Q_\pi(s,a)$, is the expected cumulative reward resulting from taking action a in state s and following policy π thereafter. A Q-learning-based RL agent learns the optimal policy π^* indirectly, by learning $Q_{\pi^*}(s,a)$ for every state-action pair (s,a) (the *Q-value matrix*).

States are often represented as tuples of *attributes*. Because the size of the state space (in the case of Q-learning, the size of the Q-value matrix) is exponential in the number of attributes considered (this is often referred to as the "curse of dimensionality"), it is essential that the number of attributes and the resolution of each attribute be contained. This helps not only in reducing storage and speed requirements in a silicon implementation of the Q-value matrix; it also allows the RL agent to *generalize*, i.e., exploit knowledge acquired through past experience–in the case of Q-learning, approximate the Q-value of a previously unseen state-action pair (s,a) with the Q-value of state-action pair (s',a), with s and s' sufficiently close in the state space.

3 A GENERAL FRAMEWORK FOR SELF-OPTIMIZING MEMORY SCHEDULERS

In this section we describe how to generalize İpek et al.'s original RL-based memory scheduler design [17] to obtain high-quality schedulers that can target arbitrary objective functions–not just performance. We first present our design approach, and then describe a practical implementation.

3.1 Design

We now determine the three main characteristics of our RL-based design: actions, state attributes, and reward structure.

[1] Not all pending requests will have actions available at any point in time: For example, if a row has not yet been activated, a read to that row is not an available action. Even among available actions, only a subset may be evaluated in order to reach a decision every DRAM cycle.

3.1.1 Available Actions

Concurrently to sensing the environment's state (Section 3.1.3), the scheduler determines whether a valid DRAM command exists for each pending memory request among:

Activate: Bring the contents of a bank's DRAM row into the bank's row buffer.

Precharge: Write the contents of a bank's row buffer back to the corresponding DRAM row. We categorize as a separate action the case where there are no active requests for a particular (open) row, yet the scheduler still may choose to preemptively precharge; we call this Preemptive Precharge.

Read(Load), Read(Store): Perform a read from a bank's row buffer.

Write: Perform a write to a bank's row buffer.

Rank Power Down (PwDn): Place the corresponding rank into a low power mode. When a rank is in low power mode, it cannot be accessed. Current DRAM subsystems already provide support for such low-power rank modes; in our implementation, we use those of the DDR3 interface [2].

Rank Power Up (PwUp): Bring the corresponding rank back to normal operation mode.

NoOp: If no legal DRAM command exists for this cycle (often due to DRAM timing constraints), the scheduler will do nothing and wait for the next cycle. (But a rank may remain powered down even if PwUp is a legal command.)

3.1.2 Reward Structure

In order to explore the environment, the scheduler implements an exploration mechanism known as ε-greedy action selection: Every DRAM cycle, with a small probability ε, the scheduler picks a random (legal) action; at all other times, it picks the (legal) action with the highest Q-value. This guarantees that there is a non-zero probability of visiting every entry in the Q-value matrix.

Each action is associated with an *immediate reward*. Once action a_t is picked and the immediate reward is determined, the Q-value prediction associated with the state-action pair (s_{t-1}, a_{t-1}) that was picked in the previous cycle $t-1$ can be updated using SARSA [32] as follows:

$$Q(s_{t-1}, a_{t-1}) \leftarrow (1-\alpha)Q(s_{t-1}, a_{t-1}) + \alpha[r_t + \gamma Q(s_t, a_t)]$$

where α is the *learning rate*, empirically determined;[2] r_t is the immediate reward collected for the action taken; and $0 \leq \gamma < 1$ is a *discount factor* that causes future rewards to be incorporated in the form of a geometric series.[3]

In the performance-oriented design of İpek et al. [17], the immediate reward function is picked solely based on expert intuition. Since the memory throughput (and ultimately execution time) of a memory-bound application tends to correlate strongly with the effective data bus utilization, the authors trivially assign an immediate reward of 1 to a read or write DRAM command, and an immediate reward of 0 to any other DRAM command. Unfortunately, this approach does not easily generalize: In a design that seeks to optimize a more sophisticated function (e.g., Et^2 or weighted speedup), an appropriate immediate reward function is not at all evident.

Automatic Derivation of Reward Structures

In this paper we propose to follow an automated approach to solve this problem. Specifically, we devise a genetic algorithm (GA) [23] to explore the search space of possible reward functions for a given objective function. [4]

Genetic algorithms (GAs) [23] are a heuristic search technique that is based on evolutionary processes. GA starts by randomly generating a population space of individuals, where each individual is a candidate solution for the problem being solved. Typically individuals in the population set are represented in binary as strings of 0's and 1's, but other encodings are also possible. Evolution is performed in generations and it starts by evaluating the fitness of the initial population according to an objective function. (Evaluation means simulations of candidate DRAM schedulers in our case.) Based on the fitness of individuals in the population set, the next generation of individuals are determined stochastically using some form of fitness based selection technique. These new individuals are then further evolved using operations like crossover and mutation, which leaves us with a population for the next generation. This is done iteratively until a certain number of generations has been evolved, or when a certain fitness level has been reached, after which the search is terminated. While many of the individuals in the initial population might not do anything useful, the evolutionary nature of GAs allow some of them to evolve into meaningful, high-performing solutions, and shed the rest in the process.

In our GA, each individual in the population stores rewards for each of the eight actions that can be performed by the scheduler. Initially, these rewards are randomly generated. We evaluate our initial population by conducting execution-driven simulations with each individual's memory scheduler configuration, using a small subset of our application set[5] and determining the fitness of each individual. The fitness-based selection criteria that we use is *tournament selection* combined with *elitist selection* [23, 19]. To perform crossover, we randomly pick two individuals and swap the reward values of an action. Mutation is performed by randomly replacing the reward of an action with another value. Multiple-point crossover and mutations are performed

[2] A high learning rate quickly substitutes past knowledge with new information, whereas a small learning rate incorporates new knowledge slowly.

[3] Intuitively, γ can be interpreted as a knob that controls how important future rewards are relative to immediate rewards; larger γ values introduce more foresight at the expense of longer training times.

[4] We did try "simpler" search techniques, such as manual trial-and-error or automatic hill-climbing with random restarts and momentum, but the end result was significantly inferior. We believe GA offers a good trade-off between simplicity and effectiveness in this context.

[5] The applications that we use for training are *fft*, *mg*, and *radix* (Section 5.2). We picked these because they are the fastest to simulate among the parallel applications that we evaluate. By using a small subset, picked *not* based on behavior but simply on execution time, we speed up training and at the same time minimize the chance of overfitting the final solution to our application set.

in our experiments, which means that reward values can be swapped or replaced multiple times within a given individual. Once we have the population set for the next generation, it is evaluated against the fitness criteria, and this iterative evolutionary search process continues until we reach 50 generations, at the end of which we are left with a set of rewards, one per possible action, which together constitute our reward function.

In theory, it should be possible to periodically re-calibrate the reward values as the application goes through different execution phases. These rewards would need to be re-learned on the fly by the hardware. We are currently investigating this aspect, but in this paper we confine our solution to static rewards learned offline, which still yields good results and simplifies the design of the scheduler (as we will see, the reward structure is just a small table).

3.1.3 State Attributes

Every memory cycle, the scheduler senses the environment's state via a set of attributes. During the design of the scheduler, it is important to pick the right kind of state attributes that will adequately represent the system environment. There are many candidate attributes that can be used to describe the state of the system. However, to obtain an implementation with reasonable delay and silicon area, it is critical to use a good selection mechanism that picks the right (small) set of state attributes.

Multi-factor Feature Selection

A quick and relatively simple way to accomplish this is to use a linear feature selection process [17]. The designer picks a set of N candidate attributes based on expert intuition. The first step involves simulating N schedulers, each of which uses only one of the N candidate attributes to determine the state of the memory system. Among these N attributes, the designer picks the attribute t_1 that optimizes an objective function (e.g., performance). Then, the designer repeats the selection process with $N-1$ schedulers, each one considering t_1 and one of the remaining $N-1$ attributes. After $i \ll N$ iterations, the process concludes, and the i attributes picked determine the state representation.

This linear procedure ignores potentially important interactions between attributes (e.g., attribute t_x alone yields the highest-performing scheduler during iteration 1, but combination $< t_y, t_z >$ may be superior than $< t_x, t_k >$ for any $1 \leq k \neq x \leq N$), which we experimentally observed are important in our context. In this paper we propose a *multi-factor* approach that takes into account first-order attribute interactions.[6] At the end of the first iteration, we pick the top *two* attributes, and explore the resulting two branches concurrently; at the end of the second iteration, we again pick the top two attributes from each of the two branches, and proceed down four branches; etc. The obvious downside of this approach is that the number of simulations is much higher: for $N = 50$ and $i = 6$, our methodology yields on the order of 8,600 simulations, using the same three training applications per design point. Fortunately, feature selection is a one-time effort made at design time.[7] The resulting attributes are, in principle, inextricably linked to the objective function targeted in the simulations. In Section 4, however, we will show that a carefully-trained design can successfully tackle variations of an objective function, by simply reprogramming the reward structure.

Finally, the astute reader will notice that there is a "circular dependence" between automatic feature selection and automatic reward structure derivation: both search a space of completely specified memory scheduler designs. What we do in our paper is to impose a basic ad hoc reward structure during feature selection (Read = Write = 1, rest = 0), but still use the appropriate objective function when evaluating candidate state attributes, and then use the resulting state attributes in the computation of the true reward structure. One could conceive iterating over these two steps to potentially refine the outcome, however for simplicity we do not explore this in this work.

3.2 Implementation

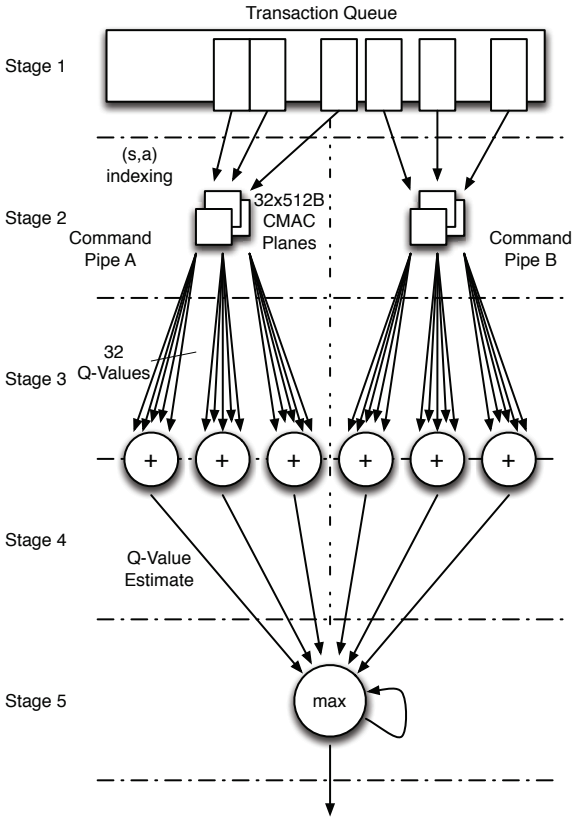

Figure 2. Snapshot of the dual five-stage pipeline used in this study to pick the DRAM command with the highest Q-value, among up to 24 eligible actions (four waves of six actions) which can be evaluated every DRAM cycle.

The basic structure of our implementation is necessarily similar to the one described by İpek et al. [17]. In general, only a fraction of the (maximum) 64 outstanding memory

[6] Our mechanism trivially generalizes to higher orders, however we experimented with second-order interactions as well and found no differences in the final state representation.

[7] For each design, we were able to complete all 8,600 simulations in one day.

requests will have an associated ready DRAM command at each point in time (i.e., one that can be issued to process a memory request without violating timing constraints). We empirically determine that evaluating 24 ready DRAM commands per DRAM cycle is sufficient to deliver performance that is very close to peak.

For a DRAM scheduler pipeline clocked at 4.27 GHz (same as the CPU, since the memory scheduler sits on chip)[8], controlling a DDR3-1066 system, the Q-value estimation pipeline can be clocked eight times every DRAM cycle. Thus, we use two five-stage command pipes, each capable of considering three ready DRAM commands every processor cycle, for a total of up to 24 ready DRAM commands (four waves of six commands) every DRAM cycle. If there are more memory requests with ready DRAM commands on the queue, the scheduler simply takes the ready DRAM command with the highest Q-value found by the end of the DRAM cycle. (Alternatively, more commands can be considered per cycle if the more pipes are added.) Figure 2 shows the five-stage pipeline structure that is used to calculate the Q-values of the proposed scheduler.

In the first stage of the pipeline, the scheduler retrieves six ready DRAM requests (three per command pipe), and it generates the corresponding state-action pairs. (Information about state attributes is actually sensed during the previous DRAM cycle.) In the second pipeline stage, the indices for the Q-value tables are generated and are use to read the Q-values out of the CMAC arrays. The Q-value tables follow a CMAC organization [31]: each Q-value is in fact the result of indexing multiple relatively small tables, each index shifted by an amount predetermined randomly at design time, then adding the result from all such tables together. This structure provides a good balance between resolution and generalization [17]. In our design, we carefully account for the delay and power incurred by this structure, which essentially amounts to 32 SRAM tables, each with 256 two-byte (fixed-point) entries, three read ports and one write port, and where the updates to the Q-values are done using fixed point arithmetic. The index for the CMACs are generated by concatenating the higher order bits of the state attributes. This concatenated value is then XOR-ed with a constant, depending on the corresponding action that was chosen. Finally, the XOR-ed value is passed through a hash function (to reduce storage requirements) to get the index into the Q-value tables. The next two stages are used to add up the 32 Q-values of the three different commands analyzed per command pipe, which have been read out from the CMAC arrays. In the fifth and final stage of the pipeline, the maximum Q-value seen so far is compared against the six new Q-values and updated as needed.

4 RECONFIGURABILITY

We have described our general framework as a design-time mechanism to build self-optimizing memory schedulers. In this section, we briefly outline how one can in fact implement a self-optimizing scheduler that can accommodate multiple objective functions, and target each as desired on the field (post-silicon), either at boot time or even dynamically by the operating system.

[8]The clock frequency of IBM's server class Power7 CPU is 4.25 GHz at 45 nm (we target 32 nm in our calculations), with four processor cores and all memory controllers simultaneously on.

Actions – The actions available to the self-optimizing scheduler are simply the possible DRAM commands, irrespective of the objective function. Therefore, no hardware changes are required to map actions to commands for a different configuration.

Rewards – There are as many rewards as there are actions in the RL agent. These rewards can be stored easily using a table inside the RL agent. There are two approaches to storing rewards for multiple configurations: (a) A single table that can be programmed as needed (e.g., by the operating system) to store the values relevant to the desired objective function. (b) The memory scheduler can store a small number of pre-programmed tables, and use the appropriate one as needed. A combination of both is also possible (i.e., multiple programmable tables).

State Attributes – Although selecting state attributes is an offline process, the resulting state attributes must be sensed by adding the appropriate hardware (e.g., a counter, a read port, etc.). However small this may be, as the number of metrics of interest increase (e.g., to support more than one possible objective function), so will the aggregate hardware overhead. In general, the solution will be a compromise in the number of observable state attributes, driven by potential gains vs. area and complexity.

Fortunately, the CMAC structure that stores the Q-values (the main storage overhead) is itself attribute-agnostic–the differentiation across objective function resides in what attributes are actually sensed, but the indexing into the CMAC is identical regardless of the attribute [17]. Thus, we can easily add multiplexers to steer the right set of attributes to the CMAC depending on the target objective function.

5 EXPERIMENTAL METHODOLOGY

5.1 Architecture Model

The baseline processor model integrates eight cores and supports a DDR3-1066 memory subsystem with four independent, address-interleaved memory channels. Our memory subsystem model (DIMM structure, timing, and power) follows the Micron DDR3 DRAM specification [2, 3, 4], including refresh. The micro-architectural features of the baseline processor are shown in Table 1; the parameters of the L2 cache, the memory system, and the DDR3 SDRAM power model are shown in Tables 2 and 3. We implement our model by extending the SESC simulation environment appropriately [27].

We compute the energy overhead of the self-optimizing scheduler designs as follows, assuming a 32 nm process except where noted.

Dynamic power – Q-value computation: Computing a Q-value consists of three basic steps: (a) generating the array indices, (b) reading the Q-values, and (c) adding the Q-values and determining the maximum Q-value. To generate the array indices, we first read the six selected state attributes and concatenate the higher order bits. This is then XOR-ed with a random number and passed through a hash function. Reading the state attributes and indexing into a hash function can be approximated as a dynamic SRAM read each and consumes 1.4 pJ per read (from CACTI 6.5 [1]). The XOR function takes 0.23 pJ (an XOR function is conservatively approximated to consume the same power as an adder implemented in an older 70 nm technology [18]). We use

CACTI to estimate the energy expended in reading out the Q values from the SRAM arrays. Each SRAM read consumes 0.78 pJ, and the total dynamic SRAM energy for reading out the Q values from the 32 matrices per command is 24.96 pJ. We estimate the power consumed by the adders that sum up the 32 Q-values to be 1.0 mW each [18], and accordingly calculate the energy consumed by the 16 adders used in each RL pipeline to be 3.75 pJ (adding the 32 Q-values takes up two pipeline cycles). We conservatively assume that the comparator consumes the same power as the adder. Since a maximum of 24 commands can be analyzed every DRAM cycle, a maximum of 24 final Q-values need to be compared each cycle, and the energy estimated to do so is 3 pJ. The total RL pipeline energy consumed is then 32 pJ.

Dynamic power – Q-value update: To update the Q-values using the SARSA update rule we use three multipliers and three adders. The energy consumed to perform this operation is 2.1 pJ [18]. Finally the Q-values need to be written back into the SRAM arrays (32 per command, 64 in all). This consumes 49 pJ as estimated from CACTI.

Leakage: Using CACTI, we also estimate the total leakage power per CMAC matrix to be 0.36 mW, and consequently the leakage energy to be 5.4 pJ per DRAM cycle.

Overall, we find the energy overhead of the self-optimizing schedulers to be negligible (the equivalent of about 2% of the energy consumed by the DRAM on average). Nevertheless, the energy and Et^2 results reported in sections 6.1, 7.1, and 8 *do* include this overhead. Moreover, in our results we effectively assess zero energy overhead for the competing FR-FCFS and Pwr-FR-FCFS schemes.

5.2 Applications

We evaluate our proposed MORSE scheduler on a wide variety of parallel and multi-programmed workloads from the server and desktop computing domains. We simulate nine memory-intensive parallel applications, running eight threads each, to completion. Our parallel workloads constitute a good mix of scalable scientific programs from different benchmark suites, as shown in Table 4. For our multi-programmed workloads, we use 14 four-application bundles from the SPEC 2000 and NAS benchmark suites, which constitute a healthy mix of CPU-, cache-, and memory-sensitive applications. Table 5 describes the bundles. In each case, we fast-forward each application for half a billion instructions, and then execute the bundle concurrently until *all* applications in the bundle have committed at least half a billion more instructions.

6 CASE I: PERFORMANCE

In this section we use our general framework to design a performance-oriented self-optimizing memory scheduler, and provide quantitative evidence of its superiority with respect to İpek et al.'s original design [17].

İpek et al.'s scheduler uses a simple ad hoc reward structure, which assigns a reward of 1 for reads and writes (immediately "productive" actions), and 0 otherwise. To allow the controller to use the most appropriate set of state attributes for our experimental setup (different from theirs), we re-run their proposed linear feature selection [17], using their six winning attributes, plus another 44 relevant ones that we come up with. By using their original attributes as part of the set, we make sure the resulting scheduler is *at least* as good as the original one. In our experiments, we call this configuration *Ipek*.

Table 1. Core Parameters.

Technology	32 nm
Frequency	4.27 GHz
Number of cores	8
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	32 B/32 B
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
RL discount rate parameter γ	0.95
RL learning rate parameter α	0.1

MORSE-P is our performance-oriented self-optimizing scheduler. It is derived using our proposed automatic reward derivation and multi-factor feature selection procedures, using performance as the objective function and the same 50 candidate state attributes.

The state attributes selected via multi-factor feature selection for MORSE-P are: (1) Number of reads (load/store misses) in the transaction queue. (2) Number of writes in the transaction queue. (3) Number of reads for the current rank under consideration. (4) If the memory request is related to a load miss, the order of the load relative to the other loads in the transaction queue for the corresponding core. (5) Number of writes in the transaction queue that reference rows that are open. (6) Number of commands for the rank under consideration when it is powered down.

Attributes (1) and (2) help the scheduler determine how to balance reads and writes in the transaction queue ; (3) prioritizes among reads from different ranks ; (4) is used to prioritize among load misses from the same core ; (5) helps the RL scheduler issue writes in bursts so as to better manage write-to-read delays ; and finally (6) determines if it is time to power up ranks if there are memory requests waiting to access the rank.

The rewards obtained from the GA-based automatic reward derivation process are: Activate = 1.59, Precharge = -1.47, Preemptive Precharge = -1.47, Read(Load) = 2.00, Read(Store) = 1.59, Write = 0.88, PwDn = -0.27, PwUp = 0.30, NoOp = 0.58.[9] The resulting values are very interesting. On the one hand, in many cases their magnitude relative to each other makes intuitive sense: For example, reads and writes have a high positive reward. Precharge is negative while Activate is positive, which hints at the importance of exploiting row buffer locality. PwDn is negative, as the

[9]We arbitrarily set up the reward structure to use higher (lower) values as positive (negative) rewards, which is typical in machine learning texts.

Table 2. Parameters of the shared L2 and DRAM.

Shared L2 Cache Subsystem	
Shared L2 Cache	4 MB, 64 B block, 8-way
L2 MSHR entries	64
L2 round-trip latency	32 cycles (uncontended)
Write buffer	64 entries
Micro DDR3-1066 DRAM [2]	
Transaction Queue	64 entries
Peak Data Rate	6.4 GB/s
DRAM bus frequency	533 MHz (DDR)
Number of Channels	4
DIMM Configuration	Quad rank
Number of Banks	8 per rank
Row Buffer Size	1 KB
Address Mapping	Page Interleaving
Row Policy	Open Page
Burst Length	8
tRCD	7 DRAM cycles
tCL	7 DRAM cycles
tWL	6 DRAM cycles
tCCD	4 DRAM cycles
tWTR	4 DRAM cycles
tWR	8 DRAM cycles
tRTP	4 DRAM cycles
tRP	7 DRAM cycles
tRRD	4 DRAM cycles
tRTRS	2 DRAM cycles
tRAS	20 DRAM cycles
tRC	27 DRAM cycles
Refresh Cycle	8,192 refresh commands every 64 ms
tRFC	59 DRAM cycles

Table 3. Parameters of the Micron DDR3-1066 DRAM power management features [2, 4, 3].

IDD0	90 mA
IDD3PF	55 mA
IDD3PS	55 mA
IDD2PF	35 mA
IDD2PS	12 mA
IDD2N	70 mA
IDD3N	80 mA
IDD4R	200 mA
IDD4W	255 mA
tFAW	20 DRAM cycles
tACTPDEN	1 DRAM cycles
tPREPDEN	1 DRAM cycles
tRDPDEN	12 DRAM cycles
tWRPDEN	18 DRAM cycles
tXP	4 DRAM cycles
tXPDLL	13 DRAM cycles
Vdd	1.8V

Table 4. Simulated parallel applications and their input sets.

Data Mining		[26]
scalparc	Decision Tree	125k pts., 32 attributes
NAS OpenMP		[6]
mg	Multigrid Solver	Class A
cg	Conjugate Gradient	Class A
SPEC OpenMP		[5]
swim-omp	Shallow water model	MinneSpec-Large
equake-omp	Earthquake model	MinneSpec-Large
art-omp	Self-organizing Map	MinneSpec-Large
Splash-2		[33]
ocean	Ocean movements	514×514 ocean
fft	Fast Fourier transform	1M points
radix	Integer radix sort	2M integers

Table 5. Multiprogrammed Configurations evaluated. C, P, and M stand for Cache-, Processor-, and Memory-sensitive, respectively [6, 13].

Art Mcf Ep Vpr (TFEV)	M M M C
Mg Sp Mesa Vpr (GPMV)	M P P C
Mg Cg Apsi Crafty (GCAY)	M M P C
Apsi Art Mg Swim (ATGS)	P M M M
Mg Cg Mesa Mgrid (GCMD)	M M P M
Art Is Vpr Parser (TIVR)	M M C C
Mg Cg Vpr Swim (GCVS)	M M C M
Cg Mesa Is Crafty (CMIY)	M P M C
Twolf Cg Mesa Is (OCMI)	C M P M
Cg Lu Sp Art (CLPT)	M P P M
Art Is Vpr Wupwise (TIVW)	M M C C
Cg Vpr Wupwise Mesa (CVWM)	M C C P
Twolf Cg Mesa Parser (OCMR)	C M P C
Lu Cg Mesa Is (LCMI)	P M P M

penalty to bring up a rank once it has been powered down is 4-13 cycles.

On the other hand, other aspects of this reward assignment are not so obvious. For example, the specific ratios among the reward values for the different actions are non-intuitive. It is intriguing that, despite the fact that the objective function is straight performance, a PwDn-PwUp action sequence yields a slightly positive aggregate reward (-0.27+0.3), even though powering up a dormant rank incurs a penalty of 4-13 cycles. Note also that NoOp (which competes with PwUp when a rank is powered down) is assigned a definitely positive reward, even though keeping a rank powered down does not directly benefit performance. We will revisit this later.

6.1 Evaluation

Figure 3 shows performance data for all the configurations studied. The proposed MORSE-P scheduler has a speedup of 15.5% (18.4% excluding the three applications used during training) and 7% (8.6%) with respect to FR-FCFS and Ipek, respectively. It would seem that the additional sophistication in picking state attributes and immediate reward values does pay off. We now try to understand how MORSE-P achieves its superiority.

Figure 4 breaks down the expectation of page status for a newly arrived command: *Open-Hit* if the page is already open in the corresponding bank; *Open-Miss* if there's an open page which is not the right one (which will impose a Precharge-Activate sequence to get the right page); and *Closed* if there's no open page at that bank–i.e., the scheduler has preemptively precharged a page, probably in an attempt to save time the next time a page needs to be activated. Evidently, FR-FCFS, which is an open-page algorithm, simply does not expect to find a closed page in steady state. On the other hand, the self-optimizing configurations Ipek and MORSE-P do precharge pages proactively if they predict a long-term benefit. The plot shows that MORSE-P is in fact superior to FR-FCFS and Ipek, in that the expectation of Open-Hit is highest almost universally. This increased hit rate correlates well with bottom-line performance in many applications.

Interestingly, the expectation of finding a bank closed in Ipek is significantly higher than in MORSE-P. It turns out that this is in part a potentially undesirable side effect of İpek et al.'s imposition that a NoOp be allowable only if no legal commands (in particular, Precharge) exist, which was put in place to speed up convergence [17]. This means that Ipek is hardwired to closing pages (instead of doing nothing) when there are no other options available, regardless of the long-term benefit of doing so. In the case of MORSE-P, the scheduler learns by itself that it may "bypass" this restriction, by judiciously exploiting the PwDn/PwUp actions available in the DDR3 interface (i.e., it can force NoOp to be a legal choice indirectly by powering down a rank, effectively making Precharge to any bank within that rank ineligible until the rank is powered back up). As indicated before, the reward values obtained by our GA-based procedure hint at the potential benefit of this subterfuge, as it assigns a slightly positive aggregate reward to PwDn+PwUp, and a definitely positive reward to NoOp.

Thus, it is conceivable that Ipek might also learn to do the same if PwDn/PwUp actions are made available, potentially closing the performance gap with MORSE-P. This is precisely what the Ipek+PwDn/Up configuration in the plots

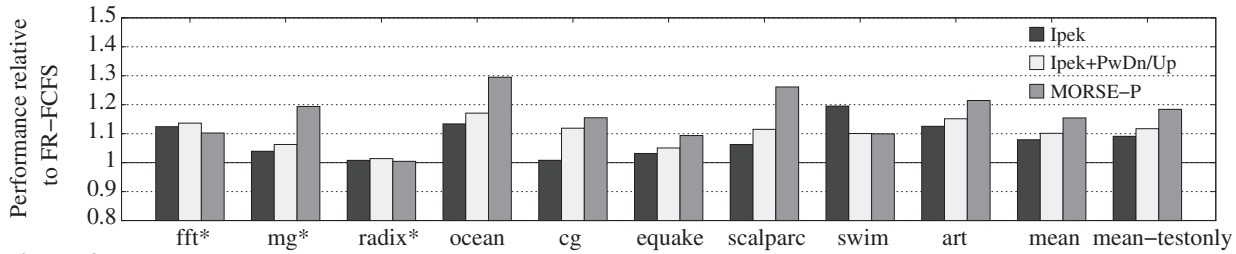

Figure 3. **Performance (higher is better) of Ipek, Ipek+PwDn/Up and MORSE-P, normalized to that of FR-FCFS. The three applications used during training are marked with an asterisk; mean-testonly excludes them.**

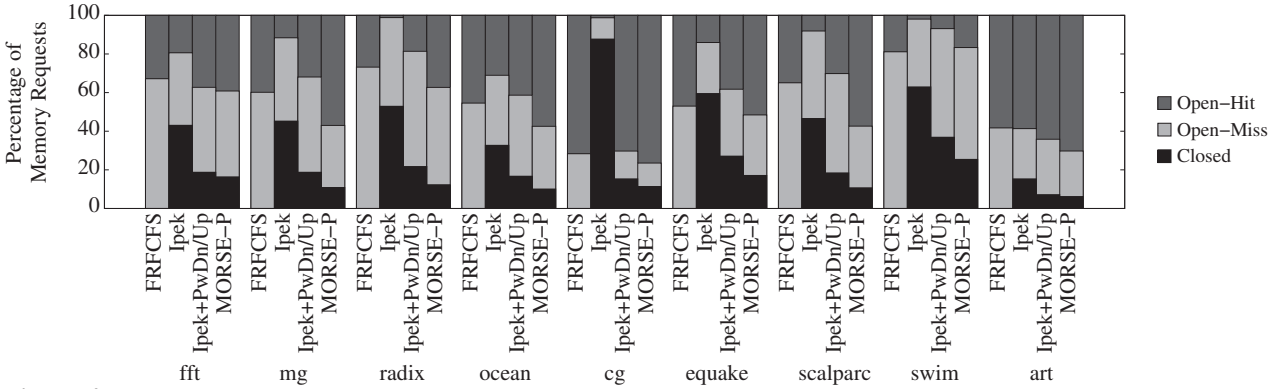

Figure 4. **Breakdown of expected page status at the time a new memory request for that page arrives for FR-FCFS, Ipek, Ipek+PwDn/Up and MORSE-P.**

tries to answer. In that configuration, PwDn/PwUp are available actions with an immediate reward of 0 (consistent with the ad hoc reward function employed), and linear feature selection is re-run. Figures 3 and 4 show the performance and expected page status for the Ipek+PwDn/Up configuration. As we can see, the expectation of finding a bank closed drops dramatically to levels similar to those in MORSE-P. When looking at overall performance, however, not much is gained on average, and a significant gap remains, which means that appropriate state attribute and reward values are, in fact, the primary contributors to performance in MORSE-P vs. Ipek.

Looking at individual applications, however, we see a couple of outliers. In *cg*, *both* Ipek+PwDn/Up and MORSE-P derive noticeable benefit from having PwDn/PwUp available, even if MORSE-P still edges out Ipek+PwDn/Up significantly (Figure 3). *cg* strongly favors an open page policy, as evidenced by the large expectation of a page hit for FR-FCFS (which is open-page). Ipek destroys such potential by being forced by design to close pages prematurely over doing nothing (Figure 4). On the other hand, *swim* seldom hits on open pages for FR-FCFS, and actually performs best with Ipek, where the NoOp restriction is unavoidable and it results in a high number of preemptive precharges. Relaxing this constraint actually hurts Ipek+PwDn/Up and MORSE-P virtually equally with respect to Ipek, although they still outperform full-open-page FR-FCFS significantly. In other words, while these configurations do learn to preemptively close pages and derive speedups as a result, in the case of *swim* they fall short of the level of aggression with which they could apply it, as (accidentally) evidenced by Ipek. (Note that *swim* is not part of our training set. While the obvious temptation is to include it, this would amount to overfitting.)

7 CASE II: ENERGY EFFICIENCY

In this section, we use our general framework to design a self-optimizing memory controller that will strive to optimize for energy-delay-squared (Et^2)–a metric that combines performance and energy consumption. Our evaluation provides quantitative evidence that the resulting controller is significantly superior to a power-aware extension of FR-FCFS that includes Hur and Lin's Queue-Aware Power-Down mechanism [15], which we refer to as *Pwr-FR-FCFS*.[10]

In this section, *MORSE-E* is our energy-efficient self-optimizing scheduler, which is naturally derived using our proposed automatic reward derivation and multi-factor feature selection procedures, using Et^2 as the objective function and the same 50 candidate state attributes used in Section 6.

The state attributes selected via multi-factor feature selection for MORSE-E are:

(1) Number of ranks that are idle and powered up. (2) Number of rows that are open with no pending commands for a given rank. (3) Number of commands that will be negatively affected if a Precharge is issued for the corresponding open

[10]We also experimented with Hur and Lin's AHB [14] and found the results to be very similar.

row (i.e., they would subsequently miss on the page being precharged). (4) Number of writes in the transaction queue. (5) Number of reads for the current rank under consideration. (6) If the memory request is related to a load miss, the order of the load relative to the other loads in the transaction queue for the corresponding core.

Attributes 1 and 2 address power, while the rest are geared primarily towards performance. This is good news, as energy efficiency targets both these metrics. Specifically, those two attributes help the scheduler determine when it is time to power down ranks. Attribute 3 helps maintain row-buffer locality, by determining the number of commands that get affected by a precharge. Attribute 4 helps keep a balance between reads and writes in the transaction queue. Attribute 5 prioritizes among reads from different ranks. Lastly, Attribute 6 is used to prioritize among load misses from the same core.

The rewards obtained from the GA-based automatic reward derivation process are: Activate = -0.21, Precharge = -3.52, Premptive Precharge = -2.06, Read(Load) = 3.71, Read(Store) = 0.86, Write = 1.38, PwDn = -2.06, PwUp = -3.10, NoOp = -1.78. As before, reads and writes always have higher positive reward (better performance) than the other actions. This time, PwUp carries a negative reward–even more so that PwDn. This makes sense, as once the scheduler has decided to power down a rank, it should be because it intends to keep it that way for a while and save energy. Still, as in the case of the straight-performance scheduler, a Precharge is more negative than a PwDn-PwUp sequence, which helps preserve row-buffer locality.

7.1 Evaluation

7.1.1 Energy-Delay Squared

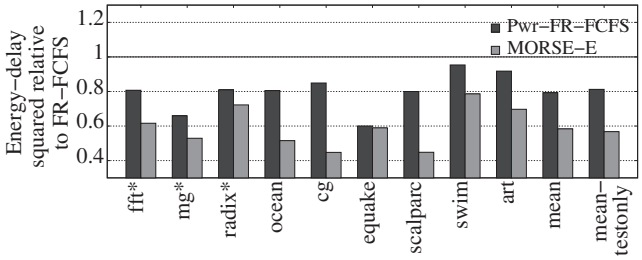

Figure 5. Energy-delay squared Et^2 (lower is better) for the energy-aware configurations considered in this study, normalized to that of FR-FCFS. The three applications used during training are marked with an asterisk; mean-testonly excludes them.

Figure 5 compares the configurations considered in this study in terms of Et^2, normalized to that of FR-FCFS (which is not energy-aware). Our proposed MORSE-E DRAM scheduler reduces Et^2 by 42% (43% excluding the three applications used during training) when comapred to FR-FCFS, and by 26% (30%) when compared to Pwr-FR-FCFS. MORSE-E outperforms Pwr-FR-FCFS by 18% or more for all applications except *radix* (11.0%) and *equake* (2.0%) .

7.1.2 Performance

Figure 6 shows performance data for all the configurations studied. The proposed MORSE-E scheduler has a

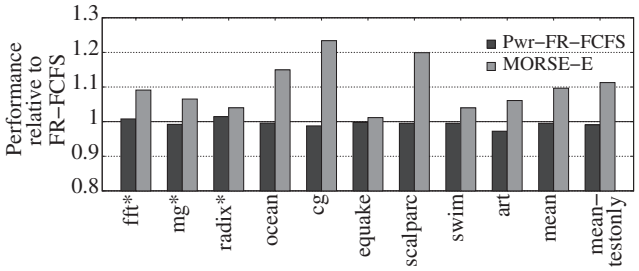

Figure 6. Performance (higher is better) of the energy-aware configurations considered in this study, normalized to that of FR-FCFS. The three applications used during training are marked with an asterisk; mean-testonly excludes them.

speedup of 9.7% (11%) and 10% (11%) with respect to FR-FCFS and Pwr-FR-FCFS respectively. This is very good news: The proposed scheme not only beats Pwr-FR-FCFS handsomely in energy efficiency, it does so while actually delivering a performance gain.

7.1.3 Energy

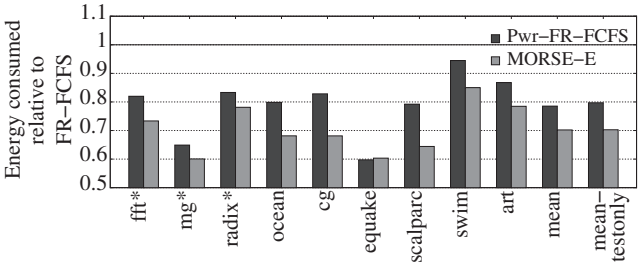

Figure 7. Energy (lower is better) consumed by the energy-aware configurations considered in this study, normalized to that of FR-FCFS. The three applications used during training are marked with an asterisk; mean-testonly excludes them.

Figure 7 shows the energy consumed in executing the various applications. Naturally, the energy-oblivious configuration (FR-FCFS) has the highest energy consumption. Our proposed MORSE-E scheduler yields energy savings of 20% (30% excluding the training apps) and 11% (12%) on average over FR-FCFS and Pwr-FR-FCFS, respectively.

7.2 Analysis

7.2.1 Number of Active Ranks

Figure 8 shows the DRAM transaction queue occupancy and active ranks, averaged over intervals of 5,000 DRAM cycles, for the NAS-OpenMP application *mg* (this behavior is representative of most of the applications studied) using Pwr-FR-FCFS. From the plot, we can see that, throughout the entire execution cycle of the application, there are relatively few instances where the DRAM queue occupancy exceeds the number of active ranks in the memory system. This is consistent with our expectation for high-end servers, where peak demand must be served efficiently to ultimately deliver good performance. Thus, it is extremely important to have an efficient power management scheme that puts idle

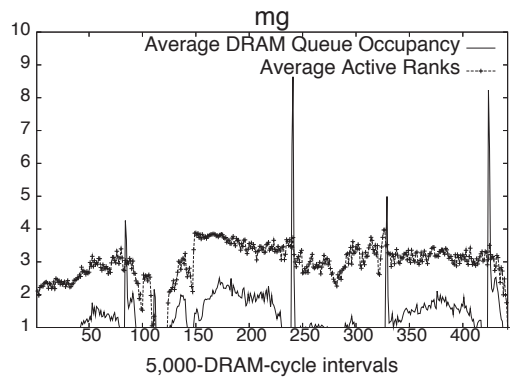

Figure 8. **DRAM transaction queue occupancy and average number of active ranks per channel, averaged over 5,000-DRAM-cycle intervals, for the *mg* application in Pwr-FR-FCFS. (The behavior is representative of the other applications.)**

devices into low-power states and activates them at the right time to avoid significant losses in performance. Figure 9, which plots the average number of active ranks per channel, shows how MORSE-E is able to reduce the average number of active DRAM devices significantly over Pwr-FR-FCFS.

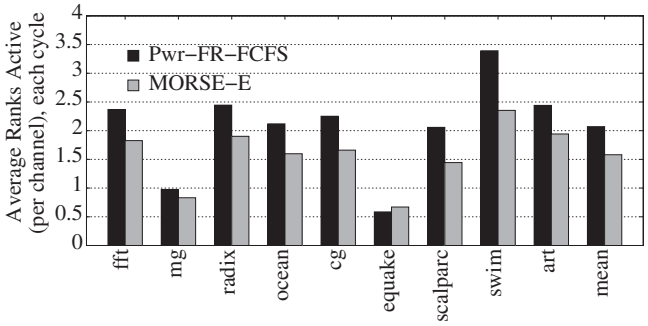

Figure 9. **Average Number of DRAM ranks (per channel) that are powered up (active) each cycle in Pwr-FR-FCFS and MORSE-E.**

7.2.2 Impact of the Selected Attributes on Energy Efficiency

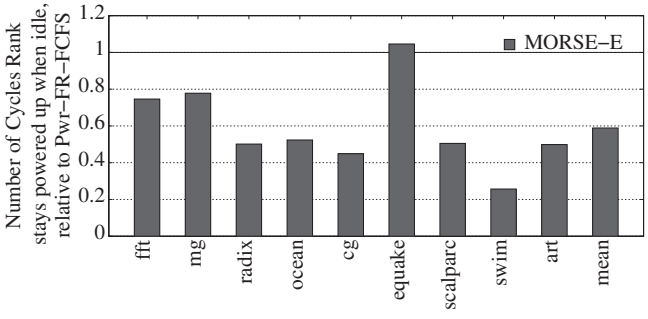

Figure 10. **Number of cycles DRAM ranks stay powered up when no outstanding command exists in the DRAM transaction queue for the corresponding rank for the MORSE-E configuration, normalized to Pwr-FR-FCFS.**

The energy-aware feature selection process in MORSE-E picked attributes that hinted at situations that help improve DRAM background power consumption. In this section, we provide insights into the benefits of picking those attributes, and how they help improve energy efficiency. Figure 10 shows the number of DRAM cycles a rank remains powered up when no outstanding command is present in the transaction queue for MORSE-E and Pwr-FR-FCFS. Recall that one of the states picked by the feature selection process evaluates the ranks that are idle and powered up (Attribute 1). MORSE-E is able to proactively determine when a rank can be placed in low power mode without hurting pending requests. As a result, ranks stay in this mode for longer on average, resulting in greater energy efficiency.

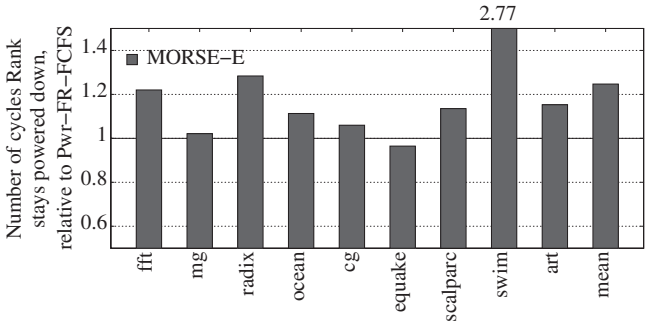

Figure 11. **Number of cycles DRAM ranks stay powered down for the MORSE-E configuration, normalized to Pwr-FR-FCFS.**

Figure 11 shows the number of DRAM cycles a rank remains powered down in MORSE-E and in Pwr-FR-FCFS normalized to that of Pwr-FR-FCFS. Powering up a rank from low power mode takes 4-13 cycles, and hence, if done prematurely, will lead to increase in energy consumption, while if done later, will lead to a loss in performance. The MORSE-E scheduler is able to learn this fine balance based on the state information and the reward function.

7.3 Effect on Multiprogrammed Workloads

We now assess the robustness of this scheduler by evaluating it in a scenario different from the one used in the design phase, namely multiprogrammed workloads.

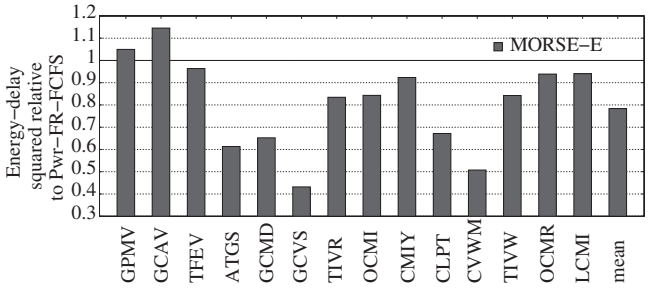

Figure 12. **Energy-delay squared Et^2 (lower is better) for the MORSE-E configuration, normalized to that of Pwr-FR-FCFS.**

Figures 12, 13, and 14 shows Et^2, energy, and performance of MORSE-E relative to that of Pwr-FR-FCFS in each case. (To compute performance and Et^2, we measure

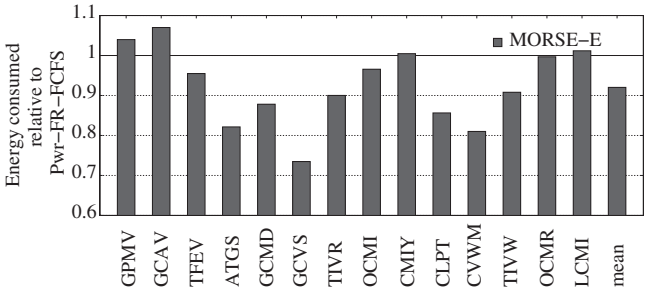

Figure 13. **Energy consumption (lower is better) for the MORSE-E configuration, normalized to that of Pwr-FR-FCFS.**

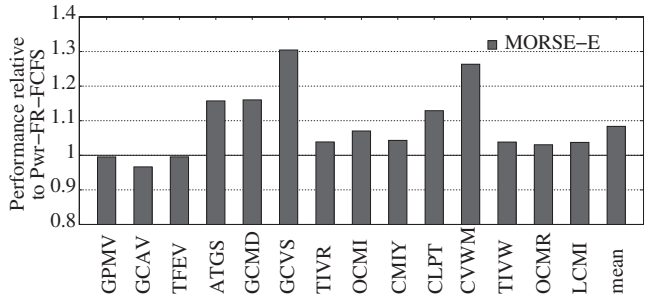

Figure 14. **Performance (higher is better) for the MORSE-E configuration, normalized to that of Pwr-FR-FCFS.**

the time needed in each case to execute *at least* half a billion instructions in *each* of the workloads after warm-up.) The results are very reassuring: Even though MORSE-E has been trained on a very small number of parallel applications and *no* multiprogrammed workloads whatsoever, our methodology has been able to produce a memory scheduler that is relatively robust in this different context. Specifically, MORSE-E improves Et^2 by 22% on average over Pwr-FR-FCFS, which results from simultaneously saving 8% energy on average and delivering a 8.3% average speedup. Only two workloads experience an Et^2 degradation (below 15% in any case), most likely stemming from the fact that MORSE-E is not specifically designed for multiprogrammed workloads.

8 CASE III: THROUGHPUT/FAIRNESS

In this section, we describe a self-optimizing scheduler design that targets system throughput/fairness for multiprogrammed workloads. We use weighted speedup as the objective function. It is the sum of the ratio, for each application, of the IPC obtained in the multiprogrammed scenario over the IPC that the application would enjoy if it were to run alone in the same system [29]. We also conducted experiments using harmonic speedups [22], but obtained virtually identical results.

MORSE-WS is a configuration for which feature selection and GA-based rewards derivation have been conducted, using weighted speedup as the objective function. The two workloads used for training are *GPMV* and *GCAY* (Section 5.2). A key detail to notice is that, while weighted speedup would be complex to observe directly on the field, in our framework it is easy to target at design time through simulations. Once features and rewards have been tuned for that metric, they will "embed" it behaviorally on the field, without ever needing to actually measure it.

We also report the performance of plain MORSE-P, and MORSE-E, which represent cases where no post-silicon changes are possible. Finally, we also evaluate a state-of-the-art scheduler form the literature that also targets weighted speedup (PAR-BS) [25].

Figure 15 shows the weighted speedups obtained for all configurations, relative to FR-FCFS. For our architecture organization and applications (different from those of PAR-BS's original paper [25]), PAR-BS offers negligible benefit over FR-FCFS. MORSE-WS, on the other hand, significantly outperforms PAR-BS. We also notice that MORSE-P matches MORSE-WS's performance, whereas MORSE-E does not. This is maybe not too surprising: Although neither MORSE-P not MORSE-E were designed to target weighted speedup of multiprogrammed workloads, MORSE-P's objective is more closely aligned with MORSE-WS's. MORSE-E's result, on the other hand, is probably representative of a scenario where the fundamental differences between train (-E) and test (-WS) objective functions are more significant, and in that case the bottom line suffers. Still, given the fact that MORSE-P was trained using very different workloads (parallel applications), the results are further testament to MORSE's robustness.

9 RELATED WORK

Hur and Lin [15] propose a simple queue-aware power-down policy for exploiting low-power modes of modern DRAMs. In addition, they also propose the use of a power-aware memory scheduler that encodes several scheduling goals in finite state machines (FSM), and chooses among the FSMs using a probabilistic arbiter. The FSMs encode the ratio of reads and writes serviced in the past, groups same rank commands together and groups commands that optimizes for expected latency. However, one drawback of their power-aware memory scheduler is that it does not consider row precharges, row activations, rank power up and rank power down as separate, individual DRAM commands and hence does not address different trade-offs involved in DRAM scheduling.

Lebeck et al. [20] explore the interaction of page placement policies with the power management techniques used in DRAM systems. Their preliminary experiments using offline profiling of memory accesses show that there is potential in employing page placement policies by an informed operating system to complement the hardware power management strategies.

Fan et al. [11] investigate memory controller policies for manipulating DRAM power states in cache-based systems. They develop an analytical model that approximates the idle time of memory devices, so that they can be powered down and powered up accordingly. However, their model does not sufficiently capture changes to workload demands, and does not learn the long-term performance impact of a scheduling decision, both of which are major benefits of our energy-efficient scheduler.

In other related work, Fan et al. [12] show that there is a positive synergistic effect between DVS and power-aware memories that can transition into low power states, which offers potential for energy savings. Sudan et al. [30] make

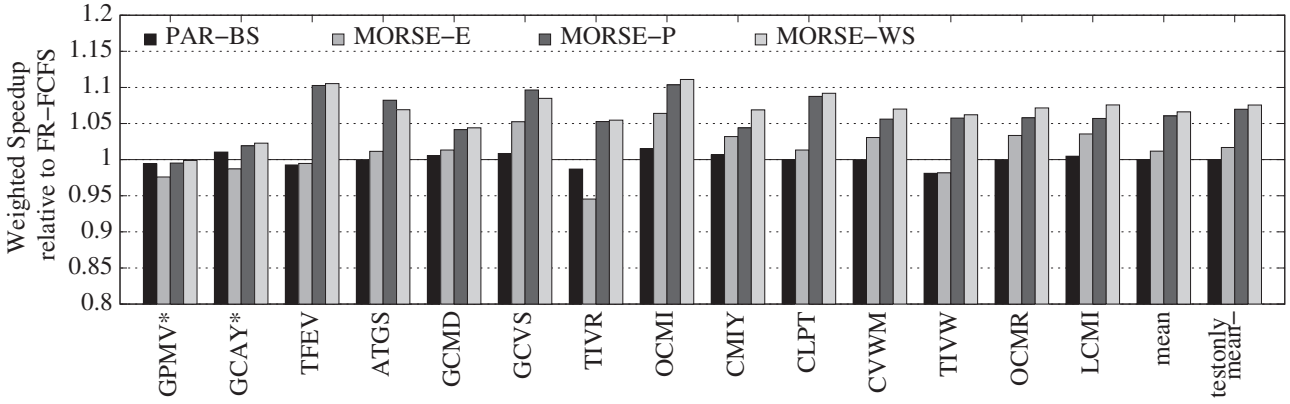

Figure 15. **Weighted speedup (higher is better), normalized to that of FR-FCFS. The two workloads used during training for MORSE-WS are marked with an asterisk; mean-testonly excludes them.**

an interesting observation that a large number of memory accesses mechanisms to heavily accessed OS pages are to a small chunk of contiguous cache blocks. Thus co-locating these chunks from different pages will improve row buffer locality, and energy consumption.

10 CONCLUSIONS

We have proposed the use of genetic algorithms as a general way to systematically derive reward functions for RL-based DRAM schedulers. We have shown that this mechanism is not only capable of targeting arbitrary objective functions, it yields higher-performing schedulers than previously-proposed self-optimizing solutions that employed an ad hoc reward structure. In the process, we have also proposed a *multi-factor feature selection* procedure for designing self-optimizing schedulers that takes into account first-order interactions among RL state attributes. We use this general mechanism to present three memory scheduler designs that target performance, energy efficiency, and throughput/fairness, respectively. Our results significantly outperform the state of the art in the literature in each case. We also show evidence that our designs are robust across workload classes and objective functions when train and test objective functions are similar enough in nature (e.g., both performance-oriented metrics).

ACKNOWLEDGMENTS

We thank the anonymous reviewers for useful feedback. This work was supported in part by NSF awards CNS-0720773 and CAREER CCF-0545995, an Intel contract, and gifts by IBM, Intel, and Microsoft.

References

[1] Cacti 6.5. `http://quid.hpl.hp.com:9081/cacti/`.
[2] 2gb ddr3 sdram component data sheet: Mt41j512m4, March 2006. `http://download.micron.com/pdf/datasheets/dram/ddr3/2Gb_DDR3_SDRAM.pdf`.
[3] Technical note tn-41-01: Calculating memory system power for ddr3, June 2006. `http://download.micron.com/pdf/technotes/ddr3/TN4101pdf`.
[4] Ddr3 sdram rdimm features, March 2009. `http://download.micron.com/pdf/datasheets/modules/ddr3/js-z-s72clg_72.pdf`.
[5] V. Aslot and R. Eigenmann. Quantitative performance analysis of the SPEC OMPM2001 benchmarks. *Scientific Programming*, 11(2):105–124, 2003.
[6] D. H. Bailey et al. NAS parallel benchmarks. Technical report, NASA Ames Research Center, March 1994. Tech. Rep. RNR-94-007.
[7] V. Cuppu and B. Jacob. Concurrency, latency, or system overhead: Which has the largest impact on uniprocessor dram-system performance? In *ISCA*, 2001.
[8] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A performance comparison of contemporary dram architectures. In *ISCA*, 1999.
[9] K. Diefendorff. Sony's emotionally charged chip: Killer floating-point 'emotion engine' to power playstation 2000. *Microprocessor Report*, 13(5):1–11, 1999.
[10] M. Eiblmaier, R. Mao, and X. Wang. Power management for main memory with access latency control. In *FeBID*, 2009.
[11] X. Fan, C. Ellis, and A. R. Lebeck. Memory controller policies for dram power management. In *ISPLED*, 2001.
[12] X. Fan, C. Ellis, and A. R. Lebeck. The synergy between power-aware memory systems and processor voltage scaling. In *PACS*, 2003.
[13] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.
[14] I. Hur and C. Lin. Adaptive history-based memory schedulers. In *MICRO-37*, 2004.
[15] I. Hur and C. Lin. A comprehensive approach to dram power management. In *HPCA*, 2008.
[16] Intel Corporation. First the Tick, Now the Tock: Next-Generation Intel Microarchitecture (Nehalem). `http://www.intel.com/technology/architecture-silicon/next-gen/whitepaper.pdf`.
[17] E. Ipek, O. Mutlu, J. F. Martinez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA*, 2008.
[18] F. Kashfi and S. M. Fakhraie. Implementation of a high speed low-power 32 bit adder in 70nm technology. In *ISCAS*, 2006.
[19] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.
[20] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power aware page allocation. In *ASPLOS-IX*, 2000.
[21] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller. Energy management for commercial servers. *Computer*, 36(12):39–48, 2003.
[22] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in smt processors. In *ISPASS*, 2001.
[23] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, 1996.
[24] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO-40*, 2007.
[25] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *ISCA-35*, 2008.
[26] 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.
[27] 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.
[28] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA-27*, 2000.
[29] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *ASPLOS*, 2000.
[30] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis. Micro-pages: Increasing dram efficiency with locality-aware data placement. In *ASPLOS*, 2010.
[31] R. Sutton. Generalization in reinforcement learning. successful examples using sparse coarse coding. In *Neural Information Processing Systems Conference*, 1996.
[32] R. Sutton and A. Barto. *Reinforcement Learning*. MIT Press, Cambridge, MA, 1998.
[33] 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.
[34] Z. Zhu and Z. Zhang. A performance comparison of dram memory system optimizations for smt processors. In *HPCA-11*, 2005.