Synchronous Q-Learning Algorithm for Enhanced Load Balancing in Multi-Core Processors

[1]Avinash Dhole and [2]Shailendra Verma
[1]Associate Professor &Head ,[2]Assistant Professor

[1] Department of Computer Science and Engg.
[1]Raipur Institute of Technology, Raipur, CG, India

[2]Department of Electrical Engineering,
[2]Shri Shankaracharya Technical Campus, Bhilai, CG, India

Abstract

Conventional processors are extensively used for applications like weather forecasting, AI, ocean modeling, and big data analysis. This research investigates parallel computing strategies aimed at reducing execution time. We developed a new dynamic load balancing technique by integrating Classical Q-Learning with Modified Q-Learning, termed Synchronous Q-Learning. Simulation results demonstrate that this method reduces execution time in multi-core systems. The Synchronous Q-Learning algorithm combines policy iteration and classical Q-Learning to optimize load balancing through reinforcement learning.

Keywords: Multi-core Processing, Reinforcement Learning, Load Balancing, Machine Learning.
DOI Number:10.48047/nq.2021.19.1.NQ21050 **NeuroQuantology2021;19(1):387-393**

1.Introduction

With the advent of multi-core processors, the complexity of load balancing through thread scheduling has significantly increased. Most schedulers prioritize threads that are ready to run when making scheduling decisions. Since multi-core processors share resources like the L2 cache, the behavior of a thread running on one core can impact the performance of threads running on other cores. For example, if two threads monopolize the L2 cache while running on separate cores, their performance may suffer compared to if they were scheduled alongside other threads that don't heavily use the cache. The situation becomes more complicated as a thread's behavior can change over time—such as being memory-bound at one moment and computationally-bound later. Additionally, threads often communicate with each other to complete tasks, which means it can be beneficial to co-schedule them on cores where they can share data via the L2 cache.

There is a need for a scheduler that can dynamically learn which threads should be co-scheduled, which should not, and which threads should run together on the same core, all while accounting for the fact that a thread's behavior may vary over time. A static algorithm wouldn't be able to consistently deliver optimal performance in such dynamic conditions.

Reinforcement Learning (RL) can address this challenge because it allows feedback to be provided to the algorithm, enabling it to learn and optimize core usage per thread. For

example, RL could monitor metrics such as how often a thread is scheduled (more is better), how long it waits (less is better), how often it's blocked (less is better), and processor performance counters, such as the number of instructions retired in a given period (more is better), cache misses (fewer is better), and failed synchronization attempts (fewer is better). These goals may conflict—for instance, reducing synchronization failures by running all threads on the same core could decrease the total instruction execution rate. This complexity requires auto-tuning, which RL can achieve.

In one approach, performance counters and address-tracing hardware (not available on all processors) were used to detect cache-sharing patterns, clustering threads based on these patterns and scheduling threads in the same cluster on the same core. A model of the cache, memory hierarchy, and CPU was also developed, using hardware performance counters to determine when threads should be moved. However, RL was not employed in this strategy, nor was inter-process communication considered.

We propose using machine learning to design a self-optimizing, adaptive memory controller capable of planning, learning, and continuously adjusting to workload demands. We frame memory access scheduling as a reinforcement learning problem. RL focuses on how autonomous agents operating in stochastic environments can learn optimal control policies through interaction. It offers a powerful framework for designing high-performance, self-improving controllers. The benefits include efficiently handling large memory capacities by leveraging hardware performance counters and additional parameters learned via RL. This approach would run on each thread to determine which thread movement would bring the greatest benefit to the overall multi-core processing system.

2. Methodology

Chip Multiprocessors (CMPs) are an attractive alternative to powerful single-core processors due to their advantages in energy efficiency, performance, and reduced complexity. Current industry trends suggest that scaling CMPs to include more cores will be the key to leveraging Moore's Law in the era of billion-transistor chips. If CMOS scaling continues as predicted by Moore's Law, CMPs could double the number of cores and available on-chip cache space every two years. However, the benefits of Moore's Law are not fully realized in traditional packaging technologies. As a result, both signal speed and the number of signaling pins grow at a much slower rate (about 10% annually), leading to potential off-chip bandwidth limitations that could hinder CMP scalability.

Providing sufficient peak bandwidth is only part of the challenge. In practice, delivering a large portion of this theoretical peak to real-world workloads requires a memory controller capable of efficiently utilizing the off-chip interface. Memory scheduling is complex, balancing access scheduling constraints, request prioritization, and adapting to a dynamically changing memory access pattern. Current memory controllers often only sustain a fraction of the peak bandwidth, resulting in significant performance losses or requiring over-provisioned and costly memory systems.

Performance loss is often caused by access scheduling constraints with modern DRAM controllers. For example, in a case-parallel application like SCALPARC, with both a contemporary controller design (using the FR-FCFS scheduling method) and an idealized design capable of achieving 100% of the controller's peak bandwidth, SCALPARC achieves over 85% of the available memory bandwidth with the idealized controller. However, when using the practical scheduler, bandwidth utilization drops to 46%. This inefficiency results in longer L2 load miss penalties, increasing from an average of 482 cycles with the idealized scheduler to 801 cycles with the practical one, causing a 55% performance loss.

One major limitation in current memory controller designs is their ad-hoc nature. Typically, a human designer selects a few attributes that are believed to be important for optimizing a specific performance target (e.g., the average wait time in FR-FCFS), based on

prior experience. The designer then creates a fixed scheduling policy incorporating these attributes and tests it using simulation models. However, these controllers often lack two key functions: they cannot anticipate the long-term effects of their scheduling decisions, nor can they learn from past experiences to make better decisions in new situations. This lack of flexibility and adaptability can lead to substantial performance degradation in many applications.

We propose designing the memory controller as a reinforcement learning (RL)-based thread scheduling agent, with the goal of automatically learning an optimal memory scheduling policy through interaction with the system. An RL-based memory controller would take input from the system state and consider the long-term performance impact of each decision. The controller's task would be to (1) associate system states and actions with long-term reward values, (2) make decisions that maximize long-term rewards (i.e., performance) for the current system state, and (3) continuously update these reward values based on feedback from the system, allowing it to adapt to changing workloads and memory access patterns. Unlike traditional memory controllers, an RL-based controller would:

- Anticipate the long-term consequences of its scheduling decisions and continuously refine its policy.
- Use learned experience from past system states to make better decisions in new, previously unseen scenarios.
- Adapt to evolving workload demands and memory access patterns.

An RL-based design approach allows hardware designers to focus on the performance goals of the controller and the system variables that could inform a good scheduling strategy, rather than creating a fixed plan that dictates exactly how the controller should achieve those goals. This approach not only reduces the human design effort involved in traditional controller development but, as our evaluation shows, also results in higher-performing controllers.

We evaluated our self-optimizing memory controller using a variety of parallel software applications from the SPEC OpenMP and NAS OpenMP benchmark suites. On a 4-core CMP with a single-channel DDR2-800 memory subsystem (with a peak bandwidth of 6.4GB/s), the RL-based memory controller improved performance by an average of 19% (up to 33%) compared to a state-of-the-art FR-FCFS scheduler. This effectively halved the performance gap between the single-channel setup and a more expensive dual-channel DDR2-800 subsystem. When applied to the dual-channel subsystem, the RL-based scheduler delivered an additional 14% performance improvement. Overall, our results demonstrate that self-optimizing memory controllers can more efficiently utilize available memory bandwidth in CMPs.

3. Synchronous Q-learning-based Load Balancing Algorithm

Input:

- N: Number of processors
- V: Communication load between nodes
- E: Communication lines corresponding to edges
- n: Number of processors
- l: Synchronization cost
- g: Bandwidth

Output:

- S: Scheduling table for communication steps between multiple cores, computation steps for parallelization, and synchronization steps.

Algorithm:

Step 1: Master Node (Core_0):

SN	Algorithm	Average Time (Uni-core)	Average Time (Multi-core)

After each C cycles ← Update the state and actions using

$$H_{t-1} = \{s_1, a_1, r_1, \dots, s_{t-1}, a_{t-1}, r_{t-1}\}$$

Step 2: Evaluate partition matrix thresholds:

$$\sum_{i=1}^{cc} \rho_{V,E} = \begin{cases} 1 \; if \; g \geq \dfrac{\sum_{i=0}^{c} w}{\sum_{j=1}^{c'} l} \\ 0 \; if \; g > \dfrac{\sum_{i=0}^{c} w}{\sum_{j=1}^{c'} l} \end{cases}$$

Where:

- V and E represent the vertex and edge of the memory page graph
- w is the local computation in progress
- c is the number of iterations in computation
- c' is the number of communication overheads.

Step 3: Repeat Steps 1-2 until all state sets are mapped.

Step 4: Initialize Q-learning:

$$Q_{t+1}(s,a) = \alpha_t(s,a)\left(R_{MDP}(s,a) + \gamma \max_{b \in U(s')} Q_t(s',b)\right) + \left(1 - \alpha_t(s,a)\right)Q_t(s,a)$$

Return V_{Max}

Step 5: Evaluate the page rank of each core's vertex based on weighted ordering of $Q_{t+1}(s,a)$

$$\varphi_V = \frac{n-E}{|V|} + \sum_{i=1}^{cc} \rho_{V,E}\, \varphi_E + l * g$$

Step 6: Compute T for all cores:

$$T = \Omega\left\{\left(\frac{V_{max}^2 \ln\left(\frac{|S||A|V_{max}}{\delta.\frac{1}{1-\gamma}.\varepsilon}\right)}{\left(\frac{1}{1-\gamma}\right)^2 \varepsilon^2}\right)^{\frac{1}{\omega}} + \left(\frac{\ln\left(\frac{V_{max}}{\varepsilon}\right)}{\left(\frac{1}{1-\gamma}\right)}\right)^{\frac{1}{1-\omega}}\right\}$$

Step 7: *While* (Core_i ≤ i) // for checking into each processing cores.

{

if ($\varphi_V < V_{Max}$) then allocate memory for computation:

$$M = \sum_{i=1}^{cc} \rho_{V,E} * \varphi_V$$

else if ($\varphi_V > V_{Max}$) then allocate memory:

Repeat steps 4, 5 & 6.

else

Skip and do the normal simulation

Core_i ++;

Return ,

$$H_{t-1} = \{s_1, a_1, r_1, \dots, s_{t-1}, a_{t-1}, r_{t-1}\}$$

}

Step 8: End process.

1	FFT	2.3608	0.00051071
2	Bit Shift Operation	1.3924	0.0034031
3	Circular Shift Operation	0.60792	0.0030516
4	Permute	6.0605	0.00038407
5	Divide	2.0872	0.0042635

This algorithm reduces memory page variation and efficiently maps parallel computational tasks. It optimizes the programming model for multi-core systems by synchronizing computational workloads with available bandwidth in real-time. Unlike traditional methods, which typically assign one segment per vertex, this approach uses a single balancing equation for parallelization, relying on page rank equations. This reduces memory mapping, prioritizes tasks based on page rankings, and avoids the need for hashing tables, ensuring effective memory usage for workload balancing.

4. Results and Discussion

Our test setup includes 32 dual-core Intel processors, each with 8 GB of memory, connected via 1 Gbps Ethernet. The Synchronous Q-Learning algorithm dynamically adjusted load balancing between computation and communication tasks. We observed that for smaller numbers of processors, the
communication overhead could outweigh the benefits of load balancing. However, as the number of processors increased, the algorithm demonstrated significant improvements in execution time.

The average execution time for different algorithms on both uni-core and multi-core processors is summarized below.

5. Conclusion:

Simulation results demonstrate that multi-processor systems require less time for load balancing. Existing literature highlights two primary algorithms for this purpose: Classical Q-learning and a Modified version known as Synchronous Q-learning. Building on these, we have developed a new dynamic load balancing technique by combining aspects of both algorithms, introducing it as the Synchronous Q-learning Algorithm. Simulations show that this proposed algorithm reduces processing time significantly.

In this paper, we present this novel Synchronous Q-learning algorithm, which merges components of policy iteration and classical Q-learning (or value iteration) to effectively learn and manage strategies for dynamic load balancing using reinforcement learning (RL) techniques. We address the issue of learning-based thread scheduling and propose this new approach for scheduling threads and memory controllers, grounded in RL principles. The mathematical results show how the number of inefficient moves can be bounded before reaching a near-optimal strategy with high certainty.

Our RL-based self-optimizing memory and thread scheduling controller continuously adapts its DRAM command scheduling policy based on system interactions, optimizing performance. This approach makes DRAM bandwidth usage more efficient compared to traditional controllers that rely on fixed scheduling policies. Additionally, it reduces the

design effort required by hardware engineers, as the scheduling strategy does not need to be manually optimized for all scenarios.

In simulations on a 4-core chip multiprocessor (CMP) with a single-channel DDR2-800 memory subsystem (6.4GB/s peak bandwidth), the RL-based memory controller improved the performance of parallel applications by an average of 34% (up to 67%) and increased DRAM bandwidth utilization by 19% over a state-of-the-art scheduler. These improvements close the performance gap between a single-channel setup and a more expensive dual-channel DDR2-800 system with twice the bandwidth. When applied to the dual-channel system, the RL-based scheduler delivers an additional 14% performance gain on average (up to 34%).

We conclude that RL-based self-optimizing memory controllers offer a promising approach to efficiently utilizing DRAM memory bandwidth in CMPs. The results suggest that a learner using current data consistently outperforms agents that explicitly explore in this computational space, making it a more practical solution for real-world applications.

References:

[1] J. Carter et al. "Impulse: Building a smarter memory controller." In HPCA-5, 1999.

[2] P. Auer, N. Cesa-Bianchi, and P. Fischer. "Finite time analysis of the multiarmed bandit problem." *Machine Learning*, 47(2-3):235-256, 2002.

[3] P. Auer, T. Jaksch, and R. Ortner. "Near-optimal regret bounds for reinforcement learning." In *Advances in Neural Information Processing Systems*, volume 21, pages 89-96, 2009.

[4] R. I. Brafman and M. Tennenholtz. "R-MAX: A general polynomial time algorithm for near-optimal reinforcement learning." *Journal of Machine Learning Research*, 3:213-231, 2003.

[5] E. Brunskill, B. R. Leffler, L. Li, M. L. Littman, and N. Roy. "Provably efficient learning with typed parametric models." *Journal of Machine Learning Research*, 10:1955-1988, 2009.

[6] E. Even-Dar and Y. Mansour. "Convergence of optimistic and incremental Q-learning." In *Advances in Neural Information Processing Systems*, volume 13, pages 1499-1506, 2001.

[7] E. Even-Dar, S. Mannor, and Y. Mansour. "PAC bounds for multi-armed bandit and markov decision processes." In *COLT '02: Proceedings of the 15th Annual Conference on Computational Learning Theory*, pages 255-270, 2002.

[8] R. Givan, T. Dean, and M. Greig. "Equivalence notions and model minimization in markov decision processes." *Artificial Intelligence*, 147(1-2):163-223, 2003.

[9] V. Aslot and R. Eigenmann. "Quantitative performance analysis of the SPEC OMPM2001 benchmarks." *Scientific Programming*, 11(2):105-124, 2003.

[10] D. H. Bailey et al. "NAS parallel benchmarks." Technical report, NASA Ames Research Center, March 1994. Tech. Rep. RNR-94-007.

[11] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. "Evidence-based static branch prediction using machine learning." *ACM TOPLAS*, 19(1), 1997.

[12] R. Caruana and D. Freitag. "Greedy attribute selection." In *International Conference on Machine Learning*, pages 28-36, New Brunswick, NJ, July 1994.

[13] Y. Chang, T. Hoe, and L. Kaelbling. "Mobilized ad-hoc networks: A reinforcement learning approach." In *International Conference on Autonomic Computing*, 2004.

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

[15] R. Glaubius, T. Tidwell, W. D. Smart, and C. Gill. "Scheduling design and verification for open soft real-time systems." In *RTSS'08: Proceedings of the 2008 Real-Time Systems Symposium*, pages 505-514, 2008.

[16] R. Glaubius, T. Tidwell, C. Gill, and W. D. Smart. "Scheduling policy design for autonomic systems." *International Journal on Autonomous and Adaptive Communications Systems*, 2(3):276-296, 2009.

[17] Advanced Micro Devices, Inc. *AMD Athlon(TM) XP Processor Model 10 Data Sheet*, Feb. 2003.

[18] J. H. Ahn, M. Erez, and W. J. Dally. "The design space of data-parallel memory systems." In *SC*, 2006.

[19] Anandtech. "Intel Developer Forum 2007." http://www.anandtech.com/cpuchipsets/intel/showdoc.aspx?i=3102.

[20] ITRS. "International Technology Roadmap for Semiconductors: 2005 Edition, Assembly and Packaging." http://www.itrs.net/Links/2005ITRS/AP2005.pdf.

[21] L. Spracklen and S. G. Abraham. "Chip multithreading: Opportunities and challenges." In *HPCA-11*, 2005.

[22] I. Hur and C. Lin. "Adaptive history-based memory schedulers." In *MICRO-37*, 2004.

[23] D. A. Jimenez and C. Lin. "Dynamic branch prediction with perceptrons." In *HPCA-7*, 2001.
[24] R. E. Kessler. "The Alpha 21264 microprocessor." *IEEE Micro*, 9(2):24-36, Mar. 1999.
[25] A. McGovern and E. Moss. "Scheduling straight line code using reinforcement learning and rollouts." In *Advances in Neural Information Processing Systems*, 1999.
[26] S. A. McKee et al. "Dynamic access ordering for streamed computations." *IEEE Transactions on Computers*, 49(11):1255-1271, Nov. 2000.
[27] Micron. *Technical Note TN-47-04: Calculating Memory System Power for DDR2*, June 2006. http://download.micron.com/pdf/technotes/ddr2/TN4704.pdf.
[28] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. "Memory access scheduling." In *ISCA-27*, 2000.
[29] T. Dunigan, M. R. Fahey, J. White, and P. Worley. "Early evaluation of the Cray X1." In *SC*, 2003.
[30] P. Kongetira, K. Aingaran, and K. Olukotun. "Niagara: A 32-way multithreaded sparc processor." *IEEE Micro*, 25(2):21-29, 2005.
[31] J. D. McCalpin. *Sustainable Memory Bandwidth in Current High Performance Computers*. http://home.austin.rr.com/mccalpin/papers/bandwidth/.
[32] M. Joshi, G. Karypis, and V. Kumar. "ScalParC: A new scalable and efficient parallel classification algorithm for mining large datasets." In *IPPS*, 1998.
[33] W. K. Zuravleff and T. Robinson. "Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order." United States Patent #5,630,096, May 1995.
[34] O. Mutlu and T. Moscibroda. "Stall-time fair memory access scheduling for chip multiprocessors." In *MICRO-40*, 2007.
[35] M. Ghosh and H.-H. S. Lee. "Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3D Die-Stacked DRAMs." In *MICRO-40*, 2007.
[36] S. I. Hong, S. A. McKee, M. H. Salinas, R. H. Klenke, J. H. Aylor, and W. A. Wulf. "Access order and effective bandwidth for streams on a direct Rambus memory." In *HPCA-5*, 1999.
[37] C. Natarajan, B. Christenson, and F. Briggs. "A study of performance impact of memory controller features in multi-processor server environment." In *WMPI*, 2004.
[38] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. "Fair queuing memory systems." In *MICRO-39*, 2006.
[39] N. Rafique, W.-T. Lim, and M. Thottethodi. "Effective management of DRAM bandwidth in multicore processors." In *PACT*, 2007.
[40] J. Shao and B. T. Davis. "A burst scheduling access reordering mechanism." In *HPCA-13*, 2007.
[41] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. "POWER5 system microarchitecture." *IBM Journal of Research and Development*, 49(4/5):505-521, 2005.
[42] G. Tesauro et al. "Online resource allocation using decompositional reinforcement learning." In *Conference for the American Association for Artificial Intelligence*, July 2005.
[43] D. Vengerov and N. Iakovlev. "A reinforcement learning framework for dynamic resource allocation: First results." In *ICAC*, 2005.
[44] Z. Zhang et al. "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality." In *MICRO-33*, 2000.