Universiteit Leiden

Master Computer Science

Exhaustive Performance Exploration of Instruction Ordering on OOO-Processors

Name:	Rens Dofferhoff
Student ID:	s1664042
Date:	07/05/2023
Specialisation:	Advanced Computing & Systems
1st supervisor:	Dr. K.F.D. Rietveld
2nd supervisor:	Dr. T.P. Stefanov

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Compiler machine code generation backends consist of highly sophisticated methods to fine-tune the generated machine code. For static in-order and VLIW architectures the order of the instructions in the machine code is especially important, because this order determines the efficiency by which the program can be executed on the target pipeline, as such pipelines cannot alter the instruction order at run-time. On the contrary, the exact instruction order within a basic block is not always considered as important for out-of-order (OOO) architectures, given that OOO processors are free to alter instruction order at run-time to better utilize available functional units and masquerade small unexpected delays such as L1 cache misses. In this thesis, we investigate the importance of instruction order to performance on modern OOO processors. To do so, we present a tool, based on LLVM, for the exhaustive exploration of instruction schedule permutations for simple kernels. Our findings indicate that different instruction orders indeed lead to performance differences and that schedules can be found that perform better compared to specifically tuned LLVM schedulers. The magnitude of these differences depends on the exact kernel but the potential for improvement is as high as 29% for specific kernels. The extension of this work may lead to improved instruction schedulers and a methodology to further optimize compute kernels for specific microarchitectures.

Contents

Chapter 1

Introduction

To compile code into efficient machine code, modern compilers use hundreds of optimization passes across both the front end, dealing with architecture agnostic intermediate representations, as well as the backend, translating to and optimizing for specific instruction set architectures (ISA). Machine code may even be optimized for specific microarchitectures that implement an ISA by considering properties related to the pipeline such as layout and depth. Generally such work is left to the compiler but hand-tuning critical subroutines is still common place [10]. Besides selecting the instructions from the target ISA, the compiler has to order the instructions. A specific instruction ordering is called a schedule and the process of generating or altering the instruction order is often called instruction scheduling. Multiple different schedules of the required operations may yield the same valid result. Instruction selection and scheduling is critical for performant execution of code on in-order and VLIW architectures, since architectures have no logic that can alter the instruction order. While aspects such as port contention are viewed as important for modern OOO-processors, the significance of instruction order is often not considered. This might seem logical since these processors can alter the order freely over a wide instruction window. Many might at first glance speculate there to be little to no effect at all.

We were inspired by an initial finding by IJpelaar [16] that showed some effect for a specific kernel, as well as an anecdote by Estes [8] about his surprise on the effects of scheduling on the OOO ARM-A57 CPU. In this thesis, we wish to investigate further on a wider range of kernels and machines. To this end, we created a tool that automatically generates and gathers performance data for all valid instruction schedules for a given compute kernel. With this tool we hope to answer our research questions:

- How relevant is instruction scheduling to performance on modern OOO-Processors?
- How do contemporary instruction schedulers fare in the scope of the complete solution space?
- What explains the difference in performance of instruction schedules on OOO-Processors?

We discovered significant spread in performance among instruction schedules for multiple kernels, indicating that instruction scheduling is still of importance for OOO-processors. We find that the

schedules generated by LLVM are not always optimal and significant improvements of over 29%) can be made for specific kernels.

1.1 Contributions

This thesis makes the following contributions:

- An open source tool based on the LLVM framework that generates all valid instruction order permutations of the the input kernel.
- The resulting performance data of our explorative study on all valid schedules of 11 different compute kernels on the Zen2 and Ivy Bridge micro architectures.
- Our results, which show that performance variations among instruction schedules do exist on OOO-processors. Additionally, our results show that the schedules generated by LLVM are not always optimal and improvement as high as 29% can be found for specific kernels.

1.2 Thesis Overview

This chapter contains the introduction of this thesis; Chapter 2 discusses background information needed to understand this thesis; Chapter 3 describes previous research related to this subject; Chapter 4 discusses the overall program design and methods developed in this research in depth; Chapter 5 gives information on how the methods and design are implemented and what issues were encountered in their implementation; Chapter 6 shows the results of system scans and performance analysis; Chapter 7 describes the limitations on the methods, their implementations and any gathered results; Chapter 8 summarizes and concludes our findings; Chapter 9 makes suggestions for further research.

Chapter 2

Background

In this chapter, the necessary background knowledge needed to understand this thesis is discussed. Definitions and explanations of various terms surrounding compilation, instruction scheduling and microarchitecture are given.

2.1 Compilation

Human readable code written in programming languages such as C, Fortran or Rust has to be transformed so that a target processor may execute it. The list of all the operations, or machine instructions, a processor is able to execute, along with various other architecture details are documented in the instruction set architecture (ISA). Programs that translate human readable code into a series of machine instructions are called compilers. The task of a compiler is to not only generate a series of instructions that are semantically equivalent to the specified program but also to ensure the resulting program runs efficiently. Contemporary compilers are extensive modular frameworks that divide the compilation work across multiple components, such as language specific frontends, intermediate representation optimizers and target specific machine code generators.

LLVM [19] is a prominent framework on which a lot of compilers are build. As shown in Figure 2.1, a LLVM based compiler is roughly divided in three components. A program language specific frontend converts the input program into the LLVM intermediate representation (LLVM IR). A common optimizer, present in what is sometimes called the middle-end, contains many passes that optimize the intermediate representation. A pass is a function that takes the current program representation as input, makes changes and then outputs the new version. Since these optimization passes operate on an intermediate representation they are agnostic to both the programming language used and the ISA that will ultimately be targeted. Finally, the backend takes the optimized LLVM IR and translates it into the instructions available in the target ISA. The backend consists of many phases including target-specific optimizations passes.

This structure is very useful. The creation of a new programming language only requires a new front-end that converts to LLVM IR. The newly formed language compiler can immediately make

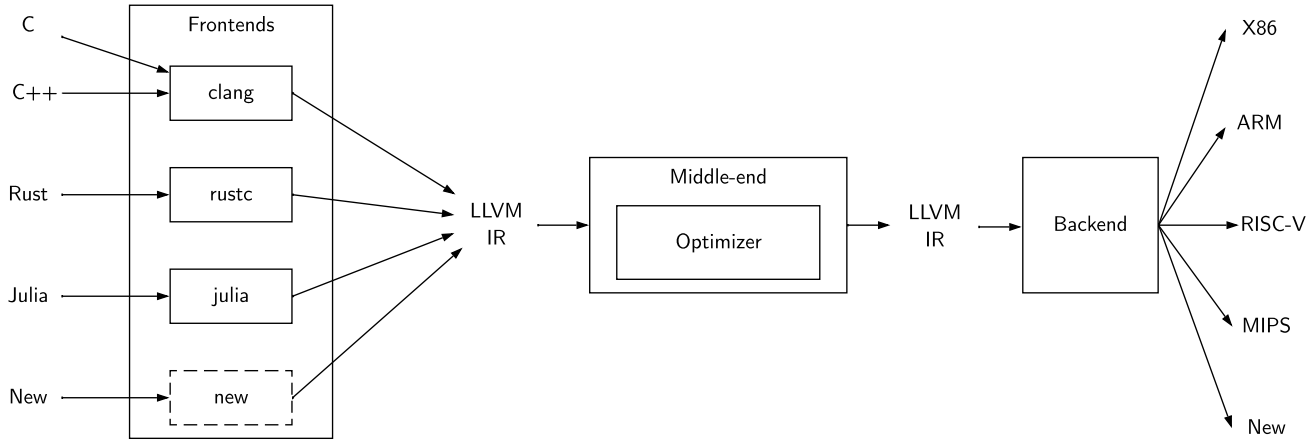

Figure 2.1: High level overview of the LLVM based compilation

use of all the optimizations and possible targets present in LLVM. Similarly, the creation of a new target ISA only requires the new target to be defined within the LLVM backend. All code written in languages with a LLVM frontend can immediately be compiled for the new target.

2.1.1 LLVM Backend

In our work, we are mostly concerned with the LLVM backend. The backend contains many passes to step by step convert LLVM IR into a legal series of instructions of the target ISA. The code generation pipeline is shown in Figure 2.2 and consist of a number of main passes highlighted in red. Passes can be target specific, only being present when compiling for a specific target, or target agnostic.

The instruction selection pass takes the LLVM IR and picks target instructions that match the semantics of the IR operations. The IR is in a form called single static assignment (SSA). In this form each variable may only be assigned once. The output of the instruction selection pass is also in this form. It is no longer target agnostic but a target specific SSA form containing mostly target instructions. After instruction selection, IR variables take the form of virtual registers (vregs). There can be infinitely many vregs, yet targets have finite register space. Coalescing vregs and assigning physical registers is done by one or more register allocation passes. If there is not enough register space a vreg is spilled, this means main memory is used instead of a register. After the register allocation (RA) phase the program is no longer in SSA form and consists mostly of valid target instructions. There are often two scheduling passes, one before and one after register allocation, often called the pre- and post-RA schedulers. These passes determine the order of the instructions. The final code emission passes generate assembly code or an executable binary object.

In between these main passes, many optimization passes are present and custom passes, written by users, can be inserted at various places within the code generation backend.

Just in time compilation (JIT) is a technique used to compile code at runtime, just before its needed.

The LLVM framework is used both for static compilers as well as JIT compilers. JIT compilers can be implemented via the LLVM ORC JIT interface. It allows users to compile LLVM IR at will from a running process, allowing the newly generated machine code to be executed immediately.

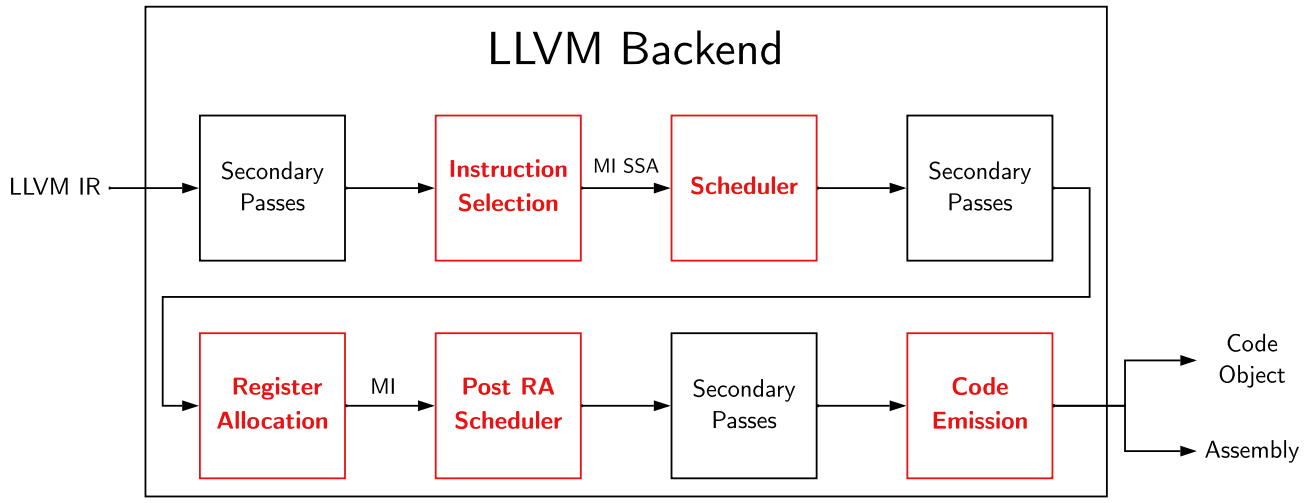

Figure 2.2: Various Passes of the LLVM backend code generation pipeline. Main passes are highlighted in red.

2.1.2 Instruction Scheduling

In this thesis, we wish to discover the relevancy of instruction orders on the performance of programs running on OOO-processors. The task of assigning the order of instructions in the final binary code object belongs to the compiler's instruction scheduler. Multiple instruction orders, or schedules, may be valid. Scheduling usually happens within a basic block, a straight sequence of instructions without branches except one at the end.

Naturally, we cannot arbitrarily change the positions of instructions and retain the same program semantics. An instruction may be dependent on the results of previous instructions. This is called a true or flow dependency. Besides true dependencies there are false dependencies that are a consequence of limited register space. There are two types of false dependencies: output- and anti-dependencies. If two instructions write their output to the same register there is an output dependency. If one instruction uses a register to write its output that another used previously as input but the instructions are not related by a flow dependency there is a anti-dependency. Both these cases could be resolved by register renaming, within the constraints of the limited register space.

The dependencies between instructions in a basic block can be captured in directed acyclic graph (DAG). Instruction Schedulers must transform these DAGs into valid efficient linear instructions sequences (schedules). Examples of flow-, anti- and output-dependence as well as register renamed versions are shown in Figure 2.3 along with the DAGs that represent these examples. An instruction may be scheduled the moment all connected instruction vertexes in the DAG have been scheduled.

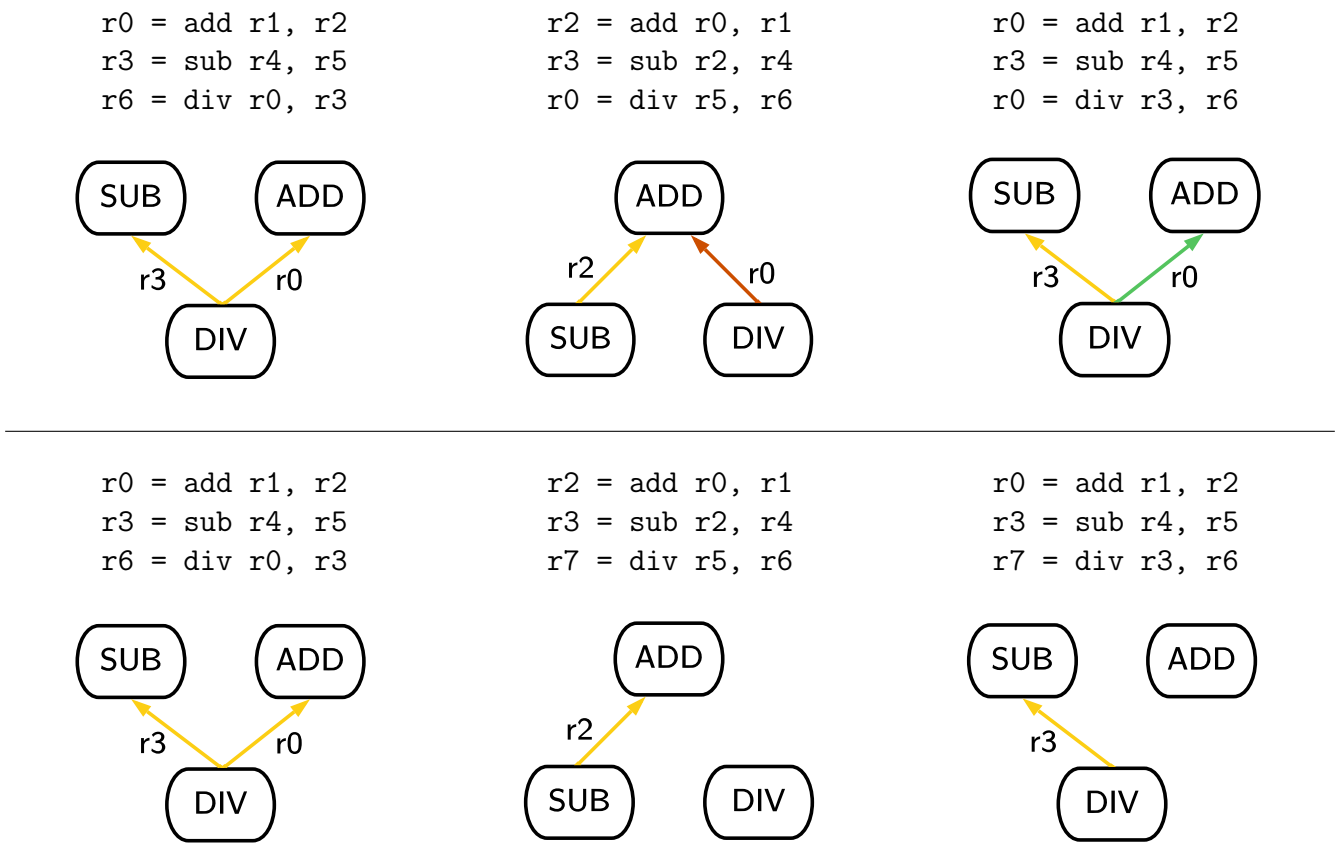

Figure 2.3: Examples of flow, anti and output dependence, marked by orange, red, green respectively, before and after register renaming.

Therefore, there may be multiple possible linear sequences when multiple instructions are ready to be scheduled at the same time. In the second example of Figure 2.3 the add instruction has to be scheduled before both the subtraction and division instructions to retain the correct semantics. However, in the register renamed version either the add or division instruction may be scheduled first.

2.2 Microarchitecture

An instruction set architecture (ISA) specifies what instructions should be implemented, which resources should be present and how they are accessible. As such it defines the interface between hardware and software and can be seen as a 'contract' with software developers. A microarchitecture (µarch) is an implementation of one or more ISAs. The exact implementation can vary between architectures. For example, the X86 ISA has been implemented in Intel 8086 in 1978 all the way to Alder Lake chips released in 2021. The implementation changed significantly over that period, while backwards compatibility with the ISA has been maintained.

The first µarchs were simple in-order machines that loaded instructions from memory and execute

these one at a time in the order they are present in memory. Registers are used to accumulate the results of computations. Data paths for such µarch are very long keeping clock frequency low. In order to speedup the execution pipelining was employed. In this case, execution of instructions is divided into multiple stages. The classic example is the 5 stage pipeline: instruction fetch, decode, execute, memory access, write back. Instead of a single instruction using all resources at a time the different stages may be used by different instructions simultaneously, partially overlapping their execution and shortening the datapath length. This can significantly increase performance. Unfortunately pipelines may stall to resolve hazards such as data hazards caused by flow dependencies, where the necessary operands calculated by previous instruction are not yet available. Instructions behind the stall have to wait to progress. Architecturally stalls might be mitigated by providing data forwards between stages so operands become available before write back. In these systems proper instruction scheduling is important, some schedules produce less stalls by putting instructions doing useful work between instructions that would otherwise cause a stall. The scheduler needs information on the pipeline, latencies, forwards, etc to determine good schedules.

The next step from pipelines are super scalar pipelines where multiple instruction may be in flight at a time. In its simplest form one simply implements two equivalent pipelines through which instructions flow at the same time along with more complex hazard detection logic. The multiple pipelines may or may not share forwards and resources such as functional units. The hazard detection logic complexity grows quickly and not all resources may be used optimally. Due to this limitation Very long Instruction Word or VLIW-architectures were considered. In such architectures the burden of hazard detection is moved from the processor to the compiler. It is left to the compiler to explicitly note which instructions may be executed in parallel by combining them in an elongated instruction word. This complicates instruction scheduling massively.

2.3 Dynamically Scheduled Architectures

Up till now all the mentioned architectures are static in-order, meaning the instruction scheduler of the compiler determines the instruction order. Out-of-order (OOO) architectures attempt to maximize resource utilization by allowing instructions to execute in a different order they were dispatched from the decode stage. A scheduling system on the chip itself dynamically alters the execution order during runtime. If an instruction stalls on a data dependency or a cache miss other instructions behind it may still progress. Furthermore, if resources are available instructions without inter dependencies can be executed at the same time. So, on these architectures instructions may be executed in an order different from the instruction schedule determined by the compiler.

An example of a modern OOO-architecture is shown Figure 2.4. In OOO-architectures functional units such as ALUs, floating point units, address generators are bundled behind multiple execution ports with either a separate or unified queue of instructions in front of it. The storage spaces of these queues are traditionally called reservation stations and contain the instructions along with its operands if available. If an operand becomes available on the bus these are added to the stations that require them. When all operands are available the instruction can be executed as soon as the functional unit is available. The reservation stations form an instruction window in which instruction order is freely altered by the architecture logic. The size of this window varies between

architectures (54 in Figure 2.4). The needed scheduling logic is intensive in both transistor count and power usage but increases instruction level parallelism. These architectures often have additional register renaming logic along with large register files to resolve anti- and output-dependencies. Static scheduling by the compiler cannot account for unexpected delays such as those generated by cache misses. In contrast, the dynamic instruction reordering of OOO-processor will alter the order during runtime to continue use full work during such delays.

A re-order buffer is present so that the results of this out-of-order execution are committed in-order guaranteeing correctness. The size of the re-order buffer is 168 units in Figure 2.4. Speculative execution using branch predictors as well as simultaneous multi threading (SMT) are common features of OOO-architectures. Implementation of these features are an extension on the logic needed for out-of-order execution. An interesting note on X86 along with other complex instruction set architectures is that instructions are often transformed to one or more RISC-like µops. Programmers cannot use µops directly. This limits instruction scheduling by the compiler compared to many RISC architectures.

2.3.1 Performance Counters

Modern processors often contain many performance counters that users may use to measure and compare code performance. Counters may reflect various performance features such instruction and data cache misses, stalls at various points of the pipeline and retired µops. These counters for various architectures can be accessed via APIs such as Perf and PAPI [22, 5]. Within this thesis, we use these counters to collect performance data on the schedules we generate. We compare the data to see if there are differences between schedules. It would for example be very interesting to see if one schedule results in more data cache misses compared to another schedule. Such differences may explain any measured differences in runtime.

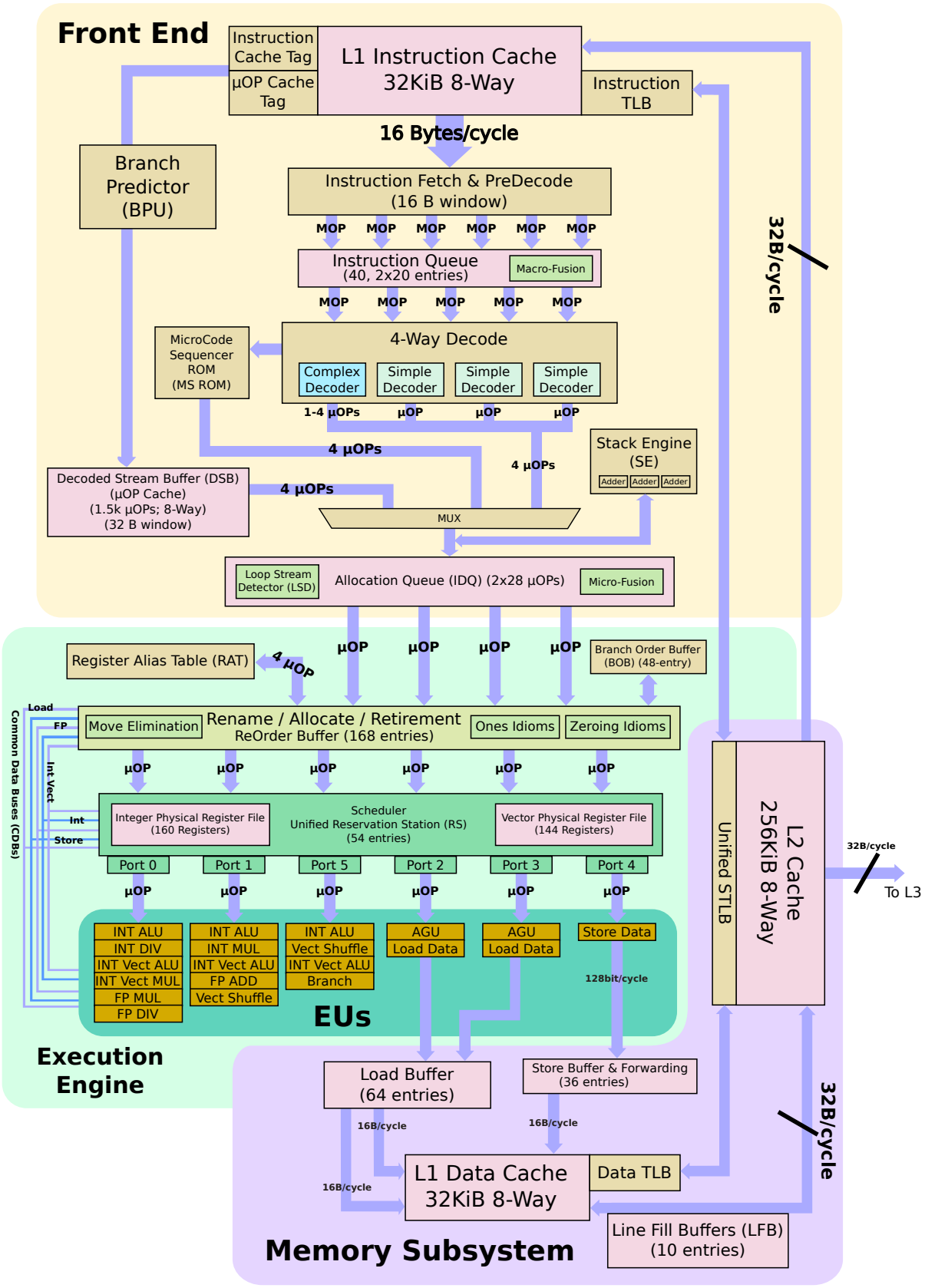

Figure 2.4: Overview of the Intel Ivy Bridge Architecture [1]

Chapter 3

Related Work

Inspired by the initial findings of an earlier project which showed some differences in performance on OOO-processors among schedules of a single kernel [16], as well as an anecdote by [8] concerning his surprise on the effects of scheduling on the OOO ARM Cortex-A57, we wish to further investigate this phenomena on a wider range of kernels and machines.

Finding optimal schedules in the face of pipeline constraints is known to be an NP complete problem [13]. As such much work has been done in design of heuristics and methods to find good schedules in reasonable time. This work has often been focused on creating efficient schedulers for targets that forgo costly issue logic and rely on compiler scheduling, namely in-order and VLIW architectures [12, 20, 9]. [25] shows that an aggressive static scheduler can have detrimental effects to performance on a simulated dynamically scheduled system of the HSA-architecture. [14] discusses optimization techniques for the PA-8000 and discusses that certain scheduling decisions that would normally hide latency on a dependency chain may lead to a bottleneck due to specific constraints of this old architecture. A significant amount of work has been done designing schedulers and heuristics to employ complex pipeline models [4, 7, 3]. For example, the LLVM instruction schedulers [19] are driven by such pipeline models specified using TableGen.

Other works focus on theoretical hybrid approaches between static compiler scheduling and dynamic scheduling. In many of these works the effective instruction window remains wide, while issue logic may be simplified by pre-ordering instructions in hardware or using other techniques such as value prediction [21, 24, 15, 17, 23].

While some prior works give indications that scheduling decisions may affect performance of the program when executed on OOO-architectures, to the best of our knowledge there is no prior work that investigates this in depth. Also, no prior work fully explores the scheduling solution space of a variety of kernels on modern OOO-processors, or provides the tools for doing so. While there are works that compare the performance of various schedulers relative to each other [27, 18], often for statically scheduled target architectures, we have found no such work detailing an exhaustive analysis of instruction scheduler performance in the scope of all possible solutions. In this thesis, we describe a tool to perform an exhaustive exploration of the scheduling solution space as well as present an analysis of such an exploration on two x86 microarchitectures.

Chapter 4

Design Overview

We wish to investigate the influence of instruction order on the performance of execution on OOO-architectures. In order to do so, we need to generate and assess the performance of all valid schedules of provided compute kernels. To accomplish this we created a tool that traverses the full scheduling decision space and collects performance data on all the schedules. An overview of the tool is shown in Figure 4.1.

The program input includes a LLVM IR file containing the kernel and optional definitions of data and preparation function. The user can specify for which function and optionally basic blocks the design space is to be explored. The tool uses the LLVM ORC JIT interface along with additional machine function passes and a custom register allocator for the LLVM backend. The custom register allocator minimizes false dependencies so we explore the full design space, as discussed in Section 5.3.

The LLVM IR will first flow through the normal stages of code generation such as instruction selection where it is converted into Machine Instruction SSA form (MI SSA). A myriad of optimization and legalization passes follow. It will then go through the pre-register allocation machine instruction scheduler. Afterwards it will be processed by our custom register allocator pass discussed in Section 5.3.

To accomplish the generation of valid permutations, an additional scheduling pass is added as the second to last pass in the pipeline. We call this pass the pre-emit scheduler. It first determines the schedule regions. These are sequences of instructions to be scheduled. These sequences are often whole basic blocks ending on branch boundaries. Afterwards a schedule DAG for each of the regions is created. The DAGs contain the dependencies among the instructions as discussed in Section 2. The pre-emit scheduler forwards all scheduling decisions to the schedule permuter component of the main program that maintains a scheduling decision tree. At each step the permuter can choose from of a list of valid instructions for which the dependencies have been satisfied. After each scheduling decision, this set is recalculated. This way it is trivial to dictate how to traverse the solution space and accumulate performance data tied to the decision nodes. The accumulated data can later be used to learn which decisions influence performance and also to express the relative importance of the decision. Using this data, we aim to draw conclusions why certain schedules exhibit better performance.

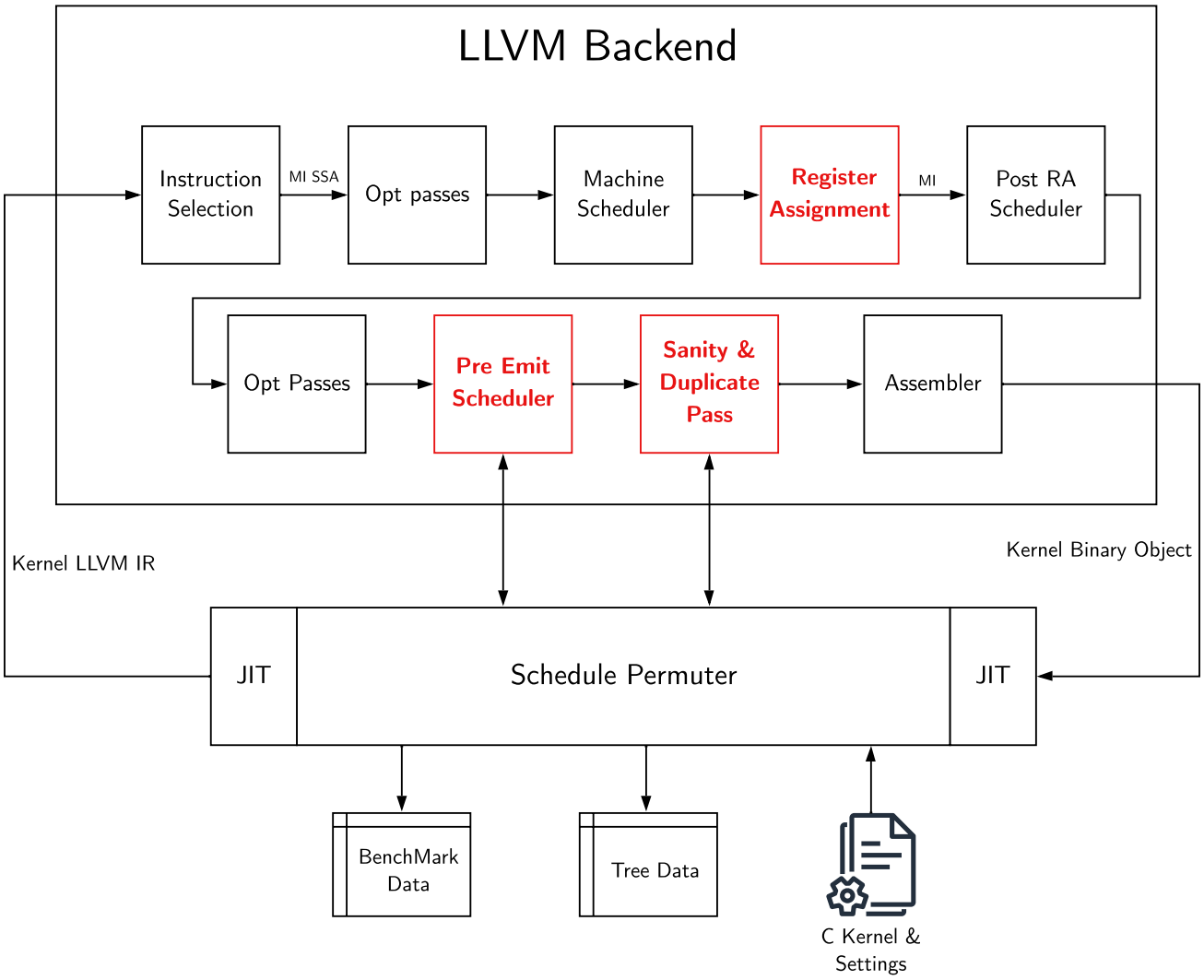

Figure 4.1: Overview of tool components. Red items represent passes we added or altered.

Each node of the scheduling decision tree consists of the chosen instruction, accumulated performance data, a list of child nodes and a reference to the parent node. An example of a processed scheduling decision tree is shown in Figure 6.5, the number in the node represents the instruction scheduled. The color of the node represents average runtime of all resulting schedules, represented by the leaves of the tree. One can infer that the first scheduling decision between scheduling instruction 7 or 1 is not a significant decision.

The pre-emit scheduler is followed by a sanity check pass that checks for duplicates. It counts the number of all present instructions, these must match with all the other schedules. The pass stores a hash for each schedule in a hash set. The pass will notify the user if any duplicates are generated. Duplicate schedules would point to an issue in code generation and the user is notified by an error.

The generated code object is linked and loaded by the LLVM ORC JIT interface. The user specified preparation function is called to initialize any data used by the compiled compute kernel. The program will call the compute kernel multiple times before benchmarking in order to promote

residence in instruction and data caches as well as training branch predictors. Subsequently, the program will measure the specified performance features using PAPI [22] over a specified number of iterations. Such measurements are taken a specified amount of times for each generated schedule. The results are written to an output file for later analysis.

As described in Section 5.5, care is taken so that interrupts, context switches and multi-threading do not disturb measurements. The kernels are tested in isolation on the same dedicated singular core. This way we ensure our experiments have a better chance to find performance differences among instructions schedules by reducing noise in our measurements that may be introduced by the operating system and other processes. If significant performance differences exist between instruction schedules an interesting direction of future work would be to test the robustness of instruction schedule performance when noise is introduced from a competing thread.

Chapter 5

Implementation

This chapter describes various details of the implementation of our tools, along with problems and choices encountered during implementation.

5.1 Pre- vs Post-Register Allocation Scheduling

At the onset of our research, we investigated the merits of making scheduling decisions before or after register allocation. A problem with scheduling after register allocation is the creation of additional false anti-, output- or artificial dependencies by the register allocator. These dependencies lock in the relative order of instructions and therefore preclude the creation of schedule permutations that would have been valid without their presence. LLVM provides a simple mechanism to alter the scheduler policy used at the existing pre-register allocation scheduler pass that could be used for our purposes. Initial investigation into using pre-register allocation scheduling yielded good results. Unfortunately, for certain kernels and compilation parameters it was discovered that later passes would often alter instruction order, creating duplicates and limiting the amount of permutations significantly. Even worse, instructions may be altered differently depending on the initial scheduling by the pre-register allocation scheduler. This implies that some permutations end up as a different set of instructions leading to erroneous results.

For the sake of generality we decided to add an additional scheduling pass, the pre-emit scheduler, at the very end of the pipeline, as is shown in Figure 4.1. This way there are no later passes that alter the set of instructions nor our instruction orders. We solved the issue of additional false dependencies by creating a custom instruction scheduler, described in Section 5.3, that does not introduce false dependencies. Using this method we can freely generate all valid schedules of any kernel compiled with any parameters without influence of later passes.

5.2 Schedule Permuter Interaction

The pre-emit scheduler, used to generate the final instruction order, is based of the *MachineSchedulerBase* pass along with a new *MachineSchedStrategy* that uses set function pointer hooks to forward calls of the *initialize*, *pickNode* and *releaseTopNode* functions. These functions are used to notify schedulers when new instructions are available for scheduling and allow for selection of an instruction. Using the forwarded calls the tool builds the scheduling decision tree and selects nodes to traverse the entire decision tree across multiple schedule generations. The decision tree can be traversed in any order desired. Performance data is accumulated in nodes of this decision tree along with a list of child nodes and a reference to the parent node. In order to not run out of memory when processing larger compute kernels, a node is written out to disk immediately when all its children have been evaluated. The chosen file format is Apache Parquet [26]. This is a compressed columnar data format that allows writing repeating data efficiently. It reduces file sizes significantly making analysis more manageable.

5.3 Register Allocator

Any false dependencies introduced by register allocation limit the amount of schedule permutations that may be generated by scheduler passes later on in the pipeline, as discussed in Section 5.1. To mitigate this issue we created a register allocator that does not introduce false dependencies in the instruction sequences we are permuting.

The allocator generates a dependency graph for the selected schedule regions. Before register allocation the instructions use virtual registers (vregs) instead of physical registers. We must map the unlimited number of vregs upon a limited set of physical registers without introducing false dependencies in the selected regions. We generate a map I that contains for each vreg the set of instructions in which it is used. We also generate the instruction connectivity matrix C using true dependency edges of the dependency graph. For each pair of vregs (v1,v2) we verify that all the instructions containing v1 and/or v2 are connected using the instruction connectivity list. The process is described by the following pseudo code:

```
C: Instruction connectivity matrix.
I: vreg -> instruction map. Maps vregs to instructions using them.
RES: Compatibility matrix.

for v1 in VREGS
   for v2 in VREGS
       compatible = True
       for i in I[v1]
           for j in I[v2]
                compatible &= C[i,j] || C[j,i]
        RES[v1,v2] = RES[v2,v1] = compatible
```

If at the end of the procedure two vregs are deemed compatible they may share a physical register. Any additional dependency will always be created between instructions who's relative order is already set by existing data dependencies, therefore no additional constraints on instruction ordering is created but register space is still saved. A simple example of the procedure is shown in Figure 5.1.

Our method is adequate even for larger kernels especially as we do not need to concern ourselves with spills outside of the targeted basic blocks. As a fallback, registers are assigned on a LRU basis to maximize the freedom of instruction movement in scenarios where register space is fully depleted. However, up till now we have never been constrained on the number of available registers for the kernel sizes we can realistically explore exhaustively.

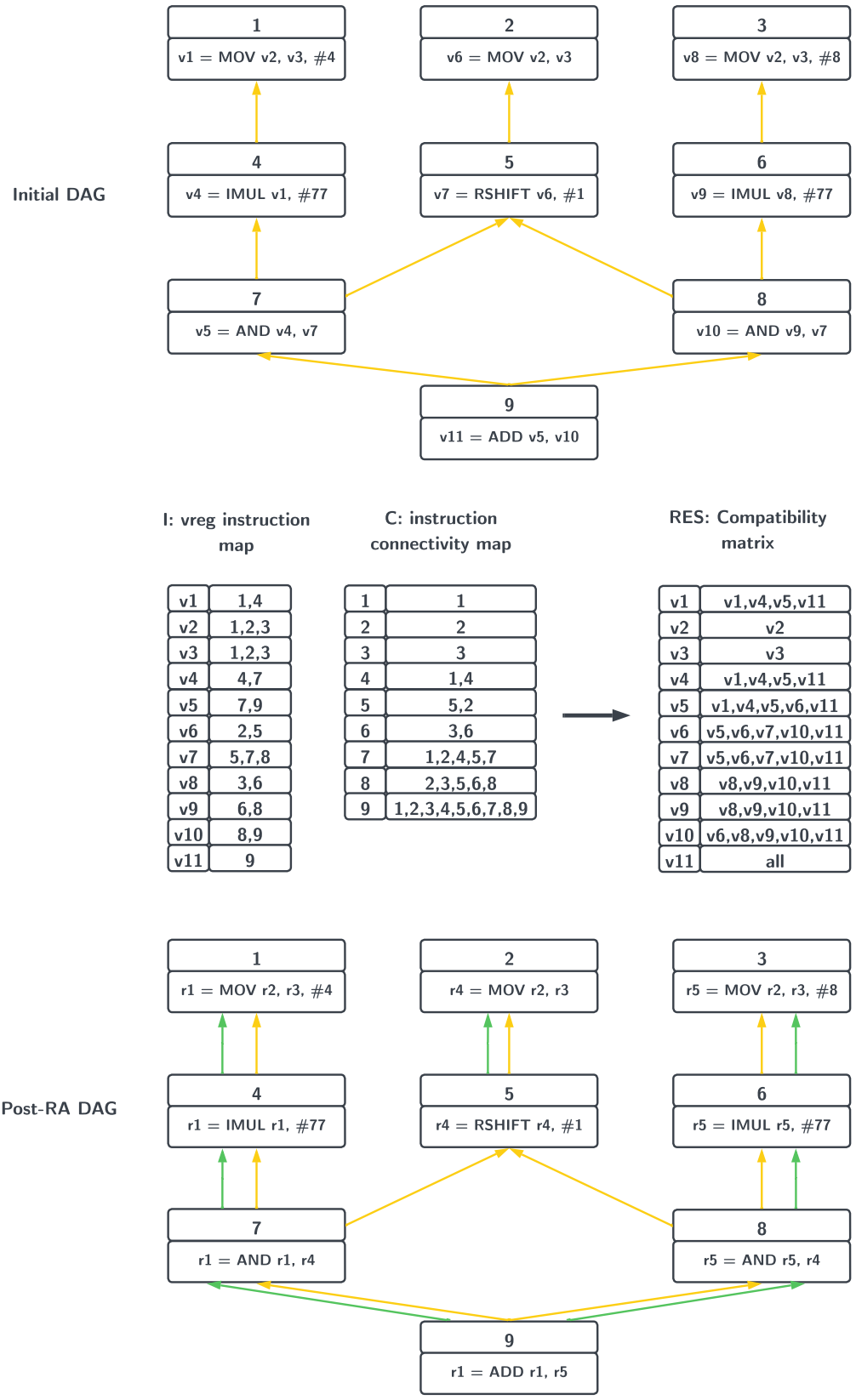

Figure 5.1: Example of Register allocation method. No additional constraints are created. Orange and green arrows depict flow- and output-dependencies respectively.

5.4 Sanity Check Pass

The last pass before code emission is required to ensure the correctness of our results. This pass ensures that schedule permutations all contain the same instructions. Furthermore this pass creates a SHA-256 hash for each permutation and checks if this hash has been seen before by looking it up in a standard hash set. Users are notified if duplicate permutations are generated or the instructions counts do not match. This pass was used to detect the optimization pass interference described in Section 5.1.

5.5 Performance Measuring and Noise Reduction

In order to collect accurate schedule runtime measurements we need a high resolution timer. The timers provided by PAPI have variable precision among platforms. To obtain accurate measurements for schedule performance we disable all dynamic frequency features on the test platforms described. This includes both frequency boost and all downclock behavior. This way the CPU operates on a single static frequency. Clock cycle counters can then be use for accurate time measurement, as validated in Section 6.2.1.

All forms of hardware multi-threading are disabled on our test platform. Other threads may interfere with our measurements with such features enabled. To strengthen this effect, the data collection loop is pinned on a core isolated by the *isolcpus* Linux kernel flag. No other process will be scheduled on the same core meaning instruction and data cache state is undisturbed and the loop is not interrupted, except for occasional hardware interrupts. All experiments are run on the same CPU core.

A specified number of measurements are taken for each generated schedule. The measurements are taken over a user specified number of repeated invocations of the schedule. Before each of the measurements the kernel is executed 100 times to prepare the caches and hardware branch predictors.

5.6 Output & Analyses

Compilation and usage of our tool is described in Appendix A. The tool will output two data files. These data files can be inspected using the parquet-tools cli utility or any other program with support for parquet files. The first data file is the *permutations* file. This file contains the result of all measurements. Each row represents a single measurement. A row contains the name of the scheduler that generated the schedule, a permutation number, a hash of the schedule, the kernel output followed by all the specified performance counters we command the tool to measure. The second data file called *tree* contains the scheduling tree. Each row in this data file represents a single node. A row contains the NodeID the ID of its parent, the instruction text and the list of aggregated performance counters of all schedules that result from this node.

The data files can be analyzed by various scripts created in this thesis. The *analyse.py* script can generate scatter plots of some measured property like runtime for all schedule permutations, an example is shown in Figure 6.4. A second performance feature parameter can be provided to for example scatter schedule runtime against cache misses generated by the schedules. This script can also generate the schedule tree between the provided start and end depth with the color gradient of each node signifying some normalized measured property like runtime, an example is shown in Figure 5.2. Usage of this script is described in Appendix A.3.

Other scripts to generate the various figures and outcomes in this thesis are provided. These may not yet be parameterized but can be easily altered and may serve as a guideline on how to parse the generated data. Scripts for feature exploration, simple ANOVA analysis and correlation are provided as well.

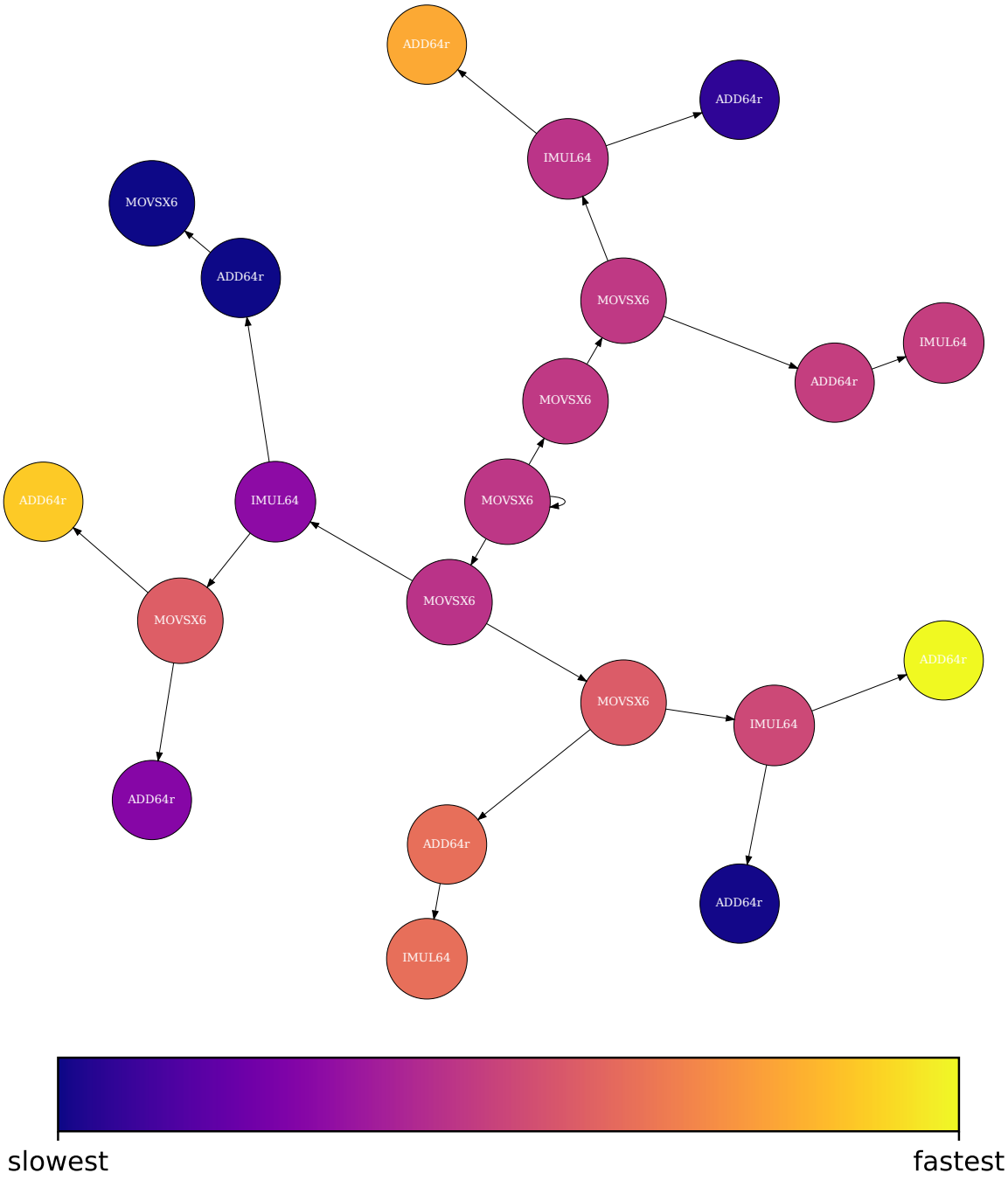

Figure 5.2: Example scheduling decision tree. The color gradient of each node signifies min-max normalized runtime.

Chapter 6

Results

This Chapter describes the results collected in this thesis. In this thesis, we wish to answer the following questions:

- How relevant is instruction scheduling to performance on modern OOO-Processors?
- How do contemporary instruction schedulers fare in the scope of the complete solution space?
- What explains the difference in performance of instruction schedules on OOO-Processors?

Using the tool described in the preceding chapters we have conducted a number of experiments to answer these research question.

This Chapter is organized as follows. Section 6.1 describes the test systems on which experiments were conducted. Section 6.2 describes the validation of our methods and how the collection of the generated preliminary data steered our research. Section 6.3 answers the first two of our research questions. We show significant performance differences among schedules and the relative performance of the schedules generated by LLVM. We expand upon this in Section 6.4 by investigating the relationship between scheduling and stability of performance. Section 6.5 investigates the difference in response to our schedule exploration between micro architectures. Section 6.6 attempts to answer our last research question by finding possible explanations for the observed performance variance among schedules of the same compute kernels.

6.1 Experimental Setup

We have conducted experiments on two systems. These systems implement the same ISA, however the underlying microarchitecture is significantly different. This will allow us to see if any of our findings are present on both architectures. Relevant specifications are shown in Table 6.1.

A block diagram detailing microarchitecture details of the Intel Ivy Bridge architecture is shown in Figure 2.4. A similar diagram is shown for the AMD Zen 2 Architecture in Figure 6.1. Note the

differences in the backend of both processors. While the Intel architecture has a single scheduler with a unified reservation stations for both integer and floating operations across all its execution ports, the AMD architecture splits floating point and integer scheduling in two separate sections. Furthermore, the Zen 2 architecture has separate scheduling and reservation stations for each execution port. The exact implementation and behavioral details of these schedulers are unknown.

LLVM 14 compiles with a post register allocation scheduling pass for the Zen 2 architecture by default. This pass is not enabled for the Ivy Bridge architecture.

As described in Section 5.5 all dynamic frequency behavior and multi-threading features are disabled. All experiments run on the same isolated core. To measure the performance of a schedule a specified number of measurements are taken. These measurements contain a user specified number of repeated invocations of the schedule. Before each of the measurements the kernel is executed 100 times to prepare the caches and hardware branch predictors.

Table 6.1: Description of test systems.

CPU	Memory	Architecture	L1d	L1i	L2	L3
Intel i7-3770 @ 3.40GHz	16 GB DDR3	Ivy Bridge	32K	32K	256K	8192K
AMD R5 3600 @ 3.50GHz	16 GB DDR4	Zen 2	192K	192K	3M	32M

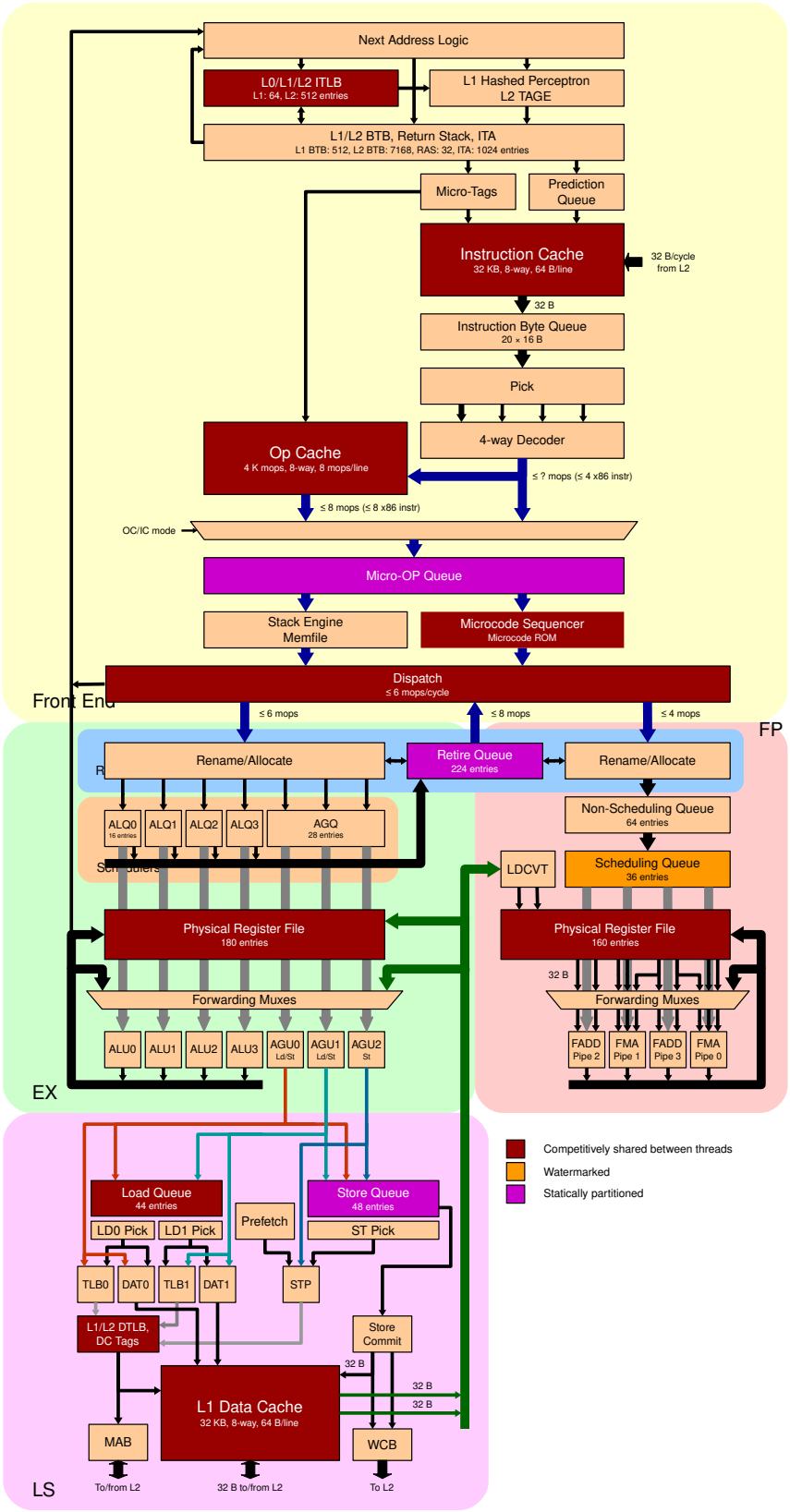

Figure 6.1: Overview of the AMD Zen 2 Architecture [2]

6.1.1 Kernels

All the compute kernels used in the experiments are shown in Appendix B. To further improve the accuracy of our measurements kernels will usually use data structures that fit within the L1 or L2 data cache as not to be disturbed by the dynamic nature of the rest of the memory subsystem that is influenced by other processes.

The set of kernels contains kernels with varying levels of manual unrolling. This unrolling yields more instructions that may be executed concurrently. This gives schedulers more freedom and therefore a larger quantity valid of instruction schedule permutations. Inter dependencies among instructions can additionally be decreased or increased by adding partial sums only after the loop body. This way some inter-loop dependencies among iterations may also be broken. To study the effects of instruction type and latency on schedule performance variance, the kernels contain different operations of varying latency such as addition, multiplication and division. The 3dot and error kernels were chosen because they represent real world calculations of the mean squared error and vector dot product respectively.

6.2 Validation & Preliminary Data Collection

In this section we validate our tool and test systems. We additionally show some of our preliminary data collection and analysis.

6.2.1 Runtime Measurement Validation

In order to collect performance data on the generated schedules we must be able to measure the runtime of the schedules accurately. PAPI provides platform independent timers. The resolution of these timers varies between platforms. Some processors provide reference clocks that have a static frequency from which such timers may be derived. These features may be lacking on other processors or the resolution may vary. Besides a high resolution timer, we need repeatable results. By default, modern processors employ features such as dynamic clock frequency and hardware multithreading. These features may disturb measurements and insert too much variance to discern small performance difference. Therefore, to obtain accurate measurements for schedule performance, we disable all dynamic frequency and multithreading features on the test platforms described in Section 6.1. This includes both frequency boost and all downclock behavior. This way the CPU operates on a single static frequency. Together with a static frequency common clock cycle counters present on the processor form a very accurate way to measure runtime. We verified this on an Intel platform that provides a reference clock. Figure 6.2 shows average measured runtime among 1024 measurements of 1000 iterations of the error-u2-2c kernel. There is a strong linear correlation between the reference clock and the normal clock cycle counter.

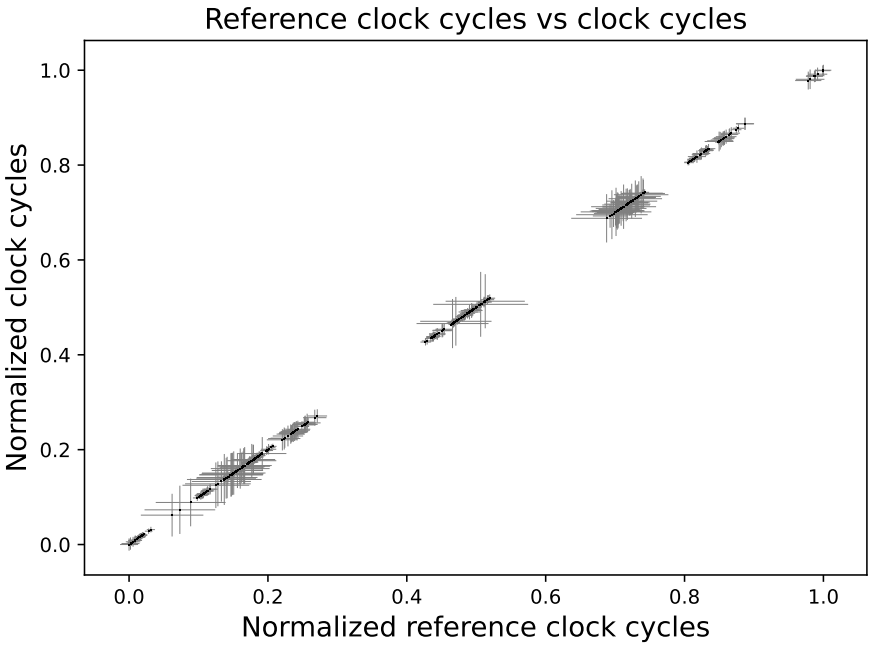

Figure 6.2: Linear relationship between reference clock cycles and normal cycle counter. 1024 measurements of 1000 iterations on the Intel test platform. Data is min-max normalized. Error bars depict standard deviation.

6.2.2 Data Collection Examples

In order to validate that our tool and test platforms work correctly and resultant data has sufficient resolution for meaningful analysis, we collected preliminary data on a few different kernels. We present these here to illustrate the type of data we collect, the accuracy of that data, some of the analysis that may be done upon it and how these results steered our research.

We collect the runtime of all valid schedules of a kernel along with multiple performance metrics such as cache misses, backend stalls, etc. These features may point to the reason behind any possible runtime differences among schedules of the same kernel. We can normalize the schedule runtimes to the runtime of the default schedule created by LLVM and create scatter plots to see if performance differences exist.

The performance data is also accumulated along the edges of the scheduling decision tree. We can plot the resulting decision tree and color the nodes to show the average performance of all schedules that take this route along the decision tree. This should allow us to see which decisions effect performance.

We illustrate how this data is represented by example of the addmul-u2-2c kernel shown in Appendix B. The kernel multiplies each element of an 4096 integer array by a constant and sums the result in two dependency chains. The instruction DAG of the kernel is shown in Figure 6.3. There are 90 valid schedules that can be derived from the DAG. Figure 6.4 shows a scatter of these 90 unique schedules sorted by runtime the normalized LLVM Default scheduler. Data was collected on the

AMD Zen 2 system by taking 4096 measurements of 2000 kernel iterations.

From the figures follows that even for such a small and simple kernel statistically significant differences between the various schedules exist. Compared to the default output, schedules with a median runtime 1.9% faster exist. There are also schedules that are 6.6% slower. Turning off the Post register allocation scheduling severely degrades the performance of the schedule generated by LLVM.

Note that there are always two points with equivalent performance. The reason why becomes obvious when looking at the center the scheduling decision tree, shown in Figure 6.5. The associated DAG is shown in Figure 6.3. Flow-, anti- and output- dependences are marked by orange, red, green respectively. The DAG depicts that we may start our schedule with either of the move instructions that function as data loads. These are depicted as instructions 1 and 7 in the scheduling tree. The decision of which load comes first does not matter for performance, therefore we see two equivalent branches emerging from the root node.

We can use the tree to determine what the commonalities are between fast schedules and deduce which decisions are important. We see that fast permutations schedule the move instructions first in either order. Additionally the loop iterator increment add-instruction instruction appears early in the schedule. Slow permutations tend to schedule one dependency chain fully before the scheduling instructions of the other chain. These observations would make sense in a statically scheduled parallel pipelined machine were such instruction interleaving and filling otherwise empty slots post data load would increase throughput. Not all differences are intuitive. Comparing the default LLVM schedule to the best possible schedule we notice that they are identical except that the position of the loop iteration addition and the first multiplication are reversed.

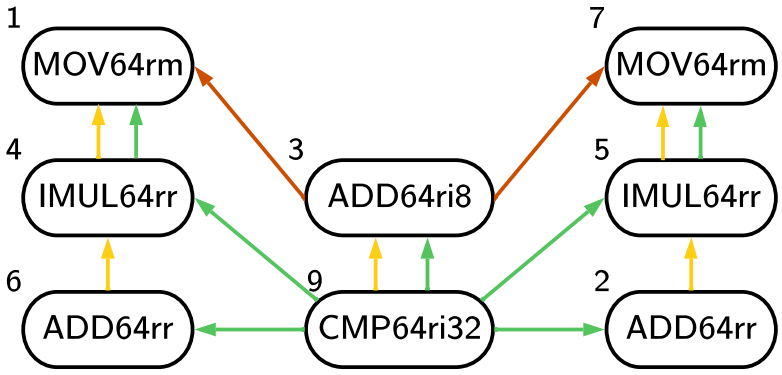

Figure 6.3: Instruction DAG of the addmul-u2-2c kernel. Flow-, anti- and output- dependences are marked by orange, red, green respectively.

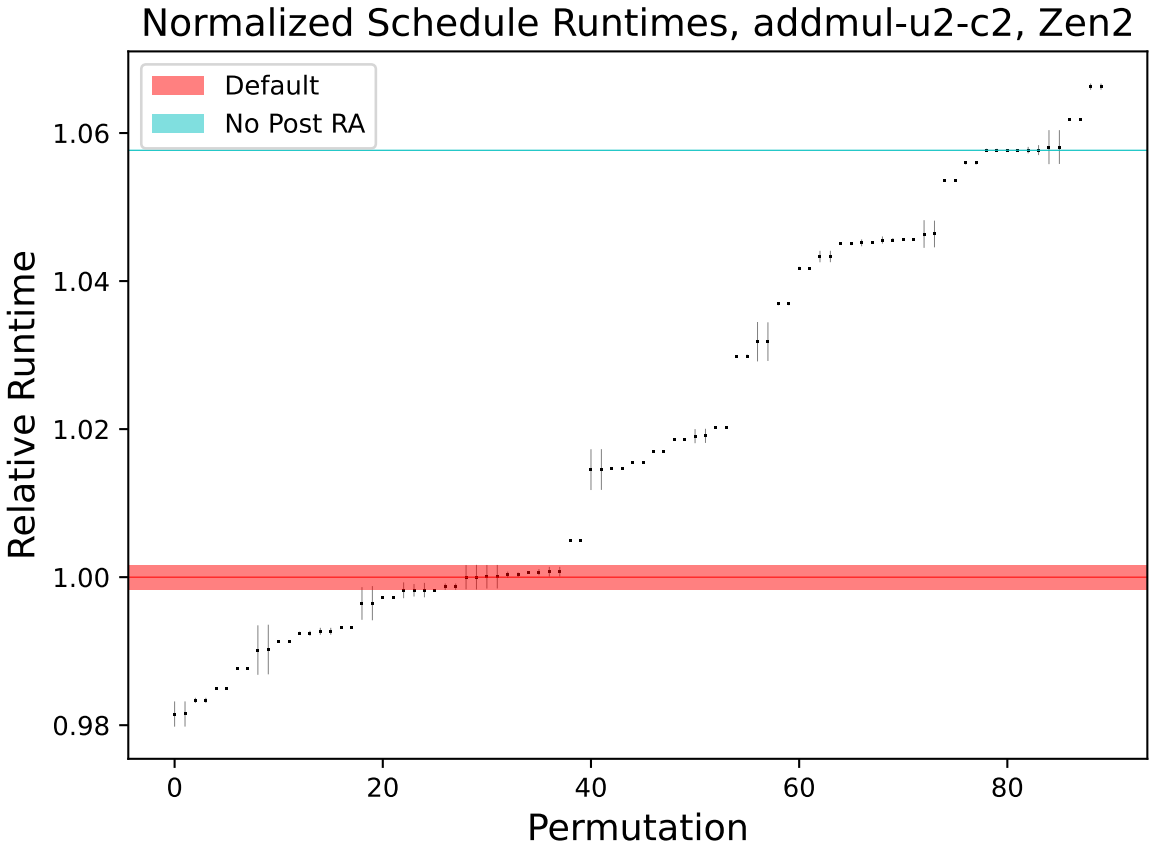

Figure 6.4: Scatter plot of median normalized runtime of schedules of the addmul-u2-2c kernel collected over 4096 measurements of 2000 kernel iterations on the Zen 2 microarchitecture. Error bar depicts standard deviation.

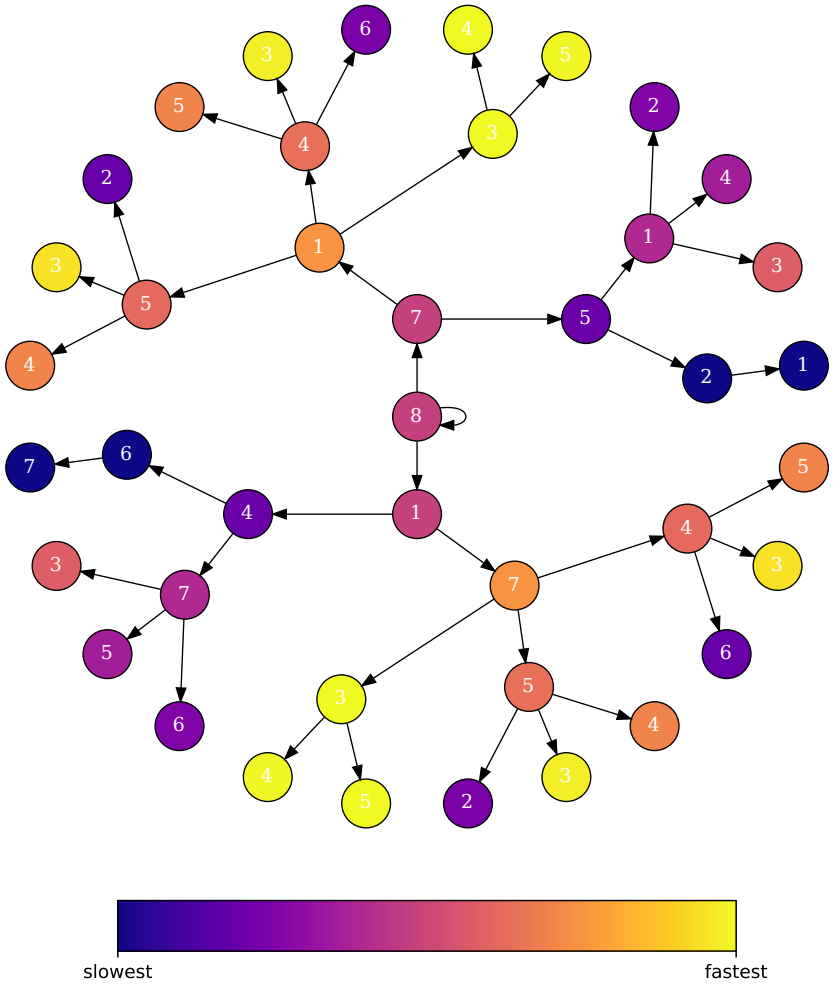

Figure 6.5: Scheduling decision tree for the addmul-u2-2c kernel. The color gradient of each node signifies min-max normalized runtime.

Larger performance differences are seen among schedules of the error-u2-2c kernel shown in Appendix B. This kernel calculates the summation part of a mean squared error calculation:

$$MSE = \frac{1}{n}\sum_{i=1}^{N}(x_i - \hat{x})^2$$

Figure 6.6 shows runtime spread of $(-12.7\%, 15.6\%)$ compared to the LLVM default among the 448 valid schedules of the inner loop. The schedule created when post register allocation is turned off is significantly better, showing that additional optimization passes can be detrimental in certain cases.

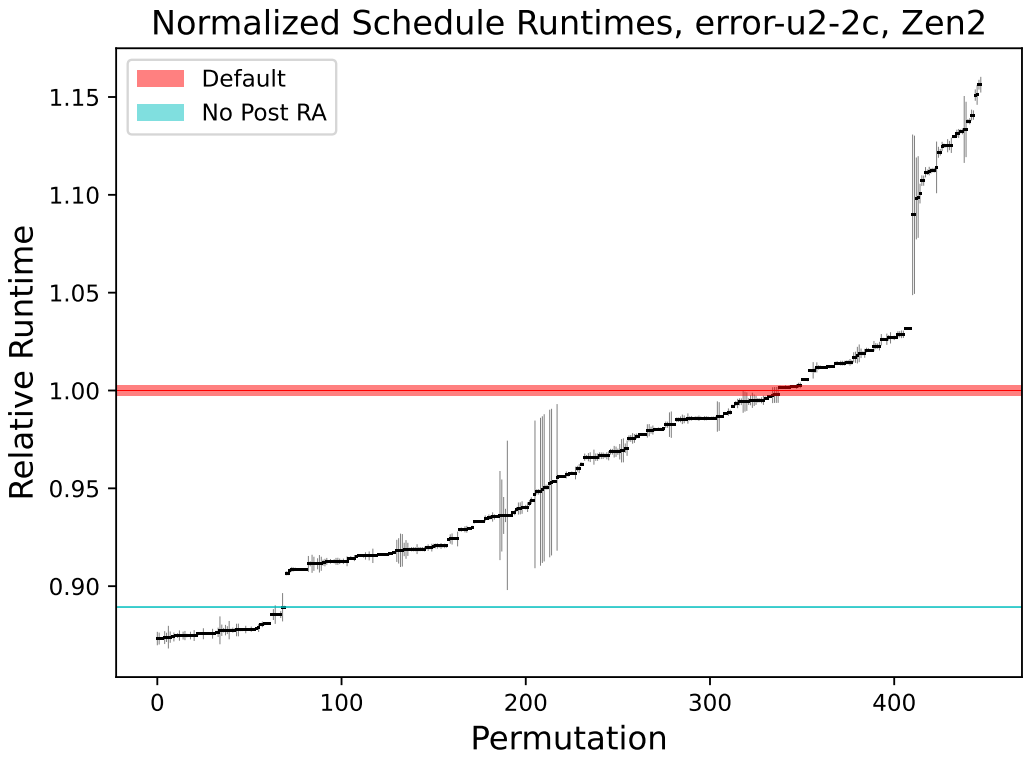

Figure 6.6: Scatter plot of median normalized runtime of schedules of the error-u2-2c kernel collected over 4096 measurements of 2000 kernel iterations on the Zen 2 microarchitecture. Error bar depicts standard deviation.

Note that the magnitude of the runtime standard deviation varies multiple orders of magnitude among schedules. These effects are persistent between repeated experiments. Some schedules seem to have inherently increased variability in their performance. This initial finding led to the investigation presented in Section 6.4.

Another interesting note is the empty regions in Figure 6.4, 6.6. Schedules are not uniformly spread. This effect is even more obvious for the schedules of the 3dot kernel shown in Figure 6.7. The majority of schedules have similar performance followed by a group of schedules that are 49% slower than the default output. As seen in Figure 6.8 similar behavior is not shown on the Intel Ivy Bridge architecture. This difference in response among architectures led us to investigate cross architecture optimization in Section 6.5.

For the presented kernels the variance in schedule runtime measurements is small compared to the performance differences among the schedules meaning that useful analysis can be done on the data collected by using our tool and setup.

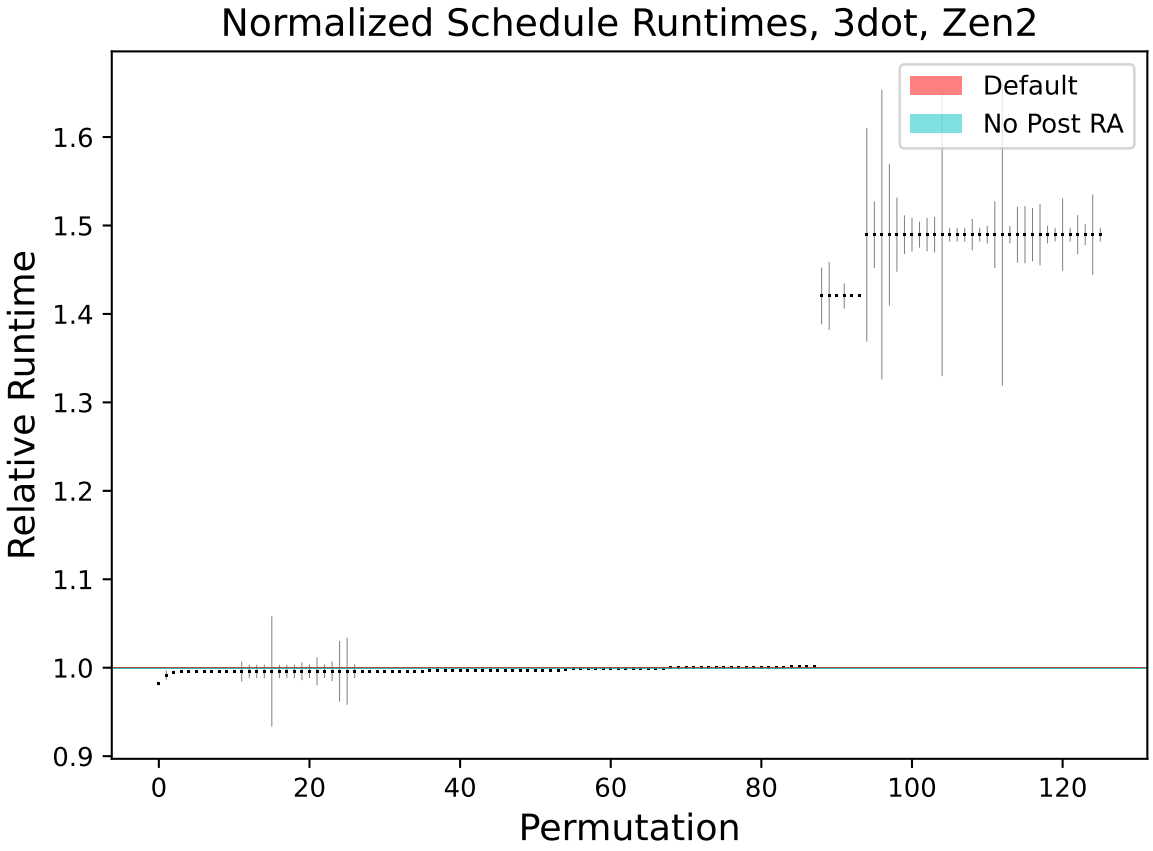

Figure 6.7: Scatter plot of median normalized runtime of schedules of the 3dot kernel collected over 4096 measurements of 2000 kernel iterations on the Zen 2 microarchitecture. Error bar depicts standard deviation.

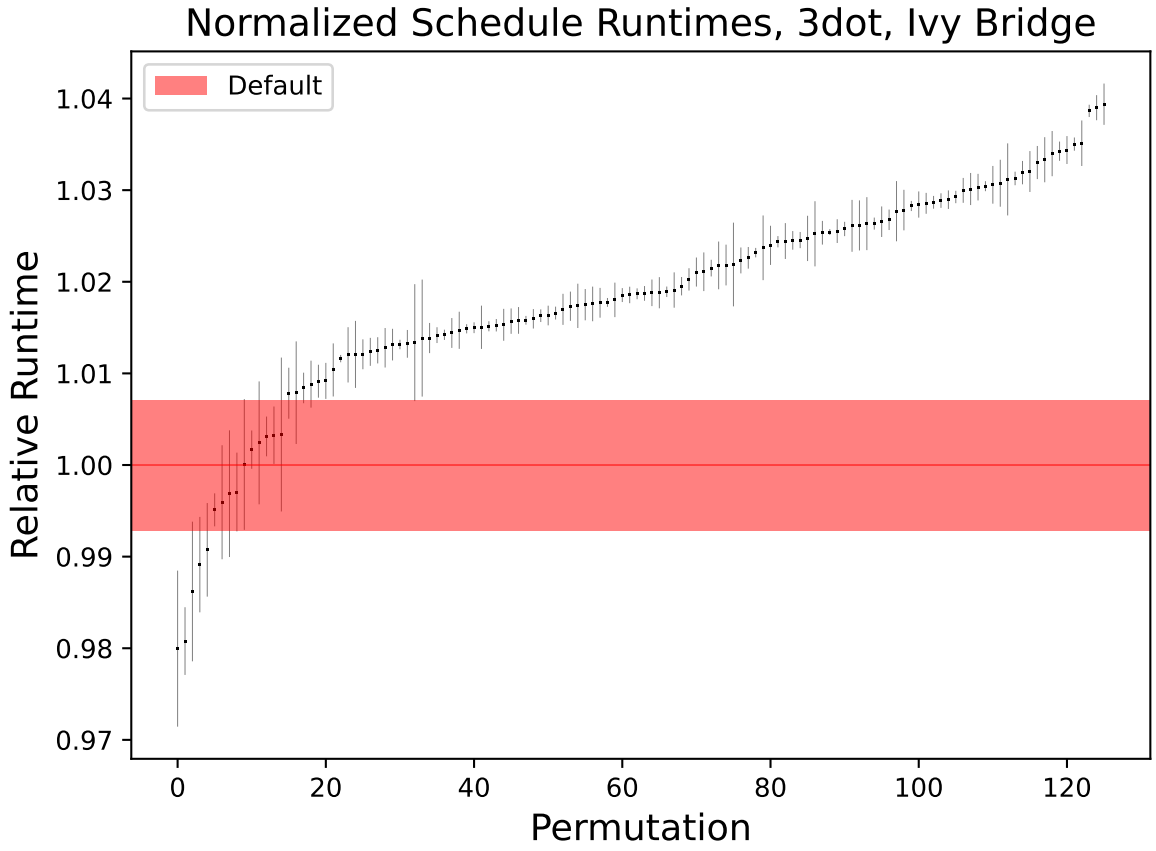

Figure 6.8: Scatter plot of median normalized runtime of schedules of the 3dot kernel collected over 4096 measurements of 2000 kernel iterations on the Ivy Bridge microarchitecture. Error bar depicts standard deviation.

6.2.3 Exclusion of Order Influences

When running experiments on modern hardware it is important to take note of system caches at various levels of the system. Modern processors have large caches and various predictors that influence runtime performance. Subsequent experiments may influence each other through these resources. We run a specified number of iterations before each measurement to make sure that all such state is properly flushed. Additionally the number of iterations and sample size for each schedule is large enough meaning that the order in which schedules are explored should have no influence on our results. To validate this empirically we run 4 experiments each with 4096 measurements of 1000 iterations of the error-u2-2c kernel on the AMD Zen 2 test system. The order in which the schedules are generated is random instead of the normal depth first exploration. We scatter these runs in red along with the normal results in black. This plot is shown in Figure 6.9. This plot makes it visually obvious that there is almost no variance since all but two points are directly covered by there black counterparts.

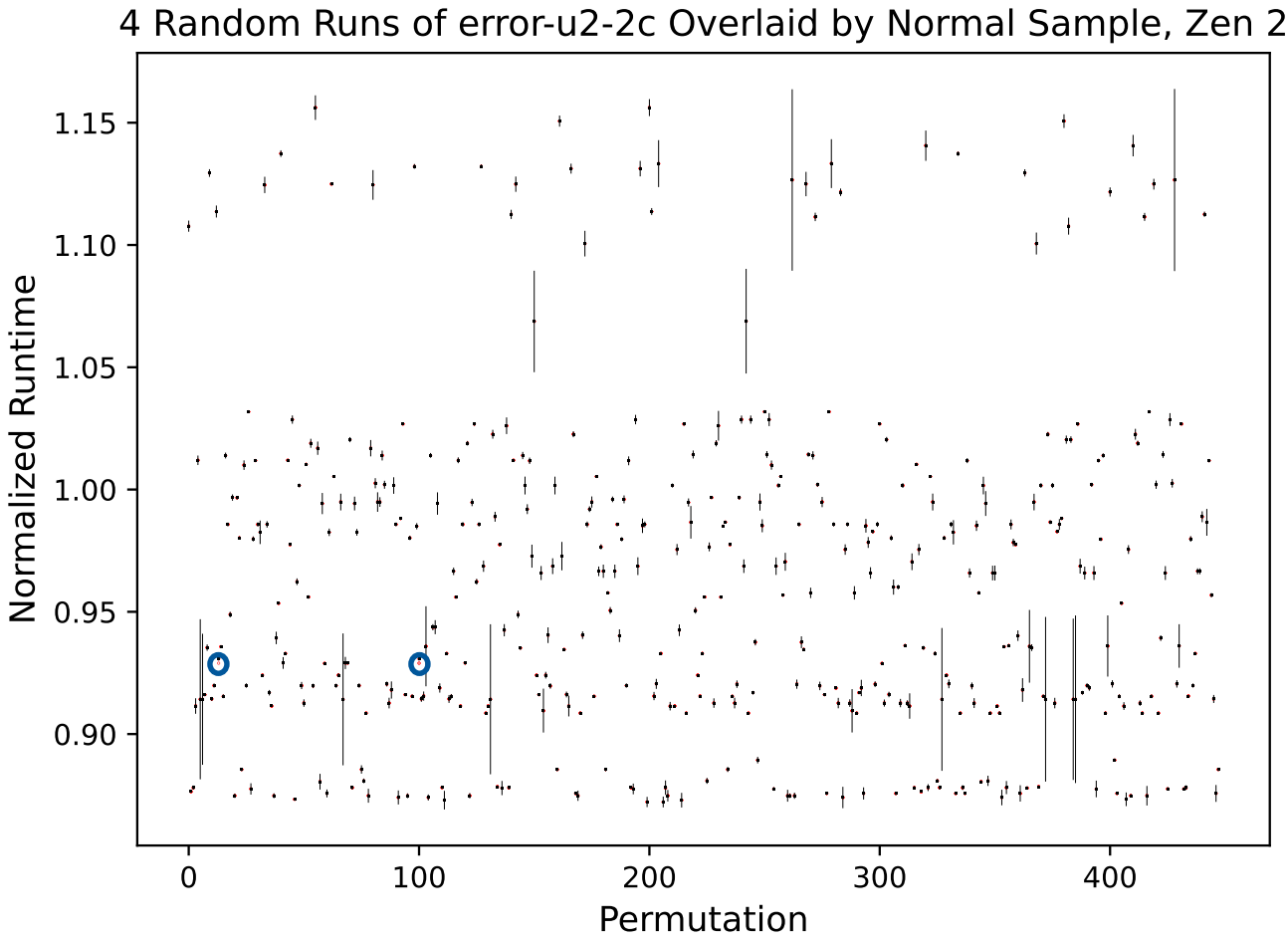

Figure 6.9: Scatter plot of median normalized runtime of schedules generated in 4 random runs of the error-u2-2c kernel collected over 4096 measurements of 1000 kernel iterations on the Zen 2 microarchitecture shown in red. Overlaid by the results of a normal run in black. Error bar depicts standard deviation. Two anomalies are encircled in blue.

6.3 Schedule Performance Variation

In this thesis, we wish to investigate if different instruction schedules show meaningful different performance on modern OOO-processors. To answer this question, we generate and benchmark all valid schedules of a large variety of different kernels. The kernels are shown in Appendix B. Figures 6.10 and 6.11 show an overview of the runtime spread of all the tested kernels for the AMD Zen 2 and Intel Ivy Bridge architecture, respectively. Large performance differences can be seen among schedules of the same kernel. Moreover, the default LLVM scheduler does not always produce optimal or even good schedules. Large runtime differences are not observed for all kernels. Kernels containing comparatively more high latency instructions, such as the floating point kernels, show less variance.

Yet even in these cases small differences do still exist. Table 6.2 shows the spread among the medians of schedule runtimes. To test the significance of these differences we apply Welch ANOVA. Classic ANOVA is not applicable since variance among schedules is not equal, an interesting observation discussed in Section 6.4. Welch ANOVA allows us to test a null hypothesis which states that the population mean of multiple groups, in this case schedules, are equal. Using this method we have a statistical measure by which we may state if scheduling has effect for specific kernels. Results of this test for each kernel is shown in Table 6.3. For every tested kernel it is so unlikely that our data could result from equal runtime means that the p-values fall within the limits of the floating point units. We can therefore state with a very high degree of confidence that scheduling affects runtime performance in all tested cases, as the mean runtimes show statistically significant differences. This was visually obvious for most kernels. The amount of variance explained by grouping by schedule is represented by the η_p^2 values. For kernels less influenced by scheduling variance within a group, measurement error becomes the major component of variance. This is especially noticeable on the AMD platform where variance within schedule groups tends to be higher.

Figure 6.10 and 6.11 depict an overview of the runtime spread of all the tested kernels for the AMD Zen 2 and Intel Ivy Bridge architecture respectively. As can be seen these figures are visually different, meaning the architectures have different responses to scheduling. The results in Table 6.2 confirm this. We observe that the spread of measured differences among schedules are different between the architectures. In Section 6.5 we take a closer look at these differences in the hopes of identifying scheduling that delivers good result for all large window dynamically scheduled processors.

An interesting cross architecture difference is observed for the adddiv-u2-2c kernel. There is almost no difference between schedules on the AMD platform but a sizeable one on the Intel platform. This is likely due to the way IDIV instructions are handled by these respective architectures. This instruction translates to 2 μops with an issue latency of 13 on the Zen 2 architecture, compared to 9 μops with an issue latency of 8 on the Intel Ivy Bridge architecture [11]. The higher amount of μops puts pressure on reservation station resources and increases the importance of compiler scheduling.

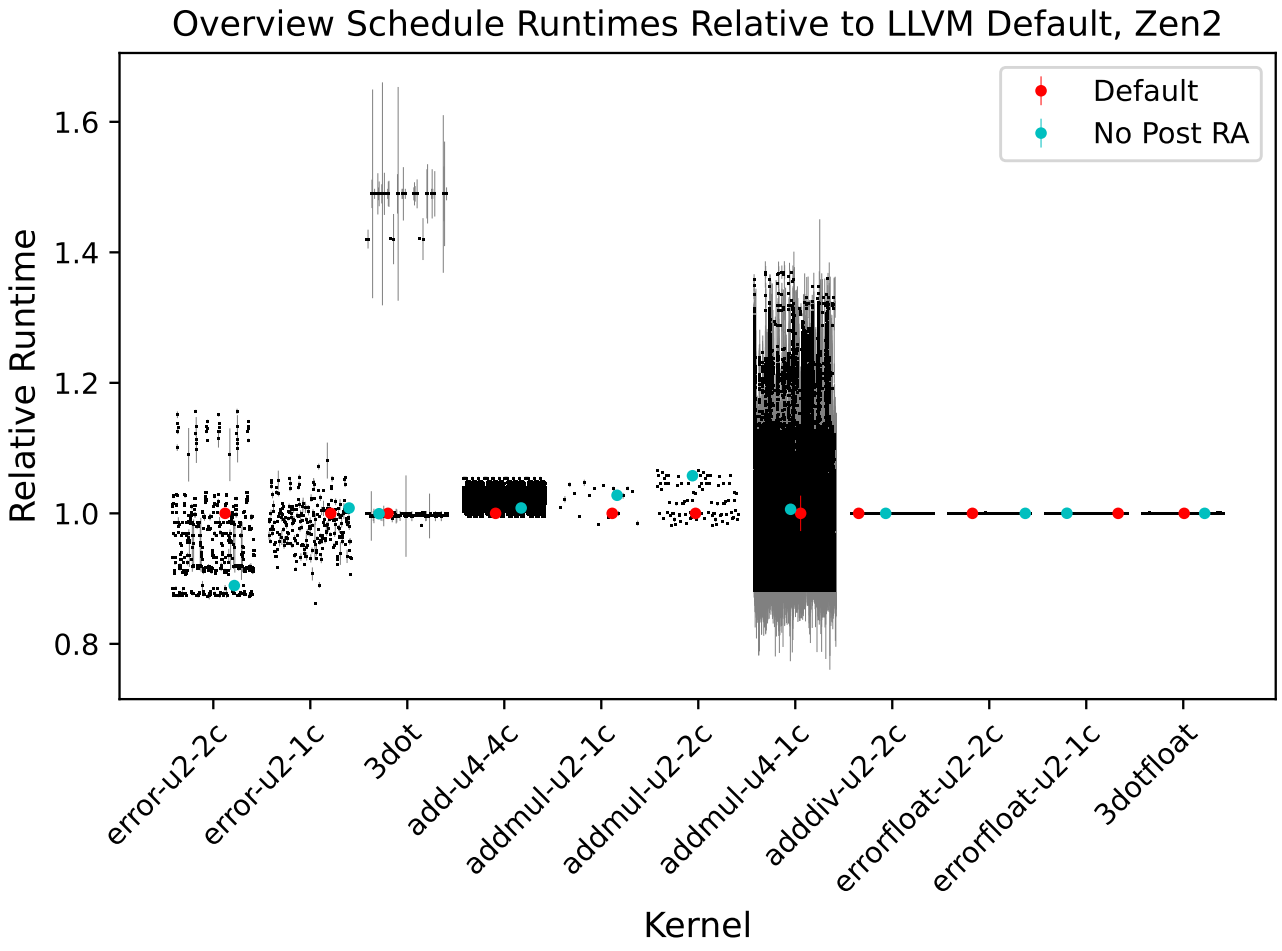

Figure 6.10: Overview of performance differences among schedules for all tested kernels on the AMD test system. Error bar depicts standard deviation.

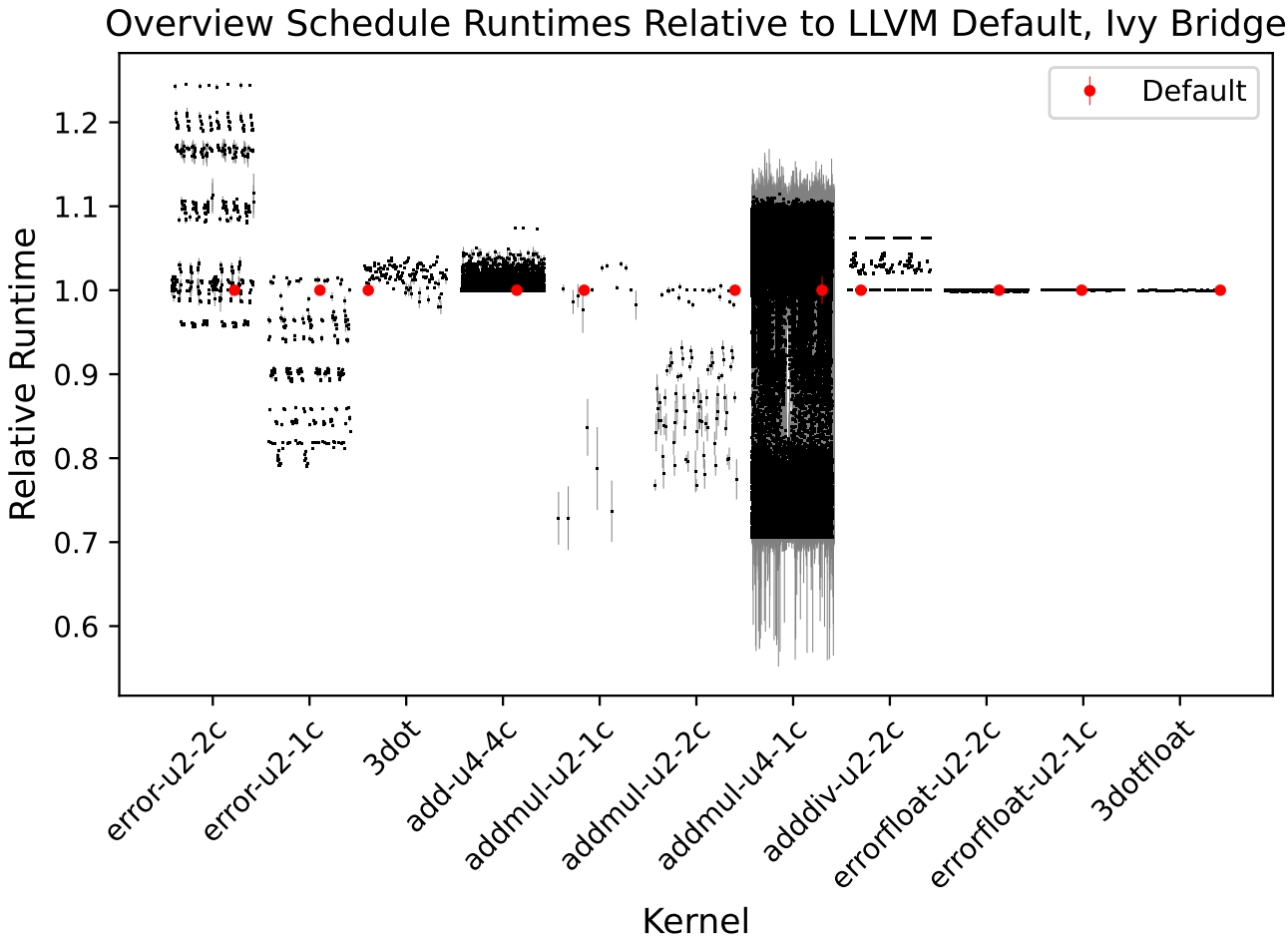

Figure 6.11: Overview of performance differences among schedules for all tested kernels on the Intel test platform. Error bar depicts standard deviation.

Table 6.2: Runtime spread of the tested kernels on both platforms normalized by the runtime of the LLVM default. 4096 measurements of 2000 iterations.

Kernel	#schedules	AMD slowest	AMD fastest	Intel slowest	Intel fastest
error-u2-2c	448	+15.6%	-12.7%	+24.5%	-4.37%
error-u2-1c	259	+8.10%	-13.8%	+1.58%	-21.0%
3dot	126	+49.0%	-1.79%	+3.94%	-2.00%
add-u4-c4	9216	+5.36%	-0.455%	+7.41%	-0.0829%
addmul-u2-1c	17	+4.67%	-1.70%	+3.12%	-27.2%
addmul-u2-2c	90	+6.63%	-1.85%	+0.486%	-23.2%
addmul-u4-1c	65178	+37.0%	-11.8%	+11.4%	-29.5%
adddiv-u2-2c	245	$-4.79e^{-5}$%	$-2.40e^{-5}$%	+6.24%	-0.00204%
errorfloat-u2-2c	1680	+0.0651%	-0.0162%	+0.0927%	-0.200%
errorfloat-u2-1c	840	+0.0410%	-0.00607%	+0.0355%	-0.0201%
3dot-float	63	+0.000198%	+0.0569%	-0.000580%	-0.0504%

Table 6.3: Results of Welch's ANOVA on collected data. p-values describe chance the means of all schedules are equal. Partial eta-squared values describes the amount of variance explained by grouping by schedule.

Kernel	AMD p	AMD η_p^2	Intel p	Intel η_p^2
error-u2-2c	0	0.990481	0	0.996519
error-u2-1c	0	0.978072	0	0.997858
3dot	0	0.978665	0	0.943515
add-u4-c4	0	0.990957	0	0.977883
addmul-u2-1c	0	0.982329	0	0.957579
addmul-u2-2c	0	0.998269	0	0.970181
addmul-u4-1c	0	0.968029	0	0.961266
adddiv-u2-2c	0	0.000243	0	0.999979
errorfloat-u2-2c	0	0.314625	0	0.99296
errorfloat-u2-1c	0	0.882331	0	0.914607
3dot-float	0	0.359115	0	0.921564

6.4 Performance Stability

While collecting data, we observed that the measured standard deviation for certain schedules was significantly larger than the average. This effect was present no matter the sample size and consistent among multiple samples. Given a sufficiently large and equal sample size n the standard deviation on runtime of different schedules should be more or less uniform, since magnitude of the standard deviation should be proportional to the time measurement error. This error is the

same for all schedules. That is unless scheduling does not only influence runtime but also runtime stability, thus influencing the sample variance.

Given constant measurement error on runtime, each measurement in a sample will be drawn from an unknown distribution over a domain proportional to the magnitude of the error. Let our null hypothesis H_0 state that scheduling does not influence performance stability. The population of runtime of each schedule should then have the same variance:

$$H_0 : \sigma_0^2 = \sigma_1^2 ... = \sigma_n^2$$

with n the number of schedules.

In the alternative hypothesis H_a scheduling does influence stability and as a result variance may differ among schedules.

$$H_a : \sigma_i^2 \neq \sigma_j^2,\ i, j \in [0, n]$$

We can use the Brown-Forsythe test for equality of variances to test these hypotheses. This test is robust against non-normal data and skewed distributions. Together with our large sample size of 4096, this should yield accurate results. Table 6.4 shows the resulting p-values for the different kernels. Due to the large sample sizes and the number of valid schedules possible for the kernels the p-values fall within the rounding error of the floating point unit. Taking a cut off value for $p < 0.01$ we can reject the null hypothesis for all but two kernels. The results confirm what was visibly already obvious. The differences in variance among the schedules of most of the kernels cannot be explained by the measurement error alone. Therefore a significant component of the variance is a result of the properties of the schedules themselves.

As a sanity check we plot a scatter of the Intel 3dot-float result for which we strongly **cannot** reject H_0. The scatter plot is shown in Figure 6.12. Indeed the magnitude of the standard deviation is fairly uniform among the schedules and any differences are small.

Table 6.4: p-value for the tested kernels resulting form the Brown-Forsythe test for equality of variances on samples of possible schedules

Kernel	AMD p	Intel p
error-u2-2c	0	0
error-u2-1c	0	0
3dot	0	0
add-u4-c4	0	0
addmul-u2-1c	0	0
addmul-u2-2c	0	0
addmul-u4-1c	0	0
adddiv-u2-2c	0.48796	0
errorfloat-u2-2c	0	0
errorfloat-u2-1c	0	0
3dot-float	0	0.99999

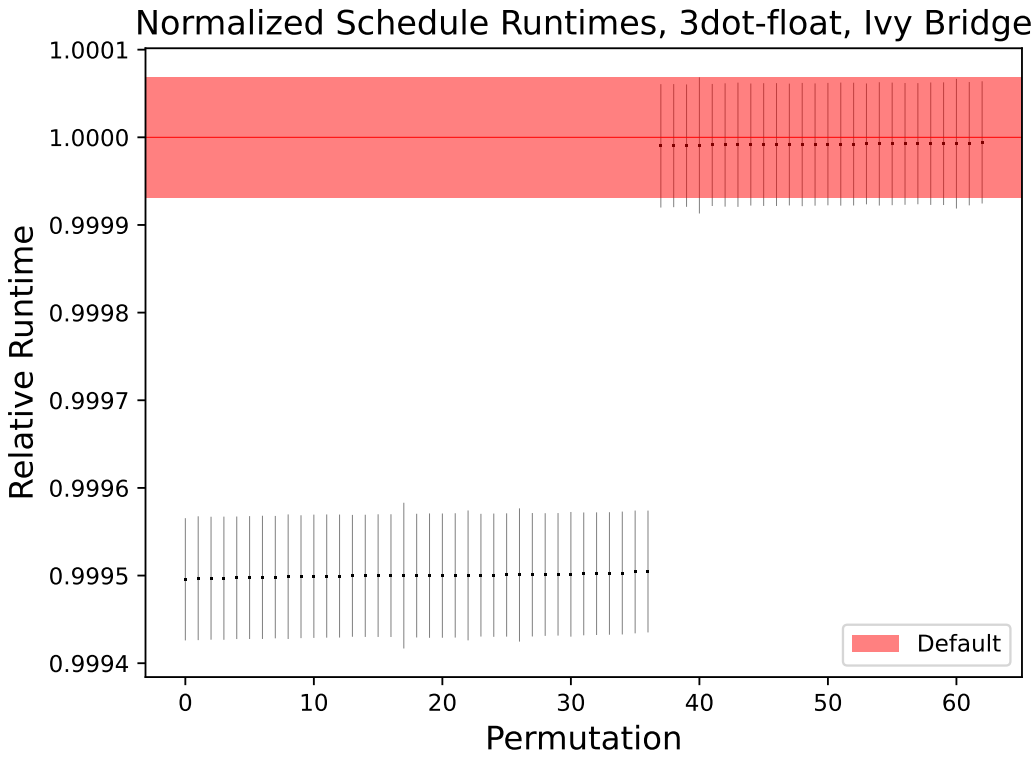

Figure 6.12: Scatter plot of median normalized runtime of schedules of the 3dot kernel collected over 4096 measurements of 2000 kernel iterations on the Ivy Bridge microarchitecture. Error bar depicts standard deviation.

6.5 Cross Architecture Optimization

As shown in Section 6.3, the response of the architectures of our test systems to schedule exploration of the kernels is visibly different. Much work is done to model pipelines and creation of tailored heuristics for each architecture in order to create optimal machine code. Yet, out of order architectures operate similarly from a high-level view. One might expect precise optimization details to have relatively minor impact, due to, for example, the large instruction window size the processors themselves optimize over. The fact that out-of-order architectures operate similarly from a high-level point of view allowed much work to be done to model pipelines in a parametrized fashion and the creation of tailored heuristics for each architecture in order to create optimal machine code. Our results in Section 6.3 show that the response of the architectures of our test systems to the schedule exploration of varying kernels is visibly different. This implies that modeling and tuning work in the compiler for different microarchitectures is indeed necessary. Despite all of this work, in Section 6.3 we also show that schedules exist that (significantly) outperform the LLVM default scheduler. Besides, the modeling and tuning needs to be performed over and over again for new microarchitectures and pipelines that are introduced.

Our results thus further reinforce that extensive tuning for a particular target microarchitecture is

a necessity in order to near optimal performance. However, such architecture tuning is rarely done outside of the domain of High Performance Computing (HPC). Pre-compiled packages that are microarchitecture agnostic are commonplace and this also concerns packages that are performance sensitive such as physics engines for video games, 3D modeling software and simulation suites. This raises the question whether our work can be used to create schedulers that deliver good, but not optimal, performance on large ranges of OOO-microarchitectures. By plotting the relative runtime of schedules on both microarchitectures of our test systems we may discover if a common optimizer could be effective if architecture-specific optimizers are needed.

Figure 6.13 shows a plot of schedule performance for the error-u2-2c kernels on each test system on either axis. It is evident by the cluster on the bottom left that the fastest schedules on one architecture are also fastest on the other. Similar results are found for the 3dot, addmul-u2-1c kernels. This might suggest a common optimizer may be created. However this result is contrasted by the results for the kernels shown in Figure 6.14 and 6.15. In these cases the optimum for either architecture is either mediocre or the worst case for the other architecture. This result shows that a common optimizer can not generate optimal results for both the architectures for all input. Creation of a scheduler that generates acceptable trade offs by finding the Pareto front, the set containing only non-dominated schedules, could be an interesting area of future research for cross architecture benchmarking purposes or the generation of semi-optimal architecture agnostic binaries. In doing so research in the field of multi-objective optimization may be drawn upon. Efficient exploration of the solution space may be achieved by recognizing repetition of results in the decision tree as well as the various other techniques such as Monte Carlo Tree Search.

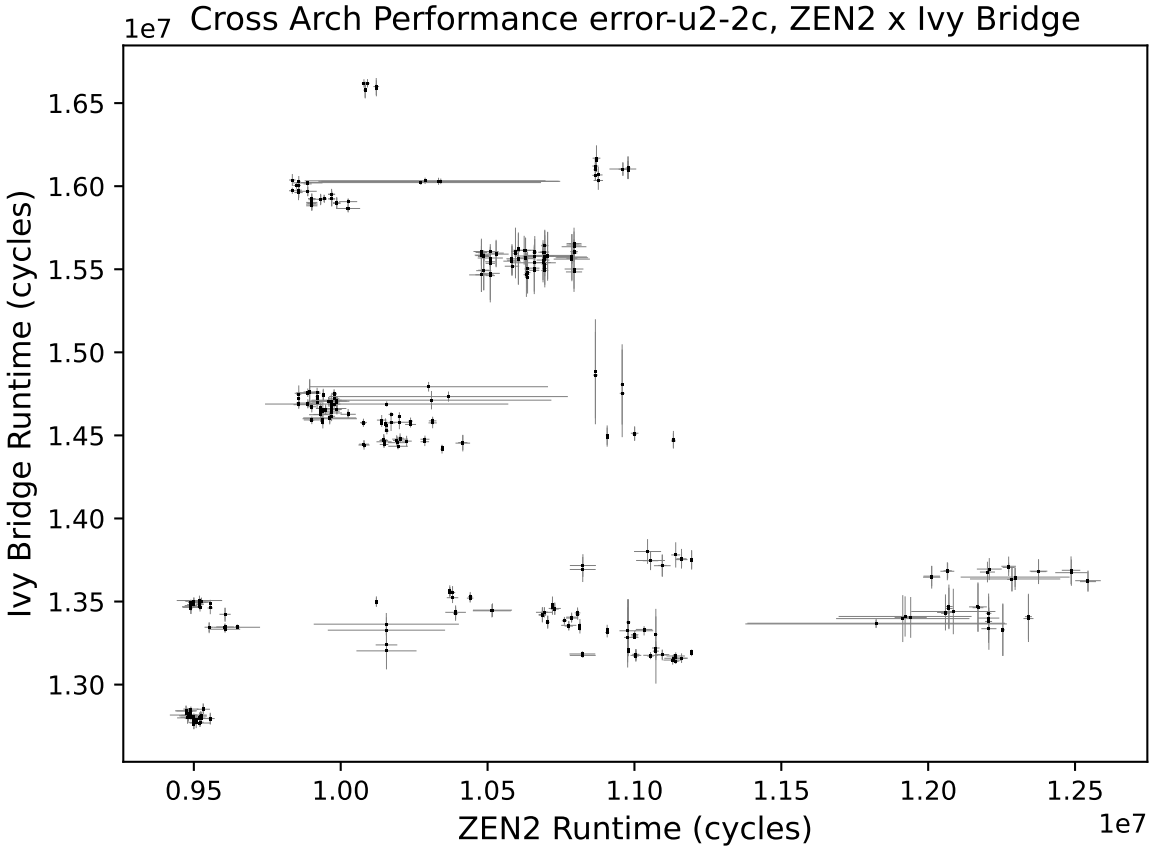

Figure 6.13: Scatter of schedule performance on both test systems for the error-u2-2c kernel collected over 4096 measurements of 2000 kernel iterations. Error bars depicts standard deviation.

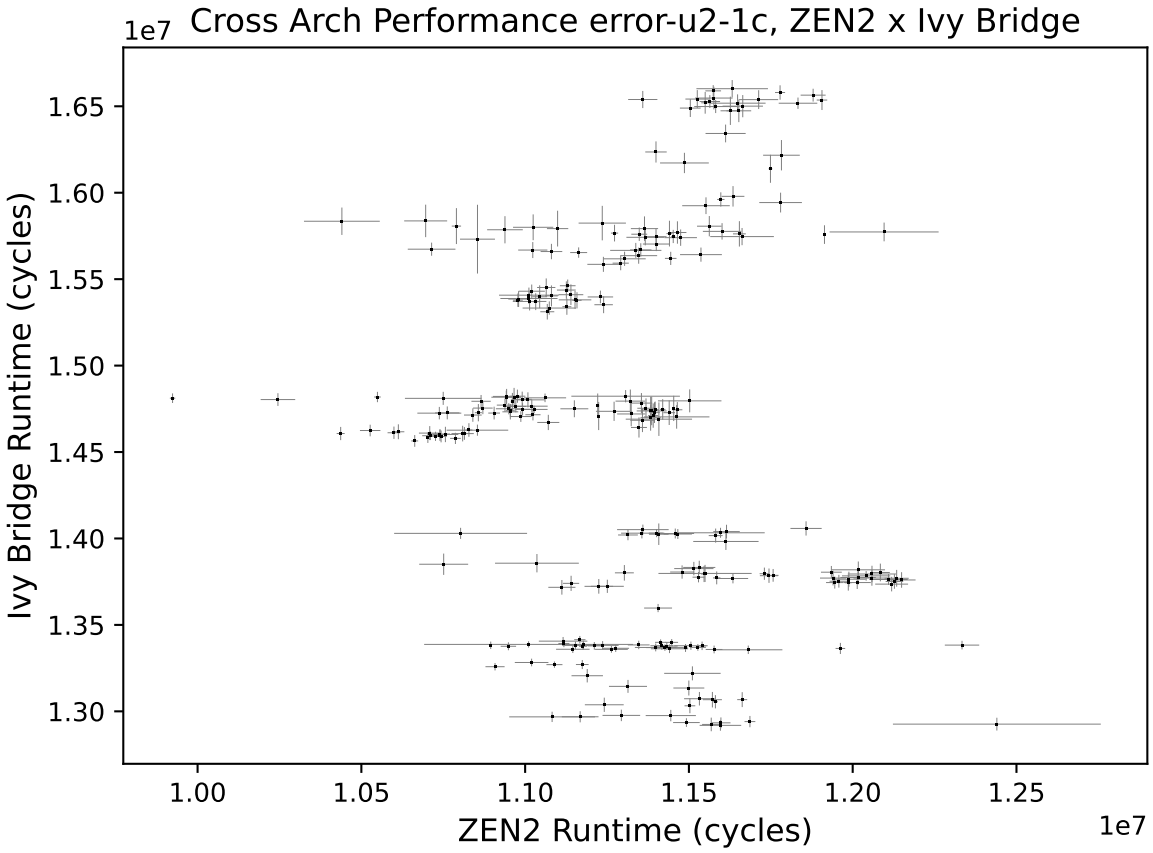

Figure 6.14: Scatter of schedule performance on both test systems for the error-u2-1c kernel collected over 4096 measurements of 2000 kernel iterations. Error bars depicts standard deviation.

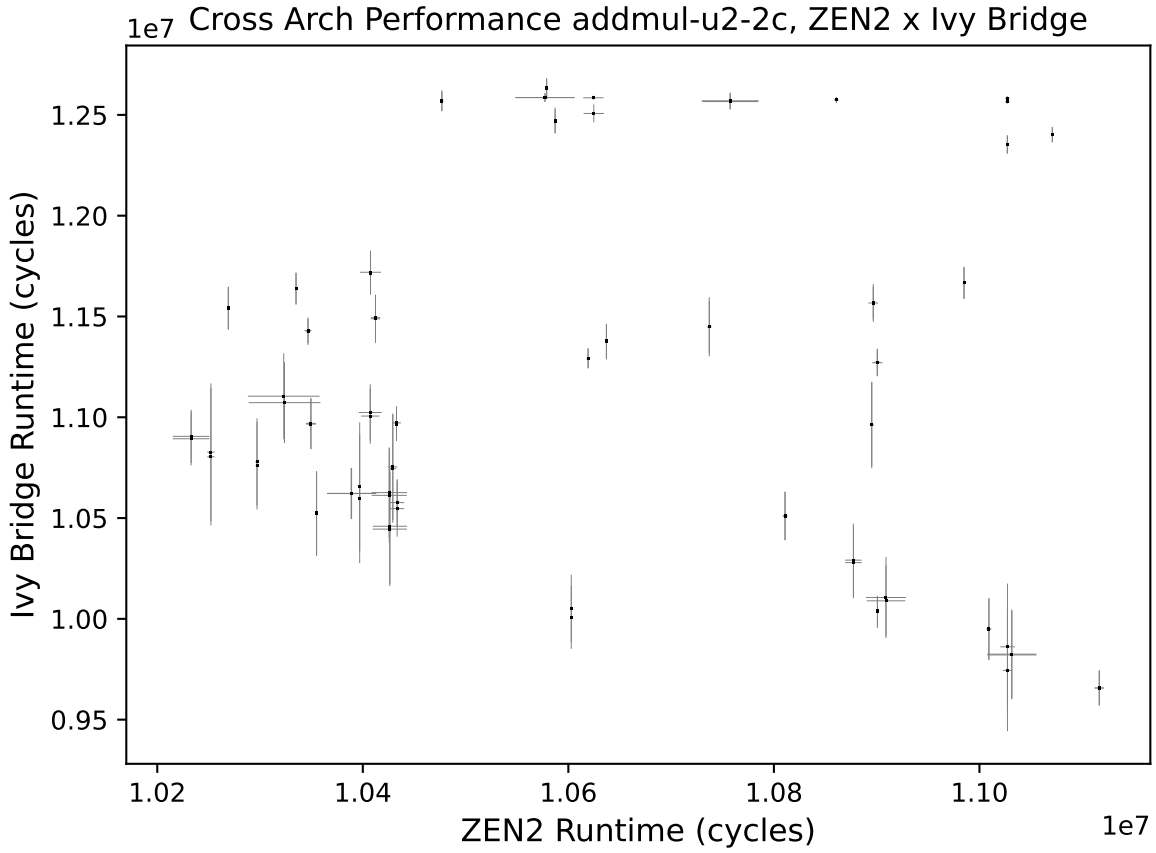

Figure 6.15: Scatter of schedule performance on both test systems for the addmul-u2-2c kernel collected over 4096 measurements of 2000 kernel iterations. Error bars depicts standard deviation.

6.6 Exploration of Variance

We wish to determine the reasons behind the differences in runtime among schedules of the same kernel. This knowledge could help compiler developers improve instruction schedulers for specific microarchitectures. Along with runtime data, we gather various features such as instruction and data cache misses, frontend dispatch stalls and back pressure (issue stalling). These features allow us to pinpoint if bottlenecks are located in the frontend or backend of the processor or whether they are caused by the memory subsystem. From this initial observation we may investigate further.

Table 6.5 shows the spread of measured feature means among schedules for each tested kernel. The magnitude of the feature's spread compared to runtime spread allow us to determine the importance of the feature. It quickly becomes obvious that both the instruction and data cache misses and frontend stalls offer no predictive capability over the large spread in runtime. Even in a highly conservative scenario, where a single cache miss causes a 1000 cycle delay, the low order of magnitude and low spread on cache misses do not offer explanations of variance within the millions of cycles. Additionally, there is high variance within the measurements of these features even when

grouping by schedule, shown by the partial eta-squared values resulting from Welch ANOVA shown in Table 6.6. These give an estimate of how much of the variance in feature measurements might be caused by scheduling instead of plain measurement error. Only backend stalls have an adequate magnitude to explain the variance among schedules. This is the same on the Intel test system. The correlation coefficients between backend stalls and runtime cycles for both test systems is shown in Table 6.7. Correlation between stalls due to backpressure is very high for all kernels that show high variance and the magnitude of such stall cycles measures up to the magnitude of runtime differences.

We have determined that variance among schedules is correlated to resource stalls after the dispatch queue for all kernels with large effect sizes. We can now further investigate which exact resource limits are causing these stalls.

For the both the AMD Zen 2 and Intel Ivy Bridge systems the highest correlation of runtime is with stalls due to limited reservation station space. No other resources such as register space, load or store queues seem to show any significant correlation or sufficient magnitude. On the AMD Zen 2 architecture each execution port has its own reservation stations. We can measure the amount of stalls on each of these reservation stations to more accurately pin down the resource forming the bottleneck.

A large amount of stalls always seem to be generated on a single ALU reservation station. As shown in Figure 6.16, there seems to be a linear relationship between the runtime and the amount of stalls on the ALU-4 queue. This relationship is disrupted by a smaller second clusters of schedules which generate increased number of stalls on a different ALU. The amount of stalls on the ALU-4 queue is an order of magnitude higher than that on the other ALUs. This indicates that the ALUs are being unequally loaded causing lower overall instruction throughput. Some schedules strike a better balance between the ALUs resulting in less overall stalls.

This effect is also seen on the Intel system, as shown in Figure 6.17. Take note that in this case we measure the amount of μops through execution port 1. As the amount of μops passing through port 1 increases overall performance decreases. Figure 6.18 shows an opposite relation for port 0. Schedules with a high load on port 0 tend to perform better. This again indicates that certain schedules strike a better load balance between execution ports increasing overall performance.

Clustering of schedules is more pronounced on the Ivy Bridge architecture. This clustering effect is likely caused by the CPU scheduler shifting around work to different execution ports. We are seeing interactions between the CPU scheduler and the instruction orders it receives.

Table 6.5: Spread of measured features for all kernels on the AMD platform.

Kernel	Runtime (cycles)	L1D MISS	L1I MISS	Frontend Stalls (cycles)	Backend Stalls (cycles)
error-u2-2c	[9.473e6, 1.254e7]	[118, 130]	[0, 6]	[2069, 2078]	[1.216e6, 4.278e6]
error-u2-1c	[9.924e6, 1.244e7]	[126, 137]	[1, 4]	[2071, 3065]	[1.673e6, 4.178e6]
3dot	[3.498e6, 5.307e6]	[112, 123]	[0, 1]	[2073, 2077]	[2.444e6, 4.246e6]
add-u4-c4	[3.267e6, 3.458e6]	[66.5, 81.5]	[0, 5]	[1067, 2073]	[1.190e6, 1.380e6]
addmul-u2-1c	[1.047e7, 1.115e7]	[131, 139]	[0, 2]	[2074, 2077]	[2.213e6, 2.887e6]
addmul-u2-2c	[1.023e7, 1.112e7]	[130, 136]	[1, 3]	[2078, 4079]	[1.980e6, 2.859e6]
addmul-u4-1c	[4.118e6, 6.399e6]	[66, 106]	[0, 6]	[1063, 3072]	[2.046e6, 4.327e6]
adddiv-u2-2c	[1.502e8, 1.502e8]	[393, 512]	[1, 7]	[2082, 2105]	[1.460e8, 1.461e8]
errorfloat-u2-2c	[1.229e7, 1.230e7]	[130, 146]	[1, 3]	[2074, 2081]	[4.032e6, 4.045e6]
errorfloat-u2-1c	[2.446e7, 2.447e7]	[196, 216]	[0, 5]	[2090, 2097]	[1.621e7, 1.622e7]
3dot-float	[3.540e6, 3.542e6]	[109, 115]	[0, 3]	[2083, 2092]	[1.484e6, 1.492e6]

Table 6.6: Partial eta-squared values for features resulting from Welch ANOVA grouping by schedule for the AMD system

Kernel	Runtime (cycles)	L1D MISS	L1I MISS	Frontend Stalls (cycles)	Backend Stalls (cycles)
error-u2-2c	0.9905	0.0113	0.2111	0.0781	0.9904
error-u2-1c	0.9781	0.0076	0.1116	0.5664	0.9782
3dot	0.9787	0.0206	0.0448	0.0161	0.9785
add-u4-c4	0.991	0.0033	0.1278	0.8792	0.9909
addmul-u2-1c	0.9823	0.0076	0.0584	0.0159	0.9824
addmul-u2-2c	0.9983	0.0028	0.0762	0.9813	0.9982
addmul-u4-1c	0.968	0.007	0.1836	0.8591	0.968
adddiv-u2-2c	0.0002	0.0375	0.1403	0.0084	0.0002
errorfloat-u2-2c	0.3146	0.0061	0.0412	0.0206	0.4006
errorfloat-u2-1c	0.8823	0.0073	0.1277	0.0204	0.5798
3dot-float	0.3591	0.0014	0.1282	0.0197	0.4297

Table 6.7: Kendall rank correlation coefficients on runtime.

Kernel	Frontend Stalls Intel	Backend Stalls Intel	Frontend Stalls AMD	Backend Stalls AMD
error-u2-2c	-0.0102	0.9940	0.0198	0.9925
error-u2-1c	0.0271	0.9957	0.0609	0.9945
3dot	0.0392	0.9751	0.1748	0.9245
add-u4-c4	0.0385	0.8375	-0.0041	0.9687
addmul-u2-1c	-0.0735	1.0000	0.0170	0.9853
addmul-u2-2c	-0.3906	0.9940	-0.0334	0.9829
addmul-u4-1c	0.2736	0.9950	0.0018	0.9945
adddiv-u2-2c	0.0558	0.0205	0.0194	0.0781
errorfloat-u2-2c	0.0393	0.7702	-0.0020	0.5330
errorfloat-u2-1c	-0.0637	0.2544	-0.0028	0.7319
3dot-float	0.5482	-0.0365	0.2775	-0.1576

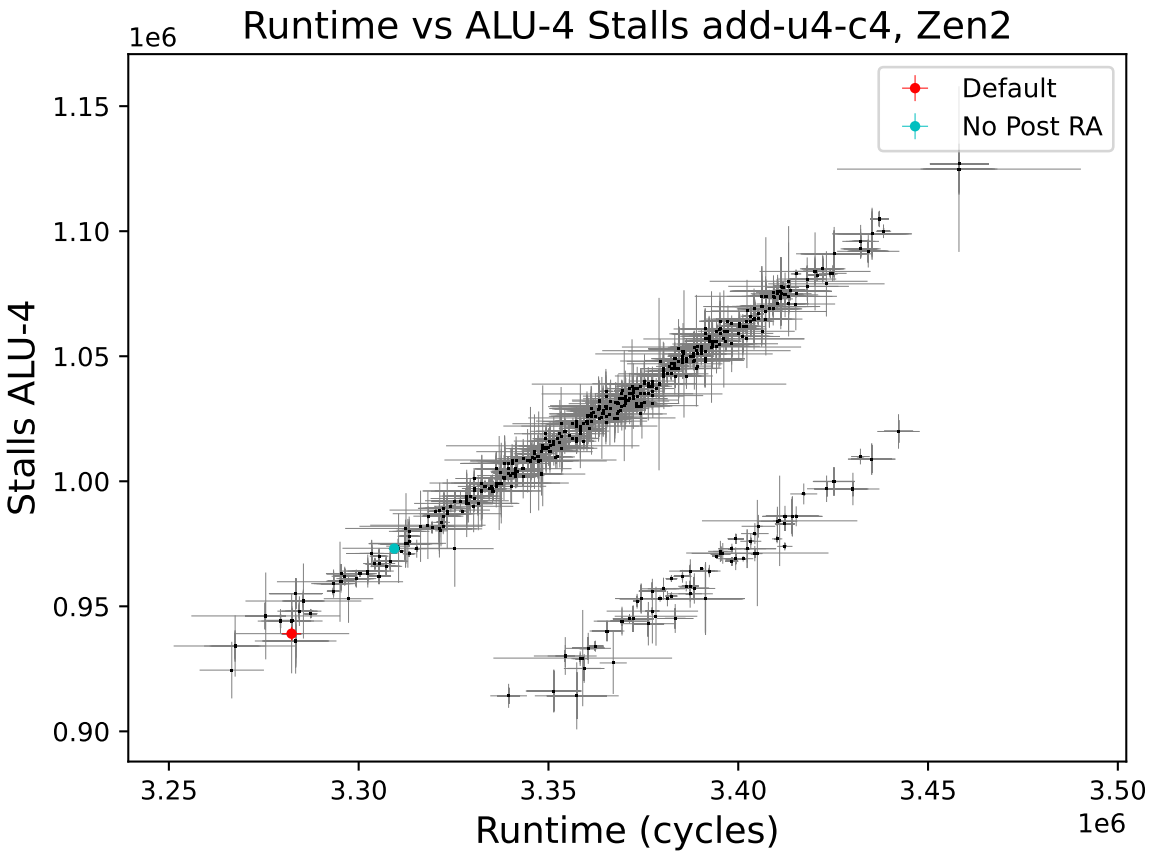

Figure 6.16: Scatter of schedule runtime against the amount of stalls on the ALU-4 reservation station. add-u4-c4 kernel, 128 measurements, 1000 iterations per schedule, AMD Zen 2 architecture. LLVM default schedule is marked in red.

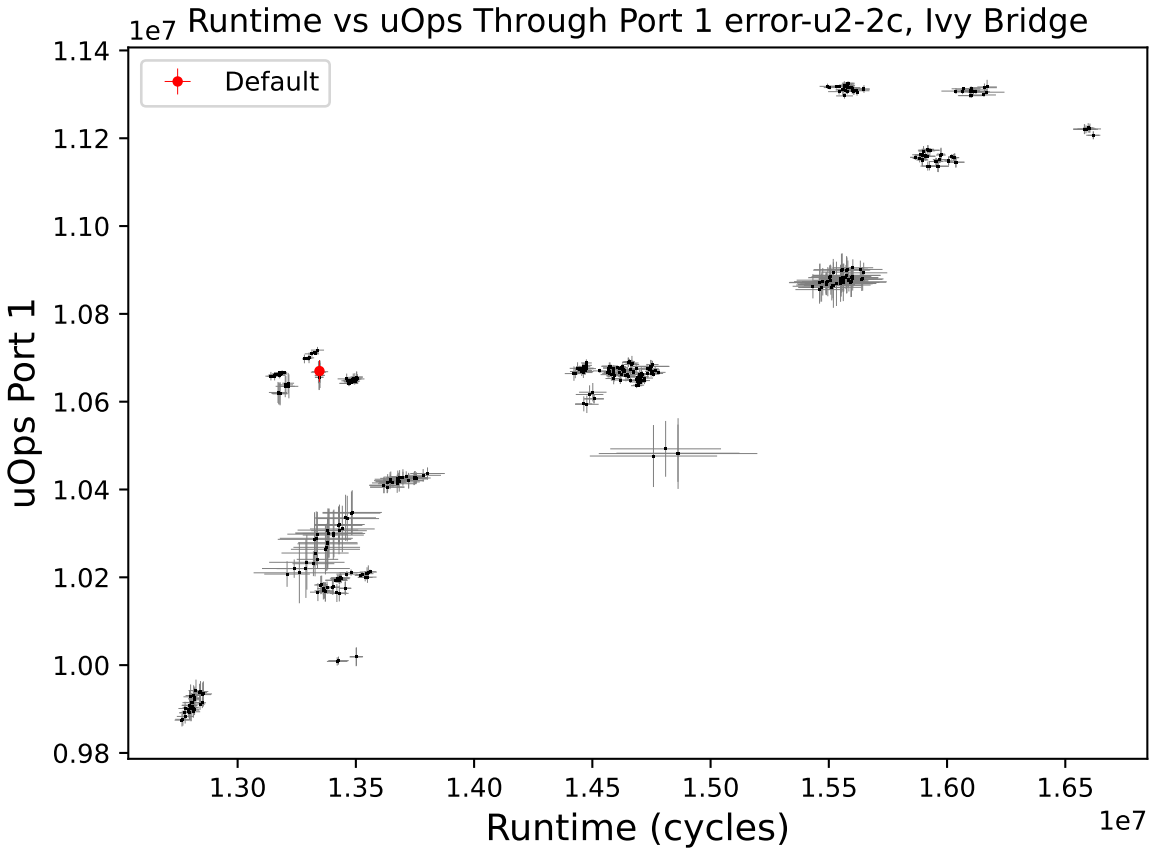

Figure 6.17: Scatter of schedule runtime against the amount μops handled by execution port 1. error-u2-2c kernel, 4096 measurements, 2000 iterations, Intel Ivy Bridge architecture. LLVM default schedule is marked in red.

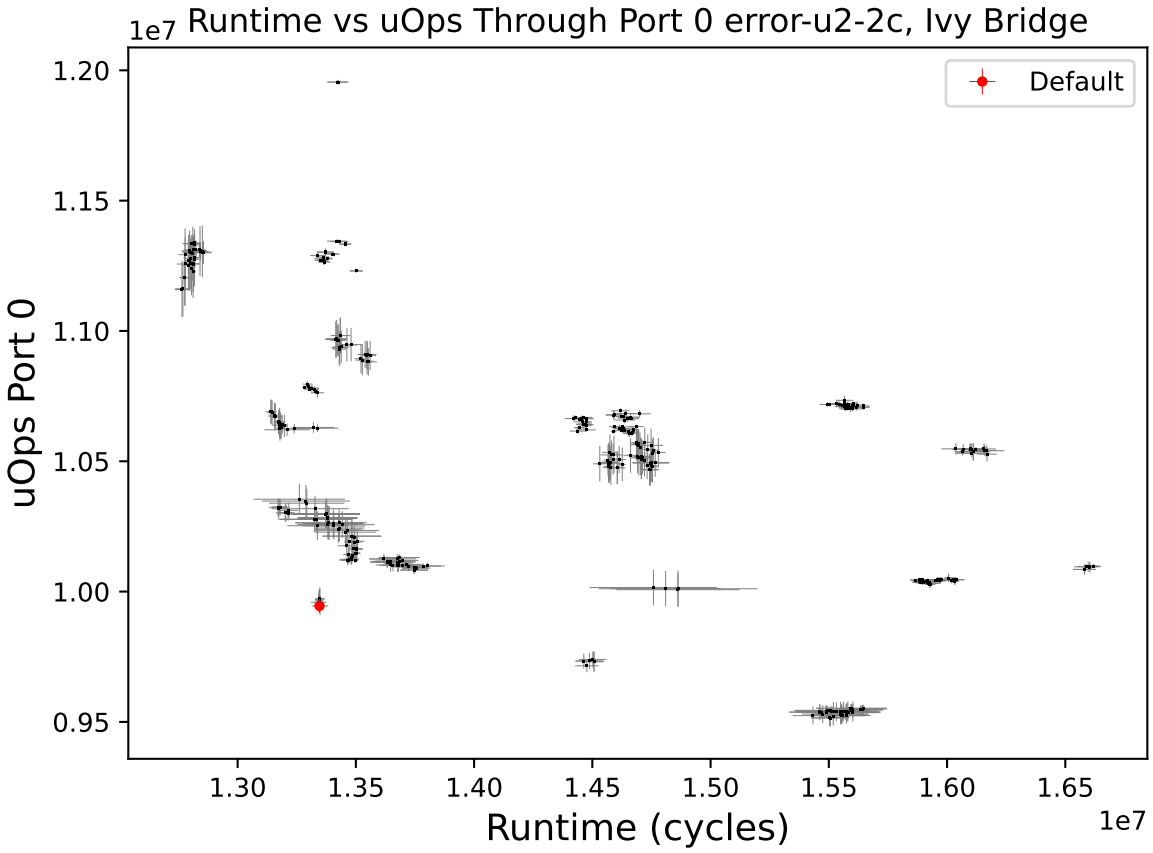

Figure 6.18: Scatter of schedule runtime against the amount μops handled by execution port 0. error-u2-2c kernel, 4096 measurements, 2000 iterations, Intel Ivy Bridge architecture. LLVM default schedule is marked in red.

Chapter 7

Limitations

Preferably a larger variety of micro architectures implementing different ISAs would also be tested to strengthen our results. Our tool based on LLVM would allow this to be done with relative ease in future work. Our experiments were conducted on two x86 microarchitectures. However, many x86 microarchitectures exist. A larger set of x86 microarchitectures would also strengthen our results for cross architecture optimization in Section 6.5.

As architectures are free in their implementation of their ISAs we can not account for instructions being transformed to multiple μops. Schedulers may only generate the order of the instructions not the μops. Therefore, the total control we have over the order in which μops enter the CPU's schedulers is limited for CISC architectures. It is likely that the set of total valid schedules is greater when compiled for a RISC ISA compared to a CISC architectures like x86. Similarly kernels of higher complexity will result in a higher number of valid schedules. This will increase the runtime of our data collection methods. Our tool could be extended with a method that prunes uninteresting parts of the scheduling tree to speed up data collection.

While we isolate the core on which our experiment is running using the *isolcpus* Linux kernel flag, hardware interrupts could still be served to this core. This could skew data for very small sample sizes since it disturbs the cache and branch predictors. It might be possible to remedy this with a custom OS-kernel but we did not deem it necessary.

While the limited register space on x86 was not constraining for our register allocator, described in Section 5.3, kernels of higher complexity may present a problem and generate assembly code with a lot of register spill instructions that modern compilers would otherwise not generate.

Chapter 8

Conclusions

In this thesis, we have presented a tool to collect performance data on valid instruction schedules for given compute kernels. Using this tool, we show that significant performance differences exist between instruction schedules of the same compute kernel. The differences are especially significant for kernels containing low latency operations. Furthermore, not only does performance vary between schedules, the stability of this measured performance also varies between schedules.

We have determined that these differences are mostly caused by resource stalls after the dispatch queue for all kernels with large effect sizes. Certain schedules result in a better balance of computational load through the various execution ports increasing performance.

We show that contemporary instruction schedulers in LLVM do not generate optimal schedules in many cases. For specific kernels, schedules of up to 29% faster exist. The magnitude of the performance difference varies between architectures. Moreover, in our research we discovered that optimal schedules on one micro architecture can be mediocre for another.

The tool presented in this thesis can help explore and understand the optimization space. Moreover, our tool allows for a similar analysis to be done for other ISAs such as ARM and POWER. Overall, our findings stress that instruction scheduling is still very relevant on modern OOO-processors. Our tool may used directly in the high performance computing (HPC) domain where the initial time investment to find optimal schedules for crucial kernels pays off for long computations. We see opportunities for improving commonly used instruction schedulers.

Chapter 9

Future Work

The tool and experimental results presented in this thesis open up many avenues for further work.

Firstly, our tool could be used to experiment on a larger set of micro architectures implementing a variety of ISAs. Especially implementations of RISC ISAs, such as ARMv8 or RISC-V, are of interest as their instructions are usually not translated to internal μops. This may result in a larger amount of valid operation orders.

Our tool may be extended with methods to efficiently search for good or optimal schedules by pruning the scheduling tree based on the collected performance data and to automatically derive which decisions lead to better performance. In turn this knowledge can be used to improve instruction schedulers. A variety of pattern recognition approaches could be tried to compare the decision trees of optimal schedules to those of sub optimal ones, in order to recognize common patterns of bad scheduling behavior. A trained model may be used to judge schedule permutations of specific kernel classes without the need for execution.

As noted in Section 6.5, the creation of a scheduler that generates schedules that represent fair trade offs for cross architectural performance could be an interesting research direction related to multi-objective optimization.

As noted in Section 4.1, cross thread interference could make for an interesting research direction. Perhaps certain pairs or sets of schedules provide better overall performance than others when run simultaneously as hardware threads on the same core. This could be extended by trying to find optimal pairs of schedules, where each schedule represent a different kernel.

Bibliography

[1] Ivy bridge - microarchitectures - intel. URL: https://en.wikichip.org/wiki/intel/microarchitectures/ivy_bridge_(client).

[2] Zen 2 - microarchitectures - amd. URL: https://en.wikichip.org/wiki/amd/microarchitectures/zen_2.

[3] 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*. Citeseer, 1996.

[4] Chia-Ming Chang, Chien-Ming Chen, and Chung-Ta King. Using integer linear programming for instruction scheduling and register allocation in multi-issue processors. *Computers & Mathematics with Applications*, 34(9):1–14, 1997.

[5] Arnaldo Carvalho De Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, volume 18, pages 1–42, 2010.

[6] Rens Dofferhoff. Rensdofferhoff/schedulepermuter: Schedulepermuter. URL: https://github.com/RensDofferhoff/SchedulePermuter/.

[7] M Anton Ertl and Andreas Krall. Optimal instruction scheduling using constraint logic programming. In *Programming Language Implementation and Logic Programming: 3rd International Symposium, PLILP'91 Passau, Germany, August 26–28, 1991 Proceedings 3*, pages 75–86. Springer, 1991.

[8] Dave Estes. 2014 llvm developers' meeting: "adding and optimizing a subtarget for mischeduler ", 2014. URL: https://www.youtube.com/watch?v=Uge_whHVmtM.

[9] Paolo Faraboschi, Joseph A Fisher, and Cliff Young. Instruction scheduling for instruction level parallel processors. *Proceedings of the IEEE*, 89(11):1638–1659, 2001.

[10] Agner Fog. Optimizing subroutines in assembly, Jan 2021. URL: https://www.agner.org/optimize/optimizing_assembly.pdf.

[11] Agner Fog. Optimizing subroutines in assembly, Nov 2022. URL: https://www.agner.org/optimize/instruction_tables.pdf.

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

[13] John L Hennessy and Thomas R Gross. Code generation and reorganization in the presence of pipeline constraints. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 120–127, 1982.

[14] Anne M Holler. Optimization for a superscalar out-of-order machine. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pages 336–348. IEEE, 1996.

[15] Jie S Hu, Narayanan Vijaykrishnan, and Mary Jane Irwin. Exploring wakeup-free instruction scheduling. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, pages 232–232. IEEE, 2004.

[16] Mariska IJpelaar. An initial exploration of the importance of program instruction order for dynamically scheduled processors. 2020.

[17] Timothy M Jones, Michael FP O'Boyle, Jaume Abella, and Antonio González. Software directed issue queue power reduction. In *11th International Symposium on High-Performance Computer Architecture*, pages 144–153. IEEE, 2005.

[18] Georgia Kouveli, Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. Exploring the benefits of randomized instruction scheduling, 2011.

[19] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[20] Rainer Leupers. Instruction scheduling for clustered vliw dsps. In *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00622)*, pages 291–300. IEEE, 2000.

[21] P Michaud and A Seznec. Data-flow prescheduling for large instruction windows in out-of-order processors. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 27–36. IEEE, 2001.

[22] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710. Citeseer, 1999.

[23] Arthur Perais and André Seznec. Eole: Combining static and dynamic scheduling through value prediction to reduce complexity and increase performance. *ACM Transactions on Computer Systems (TOCS)*, 34(2):1–33, 2016.

[24] Joseph J Sharkey, Dmitry V Ponomarev, Kanad Ghose, and Oguz Ergin. Instruction packing: reducing power and delay of the dynamic scheduling logic. In *ISLPED'05. Proceedings of the 2005 International Symposium on Low Power Electronics and Design, 2005.*, pages 30–35. IEEE, 2005.

[25] D. Tate, G. Steven, and F. Steven. Static scheduling for out-of-order instruction issue processors. In *Proceedings 5th Australasian Computer Architecture Conference. ACAC 2000 (Cat. No.PR00512)*, pages 90–96, 2000. doi:10.1109/ACAC.2000.824329.

[26] Deepak Vohra. *Apache Parquet*, pages 325–335. 09 2016. doi:10.1007/978-1-4842-2199-0_8.

[27] Kent Wilken, Jack Liu, and Mark Heffernan. Optimal instruction scheduling using integer programming. *Acm sigplan notices*, 35(5):121–133, 2000.

Appendix A

Tool Usage and Compilation

This Appendix describes how to compile the tools and execute analyses made in this thesis.

A.1 Compilation

Our data collection tool named `SchedulePermuter` is based on LLVM 14.0.5. Its only dependencies are Apache Parquet [26], cmake and ninja. Most package managers provide packages for these dependencies. To compile our tool on Unix-like operating systems for the X86 architecture copy our source folder or clone our git repository [6] and run the following commands from the top of the folder:

```
mkdir llvm/build
cd llvm/build/
cmake -DCMAKE_BUILD_TYPE=Debug -DLLVM_TARGETS_TO_BUILD=X86 \
        -DBUILD_SHARED_LIBS=ON ../ -GNinja
ninja SchedulePermuter
```

The `SchedulePermuter` binary can now be found in `build/bin/`.

A.2 Usage

The tool can be used in the following way:

```
Tool Usage:
-view-mbb: Shows all the Machine Basic Blocks for the target function and quits
-target-function-symbol: required function symbol target
    for the machine instruction schedulers
```

```
-target-basic-block: optional specification of the
    target basic block of the function
-prep-function-symbol: optional function symbol target that is run
    before a measurement to prepare/allocate initial data
-IR-file: Necessary path to file containing compute and prepare symbol
-output-dir: Necessary path to output directory
-papi-event-list: , separated list of PAPI events to measure for each schedule
-num-exp: Number of measurements per schedule
-exp-iters: Number of iterations in a measurement
-prep-iters: Number of iterations executed in preparation of an measurement
-v: Verbose

example:
SchedulePermuter -target-function-symbol compute -IR-file test.ll \
    -output-dir out -papi-event-list PAPI_TOT_CYC,PAPI_TLB_IM \
    -prep-function-symbol prepare
```

The tool takes as input an LLVM IR input file containing the kernel function. Such a file can be generated from many languages but for a simple C kernel it can be done with a command like this:

```
clang -O1 -march=native  -S -emit-llvm kernel.c -o kernel.ll
```

We specify for which function symbol and optionally basic block we wish to generate and test all valid schedules, the symbol name for the kernel can be found in the `.ll` file. It will usually be the same as the function name. We specify a comma separated list of PAPI counters to collect, such as the amount of clock cycles for runtime measurement or the amount of cache misses. We can define the amount of measurements and iterations we take of each generated schedule. Besides the compute kernel to permute we may specify a data initialization function using the `prep-function-symbol` option, this function will be called before each measurement.

The program will output a DAG for each machine basic block in the target function. To just generate these DAGs and print the generated machine basic blocks use the `view-mbb` option. This can help you find the machine basic block number you wish to permute the schedule for.

Note that permissions may be necessary to access certain performance counters. Run the tool under the root user or alter the needed permissions for access. On Linux this can be done:

```
sudo sh -c 'echo -1 >/proc/sys/kernel/perf_event_paranoid'
```

As described in Section 5.5 on our test platforms multi-threading is disabled and a single static frequency is set, how to do this varies among systems. The tool sets its processor scheduling affinity to processor/core 2. As described in Section 5.5 this core is isolated using the kernel flag `isolcpus=2` to guarantee our tool is the only process to use this resource.

A.3 Data Analysis Script

Our data analysis script to generate summary statistics can be used in the following way:

```
usage: python analyse.py [-h] [-p] [--scatter_feature SCATTER_FEATURE] \
[--scatter_feature2 SCATTER_FEATURE2] [-t] [--tree_feature TREE_FEATURE] \
[-s START_DEPTH] [-e END_DEPTH] input_dir

Process parquet output data folder.

positional arguments:
  input_dir             Input directory to process.

optional arguments:
  -h, --help            show this help message and exit
  -p, --plot            Make the scatter plots with all permutations.
  --scatter_feature SCATTER_FEATURE
                        Feature to scatter against permutation num.
  --scatter_feature2 SCATTER_FEATURE2
                        Feature to scatter against the first.
  -t, --tree            Render tree with given depth parameters.
  --tree_feature TREE_FEATURE
                        Feature to color the tree with.
  -s START_DEPTH, --start_depth START_DEPTH
                        Starting depth of tree render.
  -e END_DEPTH, --end_depth END_DEPTH
                        End depth of tree render.

Example:
python3 analyse.py . -p --scatter_feature PERF_COUNT_HW_CPU_CYCLES \
--scatter_feature2 PERF_COUNT_HW_CPU_CYCLES
-t -s 0 -e 4 --tree_feature PERF_COUNT_HW_CPU_CYCLES
```

Appendix B

Kernels

In this thesis, we have collected data on all instruction schedules of many different kernels. This Appendix shows all the kernels used in our experiments described in Section 6.

B.1 error-u2-2c

```
#define N 4096
int array[N];
int mean;

void prepare() {
    for (int i = 0; i < N; i++) {
        array[i] = i;
        mean += i;
    }
    mean /= N;
}

long compute() {
    long sum, sum2 = 0;
    for (int i = 0; i < N; i += 2)
    {
        sum += (array[i] - mean) * (array[i] - mean);
        sum2 += (array[i+1] - mean) * (array[i+1] - mean);
    }
    return sum + sum2;
}
```

B.2 error-u2-1c

```
#define N 4096
int array[N];
int mean;

void prepare() {
    for (int i = 0; i < N; i++) {
        array[i] = i;
        mean += i;
    }
    mean /= N;
}

long compute() {
    long sum = 0;
    for (int i = 0; i < N; i += 2)
    {
        sum += (array[i] - mean) * (array[i] - mean);
        sum += (array[i+1] - mean) * (array[i+1] - mean);
    }
    return sum;
}
```

B.3 errorfloat-u2-2c

```
#define N 4096
float array[N];
float mean;

void prepare() {
    for (int i = 0; i < N; i++) {
        array[i] = i;
        mean += i;
    }
    mean /= N;
}

long compute() {
    float sum, sum2 = 0;
    for (int i = 0; i < N; i += 2)
    {
```

```
        sum += (array[i] - mean) * (array[i] - mean);
        sum2 += (array[i+1] - mean) * (array[i+1] - mean);
    }
    return sum + sum2;
}
```

B.4 errorfloat-u2-1c

```
#define N 4096
float array[N];
float mean;

void prepare() {
    for (int i = 0; i < N; i++) {
        array[i] = i;
        mean += i;
    }
    mean /= N;
}

long compute() {
    float sum = 0;
    for (int i = 0; i < N; i += 2)
    {
        sum += (array[i] - mean) * (array[i] - mean);
        sum += (array[i+1] - mean) * (array[i+1] - mean);
    }
    return sum;
}
```

B.5 3dot

```
#define N 1500
int vectors1[N];
int vectors2[N];
int res[N];

void prepare() {
    for (int i = 0; i < N; i++) {
        vectors1[i] = i;
        vectors2[i] = i;
```

```
    }
}

long compute() {

    for (int i = 0; i < N; i += 3)
    {
        res[i] = vectors1[i] * vectors2[i] + vectors1[i + 1] * \
         vectors2[i + 1] + vectors1[i + 2] * vectors2[i + 2];
    }
    return 0;
}
```

B.6 3dot-float

```
#define N 1500
float vectors1[N];
float vectors2[N];
float res[N];

void prepare() {
    for (int i = 0; i < N; i++) {
        vectors1[i] = i;
        vectors2[i] = i;
    }
}

long compute() {

    for (int i = 0; i < N; i += 3)
    {
        res[i] = vectors1[i] * vectors2[i] + vectors1[i + 1] * \
        vectors2[i + 1] + vectors1[i + 2] * vectors2[i + 2];
    }
    return 0;
}
```

B.7 add-u4-c4

```
#define N 4096
int array[N];
```

```
void prepare() {
    for (int i = 0; i < N; i++)
        array[i] = i;
}

long compute() {
    long sum0, sum1, sum2, sum3 = 0;
    for (int i = 0; i < N; i += 4)
    {
        sum0 += array[i];
        sum1 += array[i+1];
        sum2 += array[i+2];
        sum3 += array[i+3];
    }
    return sum0 + sum1 + sum2 + sum3;
}
```

B.8 addmul-u2-1c

```
#define N 4096
int array[N];

void prepare() {
    for (int i = 0; i < N; i++)
        array[i] = i;
}

long compute() {
    long sum = 0;
    for (int i = 0; i < N; i += 2)
    {
        sum += array[i] * 123;
        sum += array[i+1] * 123;
    }
    return sum;
}
```

B.9 addmul-u2-2c

```
#define N 4096
```

```
int array[N];

void prepare() {
    for (int i = 0; i < N; i++)
        array[i] = i;
}

long compute() {
    long sum,sum2 = 0;
    for (int i = 0; i < N; i += 2)
    {
        sum += array[i] * 123;
        sum2 += array[i+1] * 123;
    }
    return sum + sum2;
}
```

B.10 addmul-u4-1c

```
#define N 4096
int array[N];

void prepare() {
    for (int i = 0; i < N; i++)
        array[i] = i;
}

long compute() {
    long sum = 0;
    for (int i = 0; i < N; i += 4)
    {
        sum += array[i] * 123;
        sum += array[i+1] * 123;
        sum += array[i+2] * 123;
        sum += array[i+3] * 123;
    }
    return sum;
}
```

B.11 adddiv-u2-2c

```
#define N 4096
int array[N];
int div;

void prepare() {
    for (int i = 0; i < N; i++)
        array[i] = i;
    div = 3;
}

long compute() {
    long sum,sum2 = 0;
    for (int i = 0; i < N; i += 2)
    {
        sum += array[i] / div;
        sum2 += array[i+1] / div;
    }
    return sum + sum2;
}
```