Smart Scheduling: Optimizing Tilera's Process Scheduling via Reinforcement Learning

ARCHIVES

by

Deborah Hanus

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2013

© Massachusetts Institute of Technology 2013. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
February 5, 2013

Certified by.....
David Wingate
Research Scientist
Thesis Supervisor

Accepted by..............
Dennis Freeman
Chairman, Department Committee on Graduate Theses

Smart Scheduling: Optimizing Tilera's Process Scheduling via Reinforcement Learning

by

Deborah Hanus

Submitted to the Department of Electrical Engineering and Computer Science
on February 5, 2013, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Electrical Engineering and Computer Science

Abstract

As multicore processors become more prevalent, system complexities are increasing. It is no longer practical for an average programmer to balance all of the system constraints to ensure that the system will always perform optimally. One apparent solution to managing these resources efficiently is to design a self-aware system that utilizes machine learning to optimally manage its own resources and tune its own parameters. Tilera is a multicore processor architecture designed to highly scalable. The aim of the proposed project is to use reinforcement learning to develop a reward function that will enable the Tilera's scheduler to tune its own parameters. By enabling the parameters to come from the system's "reward function," we aim eliminate the burden on the programmer to produce these parameters. Our contribution to this aim is a library of reinforcement learning functions, borrowed from Sutton and Barto (1998) [35], and a lightweight benchmark, capable of modifying processor affinities. When combined, these two tools should provide a sound basis for Tilera's scheduler to tune its own parameters. Furthermore, this thesis describes how this combination may effectively be done and explores several manually tuned processor affinities. The results of this exploration demonstrates the necessity of an autonomously-tuned scheduler.

Thesis Supervisor: David Wingate
Title: Research Scientist

Acknowledgments

I would like to thank my advisor and mentor, David Wingate, for his wisdom, patience, and support. His incredibly positive outlook was refreshing and motivating, even when things did not go as planned. I appreciate the time resources, and ideas that Profs. Anant Agarwal, Josh Tenenbaum, and Alan Willsky contributed to this project. Sharing workspace with other inference-minded grad students in the Stochastic Systems Group and the Computational Cognitive Science Group made the time I spent working twice as productive and three times as entertaining (especially on days when cbreezy came to lab). Additionally, Hank Hoffman and Jonathan Eastep's conversations brought me up to speed on the intricacies of the TILE64 architecture.

I appreciate the continued support and mentorship of my undergraduate research advisors, research advisors Profs. Nancy Kanwisher and Ed Vul. Without Ed's introduction to David, I may not have been able to find a Master's thesis that combined my interests in machine learning and computer systems.

Finally, I would like to thank my friends and family for their love and support.

Contents

1 Vision **13**

1.1 Why a Smart Scheduler? 13

1.2 Summary 14

2 Previous Work **17**

2.1 Advances in Multicore Processing 17

2.2 Why Reinforcement Learning? 19

2.3 Reinforcement Learning in Systems 20

3 Tools **23**

3.1 Reinforcement Learning 23

3.1.1 Overview 23

3.1.2 Off-policy v. On-policy learning 24

3.2 TILE64^{TM} Architecture 25

4 Problem Statement **27**

4.1 Phase 1: Learning 27

4.2 Challenges 28

4.3 Phase 2: Controlling 28

4.4 Phase 3: Scheduling 29

4.5 Challenges 29

5 Measuring performance **31**

6 **Results** **33**

6.1 Default 34

6.2 Stripes 35

6.3 Minimum Distance from DRAM 35

6.3.1 Strips 36

6.3.2 Tiles 37

6.4 Scattered Blocks 37

6.5 Streamlined Blocks 38

6.6 Summary 39

7 **Concluding Remarks** **41**

7.1 Contributions 41

7.2 Future Directions 42

List of Figures

3-1 Reinforcement learning uses the information acquired from online observations to plan an optimal set of actions. For example, this robot uses information he observes from his state space, such as the weather, obstacles, and potential reward, to plan an optimal set of actions to navigate his environment [43]. 24

3-2 TILE64^{TM} is a 64-core processor developed by Tilera, which holds an 8x8 fully connected mesh of 64 "tiles," which contain a processor, cache, and non-blocking router [7]. 25

4-1 An autonomous system must be able to *observe* the state of the system; use a policy to *decide* how to act; and make then *act* to make the next state of the system [6]. 29

4-2 In the operating system, the reinforcement learning algorithm will uses the information acquired from online observations in the scheduler to plan an optimal set of actions, its policy. 30

5-1 Schematic of one-stage of our eight-stage pipelined benchmark 31

6-1 (left) Each of the processors 64 cores will be arranged in the **Default** configuration, so that the threads for each stage are scheduled by default. (right) Performance of the default configuration, measured by runtime as the number of worker threads per stage varies. 34

6-2 (left) Each of the processors 64 cores will be arranged in the **Stripes** configuration, so that the Stage 1 threads, execute on cores 0-7, Stage 2 on 8-15, etc. (right) Performance of the Stripes configuration, measured by runtime as the number of worker threads per stage varies. . 35
6-3 (left) Each of the processors 64 cores will be arranged in the **Minimum Distance Strips** from DRAM configuration, so that the threads for each stage are scheduled as shown. (right) Performance of the minimum distance configuration, measured by runtime as the number of worker threads per stage varies. 36
6-4 (left) Each of the processors 64 cores will be arranged in the **Minimum Distance** from DRAM configuration, so that the threads for each stage are scheduled as shown. (right) Performance of the minimum distance configuration, measured by runtime as the number of worker threads per stage varies. 37
6-5 (left) Each of the processors 64 cores will be arranged in the **Scattered Blocks** configuration, so that the threads for each stage is scheduled as far away as possible from its adjoining stages. (right) Performance of the default configuration, measured by runtime as the number of worker threads per stage varies. 38
6-6 (left) Each of the processors 64 cores will be arranged in the **Streamlined Blocks** configuration, so that the threads for each stage is scheduled as far away as possible from its adjoining stages. (right) Performance of the default configuration, measured by runtime as the number of worker threads per stage varies. 39
6-7 Comparison of all CPU affinity configurations. Performance of the default configuration, measured by runtime as the number of worker threads per stage varies. 39

List of Tables

Chapter 1

Vision

1.1 Why a Smart Scheduler?

The increasing need for faster computers with intense computational powers dictates that the concurrent processing capability of multicore processors not only become an integral component of current systems research but also continue to escalate through the future [26]. As multicore processors become more prevalent, system complexities increase. It is no longer practical for an average programmer to balance all of the system constraints to ensure that the system will always perform optimally. One apparent solution to managing these resources efficiently is to design a self-aware system that utilizes machine learning to optimally manage its own resources and tune its own parameters [17].

Recent advances in multicore processing now enable researchers to simulate programs using thousands of cores [25]. In practice, however, not all of these cores operate identically. Each of these cores may be running separate processes with disparate loads and latencies. Manufacturing imperfections may even cause asymmetry among the abilities and specifications in each of the cores themselves. Nonetheless, to fully harness the scalability of multicore processors, we must develop some way to operate all cores concurrently, so that the entire system may function optimally.

To achieve this objective, the scheduling algorithm of traditional Unix operating systems must fulfill several competing objectives. It must have a fast process re-

sponse time, good throughput for background jobs, avoidance of process starvation, reconciliation of the needs of low- and high-priority processes, and more. Optimally tuning the parameters of a well-designed scheduler can massively speed up almost any operating system [10].

The aim of our project is to create the tools necessary to enable the Linux scheduler to tune its own parameters. By enabling the system's "reward function" to create the parameters, we eliminate the burden on the programmer to produce these parameters. Additionally, the scheduler will tune its parameters based on its previous "experiences," allowing the parameters to be more specialized than a programmer may produce. Our contribution to this aim is a library of reinforcement learning functions, borrowed from Sutton and Barto (1998) [35] and a lightweight benchmark, capable of tuning processor affinities. When combined, these two tools should provide a sound basis for Tilera's scheduler to tune its own parameters. This thesis describes how this combination may effectively be done and explores several manually tuned processor affinities. This exploration demonstrates the necessity of an autonomous scheduler.

1.2 Summary

This thesis is organized in the following manner. **Chapter 1** lays out the importance of our research, general aims, and the organization of the remaining chapters. **Chapter 2** provides a description of the foundational works upon which our research builds. First, I discuss Tilera's development and how it relates to multicore computing and large-scale data analysis. Next, I discuss some notable attributes of reinforcement learning. Finally, I discuss some successful research projects using reinforcement learning to develop self-aware elements of the computer system. **Chapter 3** discusses reinforcement learning and Tilera in more detail. **Chapter 4** describes one viable approach to developing a self-aware scheduler, given our experimentation. **Chapter 5** discusses the details of our benchmark and performance measurement. **Chapter 6** details our results and briefly discuss their implications. Finally, **Chap-**

ter 7 enumerates our contributions and discuss potential avenues, which may provide fruitful future research.

Chapter 2

Previous Work

2.1 Advances in Multicore Processing

As computer hardware portability and processing power has increased from mainframe to personal computer's, tablets, and even phones that are capable of producing more than the mainframe ever could, the speed of data processing is ever-increasing. Consequently, the amount of data that a user expects to process quickly has also increased exponentially [26]. Theoretically, multicore processors, processors which are capable of processing multiple streams of instructions in parallel, appear to pave the path to the next frontier of data processing capabilities. In practice, however, writing programs that actually utilize this parallel processing capability is extremely challenging. A number of recent advances in memory use, simulation, operating systems, and architecture suggest that efficient, scalable multicore processing is feasible.

First, in the early 1990's, the MIT Carbon Research Group aimed to develop scalable multicore processors built from a conglomeration of single-chip processors. The MIT Alewife machine accomplished this aim by integrating both shared memory and user-level message passing for inter-node communications [2, 1]. Soon, the group's aim evolved. The subsequent Reconfigurable Architecture Workstation (RAW) Project aimed to connect multiple cores via a fully-connected mesh interconnect. The group proved the effectiveness of the mesh and compiler technology, creating the first 16-core processor [41, 38] and setting a standard for future multicore processors. Subsequent

improvements spawned the Tilera, LLC products: a line of massively parallel multicore chips, in which each processor sits on a separate *tile*, and each tile is connected to neighboring tiles [7].

Second, advances in simulation occurred. Building a massively parallel computer with hundreds or thousands of cores before testing potential architecture designs would be extremely costly. Simulation enables researchers to test their designs before building them. Graphite, an open-source parallel multicore simulator, allows exploration of the physical abilities of future multicore processors containing dozens, hundreds, or even thousands of cores. It provides high performance for fast design space exploration and software development for future processors. It can accelerate simulations by utilizing many separate machines, and it can distribute the simulation of a threaded application across a cluster of Linux machines with no modification to the source code by providing a single, shared address space and consistent single-process image across machines [25] .

Similarly, to work efficiently even on massively parallel systems, even operating systems will eventually need to be redesigned for scalability. Wentzlaff, et al. (2009) introduced FOS, a factored operating system, which uses message passing to communicate among its component servers [42]. Each operating system service is factored into a set of communicating servers, which in aggregate constitute a system service. Inspired by Internet service design, these servers provide traditional kernel services, rather than Internet Services and replace traditional kernel data structures with factored, spatially distributed structures. Furthermore, FOS replaces time sharing with space sharing, so that FOS's servers must process on distinct cores and by doing so do not contend with user applications for implicit resources.

In the same vein as these advances, we the constant feedback provided by the huge number of operations performed in a massively parallel system seems to create the perfect scenario to engage a learning algorithm. Applying learning to these systems enables the machine to "learn" from its mistakes to constantly improve its performance.

2.2 Why Reinforcement Learning?

Reinforcement learning, as described by Sutton and Barto (1998), fuses three traditions from various fields [35, 14]. The first is the "law of effect" from the trial-and-error tradition in psychology, which states that "responses that produce a satisfying effect in a particular situation become more likely to occur again in that situation, and responses that produce a discomforting effect become less likely to occur again in that situation [15]." The second is optimal control theory in engineering, a mathematical optimization method for deriving control policies [9]. The third is secondary reinforcement tradition in learning as in classical conditioning [28]. The last is the use of decaying stimulus traces in such works as Hull's (1952) concept of a goal gradient [18] and Wagner's (1981) model of conditioning [40]. This combination has inspired artificial intelligence researchers to create computer algorithms that learn a *policy* that maximizes the *agent*'s long term *reward* by performing a task.

Notably, reinforcement learning does not characterize a learning method, but instead characterizes a learning problem. Reinforcement learning differs from supervised learning, the learning studied statistical pattern recognition and artificial neural networks. Supervised learning models a learning method – learning from examples provided by a knowledgable external supervisor. But what happens if there is no supervisor knowledgable enough to select adequate training examples? If we have a good characterization of the learning problem, like that provided by reinforcement learning, perhaps the system can learn independently[35]. Furthermore, reinforcement learning characterizes two challenges better than other branches of machine learning.

First, reinforcement learning uniquely approaches the trade-off between *exploration* vs. *exploitation*, which is often introduced in the context of the *N-armed bandit.* This introduction poses the problem as an "N-armed bandit," or slot-machine, with N-levers, each of which when pulled provides a *reward*, ranging from 0 to 100 units. The agent wants to develop a *policy* to maximize its reward. At one extreme, if the agent's policy entails only *exploration*, the agent will test each lever. In the case

where many levers exist (e.g, 1000) and only a few (e.g., 5) provide a non-zero reward, the "only exploration" policy will be extremely inefficient. At the other extreme, an agent may choose a policy that employs only *exploitation*. In this case, the agent would select one lever and continuously exploit that lever without trying any of the other levers, while those levers may produce a greater reward. The optimal policy will likely find a balance between exploration and exploitation.

Second, reinforcement learning characterizes the holistic learning problem instead of simply modeling one method of learning, so it may provide us with insights of a more organic type of learning – perhaps demonstrating how real agents in the universe learn to optimize reward. Indeed, in the last 30 years, considerable research suggests that real biological systems, human and animal neurons, utilize a type of reinforcement learning in synapse formation, or Hebbian Learning [20, 21, 22, 23, 29, 31].

Given that we aim to control a large system, given organic data, without anticipating all possible training examples, the potential ubiquity of reinforcement learning is a great asset, and seems the best form of learning to use for this problem.

2.3 Reinforcement Learning in Systems

Reinforcement learning is a branch of machine learning that has been applied to a number of fields for many years. These algorithms are particularly useful when used in high-data contexts, so the policy can be most efficiently tuned [35]. Efficiently utilizing CPU-time in multicore systems has been a major challenge since these systems were developed. Only recently, a number of research groups have employed reinforcement learning to transform standard operating system components, integral to the system's appropriate operation, into "self-aware" components.

First, Hoffman, et al. (2010) utilized this approach on the software level when they presented *Application Heartbeats*, a framework to enable adaptive computer systems. It provides a simple, standard programming interface that applications can use to indicate their performance and system software (and hardware) can use to query an

applications performance. An adaptive software showed the broad applicability of the interface, and an external resource scheduler demonstrated the use of the interface by assigning cores to an application to maintain a designated performance goal [17]. Since *Application Heartbeats* was introduced, several applications have utilized these "heartbeats" to monitor their state. It has been used in multiple learning-based applications to monitor performance [3, 34, 33, 32].

Second, race conditions can arise whenever multiple processes execute using a shared data structure. The standard method of ameliorating this challenge is to use locks that allow only one process access to the protected structure at any given time. Eastep, et al. (2010) introduced an open-source self-aware synchronization library for multicores and asymmetric multicores called Smartlocks [13]. This is a spin-lock library that adapts its internal implementation during execution using heuristics and machine learning to optimize the locks' for a user-defined goal (e.g. performance, optimization), using a novel form of adaptation designed for asymmetric multicores. Their results were encouraging: Smartlocks significantly outperformed conventional and reactive locks for asymmetries among multiple cores [13]. This use of reinforcement learning to develop a self-tuning spin-lock builds upon a larger body of literature, describing reactive algorithms for spin-locks [4, 30, 16].

Similarly, Eastep, et al. (2011) developed *Smart Data structures*, a family of multicore data structures, which leverage online learning to optimize their performance. Their design augmented the Flat Combining Queue, Skiplist, and Pairing Heap data structures with an efficient Reinforcement Learning engine that balances complex trade-offs online to auto-tune data structure performance. They demonstrate significant improvements in data structure throughput over the already surprising efficiency of Flat Combining and greater improvements over the best known algorithms prior to Flat Combining, in the face of varied and dynamic system load [12].

Third, another use of reinforcement learning in computer systems was demonstrated by the *Consensus Object*, a decision engine informed by reinforcement learning. This object employed reinforcement learning to coordinate the behavior and interactions of several independent adaptive applications called *services*. It worked

by first gathering information, analyzing the runtime impact of services in the system. Then given their decision policies and the performance goals of the system, the *Consensus Object* can halt or resume the services behavior. They found their modified system was highly adaptable and performed well in many diverse conditions [39].

Fourth, Ipek et al. (2008) used reinforcement learning in schedulers proposed a self-optimizing memory controller to efficiently utilize off-chip DRAM bandwidth. This is a critical issue in designing cost-effective, high-performance chip multiprocessors (CMPs). Conventional memory controllers deliver relatively low performance in part because they often employ fixed access scheduling policies designed for average-case application behavior. As a result, they cannot learn and optimize the long-term performance impact of their scheduling decisions, and cannot adapt their scheduling policies to dynamic workload behavior. The self-optimizing memory controller observes the system state and estimates the long-term performance impact of each action it can take, enabling it to optimize its scheduling policy dynamically and maximize long-term performance. They showed that their memory controller improved the performance of a set of parallel applications run on a 4-core CMP, and it improved DRAM bandwidth utilization by 22% compared to a state-of-the-art controller [19].

Finally, Most self-aware systems require the learning algorithm to run on a separate process thread, sharing the same resources as the process of interest. The associated context switching creates a high overhead. Recently, Lau, et al (2011) proposed an alternative way to *Partner Cores*. The authors propose that a system be designed so that each high-power *main* core is paired with low-power *partner* core. The self-aware runtime code can run on the partner core, freeing the main core to work only on the process of interest. The partner cores are optimized for energy efciency and size, allowing allows the self-aware algorithms to be run inexpensively, more easily producing a net positive effect [24]

The results of the aforementioned foundational work are extremely encouraging. My research builds upon these findings in order to optimize CPU (Central Processor Unit) utilization in the process scheduler on the Tilera architecture.

Chapter 3

Tools

I aimed to build on the encouraging results of previous work using Reinforcement Learning to provide computer systems with the necessary instructions to better manage themselves, given some feed back. To develop a self-aware process scheduler by applying reinforcement learning to TILE64^{TM}'s process scheduling, I researched available reinforcement learning functions, implemented a library of reinforcement learning functions, and wrote a lightweight benchmark that sets CPU affinities. Here, I review the tools I used, reinforcement learning and Tilera's TILE64^{TM} processor, in more detail.

3.1 Reinforcement Learning

3.1.1 Overview

Reinforcement learning allows an agent to learn and plan by interacting with its environment. Our scheduler also need this ability to plan and learn, suggesting that reinforcement learning may be an optimal tool to solve this problem. The classical formalization of this specialized machine learning technique is that given (a) a state space, (b) an action space, (c) a reward function, and (d) model information, it will find a policy, a mapping from states to actions, such that a reward-based metric is maximized [35, 43].

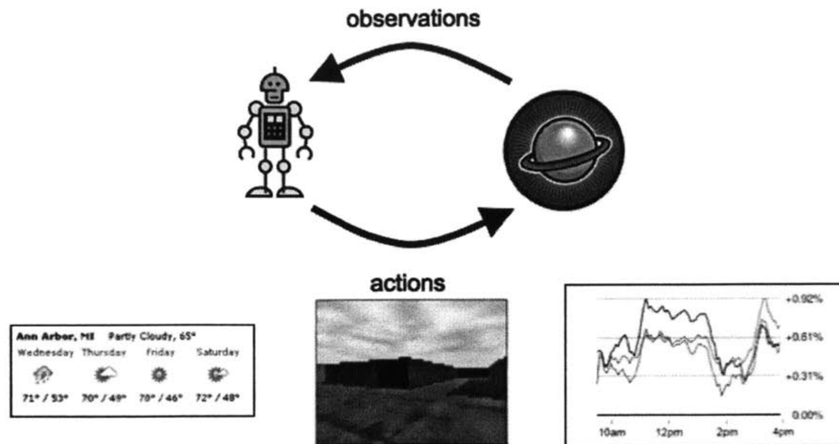


Figure 3-1: Reinforcement learning uses the information acquired from online observations to plan an optimal set of actions. For example, this robot uses information he observes from his state space, such as the weather, obstacles, and potential reward, to plan an optimal set of actions to navigate his environment [43].

Figure 3-1 demonstrates a basic example of an *agent* (e.g. robot) that makes *observations* of a *model* of its environment. The relevant observations it makes (e.g. location, obstacles) make up elements of his *state*. This *reward function* uses this *state* information to dictate the potential reward. Reinforcement learning enables the robot to develop a *policy*, which maps any possible state (e.g. vector of [location, obstacles]) to a reward-metric maximizing *action*.

3.1.2 Off-policy v. On-policy learning

To fully characterize the potential capabilities of reinforcement learning, it is necessary to differentiate between *on-policy* and *off-policy* learning. On-policy methods consider the reward resulting from every possible sequence of states from the start state to the goal, and after evaluating all possible options, it produces an optimal fixed policy, π – a series of state-action pairs the agent can traverse to maximize its reward. The agent behaves according to π, and while doing so it computes the value function, Q, for π. In contrast, in an off-policy method, the agent behaves according to π, but actually computes the value function for π^* – that is, it learns about the optimal value function, Q^*, (and therefore, the optimal policy) while behaving according to a

different policy π. Off-policy learning acts based on the perhaps suboptimal policy, π,evaluates the potential reward of all possible state-action pairs from the current state and then selects the one that produces the highest reward for the optimal policy, π^*. This means that the off-policy learning acts based on only local information. On one hand, this iterative state-action selection is what enables reinforcement to create policies to control real-world systems, because in the real world it is nearly always impossible to see the future. On the other hand, because off-policy methods choose the next state based on local information, it become more likely that the algorithm may not converge to a workable policy. Although there have been several attempts to remedy this problem of potential non-convergence, one of the most common is *function approximation* [5, 11, 37].

3.2 TILE64^{TM} Architecture

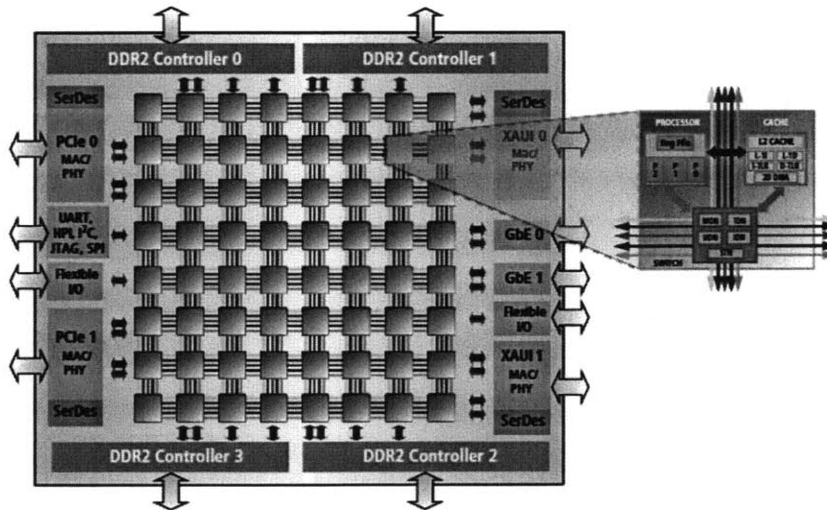


Figure 3-2: TILE64^{TM} is a 64-core processor developed by Tilera, which holds an 8x8 fully connected mesh of 64 "tiles," containing a processor, cache, and non-blocking router [7].

TILE 64 is a 64-core processor, which runs Linux, developed by Tilera. This processor consists of is an 8x8 mesh of 64 tiles, where each tile contains a general purpose processor, cache, and a non-blocking router, used to communicate with the

other tiles on the processor [7]. Learning algorithms are often most useful in high-data contexts, because the algorithm needs many training examples with which to learn the appropriate behavior. The communication among so many tiles provides an ideal environment to test the effectiveness of learning in improving scheduling efficiency.

Chapter 4

Problem Statement

4.1 Phase 1: Learning

The process scheduler of an operating system must fulfill several competing objectives. It must respond quickly, have good throughput for background jobs, avoid process starvation, reconcile the needs of low- and high-priority processes, and more [10]. Because developing a scheduling algorithm that fulfills all of these objectives is a daunting task for most programmers, we aim to create the tools for a scheduler to be able to "learn on the job" from its failures and successes. To this aim, I created a python library of the reinforcement learning functions presented in *Reinforcement Learning: An Introduction* [35] and improvements to those basic algorithms found in our research [36].

Given a state-action pair and a reward function, each of these algorithms should return a policy. To use these functions appropriately, however, we are faced with the challenge of determining what an effective state space is, what corresponding actions they should have, and how they should be rewarded. There are many factors that can affect scheduling efficiency, such as the distance between cores, information recently used by neighboring threads, and cache coherence.

4.2 Challenges

1. **What is the most effective state space?** In an operating system, there are a multitude of characteristics which compose the "state of the system." In order to design an effective reward function, we will need to select a subset of these states that are most likely to affect the variables that we choose to control.

   The state should contain the factors that most efficiently predict our goal. Despite the diverse array of potentially contributing factors, we can only include a few of these in our state to avoid suffering from the *curse of dimensionality.* As the number of our dataset's dimensions increases, our measure of distance and performance become effectively meaningless (See review Parsons, 2004 [27]). Continuing to explore potential states to create an optimal state space would be an important component of future work.

2. **What is the best reward function?** We anticipate the first great challenge in our project to be developing the reward function that we will use to facilitate the scheduler's learning. We will keep the reward function running on a thread in the background as the scheduler performs actions online.

4.3 Phase 2: Controlling

For an autonomous system to control itself, it must have some way of observing its own state, so that it can utilize that information from the policy to decide the most appropriate action to maximize its reward. We represent this, using the **Observe–Decide–Act control loop** (ODA) (Figure 4-1). This ODA cycle differs from that of a non-autonomous system, because a non-autonomous system depends on the programmer to observe the state and decide how she should program the system to act. Figure 4-1 was adapted from [6]

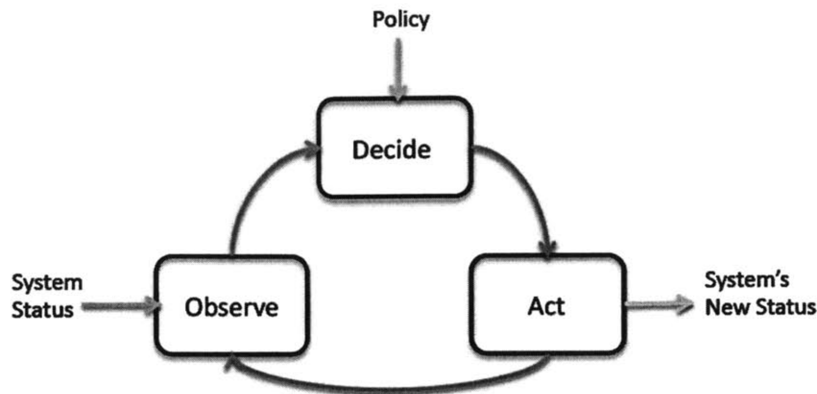


Figure 4-1: An autonomous system must be able to *observe* the state of the system; use a policy to *decide* how to act; and make then *act* to make the next state of the system [6].

4.4 Phase 3: Scheduling

Our aim is to tune processor affinities and arbitrarily set variables in the scheduler and use our reward function to tune these parameters to enable the scheduler to perform more optimally. To do this we must develop an interface over which the reinforcement learning and the scheduler can communicate (Figure 4-2).

4.5 Challenges

1. **How will the reward function interact with the scheduler?**

   To use the reinforcement learning to modify processor affinities, we must develop an interface over which the process thread running the reinforcement algorithm and the scheduler will be able to communicate. To create this interface, the reinforcement learning algorithm runs on a background thread, adjusting the affinity of each process thread to a given processor. This is a vital element of future work.

2. **How will we benchmark the system's performance?**

   We benchmark the system's performance, using an pipelined benchmark, mod-

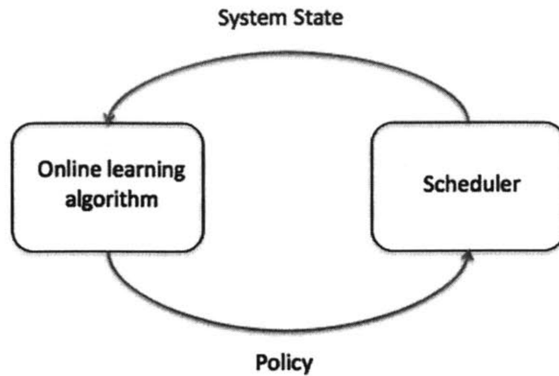


Figure 4-2: In the operating system, the reinforcement learning algorithm will uses the information acquired from online observations in the scheduler to plan an optimal set of actions, its policy.

eled after the Ferret package of PARSEC, a well-established computer benchmark for computer architecture [8]. This benchmark consists of an 8-stage pipeline. Each stage has a workpile and some number of worker threads, which perform "work" by "processing" an image. To "process" each image, the worker thread must increment 256*256 counters.

In the results, I plot the benchmark's performance as we modify the number of threads per stage.

Chapter 5

Measuring performance

We measured performance using a benchmark based of off the Ferret package of the PARSEC benchmark suite [8]. The Ferret package measures performance by pipelining the processing of several images. In each stage of the pipeline, a different part of the image processing is performed. Our benchmark imitates this behavior and can set processor affinity, so that certain threads run on certain processors.

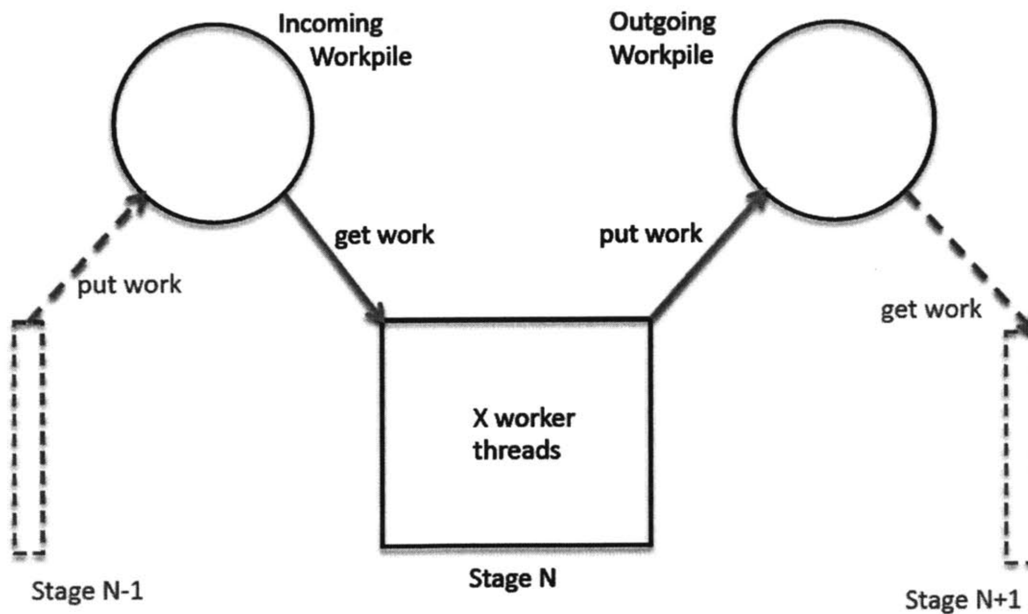


Figure 5-1: Schematic of one-stage of our eight-stage pipelined benchmark

In our benchmark, each of the eight stages of our benchmark contains eight of the

64 available processing tiles on which the threads in that stage may execute. Each stage of the pipeline "does work" by incrementing a 256*256 counter. This benchmark has N stages, where we chose $N = 8$. To understand the details of our benchmark, it is useful to discuss some of our data structures and terminology.

- **Stage**: A sequence of processes, including getting work from the incoming work pile, doing the work, and putting that work on the outgoing work pile. The first stage creates the work and each of the worker threads, because it has no incoming work pile. The final stage has no outgoing work pile, so it finalizes each image and frees the corresponding data structures.

- **Work Pile**: There are $N - 1$ work piles shared by each pair of stages (because the first stage has no incoming pile and the last stage has no outgoing pile). Each work pile can hold a certain amount of work. It can be accessed by the functions "get work" and "put work."

- **Worker Threads**: Each stage of our eight-stage pipeline contains a certain number of worker threads, M, which access the work piles and do work. We implemented these using pthreads.

In the next section, we discuss the results that we obtained using this benchmark.

Chapter 6

Results

All of the reported results show an 8-stage pipelined benchmark. Each of the eight stages contain eight of the 64 available processing tiles on which the threads in that stage may execute. Each stage of the pipeline "does work" by incrementing a 256*256 counter. We found that its performance scaled with the amount of work it was given, so all of these results were collected, using 5,000 units of work.

Initially, we expected that there would be a hand-coded optimal configuration which performs better than the default configuration. In our exploration of hand-coded configurations, we found that our hand-coded configurations were often outperformed by the default. We expect that this is because our hand-coded configurations do not dynamically utilize all possible cores. By using reinforcement learning to tune the scheduling affinity, thus allowing dynamic allocation, it may be possible for the reinforcement learning to outperform the default configuration. We leave this to future work.

Here, we investigate the TILE64 processors performance given a range of hand-coded configurations. In the first several sections of this chapter, I discuss our motivation for choosing each individual configuration and my findings. In the final section, I compare and discuss these results with respect to one another.

6.1 Default

When we run our benchmark using the default processor affinity, meaning that the processor dynamically allocates which threads are allowed to run on which processor. The y-axis shows the benchmark's runtime. The x-axis varies the number of worker threads in each stage. For example, when there is one worker per stage, there are 8 total workers (one per stage), performing work within the pipeline, and when there are 28 workers per stage, there are 224 total threads working over all of the eight stages.

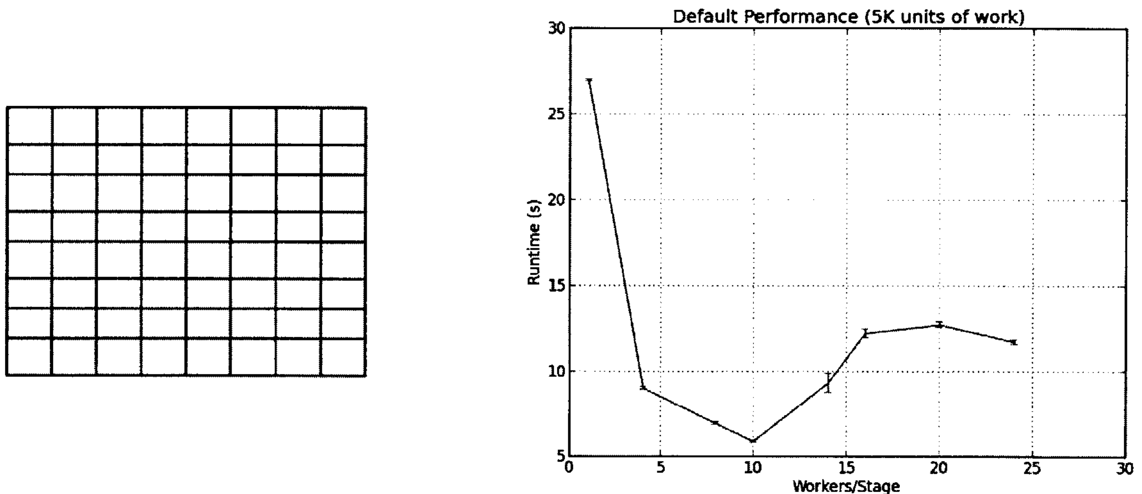
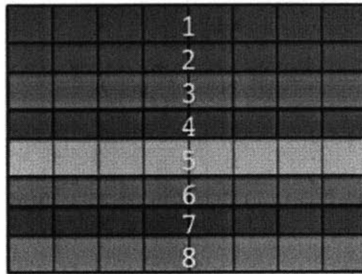


Figure 6-1: (left) Each of the processors 64 cores will be arranged in the **Default** configuration, so that the threads for each stage are scheduled by default. (right) Performance of the default configuration, measured by runtime as the number of worker threads per stage varies.

The default configuration is unique among our configurations in that any of the work can be scheduled on any of the available 64 tiles. Taking full advantage of this processing power is important to create an optimally efficient scheduling.

6.2 Stripes



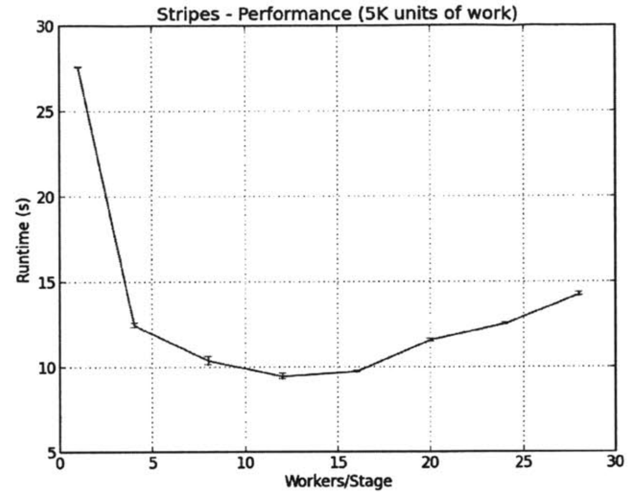


Figure 6-2: (left) Each of the processors 64 cores will be arranged in the **Stripes** configuration, so that the Stage 1 threads, execute on cores 0-7, Stage 2 on 8-15, etc. (right) Performance of the Stripes configuration, measured by runtime as the number of worker threads per stage varies.

Here we aimed to best the default implementation, by making it easy for the workers in each phase of the pipeline to pass their completed work to the next stage of the pipeline. To make the passing of work easier, we put each stage of the pipeline in sequence. Interestingly, in this configuration, the processors' performance was actually worse than default, with Stripes best performances at 10s and default's best performance at 5s per stage. We expect that is because each stage had only eight tiles on which to perform their work rather than sixty-four. Thus, we expect this pattern of decreased performance to repeat through many of our other hand-coded configurations.

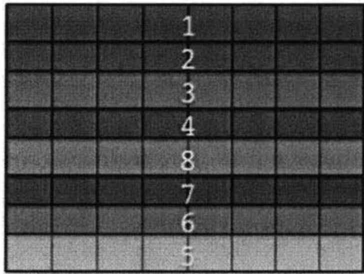
6.3 Minimum Distance from DRAM

Next, it seemed that minimizing the distance between the Dynamic Random Access Memory (DRAM) and the first and last stage, where most of the memory reads and

writes occur, may enhance the processor's performance. We tried this in a 8 tile by 1 tile strip configuration (analogous to the "Stripes" configuration) and in tile-by-tile configuration, assigning the eight stages to 8 blocks of 2 tiles by 4 tiles. Minimizing this distance did not have the effect that we expected, as this was the worst performing of the configurations we explored.

6.3.1 Strips

We minimized the distance from the DRAM, using an eight 8 tile by 1 tile strip. We found that this configuration's performance curve behaved more like that of the "Stripes" configuration, although its overall performance was significantly more depressed. The strips configuration's best performance was 12s, while the stripes configuration best was 10s.



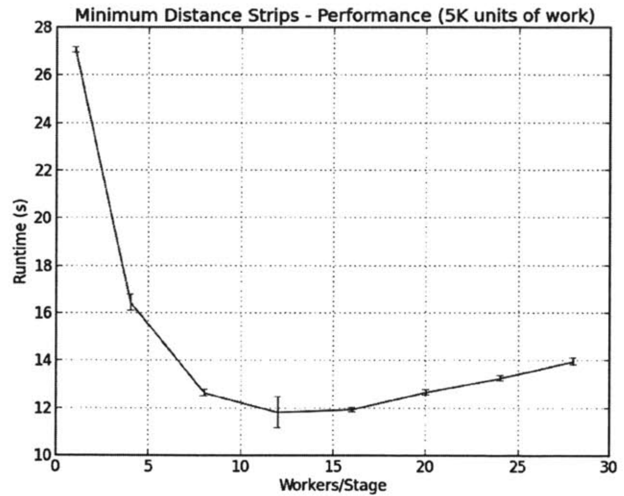


Figure 6-3: (left) Each of the processors 64 cores will be arranged in the **Minimum Distance Strips** from DRAM configuration, so that the threads for each stage are scheduled as shown. (right) Performance of the minimum distance configuration, measured by runtime as the number of worker threads per stage varies.

6.3.2 Tiles

We minimized the distance from the DRAM on a tile-by-tile basis, assigning the eight stages to 8 blocks of 2 tiles by 4 tiles. We found this to be the worst-performing configuration, never performing better than an average of 14 seconds.

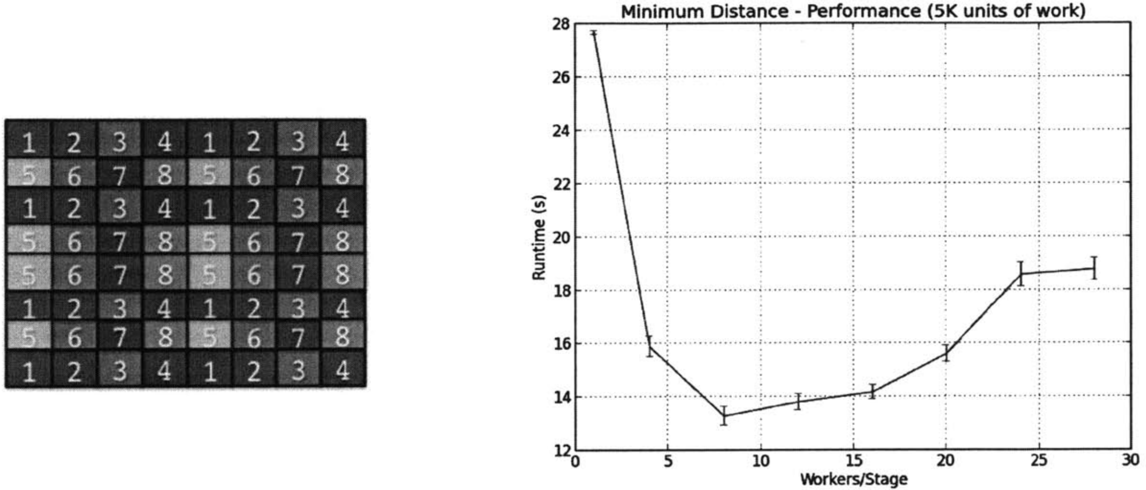


Figure 6-4: (left) Each of the processors 64 cores will be arranged in the **Minimum Distance** from DRAM configuration, so that the threads for each stage are scheduled as shown. (right) Performance of the minimum distance configuration, measured by runtime as the number of worker threads per stage varies.

We expect this poor performance results from costly switching between tiles located far from one another. For example, if one of the Stage 1 cores needs to switch one of its threads to another core, it would not be able to move it to a neighboring core, but would be forced to switch to a core a minimum of 4 cores away. This switch takes a lot of time relative to other operations, and so it is extremely costly.

6.4 Scattered Blocks

By manually coding the Scattered Blocks configuration, we aimed to create a pessimally performing configuration. We were surprised to find that this configuration actually performed quite a bit better than several of our other configurations.

We expect that it performed better than the tile-by-tile Minimum Distance from DRAM configurations, because it was less costly for workers within a single stage to switch their work to a neighboring tile than to switch to a tile that is at least four tiles away. We expect that it performed worse than Stripes because of the time required to switch between stages that were positioned on opposite sides of the chip.

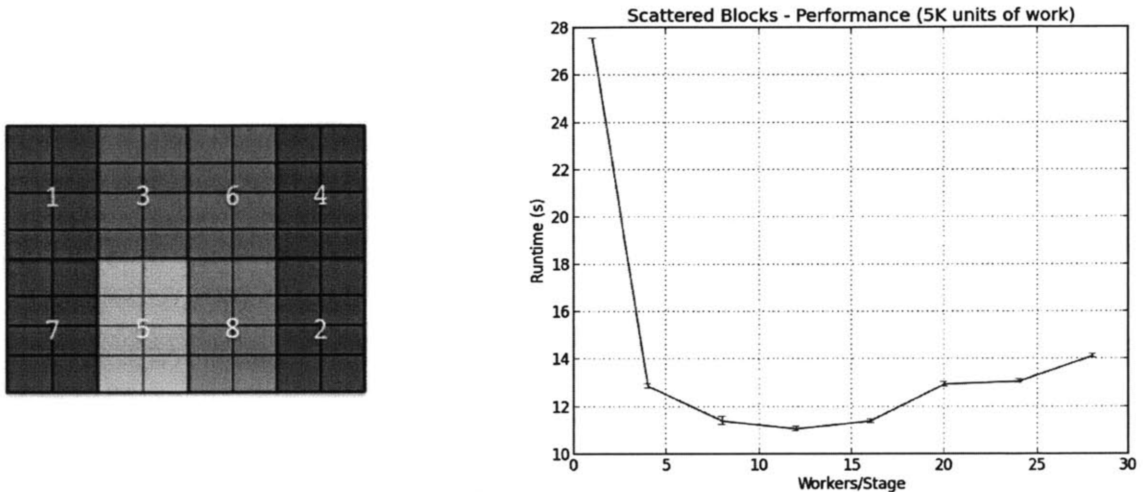


Figure 6-5: (left) Each of the processors 64 cores will be arranged in the **Scattered Blocks** configuration, so that the threads for each stage is scheduled as far away as possible from its adjoining stages. (right) Performance of the default configuration, measured by runtime as the number of worker threads per stage varies.

6.5 Streamlined Blocks

Given that we expected the Scattered Blocks to be pessimal, we were surprised to find that its performance was the third best overall. To further explore why this might be, we developed the Streamlined Blocks configuration, which draws upon both the physical adjacency of adjacent stages of the Stripes configuration and the block configuration of the Scattered Blocks. We expect this to perform similarly to the Stripes configuration, and it does. In fact, we find the Streamlined Blocks configuration appears to be functionally equivalent to the Stripes configuration.

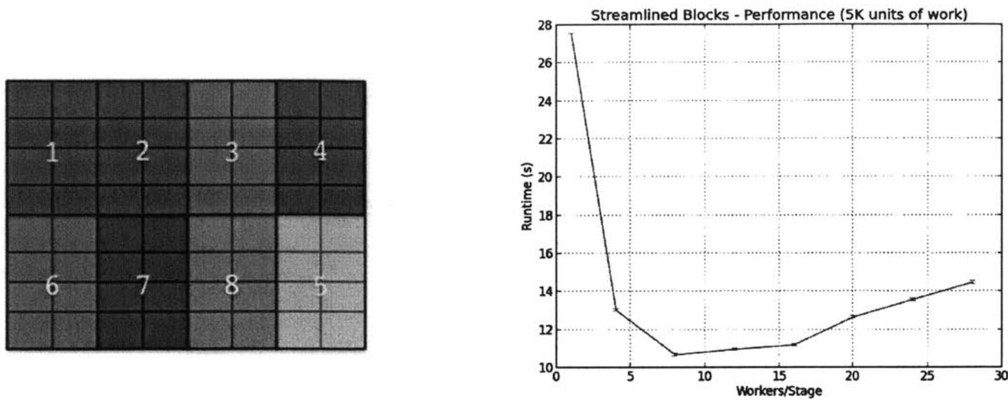


Figure 6-6: (left) Each of the processors 64 cores will be arranged in the **Streamlined Blocks** configuration, so that the threads for each stage is scheduled as far away as possible from its adjoining stages. (right) Performance of the default configuration, measured by runtime as the number of worker threads per stage varies.

6.6 Summary

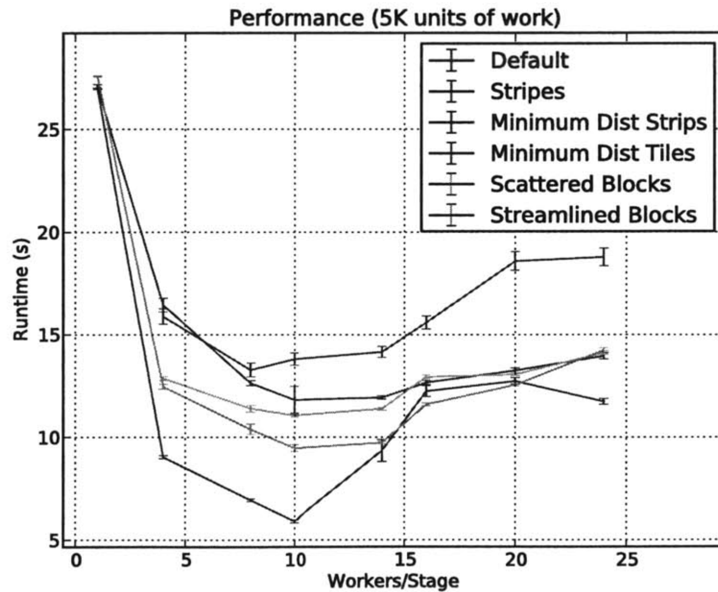


Figure 6-7: Comparison of all CPU affinity configurations. Performance of the default configuration, measured by runtime as the number of worker threads per stage varies.

Figure 6-7 shows each configuration discussed in our results on one plot for conve-

nient comparison. There are several lessons that we can take from this information.

- Determining an optimal scheduling manually is extraordinarily difficult for a human programmer, meaning that it is important to make the scheduling completely autonomous. Reinforcement learning seems like a promising venue through which we can improve this scheduling.

- Fully utilizing all available cores offers an important opportunity to improve scheduling efficiency. Our manually set affinities did not allow dynamic load balancing among all 64 cores, but only among the eight cores within a given stage, resulting in decreased performance.

- Switching among non-neighboring cores is significantly more costly than switching among neighboring cores. As long as the cores are able to balance their load among neighboring cores, they appear to perform quite well.

- Notably, this cost of balancing among cores appears to outweigh the actual configuration of the blocks. This is shown by the fact that Stripes and Streamlined Blocks appear to be completely equivalent to one another, despite differing physical arrangements.

Chapter 7

Concluding Remarks

7.1 Contributions

In summary, my thesis makes the following contributions:

- Implemented a Python library of reinforcement learning algorithms, as described in Sutton & Barto (1998) [35].

- Implemented a lightweight pipelined benchmark to run on Tilera, modeled after the Ferret package of PARSEC [8].

- Determined a plausibly appropriate learning algorithm: Gradient Temporal Difference Algorithm [36].

- Demonstrated that it is extraordinarily difficult for humans to determine what scheduling will be fastest, thus confirming the need for an autonomous scheduling system.

- Demonstrated a significant difference between the performance of different scheduling policies, suggesting that there is space for machine learning to improve scheduling policies.

7.2 Future Directions

When asking a question that is large enough to be interesting, one of the largest difficulties is scoping the problem, such that it can be finished in a limited time. As my research draws to a close, there still exist several interesting and compelling ways in which this research could be extended.

- ***Use Reinforcement Learning to Tune Processor Affinities.*** In this thesis, we have set forth many of the tools necessary to create a smart scheduler, a library of reinforcement learning algorithms, and a benchmark that can set processor affinities. Combining these to create a autonomous process scheduler would be an exciting future addition to this work.

- ***Continue Exploring State Variables.*** Even a simple modern computer has a nearly infinite number of potential state variables. These state variables range from local process thread configuration to installed programs, running programs, processor type, operating system, pixels on the computer monitor, and even external entropy caused by the computer's storage room. Did we choose the best of these variables to represent the state of the processor? We identified a subset of state variables that give us information we need to improve scheduling; however, there may exist another subset of state variables that better represent the information necessary to predict a maximally optimal scheduling.

- ***Choosing an Appropriate Policy*** We chose to use the Gradient Temporal Difference (GTD) algorithm for two reasons:

  1. As a Temporal Difference algorithm, GTD allows the scheduling policy to be updated on each iteration of the algorithm.
  2. The textbook Temporal Difference algorithm does not guarantee convergence. The GTD algorithm uses gradient descent to guarantee that it will converge to a locally optimal policy.

As we determine the most applicable state space, we may also find that another policy may serve may perform better. This is another area that would be rich for future additions.

Bibliography

[1] A. Agarwal, R. Bianchini, D. Chaiken, F.T. Chong, K.L. Johnson, D. Kranz, J.D. Kubiatowicz, B.H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine. *Proceedings of the IEEE*, 87(3):430–444, 1999.

[2] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B.H. Lim, G. Maa, and D. Nussbaum. The mit alewife machine: A large-scale distributed memory multiprocessor. *Scalable shared memory multiprocessors*, pages 239–262, 1991.

[3] A. Agarwal, M. Rinard, S. Sidiroglou, S. Misailovic, and H. Hoffmann. Using code perforation to improve performance, reduce energy consumption, and respond to failures. 2009.

[4] T.E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 1(1):6–16, 1990.

[5] L. Baird et al. Residual algorithms: Reinforcement learning with function approximation. In *Machine Learning International Wokshop*, pages 30–37. Morgan Kaufmann Publishers, Inc., 1995.

[6] D.B. Bartolini. *Adaptive process scheduling through applications performance monitoring.* PhD thesis, University of Illinois, 2011.

[7] S. Bell, B. Edwards, J. Amaann, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C. Miao, C.Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64^{TM} processor: A 64-core soc with mesh interconnect. In *International Solid-State Circuits Conference*, 2008.

[8] C. Bienia, S. Kumar, J.P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[9] V. G. Boltyanskii, R. V. Gamkredeledze, E. F. Mischenko, and L. S. Pontryagin. *Mathematical theory of optimal processes.* Interscience, New York, 1962.

[10] D.P. Bovet and M. Cesati. *Understanding the Linux Kernel.* O'Reilly, 2000.

[11] J. Boyan and A.W. Moore. Generalization in reinforcement learning: Safely approximating the value function. *Advances in neural information processing systems*, pages 369–376, 1995.

[12] J. Eastep, D. Wingate, and A. Agarwal. Smart data structures: A reinforcement learning approach to multicore data structures. In *ICAC 2011 Proceedings*, June 2011.

[13] J. Eastep, D. Wingate, M. D. Santambrogio, and A. Agarwal. Smartlocks: Lock acquisition scheduling for self-aware synchronization. In *ICAC 2010 Proceedings*, June 2010.

[14] CR Gallistel. Reinforcement learning rs sutton and ag barto. *Journal of Cognitive Neuroscience*, 11:126–129, 1999.

[15] Peter Gray. *Psychology.* Worth Publishers, New York, 6th e edition, 2010.

[16] P.H. Ha, M. Papatriantafilou, and P. Tsigas. Reactive spin-locks: A self-tuning approach. In *Parallel Architectures, Algorithms and Networks, 2005. ISPAN 2005. Proceedings. 8th International Symposium on*, pages 6–pp. IEEE, 2005.

[17] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal. Application heartbeats for software performance and health. Technical report, MIT, 2010.

[18] C.L. Hull. A behavior system; an introduction to behavior theory concerning the individual organism. 1952.

[19] E. Ipek, O. Mutlu, J. F. Martìnez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA 2008 Proceedings*, June 2008.

[20] A.H. Klopf. Brain function and adaptive systems: a heterostatic theory. Technical report, DTIC Document, 1972.

[21] A.H. Klopf. *The hedonistic neuron: A theory of memory, learning, and intelligence.* Hemisphere Publishing Corporation, 1982.

[22] A.H. Klopf. A drive-reinforcement model of single neuron function: An alternative to the hebbian neuronal model. In *AIP Conference Proceedings*, volume 151, page 265, 1986.

[23] A.H. Klopf. A neuronal model of classical conditioning. *Psychobiology*, 1988.

[24] E. Lau, J.E. Miller, I. Choi, D. Yeung, S. Amarasinghe, and A. Agarwal. Multicore performance optimization using partner cores. In *Proceedings of the 3rd USENIX conference on Hot topic in parallelism (HotPar'11). USENIX Association, Berkeley, CA, USA*, pages 11–11, 2011.

[25] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *HPCA*, pages 1–12, 2010.

[26] Gordon Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1998.

[27] L. Parsons, E. Haque, and H. Liu. Subspace clustering for high dimensional data: a review. *ACM SIGKDD Explorations Newsletter*, 6(1):90–105, 2004.

[28] I.P. Pavlov. *Conditioned reflexes: An investigation of the physiological activities of the cerebral cortex.* Oxford University Press, London, 1927.

[29] W. Schultz, P. Dayan, and P.R. Montague. A neural substrate of prediction and reward. *Science*, 275(5306):1593–1599, 1997.

[30] M.L. Scott and W.N. Scherer. Scalable queue-based spin locks with timeout. In *ACM SIGPLAN Notices*, volume 36 (7), pages 44–52. ACM, 2001.

[31] P. Shizgal. Neural basis of utility estimation. *Current opinion in neurobiology*, 7(2):198–208, 1997.

[32] F. Sironi, D.B. Bartolini, S. Campanoni, F. Cancare, H. Hoffmann, D. Sciuto, and M.D. Santambrogio. Metronome: operating system level performance management via self-adaptive computing. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 856–865. IEEE, 2012.

[33] F. Sironi and A. Cuoccio. Self-aware adaptation via implementation hot-swap for heterogeneous computing. 2011.

[34] F. Sironi, M. Triverio, H. Hoffmann, M. Maggio, and M.D. Santambrogio. Self-aware adaptation in fpga-based systems. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 187–192. IEEE, 2010.

[35] Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction.* MIT Press, 1998.

[36] R.S. Sutton, H.R Maei, D. Precup, S. Bhatnagar, D. Silver, C. Szepesvári, and E. Wiewiora. Fast gradient-descent methods for temporal-difference learning with linear function approximation. In *Proceedings of the 26th International Conference on Machine Learning*, Montreal,Canada, 2009.

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

[38] M.B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.W. Lee, W. Lee, et al. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *Micro, IEEE*, 22(2):25–35, 2002.

[39] M. Triverio, M. Maggio, H. Hoffmann, and M.D. Santambrogio. The consensus object: Coordinating the behavior of independent adaptive systems. 2011.

[40] A.R. Wagner, NE Spear, and RR Miller. Sop: A model of automatic memory processing in animal behavior. *Information processing in animals: Memory mechanisms*, 85:5–47, 1981.

[41] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, et al. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, 1997.

[42] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, 43(2):76–85, 2009.

[43] D. Wingate. Reinforcement learning in complex systems. Presentation for Angstrom Student Seminar, 2010.