# EASE: Energy-Aware Self-Optimizing DRAM Scheduling

Janani Mukundan and José F. Martínez

Computer Systems Laboratory
Cornell University
Ithaca, NY 14853 USA

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

## ABSTRACT

*We propose an energy-aware self-optimizing memory controller in which DRAM energy management is integral to the command scheduler. Experiments conducted on a 8-core CMP model show that, for the parallel applications considered, our scheduler reduces memory's energy-delay squared $Et^2$ by 18% while delivering a 5% speedup with respect to a state-of-the-art power-aware solution.*

## 1 INTRODUCTION

Modern high-performance memory subsystems exhibit a high degree of concurrency. This is primarily accomplished by increasing the number of independent channels and/or increasing number of independent banks in a channel [4, 9]. Partly as a result, in current and upcoming multicore-based servers, DRAM accounts for a significant fraction of power consumption–about 40% of the total system for large configurations [5, 14].

Modern DRAMs have the capability of placing ranks in low-power mode when not in use [2]. Several techniques that try to exploit this feature have been proposed, from cooperative hardware/software approaches to page placement [13], to opportunistic use of low-power modes [5, 8]. There are, however, important potential limitations of prior proposals: (1) they are relatively unsophisticated ad hoc heuristics, based almost exclusively on a human expert's intuition; (2) they are largely add-ons to existing memory scheduling policies; and (3) for the most part, they are static policies. (We comment on related work in Section 5.)

Recently, İpek et al. [10] propose the use of reinforcement learning (RL) [20] 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–memory throughput in their case. İpek et al. show that this approach can outperform existing ad hoc designs by a significant margin.

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

Presented at Workshop on Energy Efficient Design (WEED), conc. with ISCA, June 2011

## Contributions

In this work, for the first time, **we propose a *systematic* and *general* mechanism to target arbitrary objective functions** when designing self-optimizing DRAM schedulers (Section 3.3). Using this general approach, **we propose a class of self-optimizing memory controllers which incorporates energy efficiency in the scheduler's decision-making process**. In the context of our evaluation, our energy-aware scheduler reduces the memory energy-delay squared product ($Et^2$) by 18%, while delivering a 5% speedup, with respect to a state-of-the-art power-aware memory controller based on Hur and Lin's Queue-Aware Power Down mechanism [8]. In the process, we also propose a *multi-factor feature selection* procedure for designing self-optimizing schedulers that takes into account first-order interactions among RL state attributes.

## 2 BACKGROUND

### 2.1 Power-Aware DRAM Interfaces

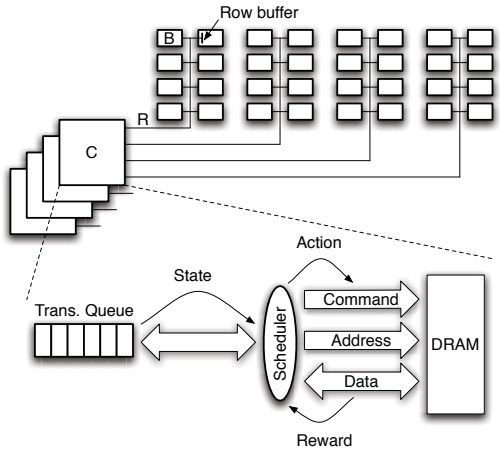


Figure 1: Basic DRAM Interface, with four independent channels (C), one quad-ranked DIMM per channel (R), and eight internal banks (B) per rank.

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

Current DDR3 SDRAMs have the capability of placing a rank in low-power mode; it may take as little as 1 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 6-14 DRAM cycles [2]. Section 4.1 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 [20]. 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 CMP and the memory subsystem constitute the environment, and the legal DRAM commands at each point in time are the actions that the RL agent can perform. 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 $\pi$, $Q_\pi(s, a)$, is the expected cumulative reward resulting from taking action $a$ in state $s$ and following policy $\pi$ thereafter. A Q-learning-based RL agent learns the optimal policy $\pi^*$ indirectly, by learning $Q_{\pi^*}(s, a)$ for every state-action pair $(s, a)$ (the *Q-value matrix*).

[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.

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 PROPOSED ARCHITECTURE: EASE

We describe the three main characteristics of our RL-based Energy-Aware SElf-optimizing DRAM scheduler: actions, state attributes, and reward structure.

## 3.1 Available Actions

Concurrently to sensing the environment's state (Section 3.2), the scheduler determines whether a valid DRAM command exists for each pending memory request among:
(1) **Activate:** Bring the contents of a bank's DRAM row into the bank's row buffer.
(2) **Precharge:** Write the contents of a bank's row buffer back to the corresponding DRAM row.
(3) **Read(Load), Read(Store):** Perform a read from a bank's row buffer.
(4) **Write:** Perform a write to a bank's row buffer.
(5) **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].
(6) **Rank Power Up (PwUp):** Bring the corresponding rank back to normal operation mode.
(7) **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. (To avoid starvation and to speed up convergence, the scheduler is disallowed from choosing NoOp when legal DRAM commands for pending memory requests exist.)

## 3.2 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. Typically, the memory system environment can be represented using hundreds of different state attributes. However, it is not feasible to use all of them for online training of the RL agent ("curse of dimensionality"), and hence it is necessary to have a good selection mechanism that picks the right set of state attributes from the hundreds that are available.

### Multi-factor Feature Selection

A quick and relatively simple way to accomplish this is to use a linear feature selection process [10]. 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.

Because EASE targets energy efficiency, we use $Et^2$ as our objective function. 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 the number of simulations increases exponentially with each iteration. [2] Fortunately, feature selection is a one-time effort made at design time.[3] The resulting attributes will determine the state for each of the queued memory requests considered. They are:

(1) Number of reads in the transaction queue. (2) Number of reads for the current rank under consideration. (3) Number of writes in the transaction queue that reference rows that are open. (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 cycles a rank is powered up when it has no commands in the transaction queue. (6) Number of commands for the rank under consideration when it is powered down.

Attributes 1-4 relate chiefly to performance, while attribute 5 and 6 deal primarily with energy consumption. Specifically, (1) prioritizes reads over writes; (2) prioritizes among reads from different ranks ; (3) helps the RL scheduler issue writes in bursts so as to better manage write-to-read delays ; and (4) is used to prioritize among load misses from the same core. Attribute (5) helps decide when it is time to power down idle ranks; and finally (6) determines if it is time to power up ranks if there are memory requests waiting to access the rank.

## 3.3 Reward Structure

In order to explore the environment, the scheduler implements an exploration mechanism known as $\epsilon$-greedy action selection: Every DRAM cycle, with a small probability $\epsilon$, 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 [20] 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 $\alpha$ is the *learning rate*, empirically determined;[4] $r_t$ 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.[5]

The performance-oriented RL scheduler proposed by İpek et al. [10] trivially assigned rewards that learned to maximize long-term data bus utilization. As memory throughput (and ultimately performance) strongly correlates to data bus utilization, the authors assign a reward of +1 for any DRAM command that improves data bus utilization (reads and writes) and 0 for the other commands. Unfortunately, this approach does not easily generalize: In a design that seeks to optimize a more sophisticated function (e.g., $Et^2$ in our case), 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 [15] to explore the search space of correlating rewards for actions in terms of both energy and performance. Genetic algorithms (GAs) are a heuristic search technique based on evolutionary processes. GAs start by randomly generating a population space of individuals, where each individual is a candidate solution for the problem being solved. Evolution is performed in generations, and usually starts by evaluating the fitness of the initial population using a performance-based criteria. 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. Individuals are further evolved using operations like crossover and mutation. 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.

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 our two fastest applications, *fft* and *mg*,[6] and determining the fitness ($Et^2$) of each individual. The fitness-based selection criteria that we use is *tournament selection* combined with *elitist selection* [15, 12]. 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 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 55 generations, at the end of which we are left with a set of rewards, one per possible action, which together constitute our reward function.

The reward function obtained in this way is: Activate = 1.59, Precharge = -1.47, Read(Load) = +2.00, Read(Store) = +1.59, Write = +0.88, PwDn = -0.27, PwUp = +0.30, NoOp = +0.58.[7] The resulting values make intuitive sense: For example, reads and writes have a high positive reward (better

[2] The applications that we use for feature selection are *fft, mg*, and *radix*. We picked these because they are the fastest to simulate.

[3] We were able to complete all 8,600 simulations in one day.

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

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

[6] As in the case of multi-factor feature selection, by searching on a small subset of our application set which are *not* picked on the basis of aptitude, we greatly speed up the process and at the same time minimize the chance of overfitting the final solution.

[7] We arbitrarily set up the reward structure to use higher (lower) values as

performance) than the other actions; PwDn is negative while PwUp is positive, as the penalty to power up a rank once it has been powered down is 4-13 cycles and hence PwDn needs to be used more judiciously; and Precharge is negative while Activate is positive, allowing for higher row buffer locality and hence performance. On the other hand, the specific ratios among the reward values for the different actions are non-intuitive. And in any case, the connection between the reward function and the objective function $Et^2$ is not at all obvious.

## 3.4 Implementation

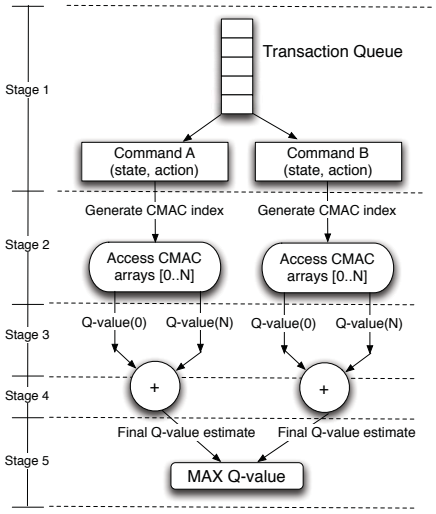


Figure 2: The Q-value estimation pipeline for two candidate commands in the DRAM transaction queue.

The basic structure of our implementation is similar to the one described by İpek et al. [10]. Figure 2 shows the five-stage pipeline structure that is used to calculate the Q-values of the proposed scheduler. This pipeline structure scans the pending memory requests and calculates two Q-values every processor cycle. In the first stage of the pipeline, the scheduler determines the DRAM commands for two memory requests and generates the corresponding state-action pairs. (Information about the state attributes are obtained in the previous DRAM cycle from the DRAM transaction queue.) In the second pipeline stage, the indices for the Q-value tables 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. Next, the XOR-ed value is passed through a hash function (to reduce storage requirements) to get the index into the Q-value tables.

In the next two pipeline stages, the Q-values of the DRAM commands are read. (The Q-value tables follow a CMAC organization, where each Q-value is in fact the result of indexing multiple 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 [10, 19]. In our design, we carefully account for the delay and power incurred by these multiple tables.)

positive (negative) rewards, which is typical in machine learning texts.

In the fifth and final stage of the pipeline the two Q-values coming out of the pipeline are compared with the maximum Q-value seen so far, and any necessary updates are made to the maximum Q-value.

For a DRAM scheduler pipeline clocked at 4.27 GHz (same as the CPU)[8] controlling a DDR3-1066 system, the Q-value estimation pipeline can be clocked eight times every DRAM cycle. Therefore our scheduler can consider a minimum of eight DRAM commands for scheduling every DRAM cycle. (If there are more memory requests on the queue, the scheduler simply takes the DRAM command with the highest Q-value found by the end of the DRAM cycle. Alternatively, more memory requests can be considered per cycle if the width of the pipeline is increased.)

The Q-values are stored in 32 matrices per memory request. These are SRAM tables, each with 256 entries, one read port and one write port. The updates to the Q-values are done using fixed-point arithmetic. Since our power-aware scheduler introduces two new DRAM actions (PwUp/PwDn), the table storing the rewards for the various actions taken is increased by two when compared to the implementation in İpek et al. [10].

# 4 EVALUATION

## 4.1 Experimental Methodology

We evaluate our proposal by using the configurations shown in Table 5. We use Rixner et al.'s *FR-FCFS* memory scheduler as the base design [17, 18], which prioritizes CAS over RAS, older over newer loads, and it is energy-oblivious. The *Pwr-FR-FCFS* configuration couples the FR-FCFS algorithm with the state-of-the-art Queue-Aware Power-Down mechanism proposed by Hur and Lin [8].[9] *Ipek* is İpek et al.'s original performance-oriented RL-based scheduler [10]. We do re-run feature selection to accommodate our server DDR3 setup (vs. İpek et al.'s original desktop DDR2 setup). Finally, *EASE* is our Energy-Aware SElf-optimizing scheduler, which includes the DDR3-based power management scheme (which adds the two new actions, PwDn/PwUp), and is obtained using our proposed reward derivation and multi-factor feature selection procedures, targeting memory's $Et^2$.

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 Micron's DDR3 DRAM specification [2, 3], including refresh.

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 *do* include this overhead. Moreover, in our results we effectively assess zero energy overhead for the competing FR-FCFS and Pwr-FR-FCFS schemes.

**Dynamic power – Q-value computation:** The RL pipeline structure used for computing Q-values includes 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. We use CACTI 5.3 [1] to estimate the energy expended in reading out the Q-values from the SRAM

[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.

[9]We also experimented with Hur and Lin's AHB [7] and found the results to be very similar.

arrays. We assume 32 nm technology for our calculations. Each SRAM read consumes 0.2 pJ, and the total dynamic SRAM energy for reading out the Q-values from the 32 matrices per command is 6.4 pJ. We estimate the power consumed by the adders that sum up the 32 Q-values to be 0.97 mW each [11], and accordingly calculate the energy consumed by the 31 adders used in the RL pipeline to be 14.4 pJ (adding the 32 Q-values takes up two pipeline cycles). Conservatively assuming that the comparator consumes the same power as the adder, we determine the energy of the comparator to be 0.9 pJ. To generate the 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 each be approximated as a dynamic SRAM read and consumes 1.4 pJ per read. The XOR function takes 0.07 pJ (an XOR function is again conservatively approximated to consume the same power as an adder). The total RL pipeline energy consumed is then 23 pJ per cycle.

**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 0.8 pJ [11]. Finally the Q-values need to be written back into the SRAM arrays (32 per command, 64 in all). This consumes 12.8 pJ as estimated from CACTI 5.3.

**Leakage:** Using CACTI 5.3, we also estimate the total leakage power per CMAC matrix to be 0.08 mW, and consequently the leakage energy to be 0.15 pJ per DRAM cycle.

The micro-architectural features of the baseline processor is 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 using a heavily-modified cycle-accurate version of the SESC simulation environment [16].

We simulate nine memory-intensive parallel applications from different domains, running eight threads each, as shown in Table 4. We execute all applications to completion, and use whole-program execution time and total DRAM energy consumption (plus DRAM scheduler's energy overhead in the case of EASE) in all cases.

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 |
| **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 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 |
| **Write buffer** | 64 entries |
| **Micron 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 |
| **RL discount rate parameter $\gamma$** | 0.95 |
| **RL stochastic action selection parameter $\epsilon$** | 0.05 |
| **RL learning rate parameter $\alpha$** | 0.1 |

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

| | |
|---|---|
| **IDD0** | 100 mA |
| **IDD3PF** | 35 mA |
| **IDD3PS** | 35 mA |
| **IDD2PF** | 35 mA |
| **IDD2PS** | 12 mA |
| **IDD2N** | 55 mA |
| **IDD3N** | 57 mA |
| **IDD4R** | 160 mA |
| **IDD4W** | 190 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 applications and their input sets.

| | | |
|---|---|---|
| **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 5: Configurations evaluated.

| | |
|---|---|
| **FR-FCFS** | Rixner and Dally's performance-oriented scheduler [18] |
| **Pwr-FR-FCFS** | Rixner and Dally's scheduler [18] + Hur and Lin's Queue-Aware Power Down mechanism [8] |
| **Ipek** | İpek et al.'s performance-oriented scheduler [10] |
| **EASE** | Energy-Aware Self-Optimizing DRAM Scheduler |

## 4.2 Results

### 4.2.1 Energy-Delay Squared

Figure 3 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 EASE DRAM scheduler reduces $Et^2$ by 31% when comapred to FR-FCFS and by 18% when compared to Pwr-FR-FCFS. EASE is significantly superior to Pwr-FR-FCFS (improvement of 10% or more) in all but two applications: *cg*, and *radix*. Upon further analysis of the *cg* application, we found that the increase in $Et^2$ (over Pwr-FR-FCFS) was because EASE suffers from premature row precharges, as well as premature rank shutdown. However, for *radix* we find that EASE is only marginally better than Pwr-FR-FCFS because of delayed shutdown of ranks. Recall that our mechanism is designed with a training set that

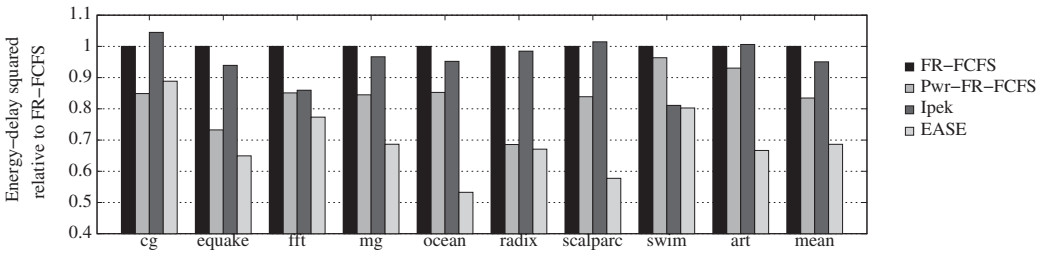


Figure 3: Energy-delay squared $Et^2$ (lower is better) for the configurations considered in this study, normalized to that of FR-FCFS.

is a small subset of the final test set; the majority of applications, among them *cg* and *radix*, are not used during training rewards. The obvious temptation, which is to include *cg* and *radix* in our training set, would essentially amount to overfitting the result. Finally, Ipek hardly edges out FR-FCFS in terms of $Et^2$, and loses badly to all energy-aware configurations.

### 4.2.2 Performance

Figure 4 shows performance data for all the configurations studied. The proposed EASE scheduler has a speedup of 5% with respect to Pwr-FR-FCFS. This is very good news: The proposed scheme not only beats Pwr-FR-FCFS handsomely in energy savings, it does so while actually delivering a performance gain. Ipek outperforms FR-FCFS as in the original paper, but in this server DDR3 configuration it does so much more modestly. Interestingly, EASE delivers a speedup even with respect to Ipek. This means that EASE, on top of leveraging its energy awareness, actually delivers a better mechanism from the performance standpoint, due to its automatic offline configuration process.

### 4.2.3 Energy

Figure 5 shows the energy consumed in executing the various applications. Naturally, the energy-oblivious configurations have the highest energy consumption. Our proposed EASE scheduler yields energy savings of 25% and 10% on average over FR-FCFS and Pwr-FR-FCFS, respectively.

### Analysis

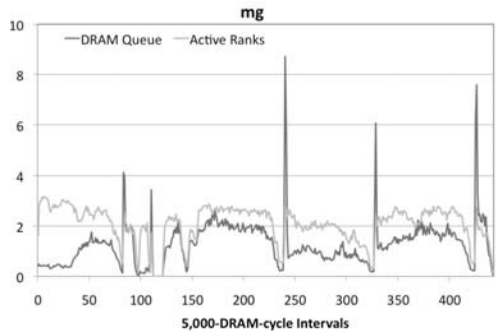


Figure 6: DRAM transaction queue occupancy and average number of active ranks per channel, averaged over 5,000-DRAM-cycle intervals, for the *mg* application in Pwr-FR-FCFS. (The behavior is representative of the other applications.)

Figure 6 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). 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 devices into low-power states and activates them at the right time to avoid significant losses in performance. Figure 7, which plots the average number of active ranks per channel, shows how EASE is able to reduce the number of active DRAM devices every cycle by about 33% on average over Pwr-FR-FCFS.

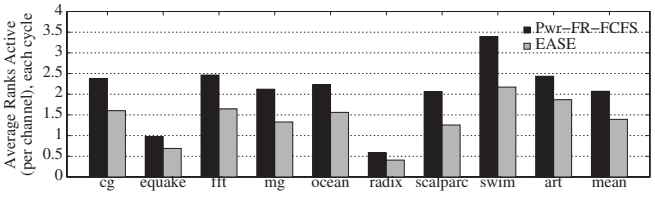


Figure 7: Average Number of DRAM ranks (per channel) that are powered up (active) each cycle in Pwr-FR-FCFS and EASE.

## 5 RELATED WORK

Hur and Lin [8] propose a simple 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 the 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. We incorporated the adaptive-history-based memory scheduler in our simulation environment and found that it did not provide significant benefits over the FR-FCFS scheduler coupled with the queue aware power down policy.

Lebeck et al. [13] 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

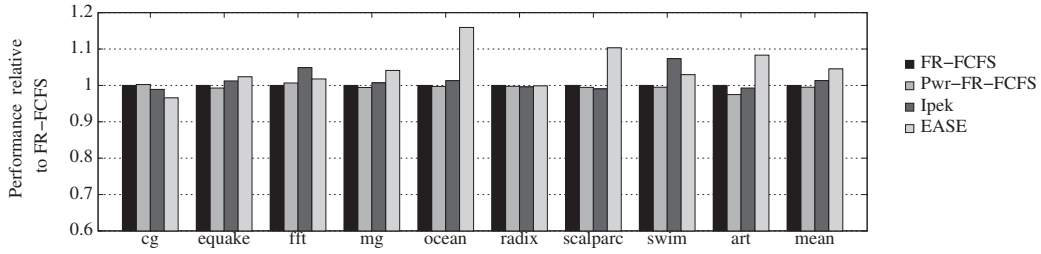


Figure 4: Performance (higher is better) of the configurations considered in this study, normalized to that of FR-FCFS.

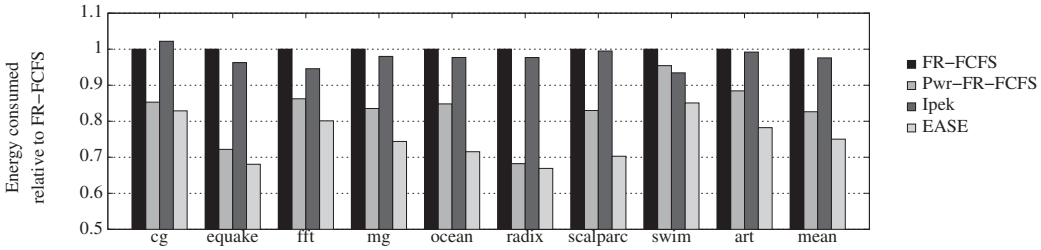


Figure 5: Energy (lower is better) consumed by the configurations considered in this study, normalized to that of FR-FCFS.

employing page placement policies by an informed operating system to complement the hardware power management strategies.

Fan et al. [6] 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.

## 6 CONCLUSIONS

We have proposed a self-optimizing energy-aware DRAM scheduler. On a 8-core CMP coupled with a contemporary DDR3-1066 memory subsystem, our proposed scheduler reduces memory's energy-delay squared $Et^2$ by 18% with improved performance (a 5% speedup) when compared to a state-of-the-art power-aware memory controller while running a diverse set of parallel applications. 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 capable of targeting arbitrary objective functions. 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.

## REFERENCES


[1] CACTI 5.3. http://quid.hpl.hp.com:9081/cacti/.

[2] 1 GB DDR3 SDRAM component data sheet: Mt41j128m8, March 2006. http://download.micron.com/pdf/datasheets/dram/ddr3/1Gb_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] V. Cuppu and B. Jacob. Concurrency, latency, or system overhead: Which has the largest impact on uniprocessor DRAM-system performance? In *ISCA*, 2001.

[5] M. Eiblmaier, R. Mao, and X. Wang. Power management for main memory with access latency control. In *FeBID*, 2009.

[6] X. Fan, C. Ellis, and A. R. Lebeck. Memory controller policies for DRAM power management. In *ISPLED*, 2001.

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

[8] I. Hur and C. Lin. A comprehensive approach to DRAM power management. In *HPCA*, 2008.

[9] Intel Corporation. First the Tick, Now the Tock: Next-Generation Intel Microarchitecture (Nehalem). http://www.intel.com/technology/architecture-silicon/next-gen/whitepaper.pdf.

[10] E. İpek, O. Mutlu, J. F. Martínez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA*, 2008.

[11] F. Kashfi and S. M. Fakhraie. Implementation of a high speed low-power 32 bit adder in 70 nm technology. In *ISCAS*, 2006.

[12] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

[13] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power aware page allocation. In *ASPLOS*, 2000.

[14] 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.

[15] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, 1996.

[16] 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.

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

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

[19] R. Sutton. Generalization in reinforcement learning. successful examples using sparse coarse coding. In *NIPS*, 1996.

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