Improving Fairness in Memory Scheduling
Using a Team of Learning Automata

Aditya Kajwe and Madhu Mutyam

Department of Computer Science & Engineering,
Indian Institute of Technology - Madras

June 14, 2014
Outline

1. Introduction
2. Related Work
3. Our Learning Automata-based Algorithm
4. Experiments
5. Conclusion
DRAM scheduling

- The order in which memory access requests from the CPU are processed at DRAM.
Introduction

DRAM scheduling
- The order in which memory access requests from the CPU are processed at DRAM.
- Impacts main memory fairness, throughput & power consumption.
Introduction

DRAM scheduling
- The order in which memory access requests from the CPU are processed at DRAM.
- Impacts main memory fairness, throughput & power consumption.

Metrics for evaluating a scheduling algorithm
- harmonic speedup, execution time, sum-of-IPCs, maximum slowdown, weighted speedup
Introduction

DRAM scheduling
- The order in which memory access requests from the CPU are processed at DRAM.
- Impacts main memory fairness, throughput & power consumption.

Metrics for evaluating a scheduling algorithm
- harmonic speedup, execution time, sum-of-IPCs, maximum slowdown, weighted speedup
- harmonic speedup $= \frac{N}{\sum \frac{IPC_{i^{alone}}}{IPC_{i^{shared}}}}$
DRAM scheduling
- The order in which memory access requests from the CPU are processed at DRAM.
- Impacts main memory fairness, throughput & power consumption.

Metrics for evaluating a scheduling algorithm
- harmonic speedup, execution time, sum-of-IPCs, maximum slowdown, weighted speedup
- harmonic speedup = \( \frac{N}{\sum_i \frac{IPC_{\text{alone}}}{IPC_{\text{shared}}}} \)
- Provides a good balance between fairness and system performance
- ATLAS [2]: prioritizes threads that have attained the least service
- ATLAS [2]: prioritizes threads that have attained the least service
- PAR-BS [5]: processes DRAM requests in batches, and uses the SJF principle within a batch
- **ATLAS [2]**: prioritizes threads that have attained the least service.
- **PAR-BS [5]**: processes DRAM requests in batches, and uses the SJF principle within a batch.
- **MORSE [4]**: extends Ipek et.al’s learning technique [1] to target arbitrary figures of merit.
- **MISE [6]**: estimates slowdown of each application and accordingly redistributes bandwidth.
Related Work

- ATLAS [2]: prioritizes threads that have attained the least service
- PAR-BS [5]: processes DRAM requests in batches, and uses the SJF principle within a batch
- MORSE [4]: extends Ipek et.al’s learning technique [1] to target arbitrary figures of merit.
- MISE [6]: estimates slowdown of each application and accordingly redistributes bandwidth
Related Work

- ATLAS [2]: prioritizes threads that have attained the least service
- PAR-BS [5]: processes DRAM requests in batches, and uses the SJF principle within a batch
- MORSE [4]: extends Ipek et.al’s learning technique [1] to target arbitrary figures of merit.
- MISE [6]: estimates slowdown of each application and accordingly redistributes bandwidth

Thread Cluster Memory Scheduling (TCMS) [3]
- divides threads into two clusters
Related Work

- ATLAS [2]: prioritizes threads that have attained the least service
- PAR-BS [5]: processes DRAM requests in batches, and uses the SJF principle within a batch
- MORSE [4]: extends Ipek et.al’s learning technique [1] to target arbitrary figures of merit.
- MISE [6]: estimates slowdown of each application and accordingly redistributes bandwidth

Thread Cluster Memory Scheduling (TCMS) [3]
- divides threads into two clusters
- latency-sensitive cluster > bandwidth-sensitive cluster
Related Work

- ATLAS [2]: prioritizes threads that have attained the least service
- PAR-BS [5]: processes DRAM requests in batches, and uses the SJF principle within a batch
- MORSE [4]: extends Ipek et.al’s learning technique [1] to target arbitrary figures of merit.
- MISE [6]: estimates slowdown of each application and accordingly redistributes bandwidth

Thread Cluster Memory Scheduling (TCMS) [3]

- divides threads into two clusters
- latency-sensitive cluster > bandwidth-sensitive cluster
- periodically shuffles priority in the bandwidth cluster
Overview of a Learning Automaton (LA)

A simple model for dynamic decision making in unknown environments.
Overview of a Learning Automaton (LA)

A simple model for dynamic decision making in unknown environments.

Structure of FALA (Finite Action Learning Automaton)

Formally, a FALA can be described by the quadruple \((A, B, \tau, p(k))\):

- \(A = \{\alpha_1, \alpha_2, \ldots, \alpha_r\}\): finite set of actions.
- \(B\): set of all possible reinforcements
- \(\tau\): learning algorithm to update \(p(k)\)
- \(p(k) = [p_1(k), p_2(k), \ldots, p_r(k)]\)

Higher the probability value for a thread, higher is its priority for DRAM scheduling.
Overview of a Learning Automaton (LA)

A simple model for dynamic decision making in unknown environments.

Structure of FALA (Finite Action Learning Automaton)

Formally, a FALA can be described by the quadruple \((A, B, \tau, p(k))\) :

- \(A = \{\alpha_1, \alpha_2, \ldots, \alpha_r\}\) : finite set of actions.
Overview of a Learning Automaton (LA)

A simple model for dynamic decision making in unknown environments.

Structure of FALA (Finite Action Learning Automaton)

Formally, a FALA can be described by the quadruple \((A, B, \tau, p(k))\):

- \(A = \{\alpha_1, \alpha_2, \ldots, \alpha_r\}\): finite set of actions.
- \(B\): set of all possible reinforcements.
Overview of a Learning Automaton (LA)

A simple model for dynamic decision making in unknown environments.

Structure of FALA (Finite Action Learning Automaton)

Formally, a FALA can be described by the quadruple \((A, B, \tau, p(k))\):

- \(A = \{\alpha_1, \alpha_2, \ldots, \alpha_r\}\) : finite set of actions.
- \(B\) : set of all possible reinforcements
- \(\tau\) : learning algorithm to update \(p(k)\)
Overview of a Learning Automaton (LA)

A simple model for dynamic decision making in unknown environments.

Structure of FALA (Finite Action Learning Automaton)

Formally, a FALA can be described by the quadruple \((A, B, \tau, p(k))\):

- \(A = \{\alpha_1, \alpha_2, \ldots, \alpha_r\}\) : finite set of actions.
- \(B\) : set of all possible reinforcements
- \(\tau\) : learning algorithm to update \(p(k)\)
- \(p(k) = [p_1(k), p_2(k), \ldots, p_r(k)]^T\) : action probability vector at instant \(k\)
Overview of a Learning Automaton (LA)

A simple model for dynamic decision making in unknown environments.

Structure of FALA (Finite Action Learning Automaton)

Formally, a FALA can be described by the quadruple \((A, B, \tau, p(k))\):

- \(A = \{\alpha_1, \alpha_2, ..., \alpha_r\}\) : finite set of actions.
- \(B\) : set of all possible reinforcements
- \(\tau\) : learning algorithm to update \(p(k)\)
- \(p(k) = [p_1(k), p_2(k), ..., p_r(k)]^T\) : action probability vect at instant \(k\)

Higher the probability value for a thread, higher is its priority for DRAM scheduling.
1. Choose action (schedule a memory request) based on action probability vector.
Operation of a Single FALA

1. Choose action (schedule a memory request) based on action probability vector.
2. Get reinforcement (harmonic speedup) from the system.
Operation of a Single FALA

1. Choose action (schedule a memory request) based on action probability vector.
2. Get reinforcement (harmonic speedup) from the system.
3. Update the action probabilities (thread priorities) using equation 2.
1. Choose action (schedule a memory request) based on action probability vector.
2. Get reinforcement (harmonic speedup) from the system.
3. Update the action probabilities (thread priorities) using equation 2.
- This cycle repeats forever
The Learning Algorithm $\tau$

Linear Reward-Inaction ($L_{R-I}$) [7] is one learning algorithm:

$$p_i = p_i + \lambda \cdot \beta \cdot (1 - p_i)$$
$$p_j = p_j - \lambda \cdot \beta \cdot p_j, \quad \forall j \neq i$$

The above 2 equations can be combined using vector notation:

$$\mathbf{p}(k + 1) = \mathbf{p}(k) + \lambda \beta(k)(\mathbf{e}_i - \mathbf{p}(k)) \quad (1)$$
Our Learning Automata-based Algorithm

The Learning Algorithm $\tau$

Linear Reward-Inaction ($L_{R-I}$) [7] is one learning algorithm:

\begin{align*}
    p_i &= p_i + \lambda \cdot \beta \cdot (1 - p_i) \\
    p_j &= p_j - \lambda \cdot \beta \cdot p_j, \quad \forall j \neq i
\end{align*}

The above 2 equations can be combined using vector notation:

\begin{equation}
    p(k + 1) = p(k) + \lambda \beta(k)(e_i - p(k)) \tag{1}
\end{equation}

Equation for a team of N FALA

\begin{equation}
    p_i(k + 1) = p_i(k) + \lambda \beta(k) [e_{\alpha_i(k)} - p_i(k)], 1 \leq i \leq N \tag{2}
\end{equation}
The Learning Algorithm \( \mathcal{T} \)

Linear Reward-Inaction \((L_{R-I})\) [7] is one learning algorithm:

\[
\begin{align*}
p_i &= p_i + \lambda \cdot \beta \cdot (1 - p_i) \\
p_j &= p_j - \lambda \cdot \beta \cdot p_j, \quad \forall j \neq i
\end{align*}
\]

The above 2 equations can be combined using vector notation:

\[
p(k + 1) = p(k) + \lambda \beta(k) (e_i - p(k)) \quad (1)
\]

Equation for a team of \( N \) FALA

\[
p_i(k + 1) = p_i(k) + \lambda \beta(k) [e_{\alpha_i(k)} - p_i(k)] , 1 \leq i \leq N \quad (2)
\]

The automata implicitly cooperate to perform a stochastic search over the space of rewards [7] : coordination among multiple memory controllers.
Algorithm 1 Request prioritization in each memory controller

1: **Sampled action first:** Select a request according to the action probability vector.
2: **Row hit first:** Select a request which hits the row-buffer.
3: **Oldest first:** Select the oldest request.

---

Algorithm 2 Sampling an action

1: $cum\_prob[0] = p[0]$
2: for $count \leftarrow 1, (numThreads - 1)$ do
3: \hspace{1cm} if $rnd < cum\_prob[count - 1]$ then
4: \hspace{2cm} break
5: \hspace{1cm} else
6: \hspace{2cm} $cum\_prob[count] = cum\_prob[count - 1] + p[count]$
7: \hspace{1cm} end if
8: end for
9: action $\leftarrow count - 1$
Implementation

- Storage cost per controller: 3.3 Kbits (TCMS = 2.6 Kbits)
Implementation

- Storage cost per controller: 3.3 Kbits (TCMS = 2.6 Kbits)
- Additional logic is required for calculating the reward and updating $p(k)$

Calculating HS on-the-fly: Requires instantaneous IPC alone. We use overall IPC alone, obtained by running a benchmark alone on the same baseline system, to get a rough estimate of HS.

Updating $p(k)$ is not on critical path. Can be performed in many tens of CPU cycles.

As an approximation, we consider the latency for determining the reward for a scheduling decision to be 90 cycles.
Our Learning Automata-based Algorithm

Implementation

- Storage cost per controller: 3.3 Kbits (TCMS = 2.6 Kbits)
- Additional logic is required for calculating the reward and updating \( p(k) \)

- Calculating HS on-the-fly: Requires instantaneous \( IPC_i^{alone} \). We use overall \( IPC_i^{alone} \), obtained by running a benchmark alone on the same baseline system, to get a rough estimate of HS.
Our Learning Automata-based Algorithm

Implementation

- Storage cost per controller: 3.3 Kbits (TCMS = 2.6 Kbits)
- Additional logic is required for calculating the reward and updating $p(k)$
- Calculating HS on-the-fly: Requires instantaneous $IPC_{i}^{alone}$. We use overall $IPC_{i}^{alone}$, obtained by running a benchmark alone on the same baseline system, to get a rough estimate of HS.
- Updating $p(k)$ is not on critical path. Can be performed in many tens of CPU cycles.
Implementation

- Storage cost per controller: 3.3 Kbits (TCMS = 2.6 Kbits)
- Additional logic is required for calculating the reward and updating $p(k)$

- Calculating HS on-the-fly: Requires instantaneous $IPC_i^{alone}$. We use overall $IPC_i^{alone}$, obtained by running a benchmark alone on the same baseline system, to get a rough estimate of HS.
- Updating $p(k)$ is not on critical path. Can be performed in many tens of CPU cycles.
- As an approximation, we consider the latency for determining the reward for a scheduling decision to be 90 cycles.
Our Learning Automata-based Algorithm

Implementation

- Storage cost per controller: 3.3 Kbits (TCMS = 2.6 Kbits)
- Additional logic is required for calculating the reward and updating $p(k)$

- Calculating HS on-the-fly: Requires instantaneous $IPC_i^{alone}$. We use overall $IPC_i^{alone}$, obtained by running a benchmark alone on the same baseline system, to get a rough estimate of HS.
- Updating $p(k)$ is not on critical path. Can be performed in many tens of CPU cycles.
- As an approximation, we consider the latency for determining the reward for a scheduling decision to be 90 cycles.
Experimental Setup

- Modified version gem5 simulator
Experimental Setup

- Modified version gem5 simulator
- 16 CPU cores and 4 memory controllers
Experiments

Experimental Setup

- Modified version gem5 simulator
- 16 CPU cores and 4 memory controllers
- PARSEC: Eight multi-threaded benchmarks with simmedium input set.
Experimental Setup

- Modified version gem5 simulator
- 16 CPU cores and 4 memory controllers
- PARSEC: Eight multi-threaded benchmarks with simmedium input set.
- SPEC CPU2006: Eight multiprogrammed workloads of varying memory intensity run for 500mn instructions
Experimental Setup

- Modified version gem5 simulator
- 16 CPU cores and 4 memory controllers
- PARSEC: Eight multi-threaded benchmarks with *simmedium* input set.
- SPEC CPU2006: Eight multiprogrammed workloads of varying memory intensity run for 500mn instructions
Results

PARSEC

SPEC CPU2006
Scalability

% improvement over TCMS

Number of cores / memory controllers

8 / 2
16 / 4
24 / 6
Future Work

- Improve the reward mechanism

- Evaluate on a wider variety of workloads (SPLASH and NAS benchmarks)
- Compare against more recent scheduling algorithms (MISE)
- A more accurate hardware feasibility analysis
- Evaluate on a synthetic workload where the outcome should be predictable.
Future Work

- Improve the reward mechanism
- Evaluate on a wider variety of workloads (SPLASH and NAS benchmarks)
Future Work

- Improve the reward mechanism
- Evaluate on a wider variety of workloads (SPLASH and NAS benchmarks)
- Compare against more recent scheduling algorithms (MISE)
Future Work

- Improve the reward mechanism
- Evaluate on a wider variety of workloads (SPLASH and NAS benchmarks)
- Compare against more recent scheduling algorithms (MISE)
- A more accurate hardware feasibility analysis
Future Work

- Improve the reward mechanism
- Evaluate on a wider variety of workloads (SPLASH and NAS benchmarks)
- Compare against more recent scheduling algorithms (MISE)
- A more accurate hardware feasibility analysis
- Evaluate on a synthetic workload where the outcome should be predictable.
Future Work

- Improve the reward mechanism
- Evaluate on a wider variety of workloads (SPLASH and NAS benchmarks)
- Compare against more recent scheduling algorithms (MISE)
- A more accurate hardware feasibility analysis
- Evaluate on a synthetic workload where the outcome should be predictable.
- A learning technique is exploited to give improvement in fairness without much additional hardware cost.
- A learning technique is exploited to give improvement in fairness without much additional hardware cost.
- Scalable and works on multiprogrammed as well as parallel workloads
- A learning technique is exploited to give improvement in fairness without much additional hardware cost.
- Scalable and works on multiprogrammed as well as parallel workloads
Questions ?
E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana.  
Self-optimizing memory controllers: A reinforcement learning approach.  

Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter.  

Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter.  
Thread cluster memory scheduling: Exploiting differences in memory access behavior.  

J. Mukundan and J. Martinez.  
Morse: Multi-objective reconfigurable self-optimizing memory scheduler.  

O. Mutlu and T. Moscibroda.  
Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems.  

Mise: Providing performance predictability and improving fairness in shared main memory systems.  

M. A. L. Thathachar and P. S. Sastry.  
Networks of Learning Automata.  