Speculative Distribution of Sub-Threads Across Processors

Patrick Yost patrick.yost-1@ou.edu
IDEA Lab
School of Computer Science
University of Oklahoma, Norman, Oklahoma 73019-6151
Keywords: Multi-core Architecture, Single Thread Improvement, Reinforcement Learning, Whole Program Paths

Abstract

As multi-core architecture continues to dominate the development of new processor technology, effective utilization of processors has come to mean heavy parallelization inside of programs. However, some problems are simply not parallelizable, and these problems have been unable to make any real use of multi-core architecture. Recently, advances in algorithms used to understand the behavior of a thread when it executes have lead to clearer methods of viewing the relation that different parts of an execution share amongst one another. From that knowledge, the direction of a program can be predicted much farther into the future than can be done with current methods. This, in turn, allows for the "speculative distribution" of future portions of a thread to processors on the hope that they will likely be needed. If the predictions are incorrect, the cost is effectively nothing as it used a processor that would have otherwise stood idle, but if the predictions are correct, the program will have successfully completed two different parts of the program simultaneously. Effectively, this achieves partial-parallelization of the targeted process. It is the goal of this work to study this problem in a simplified form, proving that the concept behind using these predictions to inform a learning agent can work in a simulator space where memory dependencies are inconsequential before trying to prove this concept in a more complex simulator space.

1. Introduction

Limits on the amount of power and cooling that can cost-effectively be applied to a processor have sparked a rapid transition to a multi-core architecture dominant processor market. Two- and four-core machines have become the new standard for personal computers, and Apple recently released an eight-core machine. In the future, the trend seems destined to continue as Intel has already begun work on a prototype eighty-core processor.

1.1 Serial Programs and the Multi-Core Machine

As these multi-core machines have become more dominant, the potential power of computers has increased dramatically. In easily parallelizable applications, their potential is fully realized, but serial applications can utilize only a small fraction of the machine's potential.

A serial program is one in which each portion of code to execute is chosen based on the code that is executed before. By that very definition, there is no way to know with certainty what instructions should be run in parallel with the currently executing segment, and that in turn results in a situation where a serial program can run effectively only on a single core. When the processor is made to have a maximum potential output of eighty times that magnitude, the wasted potential will be significant.

1.2 Prediction as a Potential Solution

As pointed out above, the primary challenge in parallelizing a serial program is determining the correct code to execute preemptively. If the correct section is known and memory dependencies allow, then that portion of the program can be executed on another before it is actually reached. Perhaps more importantly, if the processing cores used by the predicted portion of the program would have otherwise been idle, then even a failed prediction effectively costs the system nothing.

Thus, it becomes a question of finding a way to both make predictions and to evaluate those predictions based on the likelihood that they will improve overall performance if assigned to a processor. Since such a thing involves the weighing not only of known rewards if prediction succeeds but also an unknowable opportunity cost if the prediction fails, a reinforcement learning agent should be able to address the difficulties much better than a handmade heuristic.

Preliminary work. Under review by Dr. Amy McGovern (DrAM). Do not distribute.

1.3 Overview

This paper describes a novel approach to making serial programs run faster on multi-core architecture, utilizing recently popularized tactics to collect information about previous executions of the program and applying reinforcement learning to evaluate the usefulness of predictions before using them.

In Section 2, the simulated environment, learning agent, and reward algorithms will be covered in detail, and in Section 3, the experimental process and results will be reviewed. Following that, related works will be discussed and compared to this work in Section 4 before moving on to Section 5 where we will discuss future directions for this work.

2. Implementation of Learning

In this section, we discuss at length the implementation of the learning agent and the environment in which it functions. The simulation consists of tools used to make predictions, the environment, and the learning agent itself.

2.1 Prediction Tools

To make predictions, we implement whole program paths (Larus, 1999), the detailed study of which will be left to the reader. As short summary, whole program paths are formed by first using a tool to instrument the compiled code of a program. In this work, that tool was Microsoft Phoenix. After instrumenting the code and running it, a *trace* is produced. The trace is a lossless collection of every acyclic path and call made in the execution of the program, and as such can be quite large and difficult search for information.

To combat the prohibitive size and lack of clear organization of the trace, the Sequitur data compression algorithm is used on the trace to form a grammar that implies clear hierarchical relationships (Nevill-Manning & Witten, 1997). Treating each symbol as a node in a directed acyclic graph (DAG) and treating nodes on the left hand side of the rule as the parents of nodes on the right hand side of the same rule creates a graphic, navigable visualization of a *whole program path* (WPP). Again, further investigation into this algorithm is left to the reader, but **Figure 1** below may help.

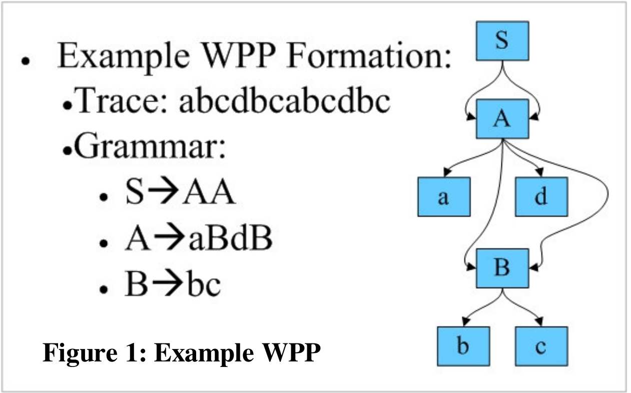


Figure 1: Example WPP

There are two key properties of whole program path formation that prove valuable in this work. First, in the grammar, each *digram*, or pair of symbols adjacent to each other, is unique as a result of the way in which the algorithm works (Nevill-Manning & Witten, 1997). This does not guarantee that every pair in the original trace is preserved, but it does allow for easy search of the whole program path for a probably analogous point to previous execution given only two symbols from the trace or grammar. While there is no guarantee that this point is actually analogous, it is very likely to be, and that is satisfactory for the purpose of speculation.

Second, the resulting grammar is very hierarchical, breaking the execution of the program down into sub-paths that are executed together. Thus, when using this for predictions, predicting one nonterminal symbol in the WPP is equivalent to predicting anywhere from two to several thousand terminal symbols, and these terminal symbols each represent a single continuous acyclic path in the program, and as such each is equivalent to predicting an entire sequence of basic blocks.

2.2 Reinforcement Learning

Reinforcement learning is a very intuitively relatable learning approach. Modeled after early work in psychology, it closely mirrors the concepts of positive and negative reinforcement in human learning. As seen in **Figure 2**, the reinforcement learning paradigm consists of two entities – the "Agent," who makes choices and receives positive and negative reinforcement, and the "Environment," which is everything the agent can perceive or with which it can interact either directly or indirectly. Between these two agents, there is a cyclical series of interactions that can be broken down into three categories. First, the agent is given a summary of the environment's status, called the state. In response to that state, the agent chooses an action to take, changing the environment in some way. Completing the cycle, the environment provides the agent with both a new state that represents the environment after the action's changes have taken effect, and also a reward. The reward is based on the effect the action has had, and can be positive or negative depending on whether the action has brought the agent closer to or farther from its goal.

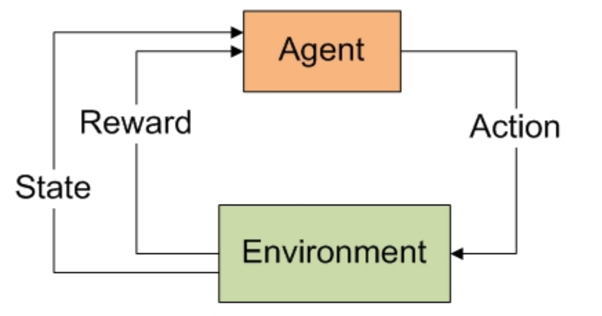


Figure 2: Reinforcement Learning Model

2.2.1 The Environment

In this case, the environment consists of a series of simulated processors that can take predicted values, check them for correctness, and hold on to them for an amount of timesteps equivalent to the size of the prediction given to the processor.

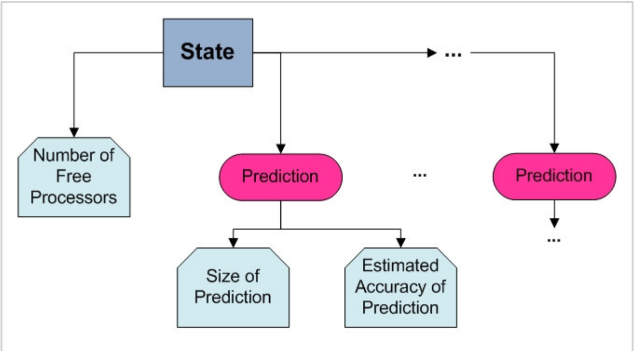


Figure 3: Description of a State

Using information such as the number of processors currently free and characteristics of possible predictions such as size and confidence levels, the environment distills its information into a state, essentially clustering similar values together to make continuous sets of information into discrete sets. For instance, a confidence value between zero and one could be distilled from a continuous real number to a value of either "confident" or "not confident" depending on whether or not it was bigger than a threshold value. In practical application, however, it is beneficial to make somewhat less coarse distinctions.

As shown in **Figure 3**, the state consists of a single record of the number of free processors and information about an arbitrary number of predictions defined at the simulation's runtime.

2.2.2 The Agent

The agent works much like a living organism; given its understanding of its world, in this case the state, it then takes an action in an attempt to improve its position. When it takes this action on the environment, the environment is changed, and the state is given a reward value evaluating the results of this choice.

2.2.3 The Learning Algorithms

(Eq 2.4.1)

$$\boldsymbol{Q \leftarrow Q + \alpha\,(R + D^{T} * MAX(Q's\ of\ Next\ State) - Q)}$$

To update the agent's understanding of its world, **Eq. 2.4.1** is used. This equation updates the value of Q, the most recently completed action taken by the agent. It is from this designation for actions that this type of learning gains its name "Q-Learning." In the equation, α is a smoothing variable between zero and one that helps keep learning more regular and predictable, and R is the reward the agent has been given for taking action Q. To discount the value of the following state to reflect the delay in its contributions, the number of timesteps spent in Action Q is multiplied by a discounting value D raised to the power T. This intuitively makes sense; if an investor can invest to earn a thousand dollars tomorrow or invest with equal certainty in something that will pay out a thousand dollars in a decade, it would be foolish not to take the shorter term investment. Finally, the number by which to multiply the new discounting term is found by taking the maximum value of all the action values in the new state that resulted from Q.

It is important to note that it would be equally valid to label the Q's in **Eq 2.4.1** 'O' because this work is technically Option Learning. Option Learning is a subset of reinforcement learning where actions are allowed to take multiple timesteps, essentially representing numerous tiny actions as one continuous and cohesive action. As a result of the single timestep approach to simple Q-Learning, the exponent T is usually left out of Q-Learning equations.

(Eq 2.4.2)

$$\boldsymbol{R(Q) = \frac{\sum_{t=1}^{T} P_Q(t) + P_{Other}(t)}{T}}$$

Eq 2.4.2 is the first and simplest Reward Function implemented for action Q in this work. Here, $P_x(t)$ is the number of processors executing a correct set of predicted values and assigned by x at time t. In this case, it breaks down simply into those assigned by the action Q and those assigned by another action. T is again the total number of timesteps taken by action Q to completely finish execution.

(Eq 2.4.3)

$$\boldsymbol{R(Q) = \frac{\sum_{t=1}^{T} P_Q(t) + d * P_{Other}(t)}{T}}$$

Eq 2.4.3 is implemented in the second set of simulations discussed in this work. Here, the value d has been added. It is a value between zero and one that is used to discount rewards not directly earned by the action Q

(Eq 2.4.4)

$$\boldsymbol{R(Q) = \frac{Cost + \sum_{t=1}^{T} P_Q(t) + d * P_{Other}(t)}{AssignmentTime + T}}$$

In order to make the task more realistic and more challenging, a cost has been associated with assigning a processor in **Eq. 2.4.4**. This cost takes two forms; first, it is applied as a negative value in the numerator. This forces the prediction to be of a certain size or incur a negative penalty; offsetting this is the fact that a small prediction may help make bigger predictions in a few timesteps. If such is the case, the value of the later opportunities will eventually propagate back to the value of the action at this point. Second, it is applied as a time cost in the denominator of the reward function. This directly corresponds to the fact that assigning a process to a processor takes CPU cycles as well, and helps reflect the fact that, assuming the predictions are all correct, assigning one large prediction will be more efficient than assigning numerous small predictions.

2.2.4 Optimistic Values

Finally, as a means to help encourage exploration early, a technique known as Optimistic Initial Values is used. In this technique, actions are given a positive starting value. Its name comes from the fact that these positive values make unexplored options seem much better, taking an "optimistic" view of the unknown. This approach encourages more aggressive early exploration of the space and can often speed learning.

3. Experiments and Evaluation

This section will go over the methods by which testing data was gathered and the methods through which the agent was tested. Furthermore, it will analyze the results of those tests.

3.1 Hypothesis

Before further describing a scientific manner of testing the learning agent, it is necessary to put forth a more precise hypothesis. It is the goal of this work to prove that a reinforcement learning agent can use the information in a program's WPP to accurately predict large portions of the program's execution ahead of time and, with a very abstractly modeled reward system, actually learn to make rational choices about which predictions to act on and which to wait through.

3.2 Testing Data

To date, this work has depended on actual trace and whole program path data taken from the execution of the SPEC 2005 benchmark GZip. To form a useful input file that was also of a size that made thousands of simulations possible in reasonable amounts of time, randomly chosen portions of an arbitrary, user-defined size were taken and copied from the trace. For reference, the simulator uses a WPP based on the entire trace of the program.

3.3 Results

This section will review the results found in several variations of the simulator space. These simulations vary greatly in the actions allowed and rewards delivered, but in all of them, a 16 processor machine was simulated and the environment was set to create ten predictions every time the agent was to make a choice. It is important to note that while the graph shapes of these simulations can and should be compared, the actual value of rewards should not because their reward functions were different.

3.3.1 The Advanced-Action Simulator

The advanced-action simulator was the first developed in this work. It takes its name from way in which it takes actions based upon predictions. As you can see in **Figure 4**, this simulator actually implemented a fairly complicated action scheme. Each action indicated two separate choices – first, which prediction on which to act, and second, how many ways to split the prediction. Effectively, this agent would take a single prediction and put part of it on each of a number of processors it chose.

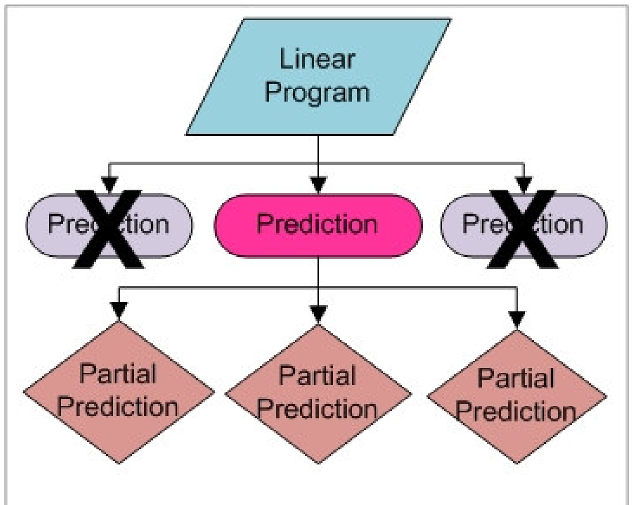
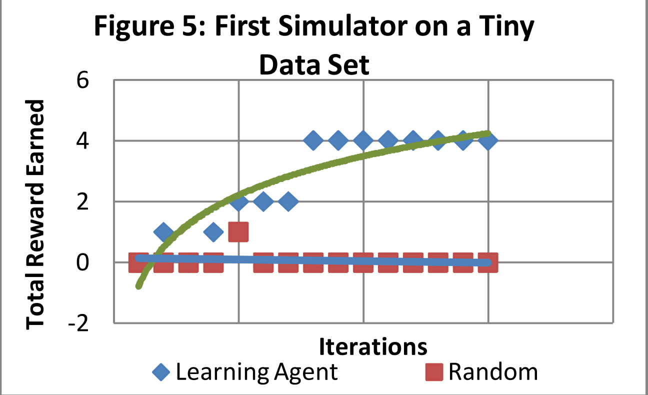
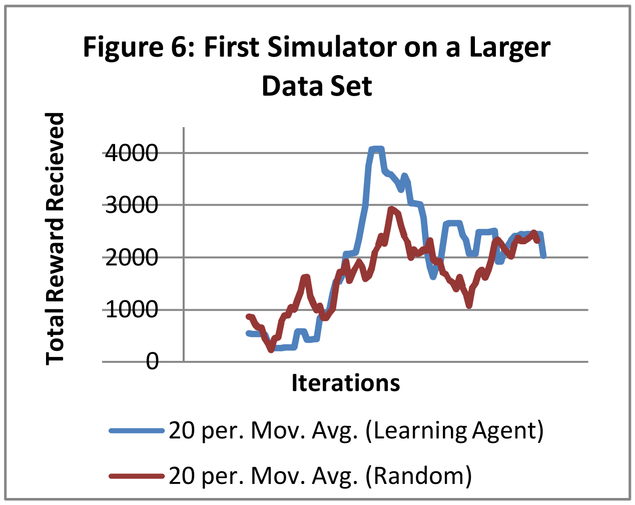


Figure 4: Actions in the First Simulator



As can be seen in **Figure 5**, this agent was readily able to master a very small state space because it used only a tiny set of its states, and its predictions could only be split a few ways because they were so small. However, **Figure 6** shows how it floundered in a larger space, barely outperforming a random-choice agent that had a lucky streak at the end of the simulations. Because this agent worked with an exponentially more complicated state-action space while implementing an overly simplified reward function (Eq. 2.4.2), it couldn't possibly succeed at such a difficult problem. In order to correct the

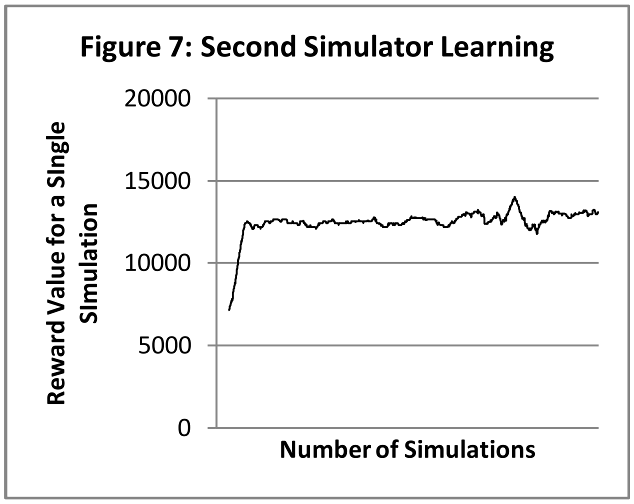


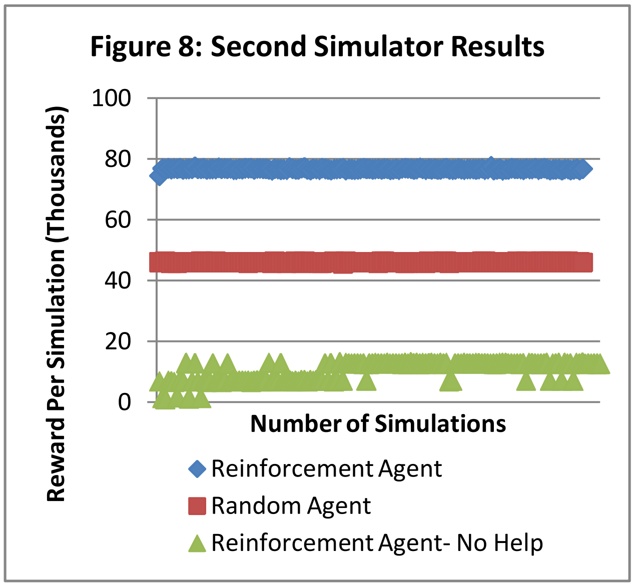
shortcomings of its reward function, work on such an advanced action space was tabled so that the focus could be on correcting and improving its basic learning process.

3.3.2 The Second Simulator- Heuristics

First, this action implemented a new, simpler approach to actions. This agent would only assign a prediction to a single processor; in this way, its potential speed increase is smaller in magnitude but its state-action space is much more reasonably sized. This, combined with an updated reward function (**Eq.2.4.3**), resulted in a more competent learning agent. This can be seen in **Figure 7**.

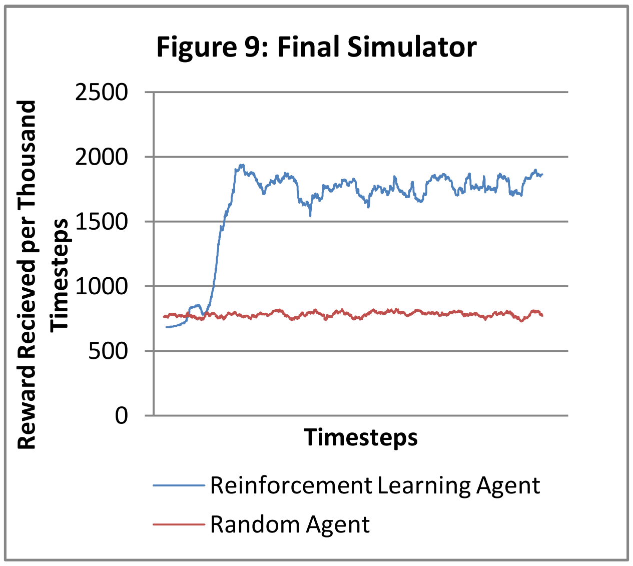
One difficulty the agent encounters in making its full sets of predictions is that it is sometimes impossible to make that many predictions. For instance, an agent can only look so far above its current position in the WPP DAG before there are simply no nodes above it. In designing the random agent for this system, a heuristic was accidentally included that would cause an invalid choice of prediction (i.e. one that was null) to be replaced with a randomly chosen valid one. This gave the random agent an unintended advantage.





To make the situation worse, the new reward function still lacked any sort of penalty for small assignments, so it was easy to get the highest possible reward. Any correct prediction was as good as predicting all its sub-parts individually, meaning that there was little to learn for the agent. In **Figure 8**, there are two entries for the learning agent; in the first, it has the benefit of the same heuristic advantages as random, and in the second, it has none of those advantages. The second includes a much more obvious representation of a learning than the one given the heuristic advantage, even if it is completely outperformed in this scenario. This is evidence that the problem that the agent is seeing is still too simple when it is given the heuristic and a reward function that is so easily satisfied; the learning agent maximizes its reward almost immediately.

3.3.3 The Final, Detailed-Reward Simulator



In the final simulator, a more detailed reward function was implemented (**Eq 2.4.4**). This forced the agents to deal with more realistic and complex issues when choosing actions. Furthermore, the heuristic protecting both the random and reinforcement learning agents was removed completely and replaced with a small penalty assessed every time an invalid choice was made. This represents the missed cycles spent looking for nonexistent predictions.

The results in this more realistic space were much more impressive. As seen in **Figure 9**, the learning agent masters its space fairly rapidly using optimistic values, and its performance dominates that of the random agent.

4. Related Works

This section reviews related works in segments categorized by their applicability to the problem at hand. First, it reviews other works in the field of Machine Learning, then it looks into other views on predictions for the purposes of parallelization, and finally it closes by examining works related to the organization and accessibility of data that could be useful in this work's prediction scheme.

4.1 Machine Learning

Past reinforcement learning works yielded agents that could do basic block scheduling on a processor capable of taking two instructions at once (McGovern, Moss, & Barto, 2002). Furthermore, these agents could often outperform the commercial compiler to which they were compared, and in doing so, they proved the viability of reinforcement learning agents in the field of instruction scheduling.

There are two primary differences between the work of McGovern, Moss, and Barto, and the work discussed here. First, from a machine learning standpoint, this work uses a much less supervised method. They had a set of minimally-correct answers to problems supplied by the commercial compiler and could judge success or failure from those. In this work, the value of an action must be judged solely by the observable effects of that action.

Second, from a practical standpoint, the limitations and potential of each are completely different. Their work provides a direct increase in speed on a small number of cores, but as presented, it would be very hard to use practically on a large number of cores. Only a very limited amount of parallelization can be performed on a serial program while still respecting the linear path it must follow from one section of code to another. In contrast, the work presented herein seeks to circumvent those limitations by allowing portions of a program to be speculatively scheduled, and that provides the opportunity for improvements in speed with little or no risk to the system's performance should the agent fail.

It is important to note that because these works operate on two different mechanisms and at two separate scales, they might conceivably be used simultaneously, allowing each part of the code assigned by this work's agent to be accelerated as it executes. However, since the other work operate on a static compiler before execution, it would remain to be seen how well it would work in conjunction with this work's more dynamic approach.

4.2 Branch Prediction

In his work "Analysis of Branch Prediction via Data Compression," Chen states that most prediction schemes in use today for branch prediction are in use for the purpose of prefetching, and most of them break down to simple Prediction by Partial Matching (PPM). Since PPM is effectively accomplished using Markhov chains of indefinite size, it is cumbersome and impractical for most applications. Thus, in real applications, it is often approximated using two-level branch prediction, where two symbols are used to predict a third (Chen, Coffey, & Mud, 1996). This can be repeated, taking the second and third to predict a fourth, and so on.

PPM has a distinct downside. Since it is predicting the next block based solely on the small set of tokens before it and then using the those to make another in a chain of predictions, there is no structure to the data. Without structure, it is easy to see that large predictions are much more challenging as their chance of correctness becomes the product of the accuracy of every prediction made in the chain. Furthermore, the very mostly likely chain as defined by the algorithm may be the one that executes the least often (Larus & Ball, Efficient Path Profiling, 1996). Such limitations are unimportant when used in a prefetching algorithm because the size of the prefetched prediction will be small and new predictions will be made frequently. On the other hand, they would be crippling for this work.

Here, the first prediction is made in a mannner very similar to two-level branch prediction, but once the prediction engine has an idea of its location in the WPP, it can infer large segments of code that can be executed together.

4.3 Data Processing

There have been a number of works in the realm of data processing that have affected this work or that might affect it in its future developments. Most of these revolve around WPPs as developed by Larus, which were already in their basic form earlier in this paper.

Several works have centered on improving the ease of interaction for the data in a WPP. One work foregoes the use of Sequitur to instead use a compaction algorithm that makes data flow analysis more practical (Zhang & Gupta, 2001). This might prove useful if we could apply static recompilation to programs based on the agent's experience on the last run of the program.

A more directly applicable discovery can be found in a work that describes in detail a way to build Extended Whole Program Paths (Tallam, Gupta, & Zhang, 2005). These EWPPs can help trace memory dependencies as the program operates, which is vitally important to the future of this work. Without a way to trace memory dependencies, there is no way this approach could ever be practically applied to real-world problems.

5. Conclusions and Future Work

This work has shown compelling evidence that learning, in conjunction with new ways of saving, collecting, and exploring runtime data from a process, can utilize free cores in multi-core architecture to provide speed enhancement to previously unparallelizable processes. However, there is more research to be done before this can be said for certain. Before moving forward, research in more realistic environments and on a broader range of data will be necessary.

It would also be interesting to try to implement this on different scales. Work should be done to see if some or all of this procedure can be applied to job scheduling on a network or block scheduling on multi-core architecture, for instance.

Perhaps most compelling is the idea of combining the functionality of the agent with other learning agents dividing responsibilities hierarchically. For instance, it would be interesting to see what could be accomplished by combining the results of this agent with an agent that can statically or dynamically recompile the program. Of even greater interest would be expanding the responsibilities of the agent to include scheduling multiple processes.

In short, with results that are very promising but still limited as of yet, there is a great deal more work to done in this area.

Acknowledgments

I would like to thank Dr. Amy McGovern and Dr. Ron Barnes for their help and guidance in this endeavor. I would also like to thank all members of the IDEA lab at the University of Oklahoma for their support. This research is partially based upon work funded by a 2008 Power-Aware Computing Award from Microsoft Research.

References

Chen, I.-C. K., Coffey, J. T., & Mud, T. N. (1996). Analysis of Branch Prediction via Data Compression. *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, (pp. 128-137). Ann Arbor.

Larus, J. R. (1999). Whole Program Paths. *ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, (pp. 259-269). Atlanta.

Larus, J. R., & Ball, T. (1996). Efficient Path Profiling. *Proceedings of the 29th Annual International Symposium on Microarchitecture*, (pp. 46-57).

Law, J., & Rothermel, G. (2003). Whole Program Path-Based Dynamic Impact Analysis. *International Conference on Software Engineering*, (pp. 308-318). Portland.

McGovern, A., Moss, E., & Barto, A. G. (2002). Building a Basic Block Instruction Scheduler with Reinforcement Learning and Rollouts. *Machine Learning* , 141-160.

Nevill-Manning, C. G., & Witten, I. H. (1997). Identifying Hierarchical Structure in Sequences: A Linear-Time Algorithm. *Journal of Artificial Intelligence Research 7* , 67-82.

Tallam, S., Gupta, R., & Zhang, X. (2005). Extended Whole Program Paths. *Parallel Architectures and Compilation Techniques*, (pp. 17-26). St. Louis.

Zhang, Y., & Gupta, R. (2001). Timestamped Whole Program Path Representation and its Applications. *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, (pp. 180-190). Snowbird.