UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

REINFORCEMENT LEARNING SCHEDULER FOR HETEROGENEOUS MULTI-CORE PROCESSORS

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

XIAOLEI YAN
Norman, Oklahoma
2013

REINFORCEMENT LEARNING SCHEDULER FOR HETEROGENEOUS MULTI-CORE PROCESSORS

A THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY

Dr. Amy McGovern (Chair)

Dr. Ronald D. Barnes

Dr. Andrew H. Fagg

Acknowledgments

I would like to my advisors Dr. Amy McGovern and Dr. Ronald Barnes for their help and support. I would also like to thank every member in Soongy lab and idea lab. Specifically, I would like to thank Lina Sawalha and Sonya Wolff, they helped me a lot by answering my technique questions about the simulator. And also, my work is largely based on Lina's work.

I would like to thank my parents for their support and understanding.

This material is based upon work supported by the National Science Foundation under Grant No. NSF-CSR–1018771.

Table of Contents

List Of Tables

List Of Figures

Abstract

Asymmetric multiprocessor systems provide opportunities for exploiting differences between differernt programs to achieve power efficiency and performance. Unfortunately, heterogeneity in multicore processor systems creates significant challenges in effectively mapping these programs to diverse type of cores. To avoid requiring a static assignment of an application to a particular core type by the programmer, various approaches have been proposed to dynamically schedule applications across cores with heterogeneous sets of capabilities. Typical scheduling approaches require either sampling through a permutation of thread schedules to find the optimal mapping or use rough heuristics for predicting the performance of an application on a particular core type. We instead introduce a new, systematic approach to automate thread assignment. We construct a reinforcement-learning-based scheduler to assign threads to the best performing core given the state of the program and the processor cores. We use tile coding and artificial neural networks(ANNs) to represent system features as states and explore linear and nonlinear relationships between states and performance estimation. We present results demonstrating the promise of this approach for single-ISA heterogeneous multicore processors using multiprogram workloads from SPEC CPU2006 benchmarks. Our initial tile coding based function approximation learning experiments are encouraging, and our reinforcement-learning-based scheduler with ANNs function approximator delivers 1.77%(*ignoring the overhead of switching*) and 4.1%(*considering the overhead of switching*) better performance and 41.4% (*ignoring the overhead of*

switching) and 41.7%(*considering the overhead of switching*) improvement towards an ideal performance estimation over heuristic-sampling-based scheduler and 6% better performance than the average of all possible static schedules for a four-core heterogeneous system. We also show the great potential of this approach using either online or offline learning. Finally, we discuss the implementation of our model, its impact on results and propose future directions for improving the reinforcement-learning-based scheduling approach.

Chapter 1

Introduction

Heterogeneous multicore processors (HMPs) have performance and power advantages over homogeneous ones—for example, 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 (or asymmetric multicore processors) offer an additional benefit that they enable the same application code to execute on cores of different types without any recompilation (Kumar et al. 2003a). The performance characteristics of the individual types of asymmetric cores result in different performance and power consumption for a particular application. Since the behavior of a typical application changes over time, a different type of core may result in better performance during each phase of an application's execution (Sawalha et al. 2011). Asymmetric multicore processors make increasing sense in the "dark silicon era" when the vast numbers of (mostly powered-off) transistors on a die can be used to implement cores that are specialized for specific phases of execution (Patsilaras et al. 2012). Unfortunately, asymmetry between cores significantly complicates scheduling threads among the available cores.

In this study, we present a novel approach for transparently scheduling threads to cores using reinforcement learning (RL). Reinforcement learning is an

effective solution for complex resource allocation problems, especially in computer systems (Nie and Haykin 1999; Tesauro 2005). Results from previous work show that RL has great potential as a powerful scheduling method for computer architecture optimization tasks. For example, McGovern et al. (2002) built a basic block instruction scheduler that generates higher performance code than a commercial scheduler. Ipek et al. (2008) presented an RL-based memory 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. (2008) 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 approximates 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. 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 approach for RL to solve more complex problems in such environments. 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.

Typical function approximations represent continuous states using simple discretization, radial basis functions, neural networks, tile coding, decision tree, or instance- and case-based approximations (Sutton et al. 2000). Tiling coding and Artificial Neural Networks (ANNs) are tested as function approximation

method to represent the continuous state of the system in this research. Tile coding uses linear functions to approximate nonlinear relationship. ANNs have a network structure which itself represents a nonlinear function. The advantages and disadvantages of tile coding and ANNs are discussed in the later chapters.

The rest of the thesis is organized as follows: Chapter 2 describes the basic algorithms including reinforcement learning, tile coding, and ANNs and specifies the implementations in heterogeneous multicore system. Chapter 3 and Chapter 4 represent empirical results for tile coding and ANNs implementations separately. Comparisons and implementation details are discussed in each chapter. Finally, Chapter 5 concludes the thesis and highlights future work.

Chapter 2

Reinforcement Learning and Function Approximation

2.1 Reinforcement Learning

2.1.1 Overview and Simple Example

Reinforcement learning(RL) is one of the important machine learning algorithms. RL origins from the combination of optimal control (Bellman 1957a,b), psychology of animal learning (Thorndike and Bruce 1911), and temporal difference methods (Sutton 1988). The RL agent learns the right thing to do in each situations that it could met in a dynamic environment. By exploring different action spaces and getting feedback from the environment, the agent learns which policy can lead to a good result in the long run. In the other word, the RL agent can learn from its own experience with the system. Although RL is a learning algorithm, it is different from the general supervised learning such as decision tree, linear regression, and neural networks. In the supervised learning algorithm, the right answers are provided. The training data are input/output pairs. And the supervised learning model adjusts its parameters or structures to generate similar outputs from the same inputs. Unlike the supervised learning, the RL agent learns from the feedback by interacting with the environment, and

RL aims to find an optimal policy for the big picture. Therefore, RL is designed for certain type of tasks that require searching for possible optimal results in a dynamic system.

A RL system should have at least four elements (Sutton and Barto 1998): a policy, a reward function, a value function, and a model of environment. During the learning, the RL agent has to choose which action to be taken in every state. In a dynamic system, available actions may change when the RL agent reach different states. The policy represents the rules that define the different action sets for every possible states. The reward function is a function that can transform the target performance to a quantity number, so that the performance can be evaluated in a mathematical equation. The value function is used by the RL agent for remembering the consequence of taking an action in the state mathematically. And the next time when the RL agent reaches the visited state , it will make a decision based on the evaluation of the output of the value functions of that state. In this way, the RL agent can learn to approximate the optimal solution from its own experience with the system. The last element is the model of environment. In the RL tasks, all other elements are designed and implemented upon the system model. So, the learned results from the RL agent are specific to the system. The model could be a real dynamic system or a system simulation model.

To better understand the RL systems/tasks, a simple example is presented. Travel in a grid world is a classical RL task. As shown in Figure 2.1, the agent want to travel from the start position to the goal position using minimum steps. Each intersection point of two lines is a discrete state in the system. Depends on the current state in the grid world, the agent may move up, down, left, and right to reach the goal. The policy in this task could be the choices of actions

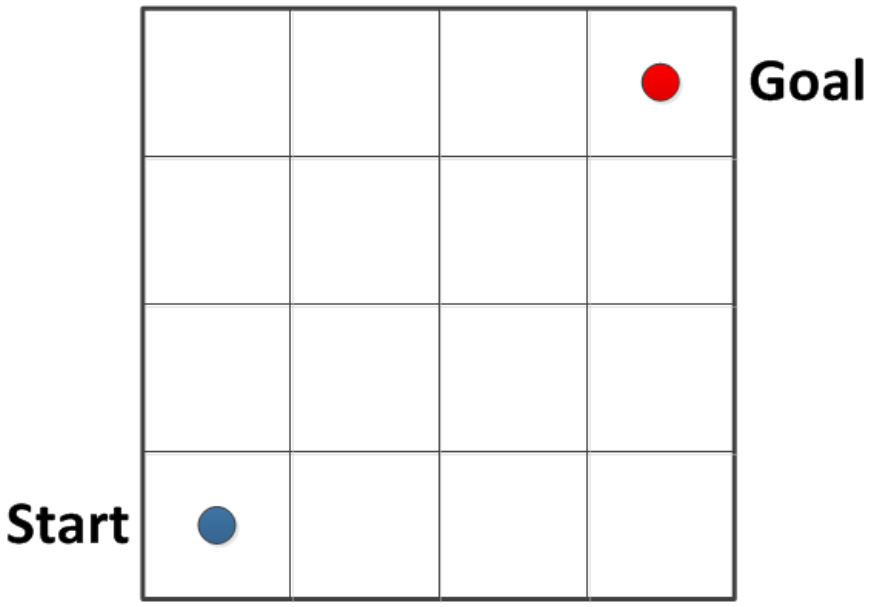


Figure 2.1: *System set up for grid world.*

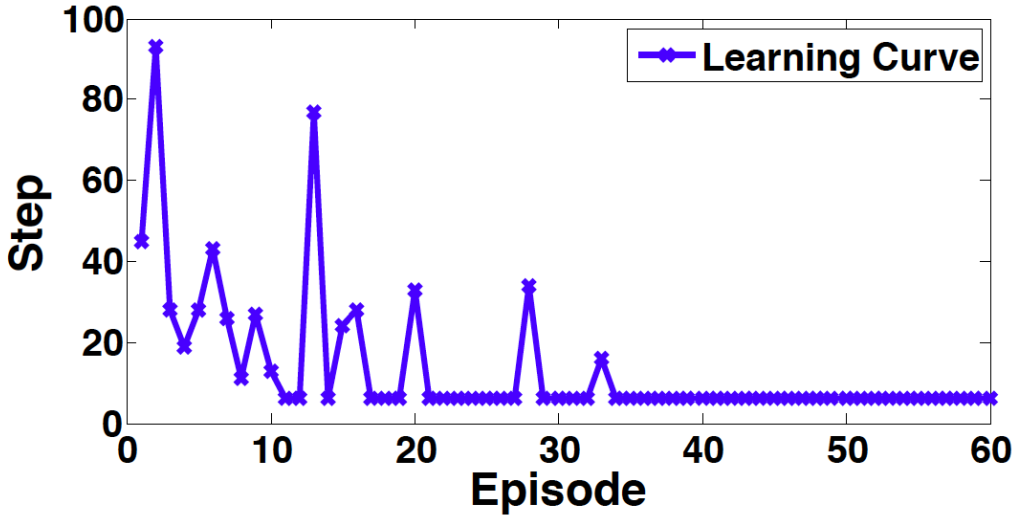


Figure 2.2: *The learning curve for the grid world task.*

to be taken in different states. For example, the agent can not move left if the goal position is on its right. The reward function could be negative one for each step. The more steps the agent takes to reach the goal, the more negative ones

it receives. The value functions for each state maintain and update the expected total received reward for each available action in that state. The grid world is a simple system that can be simulated in program easily. A RL agent aims at getting more accumulated reward which means less negative ones in this task. Figure 2.2 is a learning curve of grid world task which is used to show learning progress and learning results. The x axis is the episode number. The agent take one episode to finish the task which, to be specific, is reaching the goal position from the start position. The y axis is how many steps the agent take to finish the task for each episode. We can see that the number of steps decrease during the learning process and reach the minimum steps that are required to finish the task. Since the travel in the grid world task is really simple, the optimal result can be easily learned. For large and complex systems and tasks, the best optimized result may not be guaranteed, but the RL is aiming at approaching it.

2.1.2 Basic RL Equation

Basically, RL takes the trail and error process with memorizing the consequences in a dynamic environment. The algorithm expression of RL uses the terms of the dynamic state and value function which comes from the optimal control. The Monte Carlo method and the temporal difference methods finalize the equation. Figure 2.3 shows the evolution process of the RL value return and update equation.

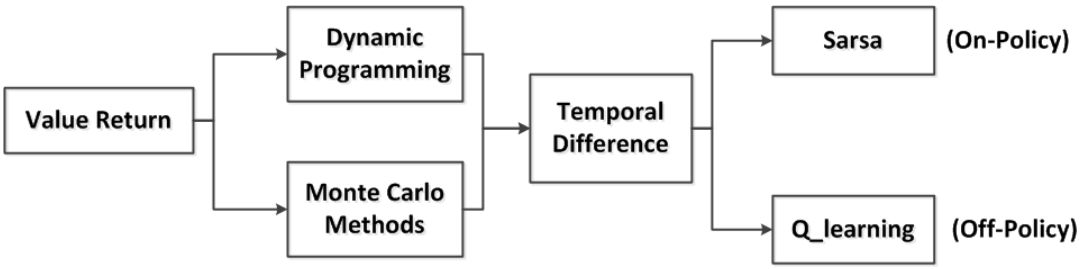


Figure 2.3: *Brief history of RL algorithm development.*

As described above, the RL agent is designed to get the highest accumulated reward. The accumulated reward can simply expressed as the addition of the acquired reward from every step in the policy, like (2.1).

$$\mathbf{R_t} = \mathbf{r_{t+1}} + \mathbf{r_{t+2}} + \mathbf{r_{t+3}} + \cdots \tag{2.1}$$

In case, the RL task is continuous which means there are infinite steps in one episode, discount rate(γ) is introduced to the (2.1) and the equation becomes (2.2).

$$\mathbf{R_t} = \mathbf{r_{t+1}} + \gamma\mathbf{r_{t+2}} + \gamma^2\mathbf{r_{t+3}} + \cdots = \sum_{k=0}^{\infty} \gamma^k \mathbf{r_{t+k+1}} \tag{2.2}$$

When γ=0, the RL agent only consider the reward from the next step like other greedy agents do. As γ approaches 1, the RL agent will consider more about future rewards and becomes more farsighted. In general, due to the discount rate, the near future rewards, like the next several steps rewards, has more impact on the decision than the rewards from the further future rewards.

In order to further develop the value function equation, we have to introduce the Markov decision process(MDP). The MDP is a simple class of RL tasks that have the Markov property. The Markov property defines that the distribution of the probabilities of the next state and reward are only depend on the current state and action. The MDPs with finite states and actions are called finite

MDPs. In finite MDPs, the probability of each possible next state is expressed as (2.3)

$$\mathcal{P}^{a}_{ss'} = \mathbf{Pr}\left\{\mathbf{s_{t+1}} = \mathbf{s}^{'} | \mathbf{s_t} = \mathbf{s}, \mathbf{a_t} = \mathbf{a}\right\} \tag{2.3}$$

and the expected next reward is expressed as (2.4).

$$\mathcal{R}^{a}_{ss'} = \mathbf{E}\left\{\mathbf{r_{t+1}} \middle| \mathbf{s_t} = \mathbf{s}, \mathbf{a_t} = \mathbf{a}, \mathbf{s_{t+1}} = \mathbf{s}^{'}\right\} \tag{2.4}$$

The $\mathbf{s}$, $\mathbf{a}$, and $\mathbf{t}$ represents the state, the action, and the step. The $\mathbf{s}^{'}$ is the next possible state. Then, combined with (2.2), the value function in finite MDPs became (2.5).

$$\mathbf{V}^{\pi}(\mathbf{s}) = \sum_{\mathbf{a}} \pi(\mathbf{s}, \mathbf{a}) \sum_{\mathbf{s}'} \mathbf{P}^{\mathbf{a}}_{\mathbf{ss}'} \left[\mathcal{R}^{a}_{ss'} + \gamma \mathbf{V}^{\pi}(\mathbf{s}^{'})\right] \tag{2.5}$$

The π is the policy and the $\pi(\mathbf{s}, \mathbf{a})$ is the available actions in state $\mathbf{s}$ defined by the policy π. By comparing all the value functions of all the available policies, the optimal policy $\mathbf{V}^{*}(\mathbf{s})$ is the one with the largest $\mathbf{V}^{\pi}(\mathbf{s})$.

Dynamic programming(DP) helps RL to move forward a further step. By using the (2.5), the value function of a policy can be calculated recursively in MDPs. This characteristic of the policy value function makes DP method applicable in this field. Rather than calculating and comparing value functions of all possible policies, DP method can find the optimal policy by policy iteration. Policy iteration consists of two components: policy evaluation and policy improvement. Policy evaluation is actually the same process of calculating current policy value function by using (2.5). Policy improvement uses (2.6)

$$\mathbf{Q}^{\pi}\left(\mathbf{s}, \pi^{'}(\mathbf{s})\right) \geq \mathbf{V}^{\pi}(\mathbf{s}) \tag{2.6}$$

as a reference to update current policy. $\mathbf{Q}^{\pi}(\mathbf{s}, \mathbf{a})$ is a action value function which represents expected reward of taking action a in state $\mathbf{s}$ under policy

π. $\pi'(\mathbf{s})$ in (2.6) means taking an action a that is not defined by policy π in state $\mathbf{s}$. If (2.6) is true, the current policy will be updated to π'. By repeatedly executing the two steps, the original policy can be improved and converged to the optimal policy in MDPs. Although, DP method guarantee policy convergence in MDPs, it requires complete probability distributions and expected rewards of all possible states. The number of state grows exponentially with the number of state variable. Both requirements limit the application of DP method in real world tasks.

Another type of methods that also contribute to the development of RL value function is Monte Carlo methods. Unlike DP, Monte Carlo methods do not require the complete prior knowledge of all states in the system. It only needs the information of every state the agent experienced in every episode. In the other word, Monte Carlo methods try to find the optimal policy by learning from its own experience in the system. This means the agent has to go through the whole task (episode) repeatedly. Hence, Monte Carlo methods require that the RL task is restrictively episodic. Usually, the return used to update the value function is the average of all the instant rewards from experienced states. Monte Carlo methods also use policy and value iteration to improve current policy to the optimal one. One popular off-policy action selection strategy that is used in policy improvement and is worthwhile to be mentioned is the ε-greedy policy. The ε-greedy policy is the agent chooses the action with the maximal estimated action value (or the action leads to the state with the maximal estimated state value) with the probability of 1-ε, but choose a random action with the probability of ε. This ε-greedy policy will also be used in later developed RL algorithm.

It is not efficient or even not possible to use dynamic programming and Monte Carlo methods for complex tasks. Temporal Difference (TD) (Sutton 1988) learning combines the ideas of DP and Monte Carlo methods and makes a breakthrough on the RL application limitations. TD learning takes the Monte Carlo methods idea of learning from experience and the DP idea of learning from the other learned estimations. In this way, the learning agent neither has to know the prior knowledge of the whole dynamic system nor waits until the task finishes to get reward returns. TD learning uses bootstrap method to get update that is based on immediate reward and an existing estimation. The basic equation of TD learning is (2.7),

$$\mathbf{V}(\mathbf{s_t}) \longleftarrow \mathbf{V}(\mathbf{s_t}) + \alpha\left[\mathbf{r_{t+1}} + \gamma\mathbf{V}(\mathbf{s_{t+1}}) - \mathbf{V}(\mathbf{s_t})\right] \tag{2.7}$$

where α is the learning rate, $\mathbf{r_{t+1}}$ is the immediate reward, and $\mathbf{V}(\mathbf{s_{t+1}})$ is the value estimation of the next state. $\mathbf{r_{t+1}} + \gamma\mathbf{V}(\mathbf{s_{t+1}}) - \mathbf{V}(\mathbf{s_t})$ is called TD error σ. When σ approaches 0, it means the value function converges and the policy has been updated to an optimal one.

There are two TD based methods that are mostly used in RL tasks: Sarsa and Q-learning. Sarsa substitutes the state value function in the basic TD learning equation with action value function and has the form of (2.8).

$$\mathbf{Q}(\mathbf{s_t}, \mathbf{a_t}) \longleftarrow \mathbf{Q}(\mathbf{s_t}, \mathbf{a_t}) + \alpha\left[\mathbf{r_{t+1}} + \gamma\mathbf{Q}(\mathbf{s_{t+1}}, \mathbf{a_{t+1}}) - \mathbf{Q}(\mathbf{s_t}, \mathbf{a_t})\right] \tag{2.8}$$

A value(Q-value) is associated with the action value function $\mathbf{Q}(\mathbf{s}, \mathbf{a})$ and is used and updated to represent the expected reward return in the long run. The $\mathbf{Q}(\mathbf{s}, \mathbf{a})$ means an action is paired with the state, and there could be as many available actions as $\mathbf{Q}(\mathbf{s}, \mathbf{a})$s in a state. Sarsa is a on-policy TD control method. This means $\mathbf{s_{t+1}}$ in the $\mathbf{Q}(\mathbf{s_{t+1}}, \mathbf{a_{t+1}})$ in (2.8) is the next state of s in $\mathbf{Q}(\mathbf{s_t}, \mathbf{a_t})$ by taking action $\mathbf{a_t}$. The $\mathbf{a_{t+1}}$ is determined by ε-greedy policy. Then, in the

next step, $\mathbf{s_{t+1}}$ in the last step becomes current $\mathbf{s_t}$ and $\mathbf{a_{t+1}}$ become $\mathbf{a_t}$. The method evaluates one policy and updates the action value function following the same policy. Q-learning is very similar to Sarsa expect it is off-policy TD control method. The equation of Q-learning is (2.9).

$$\mathbf{Q}(\mathbf{s_t}, \mathbf{a_t}) \longleftarrow \mathbf{Q}(\mathbf{s_t}, \mathbf{a_t}) + \alpha \left[\mathbf{r_{t+1}} + \gamma \max_{\mathbf{a}} \mathbf{Q}(\mathbf{s_{t+1}}, \mathbf{a}) - \mathbf{Q}(\mathbf{s_t}, \mathbf{a_t}) \right] \tag{2.9}$$

From the equation, we can see the action in $\mathbf{Q}(\mathbf{s_{t+1}}, \mathbf{a})$ is not necessary the action that the agent will take. The next state Q-value used to update the current Q-value is the one with largest value rather than the Q-value of actual taken state action pair. This method evaluates one policy while following another policy. Both Sarsa and Q-learning can converge to the optimal policy, but at different rates. Q-learning tends to be a little slower than Sarsa, but is capable of learning while changing the policy. Sarsa is a better online learning algorithm than Q-learning.

2.1.3 RL in HMPs

Single-ISA HMPs are increasingly being examined for their potential to achieve better throughput and energy efficiency (Kumar et al. 2003b). There are several scheduling techniques for assigning jobs to different types of cores in an ACMP. Typically these fall into the categories of either sampling or prediction scheduling techniques. Sampling approaches (Becchi and Crowly 2008; Kumar et al. 2004) permute the set of running programs amongst each type of core to find the best overall schedule. While sampling approaches tend to find an efficient schedule, they waste significant power and performance while sampling through many non-optimal assignments and migrating threads from one core to the other. The cost of sampling grows exponentially with the number of

types of cores as thus does not scale beyond relatively simple HMPs. Prediction approaches (Koufaty et al. 2010; Saez et al. 2010; Van Craeynest et al. 2012) seek instead to estimate the performance of each running program on each type of core without necessarily executing the program on that type of core. These approaches avoid the overhead of sampling but are heavily dependent on their heuristic to estimate the performance of an application, based on its characteristics. These heuristics must be finely tuned for each type of core to achieve highly efficient schedules. Other approaches seek to combine sampling with prediction (Sawalha et al. 2011). These approaches reduce the overhead of sampling but do not eliminate it and do not necessarily find as optimal an application mapping as could be found through more aggressive sampling. In this work, we demonstrate a more systematic mapping approach using features already commonly available via processor performance monitoring.

In our proposed approach, the scheduler is assisted by a learning agent, and the assignments of running programs to cores are the actions available to that agent. The HMP system is the environment that interacts with the scheduler, the behaviors of programs and cores represent states, and the overall system performance is the "reward." As shown in Figure 2.4, for each interval (step), the scheduler chooses one mapping assignment from all available ones, and running programs are mapped to cores in the HMP system based on this assignment. At the next interval, the system reports the performance and the behaviors of programs and cores back to the scheduler. The scheduler updates the estimated accumulated reward for the assignment for the last system state. After a fixed number of iterations, the scheduler learns to choose the appropriate mapping assignment under each circumstance to maximize the accumulated reward over

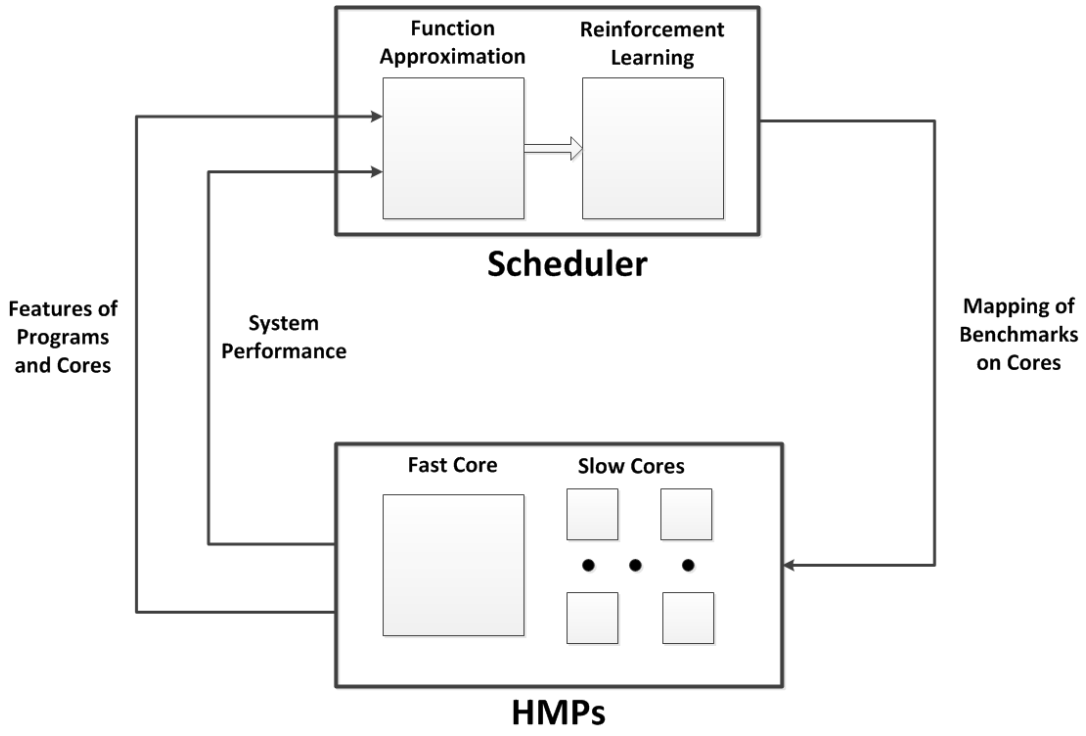


Figure 2.4: *Reinforcement learning process in HMPs.*

the long run. Because characteristics of an HMP system are basically continuous, there are virtually infinite different behaviors of programs and cores and thus they cannot all be recorded. Instead, function approximations are used to represent the relationship between these characteristics and the rewards. These functions take system features as inputs and the estimated accumulated rewards are outputs. Each available assignment is represented as a function. In this way, the scheduler can make same scheduling assignments under the same situations.

2.2 Function Approximation

2.2.1 Tile Coding and Randomization

RL algorithm works based on estimations of value or action value functions of each state or action state pair. For large multi-dimensional continuous dynamic systems, it is simply not possible to find a state without specific divisions or definitions. A generalization or abstraction method is needed for RL to solve tasks in such a system. Hence, function approximation is introduced to the RL task solving procedures. The most straightforward method is dividing the continuous state space into discrete states. It is easy to implement this method but hard to decide how to divide the state space without prior knowledge of the system. This method is adequate for the tasks in small continuous system with limited state variables but may generate contradictory information for the learning agent. Another simple way to do function approximation is linear function method. Linear function method takes the variables of state space as input and uses a linear function to represent the value function or action value function. Updating the state value or Q-value is accomplished by updating the parameters that associate with each state variable in the linear function. The problem of this method is the linear function can only represent linear relationship between the linear set of state variables and the state value or Q-value. Therefore, the linear function method is not capable of handling large complex system.

Tile coding is a function approximation method that uses both discretization and linear function representation techniques. Tile coding is also named cerebellar model articulation controllers(CMACs) and is originally proposed for memory management (Albus 1975). In general, tile coding transforms the state

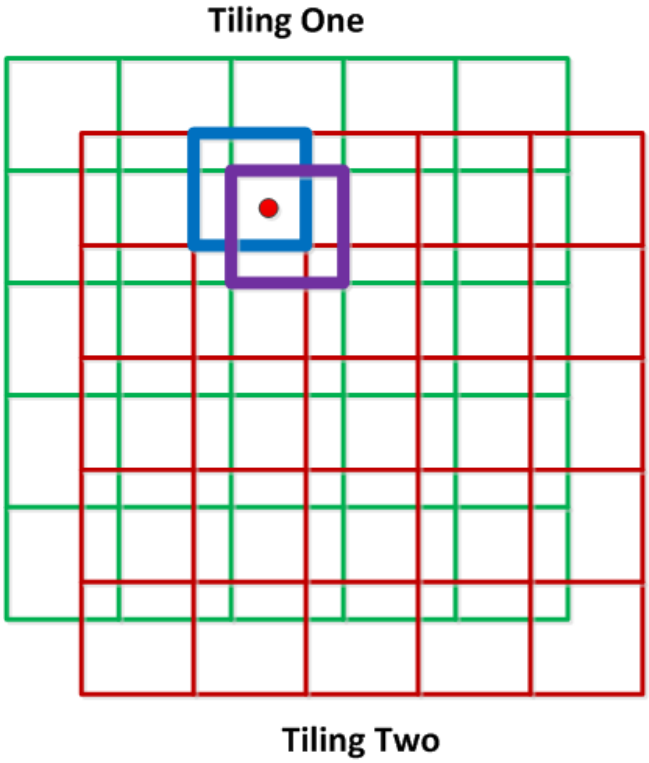


Figure 2.5: *Simple tiling partition in 2D space.*

space into binary features and inputs these features to a linear function which represents the value or action value function. In details, tile coding turn the state space into many distinctive partitions and the partitions are shifted from each other. Each partition of the state space is called a "tiling" and each element or cell on the tiling is called a "tile." When the learning agent reach a state, the state variables are used to identify which tile that the state is falling into in each tiling. Then, the tile will be set to one and all other tiles in the same tiling will be set to zero. Every tiling has exactly one tile to be set to one. All tiles in every tiling are inputs to the linear function where each entry has an associated weight. Figure 2.5 shows a simple example of tile coding. In a simple 2-D state space, tiling one and tiling two are the same state space with

different partitions. The point in the figure represents the current state. The highlighted tiles with the point in them are set to one and other tiles are set to zero. Then, these transformed binary features are inputs for the linear function like (2.10).

$$\mathbf{f}\left(\phi\left(\mathbf{s}\right),\mathbf{w}\right)=\sum_{\mathbf{i}=1}^{\mathbf{N_L}}\sum_{\mathbf{j}=1}^{\mathbf{N_i}}\phi_{\mathbf{ij}}\left(\mathbf{s}\right)\mathbf{w_{ij}} \tag{2.10}$$

$\phi\left(\mathbf{s}\right)$ are binary features, $\mathbf{w}$ are weights, $\mathbf{N_L}$ is number of tilings, and $\mathbf{N_i}$ is number of tiles in the tiling $\mathbf{j}$. The learning takes place by updating weights using gradient descent method like (2.11).

$$\mathbf{w}\longleftarrow\mathbf{w}+\alpha\left(\mathbf{r}+\gamma\mathbf{f}\left(\phi\left(\mathbf{s}'\right),\mathbf{w}\right)-\mathbf{f}\left(\phi\left(\mathbf{s}\right),\mathbf{w}\right)\right)\phi\left(\mathbf{s}\right) \tag{2.11}$$

Tile coding generates finer state representation than simple discretization and can handle a little more complex system than simple linear function method do.

In our HMPs system, the large number of state features inhibits the use of a full tile coding. Instead, we create the tile coding using randomization theory (Witten et al. 2011). Rather than using all features on each tiling, we randomly choose two features for three tilings. Since this is an initial experiment, we did not examine more possible random tiling formation methods. We call these three tilings a "tiling set." We approximate the high dimensional state space by forming a large number of tiling sets. Figure 2.6(c) shows the process of transformation from state to Q-value function input.

2.2.2 Artificial Neural Networks

Tile coding may not sufficient for tasks in large complex system. Tile coding does not overcome the limitations of discretization and linear function, and it still has the problem of curse of dimension and can not represent complex non-linear relationship between state variables and value or action value functions.

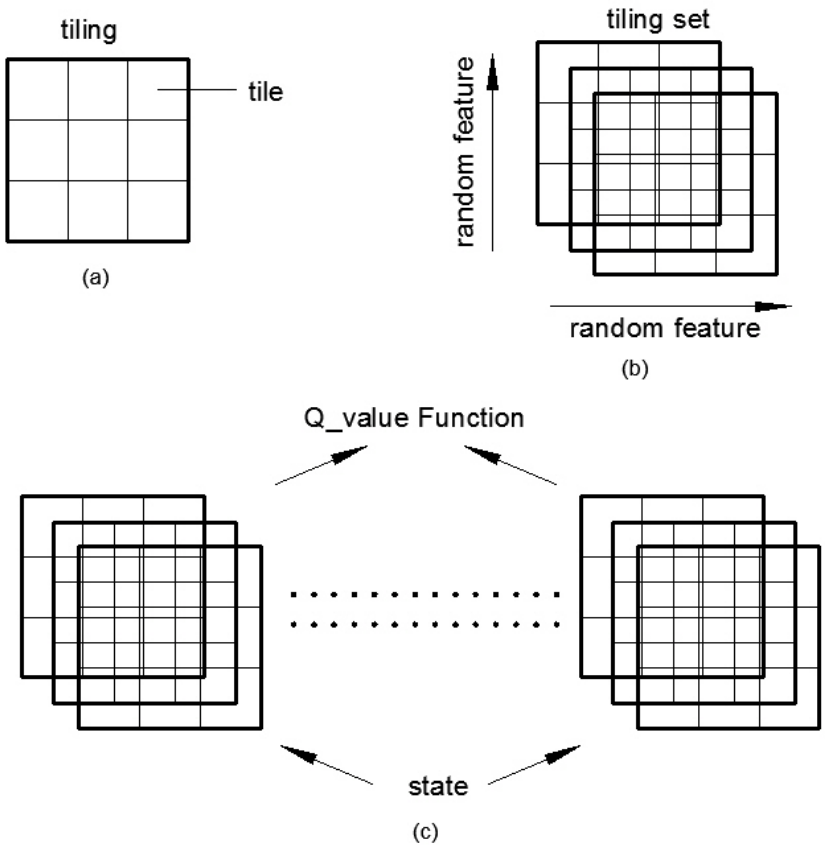


Figure 2.6: *(a) Individual tiling with 9 tiles. (b) Tiling set with three tilings. (c) State (represented by features) transformed by tile coding to Q-value function input.*

Artificial neural network is better a choice over tile coding in real complex environment.

Artificial Neural Networks (ANNs) are highly interconnected networks that are inspired by biology neural networks. ANNs can be trained to a mathematical models that can represent the complex nonlinear relationship between input and output data or data pattern. The mostly used ANN is the multilayer feed forward network which is proposed by Werbos (1974). ANNs can be used as

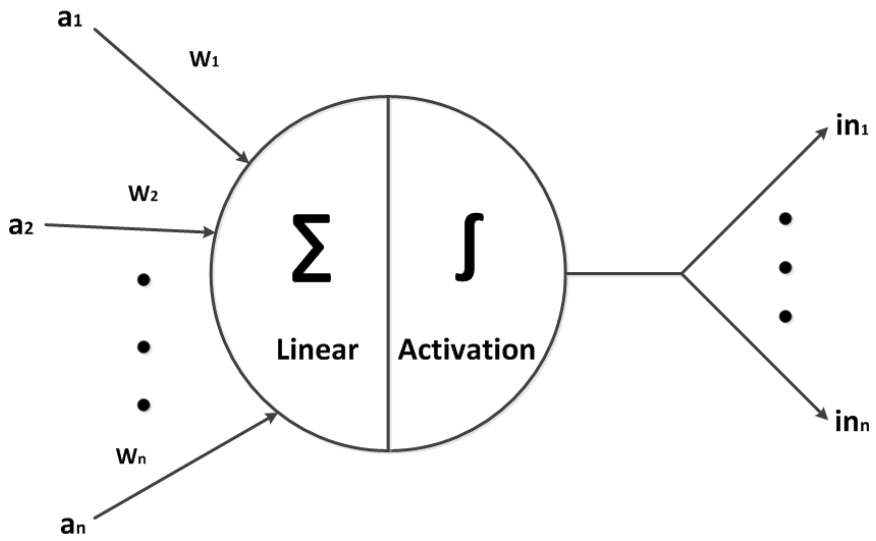


Figure 2.7: *Basic neuron structure in ANNs.*

the function approximation method in RL task (Crites and Barto 1998; Tesauro 1994). It has been proven that ANNs can be a universal function mapping method and approximate multi-variable function with a finite number of nodes in the hidden layer (Hecht-Nielsen 1987). ANNs can have various structures. The basic element in the ANNs is called neuron. Figure 2.7 shows the simple mathematical model for a neuron. There are two steps in the basic neuron model: linear function and activation function. The activation function, such as sigmoid function, takes the output of the linear function as input and forward the results to the next layer neurons. ANNs get updated by back propagation which traces back to every layer and updates the weights using gradient descent method. For the output layer, $\mathbf{w_{jk}} \longleftarrow \mathbf{w_{jk}} + \alpha \times \mathbf{a_j} \times \Delta_k$ is used. The hidden layer $\mathbf{j}$ is directly connected to the output layer $\mathbf{k}$. $\mathbf{a_j}$ is the output from the

layer $\mathbf{j}$ and also the input to output layer $\mathbf{k}$. $\mathbf{w_{jk}}$ is weights that are used for output neuron linear function. Δ_k is expressed as (2.12).

$$\Delta_k = \mathbf{Err_k} \times \mathbf{g}^{'}(\mathbf{in_k}) \tag{2.12}$$

The $\mathbf{Err_k}$ is the difference between the target value and the ANN output. In RL tasks, it is TD error σ. $\mathbf{g}(\mathbf{x})$ is the activation function and $\mathbf{in_k}$ is the output of linear function in layer $\mathbf{k}$. For the hidden layers, (2.13) is used.

$$\mathbf{w_{ij}} \longleftarrow \mathbf{w_{ij}} + \alpha \times \mathbf{a_i} \times \Delta_j \tag{2.13}$$

Δ_j is expressed as (2.14).

$$\Delta_j = \mathbf{g}^{'}(\mathbf{in_j}) \sum_{\mathbf{k}} \mathbf{w_{jk}} \Delta_k \tag{2.14}$$

ANN with multilayers is a good function approximation method for sophisticated RL tasks.

The number of ANNs that are used in a environment depends on the number of actions that are available in that system (two for two-core system and four for four-core system). There are one big core and one small core in the two core system. The two available actions are putting application one on the big core and application two on the small core, and the opposite application core mapping. In the four core system, there are one big core and three identical small cores. Since the three small cores are the same, different application core mappings on the small cores will not cause difference in performance. So there are actually four effective actions in the four core system: putting one of the four different applications on the big core with other applications on the small cores. Each ANN represents one state-action pair. In each system, all ANNs have the same structure. Generally, there are three layers in each ANN including one input layer, one hidden layer and one output layer. As Figure 2.8 shows, the

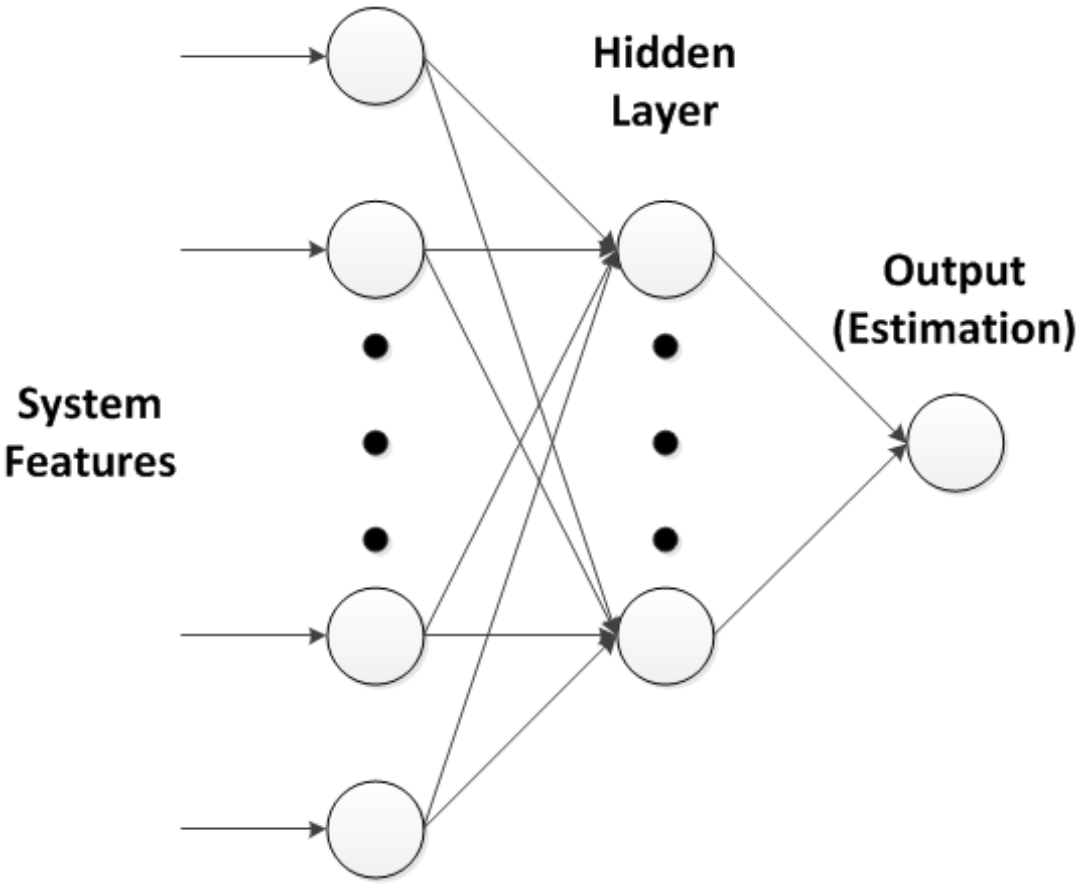


Figure 2.8: *Nerual network structure.*

input layer has the same number of neurons as the number of system features. The hidden layer has half the number of neurons as the input layer. There is only one output that represents the Q-value for each ANN. Since Q-value is calculated from system features via an ANN, updating Q-value during learning is actually done by back-propagation of ANN. In this way, ANNs are nonlinear functions used by reinforcement learning method.

Chapter 3

RL Scheduler Using Tile Coding

In this Chapter, we present the RL learning results using tile coding as the function approximation method. We show the learning curves of individual experiments and the average learning curve of all experiments in the two core system. The schedule results of trained learning agent are used to compare with heuristic schedule results and static schedule results. We also show individual test case learning curves in four core system. At the end, we discuss the results shown in this chapter and the effectiveness of tile coding as a function approximation method.

3.1 Learning System Setup

We simulated two-core and four-core ACMP systems using two different types of cores: an out-of-order core (a high-performance core) and an in-order processor (a power-efficient core). The two-core system is composed of one of each of these core types. Table 3.1 shows the configurations of these two processing cores. Since core 0 is in-order core, there is no reorder buffer, and reservation stations (shown as "-" in the table). For our initial study, the quad-core processor consists of one high-performance core and three power-efficient cores. There is a two-level data cach 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,

Table 3.1: Processor configurations

Parameter	Core 0	Core1
Execution	In-order	Out-of-order
Issue width	2	4
L1 cache	64KB	64KB
L2 cache	256KB	256KB
ROB	–	128
RS	–	32

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 (Merten et al. 2001; Sherwood et al. 2003), dynamically scheduling the applications to run on their respectively best fitting cores each period can result in the best overall system performance.

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). These features are chosen because they are directly related to the performance of the multicore system. All of these features range from 0 to 1.

The scheduler agent is trained using tile coding in these simulated experiments. 200 and 400 tiling sets are used for the two-core system and four-core system, respectively, in our experiments. 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. So there will be two possible actions (application core mappings) to be taken in two core system: putting application one on the big core and putting application two on the big core. Since the three small cores are identical, there are four possible distinct actions(possible application core mappings) to be taken in four core system. 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. The action with the largest Q-value is taken and negative one(-1) 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 when the learning agent updates the action value function, 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.

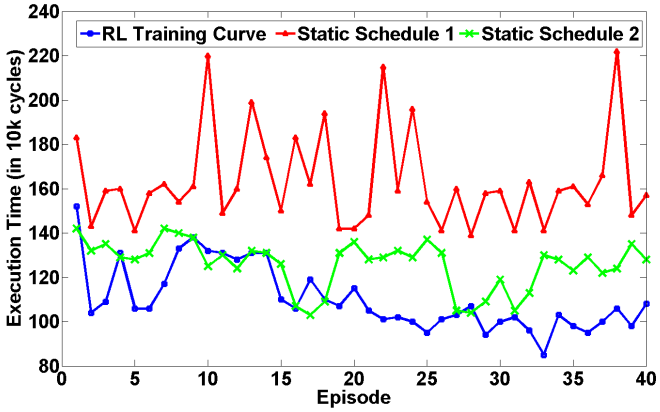
Simulated evaluation of our scheduling technique was performed using Soonergy (Freeman 2011), a cycle-accurate architectural simulator. Thirteen different benchmarks from SPEC CPU 2006 benchmark suite are used in our simulations. These benchmarks are *astar, bzip2, dealII, gcc, gobmk, hmmer, lbm, mcf, namd, omnetpp, povray, soplex*, and *xalan*. For the running time concern, we execute a small, fixed sample of instructions (50,000) and then skip a larger amount of instructions (29,950,000) until the application completes. This running scheme may skips some instruction phases but it makes the running time reasonably short. 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. The total run of each benchmark was limited to 215 million instructions chosen for their statistical relevancy using the approach in Sherwood et al. (2002).

Overall, we control what kind of features are used to represent the states, what is the reward(-1), how long the action value function get updated and reward is given (10,000 cycles), how many instructions are actually executed, and how many times the benchmarks are repeatedly executed for training period(50 times).

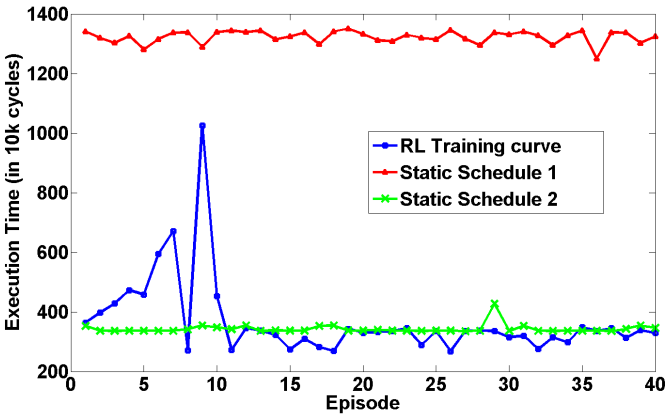
3.2 Learning Results

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 performance measurement is execution time, the best learning agent will have the lowest point on the graph representing the fastest performing schedule.

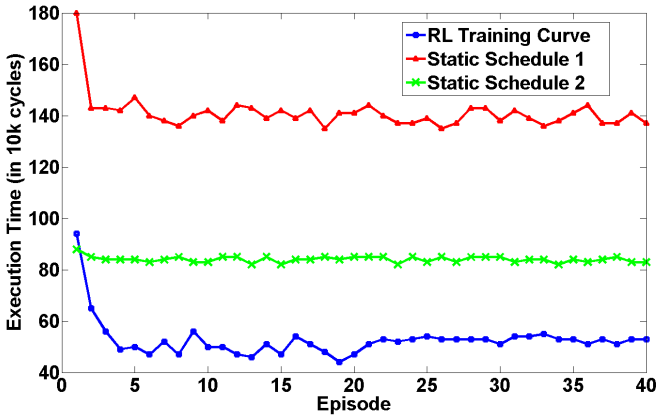
Figure 3.1 shows the learning curve of several learning-agent scheduled benchmarks compared to the two static thread-to-core assignments. Each point on the learning curve indicates the total number of cycles executed during that episode. If there is a declining trend of the learning curve means that 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 or an equally good schedule. Since, in this two-core example, there are 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 in the on-policy learning process, 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. There are variations in the static schedule



(a) *bzip2, gobmk*



(b) *lbm, dealII*



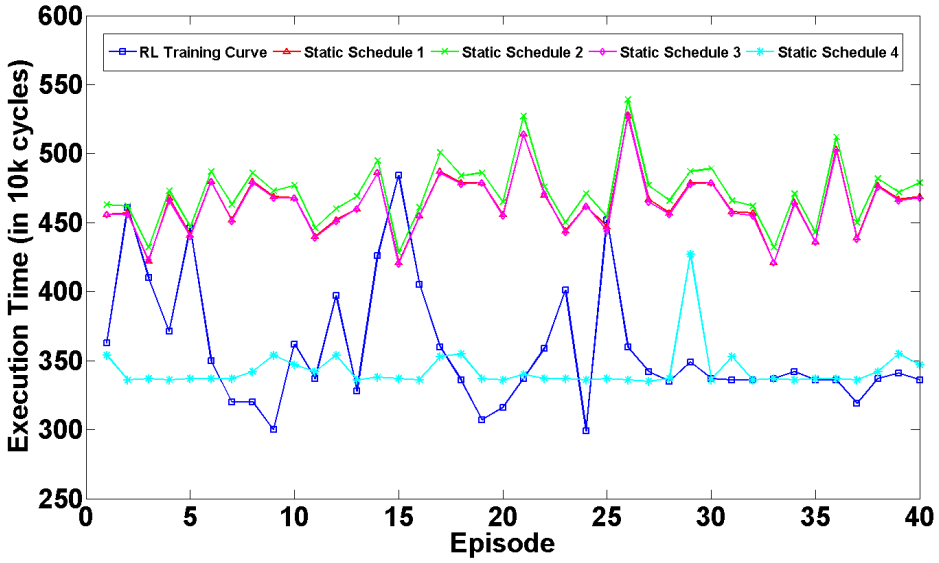
(c) *namd, povray*

Figure 3.1: *The learning results compared with the static assignments on a two dual processor.*

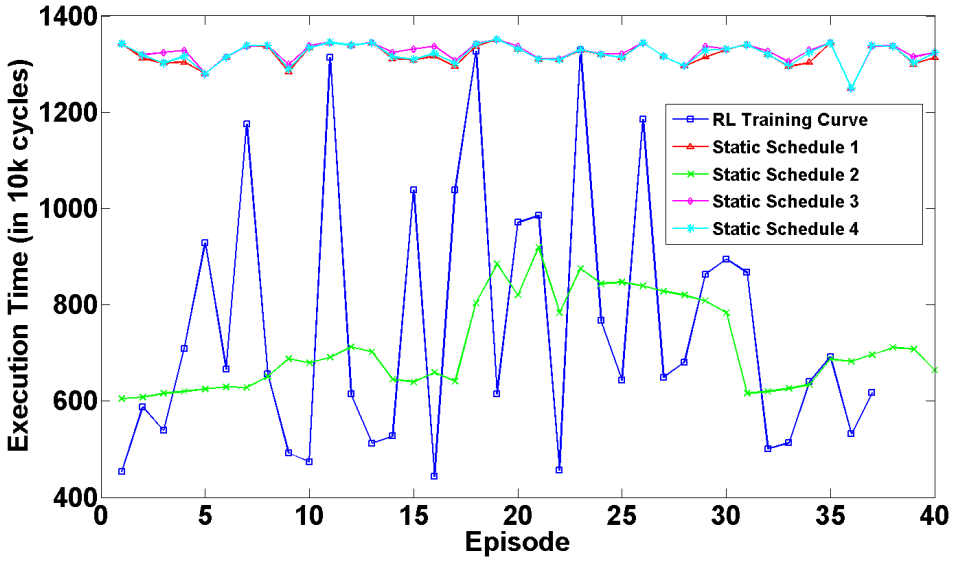
curves. The reason could be we did not clear caches for every episode in the learning process and this could leads to different starting states.

Figure 3.2 shows two selected learning results from four-core, four-benchmark systems. Comparing with Figure 3.1, we see that the learning curves of the four-core system oscillate more than the learning curves in two-core system and there is no clear decrease trend. We have done experiments using various learning rates (not shown here) in two core system, and selected the one that generates the best performance results. We use the same learning rate for experiments in four core system and the learning rate is not high. Thus, the primary reason for the wild oscillations is the non-Markovian state representation. Since the number of features that are used to represent the system state increases as the number of cores and benchmarks increases, the tile coding with randomization can not approximate the whole state space effectively.

Figure 3.3 shows the average learning result from 68 benchmark pairs in the two-core system. Since there are two possible static schedules in the two core system and these two static schedules are usually different, one of them can lead to better performance than the other. The "best schedule" result is generated by averaging results of all static schedules that can generate better performance than the other one in their running sample. The "worst schedule" is the result generated by averaging the results of the other static schedule in each running sample. 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



(a) *astar, lbm, namd, omnetpp*



(b) *astar, dealII, lbm, soplex*

Figure 3.2: *The learning results compared with the static assignments on a quad-core processor.*

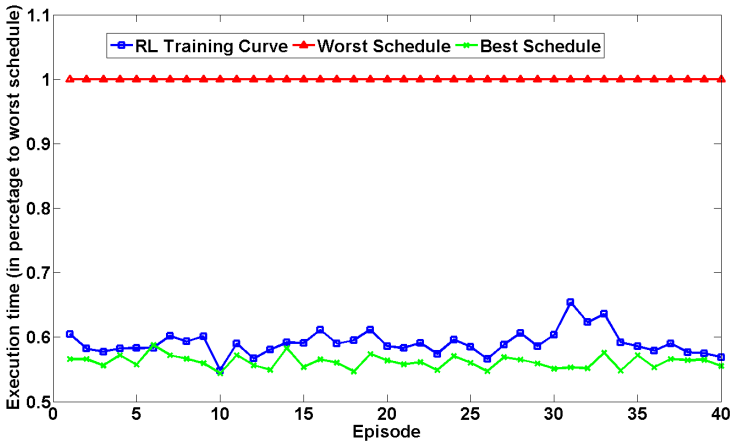


Figure 3.3: *The average execution time of RL-based scheduling on the two-core system compared to the best and worst assignments.*

the best schedule for each point. Figure 3.3 shows that the RL scheduler, on average, achieves a schedule very near to the ideal (best schedule).

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 3.4 shows weighted speedup results for a selected set of pairs of benchmarks from SPEC CPU 2006 suite, of our learning-based scheduling algorithm using tile coding compared with that of the heuristic algorithm in Kumar et al. (2004) 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,

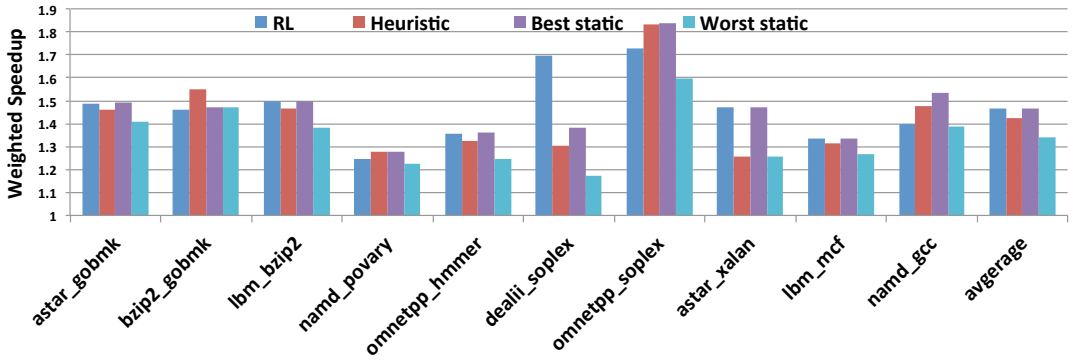


Figure 3.4: *The weighted speedup of the learning-based algorithm compared with different methods for different pairs of benchmarks.*

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 speedup than the heuristic algorithm and the best static assignment.

3.3 Discussion

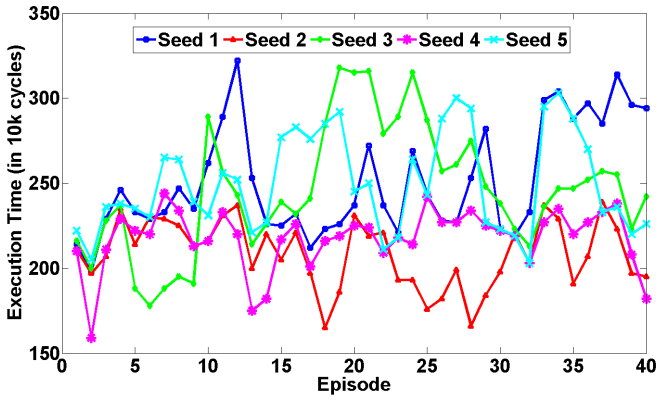
3.3.1 Tile Coding and Randomization

Tile coding is an effective way that simulates non-linear function using linear functions. However, tile coding can not avoid the curse of dimensionality, then, we define tile set and use randomization to approximate the whole tiling system. The randomization introduces stochastic information to the learning process

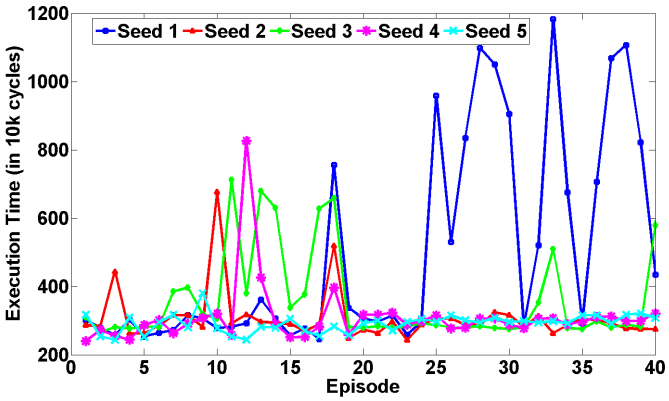
of the agent and, sometimes, causes inconsistent learning results. During the experiments, it was found that in some experiments the Q-learning approach does not effectively schedule the pair or 4-tuple of application benchmarks. These negative results occur much more frequently in the four-core system 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 may better represent the state space than others do. It is possible that different sets of features will be more effective for different application benchmarks. From Figure 3.5, we observed that performing multiple runs, with the same system and set of applications but with different random seeds, can leads to very different results.

3.3.2 Reward and Training Scheme

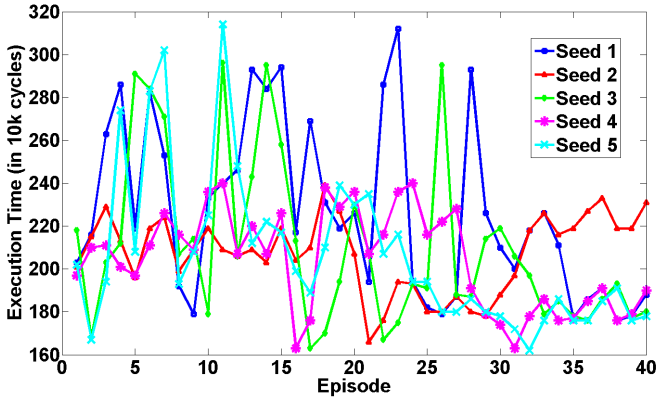
In all above experiments, the reward is defined as negative one(-1) for every 10,000 execution cycles. The goal of the learning agent is to get more accumulated reward which is get less negative ones. This setup can be easily confused and causes unnecessary complexity. And this setup leads to a confusing results representation that is using execution time as metric for the learning curve which is the lower the better. Furthermore, this reward definition is a coarse definition. For most learning processes, this reward is good enough to help learning agent to differentiate good and bad decisions. But it is not fine enough to show the differences between schedules for similar programs running together in the system.



(a) *astar,bzip2*



(b) *astar,dealii*



(c) *astar,namd*

Figure 3.5: *Comparison of learning curves with different random seeds in two-core system.*

Due to the running time consideration, we skip certain amount of instructions to make our agent training part reasonably short. We did not test more skip running schemes to choose the best one. And the learning agent may not meet all the states of the system due to large amount of skipped instructions. The learning agent will take actions according to the existing action value functions for the states that it has never met before. This could be a reason for mediocre performance of learned scheduler.

3.3.3 Features Selection

It is well known that features selection is important for RL. Large feature numbers may lead to large state space that traditional tile coding cannot handle. Ipek et al. use only six features to represent their continuous system states (Ipek et al. 2008). 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 (Coons et al. 2008). 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.

In our next experiment, we use artificial neural networks, instead of tile coding, as function approximation method. In this way, we can eliminate the randomization process and directly use nonlinear function rather than nonlinear function approximation, which making less deviation included in the learning process. We run the whole instruction set of each benchmark and use weighted speedup as reward. We also improve the features representation of state. Details are in Chapter 4.

Chapter 4

RL Scheduler Using Artificial Neural Networks

4.1 Learning System Setup

We used basically the same ACMP system as tile coding experiments with slightly different core configurations, feature representation, running scheme, and reward definition. Table 4.1 shows the configurations of these two processing cores. Since core 0 is in-order core, there is no reorder buffer, and reservation stations (shown as "-" in the table). Similar to tile coding experiment setup, states are represented by system features that are gathered during execution, and the actions represent the possible thread-to-core assignments. 38 features are used for the two-core system and 76 features for the four-core system. Fifteen different architectural and performance evaluation features are associated with each core and four features are related to L2 cache hits/misses that are associated with each benchmark. The nineteen different features are the percentage breakdown of executed instruction types (between load, store, multimedia, basic floating point, floating point multiplication, floating point division, basic integer, integer multiplication and integer division), percentage of cache hits/misses over executed instructions (L1 cache store hits/misses and L2 cache load hits/misses), and percentage of branch prediction correct/incorrect over executed instructions. All of these features range from 0 to 1. All these features are simulated features using Soonergy simulator with SPEC 2006 suite running

on it. The executed instructions are 10,000 instructions that are executed in last step of the learning process.

Table 4.1: Processor configurations

Parameter	Core 0	Core1
Execution	In-order	Out-of-order
Issue width	4	4
L1 cache	32KB	32KB
L2 cache	1MB	1MB
ROB	–	128
RS	–	32

The scheduler agent was trained using ANNs in each of these simulated experiments. For every 10,000 instructions (our chosen window size), the system reports the state features for the agent and these continuous features are inputs for all ANNs. Weighted speedup is used as the performance metric and reward. The weighted speedup is calculated as equation (4.1) with x represents the application and n represents the number of applications running in the multicore system.

$$\text{Weighted Speedup} = \sum_{\mathbf{x}=1}^{\mathbf{n}} \frac{\text{IPC of x on current core}}{\text{IPC of x on best core}} \qquad (\mathbf{n} \geq 2) \tag{4.1}$$

Weighted speedup represents the summation of the ratios between the performance (represented as instructions per cycle (IPC) since each core is assumed to have the same frequency) for each process to the single application performance on the best-performing core for each combination of applications. The higher the weighted speedup, the better the overall performance can be. To show the full potential of reinforcement learning, we attempted to give the learning agent

ample time to learn a scheduling policy. Each set of benchmarks was run repeatedly for 10 000 times to make sure a converged learning curve is generated.

Soonergy simulator and SPEC CPU2006 are used. The fifteen benchmarks are *astar*, *bzip2*, *dealII*, *gcc*, *gobmk*, *hmmer*, *lbm*, *mcf*, *namd*, *omnetpp*, *povray*, *soplex*, *h264*, *perl* and *xalan*.

Three types of simulation results are presented to evaluate our approach: the accumulated weighted speedup for each interval as the agent is iteratively trained (full learning curve), the accumulated weighted speedup while learning online (online learning curve), and an evaluation of offline scheduling. These results are compared to both/either static results and/or heuristic results. Static results are generated by averaging all results of possible static mapping assignments. Heuristic results are generated using the scheduling method in Kumar et al. (2004). Basically, this heuristic method uses sampling results to determine the mapping assignment of programs on cores. For each instruction window(10,000 instructions), the heuristic agent checks the change of IPC on each core. If the agent observes a dramatic change of IPC (more than 50% IPC difference between last window and current window over IPC of current window) on any core of the total cores in the system, it will evaluate all possible mapping assignments and chose the assignment with the highest weighted speedup for next period of run.

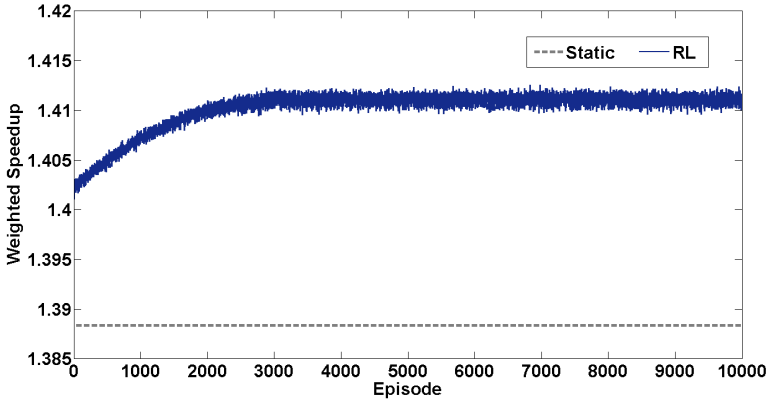
4.2 Learning Results

4.2.1 Iterative learning

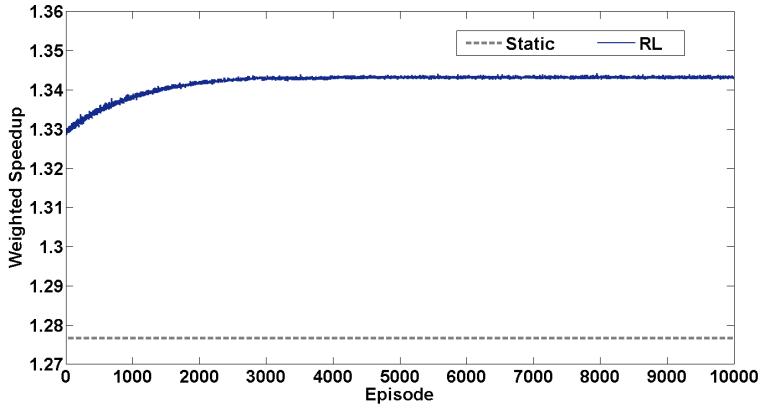
To demonstrate how the scheduling agent's policy improves during the iterative learning process, we ran (a minimum of) 250 million instruction regions of each

benchmark in each set 10,000 times and recorded the average weighted speedup of each window while all of the benchmarks executed. Note that since each benchmark has a unique IPC, each run lasted until the lowest-IPC benchmark reached 250 million instructions (higher-IPC benchmarks continue to run to provide a load against which the remaining benchmarks are run). The starting point of each 250 million instructions region was chosen to be statistically relevant using the approach in Sherwood et al. (2002). Each period of time that all benchmarks need to complete is called an episode. Results from this iterative learning are reflected in full learning curves.

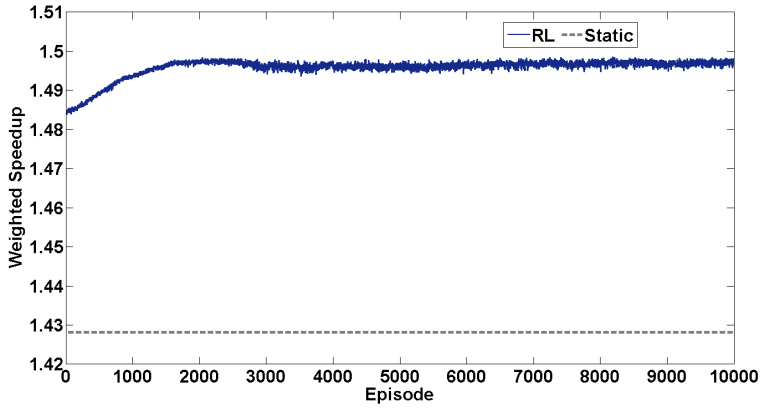
Figure 4.1(a) and Figure 4.1(b) show two full learning curves for two benchmarks run on the two-core system. Each point on the learning curve is the average weighted speedup over each window for each episode. The static line represents the average weighted speedup of all possible fixed mappings of benchmarks to cores. Thus, these static mappings do not change during execution. In our experiments, the learning curve, on average, achieves better performance than the static line. The learning agent can generate a schedule that leads to higher weighted speedup than the average static schedule does at the end of training. The figures show that the learning curves initially increase as the agent learns over each episode and then levels off. During the increasing part of the curve, the learning agent is still trying to learn a better scheduling policy; once this improvement levels off, an efficient scheduling policy has been learned and no better one is found by using this algorithm. The trained schedule policy is specific for the training benchmarks. The schedule policy are defined by the action value functions. Looking at the learning curve from one point to another, the learning curve does not always increase. This is because of the random action, i.e. exploration step, of the agent. With a small probability, the agent



(a) *lbm, xalan*



(b) *dealII, namd*



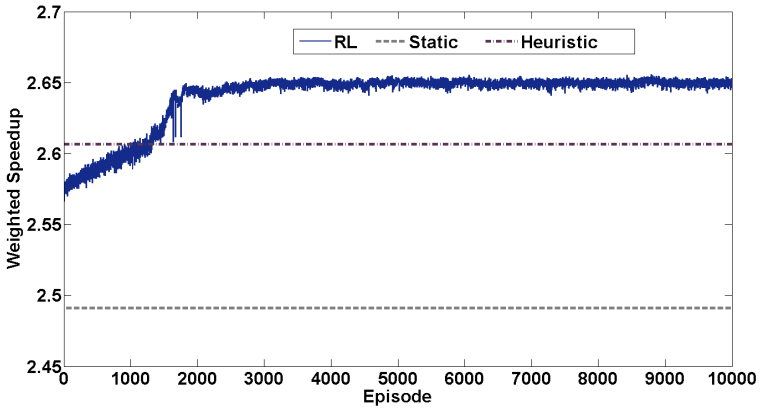
(c) *average*

Figure 4.1: *Full learning curves for a two-core system compared with the average static assignments.*

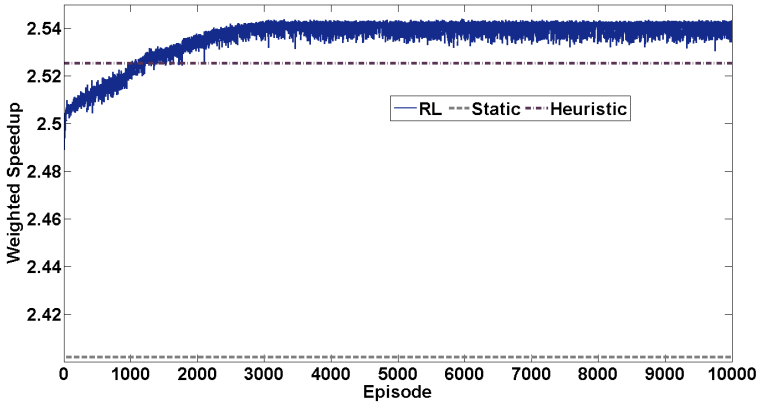
chooses a random scheduling action for next window. The exploration step helps the agent find the global optimized scheduling policy. This exploration step could temporarily result in a bad schedule for that episode; thus a reduction in performance may be reflected in the learning curve. We decreases the probability of exploration during the learning process to damp the oscillation of learning curve. Figure 4.1(c) represents the average learning curve from 105 benchmark pairs (all possible two-program combinations for 15 benchmarks) on a two-core system. Note that in each of the curves in Figure 4.1, the performance of the learning agent for the first episode is already better than that of the average static schedule. This is because one episode represents an entire run of (a minimum) of 250 million instructions for each application and the agent has quickly learned a scheduling policy during the first episode to achieve an average performance better than that of the static schedule. The progress of learning within the first episode is examined more closely in Section 4.2.2.

Figure 4.2(a) and Figure 4.2(b) show two full learning curves for four benchmarks on four-core HMP. The learning results are better than both heuristic sampling (Kumar et al. 2003b) and the average of all static assignments. To isolate the quality of the scheduling, all overhead from the sampling approach has been eliminated giving sampling an unfair advantage, yet the learning approach still outperforms it due to its superior scheduling policy. Figure 4.2(c) shows the average learning curve from 12 randomly picked benchmark combinations. These results were limited to 12 due to the significantly longer simulation time for the four core system.

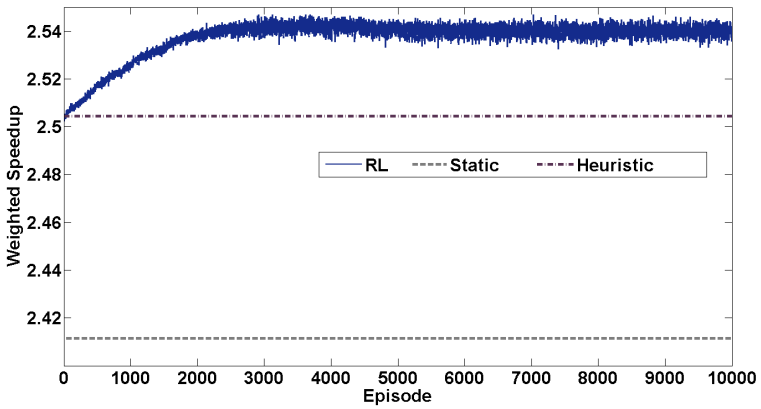
Figure 4.3 shows a comparison between trained learning results, heuristic results and static results. Trained learning results are taken from the level-off portion of learning curve. For all 12 benchmark combinations, the trained



(a) *poveray, soplex, xalan, astar*



(b) *perl, astar, gobmk, mcf*



(c) *average*

Figure 4.2: *Full learning curves for four-core system compared with static and heuristic assignments.*

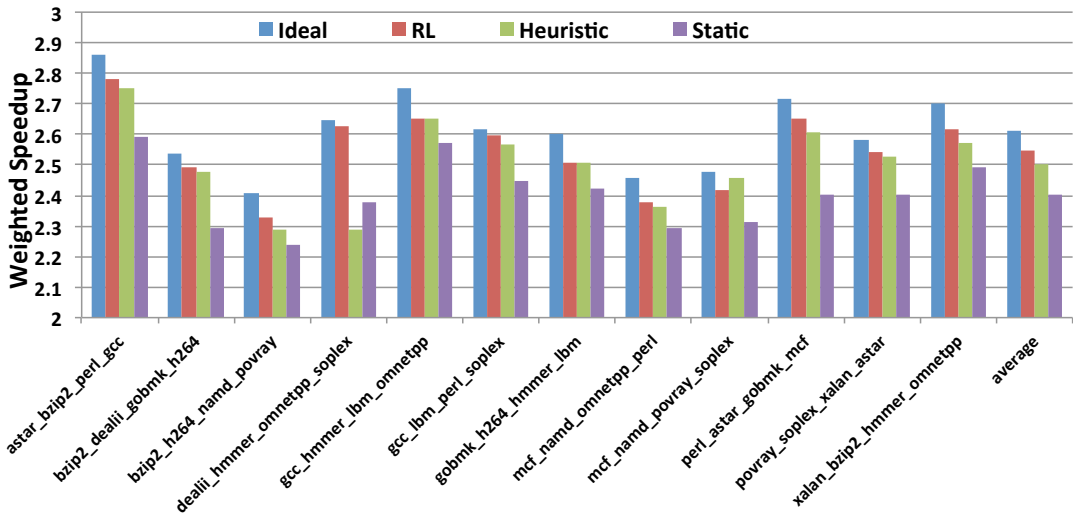
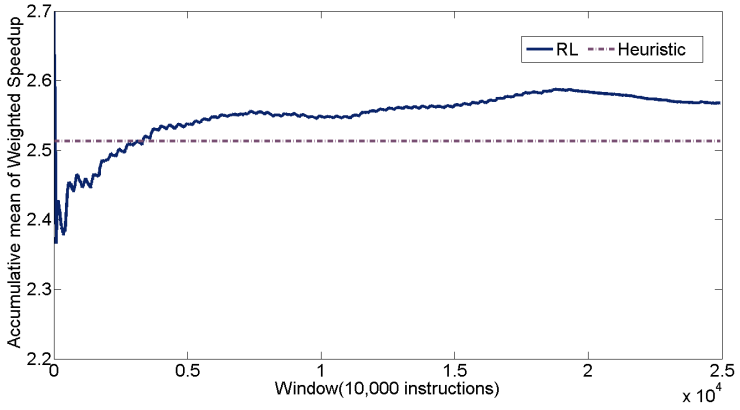


Figure 4.3: *Comparison between ideal, learning, heuristic, and static results in the four-core system.*

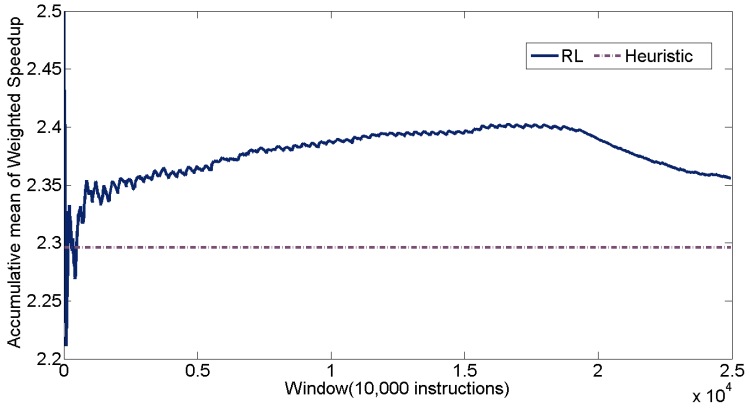
learning results are significantly better than the average static results evaluated by student's t-test ($p < 0.05$). On average, trained learning result is 1.77% better than heuristic sampling result (ignoring the cost of context switching) but the difference is not significant evaluated by student's t-test ($p > 0.05$), and 6% better than the average static result. From the aspect of the distance to estimated ideal schedule, RL schedule, on average, achieves 41.4% significant improvement towards the ideal weighted speedup over the heuristic results evaluated by student's t-test ($p < 0.05$). The results above demonstrate the potential of our reinforcement learning method. However, the iterative training process for the scheduler may be infeasible but represents the training that would occur over a relatively long run of the sets of applications. We next demonstrate that reinforcement-learning-based scheduler is effective even while learning online during the (relatively) shorter period of an individual episode of 250 million instructions.

4.2.2 Online Learning

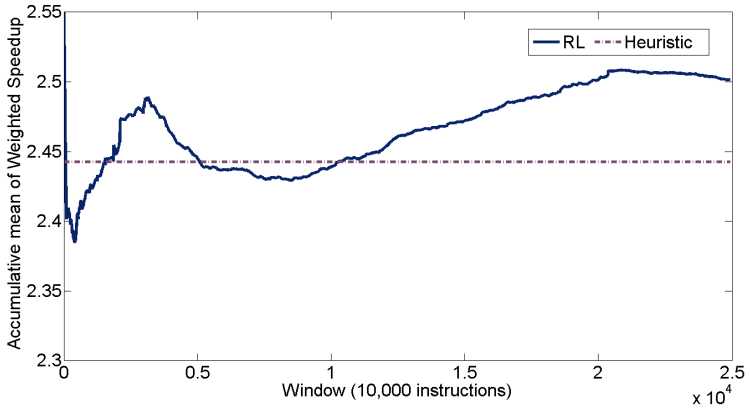
For online learning results, benchmarks were run only once and the cumulative mean for every window was recorded. Figure 4.4(a) and Figure 4.4(b) represent online learning results from two different benchmark combinations running on the four-core system. For each window, the accumulated mean of the weighted speedup is recorded and is shown compared to the average accumulated weighted speedup using heuristic sampling method. From the results, we can see that the reinforcement learning agent learns quickly to find a better schedule than heuristic method even at an early stage. Figure 4.4(c) shows the average learning process in one complete run of each of the 12 sets of benchmarks on the four-core system. From the figure, we can see that the learning agent again learns quickly to outperform the heuristic-sampling-based scheduler in early stage of program execution and eventually generates significantly better schedules. The drop of performace during the learning process is caused by exploration step of the RL agent. With the same ANNs initialization, all experiments are repeatable and will generate the same results. Figure 4.5 represents the comparison between online learning results and heuristic sampling results. Online learning results are taken from the last point of online learning curve since this point represent the average weighted speedup over each window of the whole running process. The weighted speedups for online learning are significantly better than those for heuristic sampling for most benchmark combinations evaluated by student's t-test ($p < 0.05$). On average, the online learning result is 2.4% better than heuristic results (ignoring sampling cost). Real world applications are usually longer than the portions of benchmarks simulated, so our reinforcement learning based scheduler should be capable of optimizing such applications efficiently and transparently.



(a) *poveray, soplex, xalan, astar*



(b) *mcf, namd, omnetpp, perl*



(c) *average*

Figure 4.4: *Online learning curve for four-core system compared to heuristic assignments.*

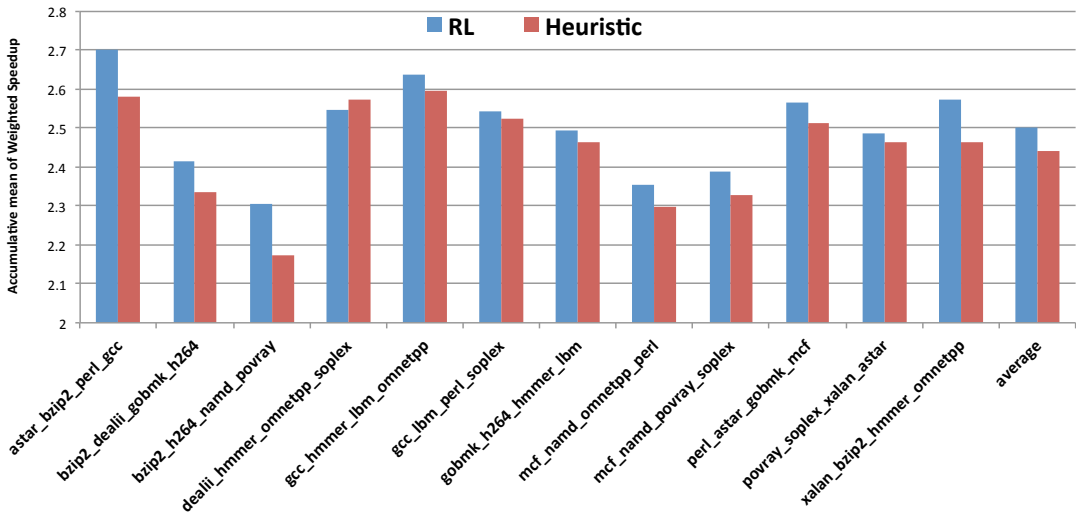
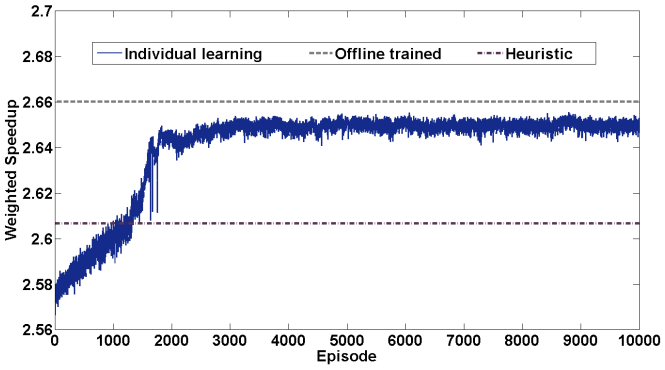


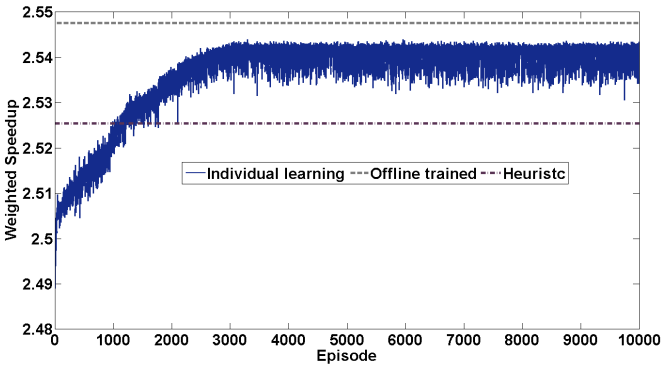
Figure 4.5: *Comparison between online learning and heuristic results in the four-core system.*

4.2.3 Offline learning

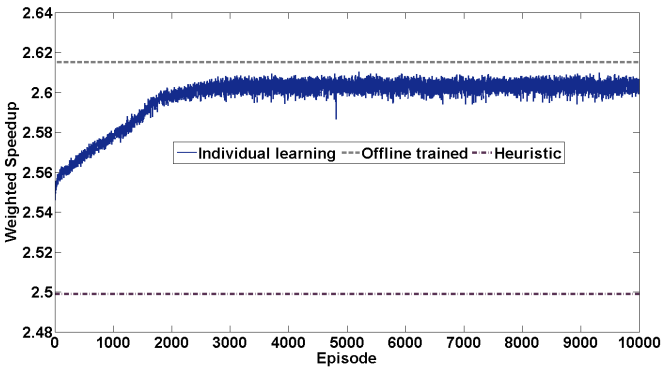
Additionally, a preliminary evaluation of an offline training approach was also performed. Instead of training the scheduler on individual benchmark combinations, we trained the scheduler by running all twelve benchmark combinations sequentially. In theory, the scheduler encounters more states than it does by training on individual ones. Thus, the scheduler will learn a general scheduling policy that can be used for all sets of programs. In this way, we could train a scheduler offline and use the fully trained scheduler on applications directly. In the case of this offline-trained system, the relationships between states are more complicated. In our experiments, this approach to offline training of the scheduler only works well for one third of the tested benchmark combinations. During the training period, all twelve benchmark combinations finish running once represents one episode. And the trained RL agent is used to schedule all the test cases. Figure 4.6 shows that the offline scheduler can generate results



(a) *perl, astar, gobmk, mcf*



(b) *poveray,soplex,xalan,astar*



(c) average of the four test cases that the offline scheduler can generate better results than the individual training results

Figure 4.6: *Comparison between individual training, offline trained, and sampling heuristic results in the four-core system.*

that are even better than the fully learned individual trained results in the best case. The results indicate that a study on offline-trained scheduler is promising, but that for such an offline-training approach, additional work on training the scheduling agent is needed to train an agent that is capable of scheduling well on a wide variety of applications. Some potential explanation for the inconsistent results from offline-trained scheduling are discussed in Sections 4.3 and 5 along with an initial evaluation of approaches for improving this offline-trained scheduling. However, we haven't done any study related to feature set selection and ANNs structure optimization for our RL offline scheduler. These are works we have to do in the future.

4.2.4 Learning in the presence of switching costs

In addition to the experiments performed neglecting any overhead for switching applications between cores, we also trained the RL agent to learn a scheduling policy while taking account of cost of switching. In these results, we assumed a direct cost of 15,000 cycles (in addition to the indirect cost) for switching applications between cores for each methods. These additional cycles impact the calculation of IPC and, as consequence, penalize our RL agent if it takes unnecessary switch actions. Thus, the agent learns not to switch when the cost of switching outweighs the benefits. We run all twelve benchmark combinations using individual trained RL schedulers and heuristic scheduler in the four core system. Figure 4.7 shows a comparison between RL scheduling results after iterative training and heuristic schedule results. On average, the RL agent gets 4.1% more weighted speedup than the heuristic result and achieves 41.7% of the difference between the performance of the heuristic result and our estimation of the maximum possible weighted speedup on the four core HMP.

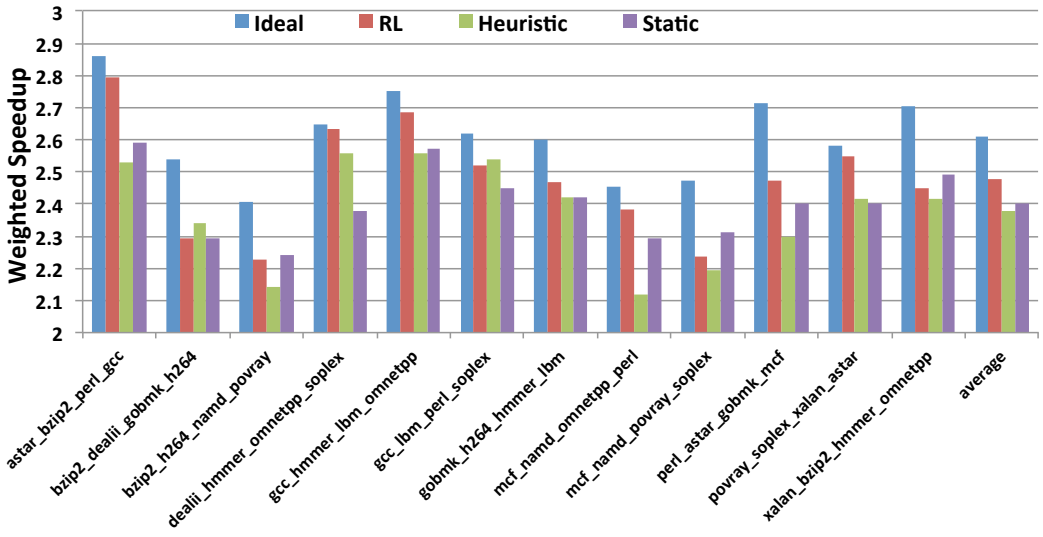


Figure 4.7: *Comparison between ideal, learning, heuristic, and static results in the four-core system with costing switch.*

Both improvements are significant evaluated by student's t-test ($p < 0.05$). Table 4.2 shows how many times the scheduler switch the applications on cores to change the mapping. Figure 4.8 shows one example of detailed scheduling using RL scheduler and heuristic scheduler in a short running period. For 100 windows (10,000 instructions as a window), RL scheduler does not take any action, while heuristic scheduler keeps changing the application core mapping. The RL scheduler learned the cost for switching is high and switch much less times than the heuristic scheduler do. Our RL agent achieves better overall system performance by intelligently avoiding unnecessary switches.

Table 4.2: Core switch times using RL and heuristic scheduler

Scheduler	RL	Heuristic
astar,bzip2,perl,gcc	2	4400
bzip2,dealii,gobmk,h264	2	3600
bzip2,h264,namd,povray	2	3422
dealii,hmmer,omnetpp,soplex	1	1906
gcc,hmmer,lbm,omnetpp	2	2252
gcc,lbm,perl,soplex	2	576
gobmk,h264,hmmer,lbm	2	2030
mcf,namd,omnetpp,perl	2	7128
mcf,namd,povray,soplex	1	5700
perl,astar,gobmk,mcf	2	7802
povray,soplex,xalan,astar	1	2492
xalan,bzip2,hmmer,omnetpp	2	3698

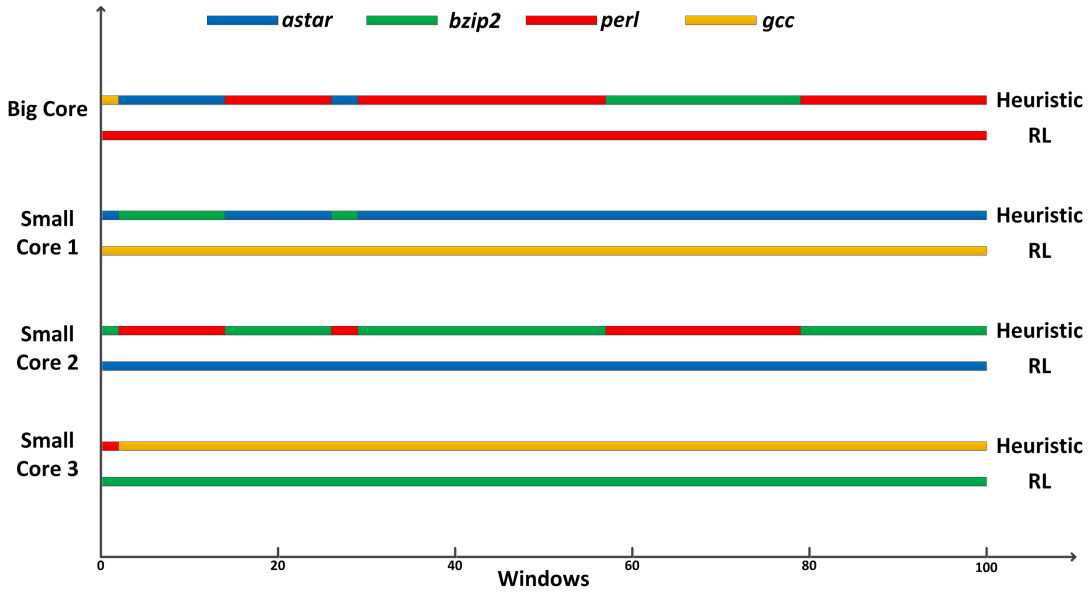


Figure 4.8: *Comparison between RL schedule and heuristic schedule for* astar, bzip2, perl, gcc *for 100 windows in the four core system.*

4.3 Discussion

4.3.1 Ideal schedule

In order to get an idea about how far away we have to improve the schedule from the optimal schedule, we estimated the ideally best weighted speedup for these benchmark tuples in this four-core system. The ideal weighted speedup for each four-tuple of benchmarks was approximated by running each application on each core type and recording the performance over fixed windows of 10,000 instruction for each run. For each window of 10,000 instructions, mapping of application to core type resulting in the highest weighted speedup for that window and the "ideal" weighted speedup was computed across all of the windows of instructions comprising the full run of that set of applications. Of course, due to varying instructions-per-cycle across the benchmarks, these fixed

windows of instructions would not align dynamically in this way–thus this approach serves only to estimate the potential of an ideal mapping of applications across the different core types. The ideal schedule generates results that are 4.29% better than heuristic results without switch cost and 9.87% better than heuristic results with switch cost in four core system. This means there is not a lot of space for RL agent to improve the schedule in the four core system.

4.3.2 Learning vs. empirical models

In this study, we compared a learning-based scheduler for heterogeneous multicore processors with a sampling-based heuristic method. In contrast to our learning-based scheduler, the sampling method does not handle system diversity very well: increasing number of core types leads to significantly higher cost of sampling overhead. Van Craeynest et al. (2012) proposed an alternate model that uses cycles per instruction, memory-level performance, and instruction-level performance information to estimate the performance of alternate mapping assignments and then dynamically update the schedule. They identify an observational relationship between this information and use this relation to build an estimation model. Their approach has something in common with ours: they use information about running applications as input to a model and generate corresponding estimations to make a schedule. However, their model is somewhat specific to the particular details of the HMP system. For more complex systems, new heuristic models would have to be tuned. Similarly, their empirical model is only applicable to optimize workload performance. In order to schedule for a different optimization goal, e.g. energy, a new traceable relationship from either experimental observation or theoretical deduction would have to be explored. In contrast, a learning-based scheduler can be easily re-targeted

to other optimization goals by training using different rewards. This training process is similar to the manual relationship exploration process but is more efficient and can be performed automatically.

4.3.3 Semi-supervised vs. supervised learning

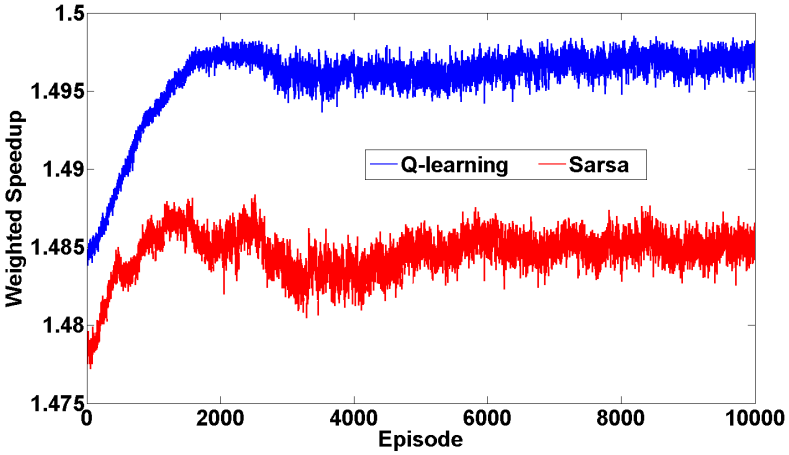
Machine learning approaches have occasionally been used in parallel programming and execution. Wang and O'Boyle (2009) use ANNs and support vector machines to build a model to predict the number of threads and scheduling policy for parallelizable applications. During the training of their model, the appropriate scheduling policy and optimal number of threads must be known. They classify different types of programs and assign the right schedule with certain number of threads. Ipek et al. (2005) use a similar approach to build a parallel program performance prediction model. Both of these techniques use supervised machine learning approaches. Supervised learning agents are trained with knowledge of the optimal or "right" answer. Our learning approach, reinforcement learning, is semi-supervised learning; the optimal schedule is not known beforehand. However, the learning agent still needs the reward function to assist learning. The reward could be defined by transforming environment feadback or other observations. In our technique, the learning agent needs to find an optimal schedule without any initial knowledge of it. As overall throughput is the scheduling goal, system performance characteristics are good feedback for the agent. For this reason, semi-supervised learning seems the most appropriate learning algorithm for this scheduling task.

4.3.4 Sarsa and Q-learning

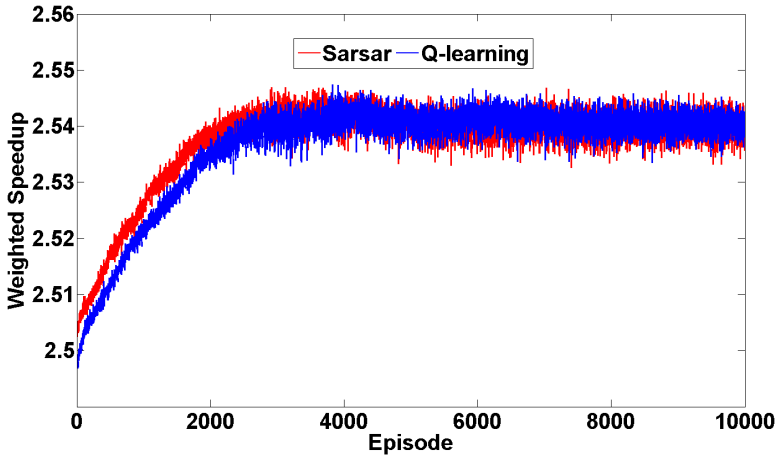
The State-Action-Reward-State-Action (Sarsa) algorithm differs from Q-learning in how much credit is given to the exploration step (Sutton and Barto 1998).Q-learning evaluates one policy, but follows another policy. While Sarsa evaluates and improves the same policy. Q-learning has more tolerance for policy change during learning than Sarsa and can eventually converge to the optimal policy. In out study, we use Sarsa for the learning tasks. We run RL agent training process of all 105 benchmark pairs/12 benchmark combinations in two core system/four core system using Q-leaning and Sarsa. Then we compare the learning curves of these two RL algorithms. Figure 4.9 shows a comparison of learning curves from scheduling using both Sarsa and Q-learning. In the two-core system, Q-learning performs better than Sarsa because there are only two possible actions and it takes less time for Q-learning to converge to the optimal policy in the simple system. On the other hand, Sarsa may stuck in an sub-optimal policy and need long time or never to get over it. In the four-core system, the Q-learning agent learns slower than the Sarsa agent because there are more available actions and it is hard to take every step right. Sarsa shows the advantage as an on-policy learner. As the learning goes on, the Q-learning agent will eventually find the optimal policy as Sarsa does.

4.3.5 Defining scheduler actions

In our studies, we found that the definition of action is an important factor for learning. Clearly defined actions help learning agent find a good scheduling policy while action aliasing may preclude the agent from learning. Figure 4.10 compares the result of learning-based scheduling for a two-core/two-benchmark



(a) *average learning curve in two-core system*



(b) *average learning curve in four-core system*

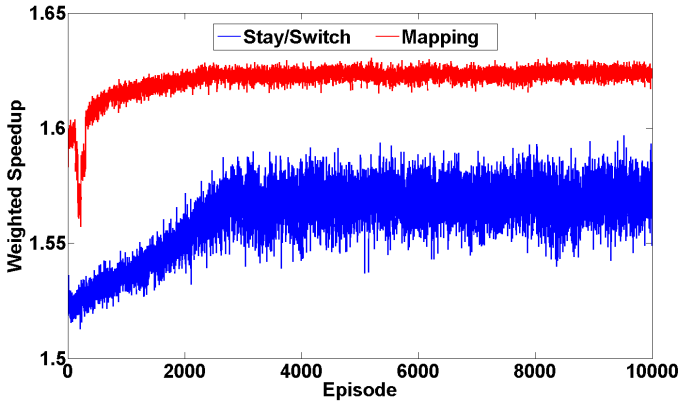
Figure 4.9: *Comparison of SARSA and Q-learning.*

system using two different action definitions. Results from learning with specific mapping actions are seen to be better than results from learning with stay/switch actions (i.e. keep the current thread-to-core mapping or switch to the alternate mapping). The average result is calculated over the RL training results of all 105 benchmark pairs in the two core system. Actions of stay/switch leads to inconsistent state representation. Thus, action definition is critical for effective learning.

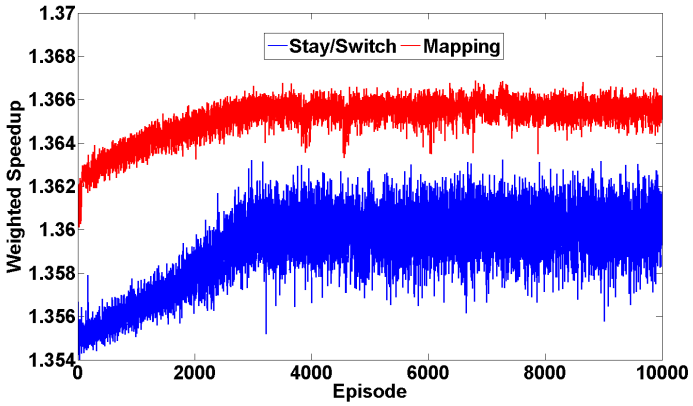
4.3.6 Choice of function approximation method

In our initial reinforcement-learning approach (Yan et al. 2012), tile-coding was used as function approximation method for reinforcement learning. While tile coding can handle nonlinear relationships to some extent, it is still based on linear function approximation. ANNs with hidden layers can represent nonlinear relationship much better than tile coding. Another advantage of ANNs over tile coding is ANNs do not have the problem of the curse of dimensionality. We used a randomization method to choose features to represent states, since tiles increase with number of features exponentially. We identified this problem as the source of their inconsistent results.

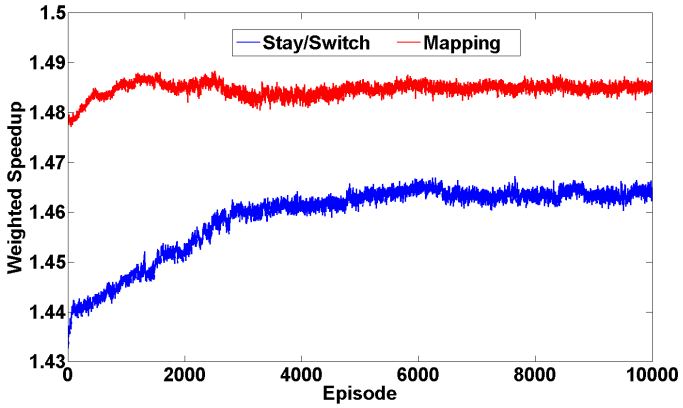
In our two-core system experiments, we found that in a minority of instances, our learning results are not better than results of heuristic sampling. We suspect that the reason might be that the ANNs are constrained to a local optimal. The reinforcement learning agent attempts to find a more globally optimal schedule through the exploration step, however, the agent takes greedy steps most of time. If initial weights are near to a local optimal policy, weights may be updated to that policy and be difficult to get over due to greedy steps. In order to test this hypothesis, we selected 10 benchmark pairs from results



(a) *gcc, h264*



(b) *namd, xalan*



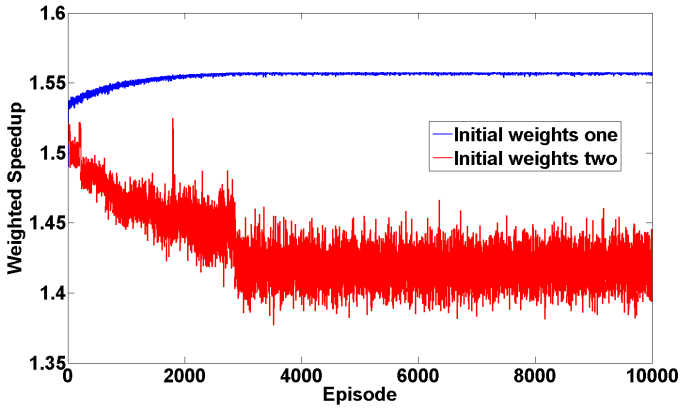
(c) *average*

Figure 4.10: *Comparison of full learning curves with different action definitions in the two-core system.*

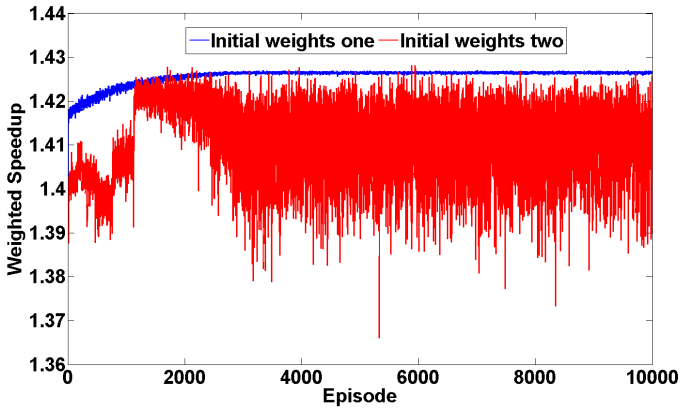
that were below average and reran them with different ANN weight initializations. Results from re-runing our iterative learning confirm this suspicion as many benchmark pairs have improved performance. Figure 4.11(a) and Figure 4.11(b) shows two examples of performance improvement due to different weight initialization. Hansen and Salamon (1990) use an ensemble to overcome the problem of local minima. Al-Shareef and Abbod (2010) use particle swarm optimization to do initial weights optimization. However, we did not further study the issue of weight initialization in this paper and all of the learning results in the four-core system are better than results of heuristic method. Local optimal constraint may also exist in our four-core system experiments, but, due to limited capacity in large system, the learning agent can still get better results.

4.3.7 Offline learning

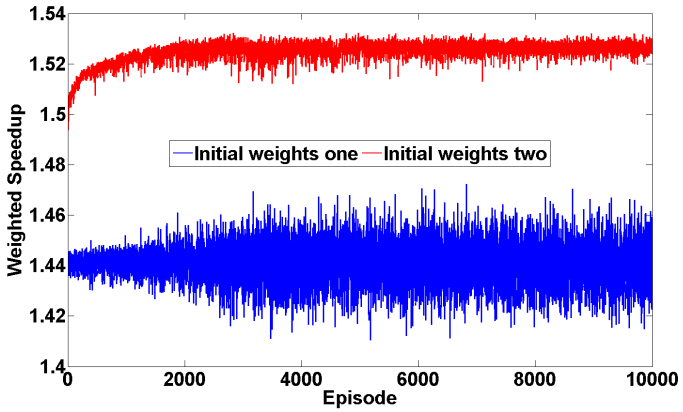
Our preliminary experiments of an offline-trained scheduler show significant potential. However, the offline scheduler trained by sequential running benchmarks can generate good schedule only for some of the tested benchmark combinations. We further conducted experiments to study the reason by shifting the sequence of training set of benchmark combinations. Figure 4.12 shows, after training on different sequence, the scheduler works for new sets of benchmark combinations that have no common member with the old working sets. Inspecting the training curve shown in this figure, we found that last several benchmarks in training sequence may overwrite the policy for some states. Thus the offline-trained scheduler only works for benchmark combinations that have similar policy on common states. Figure 4.13(a) shows the starting point of the offline-trained learning curve is lower than the start point of individual training curve. This means the trained scheduler at that point has some policies that are not optimal



(a) *bzip2,perl*



(b) *perl,h264*



(c) *namd,omnetpp*

Figure 4.11: *Comparison of full learning curves from different ANNs weights initialization in the two-core system.*

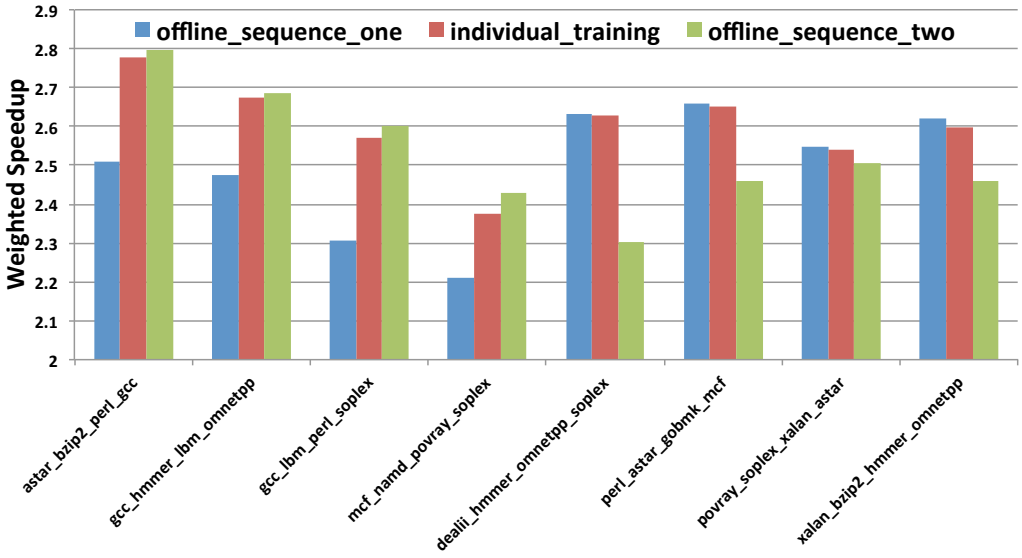
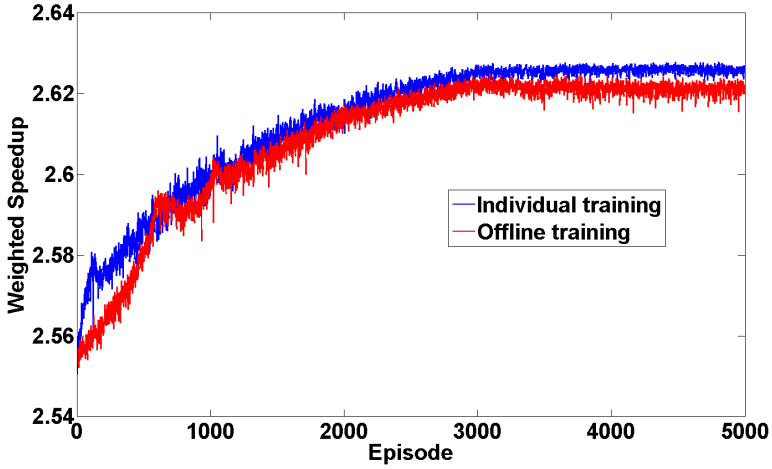


Figure 4.12: *Comparison between the results of an offline scheduler trained from the same training set but in two different training sequences and individual learning results in the four-core system.*

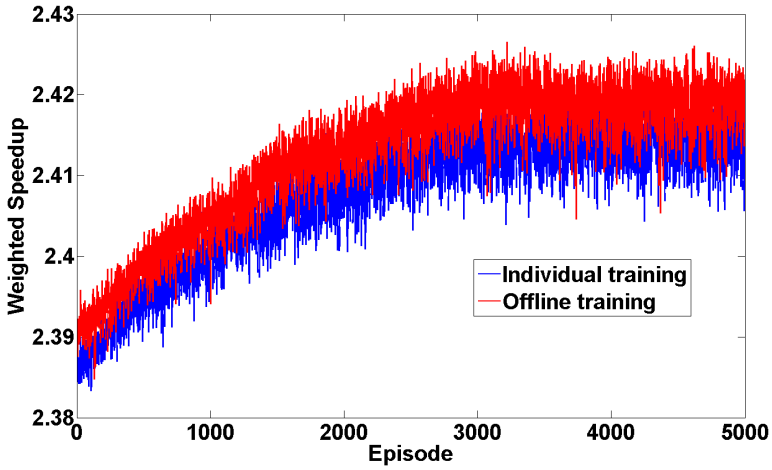
for this specific benchmark combination and this scheduler is retrained while running this benchmark combination. Figure 4.13(b) is an example where policies for some states are already optimal for the running benchmark combination and thus the scheduler needs only learn policies for other states during running this benchmark combination. As mentioned previously, states are represented by features from benchmarks and cores. Feature selection and generalization is thus important for state differentiation and, furthermore, the offline scheduler training.

In addition to the sequential offline training, we also performed experiments using a simple ensemble method. Since RL is a semi-supervised algorithm, traditional ensemble methods (Freund and Schapire 1995; Ho 1995) can not be used here. We trained the twelve four-benchmark combinations iteratively and

outputed the weights of their neural networks. We then run one benchmark combination using trained schedulers from each of the other eleven benchmark combinations. Each scheduler thus contributed equally to the final schedule. This whole process is similar to cross validation (Kohavi et al. 1995) except that the training is not supervised. This simple ensemble approach produced positive results in only a minority of benchmark combinations, including those shown in Figure 4.14. Allowing dissimilar programs to contribute to the scheduling policy produced unsatisfactory results. Thus, some form of online learning may be preferable. However, the several positive results indicate that an ensemble approach could be a direction for further offline-trained scheduler study.

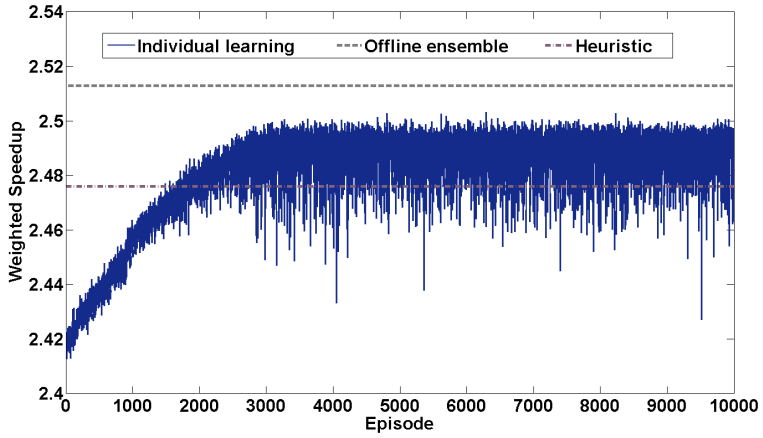


(a) *dealii,hmmer,omnetpp,soplex*

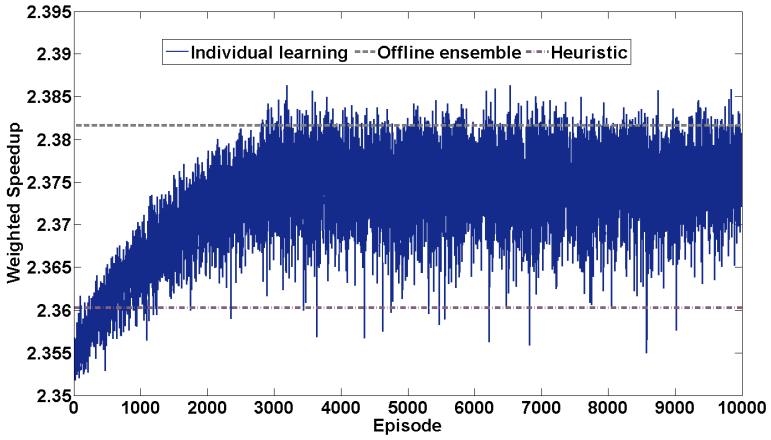


(b) *mcf,namd,poveray,soplex*

Figure 4.13: *Two examples in offline-training study.*



(a) *bzip2,dealii,gobmk,h264*



(b) *mcf,namd,omnetpp,perl*

Figure 4.14: *Two working cases of offline-trained simple ensemble.*

Chapter 5

Conclusions and Future Work

In this thesis, we demonstrated a reinforcement-learning-based scheduling approach that is effective at thread-to-core assignment in HMPs. Data show that the RL-based scheduling agent improves its scheduling policy, typically achieving schedules that outperform both static and heuristic sampling methods in our experiments. Results of full learning curves show the potential improvements of this approach over other scheduling methods. Online learning results show that this approach quickly learns effective scheduling policies when directly applied as an online scheduler. Preliminary results of offline learning confirm that offline-trained reinforcement learning schedulers can similarly achieve the benefit of this approach. However, as we have demonstrated, there are challenges to achieving a versatile offline-trained scheduler.

We conclude that semi-supervised reinforcement learning approach is an appropriate method for the task of mapping applications to cores of different types in an HMP since no optimal schedule is known in this system. In our initial tile coding experiments, results of the offline performance comparison between trained agents and previously published heuristics were somewhat mixed. 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. After switching to ANNs with appropriate defined reward (weighted speedup), we achieve significant performance improvement over initial experiments. We further discussed parameter setup and function approximation for learning. As we have demonstrated, the definition of actions is extremely important for learning approaches and experiments show that action aliasing can lead to suboptimal and even bad schedules. The function approximation approach using ANNs have an advantage of handling nonlinear relationship and are not affected by the curse of dimension, but they may have the problem of local optimal constraint (finding only a locally optimal scheduling policy). By optimizing the ANN's initial weights, this problem can be avoided. However, overall our reinforcement-learning-based scheduler show the potential to be effective and efficient for process assignment in HMPs.

In our study, states are represented by system features. Any undifferentiated states could lead to contradictory optimal policies, especially in the offline-trained scheduler process. In future work, we will examine the selection of features more closely and choose the most related features to represent the system. There is also the potential to improve the ANN structure to handle more complicated state representation. While our results in this paper indicate that the reinforcement-learning-based scheduler is an efficient online scheduler, there is still potential to improve it by shortening or eliminating online training. With more detailed knowledge of state representation, the scheduler could better trained to recognize important phases and a trained offline scheduler that is beneficial on general sets of applications could be achieved.

Reference List

Al-Shareef, A. and Abbod, M. (2010). Neural networks initial weights optimisation. In *Proceedings of the 2010 12th International Conference on Computer Modelling and Simulation*, pages 57–61.

Albus, J. S. (1975). A new approach to manipulator control: The cerebellar model articulation controller: Cerebellar model articulation controller(cmac). In *Journal of Dynamic Systems, Measurement and Control*, volume 97, pages 220–227.

Becchi, M. and Crowly, P. (2008). Dynamic thread assignment on heterogeneous multiprocessor architectures. *Journal of Instruction-Level Parallelism*, 10:29–40.

Bellman, R. (1957a). Dynamic programming and its application to optimal control.

Bellman, R. (1957b). A Markovian decision process. Technical report, DTIC Document.

Coons, K., Robatmili, B., Taylor, M., Maher, B., Burger, D., and McKinley, K. (2008). 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*, pages 32–42.

Crites, R. H. and Barto, A. G. (1998). Elevator group control using multiple reinforcement learning agents. *Machine Learning*, 33(2-3):235–262.

Freeman, N. (2011). Soonergy: A pluggable, cycle-accurate computer architecture simulator. Master's thesis, The University of Oklahoma, Norman, Oklahoma.

Freund, Y. and Schapire, R. (1995). A desicion-theoretic generalization of online learning and an application to boosting. In *Computational learning theory*, pages 23–37. Springer.

Hansen, L. and Salamon, P. (1990). Neural network ensembles. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(10):993–1001.

Hecht-Nielsen, R. (1987). Counterpropagation networks. *Applied optics*, 26(23):4979–4983.

Ho, T. (1995). Random decision forests. In *Document Analysis and Recognition, 1995., Proceedings of the Third International Conference on*, volume 1, pages 278–282. IEEE.

Ipek, E., de Supinski, B., Schulz, M., and McKee, S. (2005). An approach to performance prediction for parallel applications. *Proceedings of the 11th international Euro-Par conference on Parallel Processing*, pages 627–628.

Ipek, E., Mutlu, O., Martínez, J. F., and Caruana, R. (2008). Self-optimizing memory controllers: A reinforcement learning approach. In *Proceedings of the 35th Annual International Symposium on Computer Architechture*, pages 39–50.

Kohavi, R. et al. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In *International joint Conference on artificial intelligence*, volume 14, pages 1137–1145. Lawrence Erlbaum Associates Ltd.

Koufaty, D., Reddy, D., and Hahn, S. (2010). Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European Conference on Computer Systems*, pages 125–138.

Kumar, R., Farkas, K., Jouppi, N. P., Ranganathan, P., and Tullsen, D. M. (2003a). Processor power reduction via single-ISA heterogeneous multi-core architectures. *Computer Architecture Letters*, 2(1):2–2.

Kumar, R., Farkas, K. I., Jouppi, N. P., Ranganathan, P., and Tullsen, D. M. (2003b). Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 81–92.

Kumar, R., Tullsen, D. M., Ranganathan, P., Jouppi, N. P., and Farkas, K. I. (2004). Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 64–75.

McGovern, A., Moss, E., and Barto, A. G. (2002). Building a basic block instruction scheduler with reinforcement learning and rollouts. *Machine Learning*, 49(2/3):141–160.

Merten, M. C., Trick, A. R., Barnes, R. D., Nystrom, E. M., George, C. N., Gyllenhaal, J. C., and Hwu, W. W. (2001). An architectural framework for runtime optimization. *IEEE Transactions on Computers*, 50(6):567–589.

Nie, J. and Haykin, S. (1999). A Q-learning-based dynamic channel assignment technique for mobile communication systems. *IEEE Transactions on Vehicular Technology*, 48(5):1676–1687.

Patsilaras, G., Choudhary, N. K., and Tuck, J. (2012). Efficiently exploiting memory level parallelism on asymmetric coupled cores in the dark silicon era. *ACM Transactions on Architecture and Code Optimization*, 8(4):28:1–28:21.

Saez, J. C., Fedorova, A., Prieto, M., and Blagodurov, S. (2010). A comprehensive scheduler for asymmetric multicore systems. In *Proceedings of the 5th European Conference on Computer Systems*, pages 139–152.

Sawalha, L., Wolff, S., Tull, M. P., and Barnes, R. D. (2011). Phase-guided scheduling on single-ISA heterogeneous multicore processors. In *Proceedings of the 14th Euromicro Conference on Digital System Design Architecture, Methods and Tools*, pages 736–745.

Sherwood, T., Perelman, E., Hamerly, G., and Calder, B. (2002). Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57.

Sherwood, T., Sair, S., and Calder, B. (2003). Phase tracking and prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 336–347.

Sutton, R. and Barto, A. (1998). *Reinforcement learning: An introduction*. MIT Press, Cambridge, MA.

Sutton, R., McAllester, D., Singh, S., and Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12(22).

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44.

Tesauro, G. (1994). Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation*, 6(2):215–219.

Tesauro, G. (2005). Online resource allocation using decompositional reinforcement learning. In *Proceedings of the 20th National Conference on Artificial Intelligence*, pages 886–891.

Thorndike, E. L. and Bruce, D. (1911). *Animal intelligence: Experimental studies*. Transaction Pub.

Van Craeynest, K., Jaleel, A., Eeckhout, L., Narvaez, P., and Emer, J. (2012). Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, pages 213–224.

Wang, Z. and O'Boyle, M. (2009). Mapping parallelism to multi-cores: a machine learning based approach. *ACM SIGPLAN Notices*, 44(4):75–84.

Werbos, P. (1974). Beyond regression: New tools for prediction and analysis in the behavioral sciences.

Witten, I. H., Frank, E., and Hall, M. A. (2011). *Data Mining: Practical Machine Learning Tools and Techniques*. Elsevier, Amsterdam, third edition.

Yan, X., Sawalha, L., McGovern, A., and Barnes, R. (2012). Supporting transparent thread assignment in heterogeneous multicore processors using reinforcement learning. In *Proceedings of the 3rd Workshop on SoCs, Heterogeneous Architectures and Workloads in conjunction with HPCA-18*.