Supporting Transparent Thread Assignment in Heterogeneous Multicore Processors Using Reinforcement Learning

Xiaolei Yan†, Lina Sawalha‡, Amy McGovern† and Ronald D. Barnes‡
†School of Computer Science
‡School of Electrical and Computer Engineering
The University of Oklahoma
Norman, Oklahoma 73019
{yanxiaolei, lina_sawalha, amcgovern, ron}@ou.edu

***Abstract*—Heterogeneity in multicore processor systems creates challenges in effectively mapping processes to diverse cores. While most approaches require programmer partitioning between core types or permutation of thread schedules to find the optimal mapping, we introduce a new machine learning approach to automated thread assignment. We train a reinforcement learning agent to assign threads to the best performing core given the state of the program and the processor cores. We present preliminary results demonstrating the promise of this approach for two and four heterogeneous cores using multiprogram workloads from SPEC CPU2006 benchmarks. We further discuss the limitations of this initial approach and propose future directions for improving our technique.**

I. Introduction

Heterogeneous multicore processors (HMPs) have performance and power advantages over homogeneous ones—a large number of energy efficient cores can be combined with a smaller number of high performance cores to create a multicore system that offers a balance between the conflicting demands of power efficiency and performance, and between thread-level parallelism and single-thread performance. Single-instruction-set architecture (ISA) HMPs offer an additional benefit that they enable the same application code to execute on cores of different types. Characteristics of asymmetric cores result in different performance and power consumption for a particular application. Since the behavior of typical application changes over time, a different type of core may result in better performance during each phase of an application's execution. Unfortunately, asymmetry between cores significantly complicates scheduling threads among the available cores.

In this paper, we present a novel approach for transparently scheduling threads to cores using a reinforcement learning (RL) technique. Reinforcement learning is a machine learning technique that is known as an effective solution for complex resource allocation problems, especially in computer systems [1], [2], [3], [4]. Results from previous work show that RL has great potential as a powerful scheduling method for computer architecture optimization tasks [5], [6]. For example, McGovern et al. [7] built a basic block instruction scheduler that generates higher performance code than a commercial scheduler. Ipek et al. [8] presented an RL-based scheduler in their self-optimizing memory controller. Their results showed that the controller significantly improves the performance of parallel applications on chip multiprocessors through optimized DRAM bandwidth utilization. Coons et al. [9] used RL to optimize distributed instruction placement policy in a compiler and achieved positive results similar to the finest hand-tuned heuristics.

An RL agent attempts to learns the optimal policy through interaction with its environment. More precisely, the agent learns to choose the action in the current state that will lead to the best cumulative rewards over the long-term. An RL algorithm is not a simple greedy algorithm, but an algorithm that can find the policy that guides the agent to get maximum reward over the entire process. The basic RL algorithm was designed for finite Markov decision problems that have a fully observable discrete, finite environment. However, most real world problems are in large, continuous environments. Function approximation is a well-known method for RL to solve more complex problem in such environments. Typical function approximations represent continuous states using simple discretization, radial basis functions, neural networks, decision tree, or instance- and case- based approximations.

In order to adapt to dynamic changes in program behavior, we use an online RL-based scheduling system. Tile coding [10] is used for representing the continuous state of the system. Tile coding is a linear function approximation method that is well suited for efficient on-line learning. Tile coding partitions and groups input space into a new receptive field with binary features. Using easily obtained system features and appropriate reward (performance), the reinforcement learning agent can effectively and dynamically map threads to the dissimilar cores in a chip multiprocessor.

The rest of the paper is organized as follows: Section II describes our RL-based scheduling technique. Section III discusses our learning results and compares the weighted speedup of the RL-based scheduling technique with other

scheduling methods. Finally, Section IV concludes the paper and highlights future work.

II. Supporting Transparent RL-Based Scheduling

Single ISA HMPs, also known as Asymmetric Chip Multiprocessors (ACMPs), are being increasingly examined for their potential to achieve better throughput and energy efficiency [11]. There are several scheduling techniques for assigning jobs to the different types of cores in an ACMP [12], [13], [14]. In each of these techniques, heuristics are used to attempt to find the optimal mapping of application threads to cores of different types. In this work, we demonstrate a more-systematic mapping approach using features that are commonly available via processor performance monitoring.

We simulated two-core and four-core ACMP systems using two different types of core: an out-of-order core (a high-performance core) and an in-order processor (a power-efficient core). Intuitively, the two-core system is composed of one of each of these core types. For our initial study, the quad-core processor consists of one high-performance core and three power-efficient cores. There are two level data caches in each system. Each core possesses a private L1 data cache and all cores in a system share an L2 cache. For the simulated systems, the in-order processor's L1 cache is smaller than the out-of-order processor's L1 cache. The state of these caches will comprise some of the features utilized by our scheduling agent as will be later detailed.

In the experiments presented here, the number of executed applications is the same as the number of cores in the system; there is always one application running on each core. However, our learning-based approach could be augmented to handle time-multiplexing of application processes simply by increasing the number of actions available to the scheduler agent to include swapping a running process for a currently switched-out process. Since typical applications consist of various distinct execution phases [15], [16], dynamically scheduling the applications to run on their respectively best fitting cores each period can result in the best overall system performance.

Reinforcement learning methods are machine learning methods that find a global optimization policy by maximizing the rewards over the long run [10]. We specifically choose Q-learning to train the scheduler because Q-learning can find the optional policy through interaction with the environment. In Q-learning, each state is paired with the available actions for that state, and each state-action pair has a value (Q-value) that is compared to the value of other state-actions. An action will be taken from the state-action pair that has the largest Q-value, and the system will enter another state with its available actions. Every time an action is taken, a reward will be given based on the performance measurement, and then the Q-value of the current state-action pair is updated. Since performance is the inverse of execution time, the fewer the number of cycles used for the same set of executed application instructions, the higher the system performance. The Q-value is updated proportionally according to the given reward. In this way, the agent can learn which actions will lead to the maximum long-term reward. At each decision point, there is also a small chance that the agent will choose a stochastic action, which can help the agent explore alternative schedules to find the maximal reward.

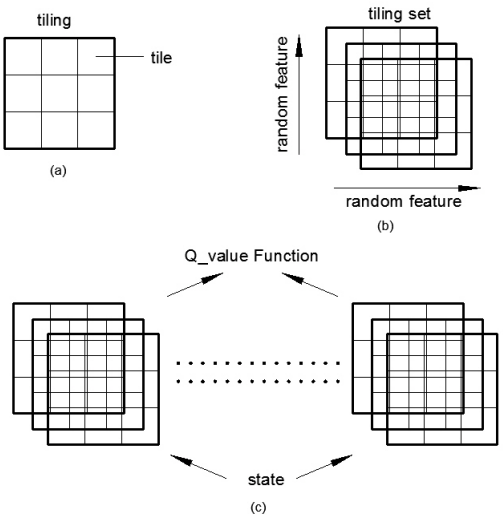


Fig. 1. (a) Individual tiling with 9 tiles. (b) Tiling set with three tilings. (c) State (represented by features) transformed by tile coding to Q-value function input.

In our ACMP systems, the states are represented by the system features that are gathered during execution, and the actions represent the possible thread-to-core assignments. There are 32 features for the two-core system and 64 features for the four-core system. Fifteen different architectural and performance evaluation features are associated with each core and one feature is related to L2 cache hit rate and is associated with each benchmark since all benchmarks share one L2 cache. The fifteen per-core features are the percentage breakdown of executed instruction type (between load, store, branch, multimedia, basic floating point, floating point multiplication, floating point division, basic integer, integer multiplication and integer division instructions), L1 cache hit rate, L2 cache hit rate, percentage of correct branch predictions, percent of available space in reservation stations, reorder buffer and load/store queues (for out-of-order cores). All of these features range from 0 to 1.

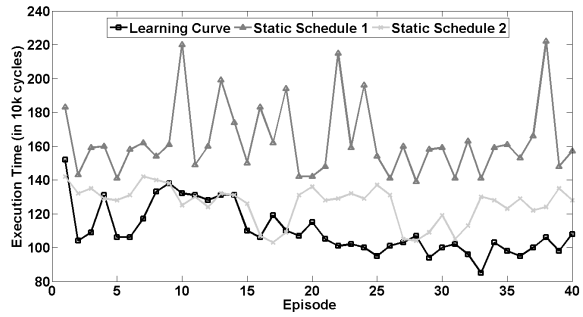
The tile coding approach is used to represent the continuous system environment. Traditional tile coding consists of many tilings, and each tiling represents the whole feature input space. Each tiling randomly divides the input space into grids (tiles). Figure 1(a) shows the state space with a tiling divided according to the feature values. The number of tiles depends on the number of state features and the fineness of the state representation. The number of tiles increases exponentially with the number of features. Since a tile represents a range that a feature can take on, each tile is a binary feature. This means if state features are within certain ranges, the corresponding tile is set to one and other tiles in the same tiling are set

to zero. Only one tile in a single tilling can be set to one depending on how the tiles are divided and the values of incoming state features. Tiles from different tilings can overlap with each other. The finer each state feature is discretized, the better tile coding represents the continuous system. In this way, state features are transformed into binary tiles, each tile has weights that are associated with the available actions, and together they form the linear functions for each action. In our system, the large number of state features inhibits the use of a full tile coding. Instead, we create the tile coding using randomization theory [17]. Rather than using all features on each tiling, we randomly choose two features for three tilings. We call these three tilings a tiling set. Tiling sets are shown in Figure 1(b). 200 and 400 tiling sets are used for the two-core system and four-core system respectively in our experiments. In this way, it is feasible to use tile coding to represent the large, continuous input space.

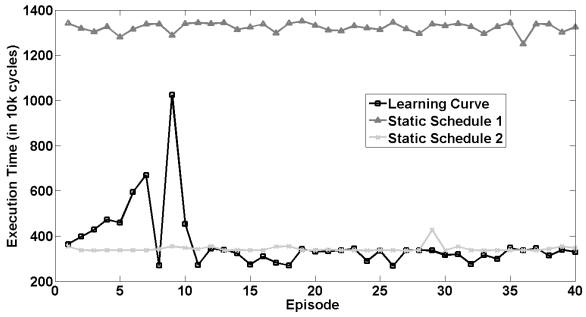
The available actions are the same for every state: schedule one of the benchmarks to run on the high-performance core and others to run on the power-efficient cores. Every 10 000 cycles, the system reports the state features for the agent and continuous features are transformed via randomized tile coding into binary inputs to the linear functions. Figure 1(c) shows the process of transformation from state to Q-value function input. The action with the largest Q-value is taken and a negative one is given to the agent as reward. The linear functions (weights of the functions) are updated according to the reward. Note that since negative reward is given for each 10 000 execution cycles, maximizing the total reward received will maximize system performance by minimizing total execution time. If applications run for a long time due to inefficient scheduling, more cycles will elapse and more negative rewards will be given to the agent. To maximize the reward, the agent must learn to schedule the benchmarks on cores in a way that they can finish faster. Since RL agents are focused on the accumulated reward, the RL agent requires sufficient experience to learn. However, running the entire benchmark for training an agent is impractical for an online scheduler. Fortunately, as long as the agent experiences the important execution phases of an application, training on the entire application is not necessary. The strategy we use is to execute a small, fixed sample of instructions (50 000) and then to skip a larger amount of instructions (29 950 000) until the application completes. The application is restarted from beginning again with same skipping approach. This process is repeated around 50 times to make sure that the agent has sufficient learning experience.

III. Results

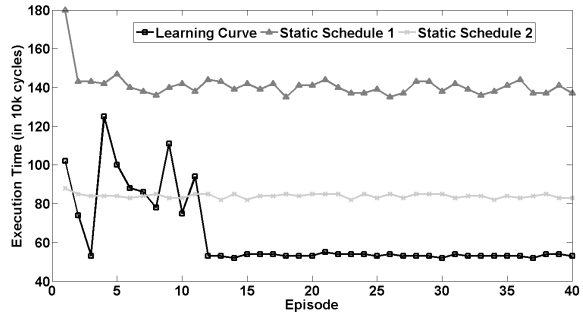
Simulated evaluation of our scheduling technique was performed using Soonergy [18], a cycle-accurate architectural simulator. The scheduler agent is trained using tile coding in these simulated experiments.Thirteen different benchmarks from SPEC CPU 2006 benchmark suite are used in our simulations. These benchmarks are *astar, bzip2, dealII, gcc, gobmk, hmmer, lbm, mcf, namd, omnetpp, povray, soplex*, and *xalan*. The total run of each benchmark was limited to 215 million instructions chosen for their statistical relevancy using the approach in [19].



(a) *bzip2, gobmk*



(b) *lbm, dealII*



(c) *namd, povray*

Fig. 2. The two-core system learning results compared with the static assignments.

Typically, our agent's performance starts near random and improves rapidly during learning. To examine the improvement in system performance as the agent learns, we plot the number of execution cycles (10 000 per unit) that it takes for the agent to complete a full set of execution samples for the pair (or 4-tuple) of benchmarks. Since the plotted measurement is execution time, the best learning agent will have the lowest plotted point on the graph representing the fastest performing schedule.

Figure 2 shows the learning curve of several learning-agent scheduled benchmarks compared to the two static thread-to-core assignments. Each point on the learning curve indicates the total number of cycles executed during that episode. The

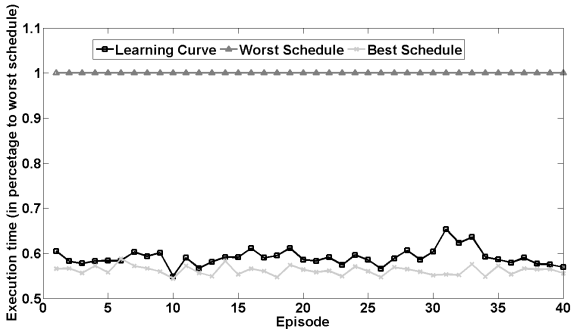


Fig. 4. The average execution time of RL-based scheduling on the two-core system compared to the best and worst assignments.

declining trend of the learning curve means that Q-learning is quickly learning a good schedule for the benchmarks pairs. From these results, we can see that the RL-based dynamic mapping finds a better schedule than any static schedule. Since, in this two-core example, there is only two actions—either swap threads between cores or not, learning typically happens rapidly. However, the learning curve is not monotonically declining; in some experiments the execution time per episode can be observed to increase and then decrease again. This is due to exploration by the learning agent. To obtain global optimization, exploration helps to find the best long run reward. However, exploration can also take a suboptimal action which can result in a worse schedule, which is reflected by increasing the learning curve. Since the exploration rate decreases during learning, the Q-values will eventually become stable at what is hopefully the optimal scheduling approach.

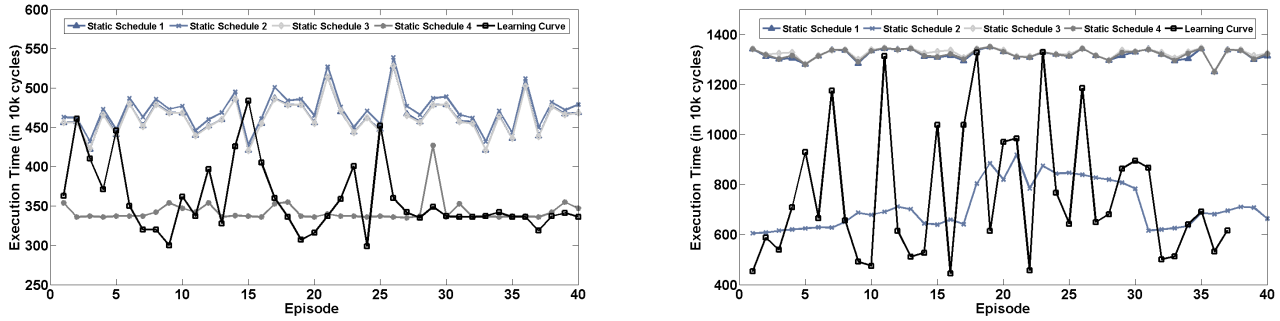
Figure 3 shows two selected learning results from four-core, four-benchmark systems. Comparing with Figure 2, we see that the learning curves of the four-core system oscillate more and decrease slower than the learning curves of the two-core system. The primary reason for this is the complex environment of the four-core system. Since the number of features that are used to represent the system state increases as the number of cores and benchmarks increases, the agent will take more time to learn the optimal schedule.

Figure 4 shows the average learning result from 68 benchmark pairs in the two-core system. The "best" schedule result is generated by comparing two static schedule results sample by sample and the point with the smaller execution time value is considered the best schedule result at that episode. The best, worst and learning results are normalized to the worst schedule result at each episode. In this way, the average learning curve will not be dominated by the results from benchmarks that have long execution times. This yields a curve with the worst schedule at a constant value of one. The learning results and the best static schedule results are thus shown as a percentage of worst static schedule result. Ideally the learning result would tie the best schedule for each point. Figure 4 shows that the RL scheduler, on average, achieves a schedule very near to the ideal (best schedule).

During the experiments, it was also found that in some experiments the Q-learning approach does not effectively schedule the pair or 4-tuple of application benchmarks. These negative results occur much more frequently in the four-core system than in the two-core system. The largest reason for these unsuccessful experiments is the method that was used to approximate the states. If the state space cannot be differentiated finely enough, the Q-value function will not get updated correctly. In this study, features are randomly chosen for each tiling set; some randomly chosen tiling sets represent the state space more than others. It is possible that different sets of features will be more effective for different application benchmarks. We observed that performing multiple runs, with the same system and set of applications but with different random seeds, sometimes achieves very different results. Experiments with negative results can be rerun to produce positive results with the selection of different random seeds and thus different sets of randomly chosen features.

It is well known that features selection is important for RL. Large feature numbers may lead to large state space that even function approximation cannot handle. Ipek et al. use only six features to represent their continuous system states [8]. Within their memory scheduler, these six features are straightforward to select without requiring a specialized selection method. Coons et al. proposed a feature selection method for their RL scheduler based on measurement of their system performance [9]. One potential solution for the discrepancy in the performance of our experiments would be careful selection of features for each tiling set rather than random generation. This indicates our future work direction: finding appropriate feature selection method or changing function approximation to a different approach, possibly a neural network.

In addition to on-line learning during the training of our learning agent, we evaluated the resulting agent for executions of the 215 million instruction sequence for dual-core HMP systems. Figure 5 shows weighted speedup results for a selected set of pairs of benchmark,s from SPEC CPU 2006 suite, of our learning-based scheduling algorithm using tile coding compared with that of the heuristic algorithm in [20] and the two possible static assignments. The weighted speedup represents the summation of the ratio between the performance (in instructions per second) for each process to the single application performance on the best-performing core for each pair of applications. Note that no learning occurs during these evaluations, instead the Q-values learned during the initial experiments for the corresponding pair of benchmarks was used for the full evaluation. For some experiments, our learning-based scheduling algorithm shows significantly higher weighted speedup, for example, *dealII_soplex*. That is because the learning algorithm allows the scheduler to investigate fine-grain changes in program behavior and switch between the different types of cores to maximize system throughput. For some other inputs, the learning algorithm performs slightly better or similar to the other scheduling methods. However, some pairs of applications show that the learning algorithm is penalized by the larger number of reschedules that it performs and results in a worse weighted



(a) *astar, lbm, namd, omnetpp* (b) *astar, dealII, lbm, soplex*

Fig. 3. The learning results compared with the static assignments on a quad-core processor.

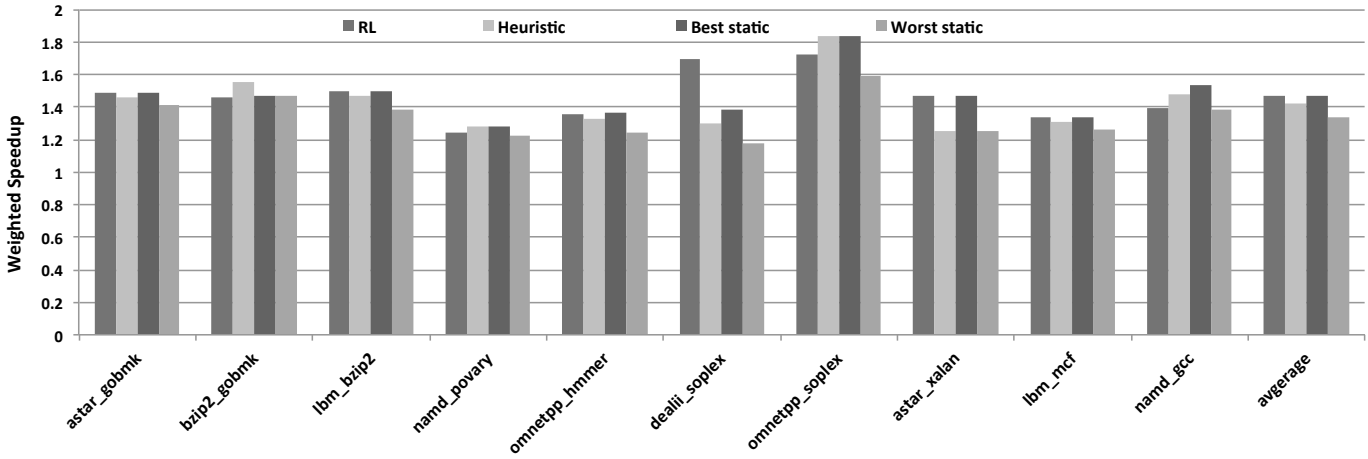


Fig. 5. The weighted speedup of the learning-based algorithm compared with different methods for different pairs of benchmarks.

speedup than the heuristic algorithm and the best static assignment.

While our algorithm attempts to learn the best application-to-core mapping policy based on observed run-time features, the heuristic algorithm in [20] checks for significant changes in program performance, and then samples performance on the different types of cores and chooses the schedule that results in the best weighted speedup. One limitation of this method is that it requires a priori knowledge of performance of each application on the best performing core. However, the reward for our RL approach could have been similarly based on the weighted speedup. This likely would have resulted in trained agents that achieved improved off-line evaluation compared to our current approach. Note that while the weighted speedup metric measures the relative performance of both running applications, the execution time reward penalizes schedules with the longest execution time for the slowest of each pair of benchmarks (as each sample is run until both applications have executed at least 10 000 instructions). Thus, the weighted speedup evaluation is expected to be more favorable if the same metric (weighted speedup) is used for the RL reward.

IV. Conclusion and Future Work

In this paper, we demonstrated a reinforcement learning based scheduling approach that is effective at thread-to-core assignment in HMPs. Plots of reward (execution time per episode) over time show that the RL-based scheduling agent improves its scheduling policy over time, typically achieving schedules that outperform any static schedule in our initial experiments. Results of number of pairs of SPEC CPU2006 benchmarks show that, on average, the on-line learning agent finds schedules that have performance close to the best dynamic assignment for each episode.

Comparing the offline performance of trained agents against previously published heuristics shows results that were somewhat mixed. Often, the RL-produced scheduling agents yield schedules with higher weighted speedups than the heuristics or scheduler or any static schedule. However, in other cases the performance of the learned scheduling policies falls short of the comparison schedules. It should be noted, however, that while our learning agents were trained to reduce the time for all running applications to complete, the heuristic approaches directly base their scheduling decisions on a weighted speedup measured at runtime which requires a priori knowledge of the

performance of each application running alone.

The randomized approach to forming tiling sets used in our preliminary experiments was effective but has some drawbacks. Occasionally the performance of the resulting scheduling policy will depend on the random seed used to select the features for tiling sets. This becomes even more evident as the number of features grows with larger multicore systems. We are currently focusing on improving the learning speed and stability of our RL-based approach. A better understanding of the relationship between the features and their relative importance will result in an improved, deterministic selection of tiling sets. We are also investigating the use of an improved function approximation using neural networks. In addition, we are extending our experiments beyond quad-core systems to demonstrating the efficacy of this approach for future, large multicore architectures.

ACKNOWLEDGMENT

This research was supported by the National Science Foundation under Grant No. NSF-CSR-1018771. The authors would like to thank the members of the Soonergy Computer Architecture Group for their work on the Soonergy Simulation Framework.

REFERENCES

[1] Y. Chang, T. Ho, and L. Kaelbling, "Mobilized ad-hoc networks: A reinforcement learning approach," in *International Conference on Autonomic Computing*, June 2004, pp. 240–247.

[2] J. Nie and S. Haykin, "A Q-learning-based dynamic channel assignment technique for mobile communication systems," vol. 48, no. 5, pp. 1676–1687, 1999.

[3] G. Tesauro, "Online resource allocation using decompositional reinforcement learning," in *Proceedings of the 20th National Conference on Artificial Intelligence*, 2005.

[4] D. Vengerov, "Adaptive utility-based scheduling in resource-constrained systems," in *Australian Joint Conference on Artificial Intelligence, Proceedings of Lecture Notes in Computer Science*, December 2005, pp. 477–488.

[5] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn, "Evidence-based static branch prediction using machine learning," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 1, pp. 188–222, Jan. 1997.

[6] D. Vengerov, H. Berenji, and A. Vengerov, "Adaptive coordination among fuzzy reinforcement learning agents performing distributed dynamic load balancing," in *Proceedings of the 2002 IEEE International Conference onFuzzy Systems*, May 2002, pp. 179–184.

[7] A. McGovern, E. Moss, and A. G. Barto, "Building a basic block instruction scheduler with reinforcement learning and rollouts," *Machine Learning*, vol. 49, no. 2/3, pp. 141–160, November-December 2002.

[8] E. Ipek, O. Mutlu, J. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *Proceedings of the 35th Annual International Symposium on Computer Architechture*, June 2008, pp. 39–50.

[9] K. Coons, B. Robatmili, M. Taylor, B. Maher, D. Burger, and K. McKinley, "Feature selection and policy optimization for distributed instruction placement using reinforcement learning," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008, pp. 32–42.

[10] R. Sutton and A. Barto, *Reinforcement learning: An introduction*. The MIT Press, 1998.

[11] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, June 2003, pp. 81–92.

[12] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *Proceedings of the 5th European Conference on Computer systems*, April 2010, pp. 125–138.

[13] J. C. Saez, A. Fedorova, M. Prieto, and S. Blagodurov, "A comprehensive scheduler for asymmetric multicore systems," in *Proceedings of the 5th European Conference on Computer systems*, April 2010, pp. 139–152.

[14] L. Sawalha, M. P. Tull, and R. D. Barnes, "Thread scheduling for heterogeneous multicore processors using phase identification," *ACM Performance and Evaluation Review*, March 2012.

[15] M. Merten, A. Trick, R. Barnes, E. Nystrom, C. George, J. Gyllenhall, and W. Hwu, "An architectural framework for runtime optimization," *IEEE Transactions on Computers*, vol. 50, no. 6, pp. 567–589, June 2001.

[16] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003, pp. 336–347.

[17] I. H. Witten, E. Frank, and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*, 3rd ed. Morgan Kaufmann, 2011.

[18] N. Freeman, "Soonergy: A pluggable, cycle-accurate computer architecture simulator," Master's thesis, The University of Oklahoma, 2011.

[19] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002, pp. 45–57.

[20] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-ISA heterogeneous multi-core architectures for multithreaded workload performance," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004, pp. 64–75.