Automatic Compiler Customization for Novel Hardware

Automatische Anpassung von Compilern für neuartige Hardwarearchitekturen
Master thesis by Janne Wulf
Date of submission: November 1, 2021

1. Review: Dr. rer. nat. Stefan Guthe
2. Review: Dr.-Ing. Daniel Thuerck
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

Interactive Graphics Systems Group

Perceptual Graphics, Capture and Massively Parallel Computing

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Janne Wulf, die vorliegende Masterarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, November 1, 2021

J. Wulf

Abstract

With the growing number of processor architectures and specialized accelerators, compilers have become once again a critical tool for effective use of new capabilities. Compilers have hardware-dependent parts that need to be manually adjusted and optimized by experts for every target hardware. One major step of the hardware-dependent compilation process is instruction scheduling. Regarding their scheduling mechanisms, there are two groups of processors, in-order and out-of-order processors, where the latter can re-schedule instructions during execution in hardware.

This thesis aims to evaluate whether this step can be automatically optimized, especially for novel hardware. We have multiple contributions through our work: First, we develop a methodology for automatically creating optimized instruction scheduling policies for any hardware using data-driven approaches. Next, we build a pipeline for automatic basic block micro-benchmark creation. A large part of that pipeline is concerned with automatically standardizing basic blocks coming from various C/C++ projects. Lastly, we collect a dataset of basic blocks extracted from the LLVM Test Suite [1].

Our experiments aim to compare the effectiveness of our approach on different processors. We use an in-order AArch64 CPU (ARM Cortex-A53) and the NEC SX-Aurora TSUBASA, an out-of-order vector accelerator. To summarize, we search for well-performing instruction schedules on a set of micro-benchmarks. In an effort to reduce inference times, we train different supervised learning models with the results of the search approach. We generate instruction schedules for the basic blocks in our test dataset, which, on average, perform 8.35% (in-order) and 0.30% (out-of-order) better than the LLVM compiler framework. Our supervised learning models generate instruction schedules that perform, on average, 1.38% (in-order) better. On the out-of-order processor, we do not achieve a speedup with the supervised learning models.

[1] `https://llvm.org/docs/TestSuiteGuide.html`

Zusammenfassung

Mit einer steigenden Anzahl an Prozessorarchitekturen und spezialisierten Beschleunigerkarten sind Compiler erneut zu einem entscheidenden Werkzeug für die effektive Nutzung von neuen Fähigkeiten geworden. Compiler haben hardwarespezifische Teile die für jeden neuen Prozessor von Experten manuell angepasst und optimiert werden müssen. Ein Hauptschritt des hardwarespezifischen Kompilierens ist das Instruction Scheduling. Bezüglich ihrer Mechanismen zum Instruction Scheduling können Prozessoren in zwei Gruppen eingeteilt werden: in-order und out-of-order Prozessoren, wobei letztere während der Ausführung Instruktionen in der Hardware noch umsortieren können.

Das Ziel dieser Thesis ist zu untersuchen ob das Instruction Scheduling durch einen automatisierten Vorgang optimiert werden kann, insbesondere für neuartige Hardware. Wir leisten mehrere Beiträge mit dieser Arbeit: Wir entwickeln mithilfe von datengetriebenen Ansätzen eine Methodik für die automatisierte Erstellung von optimierten Instruction Scheduling Strategien für jegliche Hardware. Außerdem erstellen wir eine Folge von Arbeitsschritten für die automatisierte Generierung von Mikrobenchmarks in der Größe von einzelnen Basic Blocks. Ein großer Teil dieser Arbeitsschritte befasst sich mit der automatischen Vereinheitlichung von Basic Blocks aus verschiedenen C/C++ Projekten. Zusätzlich erstellen wir einen Datensatz an Basic Blocks aus der LLVM Test Suite [1].

Das Ziel unserer Experimente ist es die Wirksamkeit unserer Methoden auf verschiedenen Prozessoren zu untersuchen. Dafür, benutzen wir einen in-order AArch64 Prozessor (ARM Cortex-A53) und die out-of-order Beschleunigerkarte NEC SX-Aurora TSUBASA. Wir suchen für die Basic Blocks in unserem Datensatz Instruction Schedules, die zu einer kürzeren Laufzeit führen. Um die langen Laufzeiten des Suchansatzes zu umgehen trainieren wir mit dessen Ergebnissen mehrere Supervised Learning Modelle. Für die Basic Blocks in unserem Testdatensatz generieren wir so Instruction Schedules die, im Vergleich zu jenen des LLVM Compiler Frameworks, durschnittlich zu einer um 8.35% (in-order) bzw. 0.30% (out-of-order) kürzeren Laufzeit führen. Die Supervised Learning Modelle generieren Instruction Schedules die im Schnitt 1.38% (in-order) schneller sind. Für den out-of-order Prozessor erreichen wir keine Beschleunigung mit den Supervised Learning Modellen.

[1] `https://llvm.org/docs/TestSuiteGuide.html`

Contents

1. Introduction

In the early days of computers, the Central Processing Unit (CPU) did all processing. Supercomputers already utilized Vector Processing Units (VPU) in the 1970s. These accelerators can execute an instruction on many data points in parallel, known as SIMD (Single Instruction Multiple Data). The Graphical Processing Unit (GPU), a new type of computing device, entered the consumer market with applications that used complex graphics in the 1990s. It was originally designed as a specialized device for parallel computations in graphics applications, but nowadays GPUs are also used for the highly parallelizable deep learning methods, even though that was not a design goal initially. The emergence of deep learning methods in the 2010s led to the development of specialized hardware for these tasks, like Tensor Processing Units (TPU) and Intelligence Processing Units (IPU).

Besides accelerators, less powerful processors created a high demand. A wide range of devices use these processors that mainly implement ARM architectures and are optimized for low power consumption. These can be hand-held devices (e.g., smartphones or tablets) or edge computing devices like single-board computers.

Compilers are programs that translate code from a programming language into a machine-executable format. There are passes of compilers that generate output specific to a given processor. These parts have to be manually adjusted and optimized to every target hardware. Experts in the field perform this time-consuming and expensive task manually.

We focus our work on one specific part of the hardware-dependent compiler phase: the instruction scheduling. Once the compiler has translated the source code into atomic instructions that the processor understands, the instruction scheduler can schedule these instructions in different orders without changing the outcome of the translated program. For example, if we want to compute $a + b + c$, it does not matter if we first compute $a + b$ and then add c or if we start with $b + c$ and then add a. However, one schedule might execute faster than the other because of the implementation details of the processor microarchitecture.

The instruction scheduling is usually not performed on the whole program at once. The instruction scheduler's unit is called a basic block, a sequence of processor instructions that always execute as a whole. Consequently, the execution of a basic block must start with its first instruction and terminate with the last instruction in the sequence.

Processors can be grouped into two categories regarding their instruction scheduling capabilities. There are in-order and out-of-order processors. While the former executes the instructions as given, the latter can re-schedule the instructions during execution in hardware. Consequently, an out-of-order processor might not execute the schedule in the compiler's defined order.

To optimize this complicated task in an automated process in order to remove the need for expensive human experts, we use data-driven and machine learning algorithms.

Research on the instruction scheduling problem already started in the 1960s, and many papers were published. Also, some research was published since the 1990s that made use of machine-learning-based approaches.

Tuning instruction schedulers to specific hardware manually is a time-consuming and challenging task. Modern processors are intricate and complex machines, which makes the instruction scheduling problem NP-complete. Since processors are that complex, the effects of re-ordering instructions are often not predictable before execution.

Computer programs can execute code significantly faster with tuned instruction schedules. While good instruction schedules benefit the runtime, lower runtimes also lead to lower energy consumption, which is especially interesting for mobile and edge computing devices.

We contribute a methodology for automatically creating optimized instruction scheduling policies for any hardware using data-driven approaches. Next, we build a pipeline for automatic basic block micro-benchmark creation. A large part of that pipeline is concerned with automatically standardizing the code coming from various C/C++ projects. Lastly, we contribute a training and test set extracted from the LLVM test suite.

This thesis seeks to find answers to multiple questions. First, how good the instruction schedules are that state-of-the-art compilers generate and if we can find better ones. Next, we investigate whether it is possible to train a model that can generate better instruction schedules than modern compilers. Lastly, we use the dataset to train various data-driven and machine learning approaches to generate well-performing instruction schedules for unknown basic blocks.

We execute all our experiments on an AArch64 in-order processor (ARM Cortex-A53) and the NEC SX-Aurora TSUBASA out-of-order processor to compare the effects of the different pipeline models. Our data-driven approach requires many basic blocks for its learnings. Therefore, we present two heuristics to select basic blocks from the LLVM Test Suite that greatly influence the overall runtime of the benchmarks. Next, we extract the selected basic blocks and integrate them into our runtime measurement framework. Our proposed pipeline starts with a Monte Carlo Tree Search to find well-performing instruction schedules for the selected basic blocks. We do this by intelligently generating different instruction schedules, executing them on the target hardware, and measuring its runtime. With the findings of this search approach, we build a dataset with evaluated instruction schedules. Lastly, this dataset is then used to train various data-driven and machine learning approaches with the goal of generating good performing instruction schedules also for unknown basic blocks.

We have found better instruction schedules for the in-order processor for 55% of the basic blocks and are on par for 37%. The resulting instruction schedules have on average 8.4% shorter runtimes than those generated by a state-of-the-art compiler. We have found better instruction schedules for the out-of-order processor for 32% and are on par for 53% of the basic blocks with an average speedup of 0.3%. Additionally, we create a nearest-neighbor model for the in-order processor that can generate instruction schedules that perform on average 1.4% better than the ones of the state-of-the-art compiler.

The thesis starts with background information in Chapter 2, which explains important concepts and details required to understand the remainder of the thesis. Next, we review existing relevant research in this field in Chapter 3. Then in detail, we explain the approach that we have used in Chapter 4. Finally, we discuss the results and findings in Chapter 5 and finish the thesis with an outlook in Chapter 6.

2. Background

In this chapter, we introduce knowledge that is required to understand the contents of this thesis. We start with an introduction to processors in Section 2.1. In Section 2.2, we give an overview on how a compiler is typically implemented and which problems it has to solve. Section 2.2.3 gives a more detailed introduction to compiler back-ends, which we aim to optimize in this thesis. We base the implementation of our proposed system on LLVM, which we introduce in Section 2.3. Lastly, we give a brief overview of the thesis's machine learning or data-driven approaches in Section 2.5.

2.1. Central Processing Unit

Almost all computer systems use one or more Central Processing Units (CPUs). It is the primary execution unit and executes the vast majority of computer programs on most systems. It also handles access to other parts of the system. These other parts can be the main memory, hard drives, network cards, or co-processors like a Graphics Processing Unit.

The CPU executes mostly simple instructions, like add, subtract, multiply, shift, or load and store data to/from the main memory. However, there are slightly more complex instructions on modern CPUs like combined multiply and add or cryptographic instructions. The Instruction Set Architecture of a processor architecture defines all of the available instructions of a CPU.

Most of the instructions do not have direct access to the main memory. The instructions typically access data stored in the CPU registers, which is the fastest memory available. Hence, before we can execute instructions on some data, we must transfer this data to the registers. We transfer the data back to the main memory once we finish our computations. For example, if we want to compute

$$a = a \times a + b \times b + c \times c \tag{2.1}$$

we have to load the data of the variables a, b, and c from memory, make the computations and store the result back to memory. Table 2.1 shows two possible example implementations in assembly language. Figure 2.2 illustrates the dependencies between the CPU instructions of the example implementations.

A CPU executes its instructions on different processing units. Typical processing units on modern CPUs are arithmetic logical units, load/store units, floating-point units, or encryption units. These units often exist redundantly on the CPUs. We refer to [20] for more detailed information.

ID	Instruction			
i_1	load	r_{arp}, @a	⇒	r_1
i_2	load	r_{arp}, @b	⇒	r_2
i_3	load	r_{arp}, @c	⇒	r_3
i_4	mult	r_1, r_1	⇒	r_1
i_5	mult	r_2, r_2	⇒	r_2
i_6	mult	r_3, r_3	⇒	r_3
i_7	add	r_1, r_2	⇒	r_1
i_8	add	r_1, r_3	⇒	r_1
i_9	store	r_1	⇒	r_{arp}, @a

(a)

ID	Instruction			
i_1	load	r_{arp}, @a	⇒	r_1
i_4	mult	r_1, r_1	⇒	r_1
i_2	load	r_{arp}, @b	⇒	r_2
i_5	mult	r_2, r_2	⇒	r_2
i_3	load	r_{arp}, @c	⇒	r_3
i_6	mult	r_3, r_3	⇒	r_3
i_7	add	r_1, r_2	⇒	r_1
i_8	add	r_1, r_3	⇒	r_1
i_9	store	r_1	⇒	r_{arp}, @a

(b)

Table 2.1.: Example instruction schedules for Equation 2.1. All the assembly code examples are written in the ILOC pseudo-code language [56]. Its instructions are of the form `instruction source` ⇒ `target`. The register r_{arp} is defined to point to the start of a memory block where all variables are stored. Variables are addressed by adding an offset to the memory address in r_{arp}. This is done by adding, e.g., `@a` for the variable a.

2.1.1. Classical Instruction Execution

The execution of instructions works in multiple stages. The stages depend on the specific implementation, e.g., the ARM Cortex A53 has eight stages. A typical example is four stages that usually exist in most CPUs. These stages are:

1. Fetch: In this stage, the CPU loads the instruction from memory, where the whole program is saved.
2. Decode: The operands of the instructions are determined. Further, the CPU computes actual memory addresses from base addresses and offsets.
3. Execution: Here, the instruction is executed with the given operands on the respective processing unit.
4. Writeback: This phase writes back the computed result to the specified register.

The CPU executes one instruction at a time in this classical execution scheme. It means that instructions can only start after all execution stages of the previous instruction have finished.

Table 2.2 compares different execution models for the computation in Equation 2.1. Table 2.1a defines the instruction schedule for this example. In this example, we assume that each of the stages, Fetch (F), Decode (D), Execution (E), and Writeback (W), take a single CPU cycle to finish. Table 2.2a shows the execution for this classical execution model. We see that the execution takes 36 cycles to finish in this example.

It is easy to see that the CPU does not fully utilize all stages. Only one stage is working at a time, and the rest of the stages are idle for the most time. This behavior is ineffective and addressed by the following execution model.

CPU Cycle	Instructions								
	i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9
1	F								
2	D								
3	E								
4	W								
5		F							
6		D							
7		E							
8		W							
9			F						
10			D						
11			E						
12			W						
13				F					
14				D					
15				E					
16				W					
17					F				
18					D				
19					E				
20					W				
21						F			
22						D			
23						E			
24						W			
25							F		
26							D		
27							E		
28							W		
29								F	
30								D	
31								E	
32								W	
33									F
34									D
35									E
36									W

(a) CPU without pipeline

CPU Cycle	Instructions								
	i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9
1	F								
2	D	F							
3	E	D	F						
4	W	E	D						
5		W	E	F					
6			W	D	F				
7				E	D	F			
8				W	E	D			
9					W	E			
10						W	F		
11							D		
12							E		
13							W		
14								F	
15								D	
16								E	
17								W	
18									F
19									D
20									E
21									W

(b) CPU with scalar pipeline

CPU Cycle	Instructions								
	i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9
1	F	F							
2	D	D	F						
3	E	E	D						
4	W	W	E						
5			W	F	F				
6				D	D	F			
7				E	E	D			
8				W	W	E			
9						W	F		
10							D		
11							E		
12							W		
13								F	
14								D	
15								E	
16								W	
17									F
18									D
19									E
20									W

(c) CPU with two-way superscalar in-order pipeline

Table 2.2.: Comparison of CPU pipeline implementations for the example in Equation 2.1 with the pipeline stages Fetch (F), Decode (D), Execution (E), and Writeback (W). The used instruction schedule is defined in Table 2.1a.

2.1.2. Scalar Pipeline Execution

This execution model uses the fact that the stages are idle most of the time. The improvement is that now, one instruction can be scheduled in every CPU cycle.

Table 2.2b illustrates that this change introduces parallelism, and the CPU executes the first three instructions interleaved. The theoretically best case is that all the stages are busy in every cycle and are never idle. This is not always possible, as we can see in the example Table 2.2b. We cannot start instruction i_4 (`mult` r_1, $r_1 \Rightarrow r_1$) in the fourth CPU cycle, because we have a dependency on the result of instruction i_1 (`load` r_{arp}, `@a` $\Rightarrow r_1$). In the fifth cycle, the result of instruction i_1 was written back and we can start instruction i_4. There are various reasons of hazards that cause these stalls in the pipeline.

- Resource Hazards: When the CPU needs to access a resource, which is already being accessed and thus blocked, the access cannot happen and has to wait. This could, for example, happen with registers that have only one connection to read their content. Redundancy of hardware units can solve or mitigate this problem.
- Data Hazards: There are three types of different data hazards when two instructions attempt to access the same register:
  - Read after write (RAW): A instruction has to wait until a previous instruction has produced the result that the latter has to read.
  - Write after read (WAR): A instruction has to wait until a previous instruction has read a register, that the latter has to overwrite.
  - Write after write (WAW): A instruction has to wait until a previous instruction has written a register, that the latter wants to write to, too.

  These hazards are typically solved by sorting the instructions or waiting and introducing No-Operation (NOP) instructions into the pipeline when it is unavoidable.
- Control Hazards: It is not always obvious which instruction should be executed next due to if/else constructs or loops. Usually, the CPU executes one branch speculatively and abort the instructions if it is the wrong branch. This problem is typically approached by branch prediction mechanisms that can be implemented in the compiler and on the hardware.

2.1.3. Superscalar Pipeline Execution

The next progression from the scalar pipeline is the superscalar pipeline model. The difference is that we can start multiple instructions in the same CPU cycle. Consequently, this increases the level of parallelism. The number of instructions that can start simultaneously depends on the specific hardware.

Table 2.2c shows an exemplary execution of a two-way superscalar pipeline. This is especially interesting to compare with the scalar pipeline in Table 2.2b. Note that now, instructions 1 and 2 are scheduled in the same CPU cycle. If we had simulated a three-way superscalar pipeline, we could have started instructions 1-3 in the first cycle. However, with 20 cycles we still have a runtime improvement of one cycle compared to the scalar pipeline.

There are various ways to implement superscalar pipelines. The three most typical implementations are:

CPU Cycle	Instructions								
	i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9
1	×	×							
2	×	×	×						
3	×	×	×						
4			×	×	×				
5				×	×	×			
6						×	×		
7								×	
8									×
9									×
10									×

(a) Execution of instruction schedule from Table 2.1a

CPU Cycle	Instructions								
	i_1	i_4	i_2	i_5	i_3	i_6	i_7	i_8	i_9
1	×								
2	×								
3	×								
4		×	×						
5		×	×						
6			×						
7				×	×				
8				×	×				
9					×				
10						×	×		
11						×			
12								×	
13									×
14									×
15									×

(b) Execution of instruction schedule from Table 2.1b

Table 2.3.: Execution of the instruction schedules Table 2.1a and Table 2.1b on a two-way superscalar in-order architecture. We assume that load and store instructions require three cycles, multiplies two cycles and adds one cycle.

- In-Order (static): These type of CPUs take the instructions as they come and check if they can be executed simultaneously. The compiler defines the instruction schedule.
- Out-of-Order (dynamic): Dynamic superscalar CPUs can see the following n instructions and reschedule them to achieve higher parallelism. This means that the compiler does not fully control the executed instruction schedule, which is essential for our work.
- Very Long Instruction Word: The compiler can define which instructions the CPU should execute simultaneously for this type of superscalar pipeline. This assignment works by wrapping multiple short instructions into a single long instruction, thus the name. These long instructions must be defined in the instruction set.

2.1.4. Instruction Schedules

Looking at Figure 2.2, we see that the dependencies allow different instruction schedules. For example, we could use a breadth-first approach and execute all the load instructions first, then all the multiplications, et cetera. On the contrary, we could use a depth-first approach and execute the load and multiplication for variable a, then for b, et cetera. Table 2.1 defines these two schedules and we compare them in this section.

With the classical execution scheme with no pipeline, the instructions schedule is irrelevant for the runtime. However, the runtime improvements that in-order CPUs gain by pipelines depend heavily on the instruction schedule.

We illustrate the execution of the two schedules in Table 2.3. This example assumes that load and store instructions need three cycles to finish, multiplications need two cycles and add instructions need one cycle. Here, we ignore the phases (F, D, E, W) of the instruction execution for simplicity. We demonstrate the execution of a two-way superscalar in-order processor. The goal here is to have an instruction schedule that allows for maximum parallelism.

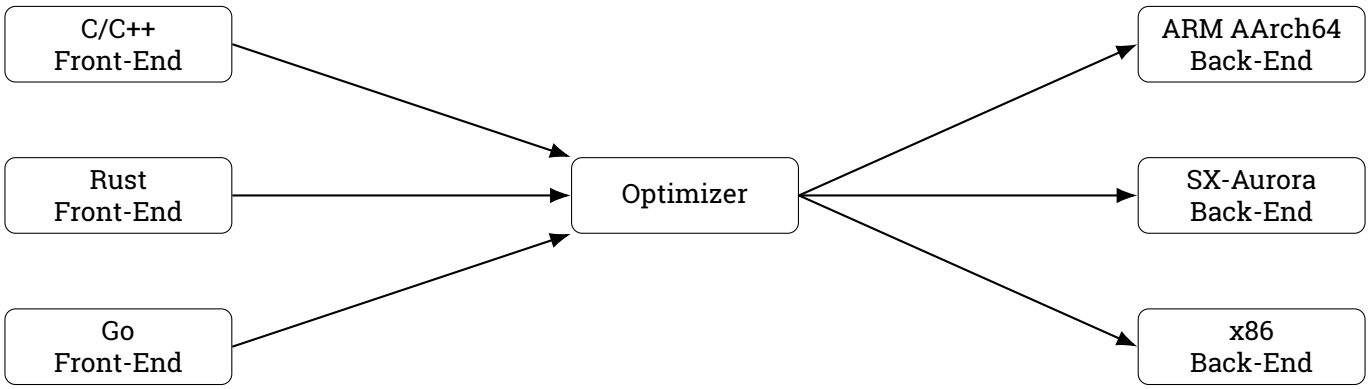


Figure 2.1.: Compiler Implementation: Compilers typically consist of a front-end, an optimizer, and a back-end phase. Exemplary, with three different source languages, and three different target architectures.

We achieve parallel execution in Table 2.3a for the instructions $\{i_1, i_2\}$ and $\{i_4, i_5\}$. We cannot schedule the instructions i_3 and i_4 together because i_4 has to wait for i_1 to finish, so that it can use its result.

In the execution in Table 2.3b, we can schedule the instructions $\{i_4, i_2\}$, $\{i_5, i_3\}$, and $\{i_6, i_7\}$ together in pairs of two. However, the overall execution is with 15 cycles slower than the 10 cycles of the execution in Table 2.3a because the CPU stalls more often.

Instruction scheduling is not a trivial task, and much research has been published on this problem. However, the more complicated CPUs become, the harder it gets to generate fast instruction schedules.

2.2. Compilers

A compiler is a tool that helps make programs, written in a high-level language, executable on a specific machine. The compilers' goal is to translate from a high-level language to a low-level language or machine code. During the translation, the compiler ensures that the output program is a correct representation of the input. Further, compilers typically aim to optimize the source program with a defined goal, e.g., runtime performance or memory usage. We refer to [56] for more detailed information.

Making a high-level language program executable includes more tasks than the translation, which is a non-trivial task. For example, other steps include linking the program with libraries, especially standard libraries, that implement basic functionalities. Next, it has to be prepared to interact with a runtime library. These topics are out of scope for this thesis, and we do not discuss them further.

The implementation of a compiler typically consists of three phases which apply modifications to the given code with a defined output format.

1. The compiler transforms from the high-level language to an Intermediate Representation (IR) in the **front-end phase**.
2. In the **optimization phase**, the compiler applies optimizations on the IR and outputs the code in the same format.
3. In the **back-end phase**, the compiler transforms the IR to assembly code or machine code.

Figure 2.1 visualizes this setup. We explain these three phases in more detail, especially the back-end phase, which we are working on in this thesis.

Basic Block

Compilers typically perform their transformations on a per basic block level. This includes the instruction scheduling in the back-end phase (see Section 2.2.3).

A basic block is a sequential collection of instructions. They have a single defined starting instruction and a single defined ending instruction. That means, once a basic block executes, all its instructions execute. Consequently, basic blocks cannot contain branches. However, function calls are allowed since they return to the next instruction after the call and continue their execution.

2.2.1. Front-End

The front-end phase is the first step in the translation process. Its implementation is dependent on the source language that has to be translated. A typical front-end includes a scanner, a syntax checker/parser, a context-sensitive analysis, and translation into an IR. The scanner translates a character stream into a token stream. The parser then takes these tokens and is checked against the grammar defined by the source language. Even with a syntactically correct program, there can still be errors in the code, e.g., assignments of incompatible types. These are checked during the context-sensitive analysis phase. Eventually, the source code is translated into an IR which is then used as input to the optimizer and the back-end.

The IR is compiler-dependent and typically defined as an Single Static Assignment (SSA) form. A SSA language, is a language in which each variable is assigned exactly once. When the value of a variable changes, the value is not written to that variable but to a new variable. This form simplifies many optimizations.

There might be additional steps required depending on the source language. C/C++ compilers, for example, use a preprocessor to replace macros like `#include` and `#define` with their actual values.

2.2.2. Optimizer

The usage of an IR abstracts away the source language and the target hardware and permits applying more standardized passes in the compilation process. These additional passes transform IR to IR. Note that this step is optional and not required to produce correct translations. This step aims to optimize the code for more efficient execution in a source- and target-independent manner. Efficient can mean different things here. The transformed code might produce, e.g., a faster program, a smaller program, or a program with less power consumption.

There are a large number of optimization passes in most compilers. They range from relatively simple optimizations, like replacing constant variables with their actual value, to more advanced ones that might, for example, simplify computations with the rules of algebra.

2.2.3. Back-end

The overall goal of the compiler's back-end is to transform the IR to code for the target hardware. Further, it also decides where the processor stores variables (registers only or memory). In the subsequent paragraphs, we explain the necessary steps in more detail.

Instruction Selection

First, the compiler translates the input IR representation. As the IR is already in SSA form, the translation step mostly maps IR instructions to instructions defined in the Instruction Set Architecture of the target hardware. The instruction selection is not a one-to-one mapping in all cases, but sometimes, IR instructions are represented by multiple hardware instructions.

Note that the compiler chooses actual processor instructions, but registers are not assigned yet. Variables are assigned to an unlimited number of virtual registers. This means that the compiler still works in an SSA form regarding the registers.

Instruction Scheduling

In the instruction scheduling phase, the compiler tries to generate an optimal ordering over the instructions of a basic block. Optimal in this context usually means to optimize for short runtimes.

We have seen how different schedules influence the runtime performance in Table 2.3. The objective of this thesis is to generate optimized instruction schedulers for novel hardware in a automized way. Here we explain how instruction scheduling typically works.

As stated in Section 2.1.4, different types of CPU pipelines are influenced by the instruction schedule. When we look at the superscalar pipelines (see Section 2.1.3), the out-of-order pipeline would benefit the least from better instruction schedules because it can reschedule the instructions themselves. The other two superscalar pipelines benefit more from better instruction schedules because they might stall completely until a hazard is cleared. However, all pipelined execution models might benefit from better instruction schedules.

Instruction scheduling is a challenging problem. One reason for that is that it is NP-complete to find an optimal schedule [23]. It is necessary to work with heuristics and generate approximations to optimal schedules. Additionally, processors get more and more optimized in special cases. That makes them faster, but it also becomes harder to predict what they would do in a given situation, i.e., to predict how long it would take to execute a given instruction. Optimizations that have an influence on this are hardware branch prediction, pipeline forwarding, hardware interlocks, or early exits. This are only a few examples. Therefore, the problem is not only NP-complete, but also has to be solved with unreliable data.

Like the transformations in the optimizer stage, the instruction scheduling is typically done on a basic block level. However, there is research on extending the scope of instruction scheduling (see Section 3.1).

Compilers typically use the list scheduling approach to find an approximately optimal solution. List scheduling is more of a framework than a specific algorithm.

The compiler typically transforms the sequence of instructions into a dependency graph to analyze a given basic block. The dependency graph defines which instructions need to be scheduled before a given instruction can be scheduled. The nodes in the dependency graph represent an instruction, and the edges represent

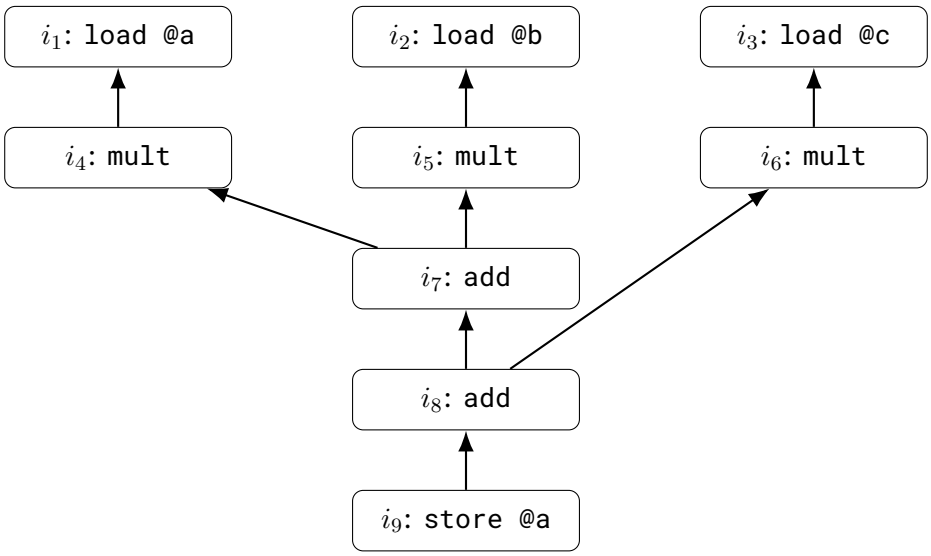


Figure 2.2.: Dependency graph for the example in Equation 2.1.

dependencies. The edges point from a given instruction to all the instructions on which it depends. Dependency graphs are modeled as Directed Acyclic Graphs (DAGs), and the valid schedules are defined as the set of the valid traversals for the given DAG.

In Figure 2.2 we see that before we can execute the `store` instruction i_9 to move the result back to memory, we have to run the last `add` instruction i_8. This `add` instruction i_8 depends on another `add` i_7 and a `mult` instruction i_6. The nodes that have no outgoing edge are leafs (i_1, i_2, and i_3) and can be scheduled directly in the beginning without any constraint.

Next, the compiler assigns priority scores to each node. Different scores could analyze the instruction runtime or the cumulated runtime from the instruction to the next leaf. A goal is often to reduce the register pressure. We explain this in the Register Allocation phase. Multiple scores can be assigned to break ties between instructions.

Eventually, the compiler collects the instructions that are ready to be scheduled. In the beginning, that are the leafs of the DAG. In the example in Figure 2.2 those are the instructions i_1, i_2, and i_3. It schedules one of these instructions. This process is repeated until all instructions are scheduled. If the compiler chooses to schedule instruction i_1 first, then the instructions available in the next round are i_2, i_3, and i_4. The described method is a top-down approach, but it is equally solvable in a bottom-up fashion.

Register Allocation

In the previous instruction scheduling phase, the compiler works with an infinite amount of virtual registers. The processors have a defined set of different classes of registers, e.g., there are often different registers for integer and floating-point numbers. In this phase, the compiler's goal is to map n virtual registers to m hardware registers, where m is a lot smaller than n.

Registers are the fastest memory type in computers to access data. It is often the only way to access data directly, i.e., data must be transferred from main memory to registers and vice versa by load and store instructions. Due to their speed, registers are also costly and thus available in a limited number.

ID	Instruction			
i_1	`load`	r_{arp}, @a	$\Rightarrow$	r_1
i_2	`load`	r_{arp}, @b	$\Rightarrow$	r_2
i_4	`mult`	r_1, r_1	$\Rightarrow$	r_1
i_5	`mult`	r_2, r_2	$\Rightarrow$	r_2
i_7	`add`	r_1, r_2	$\Rightarrow$	r_1
i_3	`load`	r_{arp}, @c	$\Rightarrow$	r_2
i_6	`mult`	r_2, r_2	$\Rightarrow$	r_2
i_8	`add`	r_1, r_2	$\Rightarrow$	r_1
i_9	`store`	r_1	$\Rightarrow$	r_{arp}, @a

Table 2.4.: Instruction schedule with reduced register pressure

The mapping to hardware registers is not always possible, because there might not be enough registers to store all required values simultaneously. In that case, memory spills are introduced. This means that the content of some registers has to be outsourced to memory. So, an additional store instruction to move the data to memory and an additional load instruction must be added to the code to get the data back into a register when it is needed. These spills are costly for the runtime performance, and the goal is to reduce them.

Like instruction scheduling, register allocation is also a challenging problem. It can be reduced to the graph coloring problem, which is NP-complete [34].

The runtime performance impact of the register allocation is also highly interdependent with the instruction scheduling [16]. Instruction schedules can be optimized for reducing the number of live variables. The instruction schedules in Table 2.1 both use the three registers r_1, r_2, and r_3. However, the instruction schedule in Table 2.4 only uses the two registers r_1, and r_2. The variable `b` is not used after instruction i_7, so its register r_2 can be reused for variable `c`. It does not mean that this is always faster, e.g., thus it is not always helpful to optimize for reduced register pressure when there are enough registers to prevent spills.

2.3. LLVM Compiler Infrastructure

LLVM [33] is an open-source framework for compiler technology. It started as a research project in 2000 and is nowadays widely used in compilers for various languages. It is most known as the back-end of the C/C++ compiler `clang`, but is also used for the languages Rust (`rustc`), Go (`gollvm`), and more.

The LLVM framework framework is easily extendable. This is especially true for new source languages and target hardwares. Additionally, details can be added to the compiler easily, like optimizations passes or instruction schedulers.

We choose to work with the LLVM framework because it gives us full control over the compilation process. It is easy to add new instruction schedulers and implement transformation passes for the LLVM IR.

The LLVM project was designed to make the three compiler phases executable in isolation. That means we can run LLVM front-ends like `clang`, or `rustc` to generate LLVM IR code. From there, the optimizer program `opt` can modify the LLVM IR code. This step is independent of the source language and the target hardware. Eventually, we can translate the LLVM IR into object or assembly code by calling the LLVM back-end `llc` and configure it to use our custom instruction scheduler.

2.3.1. Directed Acyclic Graph

As explained in Section 2.2.3, LLVM also uses a DAG representation for basic blocks internally. The C++ class `SelectionDAG` defines the LLVM DAG. LLVM comes with functionality to plot their internal DAGs. Figure 2.3 shows an example.

The arrows represent the dependencies between instructions. They point from a given instruction to one of its dependency instructions. There are three different types of edges that represent different types of dependencies:

- Black: Represents a data flow dependency. That means that an instruction consumes the result of the previous instruction.
- Dashed Blue: This edge ensures that two unrelated instructions in terms of data flow remain scheduled in the correct order. In Figure 2.3, the conditional jump instruction `t24` must be scheduled before the unconditional jump instruction `t14`, even though there is no data-flow dependency between these two.
- Red: This edge represents two instructions that must be scheduled together without any instructions between them.

The nodes in the LLVM DAG represent instructions. Outgoing dependencies are numbered and placed in the top row. If no dependencies exist, this row is missing. The instruction name is written in the next row, which is the top row if there are no outgoing dependency edges. Next, the row below shows an internally used instruction ID. The instructions output type is placed in the bottom row. There are three possible values for the output type:

- The datatype of the instructions result, e.g., `i32`, `i64`, `f32`, `f64`, or `v2f32`.
- `ch`, which represents no data but is placed together with the dashed blue dependencies.
- `glue` also does not represent any data but is placed together with the red dependencies.

Even though the instruction selection has already finished, there can still be LLVM pseudo instructions, which are resolved in later stages. The LLVM DAG always contains an `EntryToken` pseudo instruction which gets always scheduled first. Additionally, there is the `GraphRoot` node at the bottom, which does not represent any instruction and does not get scheduled. The `BasicBlock` pseudo instructions represent the `EntryToken` of another basic block.

2.3.2. Instruction Scheduling

The instruction scheduling happens in two phases in LLVM. There is one instruction scheduling phase before register allocation (pre-RA) and one after register allocation (post-RA). LLVM makes the main scheduling decisions in the pre-RA phase. The post-RA phase is only there to eliminate some small inefficiencies that can be detected, once the hardware registers are assigned. When we speak of instruction scheduling in the context of LLVM, we mean the pre-RA phase from here on.

The maintainers of the LLVM framework are working on replacing the current pre-RA scheduling classes. The new schedulers use the `MIScheduler` class as their base. Initially, instruction scheduling was integrated in the code of the instruction selection phase and worked on the `ScheduleDAG` class. We decided to base our work on the older implementation as it is still wider spread, and most production-used instruction schedulers are implemented in this manner.

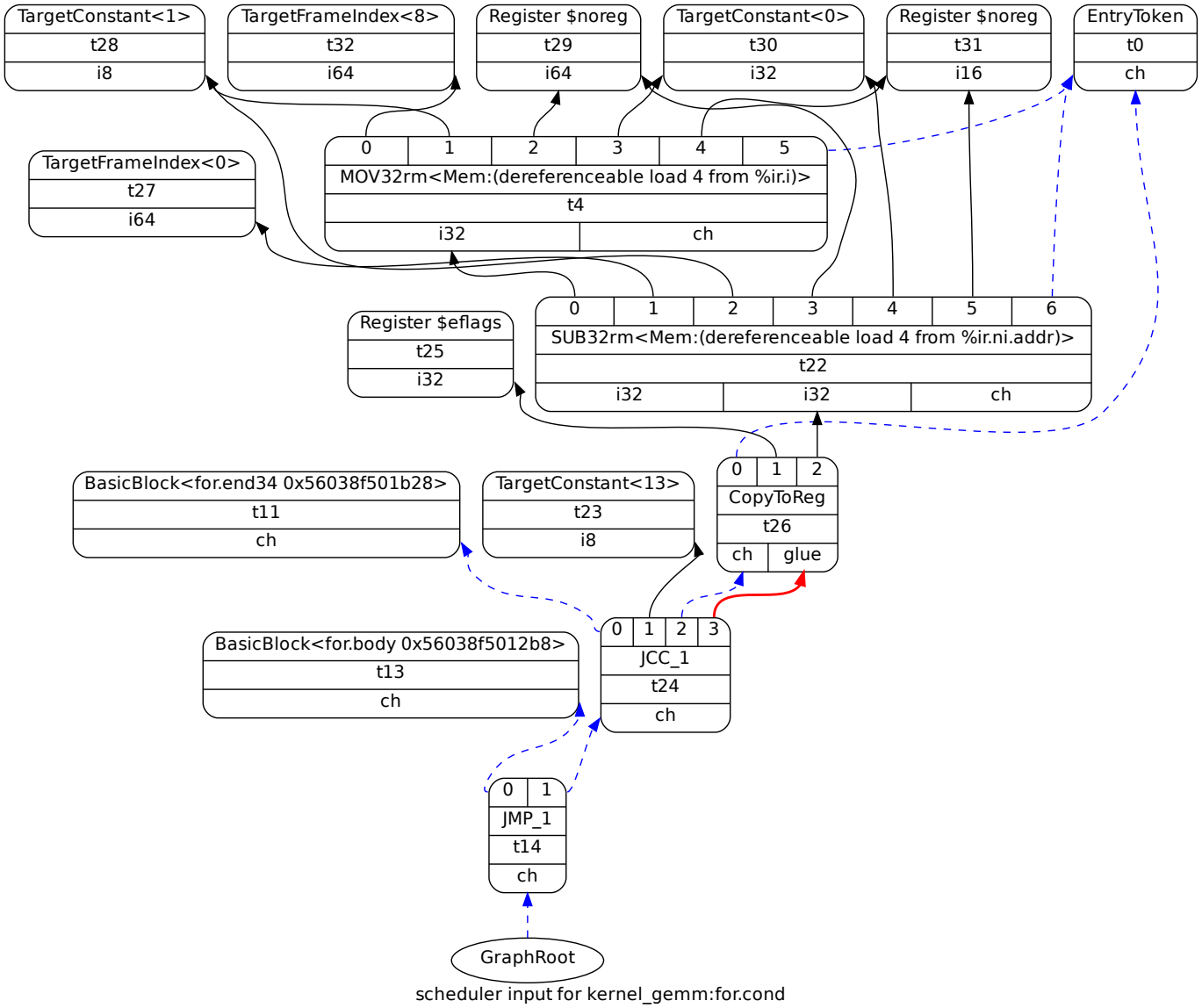


Figure 2.3.: Example DAG generated by LLVM. It shows the (AArch64) instructions defined by LLVM and the dependencies between the instructions.

There are already different instruction scheduler implementations that come with LLVM. These can be used by configuring the `llc` back-end with the command line option `--pre-RA-sched=` and one of the possible choices:

- `vliw-td`: Scheduler for VLIW processors
- `list-ilp`: List scheduler for balancing instruction-level parallelism and register pressure
- `list-hybrid`: List scheduler for balancing latency and register pressure
- `list-burr`: List scheduler for reducing register usage
- `source`: Similar to `list-burr`, but prefers the ordering from LLVM IR
- `linearize`: Does not schedule but returns instructions as they come in from LLVM IR
- `fast`: For compile-time optimized suboptimal scheduler
- `default`: The best scheduler for the used target hardware

We use the `default` scheduler as our baseline in all experiments.

2.4. LLVM Test Suite

In the LLVM project, there is a collection of publicly available benchmark programs, called LLVM Test Suite [1]. These are used to test the performance and correctness of the LLVM compiler. The test suite contains benchmarks from the following categories

- Single-Source — built from a single C/C++ source file
- Multi-Source — built from multiple C/C++ source files
- Micro-Benchmarks — separate functions that are executed by google-benchmark
- Bitcode — tests that are written in LLVM bitcode
- CTMark — symbolic links to the other benchmarks to measure compilation performance

We select the 273 benchmarks from the Single-Source and Multi-Source categories as a basis for our experiments. The selected categories contain benchmarks that are compiled from a single source file or multiple source files. The benchmarks are all written in C or C++.

2.5. Data-Driven Methods

Throughout this thesis, we utilize data-driven methods for different tasks. We use Monte Carlo Tree Search for finding good instruction schedules and to build a dataset. We describe this method in Section 2.5.1. For learning how to generate good schedules for unseen basic blocks, we use Support Vector Machines and Neural Networks, which we explain in Section 2.5.2 and Section 2.5.3.

[1] `https://llvm.org/docs/TestSuiteGuide.html`

2.5.1. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [1] is a randomized search method for optimal decisions. When, e.g., a computer's task is to play a board game, it has to make a decision for each move, and from the outcome of previous decisions, it has to make new decisions. This process continues until the game is over. From the beginning of the game until the end, we can model the decision-making process as a decision tree. For small games, like Tic Tac Toe, it is possible to use the trivial Minimax algorithm [46], which checks all possible paths in the decision tree. However, for games like Chess, there are too many possible paths for a computer to simulate. Another problem might be that the simulation itself is costly. This process is too slow for the number of possible instruction schedules that exist for a typical basic block.

MCTS works by exploring one level of decisions and finishing the path with random decisions. When all decisions in a level were seen at least once, the algorithm balances exploitation and exploration, which means it chooses the best possible decision most of the time. However, sometimes it chooses another decision to learn more in an undiscovered part of the decision tree. This balancing causes MCTS to learn how to extend the paths with better decisions.

We explain the algorithm in more detail here. The MCTS algorithm consists of four steps. These are:

1. Selection Phase: The algorithm starts from the root node of the MCTS tree and iterates down through the nodes by applying a child selection policy. It stops when it finds a node that has children, but not all children where visited yet. These nodes are called expandable nodes.
2. Expansion Phase: The MCTS tree gets expanded by adding the available children to the selected node from the previous step. The new children are created with a view count of zero.
3. Simulation Phase: From the selected node, the algorithm fills the path through the MCTS tree with randomly chosen nodes until a terminal state is reached. In this phase, no nodes are added to the tree. The randomly selected nodes are only used to fill the path. This path through the MCTS tree gets evaluated, e.g., by playing the board game with the decisions or in our case by checking the runtime of the instruction schedule.
4. Backpropagation Phase: The evaluation of that path gets applied to the statistics of all the path's nodes.

The node statistics are typically just two values. First, the count of how many times a node was included in a simulation (view count). The second value is the cumulated evaluation metric. Typically, cumulated means the average value. However, in our case, we observe that it is beneficial to use the maximum value instead of the average. This was also mentioned and generalized for single-player games in [7].

The child selection policy, mentioned in the selection phase, is a strategy to find a path through the MCTS tree. There are two possible extreme ways. The first is to choose always the nodes with the lowest view count to find out more about nodes that we do not know much about. This is known as exploration. The opposite thing to do is always follow the node with the best metric, which is called exploitation. These two aspects have to be balanced. There are many ways to balance exploitation versus exploration, and it is a field of active research. The standard policy for balancing are Upper Confidence Bounds for Trees (UCT) [29, 30].

2.5.2. Support Vector Regression

Support Vector Machine (SVM) [10] is a robust classification algorithm. There is also an extension for regression problems, which is called Support Vector Regression (SVR) [13]. We use the SVR model, but we first briefly explain the classification model, as the regression model is based on this.

Originally, SVM is a model that gets trained to separate data points into two classes. This model is then able to transfer what it has learned to new, unseen data points. SVM does this by creating a hyperplane that separates the two classes of data points. The goal is to maximize the margin between the hyperplane and all the datapoints, such that it separates the classes. However, data is often not easily separable. Therefore, SVM allows some data points inside of the margin area.

The SVR algorithm also tries to fit a hyperplane. However, SVR tries to have as many data points as possible inside of the margin area. This means that the hyperplane is located close to the data points. Then, this hyperplane is used as the modeled function, not as a separation.

If the data points are not easily separable, the non-linear extension to the SVM, the Kernel-SVM, might help. By applying the kernel-trick, the datapoints are mapped into a higher dimensional space. This makes them easier separable. The non-linearity might also help the SVR algorithm for learning non-linear functions.

2.5.3. Neural Networks

Much development in the machine learning field is based on artificial neural networks since the 2010s. Especially, the computer vision and natural language processing fields profited from neural networks. There are special types of neural networks for working on images which are called Convolutional Neural Netoworks (CNN) [35]. The weight that these networks learn are the weights of convolution operations. Successful implementations of CNNs are, e.g., [31, 21] on a image classification task. Another important type of neural networks are Recurrent Neural Networks [49]. These are often used in Natural Language Processing (NLP) problems, e.g., [18, 53]. However, in this thesis, we use standard neural networks.

The foundations of this field date back to the 1950s [48]. With increasing computation power, more and more layers were added to the neural networks. Machine learning with deep neural networks is often referred to as deep learning.

Neural networks are organized in layers of learned weights, which are called neurons. Each layer function is wrapped by a non-linear activation function a. This results in the mathematical formulation of a layer as

$$O(X) = a(WX + b) \tag{2.2}$$

with the neurons W, the bias b, and the input vector X. The input can be either the input data if this is the first layer or the output of the previous layer in the neural network. Figure 2.4 visualizes an example neural network.

To optimize neural networks, i.e., learn the weights W, usually a gradient descent approach is used. An error function is defined, which measures the distance between the prediction and the expected output. Gradient descent adjusts the weights W step by step in a direction that minimizes the error.

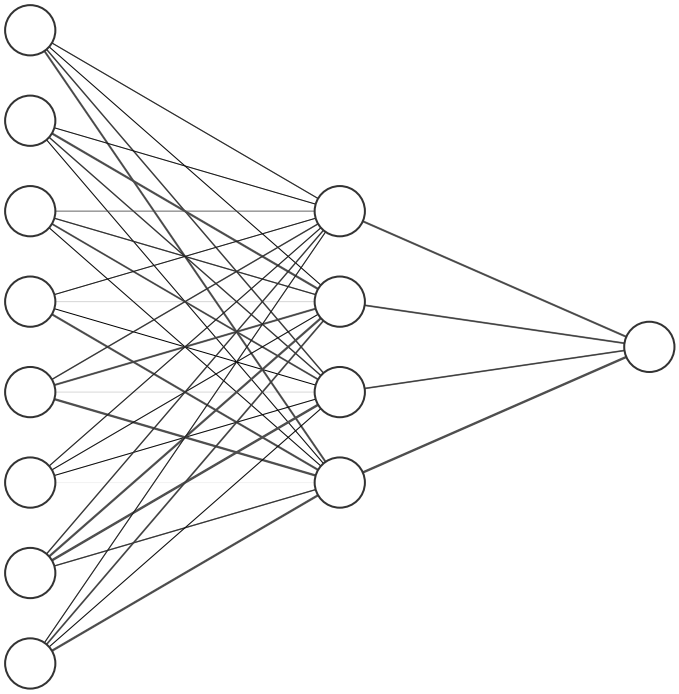


Figure 2.4.: Example Neural Network with three layers. The nodes represent input/intermediate values. The edge opacity represents the value of the corresponding weights.

3. Related Work

In this chapter, we survey the existing relevant literature covering this thesis' topics. For instruction scheduling (Section 3.1) and register allocation (Section 3.2), we first discuss classical approaches and continue with newer data-driven advancements. Further, we discuss some other relevant works in the fields of machine learning-based compiler optimizations, runtime estimation, code feature extraction, and machine learning approaches on other scheduling tasks (Section 3.3). These are interesting because we use methods like this in our approach or they are closely related.

3.1. Instruction Scheduling

3.1.1. Classical Approaches

Scheduling problems appear in many fields where tasks with dependencies and a cost need to be ordered for execution. Therefore, general scheduling is a topic with much existing research. Also, the research on instruction scheduling has a long history.

Some algorithms can generate perfect instruction schedules for simple situations with perfect information. Architectures with only one functional unit and uniform instruction latencies fulfill the requirements. The best-known algorithms in this field are the Sethi-Ullman labeling algorithm [52] and the work by Proebsting and Fischer [47].

However, these conditions are not present in modern processors, as discussed in Section 2.1. In more complex situations, the instruction scheduling problem is NP-complete [23]. Modern processors use multiple pipelines to achieve instruction parallelism, see Section 2.1. Consequently, most instruction schedulers used nowadays are based on the list scheduling framework, which Landskov et al. [32] proposed. The algorithms that follow this approach are better able to generate instruction schedules for pipelined processors. Heller [22] published early work on how to approach instruction scheduling for these processors. Further research on developments of list scheduling was published in [6, 15, 23].

As elaborated in Section 2.1, the available information on instruction latencies is often unreliable due to instruction-level parallelism and unpredictable memory latencies caused by cache hierarchies. One way to approach this problem is balanced scheduling [27, 37]. Another proposed method was to use stochastic instruction scheduling [51].

Instruction scheduling typically works on a basic block level. This means that instruction schedulers cannot schedule transitions across basic blocks. However, there is research on extending the scope to larger regions. Fisher [14] selected code paths in functions for instruction scheduling. Bernstein and Rodeh [6] define regions of strongly connected code (e.g., loops) and execute scheduling on this unit. Superblock scheduling was

proposed by Hwu et al. [25]. A superblock consists of multiple consecutive basic blocks and thus can start execution from other instructions than the first one.

3.1.2. Data-Driven Approaches

List schedulers usually have multiple heuristics used to choose instructions from the list of available instructions. The selection is based on a weighted sum of the heuristics. The first work that combined data-driven methods with instruction scheduling is a patent by Tarsy and Woodard [55], filed in 1994. They optimize weights used in cost-based heuristics. These heuristics are used in list scheduling for pipelined processors.

Similarly, Beaty, Colcord, and Sweany [4] published a work in 1996 in which they used a genetic algorithm to learn weights for different heuristics. They achieved a 5% performance increase compared to a random scheduler on three architectures.

Moss et al. [44] trained a function that picks one instruction over another when presented the previously scheduled instructions. They use decision trees, look-up tables, ELF function approximations, and feed-forward neural networks. The decision tree performs best and often finds the optimal schedule. However, they only use simulations and limit the basic block length to ten instructions.

McGovern, Moss, and Barto [41, 40] propose a reinforcement learning and a search heuristic. Their reinforcement learning heuristic sometimes found a better instruction schedule than their baseline. The long-running search approach found a better instruction schedule every time. However, their baseline was only a random instruction scheduler, and they have only used simulation results instead of using a state-of-the-art compiler and executing their benchmarks on hardware.

Russell [50] uses decision trees to create heuristics for improving instruction scheduling decisions. They show that they generate better instruction schedules 7.8 times more often than the compared heuristics. They also evaluate their work only on a simulator.

A newer work in this field was published by Jain and Amarasinghe [26]. They train a neural network to imitate the instruction schedules by the GCC compiler. However, the performance of the GCC instruction scheduler limits this approach's performance.

We conclude that machine learning-based approaches mostly performed well in theory but only against weak random baselines. The only work that we found that was actually evaluated on hardware was [4].

3.2. Register Allocation

We have discussed the implications of the instruction scheduling phase on register allocation in Section 2.2.3. Their interdependence was also shown by Goodman and Hsu [16]. Lavrov [34] showed the connection between the graph-coloring problem and register allocation and thus, the NP-completeness. The first graph-coloring-based algorithm was implemented in a compiler by Chaitin [9].

In the field of register allocation, also appeared research that builds the connection to data-driven methods. Das, Ahmad, and Venkataramanan [12] use a deep learning approach to solve the graph coloring problem. The newer and naturally better fitting approach with graph neural networks was used by Lemos et al. [36] to solve the graph coloring problem.

3.3. Related Areas

3.3.1. Compiler Optimizations with Machine Learning

Machine learning approaches are also applied to optimize other parts of the compilation process. Mammadli, Jannesari, and Wolf [38] and Huang et al. [24] successfully apply deep reinforcement learning to the phase-ordering problem. Phase-ordering means to select the compiler's optimization passes and define its execution order (see Section 2.2.2 for information on the optimization phase). Deep reinforcement learning was also used by Haj-Ali et al. [19] to translate loops into vector processing instructions (SIMD). Wang and O'Boyle [57] used machine learning to predict the optimal number of threads and the optimal scheduling policy for OpenMP parallelized loops. We refer to the surveys [58, 3] for more literature.

3.3.2. Runtime Estimation

There are various tools for throughput and runtime estimation, like Ithemal [42], llvm-mca[1], and Intel Architecture Code Analyzer (IACA)[2]. However, the listed tools only work with the x86 architecture, which we do not use. The research projects and open-source estimators might be extended to other hardware architectures. Especially the Ithemal [42] project is interesting as they use a neural network to predict the runtime from the basic block. That means they learned to extract relevant features from the basic blocks instructions.

3.3.3. Feature Extraction from Code

The previously cited works on data-driven machine learning optimizations have or might benefit from research whose goal is to extract features from code automatically. A similar approach to the word2vec [43] approach in the NLP area was proposed by [5, 2]. Cummins et al. [11] developed a method to extract features from code based on graph structures. The work proposed by Brauckmann et al. [8] works similarly – they also work on graph structures and use graph neural networks to extract features.

3.3.4. Other Scheduling Tasks with Machine Learning

Mao et al. [39] have used a deep reinforcement learning approach to schedule data-processing jobs onto computing clusters. This work is interesting because the jobs have dependencies on each other, represented in a DAG, just like the instructions in the instruction scheduling problem.

[1] `https://llvm.org/docs/CommandGuide/llvm-mca.html`
[2] `https://software.intel.com/content/www/us/en/develop/articles/intel-architecture-code-analyzer.html`

4. Approach

In the remainder of this chapter, we present the approach used for this thesis (compare Figure 4.1). We start with explaining the necessary modifications to the build steps in Section 4.1. Then we discuss the importance of basic blocks for instruction scheduling in general and our approach specifically (Section 4.2). In Section 4.3, we present a feasability study to validate the potential of our approach. That is followed by explaining the MCTS approach (Section 4.4.1), which is used to find well performing instruction schedules. We introduce our approach to train inference models (Section 4.4.2) from the many MCTS models. Finally, we discuss how we transform source code into data in a format that we can use for our data-driven approaches in Section 4.5.

4.1. Build Process

This section describes the process that we use for building the programs in the LLVM Test Suite (see Section 2.4). First, we divide the compilation into separate steps to gain full controll over the settings. We explain this in Section 4.1.1. Next, Section 4.1.2 explains the LLVM passes that we implement to add functionality to the LLVM compiler.

4.1.1. Divided Process

The test suite comes with Makefiles and CMake configurations to automatically build all benchmarks. However, these are not sufficient for our approach. We need a modified build process for all the benchmarks because we are manipulating it in several cases. Furthermore, the compilation cannot be done by a single command because not all arguments of the separate compilation steps are available in the compilers front-end. We execute the front-end, optimization, and back-end compiler phases separately to have complete control over the build process.

We, further, analyze the CMake configurations of the programs in the LLVM test suite and extract required compiler arguments on a per benchmark level. We categorize the extracted arguments into front-end, optimizer, and back-end phases. Finally, we append them to the previously described arguments in the respective compilation phase.

Some programs read data from files or require arguments when calling them. We analyze the test suite for these execution arguments. To execute the test suite programs, we generate a shell script that calls the compiled benchmark with the required arguments.

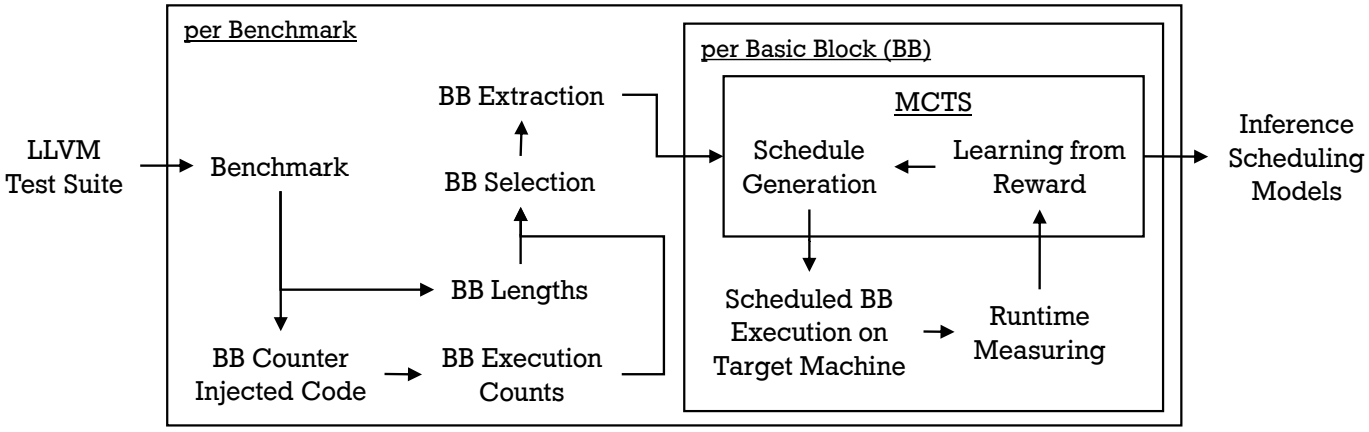


Figure 4.1.: Overview over the complete approach: First, we evaluate the basic blocks in the programs of the LLVM Test Suite (Section 2.4) with our heuristics, described in Section 4.2. Then, we extract the selected basic blocks and normalize them for isolated execution (Section 4.5.4). We train a MCTS model for each basic block (Section 4.4.1), which includes measuring the runtime (Section 4.5.2). Finally, we use the MCTS models to train various inference models (Section 4.4.2).

Front-End

We have to set some compiler arguments to make the LLVM compiler front-end output LLVM IR instead of compiling the program to an executable format. We use the compiler front-ends `clang` [1] for C programs and `clang++`[1] for C++ programs, both in version 12.0.0. The compiler arguments are the same for both front-ends. We pass the following arguments to the front-ends:

- `-O0`: To prevent optimization in this first step.
- `-Xclang -disable-O0-optnone`: The `-O0` flag alone also prevents optimization in later compilation stages. These flags still allow later optimizations.
- `-S -emit-llvm`: To emit LLVM IR for further processing — instead of creating the program's binary.

Optimizer

The LLVM optimizer `opt` [2] gives a choice to select one of the predefined optimization levels or select specific optimizer runs. When we compile a benchmark for measuring the runtime, we only use the flag `-O3` for optimizing the LLVM IR. This optimizes the code only on the IR level, and does not interfere with the instruction scheduling. We choose to use optimization in this step because it would usually be used in a production setting, too. This makes the input code to our instruction scheduler more realistic to real-world situations.

We implement various LLVM passes that we explain in detail in Section 4.1.2. The optimizer argument `-passes=<pass-names>` activates these passes.

[1] `https://clang.llvm.org/`
[2] `https://llvm.org/docs/CommandGuide/opt.html`

Back-End

The LLVM back-end compiler implements one instruction scheduling phase before the register allocation and one after the register allocation. The first instruction scheduling phase is the one that we are interested in. This first instruction scheduling phase was re-implemented in the LLVM project in the last years. Both versions do currently work. We decide to base our work on the old implementation because, the most target architectures still use it.

We manipulate the back-end by passing arguments to the LLVM static compiler `llc` [3]. This program represents the back-end phase and executes the instruction scheduling. We use the following arguments for the back-end:

- `--pre-RA-sched=`: To select our new instruction schedulers for the pre-register-allocation phase.
- `--misched-cutoff=0`: To disable the new pre-register-allocation scheduler implementation.
- `--disable-post-ra`: To disable the post-register-allocation scheduler, because we focus on optimizing the pre-register-allocation phase and do not want the post-register-allocation scheduler to manipulate our schedules. Thereby, we create a controlled environment.

4.1.2. LLVM Compiler Passes

The LLVM compiler framework makes it easy to implement compiler passes, which are functions that are executed during compilation and can modify the IR code. We implemented three of these functions: to count the number of executions of basic blocks, to count the basic block lengths, and to measure the runtime of a given function. To make these passes usable, we compile them together with the source code of the LLVM compiler framework.

Basic Block Execution Counting Pass

We implement an LLVM optimizer pass for injecting counters into a program that count the number of executions for each basic block. Optimizer passes take LLVM IR files as input and output the code in the same file format. Our pass injects one global 64-bit counter variable for each basic block into the code. Next, the pass adds instructions into the beginning of each basic block, which increases the corresponding counter by one. At the end of the main function, the pass adds a format string and injects a call to the `printf` function to print all the counter values and corresponding basic block names.

We include the optimizer pass into our build process (see Section 4.1). To actually get the execution counts, we build a program with this optimizer pass and execute it. The benchmark then prints the counter values.

Basic Block Instruction Count Pass

In order to get the number of instructions for all the basic blocks in a program we implement another pass. This pass just prints the name of the basic block and its number of instructions to the command line during compile time. It is a pure analysis pass and does not modify the IR code.

[3] `https://llvm.org/docs/CommandGuide/llc.html`

Function Runtime Measure Pass

Lastly, we implement a pass, to measure the runtime of a single function. The pass searches for the function that contains the selected basic block. Then, it injects calls to the timer functions of the C++ standard library (`std::high_resolution_clock::now`). The calls are injected at the beginning of the given function and right before the return statement. Additionally, it injects a compilation-unit-wide global variable and stores the measurements into this variable. In the destructor of that compilation-unit, the pass injects code to print all the measurements.

4.1.3. Instruction Scheduler Integration

Since the LLVM compiler framework is designed to be easily extendable, it is not complicated to implement new instruction schedulers in C++. However, these instruction schedulers must be compiled with the whole framework. Additionally, due to C++'s complexity, it is not the best language for making quick experiments.

Therefore, we implement an LLVM instruction scheduler in C++ that calls an external command for scheduling selected basic blocks and and fallsback to another instruction scheduler for other basic blocks. We define an interface to exchange scheduling information and write the information about the DAG's structure and instructions to a text file. Then our C++ scheduler calls an external shell command. The executed command is responsible to generate another text file with the scheduled basic block. Our C++ scheduler then parses the schedule and and schedules it. We rely on LLVM's internal methods to ensure that the generated schedules are legal. Our actual instruction schedulers that, e.g., interact with the MCTS model are implemented in Python.

4.2. Basic Block Selection Heuristics

Even though there is research on instruction scheduling on more extensive units (see Section 3.1), most compilers schedule instructions on a per basic block level. This is also true for the LLVM compiler framework. The advantage of this is that the limited scope avoids complicated situations and thus simplifies the scheduling problem. For example, we do not have to deal with unclear execution paths due to conditional jump instructions.

Consequently, we also choose the basic block as the unit for instruction scheduling. That means that we execute our experiments on individual basic blocks and thus, learn to schedule individual basic blocks. Later, we combine the MCTS models to generalize the knowledge with our inference models to apply them to unseen basic blocks. We describe this process in detail in Section 4.4.

Not all basic blocks provide an equal value to our experiments. It also consumes too much time to work on all the basic blocks within the scope of this thesis (e.g., compilation time, execution time, machine learning training time). Therefore, we sort our basic blocks by heuristics. From this list, we select as many as we need and can handle in our experiments. We design three heuristics which we explain in the following paragraphs. See Table 4.1 for an example basic block with these heuristics.

Function	Basic Block	Length	# Executions	Product
kernel_floyd_warshall_StrictFP	for.body6	23	536,870,912	12,348,030,976
kernel_floyd_warshall	for.body6	23	536,870,912	12,348,030,976
print_element	entry	25	1,048,576	26,214,400
kernel_floyd_warshall_StrictFP	for.cond4.preheader	17	1,048,576	17,825,792
...	...	...	...	...
xmalloc	entry	8	2	16
main	entry	11	1	11

Table 4.1.: The tables shows an excerpt of the basic block heuristics (see Section 4.2) for the floyd-warshall benchmark. The basic blocks are sorted by the last heuristic, which is the product of the length and execution count. From this list the table shows the top four and last two basic blocks in the benchmark.

4.2.1. Longest Basic Blocks

There are multiple reasons why we are more interested in longer basic blocks than in shorter basic blocks. All these reasons summarize to the fact that there are more optimization possibilities with longer basic blocks. We better exploit pipeline effects with longer basic blocks, because more instructions can potentially be executed interleaved (see Section 2.1). With more instructions, there is potentially also a higher variety of instructions, which means that we can utilize more execution units on the processor in parallel. Lastly, with short basic blocks, we only make few scheduling decisions per basic block. Summarized, this means that optimizations have a bigger effect on long basic blocks than on short basic blocks. This was also shown by Stefanovic [54].

We use the optimizer pass described in Section 4.1.2 to count the number of instructions per basic block. From the output of that pass, we put together a list with all basic blocks and their lengths (see Table 4.1).

4.2.2. Most Executed Basic Blocks

Some basic blocks are significantly more often executed than others. For example, basic blocks in the bodies of loops run often. A basic block in the error checking code in the beginning of a program might only execute once. Often executed basic blocks influence the overall runtime a lot. Hence, we are more interested in optimizing these. We count the number of executions with the LLVM optimizer pass described in Section 4.1.2.

4.2.3. Most Executed and Longest Basic Blocks

The previous two heuristics are helpful on their own, but we are most interested in basic blocks that combine the two aspects. For example, the most executed basic block can still come from a very simple loop with few instructions in the loop body. We compute the product of the two heuristics to get a combined heuristic.

4.3. Feasibility Study

Before starting to optimize a process, it is useful to determine whether there is potential for any optimizations. Therefore, we show, that different instruction schedules can indeed lead to different runtimes on our target hardware (see Section 5.1). This early experiment is also helpful to get an overview of how much the runtime can vary between different instruction schedules.

Section 2.4 describes the selection of benchmarks from the LLVM Test Suite. For this experiment, we select one basic block per benchmark, whose instruction schedule we modify. For the selection of basic blocks we use the heuristic that balances between the most executions and the longest basic blocks, which is discussed in Section 4.2.3.

We execute this experiment in an early stage and did not have a basic block extraction pipeline nor means to measure the execution time of a single basic block. Therefore, we execute the whole benchmark with the modified instruction schedule of a single basic block. However, measuring the runtime of the whole benchmark includes the execution of much overhead code that we are not interested in. Thus, we instead measure the runtime of the function that contains the basic block of interest. This corresponds to the third method in Figure 4.6.

The simplest approach to achieve this experiments goal is to use random instruction schedules. Our random instruction scheduler works on top of a basic list scheduler. This means, out of the issue-ready instructions provided by the list scheduler, our scheduler chooses one at random. This is repeated until no more instructions are left. We fix the seed of the random number generator for reproducibility and generate instruction schedules with the seeds 0-10 for each selected basic block. The basic block of interest might execute multiple times in the measured function for reasons discussed in Section 4.5.1. We choose the shortest measured runtime per benchmark run to ensure that we use the same execution path in our measurements. To check that the runtime measurements are reproducible, we run each generated instruction schedule two times.

To carry out this experiment, we choose a processor with the architecture AArch64 and the NEC SX-Aurora TSUBASA. The choice is explained in Section 5.1. Figure 4.2 shows a representative selection of experimental results for the AArch64 processor. The plots show the runtimes grouped by the different seeds for the random instruction scheduler. Runtimes that differ between two runs more than 5% are marked as outliers and plotted in gray. Figures 4.2a to 4.2d show examples where different runtimes are clearly observable for different instruction schedules. However, this is not always observable. Figure 4.2e and Figure 4.2f show examples where no difference in the runtime was observable. In summary, we see that different instruction schedules can generate measurable differences in the runtime. This means that there is potential for improvements.

Figure 4.3 shows a similar selection for the same experiment on the SX-Aurora processor. We can observe a similar outcome of the experiment. However, as this processor cannot be interrupted by the Operating System (OS), the runtimes are more stable between two runs. In fact, no measurements in the whole example where marked as outliers. In summary, we observe potential for optimizations on this processor, as well.

Multiple reasons could lead to the small or missing variations that we observe in Figures 4.2e to 4.2f and Figures 4.3e to 4.3f: One possibility is that different instruction schedules have no effect on the runtime of the basic block. The motivation for this experiment was to verify, that these cases do not dominate the instruction schedules. Other possibilities have their origin in the experimental setup, these could be:

- The basic block for which we manipulate the instruction schedule might have a low influence on the runtime of the function. We tried to minimize this effect by choosing basic blocks that are often executed.

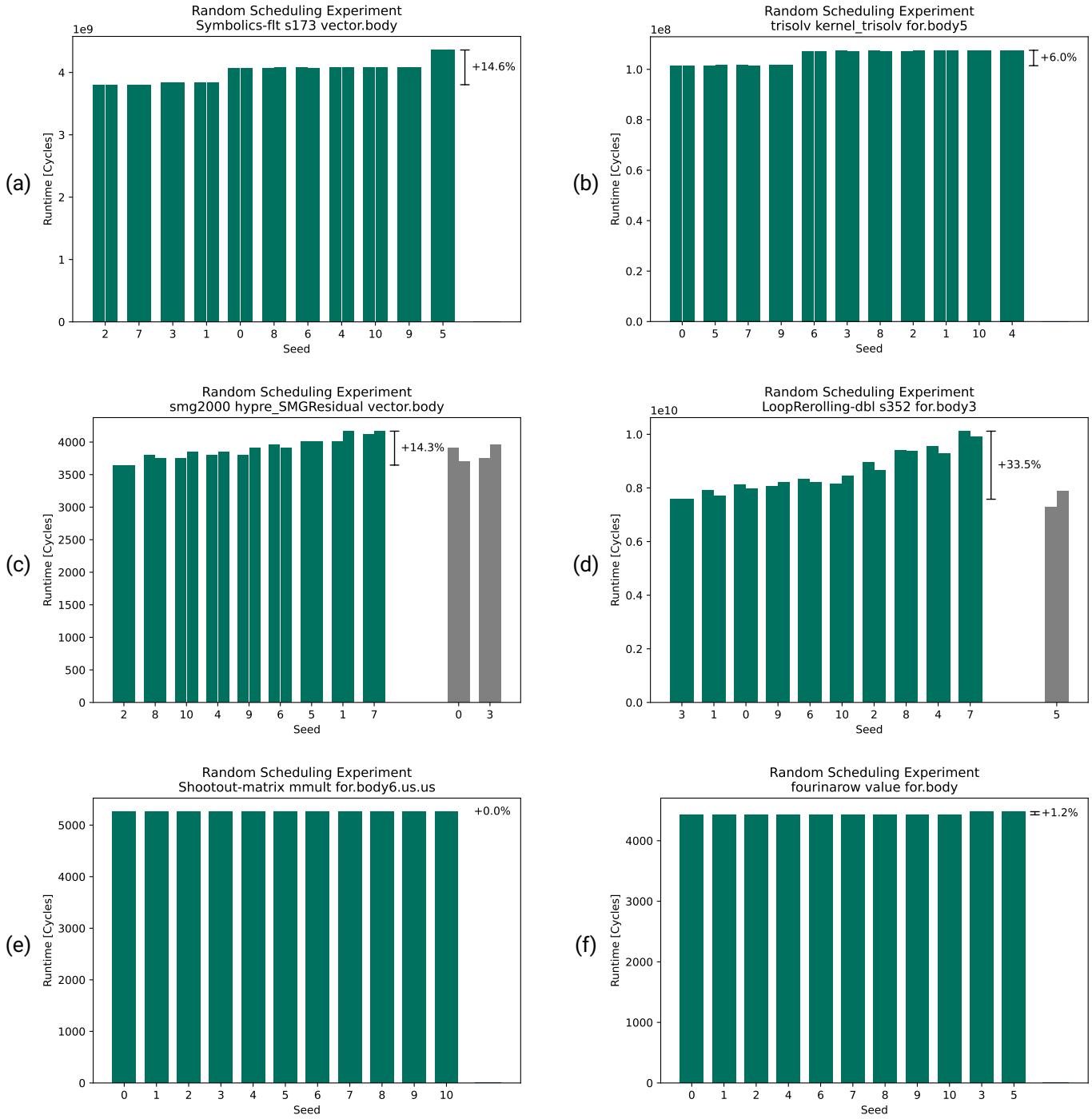


Figure 4.2.: Random Scheduling Experiment on AArch64: The bars show the runtime of a function with a random instruction schedule. The two runs of the instruction schedule are grouped together. The distance measure represents the longest execution time relative to the shortest execution time. Two runs that differ more than 5% are marked as outliers and plotted in gray.

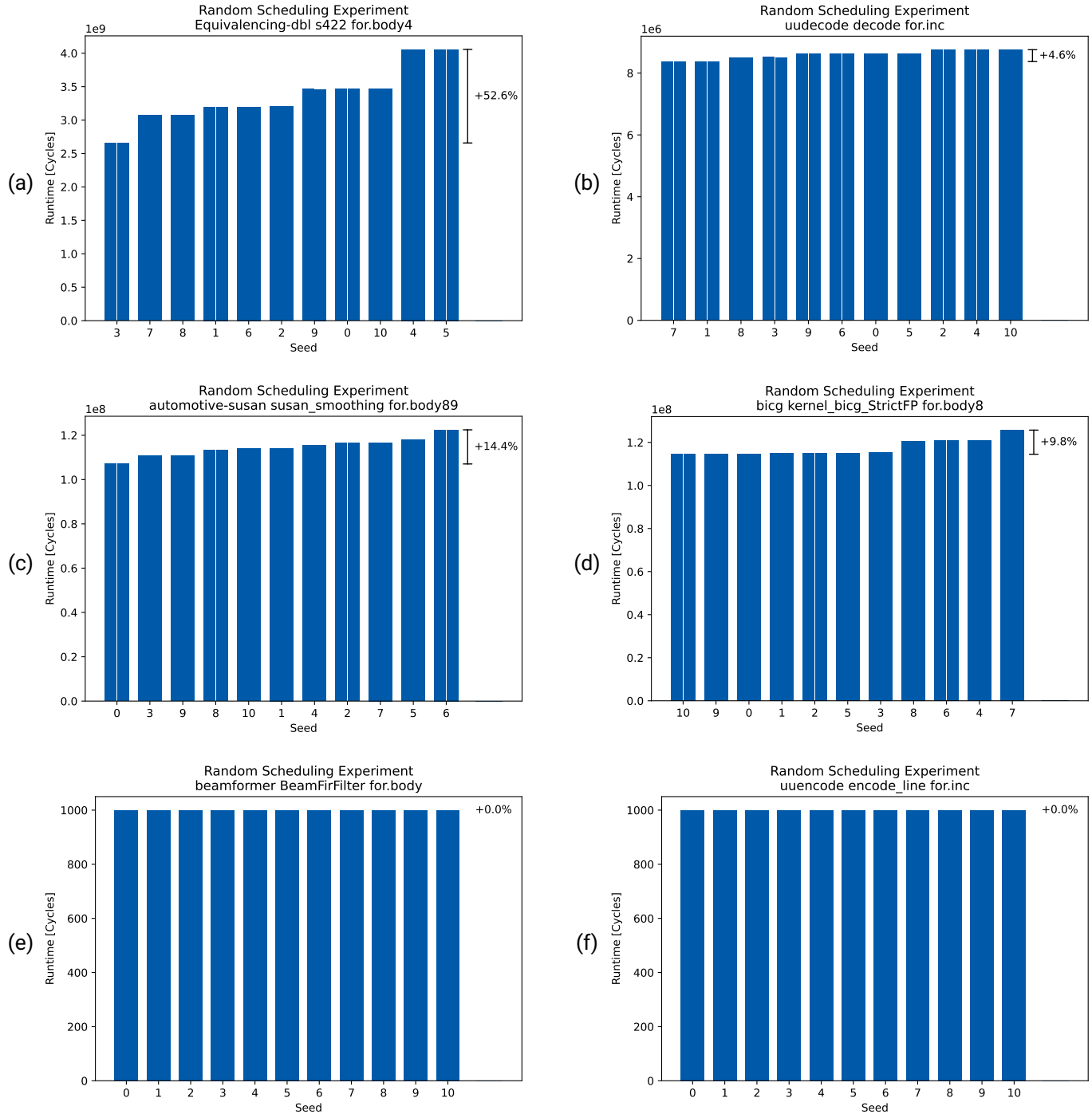


Figure 4.3.: Random Scheduling Experiment on NEC SX-Aurora TSUBASA: The bars show the runtime of a function with a random instruction schedule. The two runs of the instruction schedule are grouped together. The distance measure represents the longest execution time relative to the shortest execution time. Two runs that differ more than 5% are marked as outliers. However, this processor did not produce any outliers in our experiment.

- Our random instruction scheduler works on top of LLVM. In this stage of the back-end, LLVM makes still use of pseudo-instructions that are not represented in the binary. This means that schedules that we see as different schedules might actually end up being equal in the binary.
- There are short functions with a short execution time. We observed few changes in the runtime when the measured execution time is below 1,000 processor cycles. The underlying timer of the C++ standard library might not be able to measure such short time periods.

However, the experiment is still valid, because we show that we are able to influence the runtime by manipulating the instruction schedules beyond noise in the measurements.

In summary, we observe different runtimes for different instruction schedules and the results are reproducible over multiple runs. This is not true for all basic blocks, but the goal of this experiment was to show the existence of an effect of the instruction schedule on the runtime. These results motivate further research with two goals: First, to automatically optimizing instruction scheduling for basic blocks. Secondly, to do that only for basic blocks that influence the runtime a lot in order to speed up compilation times.

4.4. Learning to Schedule

Our goal is to create a model that can generate good instruction schedules for basic blocks. Therefore, we have to define a mapping that we want to learn with this model. The possible inputs can be complex, like characteristics of the hardware, the register states, or others. However, we decided for this thesis to begin with a simple model and limit our input to the n last scheduled instructions and the set of instructions that can be scheduled. These are important because they might utilize different execution units on the processor, and thus generate pipeline effects.

We define this mapping as

$$(h_n, \ldots, h_1) \times \{i_1, \ldots, i_m\} \mapsto i_k \tag{4.1}$$

where h defines the instructions in the history of scheduled instructions, i are candidate instructions that are ready to be scheduled and $k \in \{1, \ldots, m\}$. However, we define a second, easier to implement, mapping

$$(h_n, \ldots, h_1) \times i \mapsto r \tag{4.2}$$

where we map from the history and a single instruction to a reward value r. We lose some input information in this model about the other candidate instructions. But we can now simply run the model for all the available instructions and select the one with the best reward.

We need lots of data samples to utilize data-driven, or machine learning approaches. One way would be to generate valid random schedules. The downside of this approach is that we generate many bad schedules because the better ones are rare. A better way is to use MCTS to iteratively find better schedules (see Section 4.4.1). The MCTS approach searches for good schedules locally, i.e., for a single basic block. So, the search is not generalized for other basic blocks.

There are two reasons why we add a second step to our approach: First, the MCTS approach takes a lot of time to complete and is not usable for production environments. Second, this approach only works locally on single basic blocks. To generate a model that works globally, i.e., also for unseen basic blocks, we add a second step and learn from the locally good schedules. Therefore, we extract data samples matching the input format of Equation 4.1, or respectively Equation 4.2. We have trained different models that are explained in detail in Section 4.4.2.

4.4.1. Monte Carlo Tree Search for Instruction Schedules

We choose MCTS to find instruction schedules that perform best for a given basic block. The reason is that this approach fits naturally for sequential problems and allows us to balance between the exploration and exploitation of instruction schedules. MCTS tries all possible next steps in a given situation and chooses the further steps randomly because there is no information available. When all possibilities in a situation are already examined, the algorithm takes the best option (based on a defined metric) and tries to improve from thereon. It does not choose the best option every time but also incorporates exploration choices. This enables us to search for good instruction schedules. See Section 2.5.1 for more details.

The input to our MCTS approach is a DAG generated by the LLVM compiler, which defines the valid instruction schedules. See Figure 2.3 for an example. The LLVM instruction DAG still contains many instructions that exist just for internal reasons and are not translated to machine instructions. We remove these pseudo-instructions from the instruction DAG and adjust the dependencies between the nodes.

The nodes in an MCTS tree represent a state in the underlying model, and the edges correspond to transitions between these states. Compare Figure 4.4 to see that our MCTS nodes represent a partial instruction schedule. The edges in our MCTS tree represent a legal scheduling of a new instruction to the previous schedule. When we iterate back from any node to the root node, we get the partial instruction schedule in reverse ordering.

We do not use the default choice of node evaluation metric. Typically, to evaluate the reward of a node in the MCTS tree, the algorithm compute the average of the child nodes. In our approach, this leads to situations where we have already found a good schedule but keep searching for schedules in another area of the tree. The reason for that is that the good schedule might be surrounded by relatively bad schedules, which have a negative effect on the average evaluation. When we train MCTS for games, that makes sense because we do not want to get into situations where only one good state exists, and the surrounding ones would let us lose the game. But we are only interested in a single best schedule, therefore we do not care about similar but bad schedules. Thus, we do not take the average but the maximum. This means that in the exploitation phase, we always follow the path to the best-found schedule. Figure 4.4 illustrates the effects of both metrics, where the green path represents the maximum metric and the blue path represents the average metric. Our approach is similar to the one in [7]. However, they still use the average for the search part, which we do not.

The MCTS learning process is separated into four phases:

1. Generate schedule:
   Apply the exploration or exploitation selection phase and generate a new schedule based on the previous experience. This results in the most situations in a partial instruction schedule because the MCTS tree usually not grows until it knows all states. The partial instruction schedules are completed by using random scheduling decisions.
2. Compile the generated schedule:
   The generated schedule is transformed into a C++ file with a function that contains the inline assembly of the scheduled basic block. We then compile the schedule with our benchmarking framework (see Section 4.5.4).
3. Execute and measure the runtime:
   The compiled executable is transferred to our target machine and executed there to measure the runtime of the basic block (see Section 4.5.4).

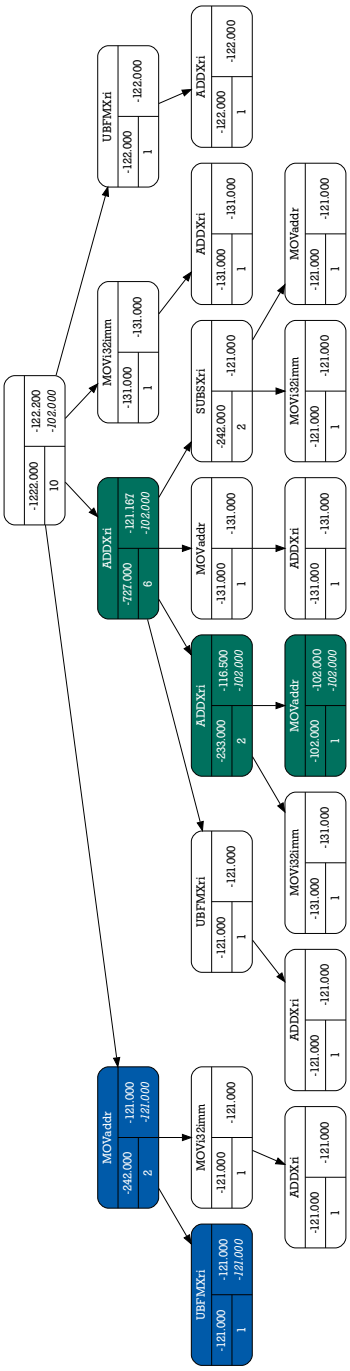


Figure 4.4.: MCTS tree with the consequences of maximum and average metric: The boxes represent a MCTS node, at the top is the name of the instruction, on the left are the cumuluated metric and the number of visits, on the right is the metric. The colored nodes have two metric values: The upper one is the average metric value and the lower italic one shows the maximum metric value. In this example, we use the negative runtime for the metric. The blue nodes show the path that was selected when we used the MCTS approach with the average metric. There is a path of green nodes which shows the path that would have been selected if we used the maximum metric. The green path results in a better metric value.

4. Progress the MCTS tree with the measured runtime:
   Then, the measurement data is copied back to the machine where we run the MCTS training. The measurement data is preprocessed and used to train the MCTS tree.

This process is repeated many times, depending on the experiment. See Section 2.5.1 for more information.

Note that this MCTS approach is not only able to generate data for our global model. It can theoretically also be used in a normal compilation process. It is not practical to use it for a whole program because of the extra compilation steps, caused by the many experiments. However, critical parts of a program could be optimized with this approach.

A positive side effect of this MCTS approach is that we generate an upper limit for the improvements of the instruction scheduling phase. The further we grow the MCTS tree, the higher is the probability that we find the actual upper limit of improvements. This is at the same time an upper limit for the global inference models in our second step. Therefore, we can see how much of our potential improvements we are able to reach with our inference models.

4.4.2. Inference Models

Once we have found good schedules by generating MCTS trees for many basic blocks, we want to generalize this knowledge for unknown basic blocks. This is necessary to transfer the knowledge to unseen basic blocks and make the model usable in practice by generating good schedules directly, in contrast to searching them with MCTS. Therefore, we have developed three different models that we discuss here.

Another way to see this two-step approach is to see the MCTS step as a training phase and this global model step as inference phase, thus the name.

Nearest Neighbor Model

Our first model is a nearest neighbor approach. That means we iterate our MCTS trees and build up a big look-up model in which we can search for situations and see what decision performed best.

In order to build the model, we first define the input and output formats. A scheduling situation has the format

$$(h_n, \ldots, h_1) \times i_1, \ldots, i_m \tag{4.3}$$

where instructions h build a sequence of last scheduled instructions and instructions i build the set of candidate instructions. To simplify our model and have more data per scheduling situation, we limited the size of the set of candidate instructions to two. The disadvantage is that we lose context information on other candidates. The data we are interested in per scheduling situation is the number of times instruction x was better than instruction y or vice versa and the number of times they were on par. This results in the data format

$$(h_n, \ldots, h_1) \times (i_1, i_2) \mapsto (w_1, d, w_2) \tag{4.4}$$

where w_1 and w_2 represent how often instruction i_1, or i_2 was the better choice. How often they were on par is counted by the variable d.

To build the model, we iterate the MCTS trees and extract the contained data. For every scheduling situation (compare Equation 4.3), we take every combination of instructions i. In the example in Figure 4.5, that

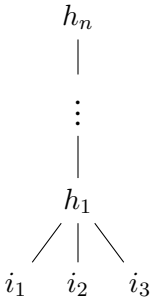


Figure 4.5.: Example scheduling situation with the previously scheduled instructions h and the candidate instructions i to choose from.

would be (i_1, i_2), (i_1, i_3), and (i_2, i_3). For each combination, we check which instruction choice resulted in a better reward. We add this situation to our model if it was not already present. Otherwise, we increase the corresponding counter variable in this situation.

In addition to this situation, we can also learn from this situation for a scheduling situation with a shorter history. So we also add the situation for all partial histories. For example, when we have a scheduling situation with the history (h_3, h_2, h_1), we also add our learnings to the model with the histories (h_2, h_1), (h_1), and the empty history $()$.

This model can be easily applied for scheduling. In a given scheduling situation, we just search for the history and all combinations of available instructions. If we have found all the combinations, we can make our decision based on the counts. For the situation where we did not find all the required combinations, we define a percentage threshold t_c. If we found more than t_c instruction combinations, we make a choice based on the instructions that we have found. If we found fewer than t_c instructions, we remove the oldest instruction h_n from the history and search again for all the instruction combinations.

Support Vector Machine Model

Further, we want to learn parametric regression models. Therefore, we use a different mapping that we learn because the inputs and outputs are scalar values.

$$(h_n, \ldots, h_1) \times i \times (d_{min}, d_{max}) \mapsto r \tag{4.5}$$

We choose to have a sequence of history instructions $(h_n, \ldots, h_1)$ with a defined length n. From the history and the candidate instruction i we map to the reward value extracted from the MCTS data.

One additional metric that we add to our input is the distance into the history of the direct DAG predecessor instructions. We do this because, instructions have to wait for the result of the direct predecessor instructions when they have a data dependency. And the bigger the distance between them is, the higher is the probability, that the predecessors already terminated. For example, instruction a depends on instruction b, and we are in the situation in which we want to schedule a. That means we have already scheduled b. There are usually also other instructions that we scheduled after b. We count the number of instructions that we scheduled between a and b. There is typically more than one instruction that a depends on, so we decided to take the maximum and the minimum number of instructions between a and its predecessors for the dataset. We refer to these values as d_{max} and d_{min} in Equation 4.5.

To build a dataset with the mapping defined in Equation 4.5 we iterate the MCTS trees. For every instruction node, we collect its previously scheduled instructions and its reward value.

However, we cannot insert these data points into the SVM model because it requires numerical values only. We transform the instructions from their textual representation into an ordinal encoding, i.e., every instruction gets a unique numerical representation. Further, we scale all numerical values into the range $[0, 1]$ because SVMs typically learn better when inputs are in this range.

With this data, we train an SVM regression model. The details and results of the experiments can be found in Section 5.3.2.

Neural Network Model

We choose to use a neural network as a second regression model, because they have a shorter inference runtime than SVMs when trained on big datasets. With this model, we try to learn the same mapping as we did with our SVR model, which is defined in Equation 4.5. Consequently, we use the same dataset that we have already built for the SVR model.

In contrast to our SVR model, we chose not to encode the instructions into ordinary values. The instructions are encoded into one-hot vectors because neural networks can handle high-dimensional inputs better. Further, we also scale numerical values into the range $[0, 1]$, like we did for the SVR model.

We use a standard neural network with three hidden layers with dimensions 512, 128, and 64. The output layer is a singular value which represents the scaled score of the given scheduling situation. All hidden layers are followed by a ReLU activation function [45]. We use the Adam optimizer [28] with a learning rate of 10^{-3}. For the loss function, we use the squared error. We discuss the details and results in Section 5.3.2.

4.5. Data Generation

We cannot build our data-driven models (see Section 4.4) directly from the programs in the LLVM Test Suite (Section 2.4), as it requires some transformation from the code to a metric. Many metrics can be chosen for optimization, e.g., runtime, energy consumption, cache misses, processor stalls. We focus on optimizing the instruction scheduling for short runtimes. To generate data for our MCTS approach, we need a mapping from instruction schedules to runtimes.

The most reliable approach is to measure the runtime while executing the code. However, there is research on approaches that rely on analytical models, see Section 3.3.2 We choose the classical approach of runtime measurements since we have access to the hardware, and the resulting data will be more accurate. Therefore, we compile the code with varying instruction schedules and execute it to measure its runtime. We discuss various aspects of this in the remainder of this section.

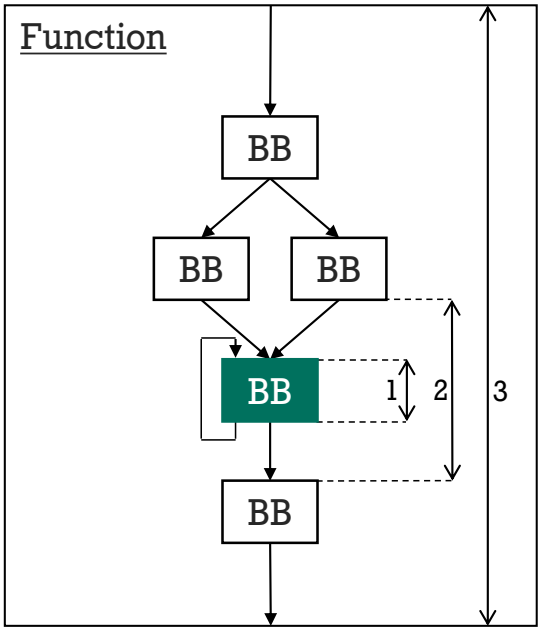


Figure 4.6.: The outer box represents a function. The function consists of multiple basic blocks (BB) where different possible branches are caused by if/else statements and loops. The diagram shows three possible runtime measuring scopes to measure the runtime of the filled basic block. Method 1 shows a measuring inside of a basic block. With method 2, the measurement starts with the end of the last basic block and ends with the start of the next one. Method 3 measures the complete function which surrounds the basic block.

4.5.1. Runtime Measurement Unit

The metric we are interested in specifically is the relative runtime compared to a baseline runtime. Runtime overheads can be ignored, as long as they are constant over multiple runs and the baseline has the same overheads. Thus, there are various possibilities for placing the start and stop events of the timer to accomplish runtime measurements of fractions of the application's code. The objective is to minimize noise and get reliable, reproducible measurements. We discuss the possibilities of making measurements on a basic block, function, and application scope and relevant aspects to these possibilities.

Application

The most straightforward approach to measuring the runtime of a piece of code is to measure the runtime of its whole application. The disadvantage is that we include many more instructions in the measurement than we are interested in. This generates lots of noise from various possible sources (e.g., from Input/Output (I/O) instructions like opening files, printing to the screen, or varying latencies for memory accesses). Also, the longer the runtime of a program, the higher is the probability that the operating system interrupts the execution for more important processes. Depending on the used measurement method (see Section 4.5.2), this may be included or not be included in the measurement. These disadvantages make this approach unreliable and not usable for us because we are interested in the short runtime of a single basic block.

Function

A finer-grained approach is to start the timer at the beginning of the function or method that contains the basic block of interest and stop the timer at the end of the function (see Method 3 in Figure 4.6). This approach creates several problems.

Compared to measuring the runtime of the whole application, this approach executes only a small fraction of the code. This results in a lower degree of variance in the measured runtimes due to fewer OS interruptions, and fewer memory and I/O accesses. Thus, the precision of these measurements is higher.

Note in Figure 4.6, that different control paths are possible in a function, based on if/else statements or loops. This can be problematic in a general setup. However, the benchmarks we use and the data we insert to them are deterministic, i.e., the same paths are taken in each benchmark execution. Theoretically, if there was no variance in the measurements, the shortest measured execution time will always correspond to the same execution path. In practice, we get many varying measurements because of different execution paths. That makes it difficult to differentiate between execution paths and variance in the measurements.

Basic Block

The most intuitive approach when scheduling basic blocks is to measure only the runtime of the basic block we are interested in. We eliminate all extra instructions that we are not interested in. However, we can still get runtime variations for different reasons:

- The basic block could call other functions which could, in turn, execute varying branches. See Section 4.5.4 for a discussion about this.

Operating System	Timer in C language
Linux	`clock_gettime()`
Windows	`QueryPerformanceCounter()`

Table 4.2.: The table shows examples for interfaces in OSs to their timing functions in the C programming language.

- The OS could interrupt the execution. But the probability decreases with shorter code fragments.

One requirement that can not be met in all situations is that of a very precise timer. These timers usually depend on the hardware directly, i.e., the hardware must provide such functionality to measure runtimes. This is possible on our chosen hardware and is discussed in more detail in Section 4.5.2.

Thus, as this is a feasible solution for our experiments, we have chosen to measure the runtime of basic blocks alone.

4.5.2. Runtime Measurement Methods

There are various possibilities to observe execution times in systems that we use for our research. These can typically be classified into the categories of profiling, OS utilities, or hardware utilities. We discuss these three in more detail.

Profiling

Profilers (e.g., gperf [17]) are programs that observe the execution of other processes running on the system. They are designed to observe the execution from start to end and to return various metrics. Some examples are execution times per function, memory consumption, and cache misses. The observation is done by taking snapshots of the process of interest. Thus, profilers are good for finding, e.g., performance problems and memory leaks in software.

However, we need cycle-accurate measurements from instruction A to instruction B for our experiments. Profilers are not able to make such precise measurements, i.e., they are not helpful to us because they are designed for other purposes.

Operating System Methods

OSs typically need to work with time values and also provide a programming interface to use its timing functions. There is the wall clock time and timers that are based on CPU functionality to get times. We are interested in the latter. Often there are various possibilities to retrieve timing values from the hardware with varying precision. The most precise timers are usually good for measuring intervals in the microseconds range or longer. OS functions have some overhead because they also handle various errors. See Table 4.2 for selected OSs. The C++ function `std::high_resolution_clock()` from the standard library falls back to one of these OS functions.

Hardware	Assembly instruction
NEC SX-Aurora TSUBASA	`smir %s60, %usrcc`
AArch64	`mrs x0, PMCCNTR_EL0`
ARM32	`mrc p15m, 0, r0, c15, c12, 1`
x86	`rdtsc`

Table 4.3.: Assembly instructions for getting the cycle counter value of a given hardware.

Some OSs not only provide timing methods, but also allow to manage how a process executes. Linux provides different methods to avoid or reduce the probability of OS interrupts during the execution of a process.

- Isolating CPU cores: The Linux kernel command line option `isolcpus=<cpu-number>` in the boot loader instructs Linux to not schedule tasks to the given CPU cores automatically.
- Pinning a process to a CPU core: The Linux program `taskset` configures the OS to schedule processes to specific CPU cores. This can also be used to schedule processes on isolated CPU cores.

If we pin a process to an isolated CPU core, this task will not be interrupted by the OS. This makes the measuring of runtimes more reliable.

Hardware Performance Counters

The most code blocks we want to measure are only 5-50 instructions long, i.e., we need very precise measurements. We use the assembly instructions of the given hardware directly to get the most precise timing values with the lowest overhead. These instructions typically return the value of a processor cycle-counter.

Depending on the used high-level language and hardware, using assembly instructions can be complicated. But it is not a problem in our experiments because the code blocks we want to measure are already written in the assembly language of the given hardware. That means we only add a timer instruction at the beginning and one at the end. We do some error checking after the execution has stopped (see Section 4.5.4 for details). See Table 4.3 for the assembly instructions per hardware and Figure 4.8 for an example.

Depending on the hardware and the OS, access to these instructions or registers might not be permitted. For the AArch64 architecture and a Linux OS, we had to implement and load a kernel module that granted access to this register.

4.5.3. Execution Unit

Due to many iterations in the learning processes, we have to execute the differently scheduled basic blocks many times. If we executed the whole benchmark that contains the basic block of interest, it would be time-intensive, and many lines of code were executed that we are not interested in. This forces us to extract the basic block from the benchmark and execute it in isolation. Here, we discuss the advantages and disadvantages of the two possible approaches.

Benchmark

The first option is to execute the complete benchmark for every new schedule that was generated during the learning process. The advantage is that we execute the original basic block as it would appear in the real benchmark. The problem with this approach is that the execution time of a benchmark can be high and makes the execution the bottleneck in our pipeline. Many benchmarks in our dataset take several minutes to execute on a RaspberryPi 3B. So, always executing the whole benchmark is costly to even get only one runtime measurement per schedule. In order to get statistically relevant runtimes, we have to measure them multiple times.

Basic Block

The alternative to executing the whole benchmark is to extract the scheduled basic block from the benchmark and execute it in isolation. By using this approach, we reduce the execution time to a minimum and do not execute code we are not interested in. Consequently, we can execute the benchmark many times and get statistically reliable runtime measurements. The disadvantage is that we have to slightly modify the basic block or the execution crashes.

4.5.4. Basic Block Isolation

As discussed in Section 4.5.3, we are forced to execute basic blocks isolated from the benchmark. To make a basic block executable, we isolate it, normalize it, and insert it as inline assembly into our minimal C++ framework.

Basic Block Extraction

First, we compile the benchmark and schedule the basic block with our modified compilation process (see Section 4.1.1) into assembly files. Next, we extract the basic block instructions from the assembly file. Then, we modify the basic block instructions to prevent execution and compilation errors, and as the last step, we inject it into our C++ framework. We describe the modifications in the next paragraphs.

Assembly files contain comments, labels, and assembler directives which we remove. Comments are useless lines for the execution task. The labels are not required, as we do not jump to other instructions. Assembler directives represent commands to the assembler program for bookkeeping, storage reservation, and other purposes. These could interfere with our compilation process and are also removed.

Basic blocks can contain jump instructions as the last instruction and function calls in any position. We only schedule the instructions in the given basic block, so we are also only interested in its own runtime. Executing other code, which could internally choose different paths, would add noise to our measurements. To prevent this uncontrolled program execution, we simply remove jump and call instructions. The removal of these instructions might slightly reduce the execution time, but the reduction is constant over all executions with different schedules.

In order to prevent segmentation faults, we have to redirect all memory accesses to a memory area that we are allowed to read and write to. This affects all load and store instructions. In our framework, we allocate 8

```
 1 .LBB9_14:                // %for.body82
 2                          //   Parent Loop BB9_2 [..]
 3                          //     Parent Loop BB9_[..]
 4                          // =>    This Inner Loo[..]
 5 sub     x24, x23, #8         // =8
 6 cmn     x24, #8              // =8
 7 ldr     d0, [x21]
 8 ldr     d1, [x19, x23]
 9 ldr     d2, [x22, #-8]!
10 ldr     d3, [x7, x23]
11 fmul    d2, d2, d3
12 fsub    d0, d0, d2
13 fdiv    d0, d0, d1
14 str     d0, [x21]
15 mov     x21, x22
16 mov     x23, x24
17 b.ne    .LBB9_14
```

(a) Original assembly

```
 1
 2
 3
 4
 5 sub     x24, x23, #8
 6 cmn     x24, #8
 7 ldr     d0, [%2]
 8 ldr     d1, [%2]
 9 ldr     d2, [%2]
10 ldr     d3, [%2]
11 fmul    d2, d2, d3
12 fsub    d0, d0, d2
13 fdiv    d0, d0, d1
14 str     d0, [%2]
15 mov     x21, x22
16 mov     x23, x24
17
```

(b) Manipulated assembly

Figure 4.7.: AArch64 Assembly Basic Block Manipulation: We remove the first line because it is a label. We remove the comments in the lines two to 6. Then, we replace the memory accesses with the placeholder `%2`, which the compiler replaces later (see Figure 4.8). Lastly, we remove the jump instruction in line 17.

byte on the stack where we redirect all memory accesses. This also means that all memory accesses are cache hits, and we have a controlled environment in this aspect.

Figure 4.7 shows an example in which we manipulate a basic block. We remove a label and comments, redirect memory accesses and remove a jump instruction.

Isolated Basic Block Execution

To execute the extracted basic block, we have to surround it with some other code to generate a valid C/C++ program. For this, we implement a small benchmarking framework. Figure 4.8 shows an example of the basic block from Figure 4.7 integrated into this framework. We also add the CPU timer calls and mark the used registers as clobbered. The optimization is turned off in this compilation step, so the basic blocks are executed as given.

The executable first executes the basic block in 100 warmup rounds to prevent side effects from caches and other reasons. The actual execution where we measure the runtime is repeated until 1,000 valid measurements are done. Invalid measurements can occur when the runtime is negative. This happens when the cycle counter overflows.

4.5.5. Computing Rewards From Runtimes

Even though we do the warmup rounds, we still get few data points that are clearly outliers and should not be included in our experiments. We sort the 1,000 runtime measurements and strip the top and bottom 50

```
1  #pragma once
2
3  #include <cstdint>
4
5  struct Benchmark {
6  public:
7      __attribute__((always_inline)) uint64_t operator() () {
8          uint64_t a[4] = { 0xcdab, 0xcdab, 0xcdab, 0xcdab };
9          uint64_t* p = a;
10         uint64_t start, end;
11         asm volatile(
12             "mrs        %0, PMCCNTR_EL0 \n\t"
13             "sub        x24, x23, #8    \n\t"
14             "cmn        x24, #8         \n\t"
15             "ldr        d0, [%2]        \n\t"
16             "ldr        d1, [%2]        \n\t"
17             "ldr        d2, [%2]        \n\t"
18             "ldr        d3, [%2]        \n\t"
19             "fmul       d2, d2, d3      \n\t"
20             "fsub       d0, d0, d2      \n\t"
21             "fdiv       d0, d0, d1      \n\t"
22             "str        d0, [%2]        \n\t"
23             "mov        x21, x22        \n\t"
24             "mov        x23, x24        \n\t"
25             "mrs        %1, PMCCNTR_EL0 \n\t"
26             :  "=r"(start), "=r"(end)
27             :  "r"(p)
28             :  "q0", "q1", "q2", "q3", "x21", "x23", "x24",  "memory", "cc"
29         );
30         return end > start ? end - start : 0;
31     }
32 };
```

Figure 4.8.: AArch64 Assembly Basic Block Integration into Execution Framework: This example shows the basic block from Figure 4.7 how it is integrated into the C++ framework. The compiler replaces the placeholder `%2` with a pointer to the array `a` (line 8). We also list the used registers, so the compiler knows that their values have changed (line 28). The first and the last instructions in the inline assembly (lines 12, 25) call the CPU timer.

Runtime	Num. Occurances
140	998
143	1
711	1

Runtime	Num. Occurances
121	708
120	289
123	1
163	1
1491	1

Runtime	Num. Occurances
114	994
121	2
115	1
129	1
130	1
694	1

Table 4.4.: Example runtime measurement distribution for three different schedules. (Benchmark: almabench, Function: anpm, Basic Block: entry)

measurements. Spot checks have shown that we remove all the outliers in all the checked experiments. See Table 4.4 for an example with three different schedules for the same experiment.

Because different basic blocks have their different runtimes, the absolute measured speedups between two basic blocks are not comparable. To have a metric m that we can compare with all other basic blocks in the data set, we normalize the experiment runtime r_e with the runtime of the baseline scheduler r_b.

$$m = \frac{r_b}{r_e} \tag{4.6}$$

5. Evaluation

This chapter explains how we use the system described in Chapter 4 to evaluate and discuss its performance. First, we review the results of the MCTS approach in Section 5.2. Next, we evaluate if we can use the results of the MCTS approach for learning to generate good schedules with supervised learning methods in Section 5.3. Lastly, we analyze and discuss the outcomes of the experiments in Section 5.4.

5.1. Hardware Selection

There are different aspects in choosing the hardware for our experiments. A processor must fulfill two aspects to be relevant for us. The first is that LLVM supports it because we base our whole approach on LLVM and use the LLVM instruction scheduler as a baseline. Secondly, we limit the hardware selection to processors that implement superscalar pipelines (see Section 2.1.3) to ensure that we use a realistic setup because most modern processors implement these.

There are additional essential aspects regarding the hardware. The most interesting one is if the processor implements an in-order superscalar pipeline or an out-of-order one. While the former comes with no restrictions for our experiments, the latter can reschedule instructions during execution. Consequently, it is interesting to see how the out-of-order model influences the performance of our approach.

Additionally, to show the versatility of our approach, it is interesting to choose different types of processors. For example, there are classical CPUs, Edge CPUs, and accelerator cards (e.g., Graphical Processing Units (GPU), Vector Processing Units (VPU), and Intelligence Processing Units (IPU)).

With these aspects in mind, we select:

- **AArch64 (Arm Cortex-A53):** This processor is an edge CPU and implements an in-order superscalar pipeline based on the AArch64 architecture. An edge device that uses it is the Raspberry Pi 3 Model B. We use this device for our experiments with Ubuntu 20.04.
- **NEC SX-Aurora TSUBASA:** This processor is interesting because it implements an out-of-order superscalar pipeline. Additionally, it is a vector processing accelerator card installed via a PCI-Express connection and thus a different type of processor.

	Processor	
MCTS Performance	AArch64	NEC SX-Aurora TSUBASA
Absolute		
Better than baseline	54.79% (7759)	31.73% (1349)
Same as baseline	36.97% (5236)	53.00% (2253)
Worse than baseline	8.24% (1167)	15.27% (649)
Runtime		
Mean Speed Up	8.35%	0.30%

Table 5.1.: Results of the MCTS approach. The MCTS approach was very successful on the AArch64 processor. We found better instruction schedules for more than the half of the basic blocks. On the SX-Aurora, we are on par with the baseline for half of the basic blocks and found better instruction schedules for a third of the basic blocks.

5.2. Monte Carlo Tree Search Schedule Search

We pursue two goals with this experiment: One is to search for instruction schedules that perform better than our baseline, for which we choose the LLVM default instruction scheduler. The second goal is to build a dataset that we can use for our supervised learning approaches. We also generate an upper limit for the supervised models' performance, because we use the MCTS results as their basis and they cannot exceed that performance.

For each basic block under consideration, we first measure the runtime of the basic block compiled with the baseline instruction scheduler. Next, we start the MCTS approach and generate an instruction schedule in each MCTS iteration. We compile the instruction schedule into an executable format (Section 4.5.4) and execute it to measure the runtime of the basic block of interest (Section 4.5.2). We train the MCTS model with the reward computed by Equation 4.6 and start the next iteration to train the MCTS model. Thereby, we generate many instruction schedules per basic block, each evaluated with a reward based on their execution time relative to the default instruction schedule.

To have enough instruction schedules available for our supervised learning methods, we select 20,032 basic blocks from the LLVM Test Suite. We generate one MCTS model for each basic block. For this experiment, we have selected the longest basic blocks in the dataset. The number of executions is not relevant in this experiment because we isolate the basic block and measure their specific runtime on the target hardware. A high number of instructions in the basic blocks helps to avoid trivial scheduling situations.

This experiment is time-consuming because of the sheer number of basic blocks and the expensive compilation and runtime measurements. After 200 iterations, we have seen that we achieve a speedup for many basic blocks. Therefore, we run the MCTS model for each basic block for 200 iterations. The experiment took five weeks to execute for the AArch64 processor, with two Raspberry Pi 3B executing one process per CPU core, i.e., four parallel processes each. An external machine compiled the benchmarks with an AMD EPYC 7702P 64-Core processor and 128 processes in parallel. The same experiment took three weeks on two SX-Aurora cards with a single process each. The compilation was done by an Intel Xeon Gold 6126 with 24 processes in parallel.

We try to avoid as many compilations and runtime measurements as possible because they take a long time. Hence, we cache the generated schedules and reuse the measurements. There are two caching levels in this

approach: First, we generate the instruction schedules in the format that the LLVM back-end uses and add the schedule to the cache. The instruction schedule is then compiled to assembly code. We cache the assemblies also because the format of the LLVM back-end still contains pseudo-instructions and two different schedules can lead to two equal assemblies.

For the AArch64 we were able to run this experiment on 14,217 basic blocks. Due to various errors, we measured an unrealistic speedup for some basic blocks. Thus, we mark all speedup factors greater than two as outliers. That leaves us with 14,162 valid instruction schedules for the AArch64 processor. Table 5.1 summarizes the results and shows that we find better-performing instruction schedules for 54.79% of these basic blocks. In only 8.24% of the basic blocks, we did not find an instruction schedule that performed at least as well as the LLVM generated one. On average, we increase the runtime performance of the basic blocks by 8.35%.

We also executed this experiment for 4,253 basic blocks on the SX-Aurora processor. The lower number of basic blocks is caused by hardware and time limitations. Only two outliers are generated during this experiment. See Table 5.1 for the summarized results. For this processor, our MCTS approach found better instruction schedules for 31.73% of the basic blocks. In 15.27% of the basic blocks, our model finds only worse instruction schedules. The average speedup of the basic blocks is 0.30%.

If we run this experiment on more basic blocks, the results could still change in favor of the SX-Aurora processor. However, the observation that the performance on this processor is worse than on the AArch64 processor was expected. The reason is that the SX-Aurora is an out-of-order processor, and the AArch64 processor is an in-order processor. That means, the SX-Aurora might reschedule the instruction in hardware when it detects problems with the instruction schedule. Thus, it does not depend on good instruction schedules as the AArch64 processor.

We showed with this experiment that we can find better instruction schedules for both our selected processors. However, our search for instruction schedules was more successful for the AArch64 processor, as it does no further rescheduling in hardware.

5.3. Supervised Schedule Generation

As discussed, the MCTS approach is not usable for production systems because of its long runtime. We use the dataset, generated by the MCTS approach, for other inference approaches in this section. First, we evaluate the nearest-neighbor model in Section 5.3.1 and then the parametric models in Section 5.3.2.

The dataset that we use for our supervised models is based on the results of the MCTS approach (Section 5.2). We split this dataset into a training set with 85% randomly selected data points and a test set with the remaining 15%. Figure 5.1 illustrates the usage of the created datasets.

These experiments aim to see if we can generate well-performing instruction schedules without search methods, like the MCTS approach.

5.3.1. Nearest Neighbor Model

We use a large map structure to search our dataset for similar scheduling situations quickly and set the threshold value $t_c = 0.8$. Section 4.4.2 explains t_c and more details of this approach. Then, we integrate this model into the LLVM compiler framework, and we use it to compile our basic blocks in the test set.

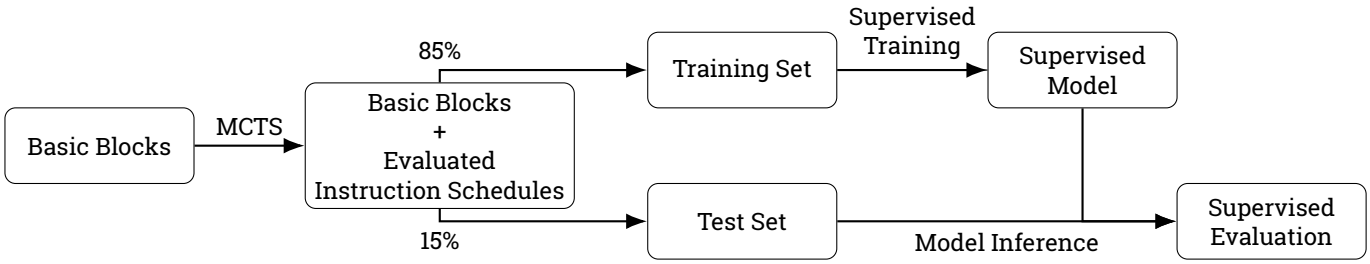


Figure 5.1.: Overview over used datasets. The supervised model is one from Section 5.3.

	Processor	
Supervised Model	AArch64	NEC SX-Aurora TSUBASA
Nearest Neighbor	**1.38%**	-3.03%
Support Vector Regression		
Balanced + Clustered	-1.14%	-3.53%
Balanced	-1.18%	-3.20%
Clustered	-1.45%	**-2.90%**
Neural Network		
Balanced + Clustered	-0.48%	-4.19%
Balanced	-0.19%	-3.19%
Clustered	-0.47%	-3.31%

Table 5.2.: Performance of our supervised models relative to the baseline: This table shows the mean speedup on the test set with our applied supervised learning models. Our nearest neighbor model performed best on the AArch64. It is the only model that generated a positive mean speedup. On the SX-Aurora, the SVR model with clustered instructions performed best. However, it is still worse than the baseline.

The instruction schedules that we compiled with the nearest-neighbor model for the AArch64 processor performed better than the baseline instruction scheduler from LLVM. The measured runtimes for the basic block are 1.38% shorter. In fact, this is the only experiment where an inference approach exceeds the baseline performance. However, the measured runtimes on the SX-Aurora processor are 3.03% longer than the basic blocks compiled with the baseline instruction scheduler (see Table 5.2).

5.3.2. Parametric Machine Learning Models

The parametric models require an additional data transformation to bring it into the form defined in Equation 4.5. This results in a dataset of 4.9 million data points for the AArch64 architecture and 1.3 million data points for the SX-Aurora architecture. We add two variations to the parametric approaches, which we describe in the following paragraphs. To evaluate the effects of these variations, we run all approaches once with both and once with only one of the variations.

There are many similar instructions in the instruction sets of the two processors. To reduce the dimensionality and simplify the dataset, we cluster some instructions into an alias-instruction. For example, we cluster the addition-instructions `ADDWri` and `ADDXri` of the AArch64 architecture into the same cluster. These two instructions only differ in that one takes 32-bit values and the other 64-bit values. See Appendix A for exact clusterings.

Further, we balance our dataset in the target dimension, which holds the reward values from the MCTS trees. The distribution of target values in our dataset roughly follows a normal distribution. However, this can be problematic because the model might only learn to predict the mean of that distribution in any situation. Therefore, we duplicate samples whose target value is further away from the mean and delete samples whose target value is very close to the mean. This results in a dataset with a distribution closer to an equal distribution than to a normal distribution. Figure 5.2 illustrates the effect on the distribution of the AArch64-dataset. The effect for the SX-Aurora dataset is similar.

Support Vector Regression

SVMs have long runtimes when trained with many data points. We reduced this by randomly selecting 200,000 data points for the model training.

For the AArch64 processor, this model is the worst-performing one of the supervised learning models. The measured runtimes of the scheduled basic blocks are 1.14% to 1.45% longer than the baseline's runtimes.

However, on the SX-Aurora, we found this SVR approach combined with clustered instructions to be the best working parametric model. Nevertheless, with a slowdown of 2.90%, it still performs worse than on the AArch64 processor.

Regarding the effects of the dataset balancing and instruction clustering, we see no apparent effect. For the AArch64 processor, the experiments with the balanced datasets perform better. However, we observe the opposite effect for the SX-Aurora. Here, the dataset balancing negatively influenced the performance.

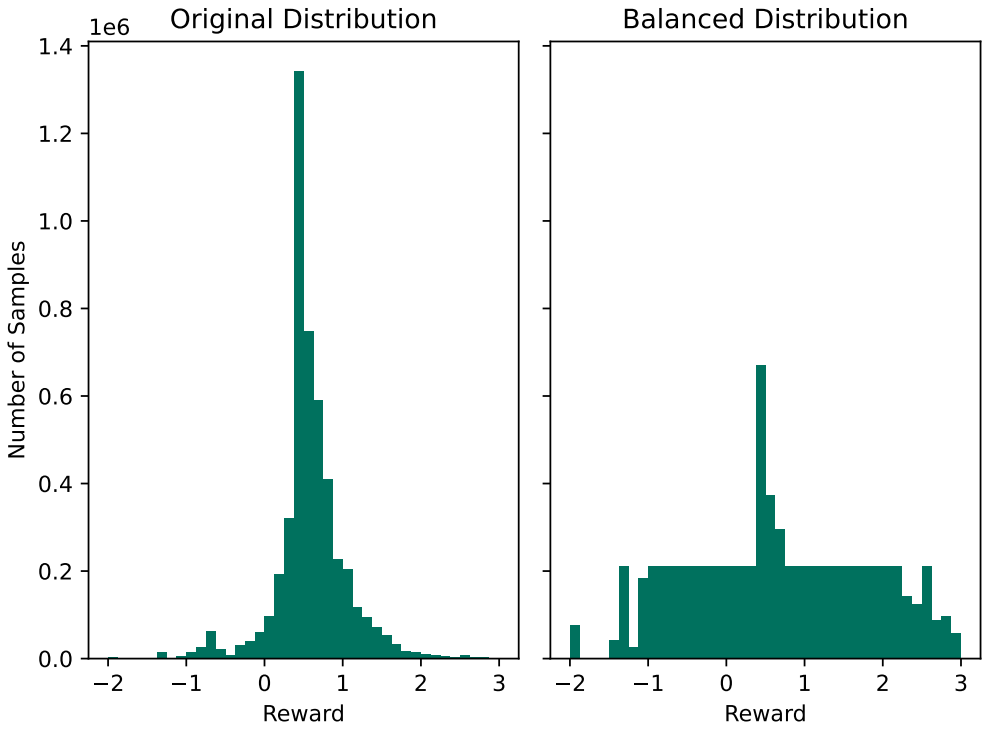


Figure 5.2.: Balancing for the AArch64 dataset: We duplicate samples with a reward further away from the distribution mean, and delete some samples that are close to the mean. To not skew the dataset too much, we limit the number of deleted or duplicated samples per bin. Therefore, we do not get an exact equal distribution, but still have more samples around the mean and fewer samples far away from the mean.

Neural Network

For training the neural network, we use the early stopping scheme. When the validation loss did not improve, for at least 10^{-6}, once in the last ten epochs, we abort the training. Computing the validation loss requires another split into training and validation set. We choose an 85/15 split again.

On the AArch64 processor, the scheduled basic blocks perform better than with the SVR approach. With the balanced dataset, we get close to the baseline performance but we still perform slightly worse than the baseline.

On the SX-Aurora, the three neural network approaches performed the worst of all supervised learning approaches on this processor. Here, the neural network approach with both, the balancing and clustering activated, is the worst of all approaches.

5.4. Discussion

5.4.1. Applied History Length for Nearest Neighbor Approach

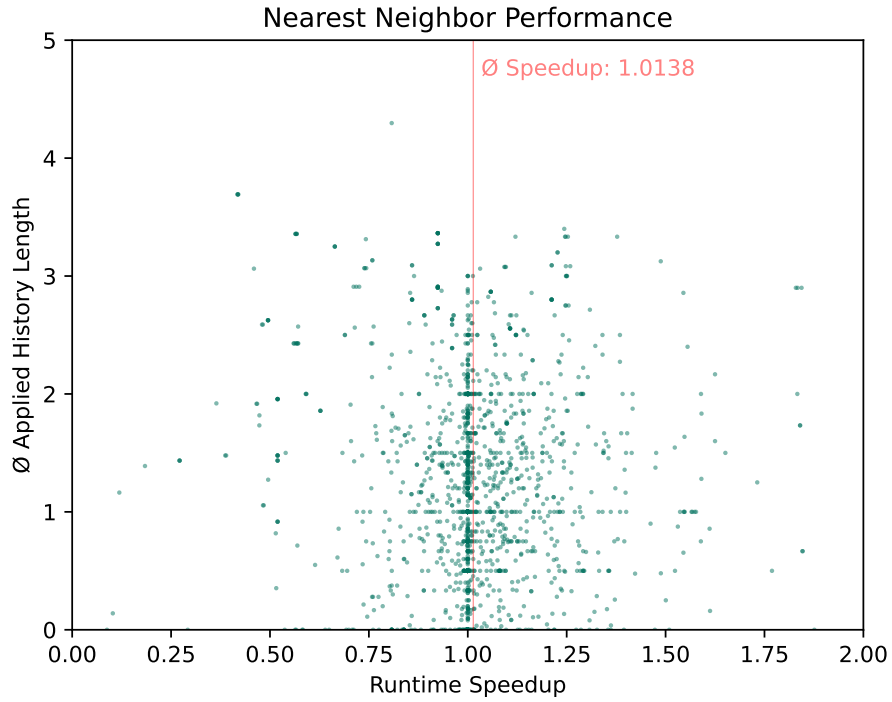
In order to understand the results of the nearest neighbor approach, we want to investigate the influence of the history length on the performance. We define the history length as the number of previously scheduled instructions, included when searching for relevant scheduling situations. We choose the maximum history length for our nearest neighbor model to be five instructions. When searching for relevant scheduling situations, we might ignore the oldest history instructions to find more relevant situations. Therefore, we speak of an applied history length and take the average of the applied history lengths of all scheduling decisions in a basic block. We refer to Section 4.4.2 for a detailed explanation.

Having a longer applied history length means that we make decisions based on more information. Therefore, we expect that the runtime speedup is higher for basic blocks with a higher applied history length. Figure 5.3 illustrates each basic block in the test set with its runtime speedup and average applied history length. However, we do not observe the expected correlation. For the AArch64 processor, Figure 5.3a shows a big point cloud with no visible correlation. Figure 5.3b shows more worse-performing instruction schedules around an applied history length of 2 for the SX-Aurora. Still, there is no clear correlation observable.

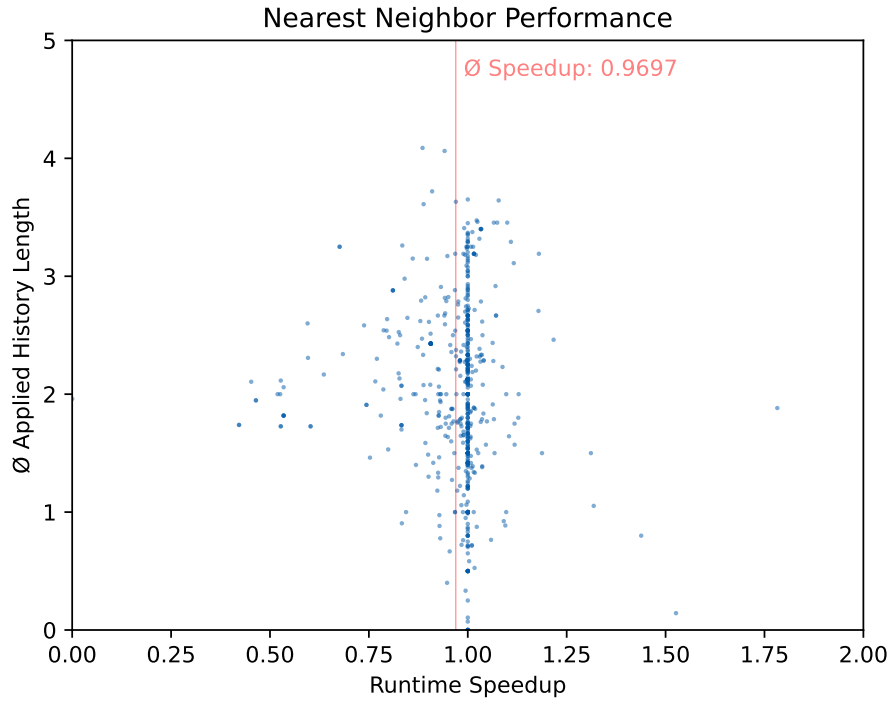
5.4.2. Effect of Partially Unknown Schedules in Monte Carlo Tree Search

The MCTS approach works well for searching well-performing instruction schedules for basic blocks. However, the approaches that depend on the results of the MCTS did not reach comparable performances. One reason for that might be the unknown latter part of the instruction schedules generated by MCTS.

As a reminder (see Section 2.5.1 for details): The MCTS approach starts with selecting instructions that already are in its tree. Once the algorithm reaches a leaf in a tree path, it selects random child nodes, i.e., random candidate instructions. Every node, i.e., partial instruction schedule, has an assigned reward. However, this reward depends not only on the partial instruction schedule already present in the tree but also on the following randomly selected instructions. Figure 5.4 explains this with an example.



(a) AArch64



(b) NEC SX-Aurora TSUBASA

Figure 5.3.: Influence of the Applied History Length on Nearest Neighbor Speedup

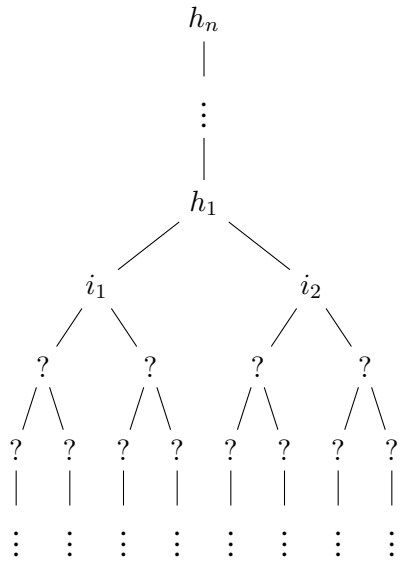


Figure 5.4.: Nodes in the MCTS tree that influence the rewards: This scheduling situation has the history of instructions $h_1, \ldots, h_n$ and the two candidate instructions i_1 and i_2. When either of the candidate instructions is selected, the schedule gets filled with randomly selected following candidate instructions. Thus, the reward that the instructions i_1 and i_2 achieve do not depend only on itself and the history, but also on the randomly selected instructions.

In tasks where nodes with similar rewards are grouped close together and nodes with differing rewards are further apart, that is no problem. Contrarily, in our case, there seem to be single scheduling decisions that can significantly change the whole schedule's performance. Figure 5.5 illustrates an example of such a decision.

As long as we use these random instructions only in the MCTS approach, it is no problem that they influence the rewards of partial schedules. However, we use the tree also to generate the datasets that we use for the supervised learning approaches. The problem is that we isolate scheduling decisions from their MCTS tree. We use the previously scheduled instructions, the candidate instructions, and their reward, but we do not have any information about the randomly selected instructions that also influence the reward. Therefore, the reliability of the rewards is limited.

The same problem applies analogously when we limit the history length and cut off instructions at the schedule's beginning. Additional research is required to find a better way to make the MCTS reward usable in inference models.

5.4.3. Conflicting Data in Regression Dataset

There could be multiple reasons why the performance of the regression models cannot compete with the performance of the MCTS approach. We investigate the number of conflicting data points in the dataset. We consider two data points to be conflicting if they describe the same scheduling situation (according to Equation 4.5) but have different rewards. Conflicting data points can hurt the training process. However, slight differences might be negligible.

There might be multiple possible ways to find conflicting data points. We compute the distance between the minimum and the maximum rewards for data points with the same scheduling situation. Figure 5.6 visualizes

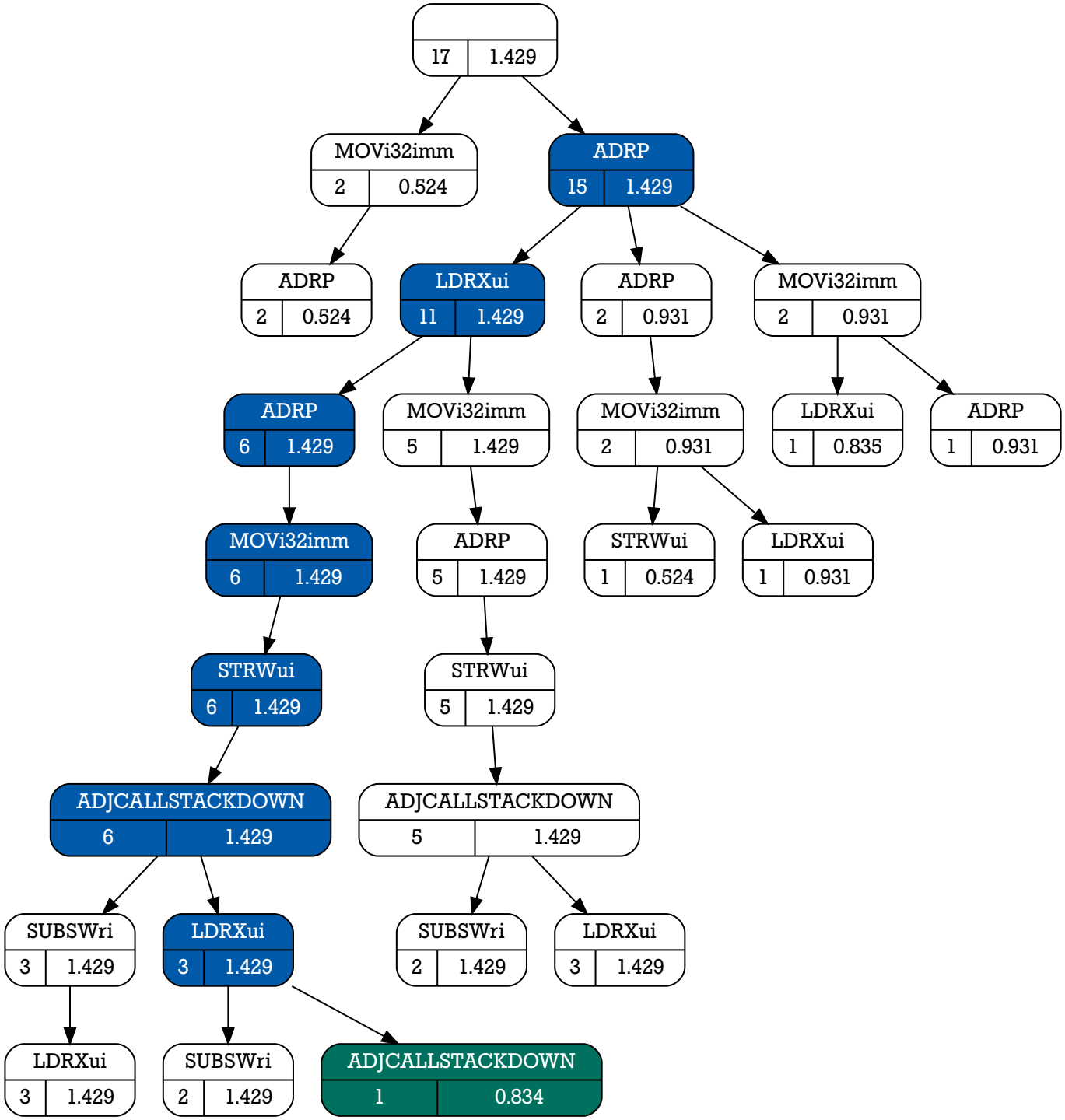


Figure 5.5.: Importance of Single Scheduling Decisions: All the blue partial paths have been evaluated with the reward 1.429. But after seven scheduling decisions, the deicision for the green instruction decreased the schedule performance to 0.834. This is an example from the MCTS approach for the AArch64 processor.

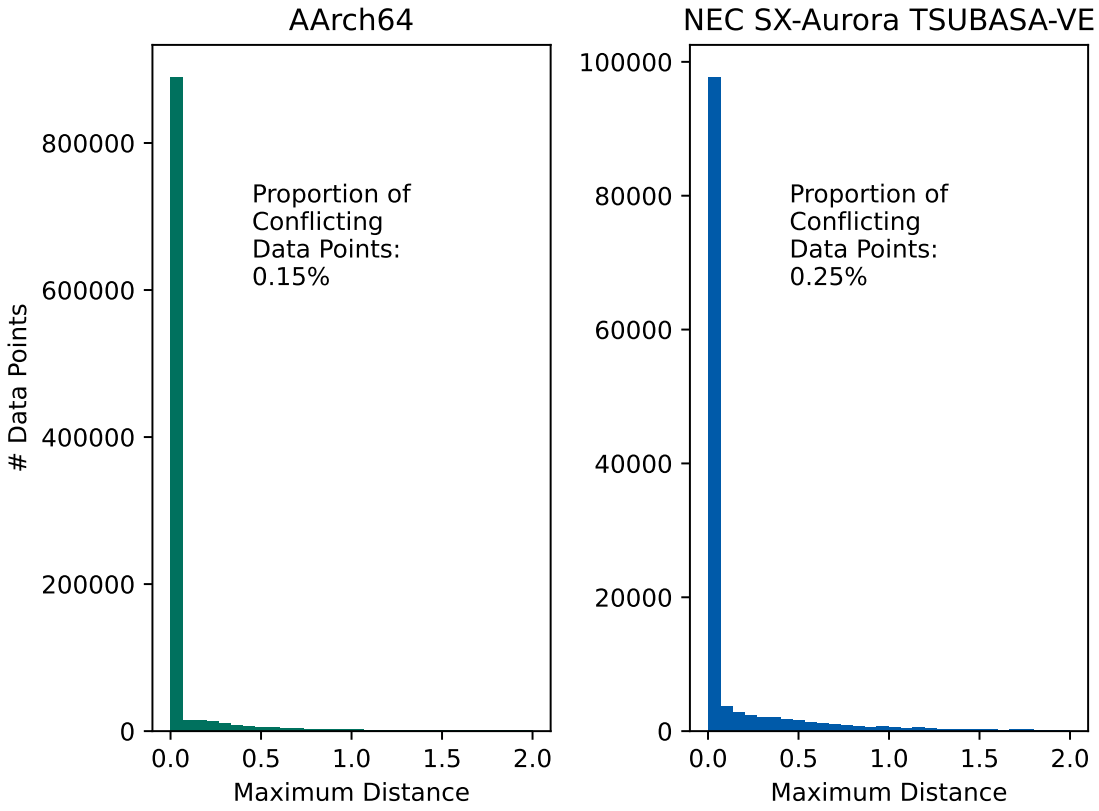


Figure 5.6.: Histogram Over the Maximum Difference Between Rewards of the Same Scheduling Situation: We see that for either architecture, there are conflicting data points. However, the number of conflicting data points is low.

the distribution of these distances. There are conflicting scheduling situations, but the large majority of data points is not conflicting. Next, we could investigate the distribution of rewards for data points with the same scheduling situations. However, this is not necessary because we have already seen, that the overall situation is acceptable.

Even though there are not many conflicting data points, it might still be worth it to clean the data. The cleaning might be a difficult task because deciding which data points to prioritize is not trivial. Note that this is one, but not the only, consequence of the problem from Section 5.4.2.

5.5. Summary

The evaluation of our experiments showed good results for the MCTS approach. We improved the average runtime of the testset's basic blocks by 8.35% for the AArch64 processor compared to the state-of-the-art LLVM scheduler. For the out-of-order SX-Auroraprocessor, we did not observe the same results but still gained a small speedup of 0.3%.

The best supervised learning model that we have found is the nearest neighbor approach on the AArch64 processor. In fact, it is the only one that performed better than the baseline instruction scheduler. The neural network that was trained with the balanced dataset gets close to the baseline performance.

For the two dataset variations where we balanced the dataset and clustered similar instructions, we can see that the balancing was helping. Table 5.2 shows that the models trained with the balanced dataset performed better than with the clustered dataset in three out of four cases. It even performed better than the approach with balanced and clustered dataset in three out of four cases. In the other case it is close. To summarise, the balancing helped overall, and the clustering had a negative influence on the performance. It seems, that it indeed is important to differentiate between instructions that only differ in small aspects like the bit width.

We have seen that our supervised approaches have all worked better on the AArch64 processor. This was expected due to the smaller dataset and the low average speedup in the MCTS approach for the SX-Aurora processor. Table 5.1 shows that the SX-Aurora processor only achieved a 0.30% speedup in the MCTS approach. This value can be seen, as an upper limit for the supervised learning approaches because they are based on the results of the MCTS approach.

We observed that the applied history length does not have a clear influence on the performance of the nearest neighbor approach. This requires additional investigation. Additionally, we have discussed the expressiveness of MCTS rewards due to partial instruction schedules and their problematic effect on the dataset for supervised learning approaches. Lastly, we showed a weakness of this dataset.

6. Conclusion and Future Work

In this last chapter, we summarize our work and outline our contributions and results. Further, we present possible research directions that this thesis opens and discuss open questions.

We showed that state-of-the-art compilers do not generate optimal instruction schedules. With a search approach, we can find better-performing instruction schedules in many cases. Further, we showed that it is possible to train a supervised model that can generate better instruction schedules than modern compilers. Our experiments confirmed that compile-time instruction scheduling has a lower influence on the benchmark runtime on out-of-order processors than in-order processors.

To build a methodology that generates optimized instruction schedulers for any novel hardware, we have built our approach on top of the LLVM compiler framework. We propose an approach that first searches for well-performing instruction schedules for a set of basic block benchmarks. However, the search takes too much time to do it at compile-time. Therefore, we build a dataset with the evaluated basic blocks. To speed up the instruction scheduler, we use this dataset to train a supervised model that the instruction scheduler then uses.

With our approach, it is possible to reduce the runtime of compiled computer programs on any hardware. The influence is especially big for in-order architectures. Mobile and edge devices widely use cheaper processors, and also accelerators like GPUs usually implement in-order architectures. Shorter runtimes are an important achievement, but shorter runtimes also come with reduced energy consumption, which is especially important for devices powered by batteries.

Our contribution is a pipeline that automatically optimizes instruction schedulers for any hardware. Therefore, we developed metrics to select basic blocks with a strong influence on the overall runtime of the program. We extract these basic blocks to execute them in our runtime measurement framework. The selected and extracted basic blocks are used in an MCTS approach to find well-performing instruction schedules for each basic block. As a consequence, we get many evaluated scheduling decisions. We use them to train supervised models to generate well-performing instruction schedules for unknown basic blocks. Additionally, we made a feasibility study that showed the potential of optimizing instruction schedulers.

While our work performs well on the in-order architecture, the results are not as good on the out-of-order processor. The search for well-performing instruction schedules achieves an average runtime speed up of 0.3%. This is an improvement, but it is also an upper limit for the supervised learning models, which all perform worse than modern compilers. Further, we can say that our supervised learning models are still a weakness of our pipeline. There is potential for improvements and could be a direction for more research.

Our research laid out multiple directions in which future work could aim. We used only basic information in our search and supervised learning models, mostly just about previously scheduled instruction and ready-to-schedule instructions. The LLVM framework includes many heuristics used in instruction scheduling, which could provide additional information. This additional information might help the supervised learning models. Common heuristics are, for example, whether or not an instruction is part of the critical path of the DAG, the register pressure, or the distance in the schedule of a candidate instruction to its predecessors.

Further, extending the supervised learning models with an initialization phase that examines the whole DAG would be interesting. It could then extract relevant information from the graph structure of the DAG. This would allow the instruction scheduler to plan multiple steps ahead instead of being limited to the information about previously scheduled instructions and the candidate instructions. A tool that could help with this is graph neural networks.

Imaginable is also to apply an instruction embedding method to transfer the instructions into a multi-dimensional vector space. We have discussed some of these approaches in the literature review in Chapter 3. Word embeddings are very successful in NLP tasks and might also help in this task.

Additionally, our analysis in Section 5.4 opens more research directions. It would be interesting to further investigate why there is no correlation between the applied history length and the runtime performance of instruction schedules (compare Section 5.4.1). Next, we might want to develop a dataset that limits the target values to the influence of that specific scheduling decision and discards the influence of unavailable information (see Section 5.4.2).

List of Figures

List of Tables

List of Acronyms

CNN Convolutional Neural Netoworks . . . 23
CPU Central Processing Unit . . . 7
DAG Directed Acyclic Graph . . . 17
GPU Graphical Processing Unit . . . 7
I/O Input/Output . . . 43
IR Intermediate Representation . . . 14
MCTS Monte Carlo Tree Search . . . 22
NLP Natural Language Processing . . . 23
NOP No-Operation . . . 12
OS Operating System . . . 33
SSA Single Static Assignment . . . 15
SVM Support Vector Machine . . . 23
SVR Support Vector Regression . . . 23

Bibliography

[1] Bruce Abramson. “Expected-outcome: A general model of static evaluation”. In: *IEEE transactions on pattern analysis and machine intelligence* 12.2 (1990), pp. 182–193.

[2] Uri Alon et al. “code2vec: Learning distributed representations of code”. In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–29.

[3] Amir H Ashouri et al. “A survey on compiler autotuning using machine learning”. In: *ACM Computing Surveys (CSUR)* 51.5 (2018), pp. 1–42.

[4] Steven J Beaty, Scott Colcord, and Philip H Sweany. “Using genetic algorithms to fine-tune instruction-scheduling heuristics”. In: *Proceedings of the international conference on massively parallel computer systems*. 1996.

[5] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. “Neural code comprehension: A learnable representation of code semantics”. In: *Advances in Neural Information Processing Systems* 31 (2018), pp. 3585–3597.

[6] David Bernstein and Michael Rodeh. “Global instruction scheduling for superscalar machines”. In: *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. 1991, pp. 241–255.

[7] Yngvi Bjornsson and Hilmar Finnsson. “Cadiaplayer: A simulation-based general game player”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 1.1 (2009), pp. 4–15.

[8] Alexander Brauckmann et al. “Compiler-based graph representations for deep learning models of code”. In: *Proceedings of the 29th International Conference on Compiler Construction*. 2020, pp. 201–211.

[9] Gregory J Chaitin. “Register allocation & spilling via graph coloring”. In: *ACM Sigplan Notices* 17.6 (1982), pp. 98–101.

[10] Corinna Cortes and Vladimir Vapnik. “Support-vector networks”. In: *Machine learning* 20.3 (1995), pp. 273–297.

[11] Chris Cummins et al. “PROGRAML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations”. In: *International Conference on Machine Learning*. PMLR. 2021, pp. 2244–2253.

[12] Dibyendu Das, Shahid Asghar Ahmad, and Kumar Venkataramanan. “Deep Learning-based Hybrid Graph-Coloring Algorithm for Register Allocation”. In: *arXiv preprint arXiv:1912.03700* (2019).

[13] Harris Drucker et al. “Support vector regression machines”. In: *Advances in neural information processing systems* 9 (1997), pp. 155–161.

[14] Joseph A. Fisher. “Trace scheduling: A technique for global microcode compaction”. In: *IEEE transactions on computers* 30.07 (1981), pp. 478–490.

[15] Philip B Gibbons and Steven S Muchnick. “Efficient instruction scheduling for a pipelined architecture”. In: *Proceedings of the 1986 SIGPLAN symposium on Compiler construction*. 1986, pp. 11–16.

[16] James R Goodman and W-C Hsu. “Code scheduling and register allocation in large basic blocks”. In: *ACM International Conference on Supercomputing 25th Anniversary Volume*. 1988, pp. 88–98.

[17] Susan L Graham, Peter B Kessler, and Marshall K McKusick. “Gprof: A call graph execution profiler”. In: *ACM Sigplan Notices* 17.6 (1982), pp. 120–126.

[18] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. “Speech recognition with deep recurrent neural networks”. In: *2013 IEEE international conference on acoustics, speech and signal processing*. Ieee. 2013, pp. 6645–6649.

[19] Ameer Haj-Ali et al. “NeuroVectorizer: end-to-end vectorization with deep reinforcement learning”. In: *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 2020, pp. 242–255.

[20] David Harris and Sarah Harris. *Digital design and computer architecture*. Morgan Kaufmann, 2010.

[21] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[22] J Heller. “Sequencing aspects of multiprogramming”. In: *Journal of the ACM (JACM)* 8.3 (1961), pp. 426–439.

[23] John L Hennessy and Thomas Gross. “Postpass code optimization of pipeline constraints”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5.3 (1983), pp. 422–448.

[24] Qijing Huang et al. “Autophase: Compiler phase-ordering for hls with deep reinforcement learning”. In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2019, pp. 308–308.

[25] Wen-Mei W Hwu et al. “The superblock: An effective technique for VLIW and superscalar compilation”. In: *Instruction-Level Parallelism*. Springer, 1993, pp. 229–248.

[26] Ajay Jain and Saman Amarasinghe. “Learning automatic schedulers with projective reparameterization”. In: *Proceedings of the ML-for-Systems Workshop at the 46th International Symposium on Computer Architecture (ISCA'19)*. 2019.

[27] Daniel R Kerns and Susan J Eggers. “Balanced scheduling: Instruction scheduling when memory latency is uncertain”. In: *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*. 1993, pp. 278–289.

[28] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[29] Levente Kocsis and Csaba Szepesvári. “Bandit based monte-carlo planning”. In: *European conference on machine learning*. Springer. 2006, pp. 282–293.

[30] Levente Kocsis, Csaba Szepesvári, and Jan Willemson. “Improved monte-carlo search”. In: *Univ. Tartu, Estonia, Tech. Rep* 1 (2006).

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012), pp. 1097–1105.

[32] David Landskov et al. “Local microcode compaction techniques”. In: *ACM Computing Surveys (CSUR)* 12.3 (1980), pp. 261–294.

[33] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation”. In: San Jose, CA, USA, Mar. 2004, pp. 75–88.

[34] SS Lavrov. “Store economy in closed operator schemes”. In: *USSR Computational Mathematics and Mathematical Physics* 1.3 (1962), pp. 810–828.

[35] Yann LeCun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[36] Henrique Lemos et al. “Graph colouring meets deep learning: Effective graph neural network models for combinatorial problems”. In: *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE. 2019, pp. 879–885.

[37] Jack L Lo and Susan J Eggers. “Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism”. In: *ACM SIGPLAN Notices* 30.6 (1995), pp. 151–162.

[38] Rahim Mammadli, Ali Jannesari, and Felix Wolf. “Static Neural Compiler Optimization via Deep Reinforcement Learning”. In: *arXiv preprint arXiv:2008.08951* (2020).

[39] Hongzi Mao et al. “Learning scheduling algorithms for data processing clusters”. In: *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 270–288.

[40] Amy McGovern, Eliot Moss, and Andrew G Barto. “Building a basic block instruction scheduler with reinforcement learning and rollouts”. In: *Machine learning* 49.2-3 (2002), pp. 141–160.

[41] Amy McGovern, Eliot Moss, and Andrew G Barto. “Scheduling straight-line code using reinforcement learning and rollouts”. In: *Computer Science Department Faculty Publication Series* (1999), p. 14.

[42] Charith Mendis et al. “Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks”. In: *International Conference on Machine Learning*. PMLR. 2019, pp. 4505–4515.

[43] Tomas Mikolov et al. “Efficient estimation of word representations in vector space”. In: *arXiv preprint arXiv:1301.3781* (2013).

[44] J Eliot B Moss et al. “Learning to schedule straight-line code”. In: *NIPS*. Vol. 97. 1997, pp. 929–935.

[45] Vinod Nair and Geoffrey E Hinton. “Rectified linear units improve restricted boltzmann machines”. In: *Icml*. 2010.

[46] John von Neumann. “Zur Theorie der Gesellschaftsspiele”. In: *Mathematische annalen* 100.1 (1928), pp. 295–320.

[47] Todd A Proebsting and Charles N Fischer. “Linear-time, optimal code scheduling for delayed-load architectures”. In: *ACM SIGPLAN Notices* 26.6 (1991), pp. 256–267.

[48] Frank Rosenblatt. “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65.6 (1958), p. 386.

[49] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. *Learning internal representations by error propagation*. Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[50] Tyrel Russell. “Learning Instruction Scheduling Heuristics from Optimal Data”. MA thesis. University of Waterloo, 2006.

[51] Philip John Schielke. *Stochastic Instruction Scheduling*. Rice University, 2000.

[52] Ravi Sethi and Jeffrey D Ullman. “The generation of optimal code for arithmetic expressions”. In: *Journal of the ACM (JACM)* 17.4 (1970), pp. 715–728.

[53] Richard Socher et al. “Recursive deep models for semantic compositionality over a sentiment treebank”. In: *Proceedings of the 2013 conference on empirical methods in natural language processing*. 2013, pp. 1631–1642.

[54] Darko Stefanovic. “The character of the instruction scheduling problem”. In: *Quantum* 2122.25693 (1997), p. 132139.

[55] Gregory Tarsy and Michael J Woodard. *Method and apparatus for optimizing cost-based heuristic instruction scheduling*. US Patent 5,367,687. Nov. 1994.

[56] Linda Torczon and Keith Cooper. *Engineering A Compiler*. 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. ISBN: 012088478X.

[57] Zheng Wang and Michael FP O'Boyle. "Mapping parallelism to multi-cores: a machine learning based approach". In: *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 2009, pp. 75–84.

[58] Zheng Wang and Michael O'Boyle. "Machine learning in compiler optimization". In: *Proceedings of the IEEE* 106.11 (2018), pp. 1879–1901.

A. Instruction Clusters

A.1. AArch64

Cluster	Instructions
ADD	ADDSWri, ADDSWrr, ADDSXri, ADDWri, ADDWrr, ADDWrs, ADDWrx, ADDXri, ADDXrr, ADDXrs, ADDXrx
ADDVv	ADDVv4i32v
ADDv	ADDv16i8, ADDv2i32, ADDv2i64, ADDv4i16, ADDv4i32
ADJCALLSTACKDOWN	ADJCALLSTACKDOWN
ADJCALLSTACKUP	ADJCALLSTACKUP
ADRP	ADRP
AND	ANDSWri, ANDSWrr, ANDWri, ANDWrr, ANDWrs, ANDXri, ANDXrr
ANDv	ANDv16i8
ASRV	ASRVWr, ASRVXr
BFM	BFMWri, BFMXri
BIC	BICSWrr, BICSXrr, BICWrr, BICWrs
BICv	BICv8i8
BSPv	BSPv16i8
CLZ	CLZWr
CMEQv	CMEQv16i8rz, CMEQv2i32rz, CMEQv4i16rz, CMEQv4i32, CMEQv4i32rz
CMGEv	CMGEv2i32rz, CMGEv2i64
CMGTv	CMGTv2i64
CMLTv	CMLTv2i32rz
COPY	COPY
COPY_TO_REGCLASS	COPY_TO_REGCLASS
CPYi64	CPYi64
CSEL	CSELWr, CSELXr
CSINC	CSINCWr, CSINCXr
CSINV	CSINVWr, CSINVXr
CSNEG	CSNEGWr
DUPv	DUPv2i32gpr, DUPv2i64gpr, DUPv2i64lane, DUPv4i32gpr, DUPv4i32lane, DUPv8i16gpr
EOR	EORWri, EORWrr, EORWrs, EORXri, EORXrr
EORv	EORv16i8, EORv8i8
EXTR	EXTRWrri
EXTRACT_SUBREG	EXTRACT_SUBREG

Cluster	Instructions
EXTv	EXTv16i8
FABD64	FABD64
FABS	FABSDr, FABSSr
FADD	FADDDrr, FADDSrr
FADDv	FADDv2f64, FADDv4f32
FCMGTv	FCMGTv2f64, FCMGTv4i32rz
FCMP	FCMPDri, FCMPDrr, FCMPSri, FCMPSrr
FCSEL	FCSELDrrr, FCSELSrrr
FCVTD	FCVTDSr
FCVTLv	FCVTLv2i32, FCVTLv4i32
FCVTMSUW	FCVTMSUWDr
FCVTMUUW	FCVTMUUWDr
FCVTNv	FCVTNv2i32, FCVTNv4i32
FCVTS	FCVTSDr
FCVTZSSX	FCVTZSSXSri
FCVTZSUW	FCVTZSUWDr, FCVTZSUWSr
FCVTZSv	FCVTZSv2f64
FDIV	FDIVDrr, FDIVSrr
FDIVv	FDIVv2f64, FDIVv4f32
FMAX	FMAXDrr, FMAXSrr
FMAXNM	FMAXNMDrr
FMIN	FMINDrr, FMINSrr
FMINNM	FMINNMDrr
FMOV	FMOVDi, FMOVSi
FMOVD0	FMOVD0
FMOVS0	FMOVS0
FMOVv	FMOVv2f64_ns, FMOVv4f32_ns
FMUL	FMULDrr, FMULSrr
FMULv	FMULv1i32_indexed, FMULv1i64_indexed, FMULv2f32, FMULv2f64, FMULv2i32_indexed, FMULv2i64_indexed, FMULv4f32
FNEG	FNEGDr, FNEGSr
FNEGv	FNEGv2f64, FNEGv4f32
FNMUL	FNMULSrr
FRINTM	FRINTMDr
FSQRT	FSQRTDr
FSUB	FSUBDrr, FSUBSrr
FSUBv	FSUBv2f64, FSUBv4f32
IMPLICIT_DEF	IMPLICIT_DEF
INSERT_SUBREG	INSERT_SUBREG
INSv	INSvi32lane
LD1Rv	LD1Rv2d_POST, LD1Rv2s, LD1Rv2s_POST, LD1Rv4s
LD2Twov	LD2Twov16b, LD2Twov16b_POST, LD2Twov2d, LD2Twov2d_POST, LD2Twov4h, LD2Twov4s
LDRBBpost	LDRBBpost
LDRBBpre	LDRBBpre

Cluster	Instructions
LDRBBroW	LDRBBroW
LDRBBroX	LDRBBroX
LDRBBui	LDRBBui
LDRBroX	LDRBroX
LDRDpost	LDRDpost
LDRDpre	LDRDpre
LDRDui	LDRDui
LDRHHpost	LDRHHpost
LDRHHpre	LDRHHpre
LDRHHroW	LDRHHroW
LDRHHroX	LDRHHroX
LDRHHui	LDRHHui
LDRHui	LDRHui
LDRQpost	LDRQpost
LDRQroX	LDRQroX
LDRQui	LDRQui
LDRSBWui	LDRSBWui
LDRSHWpost	LDRSHWpost
LDRSHWui	LDRSHWui
LDRSHXpost	LDRSHXpost
LDRSHXui	LDRSHXui
LDRSHoW	LDRSHWroW, LDRSHXroW
LDRSHoX	LDRSHWroX, LDRSHXroX
LDRSWpost	LDRSWpost
LDRSWui	LDRSWui
LDRSpost	LDRSpost
LDRSpre	LDRSpre
LDRSui	LDRSui
LDRWpost	LDRWpost
LDRWpre	LDRWpre
LDRWui	LDRWui
LDRXpost	LDRXpost
LDRXpre	LDRXpre
LDRXui	LDRXui
LDRoW	LDRDroW, LDRSWroW, LDRSroW, LDRWroW, LDRXroW
LDRoX	LDRDroX, LDRSWroX, LDRSroX, LDRWroX, LDRXroX
LDUR	LDURDi, LDURSWi, LDURSi, LDURWi, LDURXi
LDURBBi	LDURBBi
LDURHHi	LDURHHi
LDURHi	LDURHi
LDURQi	LDURQi
LDURSH	LDURSHWi, LDURSHXi
LSLV	LSLVWr, LSLVXr
LSRV	LSRVWr, LSRVXr
MADD	MADDWrrr, MADDXrrr

Cluster	Instructions
MOVID	MOVID
MOVIv	MOVIv16b_ns, MOVIv2d_ns, MOVIv2i32, MOVIv2s_msl, MOVIv4i16, MOVIv4i32, MOVIv4s_msl, MOVIv8b_ns, MOVIv8i16
MOVaddr	MOVaddr
MOVaddrCP	MOVaddrCP
MOVi32imm	MOVi32imm
MOVi64imm	MOVi64imm
MSUB	MSUBWrrr
MULv	MULv2i32, MULv4i32, MULv8i16
MVNIv	MVNIv4i32, MVNIv4s_msl
NEGv	NEGv2i32, NEGv4i32
NOTv	NOTv16i8, NOTv8i8
ORN	ORNWrr, ORNWrs, ORNXrr
ORR	ORRWri, ORRWrr, ORRWrs, ORRXri, ORRXrr, ORRXrs
ORRv	ORRv16i8, ORRv8i8
REG_SEQUENCE	REG_SEQUENCE
REV	REVWr
REV64v	REV64v2i32, REV64v4i32
SBFM	SBFMWri, SBFMXri
SCVTFUW	SCVTFUWDri, SCVTFUWSri
SCVTFUX	SCVTFUXDri, SCVTFUXSri
SCVTFv	SCVTFv1i32, SCVTFv1i64, SCVTFv2f64, SCVTFv4f32
SDIV	SDIVWr, SDIVXr
SHLv	SHLv2i32_shift, SHLv2i64_shift, SHLv4i32_shift
SMADDLrrr	SMADDLrrr
SMAXVv	SMAXVv4i32v
SMAXv	SMAXv4i32
SMULHrr	SMULHrr
SMULLv	SMULLv4i16_v4i32
SSHLLv	SSHLLv2i32_shift, SSHLLv4i16_shift, SSHLLv4i32_shift, SSHLLv8i16_shift
SSHRv	SSHRv2i32_shift
STRBBpost	STRBBpost
STRBBpre	STRBBpre
STRBBroW	STRBBroW
STRBBroX	STRBBroX
STRBBui	STRBBui
STRDpost	STRDpost
STRDpre	STRDpre
STRDui	STRDui
STRHHroW	STRHHroW
STRHHroX	STRHHroX
STRHHui	STRHHui
STRQpost	STRQpost
STRQpre	STRQpre
STRQroX	STRQroX

Cluster	Instructions
STRQui	STRQui
STRSpost	STRSpost
STRSui	STRSui
STRWpost	STRWpost
STRWpre	STRWpre
STRWui	STRWui
STRXpost	STRXpost
STRXpre	STRXpre
STRXui	STRXui
STRoW	STRDroW, STRWroW, STRXroW
STRoX	STRDroX, STRSroX, STRWroX, STRXroX
STUR	STURDi, STURSi, STURWi, STURXi
STURBBi	STURBBi
STURHHi	STURHHi
STURQi	STURQi
SUB	SUBSWri, SUBSWrr, SUBSWrs, SUBSWrx, SUBSXri, SUBSXrr, SUBSXrs, SUBSXrx, SUBWrr
SUBREG_TO_REG	SUBREG_TO_REG
SUBv	SUBv2i32, SUBv4i32, SUBv8i16
UADDLv	UADDLv8i16_v4i32, UADDLv8i8_v8i16
UBFM	UBFMWri, UBFMXri
UCVTFUW	UCVTFUWDri
UCVTFUX	UCVTFUXDri, UCVTFUXSri
UCVTFv	UCVTFv1i32, UCVTFv1i64
UDIV	UDIVWr, UDIVXr
UMADDLrrr	UMADDLrrr
UMLALv	UMLALv4i16_v4i32
UMOVv	UMOVvi32, UMOVvi64
UMULHrr	UMULHrr
USHLLv	USHLLv16i8_shift, USHLLv4i16_shift, USHLLv8i8_shift
USHLv	USHLv2i32, USHLv4i32
USHRv	USHRv4i32_shift, USHRv8i16_shift
XTNv	XTNv2i32, XTNv4i16, XTNv4i32, XTNv8i8

A.2. NEC SX-Aurora TSUBASA

Cluster	Instructions
ADDSL	ADDSLrr
ADDSWSX	ADDSWSXri, ADDSWSXrm, ADDSWSXrr
ADDSWZX	ADDSWZXrm
ADJCALLSTACKDOWN	ADJCALLSTACKDOWN
ADJCALLSTACKUP	ADJCALLSTACKUP
AND	ANDri, ANDrm, ANDrr
BCFLari_t	BCFLari_t
BRCFD	BRCFDir, BRCFDrr
BRCFL	BRCFLir, BRCFLrr
BRCFLa	BRCFLa
BRCFS	BRCFSir, BRCFSrr
BRCFW	BRCFWir, BRCFWrr
BSWP	BSWPri
CMOVD	CMOVDrm, CMOVDrr
CMOVL	CMOVLrm, CMOVLrr
CMOVS	CMOVSrr
CMOVW	CMOVWrm, CMOVWrr
CMPSL	CMPSLrr
CMPSWSX	CMPSWSXrr
CMPUL	CMPULir, CMPULrr
CMPUW	CMPUWir, CMPUWrr
COPY	COPY
COPY_TO_REGCLASS	COPY_TO_REGCLASS
CVTDL	CVTDLr
CVTDQ	CVTDQr
CVTDS	CVTDSr
CVTDW	CVTDWr
CVTLD	CVTLDr
CVTQD	CVTQDr
CVTSD	CVTSDr
CVTSW	CVTSWr
CVTWDSX	CVTWDSXr
CVTWSSX	CVTWSSXr
DIVSL	DIVSLrr
DIVSWSX	DIVSWSXir, DIVSWSXrm, DIVSWSXrr
DIVUL	DIVULrm, DIVULrr
DIVUW	DIVUWrm, DIVUWrr
EXTRACT_SUBREG	EXTRACT_SUBREG
FADDD	FADDDrm, FADDDrr
FADDQ	FADDQrr
FADDS	FADDSrm, FADDSrr
FCMPD	FCMPDrr

Cluster	Instructions
FCMPS	FCMPSrr
FDIVD	FDIVDrr
FDIVS	FDIVSrr
FMAXD	FMAXDrr
FMAXS	FMAXSrr
FMIND	FMINDrr
FMINS	FMINSrr
FMULD	FMULDrm, FMULDrr
FMULQ	FMULQrr
FMULS	FMULSrr
FSUBD	FSUBDrr
FSUBQ	FSUBQrr
FSUBS	FSUBSrr
GETSTACKTOP	GETSTACKTOP
IMPLICIT_DEF	IMPLICIT_DEF
INSERT_SUBREG	INSERT_SUBREG
LD	LDrii, LDrri
LD1BSX	LD1BSXrii, LD1BSXrri
LD1BZX	LD1BZXrii, LD1BZXrri
LD2BSX	LD2BSXrii, LD2BSXrri
LD2BZX	LD2BZXrii, LD2BZXrri
LDLSX	LDLSXrii, LDLSXrri
LDLZX	LDLZXrii, LDLZXrri
LDU	LDUrii, LDUrri
LDZ	LDZr
LEA	LEArii, LEArri
LEASL	LEASLrii, LEASLrri
LEASLz	LEASLzii
LEAz	LEAzii
MAXSL	MAXSLrr
MAXSWSX	MAXSWSXrr
MINSL	MINSLrr
MINSWSX	MINSWSXrr
MULSL	MULSLri, MULSLrm, MULSLrr
MULSWSX	MULSWSXri, MULSWSXrm, MULSWSXrr
NND	NNDrm, NNDrr
OR	ORim, ORri, ORrr
RET	RET
SLAWSX	SLAWSXmr, SLAWSXri, SLAWSXrr
SLL	SLLmr, SLLri, SLLrr
SRAL	SRALri, SRALrr
SRAWSX	SRAWSXri, SRAWSXrr
SRL	SRLri, SRLrr
ST	STrii, STrri
ST1B	ST1Brii, ST1Brri

Cluster	Instructions
ST2B	ST2Brii, ST2Brri
STL	STLrii, STLrri
STU	STUrii, STUrri
SUBSL	SUBSLir, SUBSLrr
SUBSWSX	SUBSWSXir, SUBSWSXrr
XOR	XORri, XORrm, XORrr