Improving Fairness in Memory Scheduling

Using a Team of Learning Automata

Aditya Kajwe and Madhu Mutyam

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

June 14, 2014

Outline

Introduction

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 $= \frac{N}{\sum_i \frac{IPC_i^{alone}}{IPC_i^{shared}}}$

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 $= \frac{N}{\sum_i \frac{IPC_i^{alone}}{IPC_i^{shared}}}$
- Provides a good balance between fairness and system performance

Related Work

- ATLAS [2]: prioritizes threads that have attained the least service

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

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.

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 $(A, B, \tau, p(k))$:

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.

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

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
- τ : learning algorithm to update $\mathbf{p}(k)$

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 $(A, B, \tau, p(k))$:

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

Overview of a Learning Automaton (LA)

A simple model for dynamic decision making in unknown environments.

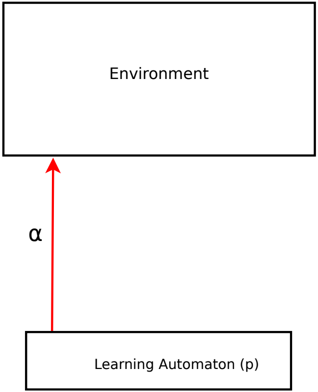
Structure of FALA (Finite Action Learning Automaton)

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
- τ : learning algorithm to update $\mathbf{p}(k)$
- $\mathbf{p}(k) = [p_1(k), p_2(k), ..., p_r(k)]^T$: action probability vect at instant k

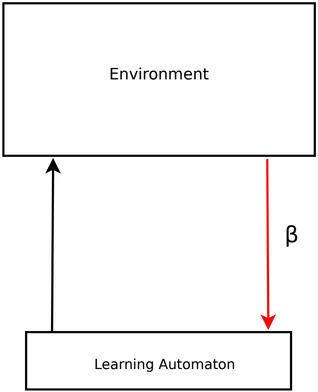
Higher the probability value for a thread, higher is its priority for DRAM scheduling.

Operation of a Single FALA



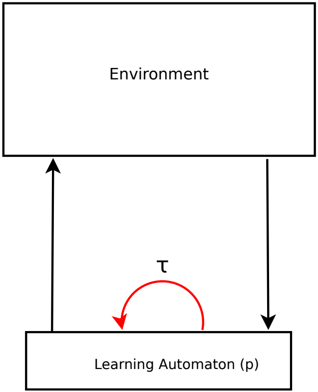
1. Choose action (schedule a memory request) based on action probability vector.

Operation of a Single FALA



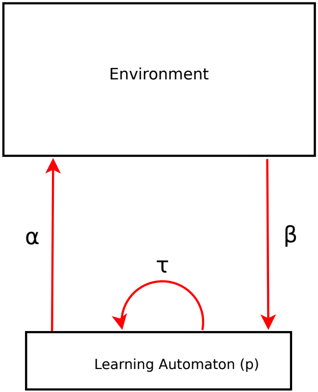
1. Choose action (schedule a memory request) based on action probability vector.
2. Get reinforcement (harmonic speedup) from the system.

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.

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.

- This cycle repeats forever

The Learning Algorithm τ

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:

$$\mathbf{p}(k+1) = \mathbf{p}(k) + \lambda\beta(k)(\mathbf{e_i} - \mathbf{p}(k)) \tag{1}$$

The Learning Algorithm τ

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:

$$\mathbf{p}(k+1) = \mathbf{p}(k) + \lambda\beta(k)(\mathbf{e_i} - \mathbf{p}(k)) \tag{1}$$

Equation for a team of N FALA

$$\mathbf{p}_i(k+1) = \mathbf{p}_i(k) + \lambda\beta(k)\left[\mathbf{e}_{\alpha_i(k)} - \mathbf{p}_i(k)\right], 1 \leq i \leq N \tag{2}$$

The Learning Algorithm τ

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:

$$\mathbf{p}(k+1) = \mathbf{p}(k) + \lambda\beta(k)(\mathbf{e_i} - \mathbf{p}(k)) \tag{1}$$

Equation for a team of N FALA

$$\mathbf{p}_i(k+1) = \mathbf{p}_i(k) + \lambda\beta(k)\left[\mathbf{e}_{\alpha_i(k)} - \mathbf{p}_i(k)\right], 1 \leq i \leq N \tag{2}$$

The automata implicitly cooperate to perform a stochastic search over the space of rewards [7] : coordination among multiple memory controllers.

Scheduling

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

```
cum_prob[0] = p[0]
for count ← 1, (numThreads − 1) do
    if rnd < cum_prob[count − 1] then
        break
    else
        cum_prob[count] = cum_prob[count − 1] + p[count]
    end if
end for
action ← 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 $\mathbf{p}(k)$

Implementation

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

- 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.

Implementation

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

- 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 $\mathbf{p}(k)$ 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 $\mathbf{p}(k)$

- 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 $\mathbf{p}(k)$ 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.

Implementation

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

- 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 $\mathbf{p}(k)$ 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

Experimental Setup

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

Experimental Setup

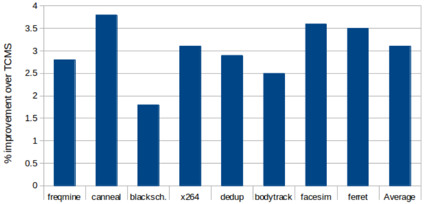
- Modified version gem5 simulator
- 16 CPU cores and 4 memory controllers
- PARSEC: Eight multi-threaded benchmarks with *simmedium* input set.
- SPEC CPU2006: Eight multiprogrammed workloads of varying memory intensity run for 500mn instructions

Experimental Setup

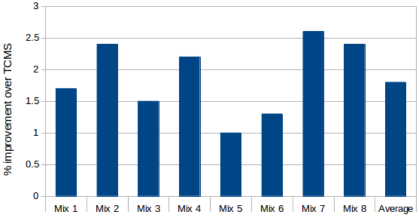
- Modified version gem5 simulator
- 16 CPU cores and 4 memory controllers
- PARSEC: Eight multi-threaded benchmarks with *simmedium* input set.
- SPEC CPU2006: Eight multiprogrammed workloads of varying memory intensity run for 500mn instructions

Results

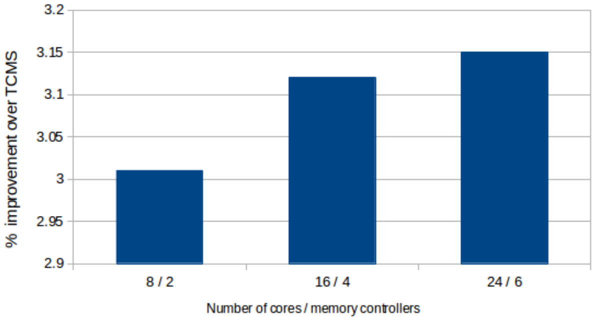
PARSEC



SPEC CPU2006



Scalability



Future Work

- Improve the reward mechanism

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.

Conclusion

- A learning technique is exploited to give improvement in fairness without much additional hardware cost.

Conclusion

- 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

Conclusion

- 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.
In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 39–50, Washington, DC, USA, 2008. IEEE Computer Society.

Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter.
Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers.
In M. T. Jacob, C. R. Das, and P. Bose, editors, *HPCA*, pages 1–12. IEEE Computer Society, 2010.

Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter.
Thread cluster memory scheduling: Exploiting differences in memory access behavior.
In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 65–76, Washington, DC, USA, 2010. IEEE Computer Society.

J. Mukundan and J. Martinez.
Morse: Multi-objective reconfigurable self-optimizing memory scheduler.
In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12, Feb 2012.

O. Mutlu and T. Moscibroda.
Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems.
In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 63–74, Washington, DC, USA, 2008. IEEE Computer Society.

L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu.
Mise: Providing performance predictability and improving fairness in shared main memory systems.
In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, pages 639–650, Washington, DC, USA, 2013. IEEE Computer Society.

M. A. L. Thathachar and P. S. Sastry.
Networks of Learning Automata.
Springer, 2004.