UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

LEARNING ASSISTED DECOUPLED SOFTWARE PIPELINING (LA-DSWP)

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

LUCIA R. FITZMORRIS
Norman, Oklahoma
2018

LEARNING ASSISTED DECOUPLED SOFTWARE PIPELINING
(LA-DSWP)

A THESIS APPROVED FOR THE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

BY

Dr. Ronald D. Barnes

Dr. Jon G. Bredeson

Dr. Joseph P. Havlicek

Ut in omnibus glorificetur Deus

Acknowledgements

I would like to thank my committee chair, Dr. Ronald Barnes, for his support during the completion of this thesis. I cannot express how happy I have been to work in your lab and share in your passion for computer architecture. I would also like to thank the other members of my Master's thesis committee, Dr. Jon Bredeson and Dr. Joseph Havlicek, for providing the tools necessary for my success.

To my peers in the Soonergy Lab, especially Sonya Wolff, without you I never would have made it through this program. You have helped me grow as an engineer and as a person. I will value our friendship always.

To my parents, thank you for showering me with your love and for instilling in me a curiosity to explore the world. And to my little brother, you always can brighten my day with your uncanny sense of humor and your kind heart. I would also like to thank the sisters at Joseph Monastery. There is nothing like having 16 women ask you every week "Are you done yet?" to put the fire under you to finish a degree program.

Finally, some of the computing for this project was performed at the OU Supercomputing Center for Education & Research (OSCER) at the University of Oklahoma (OU). This material is based upon work supported by the National Science Foundation under Grant No. 1018771.

Table of Contents

List of Tables

List of Figures

Abstract

LEARNING ASSISTED DECOUPLED SOFTWARE PIPELINING (LA-DSWP)

Lucia R. Fitzmorris, M.S.
The University of Oklahoma, 2018

Supervisor: Dr. Ronald D. Barnes

In this thesis, I introduce and implement an extension to the Decoupled Software Pipelining (DSWP) algorithm proposed by Rangan et al. [34]. This new extension is named Learning Assisted Decoupled Software Pipelining (or LA-DSWP) as it applies reinforcement learning to the partitioning problem found within DSWP. Through experimentation, the viability of DSWP and LA-DSWP as optimizations that produce significant program speedup is tested and measured.

As computer architects strive to keep up with public expectations for processor performance growth, they are increasingly turning to processor designs which utilize multiple independent cores on a single chip. Unlike most prior hardware innovations, computer programs must be written or compiled with multiple threads in mind to take advantage of these new hardware innovations. Automatic thread-extraction using Decoupled Software Pipelining seeks to extract multiple threads from a single-threaded program [28]. This is done by allowing loops within the program to execute on multiple cores on a single processor chip simultaneously without programmer intervention. DSWP focuses on splitting large recursive data structure's traversal loops into multiple threads in an attempt to increase overall program performance.

Unlike prior implementations of DSWP, this research uses a hardware and language independent implementation of DSWP using the LLVM framework. Rather than relying on custom-built hardware to facilitate communication between program threads, this implementation uses Intel's Thread Building Blocks library to create queues in the shared memory between the various on-chip processor cores. As this thesis will show, this design setup relies heavily on the memory subsystem of the targeted processors and is greatly impacted by the actual design of the memory subsystem.

Another novel addition to DSWP explored in this thesis is the application of machine learning to the partitioning process. Instead of partitioning the nodes of a loop's program dependency graph using predefined heuristics, this thesis seeks to apply reinforcement learning to allow the DSWP agent to make more informed decisions when optimizing a given loop. The DSWP agent is able to collect and analyze data about each node of a program's loop to partition the loop on a node-by-node basis. This addition constitutes LA-DSWP.

Through experimentation on modern Intel processors, this thesis tests the feasibility of LA-DSWP on current hardware. Multiple kernel programs were written to search for program patterns that can achieve performance increases using DSWP partitioning. Experiments were run using the partitioning methods discussed in earlier papers along with the proposed method utilizing machine learning.

Chapter 1

Introduction and Motivation

Since the earliest days of computing, programmers have preferred writing computer code seqentially [32]. The process of creating a sequential list of instructions for the computer to follow is not only easily understood by a human programmer, but it also mimics the way older computer processors execute the program. In older processors, a program's instructions are fetched from memory and executed in the exact order given by the program [31]. To keep increasing processor performance, computer architects have changed this model of program execution within the processor in such a way that masks the pipeline's alterations from the programmer [13]. This model of "invisible hardware optimizations" has reached its end, and now computer architects rely on processor designs that require programmers to explicitly write code that will perform better on a given hardware configuration.

In 2004, Rangan et al. proposed a method of automatic thread extraction from single-threaded program loops named Decoupled Software Pipelining or DSWP [28,34]. This optimization was created to allow programmers to easily take advantage of new multi-core, single chip processor architectures without having to explicitly write programs using multiple, independent threads. This thesis proposes an extension to the DSWP process by optimizing thread extraction within DSWP using machine learning. This new extension of Rangan's optimization is named Learning Assisted Decoupled Pipelining or LA-DSWP. Through a hardware independent implementation of LA-DSWP, this thesis attempts to discover the expected performance increase for programs running on current hardware after they have been optimized using LA-DSWP.

1.1 Motivation

Before expounding on the definition of implementation of LA-DSWP, the remainder of this chapter is used to discuss the motivation for exploring DSWP based optimizations.

1.1.1 Moore's Law

The idea that a computer is outdated as soon as it has been purchased is a common sentiment expressed by everyone from computer scientists to casual web-surfers. This sentiment is ofter ascribed to Moore's law; however, Moore's law does not directly apply to performance increases. Instead, Moore's law refers to the fact that processor complexity will double every year [16,27]. While Moore's law has since been reduced to a doubling of processor complexity every 18 months, his original prediction from only five data points – shown in Figure 1.1 – was remarkably accurate. This increased processor complexity has translated into a massive performance increase in the years since his original prediction.

While there is some disagreement over how exactly to measure performance, a commonly accepted measurement method is by comparing the execution time of an agreed upon set of programs which mimic a normal user's workload [18]. Sets of programs designed to be representative of these workloads are used as benchmarks, with one of the most popular sets being the Standard Performance Evaluation Corporation (SPEC) benchmarks [31]. SPEC was created in 1989 by a conglomerate of vendors and is updated every few years to

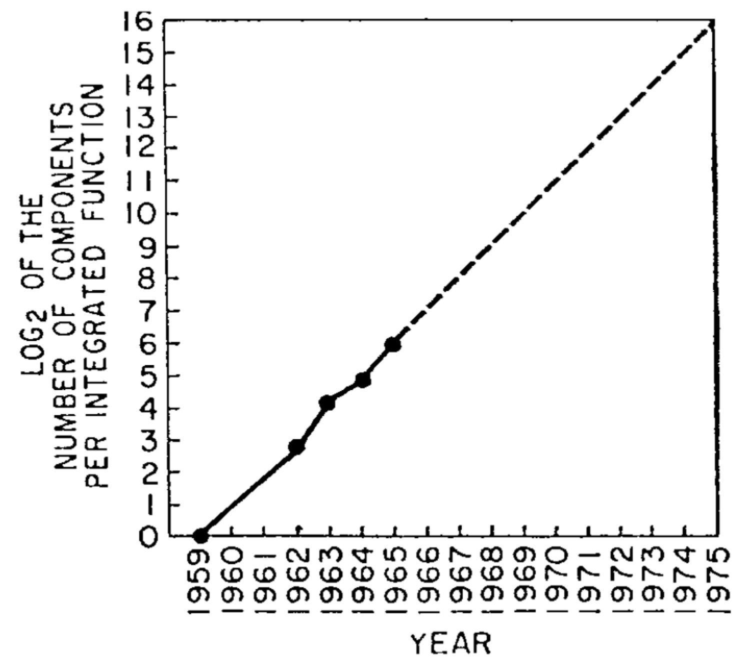


Figure 1.1: Reprint of Moore's Original Plot © 1998 IEEE [27]

include programs that represent the common classes of computing. As measured with a comparison against the Vax-11/780 using the SPEC benchmarks, processor performance increased by roughly 50% every year from 1986 to 2003 [17]. In contrast, Moore's law only calls for a processor complexity increase of 35% each year, showing that computer architects were able to increase processor performance at a higher rate than processor complexity. This rate of processor performance growth has slowed considerably since 2003, however, caused by what is known as "the power wall."

1.1.2 The Power Wall and Communication Latency

For a large portion of the history of processors, computer architects focused on designs and optimizations that allowed them to increase two primary aspects of their processors: frequency and IPC (Instructions Per Cycle) [2]. The speed at which a processor can execute instructions is primarily limited by these two aspects. Processor pipelining and memory cache structures were just a few of the major developments during the tail end of the 20th century that allowed computer architects to continue increasing processor performance by close to 50% per year [17]. These strategies came to an end in the early 2000's.

One major constraint that architects began to struggle with is what is now called the power wall. The very laws of physics that allow CMOS logic to function, also enforce an important relationship between power consumption and the frequency at which the MOSFET transistors switch. This relationship is expressed by:

$$P \propto \frac{1}{2} V_{DD}^2 f \tag{1.1}$$

where P is the power consumed by the processor, V_{DD} is the supply voltage, and f is the operating frequency of the processor [39]. In addition, energy consumption is directly proportional to the heat production of a processor.

Pushing frequency into the GHz, computer architects have bumped into the threshold where they are no longer able to feasibly cool their chips. Even with the steady decline of processor voltage from 5V in the earliest processors to almost 1V for today's processors, the power wall makes continuing to increase processor frequency at pre-2000 rates impossible [2, 13]. Compounding the effect of the power wall has been the move to mobile consumer products. Users demand smart devices and laptops that have long battery life and stay reasonably cool [31]. The power wall has, in effect, largely stalled the increase of processor frequencies.

Another effect compounding the power wall in the relationship that as frequency increases, the number of instructions executed every cycle tends to decrease [2]. Electricity can only travel at a finite speed, so as frequencies increase, data is unable to travel as far through the processor. So as computer architects have pushed frequency higher and higher, they must also work to not cause the IPC to drop and cause the overall performance increase to be negligible.

1.1.3 Multi-Core Processors

Another important rule that has held true in processor design has been Pollack's Rule [14]. Pollack's Rule states that if the complexity of a circuit is doubled, gain will be limited to at most a 50% improvement. This rate of increase corresponds to the 50% increase in processor performance discussed in early sections. After 2004, however, performance gains were impeded by the power wall. This decrease in performance gains forced the industry to change the way in which they designed processors; hence the age of the multi-core processor was born [17].

A multi-core processor is a integrated circuit that contains multiple smaller processors (or cores) that can execute completely independent threads. By including multiple cores on a single IC, Pollack's Rule should hypothetically be mitigated by allowing a performance speedup of 100% while only doubling the complexity of the chip [14]. Since roughly 2006, processor manufactures have embraced the CMP (Chip Multi-Core) revolution as a way to meet demand and continue to grow processor performance. Intel, for example, has pushed CMPs for personal computers that can have up to 6 cores in a single chip [17]. Processors used for servers can even have considerably more cores: for example, the Intel Phi processor has up to 72 cores [20].

1.1.4 Hardware Optimization Transparency

Until the rise of CMP's, the innovations in processor design that allowed steady performance increases were mostly invisible to typical computer programmers.

Most programmers work in high level languages (such as C++ or Python) and therefore are unaware of the many optimizations that are performed on their code by compilers and the hardware itself. In this manner, programmers are able to continue writing software the same way it has always been written, yet are able to reap the benefits of continued processor performance growth [13].

Take, for example, computer architects' innovations for branch prediction. Most processors since the mid 1980s have taken advantage of instruction pipelining. In a pipelined processor, many different instructions can execute inside a processor simultaneously, with each instruction being in a different "stage" of the processor [31]. Therefore, new instructions must be fetched from memory before all of the preceding instructions have been fully executed. This can cause major problems with branch instructions. Since a new instruction must be fetched from memory before a given branch has finished executing, the processor must predict the value to which the program counter should be changed to to continue fetching instructions for proper program execution. If a processor mis-predicts the program counter, for example guessing that a branch will update the PC to the next consecutive instruction in memory when it should have actually jumped to a new block of code, the processor pipeline must be flushed to remove the incorrect instructions from the data-path. Accurate branch predictions are important for processor performance since flushes can cause very long delays in program execution. Early methods of handling branch prediction such as static predictions (for example, always predicting that all branches will jump to a new block of code) were only about 62.5% accurate [40]. Later methods using small state-machines and look-up tables were able to increase correct prediction rates to 97% and now small neural networks, known as perceptrons, are used to reach prediction rates of close to

100% [23, 40]. All of these innovations have taken place to improve processor performance without programmers even needing to know that there has been a change in the hardware.

Creating hardware improvements that are invisible to the software does not extend to multi-core processors easily. Since the first assembly programs, software has almost always been written in a very sequential way: first the computer should do this, then add this, and finally store this [32]. High level languages like Fortran or C were able to abstract away the smaller steps and combine multiple machine level instructions into a single line of code, but the sequential manner of programming was unchanged. Even today's object oriented languages such as Python or C# rely on the fact that the order that instructions are performed in is important and should remain unchanged. These instructions may be scheduled by the hardware to execute in a different order by the processors using out-of-order execution, but they must always keep memory and registers consistent to match a processor working in-order. This once again keeps innovations in processor design invisible to the programmer [17].

Multi-core processors require programmers to break this sequential way of thinking. To achieve performance increases with these new processors, programers must split their programs into multiple threads that all execute at the same time. Communication between the threads is slow (limited by communication through usually the L2 or L3 cache) so programers must limit the communication between the threads along with limiting the amount of shared memory between the two [6, 22]. Enforcing memory consistency between two processors is slow so data shared between the two processors can cause many issues for the processor's cache and memory buses [13]. Most programming languages have almost no native support for the structures and processes needed to handle this new parallel way of programming, which in turn, causes programmers to be slow to adapt their code to match multi-core hardware. Institutions using legacy code that in the past have been able to benefit from “Moore’s Bounty” (the increase in performance with no change to software) are no longer able to benefit from these new improvements in performance [24].

Chapter 2

Related Work

Many new methods have been introduced to try to automatically harness parallelism in loops to allow them to be run across many different processor cores. Some new methods have been harnessed to give huge speedups for scientific computing, but those methods usually carry strict requirements to ensure proper code execution. In this section, we will focus on loop parallelization methods.

2.1 DOALL

The easiest method to understand is DOALL parallelization. A loop that is DOALL parallel is one that has no dependencies from one iteration of the loop to the next [30]. This can be a very strict definition for a loop to meet, but if a loop meets this condition every iteration of the loop can be done in parallel. This obviously has potential to give a huge gain in performance if there are many cores to execute the different iterations such as in a GPU. An example of how a loop could be broken with DOALL parallelization to be run across 4 different cores can be seen in Figure 2.1. Notice that since there are no dependencies between loop iteration, no communication is needed between the threads which means DOALL loops performance increases are independent of communication latency between threads. Hypothetically the parallelization shown in the figure could give nearly a 4X speedup compared to the non-parallelized code.

Core 1:
Iteration 1
Iteration 2
Iteration 3
...
Iteration N

(a) Before Parallelization

Core 1:	**Core 2:**	**Core 3:**	**Core 4:**
Iteration 1	Iteration 2	Iteration 3	Iteration 4
Iteration 5	Iteration 6	Iteration 7	Iteration 8
Iteration 9	Iteration 9	Iteration 10	Iteration 11
...	...	...	...
Iteration N-3	Iteration N-2	Iteration N-1	Iteration N

(b) After Parallelization

Figure 2.1: Program Execution Before and After DOALL Parallelization

2.2 DOACROSS

The next type of parallelization is DOACROSS. Just like in DOALL, DOACROSS splits iterations of the loop between multiple cores. Each iteration is completely executed by a single core. This is shown in Figure 2.2 which illustrates a DOACROSS parallelization of a loop that has 3 instructions per iteration where each iteration's instructions are given a unique shape. DOACROSS differs from DOALL in that data dependencies across iterations are allowed [30]. These are shown by the edges in Figure 2.2 pointing from one core to another. Figure 2.2 assumes a latency of 1 cycle to communicate from one core to another. This figure also shows that DOACROSS can be very sensitive to intra-core communication latency. A latency of 3 would cause this parallelization to have no performance increase because each core would be executing while the other was waiting for the value it depended on [29].

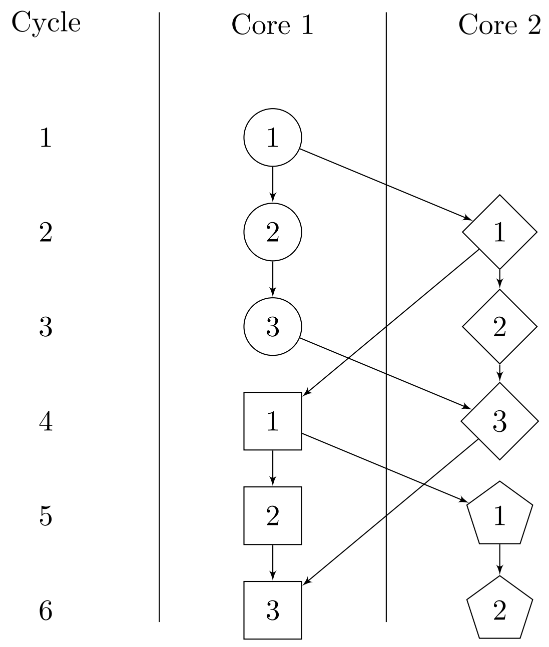


Figure 2.2: Program Execution After DOACROSS

2.3 DOPIPE

DOPIPE is very different from the two previous examples of code parallelization. In both DOALL an DOACROSS each iteration of the loop is executed completely on a single core. DOPIPE instead breaks a single iteration of a loop into multiple steps and executes each step on a different core [29]. Figure 2.3 shows an example of the same program as in Figure 2.2, but parallelized using DOPIPE. Notice that the first two instructions of every iteration occur in core 1 and the last instruction is executed in core 3. This can allow for far less communication between the two cores if the code is broken in the correct places. The primary constraint that makes DOPIPE unfeasible for most loops is that it does not work if there are any extra control dependencies inside of the loop. So for example, if the loop contains an if statement or another loop, it is no longer a candidate for DOPIPE [30].

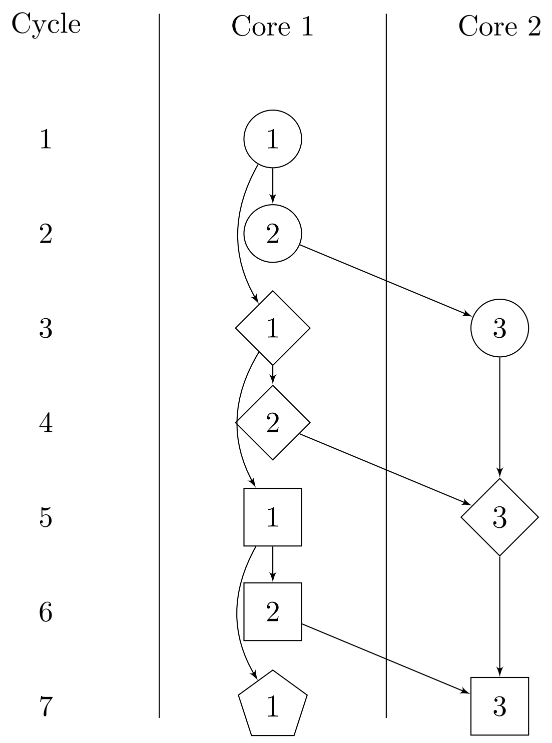


Figure 2.3: Program Execution After DOPIPE Parallelization

2.4 DSWP

Decoupled Software Pipelining (DSWP) was created to relax the control dependency restraints of DOPIPE. DSWP was originally formulated at Princeton University in 2004 [34]. The researchers noticed that many programs have loops that traverse very large recursive data structures (RDS). In a typical loop there are two main paths of instructions that are executed: traversal and computational. The traversal path is the one that traverses through memory to load each node of the RDS. The second path does the actual computations that are returned at the end of the traversal. So for instance, if a program is looking for the average value of a linked list, the traversal path loads the pointers that allow the program to move from each index in the list to the next. The computational path is the instructions that are using the values at each index to find the overall average.

DSWP seeks to split RDS loops, and other loops like them, into two separate threads that are mostly independent from each other. Not only does DSWP allow speedups by allowing multiple instructions to execute in parallel, it can also produce better behavior from the memory system of the processor. RDS's can result in poor cache performance because they usually have poor locality within memory. Repetitive cache misses degrade performance and force processors to stall for long periods of time. Ideally, the processor would fetch as many instructions along the traversal path and begin loading data from memory so that some of the latency caused by cache misses would be hidden by the fact that many of the load instructions read times from memory would overlap. This, unfortunately, is not how current processors operate; even out-of-order processors fetch and issues instructions in program order and so the loads of the traversal path have to wait on the instructions of the computational path to be fetched. DSWP helps alleviate this problem by having all traversal instructions on one core and computational instructions on another. This allows the processor to queue many of the traversal load instructions together which can greatly help the memory behavior of the processor [34].

Like DOACROSS and DOPIPE, DSWP relies on low-latency communication between multiple cores to pass data and control dependencies between the two loop threads. The original DSWP work relied heavily on what was termed the "synchronization array" to achieve this low latency communication between cores [34]. The synchronization array is a set of hardware queues that enable separate cores to both pop and push register values with a latency of 1 cycle between the two cores. The synchronization array works well in simulation and helps minimize overhead, but currently only exist on simulated hardware and therefore unable to be used for anything outside of theoretical proof of concepts. Another constraint of the original DSWP algorithms is that their automatic partitioning of loops into threads was done with a straight cut using heuristics they found experimentally. Their heuristics were usually able to find the best split for their threads, but occasionally would perform miserably, even slowing the loop, while the best hand-tuned split was able to give a respectable increase [28].

DSWP was later extended to use other methods to allow it to scale more reasonably with the number of cores available. The first of these methods was SpecDSWP [37]. Speculation has been used for both DOALL and DOACROSS loops in the past and helps relax some of the constraints on a given loop [30]. Speculation allows the processor to guess if a certain condition is expected to be a single value and so lets the compiler ignore a dependency. For instance, a loop could have only one control dependency created by an error-detection if-statement. This control dependency would need to be passed from one thread to another and could create more overhead for a DSWP partitioning. The compiler could instead choose to speculate that this value will always be false (assuming the programmer did not make many mistakes) and so this communication could be ignored and a queue from core 1 to 2 would no longer be needed. Extra mechanisms are needed to handle instances where the processor mis-speculates a given edge [37].

Another extension of DSWP combines both DOALL and DSWP into a single method called PS-DSWP or Parallel-Stage Decoupled Software Pipelining [33]. PS-DSWP splits loops into two separate threads of execution, but it then applies DOALL to the newly created threads. This allows DSWP to be easily scalable and to take advantage of large numbers of cores. PS-DSWP has been shown to give a much better speedup then DSWP alone, but still uses heuristics to find its partitions and has only ever been implemented on simulated hardware.

Chapter 3

Modern Processor Design

Modern processor design differs greatly from the simple architectures in undergraduate textbooks. As discussed in Chapter 1, modern processors use a pipeline model allowing multiple instructions to execute at once inside a processor. This pipeline organization allows the high clock frequencies seen in today's processors by allowing the work of a single instruction to be done in multiple short clock cycles, rather than one long clock cycle. Processor pipelines, currently, have also been harnessed to allow instructions to complete out of program order, but still in a valid sequence. These features, along with a complex memory system, have allowed modern processors to keep up with the growing pressure to increase hardware performance.

3.1 Out-of-Order Pipeline

In a a typical in-order processor, instructions are fetched from memory in program order, executed in program order, and then retired in program order. This pattern of execution is easiest to understand and design and was the typical design of early processors. This method, however, is not the most efficient strategy for a processor to follow. For example, a program might contain a long latency instruction followed by a set of short latency instructions that do not depend on the long latency instruction. In an in-order processor, the processor waits until the long latency instruction has finished executing before it starts to execute subsequent instructions. An out-of-order processor could instead execute

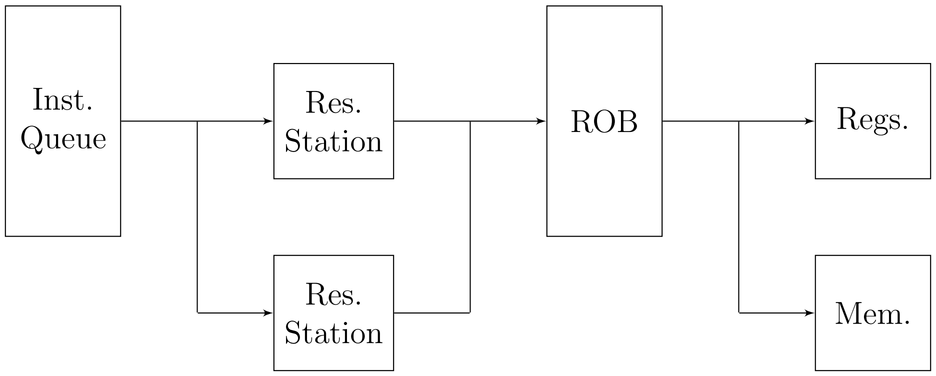


Figure 3.1: Simplified Out-Of-Order Pipeline

the subsequent instructions while it is waiting on the long latency instruction to finish execution. In doing this, out-of-order processors are better able to harness instruction-level-parallelism by executing instructions as soon as they have all of the data they need to execute, even if a preceding instruction has not yet been executed. In executing instructions as soon as their data is ready, the processor has more opportunities to allow multiple instructions to execute at once.

Figure 3.1 illustrates a simplified out-of-order pipeline. In an out-of-order pipeline, instructions are fetched from memory in program order and stored in the instruction queue, which is a FIFO (first-in-first-out) buffer. Instructions are then issued from the instruction queue to different reservation stations depending on what sub-system of the processor they need for execution. For instance, memory instructions may be stored in one reservation station, while instructions using the integer ALU may be sent to another. During a clock cycle, each reservation station checks to see if it contains instructions that have all the data needed for execution, and if so, it dispatches the instruction to be executed. This allows multiple instructions to execute at once, as long as the processor subsystems are available. After an instruction finishes executing, it is sent to be held in the reorder buffer (ROB). The reorder buffer is a queue used to keep the state of memory and registers consistent with a processor that is running in-order. To do this, the ROB only allows instructions to retire (make changes to memory and the registers) in program order, meaning it stores instructions in a queue and only allows them to pop from the front if all preceding instructions in program order have already left the ROB [17].

Since all instructions must move through the ROB before they are retired, it can become a bottleneck in the pipeline. Take for example the Intel Haswell architecture whose ROB is 192 instructions long and can have 72 load instructions executing at once [22]. As will be discussed later, a load instruction requiring main memory can take roughly 200 cycles to execute and will stay in the ROB that entire time. A program could likely exist that has two load instructions that require main memory spaced 200 instructions apart. If both instructions could be executed at the same time, some of the latency of the second load could be masked because it would overlap with the execution time of the first load instruction. This, however, will never occur because the ROB will fill up with the first load instruction and the 191 subsequent instructions. This will cause the second load not to be fetched until the first load is retired from the ROB. DSWP tries to alleviate this problem by moving all load instructions into a single core, pushing as many of the other instructions as possible into a second core. This allows the ROB of the core with mostly loads to not be diluted with non-memory instructions. Ideally this will allow many of the load instructions' executing times to overlap, thus increasing overall performance.

3.2 Memory System

All modern processors rely heavily on their memory hierarchy to perform well. Accessing main memory implemented as Random Access Memory (RAM) can require hundreds of CPU cycles. If processors dispatched load instructions to main memory one at a time and waited for them to be fulfilled before moving on, the processors would slow to a crawl compared to modern processor architectures. The primary way they are able to mask this latency is by using a cache hierarchy.

A cache is a small, very fast memory structure that exists on-chip with the main processor. Caches can be organized many ways, but the goal is to keep them full of the data the program will most likely use next. When a program loads data from memory, it first checks the L1 cache and if the data is found (termed a cache hit), it is given to the processor without having to contact main memory. If the data is not found (a cache miss) the next level of cache is queried. This continues until either the data is found in a cache or, as a last resort, main memory. The typical latency today for a L1 cache hit is on the scale of 5 cycles and an L2 cache hit being about double that [17]. This shrinks the latency of a memory load from the point of view of the processor to tens of cycles instead of hundreds allowing it to have a much improved overall performance. The trick, however, is how to choose what information should be stored in the caches.

Software usually accesses memory in a combination of two different patterns which form the idea of data locality. The first type of locality is spatial locality, which is the idea that if data is loaded from a given memory location, data from nearby will most likely be loaded next. An example of this fact is iterating through an array. Caches are structured to take advantage of this idea

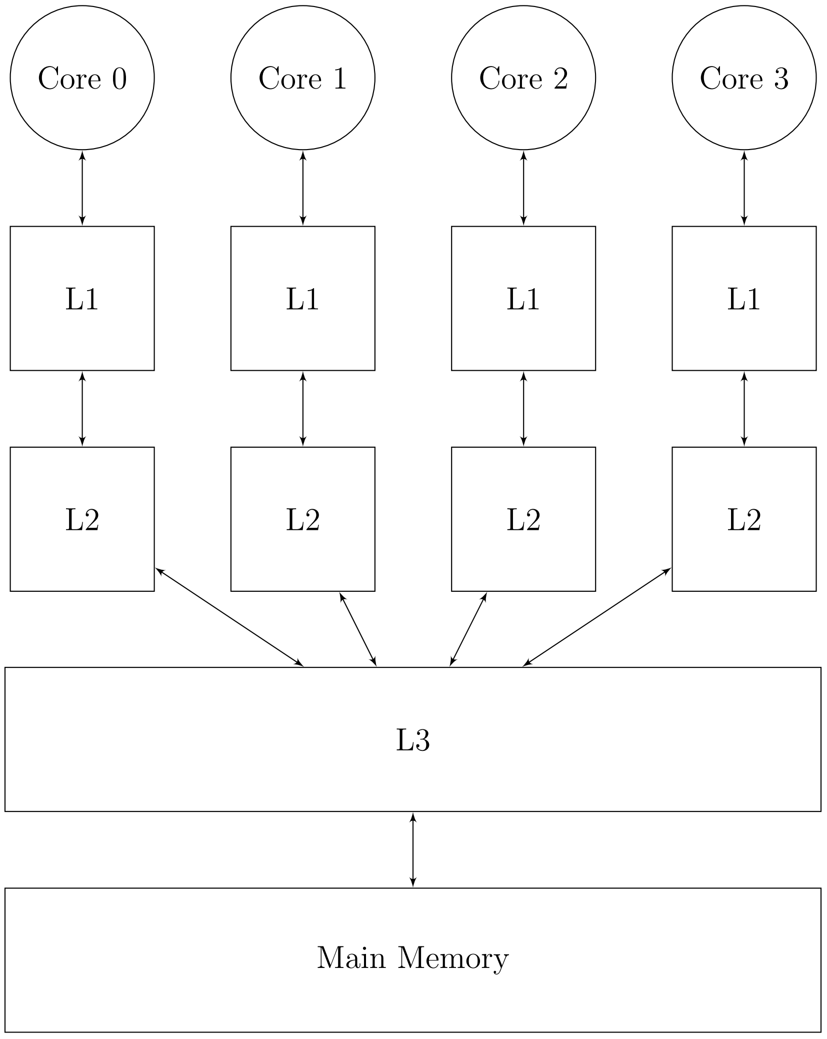


Figure 3.2: Modern Processor Memory Hierarchy

by loading data in blocks. Instead of only loading the data that is needed by the processor, a cache will load and store an entire block of data [31]. The processor used in this research, for example, always loads data into its caches as 64 byte blocks. Using this method, the data that is needed by the processor is loaded into the cache along with the bytes around it so they will already be in the cache if they are needed later [22]. The second type of locality is temporal locality, which is the idea that data that has been recently used will most likely be used again. Caches take advantage of this principle by evicting data from the cache that has been used the least recently.

3.3 Inter-Core Communication

In the original formulation of DSWP, a Synchronization Array (SA) was used to facilitate inter-core communication [34]. Instead of requiring the creation of a new piece of hardware, my work focuses on implementing DSWP on pre-existing hardware. Intel and AMD, the two largest manufactures of processors, currently use two different approaches to allow inter-core communication, but both do agree on using a shared L3 cache as is illustrated in Figure 3.2.

Intel places the L3 cache in the Uncore, the structures of the processor that don't belong to a single core [22]. In the Uncore, the L3 cache is composed of cache slices, with each slice being tied to a core. The slices are arranged into a ring of at most 8 slices that are overseen by an agent. Each slice is fully inclusive of the data held in its processor's higher level caches and uses a write-back policy. Unlike a typical cache, Intel describes the shared cache as having a clean hit and a dirty hit. A clean hit is one in which the data needed is contained within the querying core's cache slice. A dirty hit is one in which the data is found in another core's L3 cache and so the agent must facilitate a transfer of data from the supplying core to the querying core. Because of the ring configuration, depending on the location of the slice providing the data, the latency of the cache hit can be variable [22].

AMD took a slightly different approach with their Ryzen processors. Like Intel, AMD uses cache slices that are tied to a given core. Instead of using a ring of slices, however, these slices are fully connected in sets of 4 slices [6]. Also, instead of being fully inclusive, the L3 slices are victim caches. This configuration of the L3 cache allows AMD to have the same latency to access another core's L3 cache slice as a core's own slice [3].

In both cases, inter-core communication is tied directly to the memory subsystem of each processor. Two cores will never be able to communicate with a latency of less than a L3 dirty hit. This latency is a significant hurdle for DSWP to overcome and is one of the main reasons the original research relied instead on using the Synchronization Array [34].

Chapter 4

Transform Implementation

This implementation of the DSWP follows the algorithm described by Ottoni and others [28]. This algorithm can be summarized as follows:

1. Build Program Dependency Graph (PDG)
2. Contract PDG to Form Directed Acyclic Graph
3. Determine Partitions Using Machine Learning
4. Create Partitions in Software

Steps 1 and 2 are largely unchanged from the process original DSWP given by Ottoni and are implemented with a strong reliance on built-in LLVM transform passes, as discussed in the next section. Machine learning has been added to Step 3 to try to achieve a larger speedup across all benchmarks. Finally, Step 4 has been updated to partition code to run on commonly found hardware instead of relying on hardware with the custom designed Synchronization Array to allow thread communication [34].

4.1 Software Details

The DSWP transform was completed implemented in C and C++ along with a few bash scripts to facilitate machine learning training. The majority of the code was created to interface with the LLVM framework. LLVM is a compiler created at the University of Illinois at Urbana-Champaign [25] . Since its first release in 2003, LLVM has expanded in use and is currently the default compiler for all OS X machines. It was created to help facilitate the analysis and transformation of code by using an internal code representation (IR) that reads much like a RISC assembly language, but is hardware independent [1]. This IR is able to capture much of the original high level information that is useful for optimizations during the entire lifetime of the software. The IR is generated by the LLVM front-end, transformations are then run on the IR, and finally the IR is passed onto the LLVM linker and back-end to be turned into machine specific executables. This modular framework facilitates researchers to create generic transformations that operate on the LLVM IR without having to focus on language or machine specific limitations. The following figures in this section are given as LLVM IR.

The DSWP transformations used in this paper were all written as LLVM Loop Transformation Passes so that they could use all of the analysis capabilities included in LLVM. This includes using LLVMs analysis passes such as natural loop discovery, memory analysis, and post-dominator tree creation.

The other library used by this project is Intel's Thread Building Block library [19]. This library was created by Intel to facilitate parallel programming in multiple ways, but this project uses its thread safe queues to form the synchronization arrays outlined for the DSWP algorithm. These queues are non-blocking, fine-grained locking queues that can be safely accessed by multiple threads at once.

4.2 Program Dependency Graph Building

After a candidate loop has been found by LLVM, the first step of the DSWP algorithm is to construct its program dependency graph or PDG. The PDG

```
%x = load  %x.addr
%y = load %y.addr
%sum = add  %x,  %y
```

Figure 4.1: Example Code Snippet for Data Dependence

is constructed to enumerate all data, memory, and control dependencies in a given loop [10]. Within the PDG each instruction composing the target loop is a node within the graph, and all dependencies are directed edges between the instruction nodes.

4.2.1 Data Dependencies

The first of these dependencies, data, is almost trivial to enumerate in LLVM IR because it is in Static Single Assignment (SSA) form [8]. Data dependencies are caused when one instruction uses the value created by another instruction. For example, in the code snippet in Figure 4.1, there is a data dependence between the first and third instructions because the third instruction uses the value created in the first instruction. In a SSA form, each variable is only assigned a single time. So for instance in the code snippet shown in Figure 4.1, the value %y will only be assigned by the above load instruction in the given scope. Because of this, %y can be thought of as a label for the load instruction itself. This relationship is expressed in LLVM IR by the fact that instruction operands can be constants or other instructions. So for the example code snippet, %x is not just a value in LLVM IR, it is equivalent to the load instruction itself. This makes data dependencies very easy to find; the DSWP transform can simply inspect the operands of a given instruction and add dependencies between it and any of its operands that are also instructions.

```
%x = load %x.addr
%x2 = load %x.addr
%y = load %y.addr
store %s, %y.addr
```

Figure 4.2: Example Code Snippet for Memory Dependence

4.2.2 Memory Dependencies

The next kind of dependency, memory, is far more complex to find. Memory analysis is a complex subject and is outside the scope of this research, so the built-in LLVM memory analysis passes were used to facilitate this step of PDG building. This project used all of the memory analysis passes that are built into the LLVM framework. The data from these passes is then used to make alias sets. Each alias set is comprised of memory pointers that could possibly alias (refer to the same location in memory). If an alias set only contained pointers that were used to read from memory, then the set was discarded with no change to the PDG. This was done because the set only contained RAR (Read After Read) dependencies which are not true dependencies and so can be reordered as needed.

An example of this is shown in Figure 4.2 between the first two instructions. Both instructions load from the same location, but they can occur in either order without affecting program execution. If an alias set contained pointers that were given to functions that wrote memory or were used in a store instruction, the set was used to create memory dependencies between the instructions in the PDG. An example of this would be the last two instructions shown in Figure 4.2. If the store instruction was to happen before the last load instruction, it would most likely change the outcome of the program and so is marked as a dependency in the PDG.

```
IF_COND:
br %x, 0, %IF_BODY, %IF_END

IF_BODY:
store %x, %x.addr
br %IF_END

IF_END:
%y = load %y.addr
```

Figure 4.3: Example Code Snippet for Control Dependence

4.2.3 Control Dependencies

The final type of dependency that must be added to the PDG is control dependencies. These occur when an instruction can control whether or not another instruction will execute. An example of a control dependence is shown in Figure 4.3. The first "br" instruction can control whether or not the value of %x will be stored into memory, therefore this dependency needs to be added to the PDG of the example program. These control dependencies can most easily be found using a post-dominator tree (PDT). This project used LLVM to build the PDT which uses the algorithm outlined by Lenguer and Tarjan [26]. The PDT is built with the basic blocks (BB) of the loop as its nodes and the edges showing the post-dominance relationship between the different BB's. A node, D, is post-dominated by another node, E, if the program cannot reach the end of the loop starting at D without traveling through E [10]. For example, in Figure 4.3, the program cannot reach IF_END without going through the IF_COND block, therefore IF_END is post-dominated by IF_COND.

Once the PDT has been found, it can be used to find control dependencies between BB's. A control dependency from basic block D to E iff D is not post-dominated by E and there is a path from D to E in which all BB's (that aren't

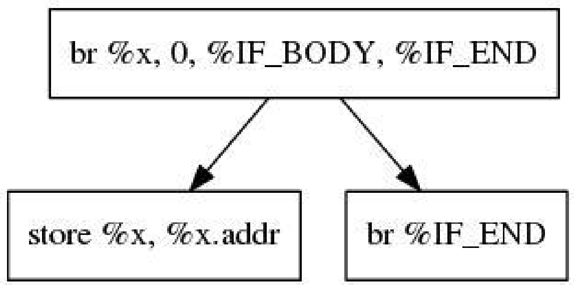


Figure 4.4: Example Control Dependence within PDG

D or E) are post-dominated by E. This relationship can be seen in Figure 4.3; IF_COND is not post-dominated by IF_BODY, and since there are not other nodes in the path between the two BB's the second condition is also met.

These control dependencies are shown within the PDG by adding an edge from each instruction in a basic block to the terminator of the controlling block. For example, for Figure 4.3, an edge would be added between the *br %x, 0, %IF_BODY, %IF_END* instruction and both instructions in the IF_BODY basic block as shown in Figure 4.4.

4.3 Directed Acyclic Graph Building

A requirement for DSWP is that all data flows must be unidirectional: data cannot flow in and out from the same thread in a valid partitioning [34]. A normal program dependency graph will have many cycles which are usually formed by program loops. To facilitate DSWP partitioning the PDG is condensed by contracting all instructions that are strongly connected components (SCC) into a single node. The search for SCC's in the PDG is done using LLVM's built in scc_iterator which uses Tarjan's strongly connected components algorithm [36].

The contraction of the SCC will condense the PDG into a directed acyclic graph (DAG) that can be used for DSWP partitioning. The DAG will not

```c
while (next != 0x00)
{
    int data = next->data;
    data = doWork(data);
    next->data = data;
    next = next->next;
}
```

Figure 4.5: Example Code Snippet for PDG and DAG

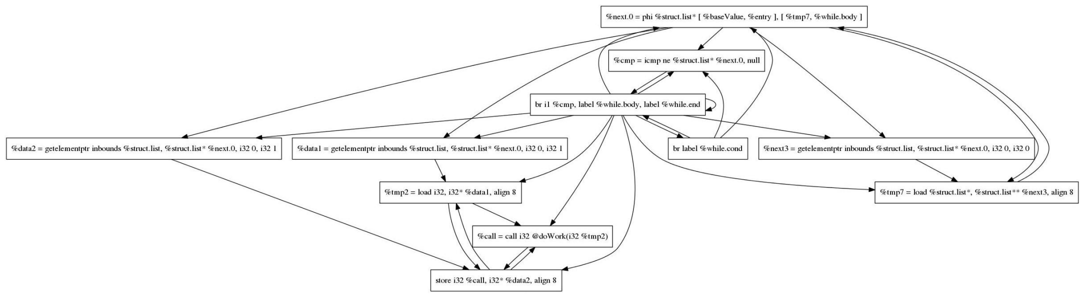
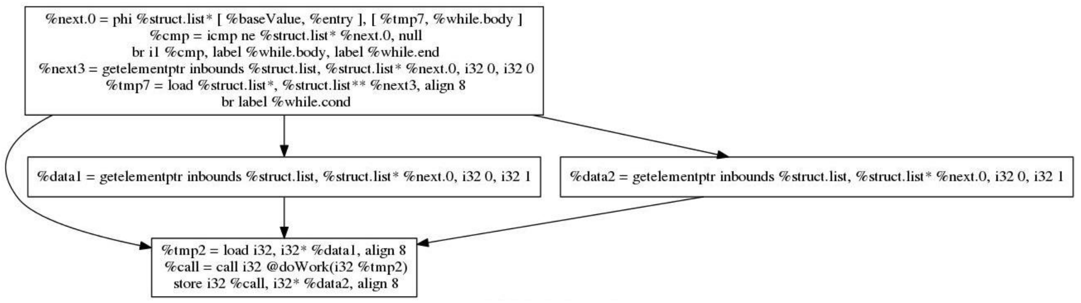

%next.0 = phi %struct.list* [%baseValue, %entry], [%tmp7, %while.body]
%cmp = icmp ne %struct.list* %next.0, null
br i1 %cmp, label %while.body, label %while.end
%data2 = getelementptr inbounds %struct.list, %struct.list* %next.0, i32 0, i32 1
%data1 = getelementptr inbounds %struct.list, %struct.list* %next.0, i32 0, i32 1
br label %while.cond
%next3 = getelementptr inbounds %struct.list, %struct.list* %next.0, i32 0, i32 0
%tmp2 = load i32, i32* %data1, align 8
%tmp7 = load %struct.list*, %struct.list** %next3, align 8
%call = call i32 @doWork(i32 %tmp2)
store i32 %call, i32* %data2, align 8

Figure 4.6: Example PDG for Code Snippet

contain any cycles that can cause problems during the later DSWP partitioning. An example of this contraction can be seen in Figure 4.7; it is the contraction of the PDG in Figure 4.6 which was generated from the code in Figure 4.5. All instructions in a strongly connected component will be partitioned into the same thread. This is similar to the partitioning of SCCs to cores used in a hardware approach to pipelining, Two Pass Pipelining [4, 5].

4.4 Determine Partitions Using Machine Learning

Previously, DSWP research has used different heuristics to choose which nodes of a loop's DAG should be assigned to each DSWP partition [28] [33] [37]. Usually these heuristics were able to create partitions that gave a decent speedup and were able to match the best partition found iteratively. Sometimes, however, the heuristics would perform very poorly on a few loops and would slow down the code greatly. This project sought to use machine learning (ML) to better handle



```
%next.0 = phi %struct.list* [ %baseValue, %entry ], [ %tmp7, %while.body ]
%cmp = icmp ne %struct.list* %next.0, null
br i1 %cmp, label %while.body, label %while.end
%next3 = getelementptr inbounds %struct.list, %struct.list* %next.0, i32 0, i32 0
%tmp7 = load %struct.list*, %struct.list** %next3, align 8
br label %while.cond
```

```
%data1 = getelementptr inbounds %struct.list, %struct.list* %next.0, i32 0, i32 1
```

```
%data2 = getelementptr inbounds %struct.list, %struct.list* %next.0, i32 0, i32 1
```

```
%tmp2 = load i32, i32* %data1, align 8
%call = call i32 @doWork(i32 %tmp2)
store i32 %call, i32* %data2, align 8
```

Figure 4.7: Example DAG Condensation of PDG

different types of loops and so hopefully perform well on a broader spectrum of loops.

4.4.1 Reinforcement Learning

Reinforcement Learning (RL) provides a system to define a problem that needs to be solved in some statistical manner. Unlike many forms of machine learning, reinforcement learning is not a supervised learning method: the exact correct solution for the problem in not known [15]. RL is instead used to solve problems where only the "goodness" of a solution is measurable. In RL an agent makes decisions in its environment to try to solve a problem. To help define the problem, an RL solution can be broken into 5 basic subelements:

- States : S
  - All configurations the environment could possible be in.
- Actions : A
  - A set of all actions that the agent can take.
- Policy : $\pi : S \rightarrow P(A = a|S)$
  - A mapping from states to actions that describes the probability that

$$\begin{bmatrix} Q(s_0,a_0) & Q(s_0,a_1) & \dots & Q(s_0,a_n) \\ Q(s_1,a_1) & Q(s_1,a_1) & \dots & Q(s_1,a_n) \\ \vdots & \vdots & \ddots & \vdots \\ Q(s_m,a_0) & Q(s_m,a_1) & \dots & Q(s_m,s_n) \end{bmatrix}$$

(a) Generic Q-Table

$$\begin{bmatrix} 1 & 3 & 6 & 8 \\ 4 & 2 & 5 & 1 \\ 1 & 6 & 8 & 2 \\ 4 & -3 & 10 & 5 \end{bmatrix}$$

(b) Q-Table Before

$$\begin{bmatrix} 1 & 3 & 5.88 & 8 \\ 4 & 2 & 5 & .76 \\ 1 & 6 & 8 & 2 \\ 4 & -3 & 10 & 5 \end{bmatrix}$$

(c) Q-Table After

Figure 4.8: Example Q-Tables for SARSA

the agent will take an action given the state the environment is currently in.

- Reward Function: $R : (s, a) \to \mathbb{R}$
  - A mapping from state-action pairs (the action taken by the agent from a given state) to a numeric value that measures the *short-term* "goodness" of an action.
- Value Function: $Q : (s, a) \to \mathbb{R}$
  - A mapping from state-action pairs to a numeric value that measures the *long-term* "goodness" of an action.

Notice that the subelements do not describe a solution to a given problem, but merely act as a framework that can be used to describe the problem [35].

4.4.2 Learning Method and Policy

The solution chosen for this research to solve the DSWP partitioning problem is State-Action-Reward-State-Action (SARSA) with an epsilon-greedy policy. SARSA seeks to create and update a table of Q-values until it converges on the true Q values for a given set of states and actions. After the table converges the learner can then exploit the learned values to take actions that will give it the most long term rewards [38]. A generic Q-table for a state space with states s_0 to s_m and an action space with actions a_0 to a_n is shown in Figure 4.8a. This table is initialized to some set of values and then needs to be updated to reach the true Q values for the current policy. This update is done iteratively by the learner as it explores its environment. For each time-step, t, the agent takes some action a_t from its current state s_t and will receive some reward from the environment r_{t+1}. This then moves the agent into a new state s_{t+1} where it will again take a new action a_{t+1} and receive another reward r_{t+2}. Each time the agent takes an action it will update its Q-table using the following formula:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha * [r_{t+1} + \gamma * Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{4.1}$$

In the formula, α is the learning rate (which is usually very small, but always between 0 and 1) and γ is the discount factor (which is also always between 0 and 1) [35]. A large learning rate can speed up the learning process but can also make the agent very sensitive to noise. A large gamma favors long term rewards over short term rewards.

An example of an agent moving through states by taking different actions can be seen in Formula 4.2.

$$\mathbf{s_0} \rightarrow_{a_2} \mathbf{s_1} \rightarrow_{a_3} \mathbf{s_3} \rightarrow_{a_1} \mathbf{s_0} \tag{4.2}$$

An example of how the Q-Table for this agent would be before and after the three actions is shown in Tables 4.8b and 4.8c, respectively. After the first action, a_2 in this case, the agent moves from state s_0 to s_2 and is given some reward; for this example it will get a reward of 4. For formula 4.1 these values can be filled in for s_t, s_{t+1}, a_t, and r_{t+1} as follows:

$$Q(0,2) \leftarrow Q(0,2) + \alpha * [4 + \gamma * Q(s_1, a_{t+1}) - Q(0,2)] \tag{4.3}$$

Next the agent takes its next action to go from s_1 to s_3 using action a_3 and gets a reward of 1. This allows the values of a_{t+1} to be added. Also for this problem α will be set to 0.1, and γ will be set to 0.8. Now the formula is the following:

$$Q(0,2) \leftarrow Q(0,2) + 0.1 * [4 + 0.8 * Q(1,3) - Q(0,2)] \tag{4.4}$$

Next the original Q-Table (Figure 4.8b) is used to fill in values for the $Q(0,2)$ and $Q(1,3)$ arguments.

$$Q(0,2) \leftarrow 6 + 0.1 * [4 + 0.8 * 1 - 6] \tag{4.5}$$

Solving we find the new value of Q(0,2) is 5.88. This is done once again to update the value at Q(1,3) and the final Q-table is shown in Figure 4.8c.

The policy used by the agent to choose which actions to take from a given state is the epsilon-greedy policy. In this policy the agent chooses to take a random action with probability ϵ and chooses the to take the action with the highest Q value with probability $1 - \epsilon$. Epsilon-greedy tries to balance both exploration and exploitation of a given environment to achieve the highest rewards over time. An ϵ close to 1 will create a learner whose actions are almost completely randomly and so will explore most of the state space. An agent with an ϵ close to 0 will always try to exploit its knowledge by choosing the action with the Q value but may miss actions that have much better reward by never exploring them and so never updating its Q-value to acknowledge the higher rewards.

Table 4.1: State and Action Definitions for DSWP ML

State Features					
Cycle Ratio	Latency	Flows	Slack	Percent	Actions
$[0.00, 0.20)$	$[0, 10]$	$[0, 0]$	$[0, 50]$	$[0, 10]$	ABOVE
$[0.20, 0.30)$	$(10, 30]$	$[1, 1]$	$(50, \infty)$	$(10, 20]$	BELOW
$[0.30, 0.40)$	$(30, 80]$	$(1, 3]$		$(20, 30]$	
$[0.40, 0.45)$	$(80, \infty)$	$(3, 5]$		$(30, 40]$	
$[0.45, 0.48)$		$(5, \infty)$		$(40, 50]$	
$[0.48, 0.52)$				$(50, 70]$	
$[0.52, 0.55)$				$(70, 100]$	
$[0.55, 0.60)$					
$[0.60, 0.70)$					
$[0.70, 0.80)$					
$[0.80, 1.00]$					

4.4.3 Problem Space

The states used to describe the environment relate to each node in the DAG. Each node is given its own state using different features that relate to different characteristics of the node. A summary of the features that make up the state of each node can be seen in Table 4.1. To determine most of the features for the nodes, the instructions of the loop are first scheduled. Since LLVM IR does not directly relate to hardware, latencies were generalized to to determine the latency for IR instructions [11]. When scheduling the instruction, an infinite-width processor was assumed and instructions were scheduled within their respective BBs. The instructions are scheduled using both bottom-up and top-down methods. The bottom-up schedule is used to determine overall estimated execution time of the loop iteration along with where each instructions falls in that schedule. The top-down schedule is used to calculate each nodes slack value, as described in the next paragraph.

After scheduling all instructions, each node is given a cycle level by finding the instruction in the node with the max cycle assignment using the bottom-up scheduling method which is then assigned to the DAG node. This cycle level is used to find the **Cycle Ratio** of the node which is simply the node's cycle level divided by the total number of cycles needed to execute the entire loop. Another feature of the node is its **Latency**, which is calculated by adding the latencies of all instructions that make it up. The third feature of the node based from its instruction latency is its **Slack**. The slack is determined by finding the difference in cycle levels from both the top-down and bottom-up scheduling methods. A node with a large amount of slack can be scheduled in a variety of cycles without changing the overall latency of the loop. Another feature that makes up a node's state is the **Flows** feature. Flows is an estimate of how many DSWP cross-core flows will be created by cutting above or below the given node. Finally, **Percent** defines the percentage of nodes that are currently assigned to both partitions.

The actions the learner is allowed to pick for a given node are ABOVE and BELOW. An action of above means the all of the parents of the node will be added to the first DSWP partition and the node and its children will be added to the second partition. An action of BELOW will do the same, except the node will be added to the first partition instead of the second. The rewards given to the learner after making actions are based on a delayed system. Each reward is the speedup of the loop from a baseline time where speedup is calculated as:

$$Speedup_{\frac{DSWP}{Base}} = \frac{(exec.\ time)_{Base}}{(exec.\ time)_{DSWP}} \tag{4.6}$$

Using this definition for the reward of a program means that the reward is only found after all instructions have been partitioned and the program has been executed.

4.5 Transform Source Code

After all of the earlier code analysis has completed and the target loop's instructions have been assigned to partitions, a new LLVM pass is called to handle the actual code transformation. New functions are created to handle each partition, communication is added between the partitions, and finally, code is inserted to allow the loop to run in parallel. As discussed earlier, execution time information is also collected to assist machine learning so function calls are added to allow precise timing of loop execution. All of these transformations are done at the level of LLVM IR so that they are language and architecture independent and can be done in tandem with any other optimization allowed by the LLVM compiler.

4.5.1 Move Instructions into New Partition Functions

After each instruction has been assigned to a partition, the code must be transformed to create the dual threads. This process begins by creating a new function for each of the DSWP partitions. After each function is created, the body of the loop is copied into both functions, keeping the BB structure of the original loop. Next, the instructions that do not belong to a partition are deleted from its function, being careful not to corrupt the original CFG of the loop. Finally, all instruction operands within the new partition functions are updated to point to the newly created instructions inside the DSWP functions. If an operand does not have a corresponding instruction within its new function it is marked as depending on an outside value which will be handled in the next

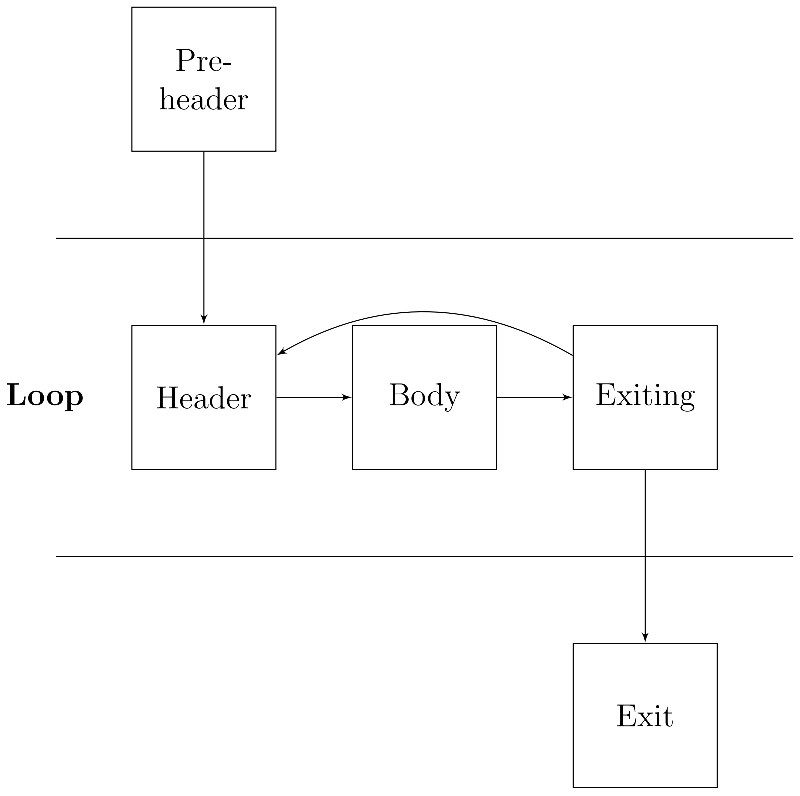


Figure 4.9: Loop BB Structure

step.

Two new basic blocks are also added to each function to act as loop pre-header and exit blocks for the loop body within each partition function. A loop pre-header is a basic block that dominates the loop which implies that the loop will never be executed by the program without the pre-header block executing first. The loop pre-header in this case also acts as the entry node for the partition function. The exit block is much like the pre-header block but instead post-dominates the loop which implies that it will also run after the loop finishes executing. Both of the blocks are used for housekeeping in later steps.

4.5.2 Adding Flows to Partitions

There are two main types of data flows that can be generated by partitioning the program: internal and external. The external flows come from the fact that some values used by the loop are created or consumed outside of the loop. Therefore we must be able to pass information in and out of the newly created loop partition functions. This is done using parameters for each of the functions. For data that must be passed into the loop, it is stored into memory outside of the loop and it is passed in by reference into the loop where it is loaded from memory in the pre-header block within the partition function. This is done in the pre-header block so that it is only loaded once instead of being loaded during each iteration of the loop. For data that must be passed out of the loop, memory is allocated outside of the loop and a memory address is passed into the loop function. The final value of the data is then stored in the exit block of the loop and loaded outside of the function so that it can be used by the main program.

The second kind of flow that can be created is a flow between the two partitions themselves. Previously, this was done using a custom built Synchronization Array that existed in hardware that could be accessed using new machine level *push* and *pop* instructions [34]. One purpose of this project was to create a hardware independent implementation of this array. This was done using a custom library built around Intel's Thread Building Block Library. The library focuses around the *dswp_flow struct* whose definition can be found in Appendix B.1. Both the TBB library and the specialized library functions written specifically for DSWP (shown in Appendix B.2) were careful to take into consideration the challenges discussed by Ragan and others in their original DSWP publications [34]. The original challenges that created considerable overhead were OS synchronization used for locking the queues to make them thread-safe, and problems with false-sharing and cache pollution because of shared memory communication between the two threads. TBB includes memory allocation functions that make sure to stride across cache blocks to reduce false sharing between communicating threads [17]. These allocators were used for all memory allocation within the *dswp_flow struct*. Care was also taken to allow all locking to be done by the TBB library which allows fine-grained locking of the queues to reduce delay when reading and writing the queues.

To pass data into a DSWP flow, the producing partition asks the flow structure for an available memory location. The flow structure responds with a pointer if there is room within the queue to add a value or the function blocks until the queue is no longer full. The producing partition then saves the value into the memory location and alerts the flow that it has finished storing into memory. To consume a value from a DSWP flow, the DSWP partition asks the flow structure for the address of the oldest value within the flow. The flow structure will then return the address or block if the flow is empty. The partition can then load from memory to obtain the value passes from the producing partition. The consuming partition does not need to alert the flow structure that it has finished loading from memory because of the design of the flow structure. This feature is possible because the consuming function always loads the shared value before asking for the pointer to the next value in the flow.

4.5.3 Insert Functions into Code

After the functions are constructed, the original code of the loop is deleted from the program. The original pre-header for the loop is changed to instead point to a newly constructed BB. The new BB is given code to construct and destruct the *dswp_flow struct* before and after the loop. It is also given code to zip

```
PREHEADER:
...
br %DSWP_BLOCK

DSWP_BLOCK:
call dswp_flow_init()
zip parameters into array
...
call create_thread(Partition2)
call Partition1()
call join_thread()
call dswp_flow_dest()
br %EXIT

EXIT:
...
```

Figure 4.10: Pseudo-BB to Inject DSWP Partitions

any parameters that need to be passed into the second partition. Next a new thread is created inside of the BB which allows the 2nd partition to run. After the thread is created, the function for the master thread (the 1st partition) is called. After both partitions are called, a join instruction is issued to make sure both threads complete before the program moves ahead in execution. Finally the BB is given a branch instruction that points to the exit block of the transformed loop. A pseudo-code implementation of this process is shown in Figure 4.10.

4.5.4 Learning Information

For the purpose of machine learning, timing data must be collected on each of the partitioned loops. This is done by using a few functions injected into the code that log timing information for each loop. The declaration of these function can be seen in Appendix B.3. Timing information for loop execution time is stored out into files and used by the learner to calculate the speedup of a particular partition and therefore the reward for a set of actions. The timing functions are added as the first instruction and 2nd to last of the new BB created at code insertion as described in Figure 4.10

Chapter 5

Experimentation

Two sets of experiments were run to verify the feasibility of the proposed optimization. All experiments were run on a set of kernels created for this specific research. The first set of experiments were used to find the best straight cut DSWP partitioning using hand tuning. This was done to verify if a performance increase could be found for each kernel. The second set of experiments focused on if the machine learner could discover a jagged cut DSWP partitioning that performed as well or better than the best partitioning found in the first experiment.

5.1 Kernels

To test the viability of the described implementations of DSWP and LA-DSWP, kernel programs were created. These kernel programs mimic real user program patterns and allowed benchmarking data to be collected for both optimizations.

5.1.1 Motivation

As mentioned in the Chapter 1, DSWP was created to take advantage of parallel processing in loops that traverse recursive data structures (RDS). These loops can usually be split into two blocks: a block that traverses the data structure, and a block that does useful calculations. By splitting the loops into two separate partitions running on multiple cores, we hope to better take advantage of the memory subsystems of modern processors.

The kernels created to be partitioned using DSWP were designed to tax a modern processor's memory system. The two principles of locality discussed in Chapter 3 usually keep caches full of the data a processor needs and thus masks the latency of main memory. These two principles, however, do not apply to some program behavior (for instance iterating through a large RDS). As an example, consider a program that moves through a large tree, doing work on each of its nodes. Each node must be loaded from memory, but most likely each node will not be stored adjacent in memory. This causes spatial locality to fail and the cache will no longer prefetch data for the processor to use. Also assume the tree is very large, maybe on the scale of gigabytes, and each node is only visited once. This means that the cache can no longer rely on temporal locality either and so will never contain the information the processor needs when loading a new node. This would force the processor to go to main memory for each and every load of a node which will severely decrease overall performance [31].

At this point the processor can still use a few optimizations to mask the latency of the main memory loads. The first is pre-fetching the data [12]. The processor is allowed to issue a prefetch to let the memory system know that the processor will need a piece of data soon. These instructions can be trigger through software or hardware. The software prefetch relies on the programmer to predict what memory addresses should be loaded early into the cache. A hardware prefetch is issued by the hardware itself and usually relies on learning a particular load pattern. Another optimization the processor can use to hide the latencies of main memory is an out-of-order pipeline [17]. As described in Chapter 3, modern processor do not actually execute the instructions of a program in the compiled order. The processor can instead reorder instruction in a set window of instructions. For instance the processor used in this research is able to reorder instructions within a 192 instruction window. Additionally, that the processor can have 72 pending loads from main memory at a time means that the processor can stack many loads together and so hide the latency of later loads behind the latency of the first [22]. These processor optimizations can still fail, however, on an RDS. A tree's nodes may be stored randomly in memory and so the hardware pre-fetcher may be unable to learn a load pattern, and the there may be more instructions used to do calculations on each node than can be held in the reorder buffer and so multiple loads cannot be sent to memory at one time.

DSWP cannot fix the locality problems in the caches themselves and it cannot help with prefetching. It can, however, help will the the reorder buffer problem. The goal of DSWP is to split out all of the calculation instructions in a RDS loop into a separate processor core so that while data will always have to be fetched from main memory, as many loads as possible can be done together. This is the driving idea behind the kernels created to test this implementation of DSWP. All of the kernels were created to have almost no temporal or spatial locality. They also do not follow any patterns the hardware pre-fetchers can learn and the calculations done on each piece of data would completely fill the processors reorder buffer so that loads can not be bundled together.

5.1.2 Kernel Definitions

The kernels written for this research focused on different patterns of RDS traversal. Three common data structures were used as the backbone of the kernels: matrices, binary trees, and linked lists. For each data structure three different algorithms were ran on the structure for a total of nine kernels. The matrices used in the kernels are C matrices built from an array of arrays. The binary

Table 5.1: Experiment Loop Summary Information

Kernal	Total Latency	DAG Node Count
Matrix Multiply	75	52
Matrix Log Loss	62	21
Matrix Arbitrary	74	20
Tree Variance	73	6
Tree Log Loss	115	18
Tree Arbitrary	112	14
Linked List Variance	16	6
Linked List Log Loss	65	20
Linked List Arbitrary	62	15

trees are sorted, unbalanced trees whose nodes are all the size of a single cache block (64 bytes). The linked lists nodes are also the size of a single cache block. A summary of the kernels used for these experiments is shown in Table 5.1. The **Total Latency** of each loop is the total number of cycles needed to execute a single iteration of the RDS loop based on a simple model of Haswell latencies. The **DAG Node Count** is the number of nodes in the DAG of the RDS loop in the kernel.

For the first three kernels, each data structure was used to store the data needed to calculate the log loss of the output of a machine learning classifier. The log loss of a machine learner across N predictions can be written as follows:

$$Log\ Loss = -\frac{1}{N}\sum_{n=0}^{N-1} y_n * \log p_n \tag{5.1}$$

where y_n is the true classification of a given sample n, and p is the probability the learner assigned to the true classification [7]. Another set of kernels find the variance across the nodes of a given data structure. The variance of a list of numbers can be calculated as:

$$s^2 = \frac{1}{N-1}\sum_{n=0}^{N-1}(y_n - \bar{y})^2 \tag{5.2}$$

where N is the number of samples, y_n is sample n, and $\bar{y}$ is the mean of the data set [9]. An example of a kernal finding variance can be found in the appendix. The matrix data structure also has a kernel that multiples two matrices. It is the only kernel that has been hand-optimized to perform well in a multi-threaded application. Finally each data structure has an "arbitrary" algorithm preformed on each of the nodes. The arbitrary algorithm simply runs a loop that performs multiple operations on the values at each node of the data structure before moving on. An example of these arbitrary calculations can be found in the appendix. This was done to guarantee an algorithm that would fill up the ROB with calculation instructions between the loading of each node.

5.2 Hardware Setup

All experiments were run on a single, unshared node of the University of Oklahoma's supercomputer, Schooner. Each node of Schooner contains two Intel Xeon Haswell E5-2650v3 10-core 2.3 GHz processors along with 32 GB of RAM. All testing was limited to a single processor so that partitions could communicate through the processor's Uncore as discussed in Chapter 2.

5.3 Experiment 1 Setup

In the first set of experiments, the viability of running DSWP optimizations on current hardware was tested. To do this, the target loop in each kernel was split into two DSWP partitions using a straight cut. This straight cut method closely matches the method of automatic partitioning done in prior research [28]. The total latency of the loop was calculated along with which cycle each node would begin running (the node's cycle level). These were then used to find the cycle ratio of the node by dividing the node's cycle level by the overall latency of the loop. After these calculation, for most kernels, all nodes with a level ratio between 0 and 0.5 went into one partition, while the rest went into the second partition. For a few kernels, however the cutoff threshold of 0.5 was changed to increase performance because the latency of the calculation and traversal paths was lopsided.

After each kernel was partitioned, a baseline test was run. Timing functions were injected into the original code at the LLVM IR level which was then compiled and run 10 times. The outputs of the timing functions were then averaged to find the baseline execution time for the target loop in the kernel. Next, the partitioned version of the kernel was run 10 times. The output of its timing functions were then averaged to find the execution time after optimization. Using these calculations the performance speedup was determined.

5.4 Experiment 1 Results

Table 5.2 and Figure 5.1 summarize the results using the straight cut method of DSWP partitioning. As can be seen from both the table and the figure, only one kernel was able to achieve a performance increase. The matrix multiply kernel does not fit into the RDS pattern, however, and is easily handled by other methods of parallelizations, such as DOACROSS. The rest of the programs run at speeds ranging from 37% to 1% of the original unoptimized speeds.

This leads to the question of why such large slowdowns? Our opinion

Table 5.2: Experiment 1 Results

Kernal Name	Cut Ratio	Speedup
Matrix Multiply	0.5	1.78
Matrix Log Loss	0.52	0.01
Matrix Arbitrary	0.5	0.15
Tree Variance	0.5	0.03
Tree Log Loss	0.5	0.37
Tree Arbitrary	0.5	0.16
Linked List Variance	0.5	0.01
Linked List Log Loss	0.85	0.12
Linked List Arbitrary	0.82	0.21

follows the same reasoning as the original DSWP researchers [34]. As discussed in Chapter 2, the DSWP partitions are only able to communicate through the processor's L3 cache. Intel estimates the average latency of a dirty hit in the L3 cache (occurs when a core tries to read data another core has written) to be about 110 cycles. They also estimate the average latency of a hit in DRAM to be about 193 cycles [21]. The DSWP partition that is on the receiving end of the data flows will have at least one dirty hit in the L3 cache every time it consumes a value from the queue. This means at best, it will be able to save 83 cycles by letting the other partition load the data first. This is, however, ignoring all of the other overhead required to handle the communication like updating queue states. Also communication queues must be created between the partitions to handle control flows which will create more dirty loads that did not exist before partitioning and so will help negate the saved cycles.

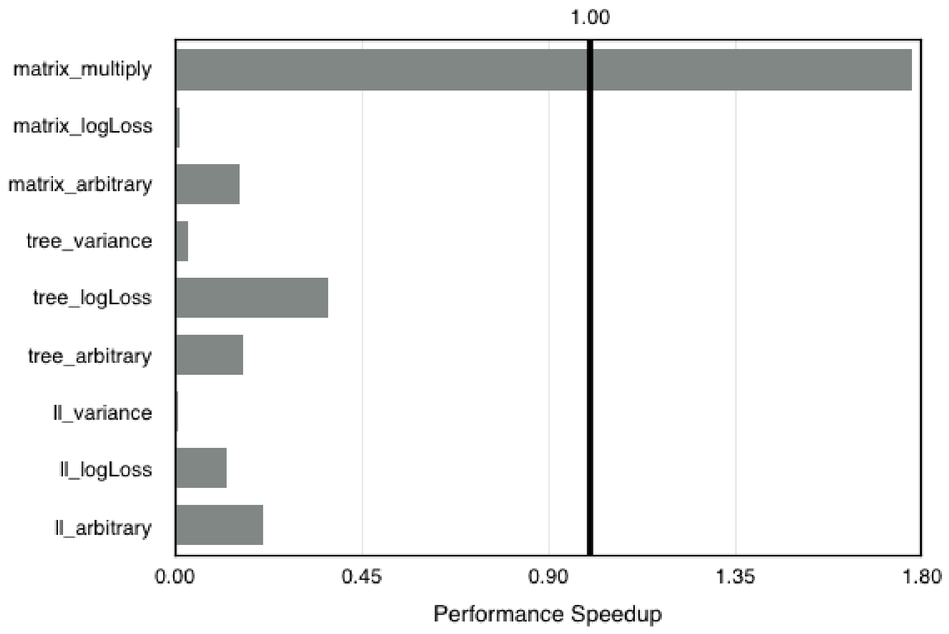


Figure 5.1: Experiment 1 Results

5.5 Experiment 2 Setup

Even with the failure of DSWP to produce a performance increase for most kernels in Experiment 1, machine learning was still applied to the problem to discover whether or not reinforcement learning was a viable method to determine how best to partition code into multiple DSWP partitions. Only two kernels were selected to perform machine learning on, matrix multiplication and the arbitrary matrix kernel. Matrix multiplication was chosen specifically because it was the only kernel in which a DSWP partitioning resulted in a performance increase.

For each machine learning run, an initial Q-Matrix was created with all values initialized to 1.5. This value was chosen as it is on the high-side of the expected reward for each state-action pair; this setup promotes early exploration of the problem space. For each trial of the ML run, the code was partitioned using an epsilon-greedy policy using $epsilon = 0.1$. After the code was partitioned, it was executed and timing information was collected. This information was used to calculate a reward for the actions chosen during that specific trial. The Q-Table was then updated using Formula 4.1. The reward for a given trial is the speedup of the optimized code over the un-optimized code. The reward was given to the last state-action pair chosen only, with all other rewards in the trial being set to 0. This method required a high γ value so that the reward is able to trickle back to earlier choices. For these experiments $\gamma = 0.95$. Each ML run included at least 1000 trial compilation and execution pairs resulting in the same number of Q-Table updates.

Two separate learners were run on each kernel, each with its own learning rate(α). A high learning rate can make the model very sensitive to noise, but the model will converge to a solution faster. The α values used for this experiment where $\alpha = 0.1$ and $\alpha = 0.3$.

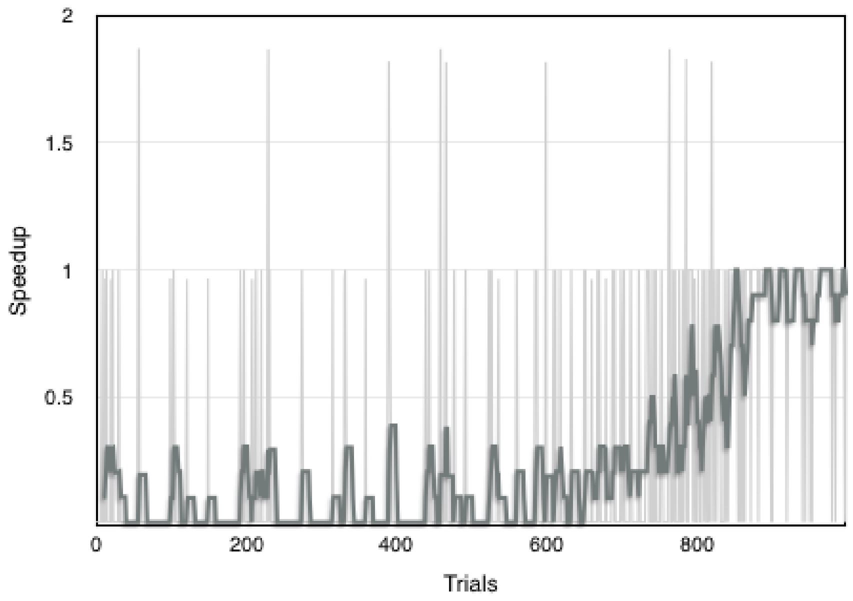
5.6 Experiment 2 Results

Learning curves for each kernel are summarized in Figures 5.2a and 5.3. The lighter line in both figures is the speedup of each trial in the given run. The darker line is a moving average filter with a window of 10 of the speedups of the trials. As was conjectured in Section 5.5, the higher α values did in fact help the learner converge on a solution quicker.

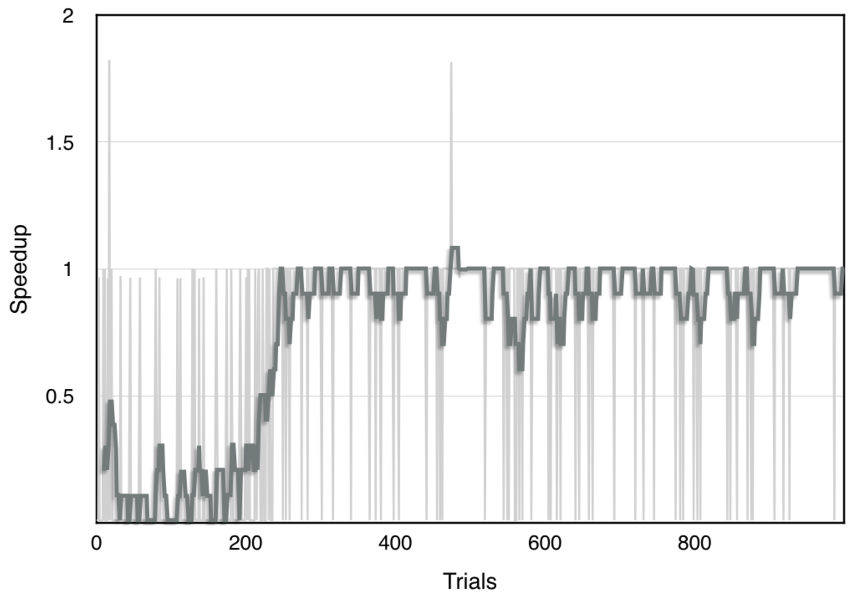
For both kernels, neither learner was able to find a partitioning that had a greater performance increase than the straight line partitioning used in Experiment 1. In fact, for the matrix multiplication kernel, the learner explored the partitioning that lead to a speedup of 1.78x, but it did not converge on this solution. Even after 10,000 trials the learner was unable to converge on the optimal solution. It instead converged on the same solution as the arbitrary kernel's learner: do not partition the loop. For the arbitrary kernel, this is the best and only solution that does not degrade performance.

Both figures also show that the speedup for a giving partitioning is very much all or nothing. For instance, in the matrix multiplication kernel all partitioning trials seem to produce a speedup of approximately 0, 1, or 1.8, with nothing really in between. These extremes may be one reason the learners have trouble converging on optimal solutions. There is no gradual increase of reward as the pattern of node choice approach the best solution. The solutions for a reward of 1 and 1.8 are local maximums that are surrounded on both sides by choices that result in a reward of close to 0. Also the learner only has to take a single good action to choose to partition all nodes in a single thread (leaving the original code unchanged) while it must take multiple good actions sequentially to partition nodes across the two partitions in the proper configuration to result in a reward of 1.8.

What this experiment does not discover, is how well the above results will generalize to other programs. Since only one kernel was found that had a performance increase, there is not a way to check to see if the Q-values learned for these kernels are overfit to these specific programs. More testing would need to be done with different kernels of a variety of patterns to discover how well this data can generalize.

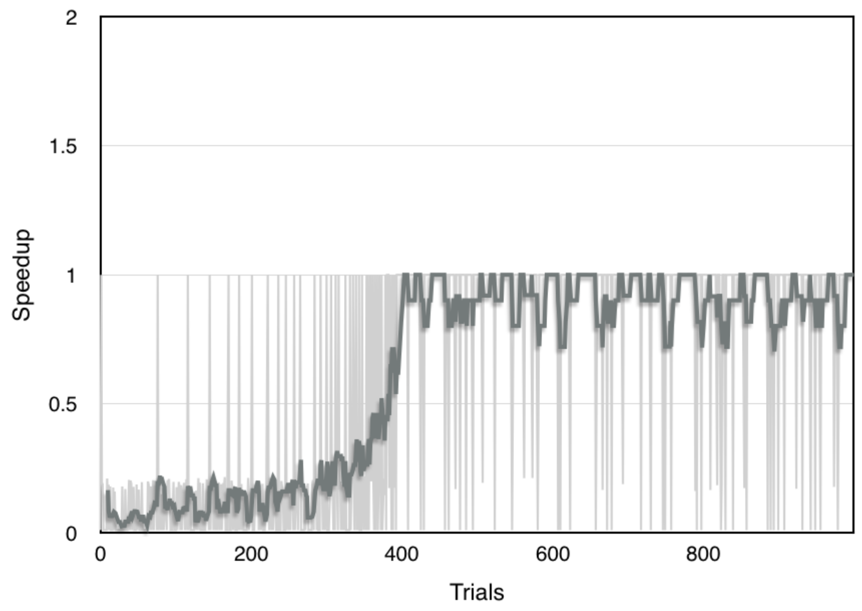


(a) Learning Curve ($\alpha = 0.1$)

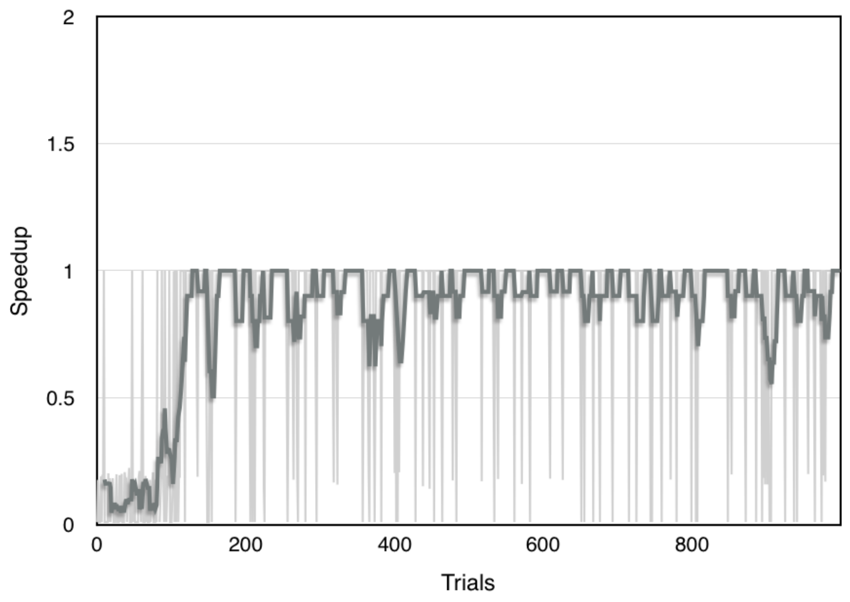


(b) Learning Curve ($\alpha = 0.3$)

Figure 5.2: Matrix Arbitrary Learning Curves



(a) Learning Curve ($\alpha = 0.1$)



(b) Learning Curve ($\alpha = 0.3$)

Figure 5.3: Matrix Multiply Learning Curves

Chapter 6

Conclusion

In this paper we discussed a new method to optimize single threaded programs to run on multi-core processors. As computer architects strive to keep up with public expectations for processor performance, they are increasingly turning to processors with multiple cores. Unlike prior hardware innovations, computer programs must be written or compiled in new methods to take advantage of these new hardware innovations. Automatic thread-extraction using Decoupled Software Pipelining seeks to extract multiple threads from a single-threaded program so that an arbitrary piece of code can make use of multiple cores on a processor chip. DSWP focuses on splitting large recursive data structure traversal loops into multiple threads to increase overall program performance.

Unlike prior implementations of DSWP, this research focused on creating a hardware and language independent implementation of DSWP using the LLVM framework. Instead of relying on custom-built hardware to facilitate communication between program threads, this implementation used Intel's Thread Building Blocks library to create queues in shared memory between the different processor cores. This cased a heavily reliance on the memory subsystems of the target processors.

Another novel approach to DSWP explored in this paper was the application of machine learning to the partitioning process. Instead of partitioning loops using predefined heuristics, this paper sought to apply reinforcement learning to allow the DSWP agent to make more informed decisions when optimizing a given loop.

Through experimentation on modern Intel processors, this research found that DSWP is unfeasible on current hardware. The overhead needed to facilitate thread communication through standard memory subsystems outweighs the performance increases caused by executing code on multiple processor cores. Machine learning was still applied, however, and while the learners were able to converge to a reasonable solution, they were unable to find the best solutions for a giving loop partitioning problem.Unfortunately, the generalizability of the learners is unknown since most programs in the experiment were unable to be partitioned in a manner that increased program performance.

6.1 Future Work

This research focused on running DSWP on RDS loops on a current Intel Processor. As discussed in Chapter 2, AMD currently uses a very different architecture to handle communication between cores in a processor. This architecture different could significantly change the outcomes of the experiments and could result in a performance increase using DSWP. Also, the current trend in processors is the scalability of the number of cores on a given chip, while not necessarily focusing on the latency of communication between cores. This can be seen in the general increase of latency in Intel chips from the original Intel Core Duo processors to the current Haswell chips [22]. This trend may someday reverse and focus on reducing latency between cores so that optimizations like DSWP have a better chance at working.

Another frontier to continue exploring is the application of machine learning to automatic thread extraction. Reinforcement learning was applied in this research using a discrete set of features and states along with a Q-table but other machine learning methods could be applied to gain greater results. A neural network or simple linear regression model could instead be used to predict Q-values for a set of continuous features so that they states used for node partitioning could be fine-tuned compared to the bulky states described in this paper. Hopefully, by exploring these new frontiers, a form of LA-DSWP could be created that was able to overcome the inter-core communication overhead of a real processor to produce performance increases in general computer programs.

Bibliography

[1] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke, “LLVA: A low-level virtual instruction set architecture,” in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36. IEEE Computer Society, December 2003.

[2] V. Agarwal, M. Hrishikesh, S. W. Keckler, and D. Burger, “Clock rate versus IPC: The end of the road for conventional microarchitectures,” in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003, pp. 84–95.

[3] AMD, “Software optimization guide for amd family 17h processors,” AMD, Tech. Rep. 55723, August 2017.

[4] R. D. Barnes, E. M. Nystrom, J. W. Sias, S. J. Patel, N. Navarro, and W. W. Hwu, “Beating in-order stalls with ‘flea-flicker’ two-pass pipelining,” in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003, pp. 387–398.

[5] R. D. Barnes, J. W. Sias, E. M. Nystrom, S. J. Patel, J. Navarro, and W. W. Hwu, “Beating in-order stalls with ‘flea-flicker’ two-pass pipelining,” *IEEE Transactions on Computers*, vol. 55, no. 1, pp. 18–33, January 2006.

[6] M. Clark, “A new x86 core architecture for the next generation of computing,” AMD, Tech. Rep., August 2016.

[7] G. Cybenko, D. O’Leary, and J. Rissanen, Eds., *The Mathematics of Information Coding, Extraction and Distribution*, ser. IMA volumes in mathematics and its applications; 107. New York, NY: Springer New York, 1999.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, Oct. 1991. [Online]. Available: http://doi.acm.org/10.1145/115372.115320

[9] B. S. Everitt, Ed., *Cambridge Dictionary of Statistics*, 2nd ed. Cambridge, UK: Cambridge University Press, 2002, ch. Variance, pp. 388–389.

[10] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, July 1987. [Online]. Available: http://doi.acm.org/10.1145/24039.24041

[11] A. Fog, *Instruction Tables*, Technica University of Denmark, May 2017.

[12] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," in *Proceedings of the 25th annual international symposium on Microarchitecture*, December 2002, pp. 102–110.

[13] S. H. Fuller and L. I. Millett, Eds., *Future of Computing Performance, The: Game Over or Next Level?* National Academies Press, 2011.

[14] P. P. Gelsinger, "Microprocessors for the new millennium: Challenges, opportunities, and new frontiers," in *2001 IEEE International Solid-State Circuits Conference. Digest of Technical Papers. ISSCC (Cat. No.01CH37177)*, Feb 2001, pp. 22–25.

[15] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2017.

[16] M. T. Heath, "A tale of two laws," *The International Journal of High Performance Computing Applications*, vol. 29, no. 3, pp. 320–330, August 2015.

[17] J. L. Hennessy and D. A. Patterson, *Computer Architecuture: A Quantitative Approach*, 5th ed. Waltham, MA: Elsevier, Inc., 2012.

[18] K. Huppler, "The art of building a good benchmark," in *Performance Evaluation and Benchmarking: First TPC Technology Conference*, R. Nambiar and M. Poess, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 18–30.

[19] Intel threading building blocks documentation. Intel Corporation. [Online]. Available: https://software.intel.com/en-us/tbb-documentation

[20] Intel® Xeon Phi™ x200 product family. Intel Corporation. [Online]. Available: https://ark.intel.com/products/series/92650/Intel-Xeon-Phi-x200-Product-Family

[21] *Using Intel VTune Amplifier XE to Tune Software on the Intel Xeon Processor E5 v3 Family*, Intel Corporation.

[22] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Intel Corporation, December 2017.

[23] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *The Seventh International Symposium on High-Performance Computer Architecture.*, January 2001, pp. 197–206.

[24] J. Larus, "Spending moore's dividend," *Commun. ACM*, vol. 52, no. 5, pp. 62–69, May 2009. [Online]. Available: http://doi.acm.org/10.1145/1506409.1506425

[25] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on.* USA: IEEE, 2004, pp. 75–86.

[26] T. Lengauer and R. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 1, no. 1, pp. 121–141, January 1979.

[27] G. E. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, Jan, 1998.

[28] G. Ottoni, R. Rangan, A. Stoler, and D. August, "Automatic thread extraction with decoupled software pipelining," in *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on.* USA: IEEE, 2005, pp. 12 pp.–118.

[29] D. Padua Haiek, "Multiprocessors: Discussion of some theoretical and practical problems," January 1980. [Online]. Available: http://search.proquest.com/docview/288131513/

[30] V. Pankratius, A.-R. Adl-Tabatabai, and W. Tichy, Eds., *Fundamentals of Multicore Software Development*, ser. Computational Science Series. CRC Press, 2012.

[31] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 5th ed. Morgan Kaufmann Publishers Inc., 2014.

[32] M. Priestley, *A Science of Operations Machines, Logic and the Invention of Programming*, ser. History of Computing. Springer, London, 2011.

[33] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August, "Parallel-stage decoupled software pipelining," in *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '08. New York, NY, USA: ACM, 2008, pp. 114–123. [Online]. Available: http://doi.acm.org/10.1145/1356058.1356074

[34] R. Rangan, N. Vacharajani, M. Vachharajani, and D. August, "Decoupled software pipelining with the synchronization array," in *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, September - October 2004, pp. 177–188.

[35] R. S. Sutton and A. G. Barto, *Reinforcement learning: An Introduction.* The MIT Press, 1998.

[36] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, June 1972.

[37] N. Vacharajani, R. Rangan, E. Raman, M. Bridges, G. Ottoni, and D. August, "Speculative decoupled software pipelining," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, September 2007, pp. 49–59.

[38] C. Watkins and P. Dayan, "Q -learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, May 1992.

[39] N. H. E. Weste and D. M. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th ed., M. Hirsch, Ed. Addison Wesley, 2001.

[40] T. Yeh and Y. N. Patt, "Alternative implementations of two-level adaptive branch prediction," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, August 1992, pp. 124–134.

Appendices

Appendix A

Notation and Nomenclature

CFG	Control Flow Graph
CMP	Chip Multiprocessor
BB	Basic Block
DAG	Directed Acyclic Graph
DSWP	Decoupled Software Pipelining
IR	Intermediate Representation
LLVM	Low Level Virtual Machine
ML	Machine Learning
PDG	Program Dependency Graph
PDT	Post-Dominator Tree
RDS	Recursive Data Structures
RL	Reinforcement Learning
SARSA	State Action Reward State Action Learning
SCC	Strongly Connected Components
SPEC	System Performance Evaluation Cooperative
SSA	Static Single Assignment
TBB	Intel's Thread Building Block Library

Appendix B

Selected Code Snippets

B.1 Inter-Partition Flow Structure

```
struct dswp_flow
{
        concurrent_bounded_queue<void*,
                cache_aligned_allocator<void*> >** queues
                    ;

        char padding[64];

        int numberOfQueues;
        void** dataMaps;

        char padding1[64];

        int* nextIndex;
        void** nextAddress;

        char padding2[64];

        size_t* sizes;

        char padding3[64];

        int* pops;
        int popIter;

        char padding4[64];

        int* feeds;
        int feedIter;

} typedef dswp_flow;
```

B.2 DSWP Flow Library Header

```
int dswp_flow_init(dswp_flow* flow,
                   int flowNumber, size_t* sizes);

void dswp_flow_dest(dswp_flow* flow);

void* dswp_flow_feedFlow_Get(dswp_flow* flow, int index);

void dswp_flow_feedFlow_Flag(dswp_flow* flow, int index);

void* dswp_flow_consumeFlow(dswp_flow* flow, int index);

void dswp_flow_start_pop_iter(dswp_flow* flow);

void dswp_flow_end_pop_iter(dswp_flow* flow);

void dswp_flow_start_feed_iter(dswp_flow* flow);

void dswp_flow_end_feed_iter(dswp_flow* flow);
```

B.3 ML Timing Library

```
void dswp_timing_start(struct timeval* tv);

void dswp_timing_end(int loopNum, struct timeval* tv);
```

B.4 Linked List Variance Kernel

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define size 100000000

struct ll_node
{
    struct ll_node* pred;
    struct ll_node* succ;
    float value;
```

```
} typedef ll_node;

void create_node(ll_node* node, float value)
{
    node->pred = 0x0;
    node->succ = 0x0;
    node->value = value;
}

ll_node* add_node(void* address, ll_node* tail, float
    value)
{
    ll_node* newNode = (ll_node*) address;
    create_node(newNode, value);

    tail->succ = newNode;
    newNode->pred = tail;

    return newNode;
}

int main()
{
    //build up a giant LL

    FILE* fp;
    float value = 0;
    fp = fopen("random.csv", "r");

    //create front
    fscanf(fp, "%f", &value);
    ll_node* front = malloc(sizeof(ll_node));
    create_node(front, value);

    ll_node* tail = front;

    //build rest of LL
    for (int i = 0; i < size-1; ++i)
    {
        void* address = malloc(sizeof(ll_node));
        fscanf(fp, "%f", &value);
        tail = add_node(address, tail, value);
    }

    printf("%f\n", front->value);


    //sum up across all values in the LL to find mean

    float sum = 0;

    ll_node* current = front;

    while (current != 0x0)
    {
        ll_node* temp = current;
        current = current->succ;

        sum += temp->value;
    }

    float mean = sum / size;

    //find variance now that we have the mean
    sum = 0;
    current = front;
    while(current != 0x0)
    {
        ll_node* currentNode = current;
        current = current->succ;

        float error = currentNode->value - mean;
                    error = error * error;
                    sum = sum + error;
    }

    float variance = sum / (size - 1);
    printf("%f\n", variance);
}
```

B.5 Arbitrary Algorithm Calculations

```


```

```
//do a whole bunch of worth with just the
//values hopfully causing the ROB to fill
//while waiting for miss
temp = 0;
for (unsigned int i = 0; i < 150; ++i)
{
   double temp = currentNode->values[i % 5] * i;
   temp = temp / (i+1);
}
```

B.6 Latency Calculation Code

```
unsigned int getLatency(Instruction* inst)
{

    unsigned OpCode = inst->getOpcode();

    switch (OpCode)
    {
        //terminators
              ----------------------------------------
        case Instruction::Ret: ;
        case Instruction::Br: return 2;

        case Instruction::Unreachable: return 0;

        //binary-----------------------------------------
        case Instruction::Add: ;
        case Instruction::FAdd: ;
        case Instruction::Sub: ;
        case Instruction::FSub: return 1;

        case Instruction::Mul: ;
        case Instruction::FMul: return 2;

        case Instruction::UDiv: ;
        case Instruction::SDiv: ;
        case Instruction::FDiv: ;
        case Instruction::URem: ;
        case Instruction::SRem: ;
        case Instruction::FRem: return 15;

        //bitwise
        case Instruction::And: ;
        case Instruction::Or: ;
        case Instruction::Xor: ;
        case Instruction::Shl: ;
        case Instruction::LShr: ;
        case Instruction::AShr: return 1;

        //memory
        case Instruction::Alloca: ;
        case Instruction::Load: ;
        case Instruction::Store: return 5;
        case Instruction::GetElementPtr: return 2;


        //other
        case Instruction::ICmp: return 1;
        case Instruction::FCmp: return 2;

        case Instruction::PHI: return 0;

        case Instruction::Call: return 50;

        default: return 1;
    }

}
```