Improving Fairness in Memory Scheduling Using a Team of Learning Automata

Aditya Arvind Kajwe and Madhu Mutyam
Department of Computer Science and Engineering,
Indian Institute of Technology - Madras
Email: {adityaka, madhu}@cse.iitm.ac.in

***Abstract*—Conventional memory controllers deliver relatively low fairness partly because they do not learn from their past decisions. This paper proposes an intelligent memory scheduling technique for multiple memory controllers. The technique is decentralized and the controllers implicitly cooperate with each other without any exchange of information among them. Our learning technique on a 16-core simulated system gives 3.12% improvement in harmonic speedup for PARSEC workloads and 1.88% for SPEC CPU2006 workloads over Thread Cluster Memory Scheduling algorithm.**

I. Introduction

The order in which DRAM accesses are scheduled can have a dramatic impact on main memory performance, power consumption and fairness. This paper targets the fairness metric. A system is fair if all the threads experience an equal slowdown compared to their performance had they been executed alone.

A prior study [1] has demonstrated the need for coordination between multiple memory controllers. Without coordination, each memory controller (MC) is oblivious of others and makes locally greedy scheduling decisions which could hamper fairness of the system. For example, consider a system with two memory controllers and four threads. Without coordination there could be a situation where both controllers are prioritizing the same thread's requests, which could degrade the system fairness.

Key idea: *We propose to design a system of multiple memory controllers as a team of finite action learning automata (FALA), with each memory controller acting as a single FALA, whose goal is to automatically learn optimal thread priorities via interaction with rest of the computer system.*

Such a technique directly takes as input a system parameter, and learns what actions to take so as to maximize that parameter. It also eliminates the need for explicit coordination between the memory controllers.

To evaluate our technique, we use a cycle-accurate multicore simulator with multiple memory channels. We compare the performance and hardware overhead of our scheduler against a state-of-the-art algorithm, using both parallel applications and multiprogrammed workloads.

II. Background

We provide a brief overview of the operation of a memory controller in modern DRAM systems and provide background information on team of FALA.

A. Memory Scheduling

DRAM has a three-dimensional structure of bank, row and column. The amount of time it takes to service a DRAM request depends on the status of the row-buffer and there are three possibilities:

- Row hit: The request is to the row that is currently open in the row-buffer. The MC only needs to issue a column access (CAS) command to the DRAM bank.
- Row closed : The row-buffer is empty. The MC needs to first issue an activate command (ACT) to open the required row, then a column access command.
- Row conflict: The request is to a row different from the one currently in the row-buffer. The MC needs to first close the row by issuing a precharge command (PRE), then issue ACT and CAS commands.

It is clear that a row hit request has the least access latency. Scheduling algorithms that focus solely on increasing the row-buffer hits result in a degradation of system fairness because they tend to unfairly prioritize threads with high row-buffer locality over those with relatively low row-buffer locality. FR-FCFS [2] is one of the earliest such scheduling algorithms.

Thread Cluster Memory Scheduling (TCMS) [3] is a state-of-the-art technique that divides threads into two separate clusters and employs different memory request scheduling policies in each cluster. This is based on the applications' memory intensity measured in last level cache misses per thousand instructions. Threads in the latency-sensitive cluster are always prioritized over threads in the bandwidth-sensitive cluster. To ensure that no thread is disproportionately slowed down, TCMS periodically shuffles the priority order among threads in the bandwidth-sensitive cluster.

B. Overview of a Learning Automaton

A learning automaton (LA) is a simple model for dynamic decision making in unknown environments. It tries different actions, and chooses new actions based on the environment's response to previous actions.

Figure 1 shows a general block diagram of a LA operating in a random environment.

FALA are a particular class of LA with finitely many actions. Formally, a FALA can be described by the quadruple $(A, B, \tau, p(k))$ where

- $A = \{\alpha_1, \alpha_2, ..., \alpha_r\}$ is the finite set of actions.

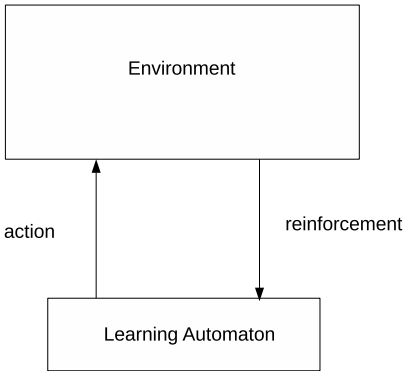

Fig. 1. Block diagram of a learning automaton

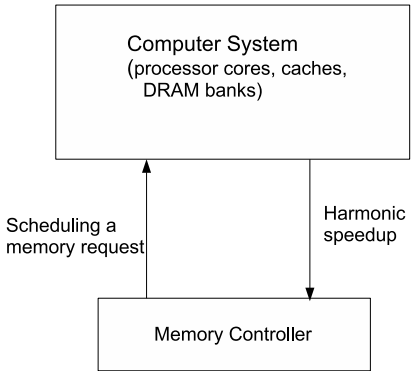

Fig. 2. Memory controller acting as a learning automaton

- B is the set of all possible reinforcements to the automaton
- τ is the learning algorithm for updating action probabilities
- $\mathbf{p}(k)$ is the action probability vector at instant k given by

$$\mathbf{p}(k) = [p_1(k), p_2(k)...p_r(k)]^T \quad (1)$$

Here $p_i(k)$ is the probability with which action α_i is chosen at instant k.

One of the popular FALA learning algorithms (τ) is the Linear Reward-Inaction (L_{R-I}) [4]. It updates action probabilities using the equation:

$$\mathbf{p}(k+1) = \mathbf{p}(k) + \lambda\beta(k)(\mathbf{e_i} - \mathbf{p}(k)) \quad (2)$$

Here $\mathbf{e_i}$ is the unit vector with i^{th} component unity where the index i corresponds to the action selected in the current instant. $\beta(k)\epsilon B$ is a real number, and is the reinforcement from the environment. λ is the learning parameter. Intuitively, we increase the probability of the selected action, and decrease the probability of others, depending on the magnitude of reward.

C. Games of FALA

In a multiautomata system, the automata can be viewed as players involved in a game. It can then be called a team of FALA. The update equation for a team of N FALA is just a generalization of equation 2 :

$$\mathbf{p}_i(k+1) = \mathbf{p}_i(k) + \lambda\beta(k)\left[\mathbf{e}_{\alpha_i(k)} - \mathbf{p}_i(k)\right], 1 \leq i \leq N \quad (3)$$

Though this game model is completely decentralized, effectively the automata are cooperating with each other. For a detailed mathematical treatment of the algorithm, the interested reader is referred to [4]. Intuitively, the algorithm performs a stochastic search over the space of rewards.

III. LA-BASED DRAM SCHEDULERS: STRUCTURE, OPERATION AND IMPLEMENTATION

In this section we build upon the theoretical background of the previous section to make a case for modelling multiple memory controllers as a team of FALA.

A. Formulation

Figure 2 shows a single memory controller acting as a learning automaton.

Actions (α_i) : Executing an action α_i is the same as scheduling a DRAM request which belongs to thread i. In a 16-core CMP, possible actions are 1 to 16.

Reward (β) : Harmonic speedup [5] is used as the reinforcement from the environment. It provides a good balance between fairness and system performance.

Action probability vector ($\mathbf{p_i}$) : Each MC has 16 probability values, one for each thread in the system. Higher the probability value for a thread, higher is its priority.

B. Algorithm (τ)

At each instant, all MCs choose actions based on their respective action probability vectors. When the system is booted up, each action has an equal probability of getting selected. Reinforcement from the system is obtained. Knowing the actions selected and reinforcement, each MC updates the action probabilities using equation 3. This cycle of selecting an action, eliciting reinforcement and updating probabilities repeats.

In case there are multiple memory requests which belong to the selected thread, a request which hits the row-buffer is executed. If there are multiple row-hit requests, then the oldest request is executed.

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.

The L_{R-I} algorithm requires a stationary environment i.e. the distribution of rewards should be constant. Though this

isn't the case for our benchmarks running on a computer system, the benchmarks run in phases which are long enough for the algorithm to achieve some learning.

C. Implementation

Our technique relies on harmonic speedup to steer memory scheduling. Calculating this parameter on-the-fly during program execution is non-trivial because it requires the instantaneous value of IPC_{alone} . We use average value of IPC_{alone}, obtained by running a benchmark alone on the same baseline system, to get a rough estimate of harmonic speedup.

The hardware cost of LA-based scheduler consists of two portions: 1) SRAM arrays required to store probability values 2) logic required to choose an action based on probabilities, update the probability values and to calculate the reinforcement value. The total storage cost of our algorithm for the baseline 16-core system is 3.3 Kbits. TCMS requires a storage of 2.6 Kbits.

Note that updating the probability vector is not on the critical path of making scheduling decisions, and can be performed in many tens of processor cycles without significantly affecting the quality of scheduling decisions.

The effect of a scheduling decision on harmonic speedup won't be apparent until some time has elapsed, and by using timestamps on memory requests, we determine this to be approximately 90 CPU cycles. Since this time is somewhat variable, it is difficult to isolate the effect of a single scheduling action on harmonic speedup (Harmonic speedup is also influenced by factors other than memory scheduling). As an approximation, we consider the latency from making a scheduling decision to determining the reward for that decision to be 90 cycles.

IV. Experiments

We evaluate the fairness of our LA-based scheduling algorithm using gem5 [6] simulator. Eight multiprogrammed workloads were formed from SPEC CPU2006 benchmarks (see table I) and and were run for 500 million instructions. Eight multithreaded PARSEC [7] benchmarks were run with the *simmedium* input set. Number of threads of each PARSEC benchmark is equal to the number of processor cores. All the numbers are for PARSEC benchmarks unless otherwise specified. Table II shows the configuration of our baseline system.

Parameters: For LA-based algorithm, we set λ = 0.1. For TCMS we set ClusterThresh to 1/4, ShuffleInterval to 800, and ShuffleAlgoThresh to 0.1.

Figure 3 shows percentage improvement in harmonic speedup of our algorithm over TCMS. Higher improvement is seen for programs like *canneal, facesim* and *ferret* which have high bandwidth requirements and also large working sets. The average improvement in fairness is 3.12 %.

Figure 4 shows percentage improvement in harmonic speedup of our algorithm over TCMS, for SPEC CPU2006 benchmarks. Higher improvement is seen for workload mixes having a higher percentage of memory-intensive benchmarks.

TABLE I. Eight SPEC CPU2006 workloads (figure in parentheses is the number of instances spawned)

Workload	Memory-non-intensive benchmarks	Memory-intensive benchmarks
Mix 1	calculix(3), dealII, gcc(2), gromacs(2)	xalancbmk, omnetpp(2), astar, hmmer(2), lbm(2)
Mix 2	gobmk, namd, dealII	GemsFDTD(3), h264ref(3), hmmer(2), libquantum(2), sphinx3(3)
Mix 3	calculix(2), gromacs(3), namd(3), povray10(3)	omnetpp(2), bzip, soplex(2)
Mix 4	gcc(3), gromacs, povray(2), sjeng(2), tonto	h264ref, astar, GemsFDTD(3), bzip(2)
Mix 5	sjeng(3), namd(3), gcc(4), gromacs(3)	astar, hmmer, bzip
Mix 6	gromacs(3), namd(2), gcc(2), sjeng, calculix(3)	bzip, astar, hmmer, libquantum(2)
Mix 7	povray(2), calculix, sjeng	xalancbmk(4), omnetpp(2), astar(2), hmmer(4)
Mix 8	gobmk, sjeng	h264ref(4), libquantum(3), astar(3), omnetpp(4)

TABLE II. Baseline CMP and memory system configuration

L1 caches	32 KB per core, 4-way set associative
L2 caches	256 KB per core, 8-way set associative
Cache block size	64B
CPU cores	16 cores, 2Ghz
DRAM controllers	4
DRAM chip parameters	Micron DDR3-1066
Ranks per channel	2
Address mapping	row:rank:bank:channel:column
DRAM queue sizes	128-entry read, 64-entry write buffer
Row-buffer size	2KB
Page policy	Open page

The average improvement in fairness is 1.88 %. The improvement is higher for PARSEC benchmarks as TCMS was designed to target thread heterogeneity, and does not show much improvement when applied to parallel workloads.

Figure 5 shows that our algorithm is also scalable as it maintains its performance improvement over TCMS when the number of cores and memory controllers are increased.

V. Related Work

With the shift to many-core era, attention has shifted from schedulers that do not distinguish between different

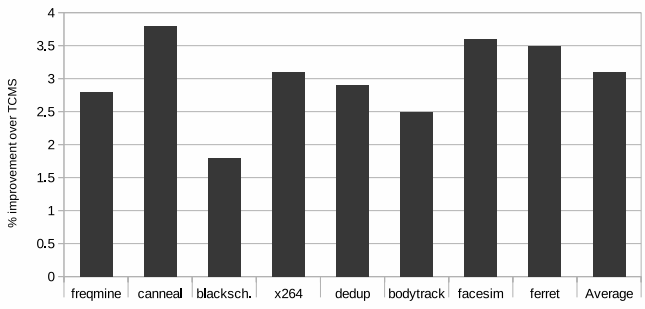

Fig. 3. Harmonic speedup for PARSEC benchmarks

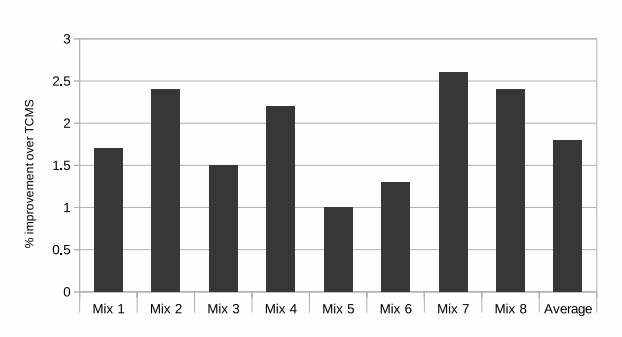

Fig. 4. Harmonic speedup for SPEC CPU2006 benchmarks

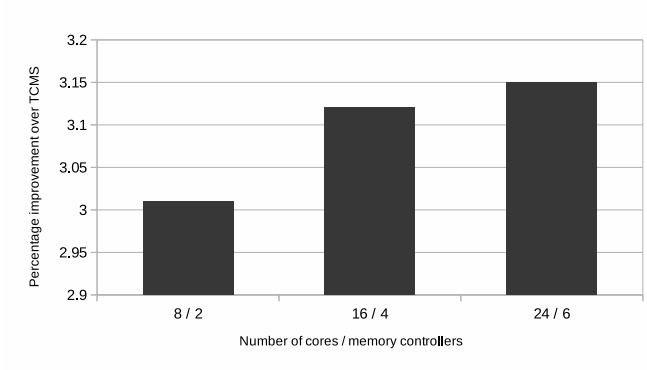

Fig. 5. Scalability of harmonic speedup

threads, to thread-aware memory schedulers. A number of algorithms have been proposed in the latter category in the recent years. Kim et al. [1] prioritize threads that have attained the least service over others in each time quantum. Multu and Moscibroda [8] propose a mechanism which processes DRAM requests in batches, and uses the shortest-job-first principle to prioritize requests within a batch. Stall-time fair memory scheduler [9] uses heuristics to estimate the slowdown of each thread and prioritizes the thread that has been slowed down the most. Ebrahimi et al. [10] prioritize threads that are likely to be on the critical path. Ghose et al. [11] prioritize load requests based on ranking information supplied from the processor side. Ipek et al. [12] propose a memory controller that uses reinforcement learning to dynamically optimize its scheduling policy. MORSE [13] algorithm extends this technique to target arbitrary figures of merit. However, both these algorithms incur a significant hardware overhead (MORSE requires 1024 Kbits of storage on our baseline system).

VI. Conclusion

We explored a novel memory scheduling algorithm that tries to learn optimal thread priorities via interaction with the computer system. It provides a reasonable improvement in fairness over the TCMS algorithm for parallel workloads. Our black-box technique directly targets the fairness metric and the improvement implies that it implicitly captures the parameters of system that help in increasing system fairness.

Acknowledgement

We would like to thank B.Ravindran for insightful discussions and providing valuable feedback on this work.

References

[1] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *HPCA*, M. T. Jacob, C. R. Das, and P. Bose, Eds. IEEE Computer Society, 2010, pp. 1–12. [Online]. Available: http://dblp.uni-trier.de/db/conf/hpca/hpca2010.html

[2] S.Rixner, W.J.Dally, U.J.Kapasi, P.Mattson, and J.D.Owens, "Memory access scheduling," in *27th Int'l Symp. on Computer Architecture*, 2000, pp. 128–138.

[3] 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*, ser. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 65–76. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2010.51

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

[5] K. Luo, J. Gummaraju, and M. Franklin, "Balancing throughput and fairness in smt processors," in *Performance Analysis of Systems and Software, 2001. ISPASS. 2001 IEEE International Symposium on*, 2001, pp. 164–171.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 72–81. [Online]. Available: http://doi.acm.org/10.1145/1454115.1454128

[8] 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*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 63–74. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2008.7

[9] ——, "Stall-time fair memory access scheduling for chip multiprocessors," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 146–160. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2007.40

[10] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt, "Parallel application memory scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 362–373. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155663

[11] S. Ghose, H. Lee, and J. F. Martínez, "Improving memory scheduling via processor-side load criticality information," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 84–95. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485930

[12] 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*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 39–50. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2008.21

[13] 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*, Feb 2012, pp. 1–12.