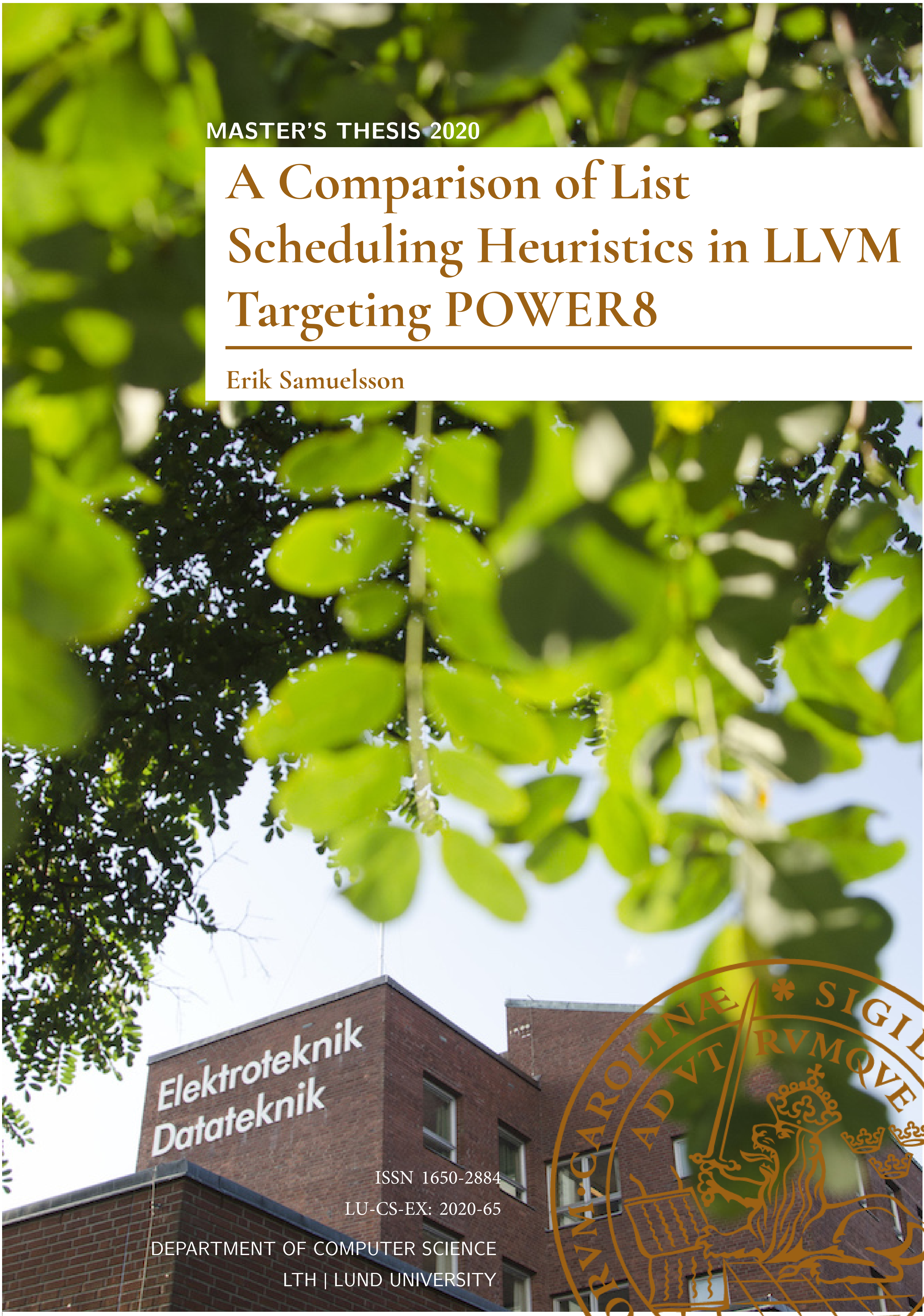


MASTER'S THESIS 2020

A Comparison of List Scheduling Heuristics in LLVM Targeting POWER8

Erik Samuelsson

ISSN 1650-2884
LU-CS-EX: 2020-65

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-65

A Comparison of List Scheduling Heuristics in LLVM Targeting POWER8

En jämförelse av olika heuristik för instruktionsschemaläggning i LLVM på POWER8

Erik Samuelsson

A Comparison of List Scheduling Heuristics in LLVM Targeting POWER8

Erik Samuelsson
dat13esa@student.lu.se

December 8, 2020

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se

Examiner: Flavius Gruian, flavius.gruian@cs.lth.se

Abstract

Instruction scheduling is an important part of an optimizing compiler and the fundamental algorithm for this task is list scheduling, which operates on one basic block at a time. List scheduling performs a pairwise comparison and utilizes heuristics in order to decide which instruction to schedule next. While most of the recent compiler research on instruction scheduling is focused on optimal solutions, list scheduling continues to be widely used in practice since the algorithm is fast and simple to implement.

In this thesis different list scheduling heuristics, both traditional and new, have been compared against each other, implemented in LLVM and targeting POWER8. The heuristics were evaluated on a subset of the SPEC CPU 2017 benchmarks, and the original program order was used as baseline. A new heuristic, specific to POWER8, was developed. Based around instruction dispatch, this dynamic heuristic tries to produce a schedule which hardware can more easily take advantage of.

Keywords: Instruction scheduling, list scheduling, optimizing compilers

Acknowledgements

I would like to thank Jonas Skeppstedt for introducing me to the wonderful topic of optimizing compilers. I would also like to thank Flavius Gruian for his many helpful comments on how to improve this thesis.

Contents

Chapter 1
Introduction

The task of a compiler is to translate a program written in one programming language into another. Translation is organized into phases, divided between the front-end, middle section and back-end [17]. In short, the phases that belong to the front-end will read the program text, analyze the syntax and semantics to make sure that the input is a correct and a valid program. The program is then translated into an intermediate representation, as a first step of translating the program into the target language. The translation of the intermediate representation into target code is done by the back-end. At the very basic level, the translation has to be correct. When correctness is maintained, we can start to think about how we could make the translated program faster, or more efficient. Improving the performance of the code that is being generated is the task of an optimizing compiler. Various metrics can be used to measure performance, such as execution time, static code size, energy consumption and so on [24]. In this thesis we are concerned with execution time. There are numerous optimizations which affect execution time, many of these are applied as code transformations in the middle section, but some of the most important optimizations with regards to execution time can be found in the back-end. The reason for this is that these optimizations are operating closer to the hardware, and while it is possible to write code at this low level, in most cases compilers are more suitable for this task [24].

In this thesis we will look at instruction scheduling, which is an optimization that belongs to the back-end. The purpose of instruction scheduling is to increase instruction level parallelism or ILP, that is, the number of instructions that the processor executes in parallel. Instruction scheduling is an important task of any optimizing compiler, but finding an optimal schedule is an NP-complete problem [9]. The traditional approach to instruction scheduling uses heuristics to find a good schedule and the fundamental algorithm for this task is list scheduling [7, 28]. It is a simple algorithm which is straightforward to implement, but the outcome of the schedule relies on the chosen heuristics. A heuristic returns either a numerical or Boolean value and essentially there are two ways to use heuristics to select which instruction we should schedule next [5, 26]. For example, we can have a function which takes as input an instruction, and assigns to it a value which depends on one or more heuristics.

Then, during scheduling we simply select the instruction which has the highest value of all the available instructions. Or, we can have a number of heuristics in a hierarchical order and then during scheduling, every time that we are faced with multiple instructions to select from, we do a pairwise comparison of the instructions. We compare the instructions against the heuristics, one heuristic at a time, and whenever a heuristic is able to differentiate between two instructions, one instruction is marked as being the best candidate to schedule next. This candidate is then compared against the next instruction and so on. The latter approach is the one that is used in this thesis. Many heuristics have been suggested for instruction scheduling, and sometimes their use is overlapping, which makes it possible to organize them into categories [26]. This can be useful when comparing existing heuristics or when developing new ones.

1.1 Research Questions

List scheduling is one of the early algorithms in the history of instruction scheduling [14, 19]. And while the focus of compiler research has moved on to other algorithms, list scheduling is still used in state of the art compilers of today, for example LLVM. Because of this I would like to revisit the algorithm, and specifically look into how different heuristics perform on a modern processor. The metric which we will use to evaluate the performance of a heuristic is execution time, or more precisely, the amount of work per unit of time. In this thesis we will compare different heuristics used by list scheduling, and try to obtain new insights about the algorithm and the heuristics. The following research questions are investigated:

- *How do individual heuristics compare against each other?*
- *Is there any category of heuristics that seem to perform better, or worse, than the others?*
- *For different combinations of heuristics, what can be said about the order in which the heuristics appear?* A combination of heuristics is a set of heuristics in a hierarchical structure.

1.2 Contributions

The results of the experimental evaluation can be used as a guideline by people who are interested in developing heuristics, for example when making a choice between one heuristic or another. It can also be used as an indication whether this type of optimization is worth it or not, for example, in some cases it might simply be better to avoid it. The dispatch heuristic that was implemented can be used as is, or used as a basis for further development of heuristics specific to the POWER8 architecture.

Chapter 2
Background

This chapter will describe instruction scheduling in more detail and define what basic blocks are. It will also provide some background on LLVM and POWER8.

2.1 Instruction Scheduling

A compiler is typically organized into phases where each phase except for the first one, takes as input, the output of the previous phase [17]. The first phase, known as *lexical analysis*, reads the source code and divides the program into *tokens*, which corresponds to keywords, variable names, operators and so on. The next phase performs a syntactic analysis in which tokens are parsed and organized into a *syntax tree*, a graph which represents the structure of the program. The syntax tree is then semantically analyzed in a third phase to assure that the program makes sense. Although phases are modeled sequentially, they can be weaved together, as is often the case of lexical analysis and syntactic/semantic analysis. These three phases constitute the front-end of a compiler. The program is then translated into an intermediate representation, or IR code, independent of the target machine. This phase is referred to as the middle section of a compiler, at this stage we can apply target-independent code optimizations to the program [24]. The back-end of a compiler translates the IR code into target-dependent machine code, a process known as code generation. This includes instruction selection, instruction scheduling and register allocation. During instruction selection the IR code will be matched to an equivalent sequence of target-dependent instructions. These instructions are then scheduled in a way which increases throughput, and finally variables are being assigned to registers, during the register allocation phase.

Throughput refers to the rate at which a processor executes instructions. Execution of an instruction is divided into stages which corresponds to different parts of the processor. Common stages in a RISC architecture are: fetch, decode, execute, memory access and write back [10]. An instruction goes through each of these steps sequentially. The clock cycle is a unit of time during which each part of the processor can compute, and at the end of a cycle the results are stored in registers of the chip [25]. For example, if each step requires one clock cycle to compute, then it will take five clock cycles to execute one instruction. In pipelined execution the processor fetch a new instruction every clock cycle. While this could potentially mean a speed up of *n* times, where *n* is the number of stages in the pipeline, some complications arise. If an instruction needs a register value that is currently being updated by the previous instruction, reading the register would result in an outdated value. In order to read the correct value the later instruction has to wait for the value to be written to register first and we say that a pipeline stall has occurred. The purpose of instruction scheduling is to decrease the number of pipeline stalls, thus increase the processor throughput. This is achieved by reorganizing the instructions in a way such that independent instructions are executed between two instructions that would otherwise cause a pipeline stall. The fact that some instructions are requiring the same registers gives rise to three types of data dependencies which constrain execution order [24]. A true dependence occurs when an instruction is producing a value that a subsequent instruction will consume and the consumer has to wait for the producer. An anti dependence occurs when an instruction has to read a register before it is overwritten. The instruction that wants to write to the register in this case has to wait until it is safe to do so. An output dependence occurs because two instructions are writing to the same register and therefore must execute in their original order.

There is an interdependence between instruction scheduling and register allocation. A variable is said to be live if its value is needed by a subsequent instruction. A *live range* of a variable starts at a definition of the variable and ends at its last use. Instruction scheduling performed before register allocation may increase a live range of a variable, which may lead to more variables being live at the same time, thereby increasing the number of registers needed. This is called register pressure. On the other hand, instruction scheduling performed after register allocation faces another problem in that instructions may now have dependencies between them, caused by being assigned the same registers. If the number of live variables exceeds the number of available registers, some variables will be *spilled* to memory. *Spill code* refers to the additional store and load instructions introduced to a program in order to move a spilled variable to and from memory. It is possible to *split* a live range to avoid some spill code. This creates a new live range that then need to be handled by the register allocator. In cases where instruction scheduling is performed before register allocation, it may be necessary to perform instruction scheduling afterwards as well in order to schedule possible spill code.

2.2 Basic Blocks

In addition to translating code from one programming language to another, an optimizing compiler aims to increase the performance of the translated code. This is achieved by a series of optimizations, or code transformations, applied to the program as it passes through the various compiler phases. Many of these optimizations take place in the middle section [17]. The IR code of the middle section is an intermediate language, in LLVM for example, it bears a resemblance to an assembly language, but is target-independent. The initial step of the middle section is to create a *control flow graph*, or CFG, which represents the various paths that can be taken during the execution of a procedure [24]. A procedure is composed of a sequence of instructions, and each instruction is associated with a type that defines it. An instruction can be for example: a *branch instruction*, a *label*, a function call, an operation and so on. Nodes in the CFG are called *basic blocks*. A basic block is a region of code in a procedure, where the first instruction of the region is a label, and the last instruction is either a branch or return instruction. In the CFG there is an arc from node u to node v if there is a branch instruction in u, leading to the label that begins v.

The CFG is the starting point for various optimizations, including instruction scheduling. However, instruction scheduling is a target-dependent optimization and so before we can begin scheduling, the IR code of the middle section has to be translated into the intermediate representation of the back-end. When we schedule instructions we can either schedule them one basic block at a time, or, we can consider instructions from different basic blocks [19]. The former is called local scheduling and it is the approach used by list scheduling. The latter is called global scheduling, and it is out of scope for this thesis.

2.3 LLVM

LLVM is an open source optimizing compiler written in C++ [16]. In the context of traditional compiler design it is focused around the middle section and back-end. The main front-end is Clang, which compiles C and C++ source code into an intermediate representation called LLVM IR. The front-end, middle section and back-end are all independent of each other and connected via the LLVM IR. This means that various languages can be translated into LLVM IR, via different front-ends, utilize the target-independent code transformations applied by the middle section, and then be compiled, via different back-ends, to a number of different target machines. Target independent code transformations and optimizations are performed in passes over the LLVM IR. LLVM passes operate on different levels of the IR, for example functions or basic blocks. The following two code listings shows a simple example of the LLVM IR.

Listing 2.1: A recursive function which computes the *n*th number in the Fibonacci sequence.

```
int f(int n)
{
  if (n < 2)
    return n;
  else
    return f(n-1) + f(n-2);
}
```

Listing 2.1 shows a sample of C code and listing 2.2 shows the corresponding LLVM IR of the middle section. Global identifiers are prefixed with the @ symbol, whereas local identifiers use the prefix %. Identifiers are either named, for example *retval* or unnamed, in which case they are identified with an integer. In listing 2.2 we have four basic blocks, labeled as: *entry*, *if.then*, *if.else* and *return*. A comment in the code is preceded by semicolon, the comments after label names tell us the predecessor of this basic block in the CFG. Line number 4, `%retval = alloca i32, align 4`, allocates a 32 bit integer on the stack, pointed to by *retval*. The `load` and `store` instructions are used to read to and from memory. The `br` instruction is used to direct control flow. There are two types of branch instructions, conditional branches, for example the last instruction in the *entry* basic block (line 9), and unconditional branches, for example the branches in the following two basic blocks (lines 14 and 25).

Listing 2.2: LLVM IR corresponding to the C code in listing 2.1.

```
 1 ; Function Attrs: noinline nounwind optnone
 2 define dso_local signext i32 @f(i32 signext %n) #0 {
 3 entry:
 4   %retval = alloca i32, align 4
 5   %n.addr = alloca i32, align 4
 6   store i32 %n, i32* %n.addr, align 4
 7   %0 = load i32, i32* %n.addr, align 4
 8   %cmp = icmp slt i32 %0, 2
 9   br i1 %cmp, label %if.then, label %if.else
10
11 if.then: ; preds = %entry
12   %1 = load i32, i32* %n.addr, align 4
13   store i32 %1, i32* %retval, align 4
14   br label %return
15
16 if.else: ; preds = %entry
17   %2 = load i32, i32* %n.addr, align 4
18   %sub = sub nsw i32 %2, 1
19   %call = call signext i32 @f(i32 signext %sub)
20   %3 = load i32, i32* %n.addr, align 4
21   %sub1 = sub nsw i32 %3, 2
22   %call2 = call signext i32 @f(i32 signext %sub1)
23   %add = add nsw i32 %call, %call2
24   store i32 %add, i32* %retval, align 4
25   br label %return
26
27 return: ; preds = %if.else, %if.then
28   %4 = load i32, i32* %retval, align 4
29   ret i32 %4
30 }
```

Instruction scheduling operates on *machine basic blocks*. These blocks are the result of going from the target-independent LLVM IR, to a target-dependent intermediate representation. A machine basic block is a collection of machine instructions, and a machine basic block correspond to a LLVM IR basic block. In LLVM instruction scheduling is performed before, and after, register allocation. The heuristics in this thesis are implemented as part of the scheduling done pre-register allocation.

2.4 Group Formation

The dispatch heuristic that was implemented is specific to POWER8, and it is based on a part of instruction execution known as group formation. It is not necessary to know how instructions are executed by the processor to understand group formation, however, readers who are unfamiliar with the process might want to look at appendix A which provides an overview of the POWER8 processor core. We say that the processor executes instructions *out-of-order* if instructions can be executed in an order that is different from the order they appear in the program code. Instruction dispatch and group formation represents the last *in-order* part of the pipeline before out-of-order execution [23]. Instructions are organized into groups before dispatch. Instructions within the same group enter the out-of-order part of the processor in parallel. Depending on how many threads are currently active, there can be either one or two dispatch groups. In single thread mode the size of a group can be up to eight instructions, where group slots are numbered from 0 to 7. There can be up to six non-branch instructions, placed in slots 0 to 5. Slot 6 and 7 are used for branch instructions. In *simultaneous multithreading* mode with 2, 4 or 8 active threads, there are two dispatch groups, group 0 and group 1. These groups can contain up to three non-branch instructions and one branch instruction each. For group 0, non-branch instructions are placed in slots 0 to 2, branches in slot 6. For group 1, slots 3 to 5 are used for non-branches, and slot 7 for branches. When instructions are fetched from the instruction cache they are placed in buffers, before group formation. These buffers has 32 rows, and each row is four instructions wide. During group formation, in single thread mode, all instructions must come from the oldest two entries in the instruction buffer. With more than one active thread the, the buffer is divided between the threads and instructions must come from the oldest two entries per thread, but, a thread can only empty one quadword per cycle [13]. In other words, the instructions in group 0 and group 1 comes from different threads.

Group formation follows a set of rules. Some instructions are marked First, these must start a group. Likewise, some instructions are marked Last, these must end a group. Some instructions cannot be executed in a single internal operation, these instructions are *cracked* into multiple simple internal operations [13]. Cracked instructions are either two-way cracked, or three-way cracked and requires two, respectively three non-branch slots. It is possible to go beyond the first branch to pick up more instructions, however no floating point operations are allowed in a group after a branch. Instructions with three source operands can go into slot 0 or 1 and reserve slot 2 for the third operand. Or similarly, use slot 3 or 4 and reserve slot 5. Three-source operations are not allowed in slots 2 or 5. If a branch and link instruction updates the Link Register, it is marked as Last. If a conditional branch is followed by certain fixed-point or store instructions, that are allowed to be predicated, the `bc` instruction is marked as First. This transformation of the branch into predicated execution is known as *instruction fusion*, two adjacent instructions are fused into one internal operation. Another type of instruction fusion involve add-immediate instructions and certain load instructions. To be adjacent it is necessary that the two instructions are in the same dispatch group.

In single thread mode, instructions that are two-way cracked takes up slots 0 and 1. It is possible to fill out a second cracked instruction in the same group if the two instructions appear next to each other in the code stream. In that case, the second instruction takes up slots 3 and 4. A three-way cracked instruction takes up slots 0, 1 and 2. It also ends the group. In simultaneous multithreading mode, two-way cracked instructions takes up slots 0 and 1 for group 0, and for group 1, slots 3 and 4. Likewise, three-way cracked instructions fills out slots 0, 1 and 2, or slots 3, 4 and 5. Some complex instructions require more than three internal operations, these are handled by the microcode engine, which is shared between the two dispatch groups. It is possible for one half to stall the other if they both require microcode at the same time.

After dispatch, the instructions are placed into issue queues, associated with the different functional units. There are three different issue queues, one queue for branch instructions, one for condition register instructions and one queue for all the other instructions. The third queue, the *unified issue queue* is divided into two halves, UQ0 and UQ1. A steering policy determines in which of the queue halves an instruction is placed [13]. In single thread mode, instructions are steered to alternating queue halves. Instructions in slots 0, 2, 4 opposite of instructions in slots 1, 3, 5. For example, if the instruction in slot 0 is steered to UQ0, then the instruction in slot 1 will be steered into UQ1, and the instruction in slot 2 to UQ0. Depending on how many instructions are assigned to a queue half in a dispatch cycle, the one that had fewer instructions assigned to it will receive the first instruction of the next group. In simultaneous multithreading mode, instructions are steered depending on thread set. Instructions in slots 0, 1 and 2 are assigned to UQ0 and slots 3, 4 and 5 are assigned to UQ1. Each queue half is associated with its own set of functional units, and in each cycle both halves select four instructions to be issued, one for each execution pipeline.

Chapter 3
Approach

The topic of this work is comparison between different list scheduling heuristics. The compiler we are using is Clang/LLVM and the target which we are compiling for is POWER8. Initially I would focus on each of them, one at a time, starting with list scheduling. After acquiring some basic knowledge of instruction scheduling, list scheduling and heuristics, I moved on to the LLVM back-end. The LLVM code base is in general very well documented, however, due to its massive size it was not always easy to navigate. For example, while adding new heuristics can be done simply by implementing a new scheduling strategy, getting the back-end to register said strategy requires editing around seven source files. As a side note with regards to working with the LLVM back-end, unlike the middle section for which new optimizations can be loaded and tested via the *opt tool*, in order to see any changes it is necessary to recompile the involved source files. Depending on your computer this might be more or less time consuming. On my personal laptop it could sometimes take upwards twenty to thirty minutes to recompile using *Make*. Using *Ninja* on POWER8, after some tweaking of *cmake* settings, I was able to get it down to five seconds for recompiling and a little under twenty minutes for a clean install. Once I felt comfortable enough with LLVM I shifted my focus towards POWER8. I found that this article [23] was very useful when trying to make sense of this intricate topic. For more specific details regarding POWER8, the user manual [13] held the answers.

The initial phase, consisting of literature study and understanding LLVM, resulted in chapter 4. The intention of this chapter is to describe how list scheduling is used to solve the instruction scheduling problem of finding a schedule that increases performance, measured in execution time. In order to understand the problem, why it is difficult, and why it is interesting, it was placed into a historical context. After introducing the algorithm, focus was shifted towards various heuristics.

For the comparison of heuristics I decided to start with the heuristics presented in [26] and compare them against the heuristics already implemented in LLVM. As I was implementing the heuristics I tested and compared them on small loops and toy programs using the Linux `time` command. For most cases I did not see any difference, which was most likely due to the code examples I was testing on. However, two heuristics did pique my interest as they would produce slightly better times: *latency weighted critical path* and *alternating instructions*. In the first case I was testing on a matrix multiplication, nested inside a loop with a high iteration count. In the second case there was 90 randomly ordered arithmetic operations inside a loop.

After implementing the traditional heuristics I turned towards implementing a heuristic of my own. I named it the dispatch heuristic as it was based on group formation and instruction dispatch, specific to the POWER8 processor. After this point I started testing on a subset of the SPEC CPU 2017 benchmarks. The experimental setup and evaluation is described in chapter 6.

Chapter 4
List Scheduling

This chapter describes the list scheduling algorithm, different heuristics, related work and how the algorithm is implemented in LLVM. The intention of this chapter is to understand the algorithm and its limitations in order to develop useful heuristics.

4.1 The List Scheduling Algorithm

The basic list scheduling algorithm is given by listing 4.1, it is a local scheduling algorithm that works within one basic block at a time [28, 7].

Listing 4.1: The basic list scheduling algorithm.

```
for each Basic Block:
    build_dag
    collect_candidates

    while candidates not empty:
        select cand from candidates using heuristics
        for each succ in cand.successors:
            decrement incoming arcs by 1
            if arcs == 0:
                add succ to candidates

        remove cand from candidates
        schedule cand
```

The first step upon visiting a basic block is to create a directed acyclic graph, or dag. Each instruction in a basic block is represented by a node in the dag and arcs between nodes represent dependencies between instructions. For any two nodes u and v such that there is a path from u to v in the dependency dag, it follows that u must execute before v to preserve correctness of the procedure. For example in figure 4.1, there is a path from node `a` to node `d`, and node `a` must be scheduled before node `d`. An arc between two nodes is commonly associated with a weight in form of instruction latency and each node is assigned a priority based on some heuristic. The goal is to find a topological ordering that increase performance. Any topological sort of the dependency dag is of course valid [7], and the outcome of the schedule relies on the chosen heuristic. The dag is constructed in either a forward or backward pass of the basic block. To determine whether two nodes shall have an arc between them can effectively be done either in a compare node by node fashion, with a worst case of $O(n^2)$, or through table building based on definitions and uses of a resource, such as registers or memory. Table building keeps track of the instruction that last modified a resource along with a set of current uses [26]. The order in which instructions of a basic block are scanned doesn't matter and to compute certain heuristics it may be necessary to perform a second pass in the opposite direction.

Once the dependency dag has been constructed list scheduling processes the dag in either top-down or bottom-up topological order [4]. In a top-down approach each node with no incoming arcs are collected in a set of ***candidates*** which constitute a set of instructions ready to be scheduled next. Once an instruction has been scheduled, it is removed from the set of candidates, and any successor node that may now be ready is added to the set. This process continues until the set of candidates is empty. Likewise, in a bottom-up scheduler the nodes that are candidates are those whose successors have all been scheduled.

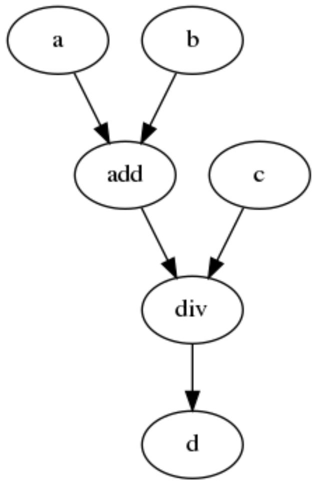


Figure 4.1: The dag of the expression: $d = (a + b)/c$.

Listing 4.2: A possible schedule of the dag in figure 4.1.

```
lwz 3, -12(1)
lwz 4, -16(1)
add 3, 3, 4
lwz 4, -20(1)
divw 3, 3, 4
stw 3, -24(1)
```

Listing 4.3: Another possible schedule of the dag in figure 4.1.

```
1 lwz 3, -12(1)
2 lwz 4, -16(1)
3 lwz 5, -20(1)
4 add 3, 3, 4
5 divw 3, 3, 5
6 stw 3, -24(1)
```

As an example, the dag of the expression $d = (a + b)/c$ is given by figure 4.1. The two listings, 4.2 and 4.3, show assembly code for two possible schedules of the dependency dag. We can see that lines 3 and 4 are switched between the two schedules. On POWER8 a fixed-point load instruction has a latency of three cycles between load and use, which means that the second schedule will execute slightly faster. On the other hand, we see that in the first schedule we are able to reuse register 4. At line 2 we use register 4 to load b, and on line 4 we use the register again to load c. In the second schedule we need three registers to load all the variables before the arithmetic operations. This is a simple example of how instruction scheduling increase the register pressure.

4.2 Brief on the History of List Scheduling

Before instruction scheduling, list scheduling was used in the microcode compaction optimization problem [6, 15]. A *microprogram* is a sequence of *microinstructions*. A *microinstruction* is a set of microoperations. The objective of the microcode compaction problem is to find a semantically equivalent microprogram with a minimized execution time. This is achieved by making better use of the available pipeline slots in each cycle. There are many similarities between microcode compaction and instruction scheduling and although the two optimization exists independently, the microcode compaction problem was in many ways the forefather of modern instruction scheduling.

The instruction scheduling problem as we think of it nowadays, grew out of the necessity to prevent the execution of a machine instruction before the instruction's operands are available. This hazard can also be handled by hardware through the use of pipeline interlocks. A pipeline interlock is a, relatively, time consuming mechanism to detect and stall an instruction from executing before the instruction's operands are available. Instruction scheduling offered an alternative to this complex hardware mechanism, by rearranging the instructions in such a way as to avoid the hazard, or to reduce the number of interlocks.

In the early days of instruction scheduling, there were two approaches to solving the problem. Either instruction scheduling was performed during code generation, or it could be performed afterwards, on generated code in a *postpass*. In case of scheduling during code generation, another problem arose: Whether to schedule before, or after, register allocation. This problem is known as the *phase ordering* problem [4]. When scheduling prior to register assignment the number of interlocks introduced by false dependencies can be minimized. A false data dependence is a dependence between two instructions that only exists because they happened to be assigned the same register, or *name*, by the register allocator. For this reason anti and output dependencies are sometimes referred to as *name dependencies* [10]. However, a scheduler that schedules too aggressively may introduce spill code that could have been avoided. On the other hand, a register allocation without any spill code, may put unnecessary constraints on the scheduler due to name dependencies.

Hennessy and Gross [9] described an algorithm for a postpass code reorganizing optimization and discussed some of the interdependence between this approach and preceding optimization phases. For example the scheme by which the register allocator reuses registers. Should the register allocator reuse registers as they become available it might prevent the postpass to reorganize some of the instructions. In that case, a round-robin method for selecting available registers might be more useful.

While rearranging code in a postpass was more general and independent of the generated code, performing instruction scheduling during code generation tended to produce better quality of code [9]. Code generation might require several passes which could potentially be very time consuming, and so the choice of approach would reflect the need of the system. The IBM PL.8 compiler [1] is an example of an early optimizing compiler that implemented instruction scheduling as part of the code generation phase. It performed scheduling before register allocation, as well as afterwards to accommodate for any spill code introduced by the allocator.

4.3 The Fundamental Algorithm for Instruction Scheduling

In the algorithm described by Hennessy and Gross [9] conflicting instructions, that is, instructions with some form of dependency between them, were able to block each other from being scheduled. To avoid these deadlocks the algorithm used lookahead. The algorithm had a worst-case runtime of $O(n^4)$ for a basic block with n number of instructions. The code reorganization algorithm presented by Gibbons and Muchnick in [7] improved on this time complexity by using a dag with more restrictions, which removed the need for lookahead, thereby giving the algorithm a $O(n^2)$ worst-case runtime. Their algorithm assumed a machine which employed hardware pipeline interlocks, which meant that the algorithm did not need to detect every pipeline hazard, unlike Hennessy and Gross whose algorithm targeted machines without this mechanism. The difference in time complexity allowed list scheduling to be efficient even on basic blocks with a larger number of instructions.

Warren [28] described an algorithm for scheduling instructions on the IBM RISC System/6000. The algorithm was a redesign of a previous scheduling algorithm used in the PL.8 Compiler. Similar to Gibbons and Muchnick, Warren used a dag to represent the scheduling constraints between instructions. An arc between two nodes in the *dependency graph* indicates that the predecessor must execute before the successor and with each arc is associated a delay. Scheduling is done by selecting from instructions that are ready, that is, all predecessors have been scheduled and enough time has elapsed to account for the required delay. There is no lookahead and in order to select among multiple ready instructions the algorithm relies on heuristics. The fundamental algorithm for instruction scheduling is list scheduling and is due to these two papers [7, 28].

4.4 Heuristics for Scheduling a Basic Block

Instruction scheduling of a basic block is essentially done in two parts: Build a dependency dag of the instructions in the basic block, and schedule the dag in some topological ordering. As finding an optimal schedule is an NP-complete problem, the scheduling relies on heuristics. In this thesis we decide which instruction to schedule next based on a pairwise comparison of the instructions in the set of candidates. For example, Gibbons and Muchnick [7] used the following three heuristics:

Interlock with an immediate successor, instructions which may cause interlocks should be scheduled early on where there are more available instructions which can be scheduled after them.

The number of immediate successors, instructions with a larger number of immediate successors will cause more instructions to become available for scheduling.

The length of the longest path from the instruction to any leaf node. Selecting an instruction based on the length of the path to a leaf node helps to balance the progress of different paths towards different leaves.

The order of the heuristics is important as it determines the priority of each heuristic. We say that the heuristics are hierarchical. Given two instructions, we compare them against each other, one heuristic at a time. If the heuristic cannot decide which instruction to schedule, we move on to the next heuristic and so on. As another example, the heuristics applied by Warren [28] selects instructions based on the following. Among the instructions in the set of candidates, select those whose *earliest time* has arrived, that is, enough time has elapsed between the instruction and its predecessors, or if there are none then those with the smallest earliest time. The algorithm will favor instructions of opposite type to the one which was previously scheduled. Of these instructions it will select the ones with maximum delay from the instruction down to any leaf node. Next it selects the instructions which has minimum *liveness weight*, followed by uncovering and finally, should there still be multiple instructions to choose from, it selects which ever came first in the original program order.

The liveness weight, described in [28], is a simple priority based scheme to reduce the number of register spills. It assigns a weight to an instruction based on the following. Move-register gets assigned the lowest weight as keeping this instruction close to the instruction that defines the source of the move will benefit the register allocator with coalescing. Next are instructions without targets as they may free up register, though they never increase the number of live registers. Most instructions sit in the middle of the weights, then come load instructions and finally instructions with no source registers, which get the highest weight since they increase the number of live registers without freeing any.

Smotherman et al. [26] conducted an extensive survey which included 26 heuristics applied in local scheduling. These heuristics were divided into categories, reflecting their intended purpose. These categories will be useful later on when we compare heuristics. The following categories were used in [26]:

Stall Behavior: Heuristics in this category prioritize instructions to avoid stall cycles. For example *earliest execution time* [28] which ensures that enough time has elapsed between scheduling two dependent instructions. *Interlock with an immediate successor* [7] indicates if an immediate successor of an instruction will be able to execute within the next cycle or not. *Execution time* [21] is the operation latency of a node. Interlock with immediate successor and execution time are two heuristics similar to each other, both of them attempt to choose instructions with long delays first.

Instruction Class: Similar to Stall behavior, this category of heuristics tries to avoid stall cycles by better utilizing the functional units in a superscalar processor. *Alternate type* [28] is a simple heuristic that prioritizes instructions that have a different type than the last scheduled instruction, allowing more instructions to be issued in each cycle. In this thesis, this category is included in *stall behavior*.

Critical Path: The longest path through a dag is called the critical path and it constitutes the shortest amount of time needed to execute the program [5]. Heuristics in this category tries to balance progress through various paths in the dag. The *maximum path length to a leaf* [7] counts the number of arcs between a candidate node and the leaf node furthest away from it. *Maximum delay to a leaf* [28] is the sum of the delays of the arcs from a candidate node to the leaf furthest away from it. *Maximum path length from root* and *maximum delay from root* are analogous to previous two heuristics except that they move upwards in the dag [26].

The *earliest start time* [26] of a node is the maximum of *earliest start time of p* plus *latency of p* over all immediate predecessors *p*. The *latest start time* [26] is the minimum of *latest start time of s* over all immediate successors *s* minus the latency of the node. The *slack* [26] of a node is defined as the difference between latest and earliest start time. It is an indication as to how much the execution of a node can be delayed, without delaying the final execution time of the program. A node with a slack of zero is on the critical path.

Uncovering: Uncovering heuristics attempt to expose more scheduling opportunities by increasing the number of nodes in the set of candidates. The *number of immediate successors* [7] is a naive estimate of how many nodes would be uncovered if an instruction is scheduled. *Sum of delays to immediate successors* [26] includes the arc delay as a weight. *Number of nodes with a single immediate predecessor* [26] gives a slightly more accurate estimate to the number of nodes that would be uncovered, this heuristic may also include delays as weights. The *number of uncovered immediate successors* [28] is a combination of the two heuristics. It gives a better estimate by only considering arcs with a delay of one.

Structural: The purpose of a structural heuristic is to balance progress through a dag. While critical path heuristics attempt to balance progress along longer paths, structural heuristics aim to balance progress across the dag. *Number of immediate predecessors* is an example of a structural heuristic which gives an indication of how many instructions must complete before a candidate. This was used as a tie-breaking heuristic by Shieh and Papachristou [21] for determining the priority of a node. It was used as an inverse heuristic, meaning that a node with a larger number of immediate predecessor would be assigned a lower priority. *Number of successors* [26] is considered a structural heuristic though it expands on the idea of uncovering. This heuristic may also include delays as weights.

Register Usage: Heuristics in this category aim to reduce the register pressure by decreasing the number of registers that are live simultaneously. The reason why register pressure is so important is because unnecessary spill code needlessly degrades performance. The interdependence between instruction scheduling and register allocation makes it a difficult task to balance. Recall that spill code inserts store and load operations. Sometimes spill code cannot be avoided. In those cases, heuristics attempts to prevent the scheduler from increasing the register usage. *Number of registers born* and *number of registers killed* can be used to indicate whether scheduling an instruction should be postponed or not due to register pressure. For example *liveness*, as defined by Warren, includes both heuristics for registers born and killed in order to determine the weight of an instruction. According to Warren [28] they tried some more elaborate schemes but the liveness weight seemed to have been working as well as any of them.

Apart from these categories, heuristics can also be divided between static and dynamic heuristics. Not to be confused with static and dynamic scheduling, which is scheduling performed by the compiler as opposed to scheduling performed by hardware. A static heuristic can be determined by traversing the dag. For example the critical path which is calculated in a pass over the dag. A dynamic heuristic depends on the schedule ordering. An example of a dynamic heuristic is the earliest execution time which is maintained by the scheduler and updated for an instruction when its immediate predecessor gets scheduled.

Heuristics can also be divided between general and specific heuristics. A general heuristic is for example the *maximum path to a leaf node*. As this heuristic only depends on counting the number of arcs on the critical path, it does not take into consideration any target specific details. On the other hand, *maximum delay to a leaf node* takes the arc delays into consideration and these delays are specific to the target, based on some assumptions such as a cache hit for example, making this a specific heuristic. Any sort of hardware related heuristic falls into the category of specific heuristics, for example, the number of registers or functional units of the processor.

4.5 Related Work

In order to learn more about the behavior of list scheduling, Cooper et al. [5] performed an experimental evaluation of the algorithm. Mainly they were trying to identify possible scenarios where list scheduling underperforms. In their experimental setup the compiler did not perform register allocation. The motivation for this was that, by eliminating the interaction between instruction scheduling and register allocation, it isolates the impact of scheduling. They used the *latency weighted depth* of a node to determine its priority. A tie between two nodes was broken arbitrarily.

They presented two alternatives to their list scheduling algorithm: Random tie breaking and backward list scheduling. Instead of breaking ties arbitrarily, the idea of random tie breaking is to run the list scheduler several times, each time breaking any ties randomly and potentially generating more and better solutions. The backward list scheduling works by reversing the directions of the arcs in the dependency graph, operations are scheduled based on *finish times*. Backward list scheduling tend to cluster instructions towards the end of a basic block rather than at the beginning. They used these two alternatives to develop a new technique which they called RBF, or randomized backward and forward list scheduling.

RBF schedules a basic block *M* times forward, and *M* times backward. Ties between instruction with equal priority are broken randomly rather than arbitrarily. This gives *2M* schedules, of which the best one is kept. Three metrics were used in the evaluation of list scheduling. Minimum schedule length, which was used to identify optimal schedules. Available parallelism within a basic block, defined as *the sum of latencies of all instructions in the dag* divided by *the length of the critical path* and the number of list schedules, an attempt to quantify the number of possible list schedules.

They compared list scheduling against RBF and another scheduling technique called iterative repair scheduling. In general, iterative repair will schedule a basic block so that each instruction is scheduled as early as possible, without any consideration towards resource constraints. Then, the algorithm will *repair* the schedule by identifying resource conflicts, and reschedule one of the conflicting instructions. This process is continued until no resource conflicts exist. Similar to RBF this algorithm is run a number of times and the best schedule is selected.

While the experiment was based on some generalizations, such as only using one heuristic to determine priority, and breaking ties arbitrarily or at random, it did lead to some interesting results, in particular those on random graphs. List scheduling performs better on certain levels of available parallelism. The observation that was made indicated that list scheduling would perform well on low, as well as, high levels of parallelism, with somewhat worse performance on moderate levels of parallelism. When the list scheduler has only a few choices to make, the probability of making a less than optimal tie-breaking choice is low. To explain the performance on higher levels of parallelism, they suggested that most tie-breaking decisions led to schedules that, while different, were of the same length.

Even in cases where the available parallelism is low, if the scheduler has to break many ties, the probability of finding a non-optimal solution increase. In a practical implementation of list scheduling there will be a set of heuristics for breaking ties, rather than breaking them randomly, this experiment shows the importance of good heuristics for tie-breaking as well as it gives an idea as to why the hand-crafting approach is so time consuming.

The size of a basic block can be a limiting factor for list scheduling. The fact that list scheduling often finds optimal schedules for small basic blocks does not mean that there cannot exist a better schedule of the whole procedure. Because list scheduling operates on one basic block at a time, it does not consider scheduling across basic block boundaries. Modern processors are superscalar, and in order to keep all functional units busy, it is necessary to achieve a certain amount of ILP. For large basic blocks this is not a problem for list scheduling, but in control intensive programs, that is, programs with many branches, list scheduling might struggle to find enough available instructions. Global instruction scheduling techniques, for example scheduling of superblocks [3] overcome this by allowing scheduling across boundaries. A super block is a trace of basic blocks, such that entering the trace can only happen at the top. A super block can have more than one exiting branch. Exit branches except the last one are called *side exits*. The scheduling process in [3] consists of trace selection, forming superblocks, building a dependence graph and performing list scheduling.

Another area where list scheduling shows limiting performance are loops, since it cannot schedule instructions from multiple loop iterations. In loop bodies with few instructions, list scheduling will not be able to find enough instructions to keep the processor busy. This becomes a bigger concern when iteration count is high. To overcome this, modulo scheduling, a form of software pipelining, can be used [24]. In list scheduling, each loop iteration waits on the previous to finish before it starts. The idea of modulo scheduling is to overlap loop iterations in order to increase throughput. In [11] Huff presents a bidirectional approach to scheduling in the context of lifetime sensitive modulo scheduling. He notes that traditional list scheduling has a tendency to unnecessarily stretch lifetime intervals, either by scheduling loads too early, or stores too late.

While the list scheduling algorithm itself is fairly simple, it offers a lot of variation through the use of heuristics. Many heuristics have been developed over the years and it continues to be an interesting research topic [26, 20]. Of the six categories described in section 4.4, the heuristics in the critical path category have traditionally been the most popular [14, 26]. Their aim is to identify instructions which should be scheduled as early as possible. Another popular category is register usage. Register pressure heuristics are only interesting for scheduling done pre-register allocation and their aim is to shorten live ranges in order to decrease the number of variables live simultaneously. Register pressure heuristics has a bigger impact if the basic block is large [8].

Modern processors employ register renaming to support out-of-order execution. Register renaming is the act of assigning logical registers to physical registers. Since the set of physical registers is larger than the set of logical, register renaming can eliminate anti and output dependencies. Silvera et al. [22] described an algorithm that would reorder instructions to reduce register pressure without negatively affecting ILP, i.e by letting the out-of-order mechanism break the name dependencies. The algorithm takes as input, the output of a traditional list scheduler. It was one of the first papers focusing on the relationship between instruction scheduling and register allocation on targets with out-of-order execution. The benefit of reducing register pressure especially for targets with out-of-order execution was also noted by Valluri and Govindarajan in [27] in which the phase ordering problem was studied and different techniques evaluated.

Heuristics are often handcrafted, and developing good heuristics is a time consuming process which requires both experience and great knowledge of the target [18]. An alternative approach, in order to speed up the process, is to make use of machine learning techniques. For example, Malik et al. [20] used a decision tree learning algorithm, a form of supervised learning, which is used for classification of data, to automatically learn good heuristics. A list scheduling algorithm use heuristics to choose between two instructions at a time. In order to express this as a classification problem it is sufficient to be able to return `true` if instruction *i* should be scheduled before instruction *j*, or `false` otherwise.

According to [20] they started out with 60 features. In machine learning, a feature is some form of measurable characteristic. In our context a feature is a heuristic, and a set of features is what we think of as a list of priorities. From this point I will use the word heuristic instead of feature. These 60 heuristics included all of the ones surveyed in [26] except for those regarding register usages and some heuristics subsumed by others. Some novel heuristics were constructed by means of applying functions to existing ones, functions such as the maximum of two or the average of multiple. Unfortunately there is no listing of all of these heuristics. As [26] only accounts for less than 26 of the heuristics, it would have been interesting to note what else was tried.

Of the 60 heuristics 17 remained after filtering, a preprocessing step by which heuristics that are deemed less useful are deleted. Filtering can improve the efficiency of the learning process, as well as improve the quality of heuristics that are learnt. The 17 remaining heuristics were then ranked according to their usefulness in classifying data. Three ranking methods were used, and the value of a heuristic was determined by the average. Of the 17 heuristics, all but the last four were static heuristics. The *critical path distance to a leaf node* was ranked as number five. According to [20] they were surprised by this, suggesting that because of its popularity it was expected to rank higher. However, I think it is worth mentioning that the highest ranked heuristic was the maximum of the novel *resource-based distance to leaf node* and *critical path distance to a leaf node*. The *resource-based distance to leaf node* incorporate a balance between critical path distances to successors, instruction classes and critical paths of successors to leaves, and it was ranked as number two. Furthermore, *path length to leaf node*, which is also a *critical path heuristic*, was ranked as number three. The heuristic with rank four was *number of successors of an instruction*, a structural heuristic, and the heuristic ranked six was *slack*. In other words, five of the six highest ranked heuristics were some form of *critical path heuristics*. Each category of heuristics had at least one representing heuristic among the 17 most useful heuristics, apart from register usage obviously.

It is interesting, and perhaps a bit more surprising, to see that the heuristic with rank seven was the *order of the instruction in the original instruction stream* as this one is usually the last tie-breaker (for obvious reasons) in a series of heuristics, should all the previous attempts to prioritize one instruction over the other have failed. This peculiarity was also noted by Beaty et al. [2], who commented that critical path heuristics are widely held to be the most important heuristic while lexical order is consider relatively unimportant.

4.6 List Scheduling in LLVM

The default instruction scheduler in LLVM, targeting POWER8, is called the *Generic Scheduler*. It has access to target specific details such as information on instructions and processor registers. Two important concepts of the scheduler are *scheduling units* and *scheduling dependencies*. Scheduling units are basic blocks, and scheduling dependencies are arcs between basic blocks. The different kind of dependencies are: Data, Anti, Output and Order. The first three are familiar to us since before, the Order dependence constitute the following dependencies: Barrier, MayAliasMem, MustAliasMem, Artificial, Weak and Cluster. Dependencies are either strong or weak. A strong dependence must be respected by the scheduler, with the exception of artificial dependencies which may be removed, but only if they are redundant with regards to another strong dependence. A weak dependence may be violated, but only if it can be proved by the scheduling strategy, or scheduling algorithm, that it is correct to do so. Each dependence is associated with a latency, which is an estimate of the minimum number of cycles that must elapse between two instructions, given that one is the immediate predecessor of the other. The depth of a node is the maximum, latency weighted, path from that node up to any root node. The height of a node is the maximum, latency weighted, path from that node down to any leaf node.

The scheduler takes as input a machine function, that is, a function which has been translated down to the target-dependent intermediate language. Each machine basic block is visited in function order. Each block is associated with a scheduling region, essentially a struct which holds the first and last instruction in the region. After each region has been determined, the scheduler will begin scheduling. The first step is to build the dag and set up trackers for register pressure. In LLVM register pressure is computed for one scheduling region at a time. If live intervals are available they are used for recording the boundary of the tracked region. A tracker only operates on one machine basic block at a time, in order to track pressure across larger regions, the register pressure is stored at block boundaries, and then adjusted to account for live-in and live-out register sets. Three types of pressures are recorded: Excess, which is the pressure beyond the targets limit, critical maximum and current maximum, which records the largest increase that exceeds the critical or current limit on some pressure set. Register pressure sets are used to keep track of register usage. Once the dag has been built, the scheduling strategy initializes ready queues and hazard recognizers for the top and bottom halves of a region.

Each region is divided into two halves, top and bottom. The default scheduler schedules instructions bottom-up, but via a scheduling policy, a scheduling strategy can force either top-down, bottom-up or bidirectional scheduling. Each half is associated with a ready queue which is updated when an instruction is scheduled. If a node from the top half is scheduled, each of its immediate successors will have the number of incoming branches decremented. Once all predecessors have been scheduled, the node is released. This means that the node becomes visible to the heuristic function and can now itself be scheduled. Likewise for the bottom half, when a node is scheduled, each immediate predecessor will have its number of outgoing branches decremented. Initially the root and leaf nodes are released for top and bottom half respectively.

For PowerPC back-ends the bidirectional approach is used as this can provide a more balanced schedule, according to a comment in the LLVM source code. Assuming there are nodes available from both halves' ready queues, the best node in each half is selected. Then, the best between the two nodes is selected and scheduled. In a top-down or bottom-up approach it is easy to see that the schedule is correct, to see that it is correct for bidirectional scheduling as well, let us consider a node from the top half, for example a root node. When the node is scheduled, the scheduler is aware that it came from the top half queue, and any successor that now became ready, is put into the top half *Pending* queue, later to be moved into the top half *Available* queue, where it can be selected by the heuristic function. The scheduler maintains a reference to the *current top* and *current bottom*. An instruction from the top queue, can only be moved to become the current top. Likewise, instructions coming from the bottom queue can only be moved to become the current bottom. The name current bottom is a bit of a misnomer, in fact the current bottom is the top of the bottom half. Ultimately these two variables are moving towards each other, shrinking the unscheduled zone. In other words, a node enters a queue once its dependent nodes have been scheduled. Depending on which queue it enters, it gets scheduled either to the top or bottom half of the scheduling region. In the final schedule, because every instruction in the top half comes before instructions in the bottom half, dependencies between instructions from different halves, will be respected.

The heuristic function takes as input two candidates of which one is currently the best candidate, and a reference to the current zone being scheduled. The zone is either top or bottom, or in case the two candidates are from different zones, such as can happen when comparing two candidates in a bidirectional schedule, the zone is `null`. Heuristics in LLVM are hierarchical, essentially working like tie-breakers. The currently best candidate is associated with a *reason* for picking that candidate. Reasons are ordered by their priority. The generic scheduler prioritize candidates which reduce register pressure. For candidates that come from different boundaries, only a subset of the heuristics are used for comparison. For example, heuristics regarding resource usage are only compared for candidates of the same zone. In case all heuristics turn out equal, candidates of the same zone fall through to node order, whereas, if the candidates are from different zones, the scheduler will favor the bottom zone.

Chapter 5
Implementation

This chapter will detail the heuristics that I chose to implement. Of the heuristics surveyed in [26] I implemented the following fifteen, these included at least one heuristic from each category.

Stall behavior: *Execution time*, this heuristic was implemented using the, in LLVM available, node latency. *Alternate types*, this heuristic was implemented by simply looking at the operation code of the previously scheduled instruction and favor a different one. In the dispatch heuristic, described below, alternate types was refined to consider the required functional unit of the previously scheduled instruction.

Critical path: The heuristics in this category was computed by traversing the dag in a depth first manner. For path length the number of arcs was counted, and for delay the latency of the arc was counted instead.

- *Maximum path length to a leaf.*
- *Maximum path length to a root.*
- *Maximum path delay to a leaf.*
- *Maximum path delay to a root.*
- *Earliest start time.*
- *Latest start time.*
- *Slack.* (Computed as latest start time - earliest start time.)

Uncovering: The heuristics in this category was implemented using the, in LLVM available, set of immediate successors.

- *Number of immediate successors*, the size of the set of immediate successors.
- *Sum of the delay to immediate successors*, the sum of latencies of the arcs leading to the immediate successors.

- *Number of uncovered immediate successors*, counts the immediate successors that only have one incoming arc, with a latency of one.

Structural: Both of the heuristics in this category were implemented by traversing the dag in a depth first manner and counting the arcs and their latencies.

- *Number of descendants.*
- *Sum of execution time of descendants.*

Register usage: Warren's liveness weight was implemented from the description in [28]. Instructions are assigned a weight depending on what type of instruction it is. For example, instructions with no target register is assigned a lower weight than instructions with no source registers. The former have *uses*, but no *definitions*, and so there is a chance that they might free a register, but they will never increase register pressure. Instructions without source registers increase the register pressure, without freeing any registers.

I also implemented the *resource-based heuristic* due to Malik et al. [20]. The resource-based heuristic is defined as: $rb(i) = max\{r1(i,t) + r2(i,t) + r3(i,t)\}$, over t [20]. The values $r1$, $r2$, and $r3$ are calculated for a node i as:

- $desc(i,t)$: Descendants of i with type t in the dag.
- $cp(i,j)$: Critical path distance between i and j.
- $r1(i,t)$: $min\{cp(i,j) \mid j \in desc(i,t)\}$
- $r2(i,t)$: $\mid desc(i,t) \mid /k(t)$, where $k(t)$ is the number of execution units that can execute instructions of type t.
- $r3(i,t)$: $min\{cp(j,l) \mid j \in desc(i,t)\}$, where l is a leaf node.

I was interested in the resource-based heuristic as it was the only novel heuristic that remained after *feature filtering* [20]. I implemented two versions of it, one according to the definition given above and one which also considered if a candidate would increase register pressure or not. As mentioned in section 4.6 LLVM keeps track of register pressure, and pressure changes are available for each candidate node.

I also wanted to try to implement a heuristic of my own. Two things intrigued me about the alternating heuristic, firstly, despite it being a rather trivial heuristic it proved adequate when compared to some more elaborate ones. Secondly, it is the simplest form of manipulating multiple functional units. Another interesting aspect is the relation between static and dynamic scheduling. If a heuristic takes the dynamic scheduling into consideration, it may produce a schedule which hardware can take advantage of and improve further.

The *dispatch heuristic* is an attempt to order instructions in a way that will benefit group formation. Since instructions are dispatched on a group basis, the idea is to schedule instructions that can go into the same group, together or in close proximity of each other. One of the reasons why instruction dispatch and group formation is of particular interest to the static scheduler is because this stage represents the last in-order part of the pipeline until completion. After this point it becomes very difficult to tell beforehand in which order the instructions will be executed.

Listing 5.1: Algorithm of the dispatch heuristic for the top half.

```
If at the start of a new group then
  prioritize 2—way cracked
  over 3—way cracked instructions

Try to get fuseable instructions into the same group

If both instructions are marked as Last then
  let another heuristic decide

If we already started a new group then
  prioritize an instruction that isn't cracked
  or marked as First over one that is

  If one is Last and the other one isn't then
    If it is dispatch slot 1 or 4 then
      we should prioritize the instruction
      that isn't marked as Last

  If neither instructions are marked as Last then
    we should try to alternate between types

  We should try to schedule instructions
  that has microcode in the same group
```

Listing 5.2: Algorithm of the dispatch heuristic for the bottom half.

```
If both instructions are marked as Last then
  let another heuristic decide

If only one is marked as Last then
  go with that instruction

If either is 3—way cracked then
  let another heuristic decide

Try to get fuseable instructions into the same group

If we have already filled out slot 5 or 2 then
  we want to either complete the group
  with a 2—way cracked instruction
  or try to fill out slot 4, or 1

  We should try to alternate between instruction types

  We should try to schedule instructions
  that has microcode in the same group
```

Instructions are dispatched on a group basis, meaning that, instructions within the same group enter the out-of-order part of the processor in parallel. Ideally we would like the groups to balance the instructions between the execution units. The dispatch heuristic is based on the rules of group formation. It makes a decision about an instruction, depending on the previously scheduled instruction. Since the PowerPC back-end in LLVM schedules instructions bidirectionally, the *pickNode* function was extended to keep track of which instruction was last scheduled and to which half, bottom or top, it was scheduled. Pseudocode of the dispatch heuristic can be found in appendix B, listing 5.1 and 5.2 give the general algorithm. Essentially the top half algorithm schedules instructions in a forward approach, starting by filling out slot 0 of group 0. In contrast the bottom half algorithm starts by filling out slot 5 in group 1. However, there are some subtle differences between them. For example, when the bottom half selects an instruction that must come first, it may end the group prematurely. Likewise, the top half algorithm may end the group prematurely when it selects an instruction that must come last.

When starting a new group, the top half algorithm will prioritize instructions that must come first and instructions that are two-way cracked, in hopes that a later instruction will fill out the remaining group slot. The algorithm also tries to place instructions that can benefit from instruction fusion in the same group. If a group has already been started, it will prioritize instructions that does not have to come first. Then, if neither instruction has to come last, we consider whether the instruction will use the same functional unit as the other instructions in the group. Or whether the instruction, or any of the instructions in the other group, will require the microcode engine.

For the bottom half, at the start of a new group, the algorithm will favor instructions that must come last. Since a three-way cracked instruction takes up all the non-branch slots in a group, we let another heuristic decided determine the best candidate. If a group has already been started, we can either complete it with a two-way cracked instruction, or try to fit two instructions in the remaining slots. Again we want to alternate between instruction types, and place instructions that require microcode into the same group.

Chapter 6

Experimental Evaluation

This chapter will describe the experimental setup and then discuss the results of the two experiments.

6.1 Experimental Setup

In order to compare heuristics against one another, two sets of experiments were performed. In the first set of experiments, each heuristic was used in isolation. In the second set of experiments different combinations of heuristics were used to select between instructions. In LLVM heuristics are ordered in a hierarchical structure. A combination of heuristics in this context is therefore a set of heuristics which are given a priority relative to each other. These combinations were selected from a subset of the heuristics from the first experiment. In both sets of experiments, if the heuristics were unable to differentiate between two instructions from the same region half, the original program ordered was used. In case the candidates were from different region halves, the bottom half was favored when a draw occurred. Use of the original program order as a tie-breaker seemed to me to be the most unbiased, while favoring the bottom half is in line with LLVM's generic scheduler. In both sets of experiments the original program order was used as a baseline, and LLVM's generic scheduler was used as reference.

Experiments were carried out on a subset of the SPEC CPU 2017 benchmarks. SPEC CPU 2017 consists of 43 benchmarks and a benchmark is organized into one of four suites depending on which metric, SPECrate or SPECspeed, is used and if it is an integer or floating point benchmark. The benchmarks are written in C, C++ and Fortran. There are 23 benchmarks for the SPECrate metric. Only the benchmarks written in C or C++ were selected. Of these Clang/LLVM failed to compile one, and of the benchmark that compiled, all but one produced the correct results. This left nine integer benchmarks and five floating point benchmarks. The SPEC `runcpu` tool used to perform tests provides two types of metrics, SPECrate and SPECspeed. For the experiments conducted in this work only the SPECrate metric was used, as it is a measure of throughput and the purpose of instruction scheduling is to increase the number of instructions executed per unit of time. SPECrate is computed as the *Number of copies * (Time on the reference machine / Time on the system under test).* There are two ways which the `runcpu` tool can run benchmarks. Either each benchmark is run three times and the tool use the median time. Or each benchmark is run twice, and the tool use the time of the slower of the two runs. In this thesis the latter approach was used for the final tests. Both approaches were tried, but due to time constraints it was decided that running each benchmark twice was preferred. When running the SPECrate benchmarks the number of copies that are run in parallel can be specified. More copies means more workload, but also potential for more throughput. For testing individual heuristics, the number of copies was set to one which is the default configuration. When performing the second experiment, the number of copies were set to 10, the number of CPUs on the target machine.

Each benchmark was compiled with the `-O3` optimization level, the `-fgnu89-inline` and `-fno-strict-aliasing` compiler flags. Even though only the 500.perlbench_r benchmark required the `-fno-strict-aliasing` flag, and likewise only the 502.gcc_r benchmark required the `-fgnu89-inline` flag, each benchmark were compiled with these two flags as required by the SPEC CPU 2017 *base* metric. SPECrate can be run either as base, or peak metric. In base mode, each benchmark is required to be run with the same optimization flags.

SPEC CPU 2017 consists of various programs which perform computational intensive tasks which makes them suitable for testing the performance of CPU, memory and compilers. The benchmarks come from a variety of application areas: Programming languages and compilers, scientific programs, simulation, planning, scheduling, search, compression, graphics. Most of the benchmarks are derived from real-world applications which makes them more interesting than synthetic benchmarks, which may still be interesting in their own right.

6.2 Evaluation of Individual Heuristics

In the first experiment individual heuristics were compared with one another. Figure 6.1 shows the result of experiments on the integer benchmarks, and figure 6.2 shows the result of experiments on the floating point benchmarks. These results are produced by the SPEC `runcpu` tool. Appendix C contains the results on individual benchmarks.

Perhaps the most interesting result shown in figure 6.1 is that only two heuristics, RP excess and RP critical, had a positive difference to baseline. It was hinted at in the end of section 4.5, that program order is a relatively strong heuristic, and since there were only two other heuristics, dispatch and RP max, that showed similar results as baseline, this belief is further reinforced. For the rest of the heuristics results vary from -2 % to -6 %. If we look at the graphs over specific benchmarks, figures C1 through C9, the heuristics that yield the better results in figure 6.1 tend to stay ahead of the rest of the heuristics with only a few exception. Likewise, the heuristics that yield a worse result overall, also tend to perform worse in the specific cases, with a few exceptions. LLVM's register pressure heuristics, *RP excess*, *RP critical* and *RP max*, perform best overall, with *RP critical* being the most stable. The dispatch heuristic, though it shows a bit of variance in its performance, is not too far off from the RP heuristics. If we look at LLVM's generic scheduler, which estimated around

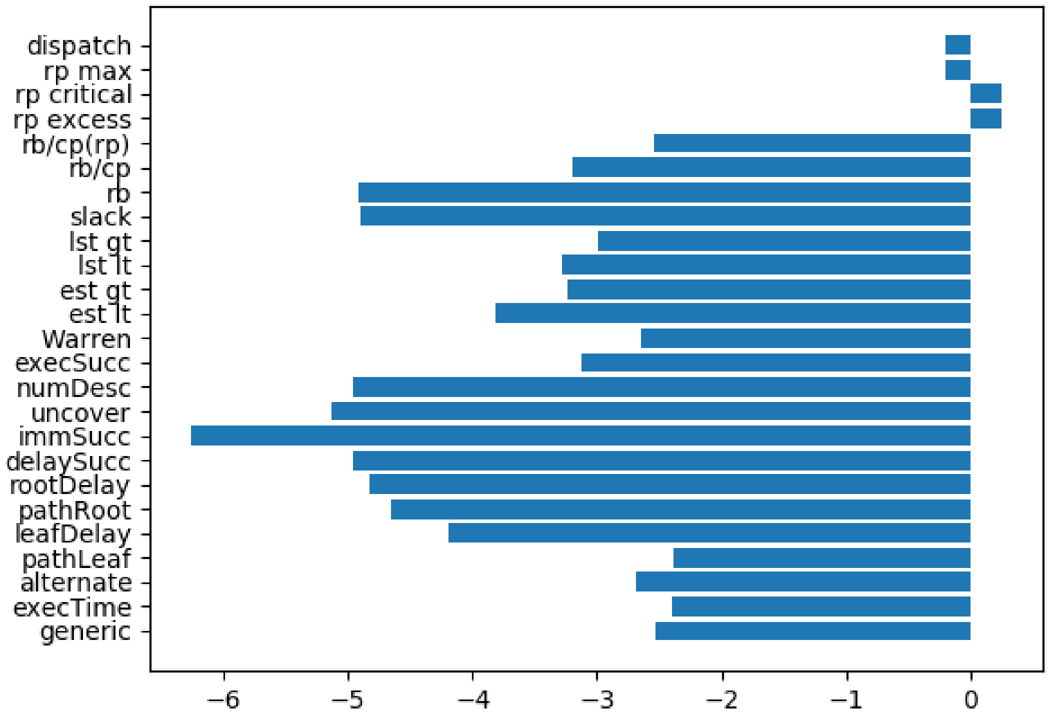


Figure 6.1: Integer benchmarks, results of the Est. SPECrate 2017 int base metric. The y-axis indicates the heuristic, and the x-axis indicates the difference, in percent, between the heuristic and baseline. Baseline is the original program order. A negative result means that the heuristic was slower than baseline.

-2.5 percent, with up to -6 percent difference on some benchmarks, we can see the benefit of using the original program order as a tie-breaker, and to break ties early on. The generic scheduler, which incorporate the RP heuristics, use a combination of 12 heuristics, of which the original program order is the last.

After register usage and dispatch, the next category in terms of performance is stall behavior. *Execution time* and *alternate type* had similar results on the integer benchmarks, with execution time being slightly better. The alternating heuristic is naive in its implementation since it only considers the most recently scheduled instruction. The dispatch heuristic expands on this idea by considering the most recently scheduled instructions within the same group. Stall behavior is followed by critical path, then structural and last uncovering.

The category that shows the most variation within itself is the category of critical path type heuristics. Apart from it being the most represented group in the experiment, this variation is likely due to the fact that the category includes heuristics that are opposites to each other, those that express the length to a root node, *rootDelay* and *pathRoot*, and those that express the length to a leaf node, *leafDelay* and *pathLeaf*. It also includes the earliest start time or *est*, and the latest start time or *lst*, which are used to compute the *slack*, but may also be used as heuristics on their own. In these cases, the values given by est and lst were compared as *greater than*, and also as *less than*.

Structural heuristics are useful to balance progression through a dag when there are multiple long paths. It is interesting to compare the two structural heuristics, *number of descendants* and *sum of execution times of successors*, with the two critical path heuristics *maximum path length to a leaf* and *maximum delay to a leaf*. Each of these heuristics aim to balance progress between various paths in a dag. In the case of critical path it appears to be more beneficial to count the number of arcs, without consideration of latency, whereas, in the case of the structural heuristics it appears to be the opposite. Compare *pathLeaf* with *numDesc*, and *leafDelay* with *execSucc*. When considering all of the descendants of an instruction, it is more beneficial to consider the sum of their delay, rather than just counting them.

Uncovering heuristics provide more options to the scheduler by expanding the list of candidates. As such they are more supportive in nature, as they themselves don't really take advantage of this increase in opportunities. This could explain their poor performance when compared individually. According to [26], the *number of descendants* is an extension of the *number of uncovered immediate successors*. Both of these heuristics aim to provide flexibility, and so rely on other heuristics to make use of this added freedom.

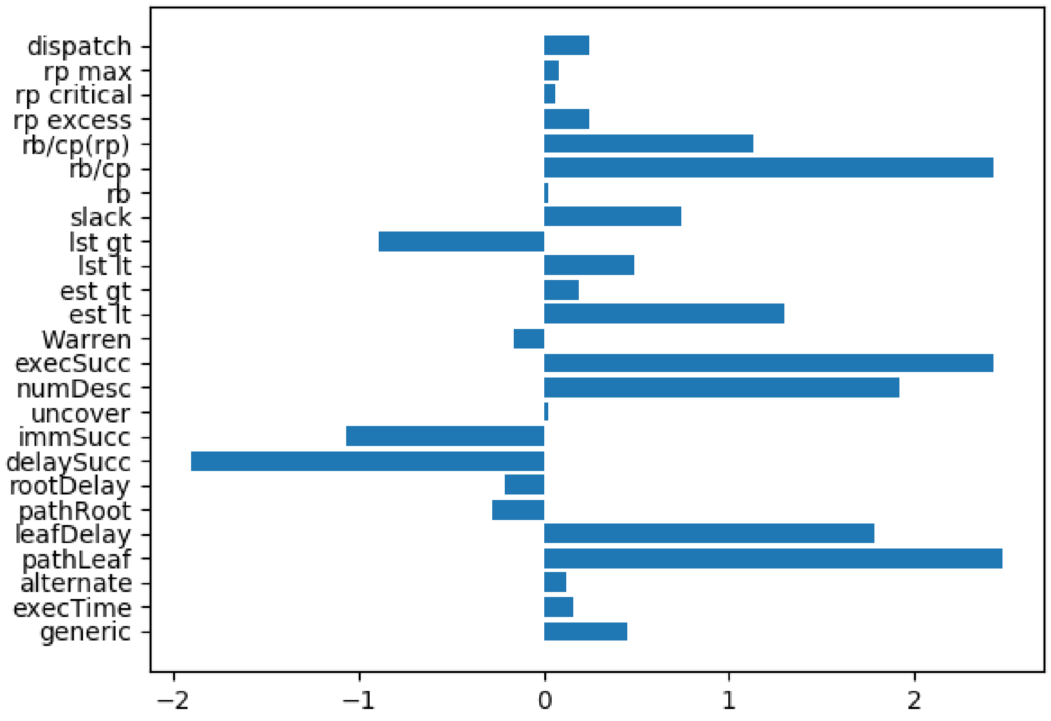


Figure 6.2: Floating point benchmarks, results of the Est. SPECrate 2017 fp base metric. The y-axis indicates the heuristic, and the x-axis indicates the difference, in percent, between the heuristic and baseline. Baseline is the original program order. A negative result means that the heuristic was slower than baseline.

The result of the floating point benchmarks show more variance around baseline. Apart from RP critical and RP excess, all of the heuristics performed better on the floating point benchmarks than they did on the integer benchmarks. The critical path category, with *pathLeaf* and *leafDelay*, and the structural category, with *numDesc* and *execSucc*, showed the best performance overall. Again we can see that between the two critical path heuristics concerned with the path to a leaf, it is better to disregard the latency of the path. And for the two structural heuristics, it is better to include the latency in the computation. That is to say, when counting arcs, the number of arcs is more interesting than the sum of the latency and when counting descendant nodes, it is worthwhile to include the latency of the node.

On both the integer and floating point benchmarks, *rb/cp*, that is, the maximum of the *resource-based* heuristic (*rb*), and the *latency weighted critical path* (*leafDelay*), performed better than either of the two heuristics on their own. When including register pressure in this heuristic, it gave a better result on the integer benchmarks, but worse on the floating point.

The *est* and *lst* are interesting because when used as heuristics outside of slack, their values were compared both as *less than*, and *greater than*. The reason for this was because it was not obvious whether it would, for example, be more beneficial to schedule an instruction with a lesser latest start time, over one with a greater latest start time. Looking at the results it is still not obvious. For both *est* and *lst*, greater than is better on integer benchmarks, and less than is better on floating point.

After critical path and structural, the next category in terms of performance on the floating point benchmarks is stall behavior, *execTime* and *alternate*, followed by register pressure and dispatch. Uncovering continues to be the worst performing category. The critical path heuristics continue to show the most variance within the same category, with path to a leaf node (*pathLeaf, leafDelay*), and path to a root node (*pathRoot, rootDelay*) being on opposite sides of baseline.

When we compare the two graphs we can see that some heuristics, for example register usage and dispatch, have similar performance in relation to the original program order. On the floating point benchmarks, most heuristics have a positive difference to baseline, which means that they add some benefit to performing static scheduling. A possible explanation as to why this is the case is because of POWER8's inherent flexibility in scheduling fixed-point instructions due to the LSU and LU being able to execute simple fixed-point instructions. In other words, the dynamic scheduler outperforms the individual heuristics to a point where, individual heuristics as the only means of differentiate between two instructions only leads to a downgrade in performance.

6.3 Combinations of Heuristics

This section will detail the combinations of heuristics that were used in the second set of experiments. A subset of the heuristics from the first experiment was used to form different combinations of heuristics. The subset consisted of the heuristics that performed best within their category, with at least one heuristic from each category. Heuristics within a combination are ordered in terms of descending priority. Combinations are named Heuristics 1 to 22. In general, combinations 4 to 20 aims to aid group formation, and reduce register pressure. If the dispatch heuristic does not make a decision on which instruction to schedule, and if register pressure is already low, the focus is on balancing the progress through the dag, while avoiding stall cycles.

Heuristics 1: *pathLeaf, execSucc, rb/cp.* This combination consist of the heuristics from the first experiments that yielded the best results on the floating point benchmarks.

Heuristics 2: *execTime, uncover, pathLeaf.* This combination was inspired by Gibbons and Muchnick [7].

Heuristics 3: *pathLeaf, dispatch, RP excess, RP critical, RP max, execTime.* This combination favors instructions with long dependences chains. It uses the, in LLVM available, register pressure heuristics and it uses them in the same order as LLVM's generic scheduler.

Heuristics 4: *RP excess, RP critical, dispatch, RP max, pathLeaf, execTime.* This combination gives less priority to the critical path category and instead focuses more on register usage.

Heuristics 5: *RP critical, execTime, execSucc, pathLeaf, dispatch.* This combination tries to balance the progress through the dag, while avoiding stall cycles.

Heuristics 6: *RP critical, pathLeaf, execTime, execSucc, dispatch.* This combination tries to balance the progress through the dag by selecting instructions with longer dependences chains first, and then selecting instructions with a larger total number of dependences.

Heuristics 7: *RP excess, RP critical, RP max, pathLeaf, execSucc, execTime.* This combination aims to reduce register pressure, and balance the progress through the dag.

Heuristics 8: *RP excess, RP critical, RP max, dispatch, pathLeaf, execSucc, execTime.* This combination aims to reduce register pressure and then to aid group formation.

Heuristics 9: *dispatch, RP excess, RP critical, RP max, pathLeaf, execTime.* This combination aims to aid group formation, and then to reduce register pressure. It favors instructions with long dependences chains, and tries to avoid stall cycles.

Heuristics 10: *dispatch, RP excess, RP critical, RP max, pathLeaf.* Same as Heuristics 9 but without considering stall cycles.

Heuristics 11: *dispatch, RP excess, RP critical, RP max, execTime.* Same as Heuristics 9 but without favoring instructions with long dependences chains.

Heuristics 12: *dispatch, RP critical, execTime, execSucc, pathLeaf.* This combination focuses more on avoiding stall cycles than balancing the progress through the dag.

Heuristics 13: *dispatch, RP excess, RP critical, execTime, execSucc, pathLeaf.* Same as Heuristics 12 but slightly more focus on register usage.

Heuristics 14: *dispatch, RP excess, RP critical pathLeaf, execTime, execSucc.* Same as Heuristics 13 but it gives less priority to avoiding stall cycles.

Heuristics 15: *dispatch, RP excess, execTime, execSucc, pathLeaf.* Same as Heuristics 12 but it uses a different RP heuristic.

Heuristics 16: *dispatch, RP max, execTime, execSucc, pathLeaf.* Same as Heuristics 12 but it uses a different RP heuristic.

Heuristics 17: *dispatch, RP critical, RP max, RP excess, pathLeaf, execTime*. Same as Heuristics 9 but it changes the order of the RP heuristics.

Heuristics 18: *dispatch, RP excess, RP critical, RP max, execTime, pathLeaf*. Same as Heuristics 9 but it gives higher priority to avoiding stall cycles.

Heuristics 19: *dispatch, pathLeaf, execSucc, execTime*. This combination aims to balance the progress through the dag. This combination, together with Heuristics 1 and 2, are the only ones that did not consider register usage.

Heuristics 20: *dispatch, RP excess, RP critical, RP max, pathLeaf, execSucc, execTime*. Same as Heuristics 19 but it considers register usage.

Heuristics 21: *dispatch, pathLeafDelay, RP excess, RP critical, RP max, uncover*. This combination was inspired by Warren [28].

Heuristics 22: *dispatch, pathLeaf, RP excess, RP critical, RP max, uncover*. This combination was also inspired by Warren [28].

6.4 Evaluation of Combinations of Heuristics

Figure 6.3 shows the result of experiments on the integer benchmarks, and figure 6.4 shows the result of experiments on the floating point benchmarks. Appendix D contains the results on individual benchmarks.

In the second experiment the number of copies, that is, the number of concurrent benchmarks run, was set to 10. LLVM's generic scheduler, which was run in both experiments, yielded a better result with more copies. This suggest that the value of static scheduling, increases when the level of throughput increases. Heuristics 1, 2 and 19 were the only combinations that did not include register usage. They fell behind a bit on the integer benchmarks, which was to be expected considering the result of the first experiment, in which the register pressure heuristics yielded the best result.

Heuristics 8 and 9 yielded the best results on the integer benchmarks, but performed worse on the floating point. Heuristics 13 performed better than the other heuristics on the floating point benchmarks, but fared worse on the integer benchmarks. It appears difficult to select heuristics suitable for both types of benchmark, indeed, a universal heuristic might be too ambitious. However, heuristics 12 and 14 are two possibilities.

Heuristics 12, 13, 14 and 15 are useful examples to demonstrate the difficulties of selecting a priority order. Starting with heuristics 12 we have: *dispatch, RP critical, execTime, execSucc, pathLeaf*. The reasoning behind this combination is, pick an instruction which can be dispatched in the same group as the previously scheduled instruction. If neither, or both, candidates can fit into the same group, pick the one with less register pressure. Otherwise avoid stalls, and balance the progress through the dag. Heuristics 15 replace *RP critical* with *RP excess*, however this seem to decrease the performance. Heuristics 13 and 14 use both *RP excess* and *RP critical*, but give different priority to *pathLeaf*.

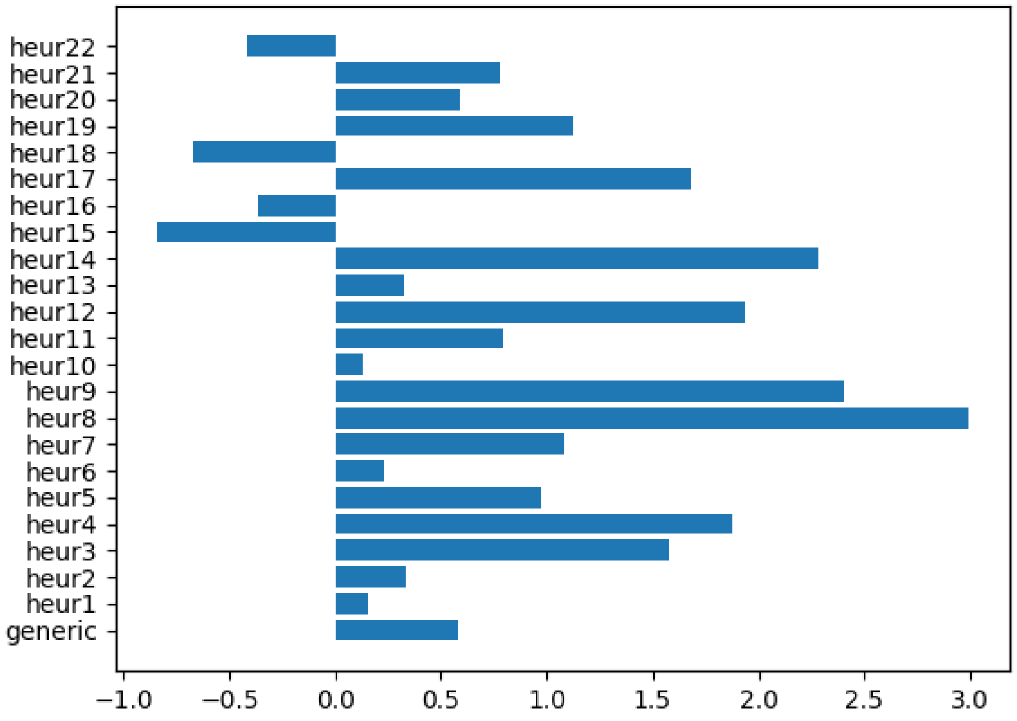


Figure 6.3: Integer benchmarks, results of the Est. SPECrate 2017 int base metric. The y-axis indicates the heuristic, and the x-axis indicates the difference, in percent, between the heuristic and baseline. Baseline is the original program order. A negative result means that the heuristic was slower than baseline.

If we look at specific benchmarks, figures D1 through D9, there is more variance around baseline than in the first experiment. Most of the combinations of heuristics focus on dispatch and register usage, giving them a high priority, based on the results from the first experiment. The RP heuristics work best when they are used together and in the order of: Excess, critical, max. On their own, it seem that RP critical works best in combination with dispatch.

Figure D2 shows some interesting results of differences up to 30 percent. Recall that each benchmark is run twice, and the slowest of the two runs is used to compute the results. Of all the benchmarks, 502.gcc_r showed the most variation between the two runs. For example, eight of the combinations had a difference of more than 10 percent between the two runs, and baseline had a difference of 27 percent between iteration one and two. This suggests that it might have been preferred to go with the option of running each benchmark three times and use the median to compute the result.

A majority of the heuristics showed an increase in performance on the floating point benchmarks, except for on the 508.namd_r benchmark on which only the generic scheduler showed a slight increase in performance. Overall, only two heuristics showed a decrease in performance. It is interesting to look at the heuristics that had a negative difference on either

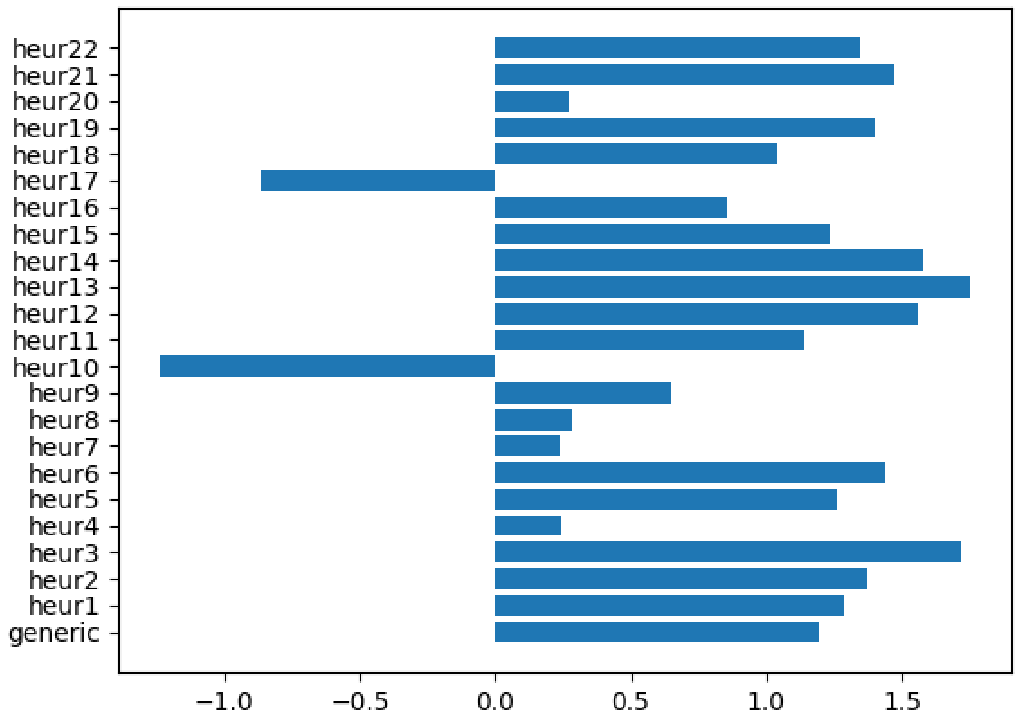


Figure 6.4: Floating point benchmarks, results of the Est. SPECrate 2017 fp base metric. The y-axis indicates the heuristic, and the x-axis indicates the difference, in percent, between the heuristic and baseline. Baseline is the original program order. A negative result means that the heuristic was slower than baseline.

the integer benchmarks or on the floating point benchmarks. These heuristics do not overlap, and it is difficult to say if the decrease in performance is due to a common denominator. By observing the combinations we can see that they have *dispatch* and *pathLeaf* in common, but since these heuristics are used in a majority of the combinations, it seem unlikely that they alone would be the reason for the decrease in performance. The following are some observations that can be made by comparing the combinations with a negative result with some of the combinations that had positive results.

- Heuristics 17 and 18 are permutations of Heuristics 9. In Heuristics 17, RP excess has moved down in priority. In Heuristics 18, pathLeaf and execTime has switched positions.
- Heuristics 10 has the same combination of heuristics as Heuristics 9, but without execTime. Heuristics 11 has the same combination of heuristics as Heuristics 9, but without pathLeaf. Heuristics 10 has worse performance than 11 on both the integer and floating point benchmarks.
- Heuristics 12, 15 and 16 use the same combination of heuristics, except that they use

different RP heuristics. Both Heuristics 15 and 16 have a negative result on the integer benchmarks.

- Heuristics 21 and 22 use the same combination of heuristics, except Heuristics 21 use pathLeafDelay, and Heuristics 22 use pathLeaf. Heuristic 22 has a negative result on the integer benchmark.
- Heuristics 10 and 18 has pathLeaf as the least prioritized heuristic.
- Heuristics 17 change the order of the RP heuristics.

From these observations we can conclude: In general it appears that *pathLeaf* should be given a higher priority than *execTime*, with the exception if *execTime* is followed by *execSucc*, in which case register usage appears to have a bigger impact on the outcome of the heuristics. The structural heuristic *execSucc* seem to be doing better on the integer benchmarks when it is used in combination with register usage. This would suggest that balance across the dag leads to more registers being live at the same time. Likewise for the uncovering heuristic *uncover* on the floating point benchmarks, which was not very useful on its own. The dispatch heuristic should be given a high priority, preferable in connection with register usage. Those heuristics which give least priority to the dispatch heuristic, fall behind on the integer benchmarks, but show similar results on the floating point benchmarks as those heuristics which give dispatch the highest priority.

Chapter 7
Conclusion

List scheduling has a long tradition in computer science and it is the fundamental algorithm for instruction scheduling. The intriguing part of the list scheduling algorithm is how we decide which instruction to schedule next. When studying the literature to learn more about list scheduling I found that Gibbons and Muchnick [7] together with Warren [28] where very helpful in laying down the details. Gibbons and Muchnick describes how to make a previous version of a post-pass, due to Hennessy and Gross [9], more efficient. Their algorithm is based on a dependency graph which prevents deadlocks and they select instructions without lookahead. They used three heuristics to determine the "best" candidate. These heuristics were based on two guidelines. If possible, schedule an instruction which will not interlock with the last one scheduled. When faced with a choice, schedule an instruction which is most likely to interlock with its successors. I.e. select instructions which are likely to cause interlocks, but not with the instruction that follows it. Warren's algorithm builds upon a previous scheduling algorithm employed in the PL.8 Compiler. The *leveling algorithm* he refers to is essentially a list scheduling algorithm which schedules instructions twice, once before and then again after register allocation.

For this thesis I was particularly interested in learning about the types of heuristics that have traditionally been employed by list scheduling algorithms. For this purpose I found that the survey *Efficient DAG Construction and Heuristic Calculation for Instruction Scheduling* by Smotherman et al. [26] was an invaluable source of information. The paper provided a detailed list of 26 heuristics, organized into six categories. This classification of heuristics have been very useful when reading other papers, being able to identify certain patterns or heuristics that tend to be popular. In this thesis, 24 individual list scheduling heuristics, as well as 22 combinations of heuristics, have been compared using Clang/LLVM and targeting POWER8. Experiments were carried out on a subset of the SPEC CPU 2017 benchmarks, both integer and floating point. During the experiments the categories, register usage, dispatch and critical path performed relatively well across both types of benchmark. In general, a heuristic that performed better than the other heuristics on the integer benchmark, would perform worse on the floating point benchmarks, and vice versa.

- ***How do individual heuristics compare against each other?*** There was not one definite heuristic, better than the rest. This might not come as a surprise since heuristics, by their definition, are not optimal. In general it seemed that the dynamic heuristics, *dispatch*, *RP excess*, *RP critical* and *RP max*, had the most stable performance. For example, while some static heuristics would show a lot of variation between different benchmarks, the dynamic heuristics did not deviate too much from baseline. It is difficult to say what makes one heuristic better than another. When developing the dispatch heuristic it became clear that the more knowledge the compiler has about the processor, the better it can make decisions. If this is true also in the case of static heuristics, or heuristics that are not target specific, I don't know.

- ***Is there any category of heuristics that seem to perform better, or worse, than the others?*** The uncovering heuristics fell behind the rest of the heuristics on both the integer and floating point benchmarks. Heuristics in the critical path category showed the most variation among themselves.

- ***For different combinations of heuristics, what can be said about the order in which the heuristics appear?*** The original program order, used as a baseline, proved to be a good heuristic, and since there is no ambiguity in program order, it serves as the final heuristic in any combination. In this sense, a good combination of heuristics, should not contain too many heuristics, preferably a few heuristics which can easily differentiate between two instructions, and fall back on program order early on. All but three of the combinations ordered the dynamic heuristics, *dispatch*, *RP excess*, *RP critical* or *RP max*, before any of the static heuristics. And only one combination, Heuristics 3, which contained both dynamic and static heuristics gave higher priority to a static heuristic. I think this is unfortunate because, while it appears that it is more beneficial to order the dynamic heuristics before the static, this can not be concluded from the results.

The results of the second experiment suggest that static scheduling, in the form of list scheduling, still has a vital role to play. At least on the floating point benchmarks a majority of the combinations had a positive difference to baseline. However, as processors evolve, so does dynamic scheduling. For example, on the integer benchmarks we see a lot more variance around baseline across the different combinations. Many of the aspects of local scheduling are better performed by dynamic scheduling. A too aggressive, or careless, list scheduling heuristic may end up degrading the performance. This relationship between static and dynamic scheduling is interesting, how the choices of the static scheduler affect the opportunities for the dynamic scheduler.

Future Work

Many things had to be left out simply due to time constraints. The number of permutations of different combinations of heuristics quickly becomes unmanageable, and only a small subset could be tested. Besides more heuristics it would have been interesting to compare, for example, bottom up, and top down scheduling approach[4] to the bidirectional approach used by the PowerPC back-end. It would also be interesting to compare list scheduling to other scheduling techniques, for example a global approach.

References

[1] M. Auslander and M. Hopkins. An Overview of the PL.8 Compiler. *SIGPLAN Not.*, 17(6):22–31, June 1982.

[2] S. Beaty, S. Colcord, and P. Sweany. Using genetic algorithms to fine-tune instruction-scheduling heuristics. *Proc. of the Int' Conf. on Massively Parallel Computer Systems*, 1996.

[3] P. P. Chang, D. M. Lavery, S. A. Mahlke, W. Y. Chen, and W. . W. Hwu. The importance of prepass code scheduling for superscalar and superpipelined processors. *IEEE Transactions on Computers*, 44(3):353–370, 1995.

[4] G. Chen. *Effective Instruction Scheduling with Limited Registers.* PhD thesis, Harvard University, Cambridge, MA, USA, 2001. AAI3011341.

[5] K. D. Cooper, P. Schielke, and D. Subramanian. An experimental evaluation of list scheduling. Rice Computer Science Technical Report 98-326, Department of Computer Science, Rice University, Houston, Texas, USA, September 1998.

[6] J. A. Fisher, D. Landskov, and B. D. Shriver. Microcode compaction: Looking backward and looking forward. In *Proceedings of the May 4-7, 1981, National Computer Conference*, AFIPS '81, page 95–102, New York, NY, USA, 1981. Association for Computing Machinery.

[7] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. *SIGPLAN Not.*, 21(7):11–16, July 1986.

[8] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 88–98, New York, NY, USA, 2014. ACM.

[9] J. L. Hennessy and T. Gross. Postpass code optimization of pipeline constraints. *ACM Trans. Program. Lang. Syst.*, 5(3):422–448, July 1983.

[10] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach.* Elsevier Science, 2011.

[11] R. A. Huff. Lifetime-sensitive modulo scheduling. *SIGPLAN Not.*, 28(6):258–267, June 1993.

[12] IBM. *Power ISA version 2.07*, May 2013.

[13] IBM. *POWER8 Processor User's Manual for the Single-Chip Module, version 1.3*, March 2016.

[14] S. M. Krishnamurthy. A brief survey of papers on scheduling for pipelined processors. *SIGPLAN Not.*, 25(7):97–106, July 1990.

[15] D. Landskov, S. Davidson, B. Shriver, and P. W. Mallett. Local microcode compaction techniques. *ACM Comput. Surv.*, 12(3):261–294, September 1980.

[16] LLVM Compiler Infrastructure. LLVM source code documentation. URL: https://llvm.org/doxygen/index.html; accessed 19-May-2020.

[17] T. Æ. Mogensen. *Introduction to Compiler Design.* Undergraduate Topics in Computer Science. Springer London, 2011.

[18] E. Moss, P. Utgoff, J. Cavazos, D. Precup, D. Stefanovic, C. Brodley, and D. Scheeff. Learning to schedule straight-line code. In *Proceedings of the 10th Conference on Advances in Neural Information Processing Systems (NIPS)*, Denver, Colorado, 1997.

[19] B. R. Rau and J. A. Fisher. Instruction-level parallel processing: History, overview, and perspective. *J. Supercomput.*, 7(1–2):9–50, May 1993.

[20] T. Russell, A. M. Malik, M. Chase, and P. van Beek. Learning basic block scheduling heuristics from optimal data. In *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '05, pages 242–253. IBM Press, 2005.

[21] J.-J. Shieh and C. Papachristou. On reordering instruction streams for pipelined computers. *SIGMICRO Newsl.*, 20(3):199–206, August 1989.

[22] R. Silvera, J. Wang, G. Gao, and R. Govindarajan. A register pressure sensitive instruction scheduler for dynamic issue processors. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, PACT '97, page 78, USA, 1997. IEEE Computer Society.

[23] B. Sinharoy, J. A. Van Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, J. E. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. W. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, and K. M. Fernsler. IBM POWER8 processor core microarchitecture. *IBM Journal of Research and Development*, 59(1):2:1–2:21, 2015.

[24] J. Skeppstedt. *An Introduction to the theory of optimizing compilers with performance measurements on POWER.* Skeppberg, 2016.

[25] J. Skeppstedt and C. Söderberg. *Writing efficient C code : a thorough introduction.* Skeppberg, 2016.

[26] M. Smotherman, S. Krishnamurthy, P. S. Aravind, and D. Hunnicutt. Efficient dag construction and heuristic calculation for instruction scheduling. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, MICRO 24, pages 93–102, New York, NY, USA, 1991. ACM.

[27] M. G. Valluri and R. Govindarajan. Evaluating register allocation and instruction scheduling techniques in out-of-order issue processors. In *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00425)*, pages 78–83, 1999.

[28] H. S. Warren, Jr. Instruction scheduling for the IBM RISC system/6000 processor. *IBM J. Res. Dev.*, 34(1):85–92, January 1990.

Appendices

Appendix A
The POWER8 Processor Core

This section will give a brief introduction to the POWER8 processor core by looking at how an instruction goes from assembler output to being executed on the machine. In the context of instruction scheduling, one of the more interesting aspects of POWER8 is its innate ability to achieve ILP. POWER8 is a superscalar multiprocessor and can have up to twelve cores enabled per chip. It is worth noting that parallelism, achieved by having multiple cores, is not the same as when we talk about ILP, even though parts of a program is certainly executed out of order and in parallel. For an introduction to the Power architecture see for example [25], for information specific to POWER8, the user's manual [13], and for details about instruction set architecture, the Power ISA [12].

Starting from memory, instructions flow through different issue queues on their way to the functional units. The memory hierarchy of the core consists of a 32 KB L1 instruction cache, a 64 KB L1 data cache and a 512 KB L2 cache. The core has a *long pipeline design*, up to 23 stages from I-cache access to write back for most floating point operations [13]. The multi-threaded design of the core allows for a single thread mode, ST, or simultaneous multithreading, SMT, two, four or eight way. The structure of the pipeline consists of a *master pipeline* and several *execution unit pipelines*. The master pipeline will feed the mapping, sequencing and dispatch functions with instructions in-order, speculatively. The execution unit pipelines can issue both speculative and non-speculative operations out-of-order. These pipelines execute independently from each other, as well as from the master pipeline [23].

The first stages involves instruction fetching and branch prediction, decoding and pre-processing. Each core has an instruction fetch unit, IFU, with a 32 KB, 8-way set-associative instruction cache. Instructions are read from the unified L2 cache, into the I-cache, either as demand loads due to an I-cache miss or instruction prefetches. There can be up to eight outstanding read requests. Prefetches are not guaranteed and instruction prefetching is only supported in ST, SMT2 and SMT4 modes. In SMT8 mode there is no instruction prefetching in order to save on memory bandwidth [23].

In each cycle up to eight instructions can be fetched from the I-cache and placed in instruction buffers. Branch prediction works by scanning the fetched instructions, looking for branches and via branch history tables, do bookkeeping for global and local prediction. There is also a third table, the selector table, which keeps track of which of the two prediction schemes works better. In POWER8, all conditional branches are predicted, and none of the unconditional branches are predicted. The outcome of a branch is known once the branch has flowed through the pipeline and been executed by the branch execution unit. If the prediction was correct the instruction completes like it would normally. In the case of misprediction the instruction fetch logic will issue a flush.

In each cycle up to eight instructions can be taken out of the instruction fetch buffer, to be decoded and dispatched. In ST mode a dispatch group consists of up to six non-branch instructions, and two branches. In SMT mode there are two dispatch groups, each can have up to three non-branch instruction and one branch. Group formation follows a set of rules, for example some instructions require that no other instructions follow them in the dispatch group. These instructions will be marked as Last, and end the current group. Some complex instructions are cracked, a process in which instructions get *cracked* into multiple simpler internal operations. Cracked instructions must be the first instruction of a group.

The instruction sequencing unit, ISU, is responsible for renaming registers to support out-of-order execution, dispatching of instructions to different issue queues, and issuing the instructions from the queues to the respective execution pipeline. In POWER8, instructions are dispatched on a group basis. In order to dispatch a group, it is required that resources such as rename registers and queue entries are available for each instruction in the group. In SMT mode the two groups dispatched can be from different threads. The ISU keeps track of the instructions after dispatch via a Global Completion table.

There are three different issue queues in which dispatched instructions are stored, the Branch Issue Queue for branch instructions, Condition Register Queue for condition register instructions and for all other instructions, the Unified Issue Queue which consists of two halves, UQ0 and UQ1. From each queue, instructions can be issued out-of-order to the corresponding execution unit. Older ready instructions have a higher priority. The POWER8 core has the following sixteen functional units:

- Two symmetric load/store units, LSU.
- Two load-only units, LU.
- Two symmetric fixed-point units, FXU.
- Four floating-point units, FPU.
- Two VMX execution units.
- One Crypto unit.
- One decimal floating-point unit, DFU.
- One branch execution unit, BR.
- One condition register logical execution unit, CRL.

In each cycle a total of ten instructions can be issued to the functional units, one branch instruction, one condition register logical, two instructions to the LSU, LU and FXU each. The LSU and LU are capable of executing simple fixed-point operations, in addition to loads and stores. Two instructions to the vector-scalar unit, VSU, which can be floating-point, VSX or VMX. DFU and Crypto instructions can also be issued via VSU slots, at most one of each per cycle.

Instructions are issued out-of-order, and executed speculatively. Sometimes speculative instructions must be flushed from the instruction pipeline, for example, when branch mis-predictions occur or, hazards due to executing load/stores out-of-order. This is handled by the ISU. If an instruction is executed successfully, that is, without being rejected, it will be marked as finished. A group can complete once all of its instructions have been marked as finished, at this point the resources, held by the instructions in a group, are released and the results of the instructions can be used by subsequent instructions.

Listing A.1: An example of PowerPC assembly, output of `objdump -d` on the object file generated from the example code in listing 2.2.

```
0000000000000000 <f>:
   0:   00 00 4c 3c     addis   r2,r12,0
   4:   00 00 42 38     addi    r2,r2,0
   8:   a6 02 08 7c     mflr    r0
   c:   f8 ff e1 fb     std     r31,-8(r1)
  10:   10 00 01 f8     std     r0,16(r1)
  14:   81 ff 21 f8     stdu    r1,-128(r1)
  18:   78 0b 3f 7c     mr      r31,r1
  1c:   70 00 df fb     std     r30,112(r31)
  20:   68 00 7f 90     stw     r3,104(r31)
  24:   68 00 7f 80     lwz     r3,104(r31)
  28:   02 00 03 2c     cmpwi   r3,2
  2c:   10 00 80 40     bge     3c <f+0x3c>
  30:   68 00 7f 80     lwz     r3,104(r31)
  34:   6c 00 7f 90     stw     r3,108(r31)
  38:   34 00 00 48     b       6c <f+0x6c>
  3c:   68 00 7f 80     lwz     r3,104(r31)
  40:   ff ff 63 38     addi    r3,r3,-1
  44:   b4 07 63 7c     extsw   r3,r3
  48:   01 00 00 48     bl      48 <f+0x48>
  4c:   78 1b 7e 7c     mr      r30,r3
  50:   68 00 7f 80     lwz     r3,104(r31)
  54:   fe ff 63 38     addi    r3,r3,-2
  58:   b4 07 63 7c     extsw   r3,r3
  5c:   01 00 00 48     bl      5c <f+0x5c>
  60:   00 00 00 60     nop
  64:   14 1a 7e 7c     add     r3,r30,r3
  68:   6c 00 7f 90     stw     r3,108(r31)
  6c:   6e 00 7f e8     lwa     r3,108(r31)
  70:   70 00 df eb     ld      r30,112(r31)
  74:   80 00 21 38     addi    r1,r1,128
  78:   10 00 01 e8     ld      r0,16(r1)
  7c:   f8 ff e1 eb     ld      r31,-8(r1)
  80:   a6 03 08 7c     mtlr    r0
  84:   20 00 80 4e     blr
```

Appendix B

Pseudocode for the Dispatch Heuristic

The dispatch heuristic takes as input two instructions, one is the currently best candidate, and the other is one of the instructions in the set of candidates that has not yet been compared against the best candidate. The heuristic returns `true` if it thinks that it is better to schedule one of the candidates over the other next, or `false`, if it does not reach a conclusion.

Listing B.1: Pseudocode describing the tryDispatchGroupTop-function.

```
bool tryDispatchGroupTop(cand, tryCand) {
  if (dispatch slot 0 is empty) {
    if (tryCand is 2-way cracked
        and cand is 3-way cracked) {
        tryCand.Reason = Dispatch;
        return true;
    }
    return false;
  }

  if (instrFusion(cand, tryCand))
    return true;

  if (dispatch slot 1 is empty
      or dispatch slot 2 is empty) {
    if ((cand is First or cand is Cracked)
         and (tryCand is not First
              and tryCand is not Cracked)) {
      if (tryDispatchSameHalfTop(Cand, TryCand)) {
        TryCand.Reason = Dispatch;
        return true;
        }
      }
      if ((tryCand is First or tryCand is Cracked)
           and (cand is not First
                and cand is not Cracked)) {
        if (Cand.Reason > Dispatch)
          Cand.Reason = Dispatch;
        return true;
      }
      return false;
    }

    if (dispatch slot 3 is empty) {
      if (tryCand is 2-way cracked
          and cand is 3-way cracked) {
          tryCand.Reason = Dispatch;
          return true;
      }
      return false;
    }

    if (dispatch slot 4 is empty
        or dispatch slot 5 is empty) {
      if ((cand is First or cand is Cracked)
           and (tryCand is not first
                and tryCand is not Cracked)) {
        if (tryDispatchSameHalfTop(Cand, TryCand)) {
          TryCand.Reason = Dispatch;
          return true;
        }
      }
      if ((tryCand is First or tryCand is Cracked)
          and (cand is not First
               and cand is not Cracked)) {
        if (Cand.Reason > Dispatch)
          Cand.Reason = Dispatch;
        return true;
      }
      return false;
    }
    return false;
}
```

Listing B.2: Pseudocode describing the tryDispatchSameHalfTop-function.

```
bool tryDispatchSameHalfTop(cand, tryCand) {
  if (cand is Last and try is Last)
    return false;

  if (cand is Last) {
    if (dispatch slot 1 is empty
        or dispatch slot 4 is empty)
      return true;
    else
      return false;
  }

  if (tryIsLast) {
    if (dispatch slot 1 is empty
        or dispatch slot 4 is empty)
      return false;
    else
      return true;
  }

  if (cand is alternate instruction
      and tryCand is not)
    return false;
  else if (tryCand is alternate instruction
           and cand is not)
    return true;

  if (dispatch slot < 3) {
    if (previous instruction needs microcode) {
      if (cand needs microcode)
        return false;
      else if (tryCand needs microcode)
        return true;
    }
  } else {
    if (other group needs microcode) {
      if (cand needs microcode)
        return true;
      else if (tryCand needs microcode)
        return false;
    }
  }
  return false;
}
```

Listing B.3: Pseudocode describing the tryDispatchGroupBot-function.

```
bool tryDispatchGroupBot(cand, tryCand) {
  if (tryCand is Last and cand is Last)
    return false;

  if (tryCand is 3-way cracked
      or cand is 3-way cracked)
    return false;

  if (current dispatch slot is 5
      or current dispatch slot is 2) {
    if (tryCand is Last) {
      tryCand.Reason = Dispatch;
      return true;
    } else if (cand is Last)
      return false;
    else if ((tryCand is not First
              and tryCand is not 2-way cracked)
              and (cand is First
                   or cand is 2-way cracked)) {
      tryCand.Reason = Dispatch;
      return true;
    }
    return false;
  }

  if (instrFusion(cand, tryCand))
    return true;

  if (current dispatch slot is 4
      or current dispatch slot is 1) {
    if (tryIsLast)
      return false;
    if (candIsLast) {
      tryCand.Reason = Dispatch;
      return true;
    }
    if (tryCand is 2-way cracked
        and cand is not 2-way cracked) {
      tryCand.Reason = Dispatch;
      return true;
    }
    if (tryCand is not First and cand is First) {
      tryCand.Reason = Dispatch;
      return true;
    }
    if (cand is neither First nor Cracked
        and tryCand is neither First nor Cracked) {
      if (tryDispatchSameHalfBot(Cand, TryCand)) {
        tryCand.Reason = Dispatch;
        return true;
      }
    }
    return false;
  }

  if (current dispatch slot is 3
      or current dispatch slot is 0) {
    if (tryIsLast)
      return false;
    if (candIsLast) {
      tryCand.Reason = Dispatch;
      return true;
    }
    if (cand is neither First nor Cracked
        and tryCand is neither First nor Cracked) {
      if (tryDispatchSameHalfBot(Cand, TryCand)) {
        tryCand.Reason = Dispatch;
        return true;
      }
    }
  }
  return false;
}
```

Listing B.4: Pseudocode describing the tryDispatchSameHalfBot-function.

```
bool tryDispatchSameHalfBot(cand, tryCand) {
  if (cand is alternate instruction and tryCand is not)
    return false;
  if (tryCand is alternate instruction and cand is not)
    return true;

  if (dispatch slot > 2) {
    if (previous instruction needs microcode) {
      if (cand needs microcode)
        return false;
      else if (tryCand needs microcode)
        return true;
    }
  } else {
    if (other group needs microcode) {
      if (cand needs microcode)
```

```
                return true;
            else if (tryCand needs microcode)
                return false;
        }
    }
    return false;
}
```

Listing B.5: Pseudocode describing the instrFusion-function.

```
bool instrFusion(cand, tryCand) {
    if (one candidate can fuse
         and the other one can not) {
        set the candidates reason to dispatch
        return true;
    }
    return false;
}
```

Appendix C

Individual Heuristics

This appendix include the results of individual heuristics on each benchmark. The baseline is the original program order. The y-axis indicates the heuristic. The x-axis indicates the difference, in percent, between the heuristic and baseline.

Integer Benchmarks

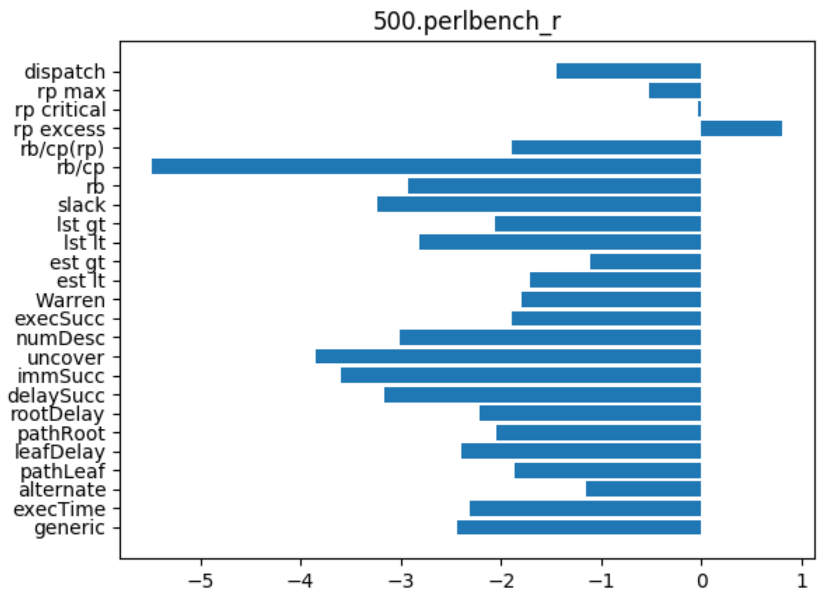


Figure C.1: 500.perlbench_r benchmark.

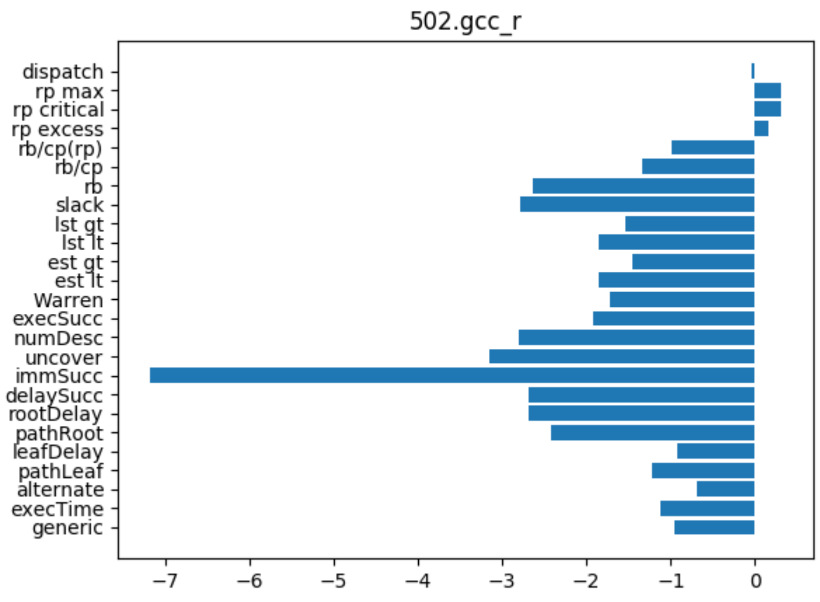


Figure C.2: 502.gcc_r benchmark.

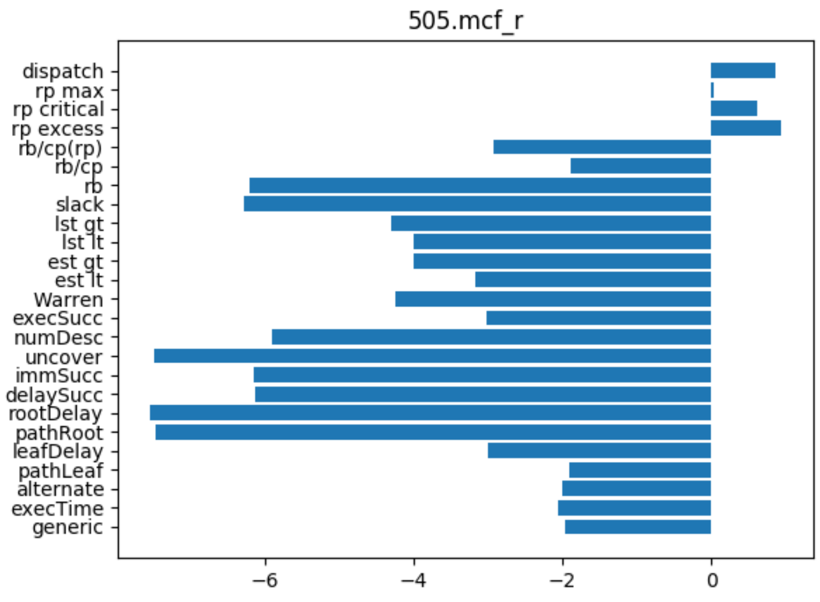


Figure C.3: 505.mcf_r benchmark.

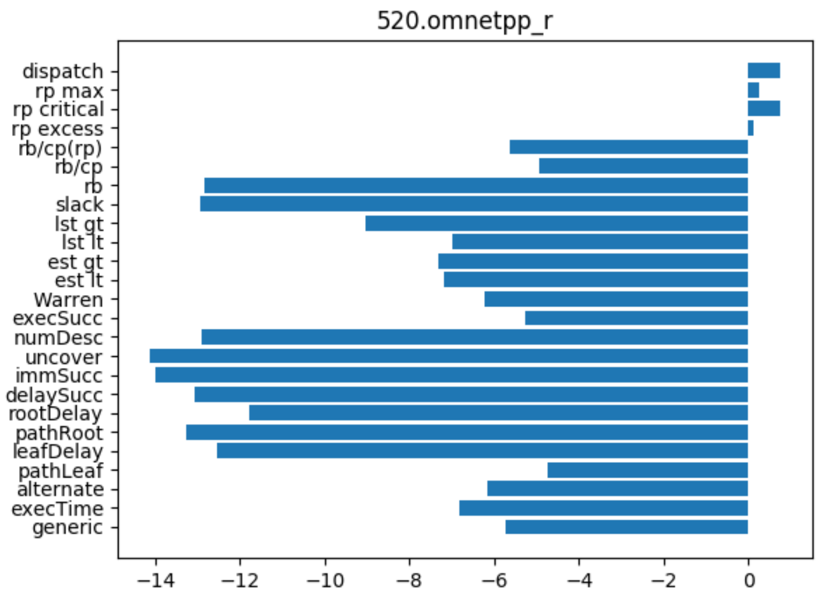


Figure C.4: 520.omnetpp_r benchmark.

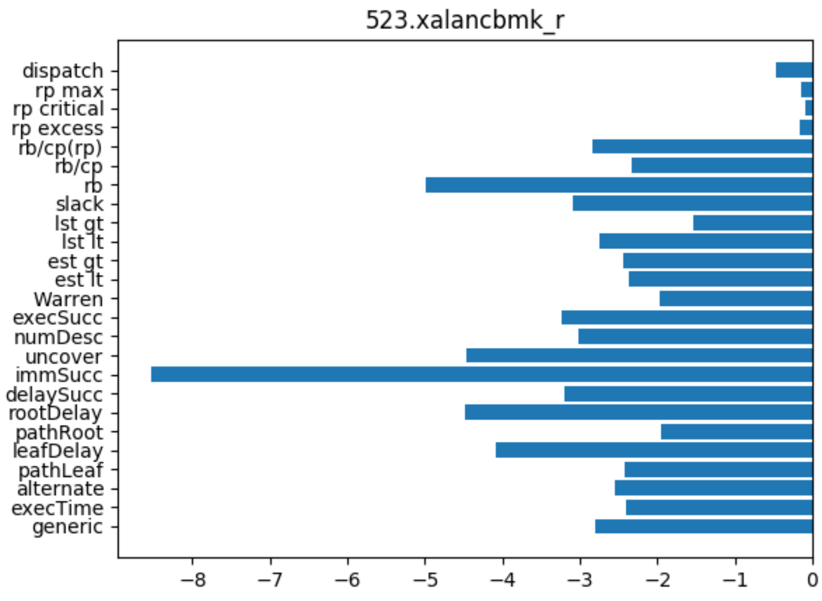


Figure C.5: 523.xalancbmk_r benchmark.

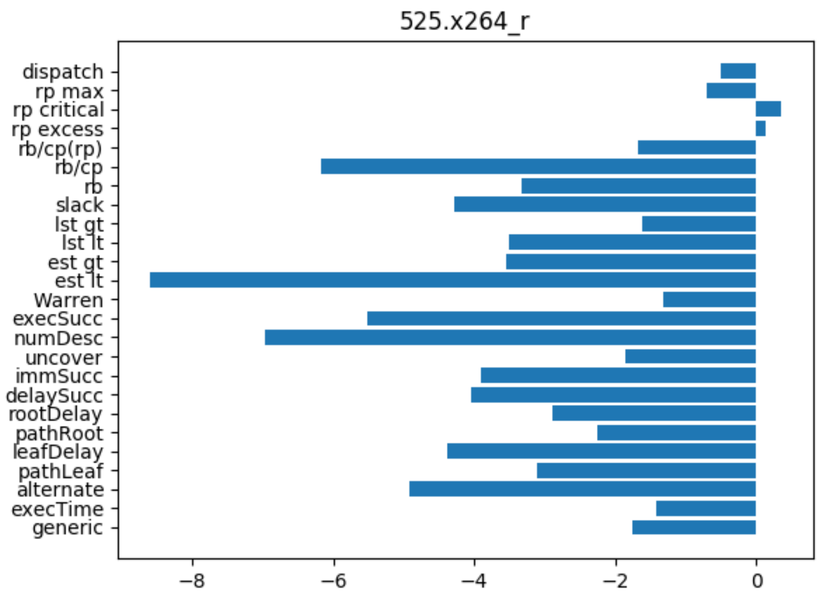


Figure C.6: 525.x264_r benchmark.

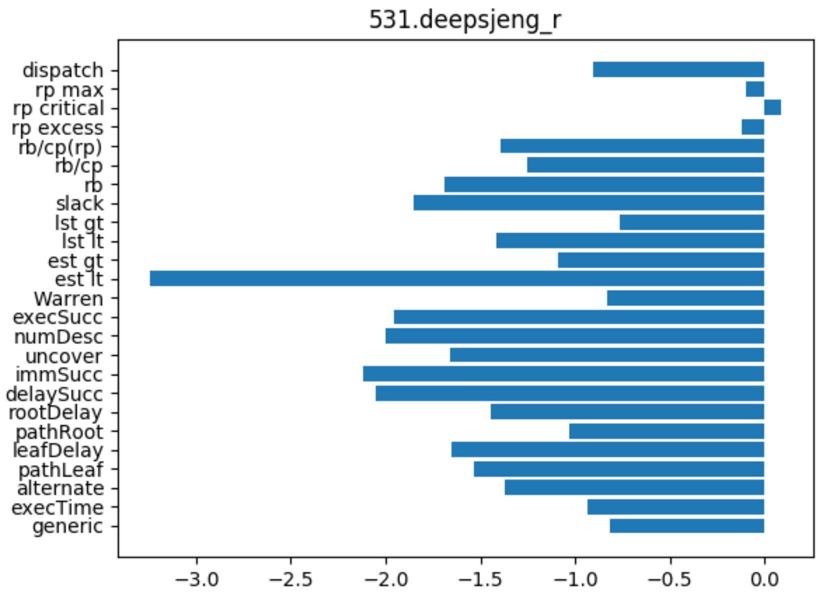


Figure C.7: 531.deepsjeng_r benchmark.

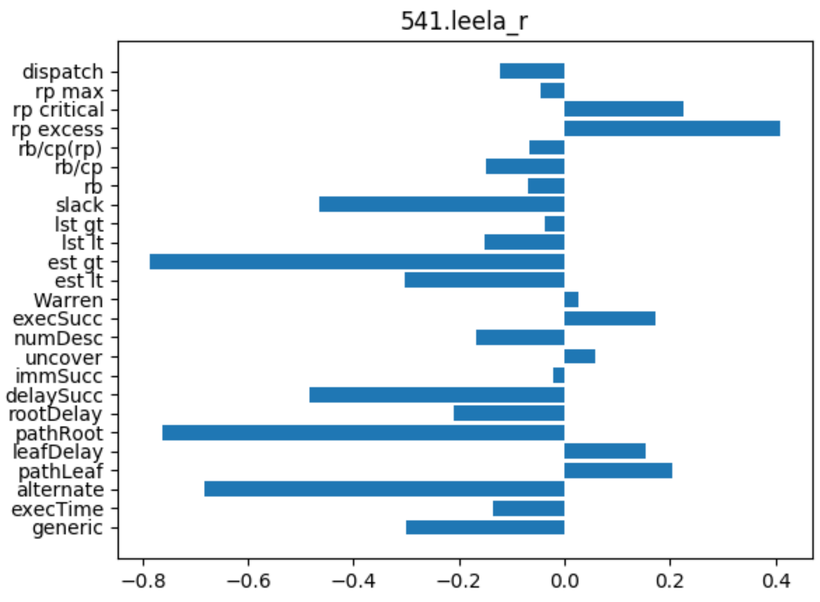


Figure C.8: 541.leela_r benchmark.

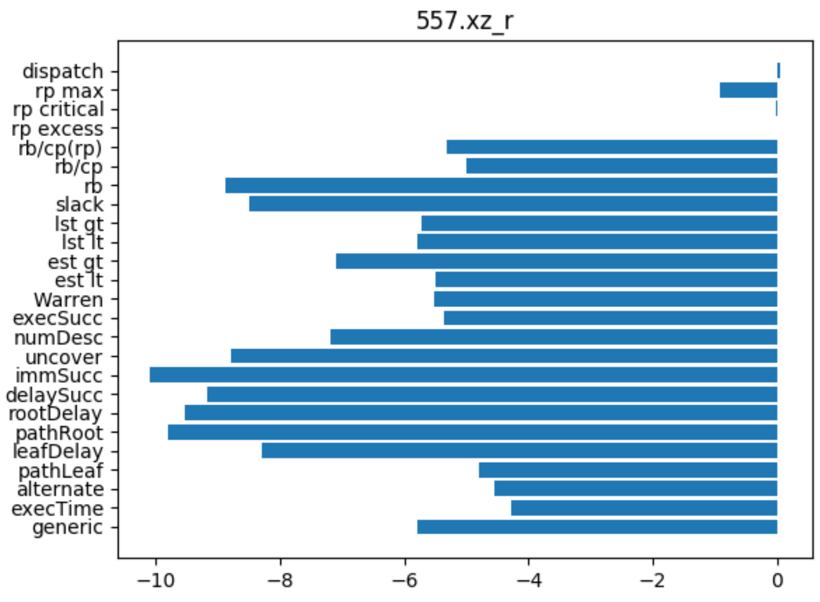


Figure C.9: 557.xz_r benchmark.

Floating Point Benchmarks

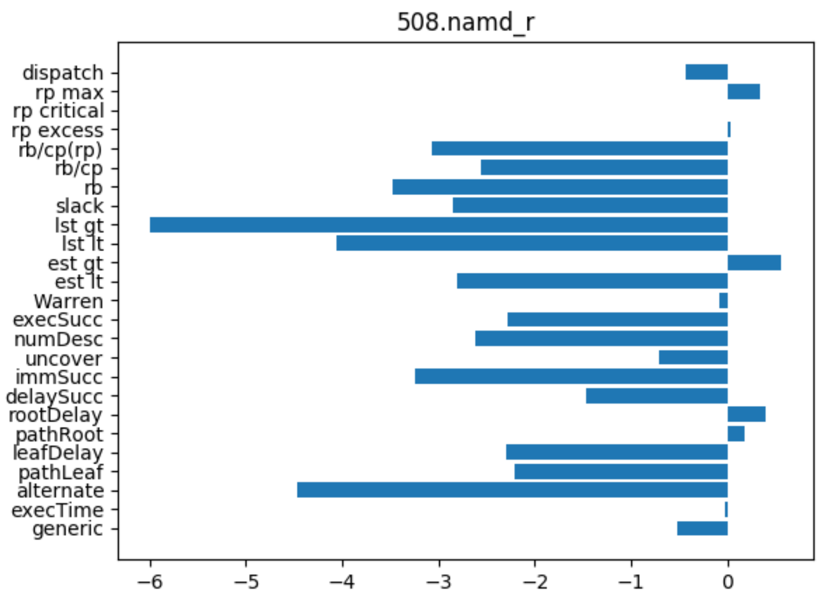


Figure C.10: 508.namd_r benchmark.

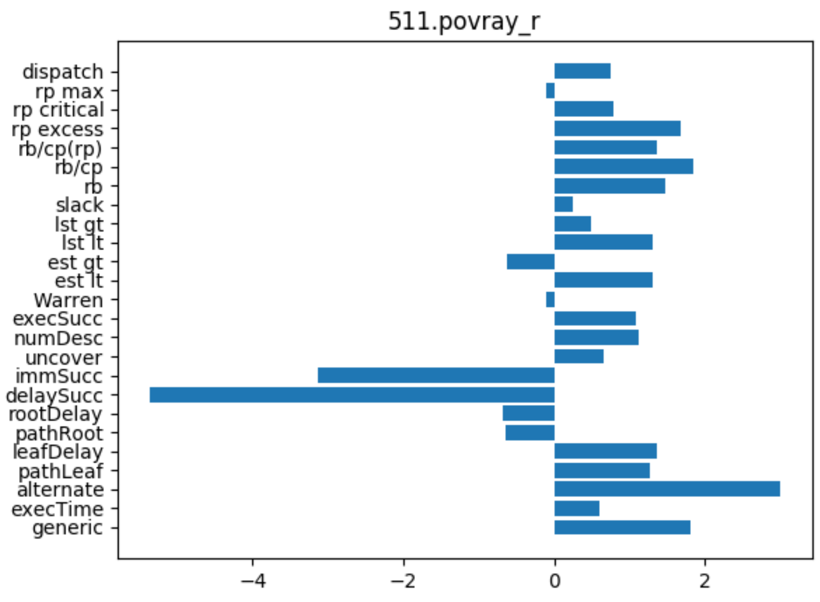


Figure C.11: 511.povray_r benchmark.

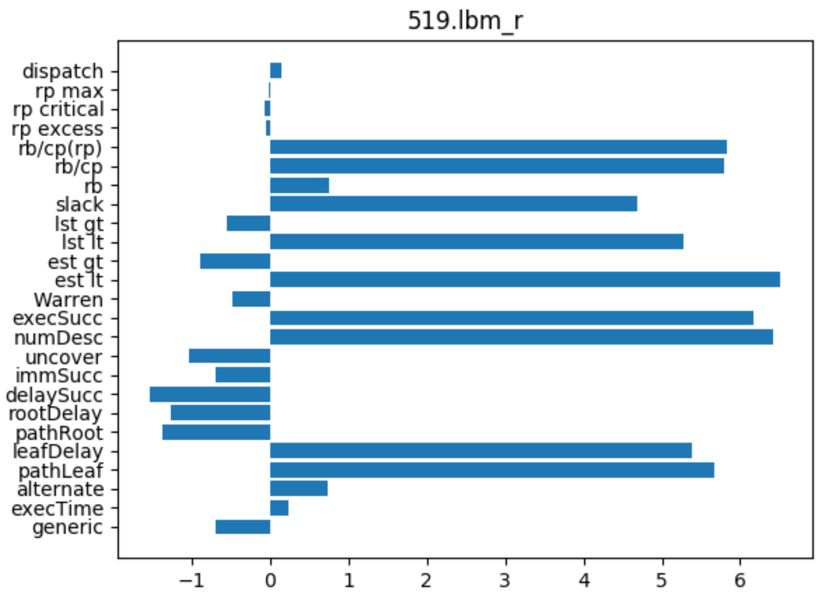


Figure C.12: 519.lbm_r benchmark.

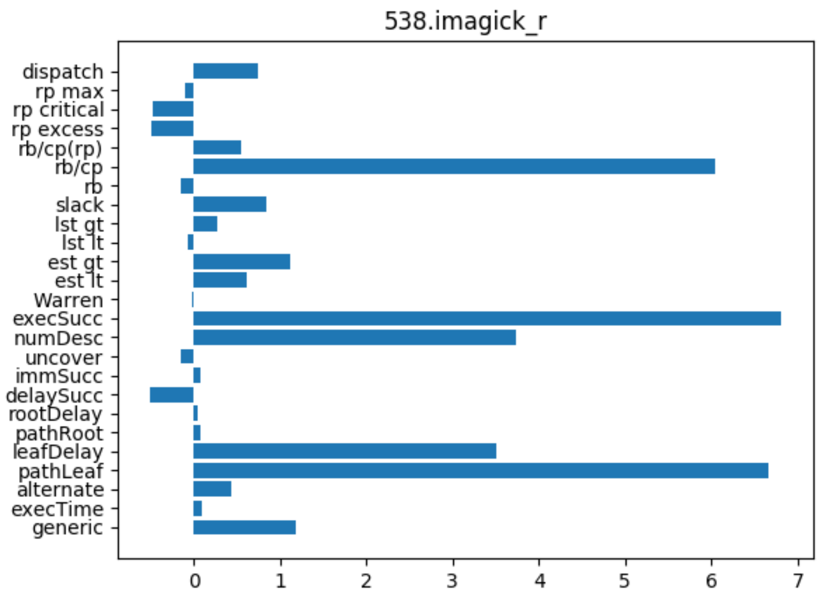


Figure C.13: 538.imagick_r benchmark.

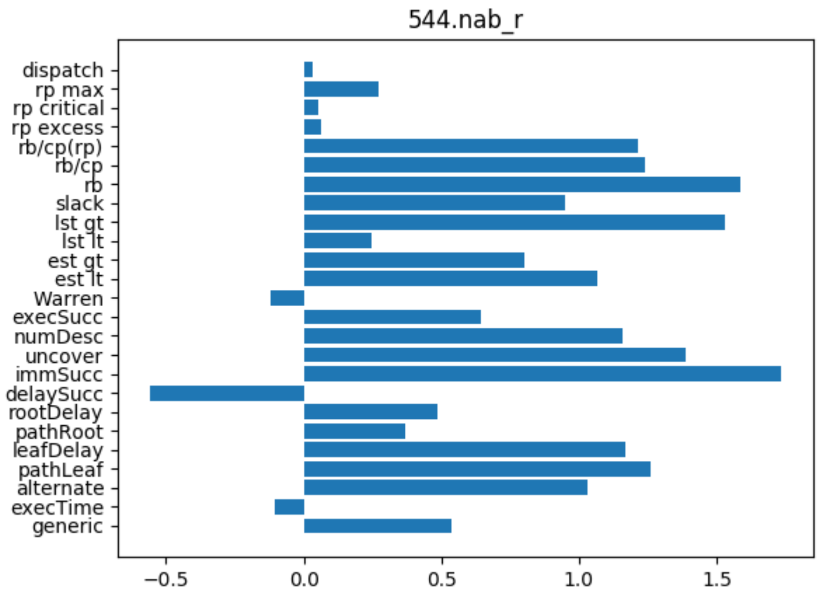


Figure C.14: 544.nab_r benchmark.

Appendix D

Combinations of Heuristics

This appendix include the results of different combinations of heuristics on each benchmark. The baseline is the original program order. The y-axis indicates the combination of heuristics. The x-axis indicates the difference, in percent, between the combination and baseline.

Integer Benchmarks

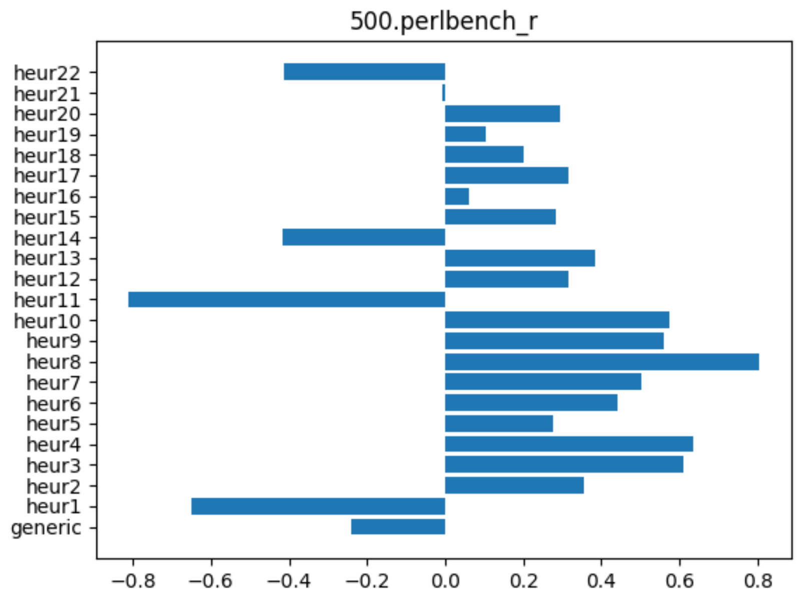


Figure D.1: 500.perlbench_r benchmark.

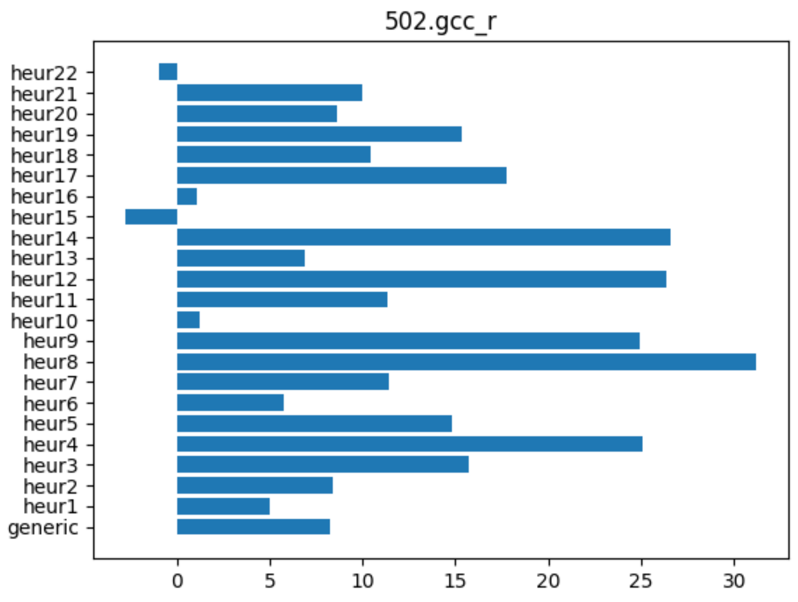


Figure D.2: 502.gcc_r benchmark.

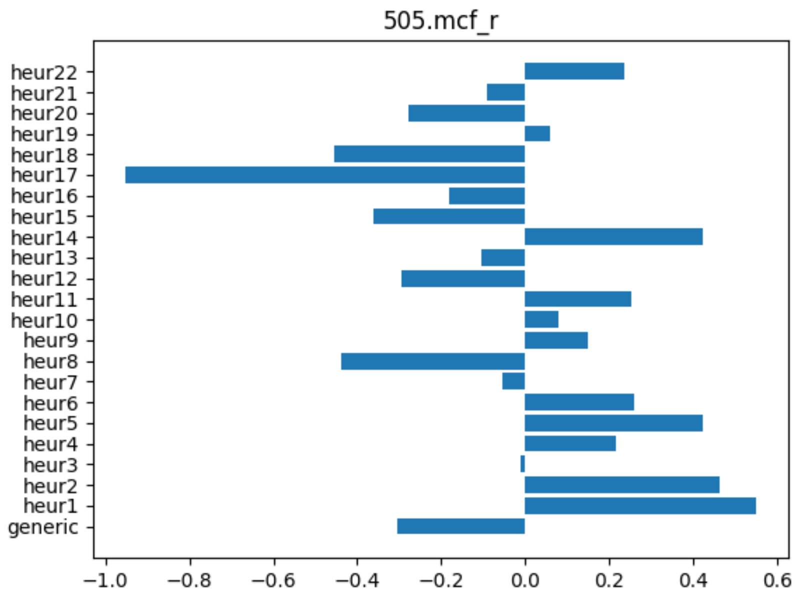


Figure D.3: 505.mcf_r benchmark.

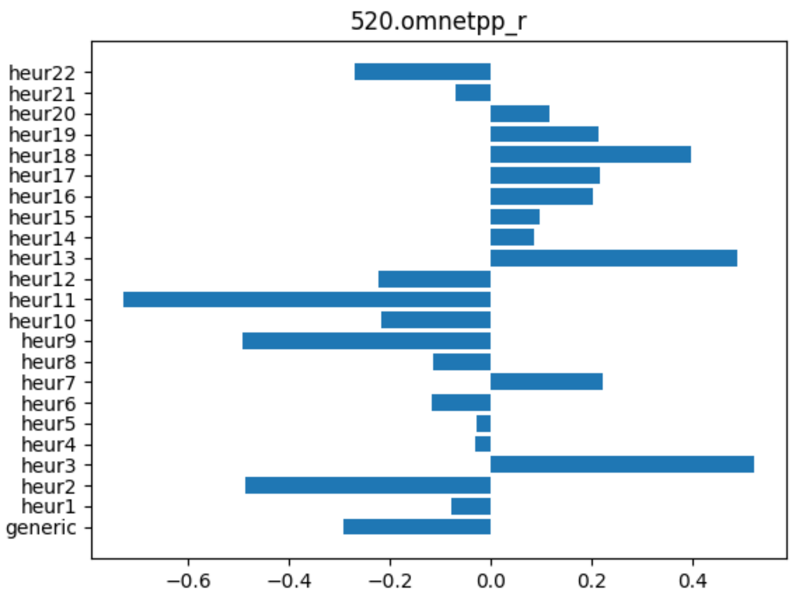


Figure D.4: 520.omnetpp_r benchmark.

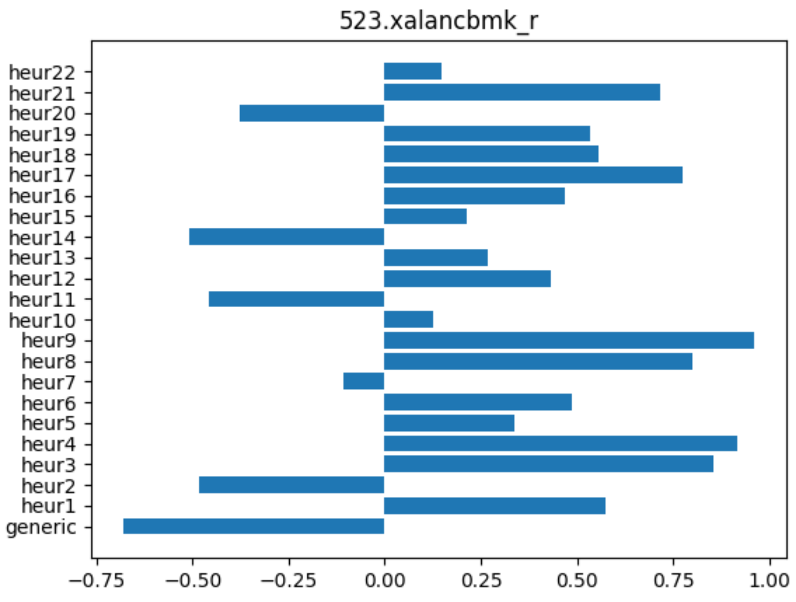


Figure D.5: 523.xalancbmk_r benchmark.

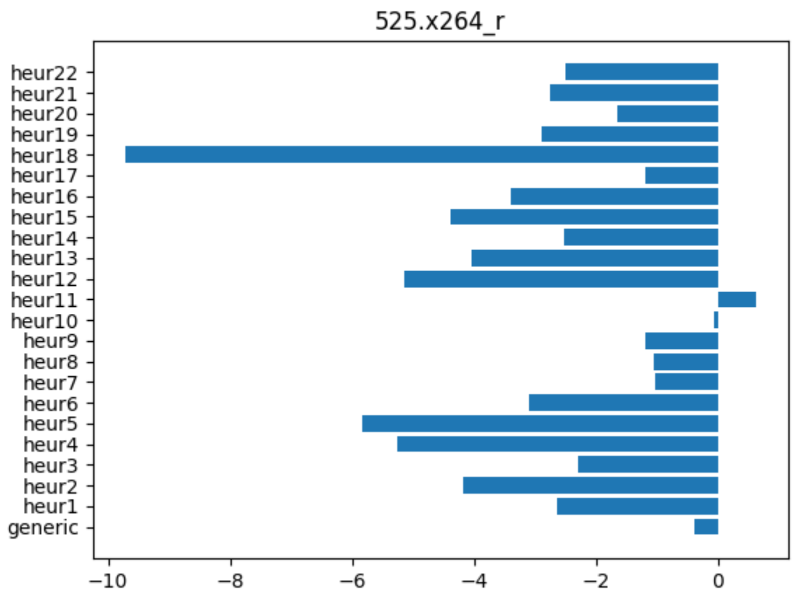


Figure D.6: 525.x264_r benchmark.

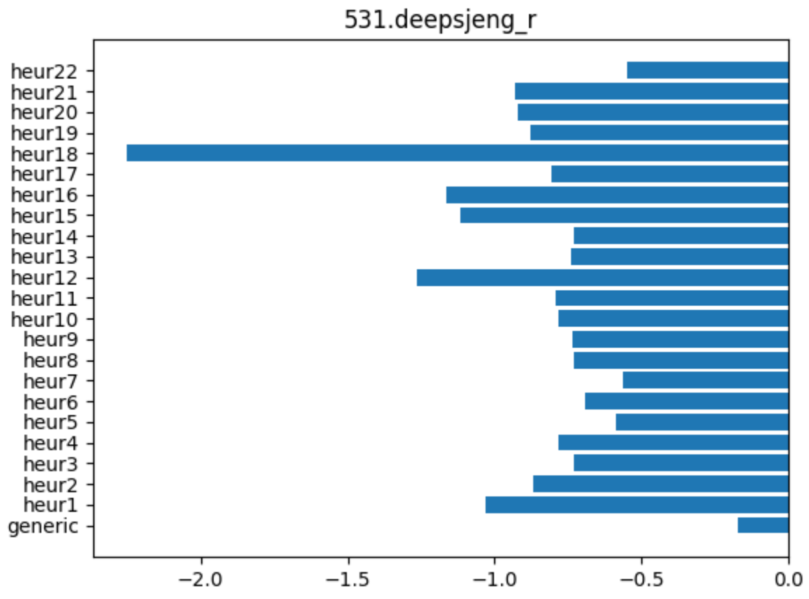


Figure D.7: 531.deepsjeng_r benchmark.

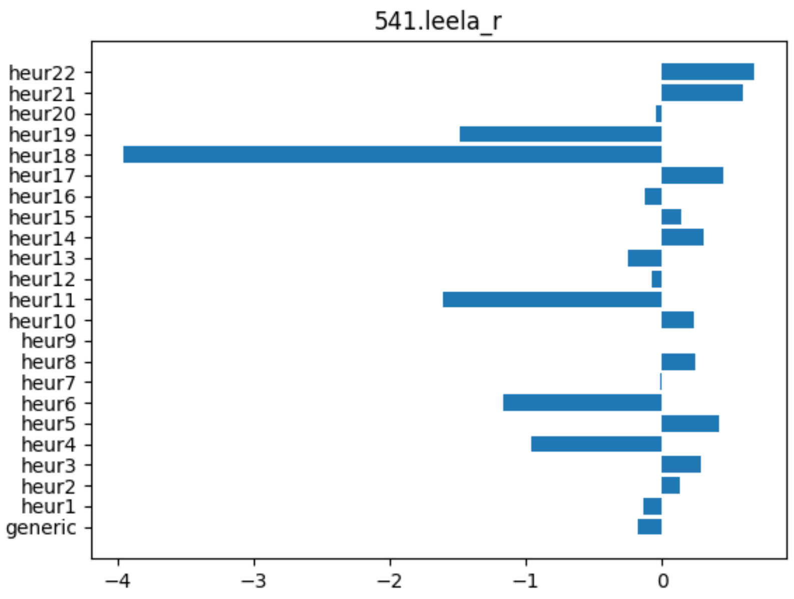


Figure D.8: 541.leela_r benchmark.

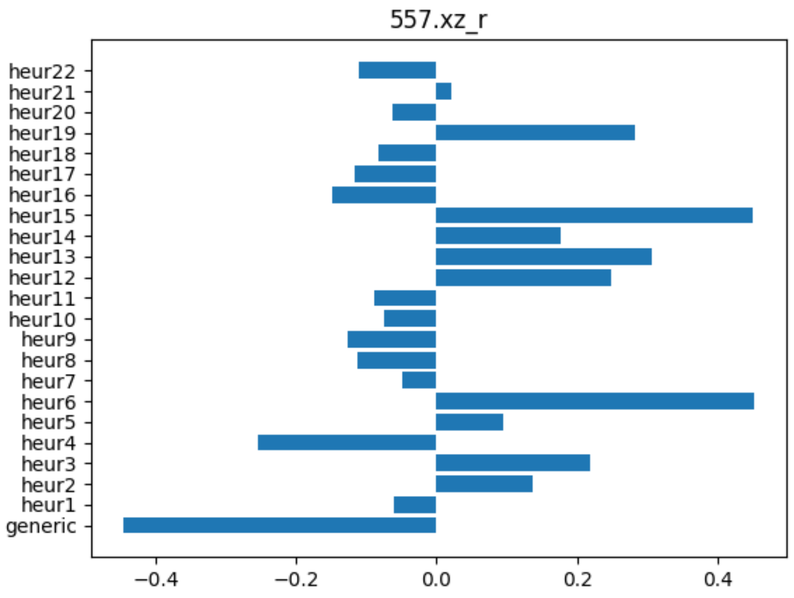


Figure D.9: 557.xz_r benchmark.

Floating Point Benchmarks

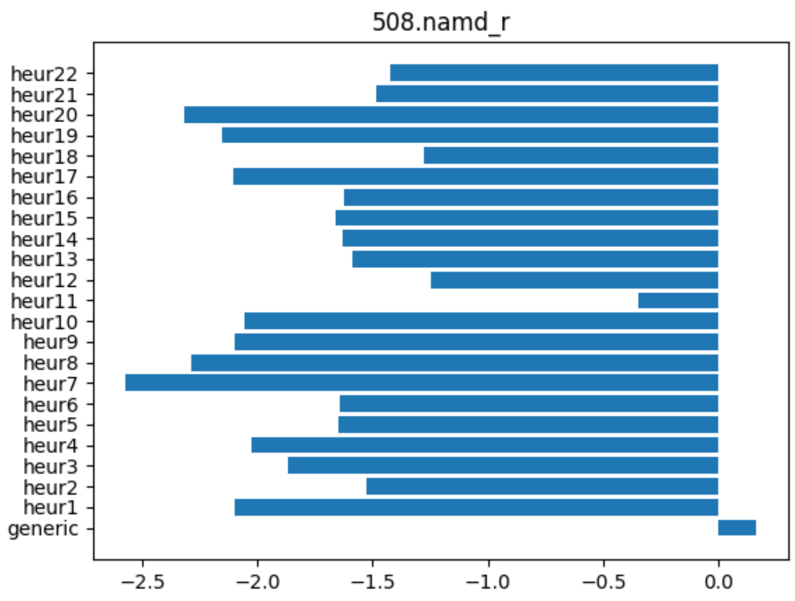


Figure D.10: 508.namd_r benchmark.

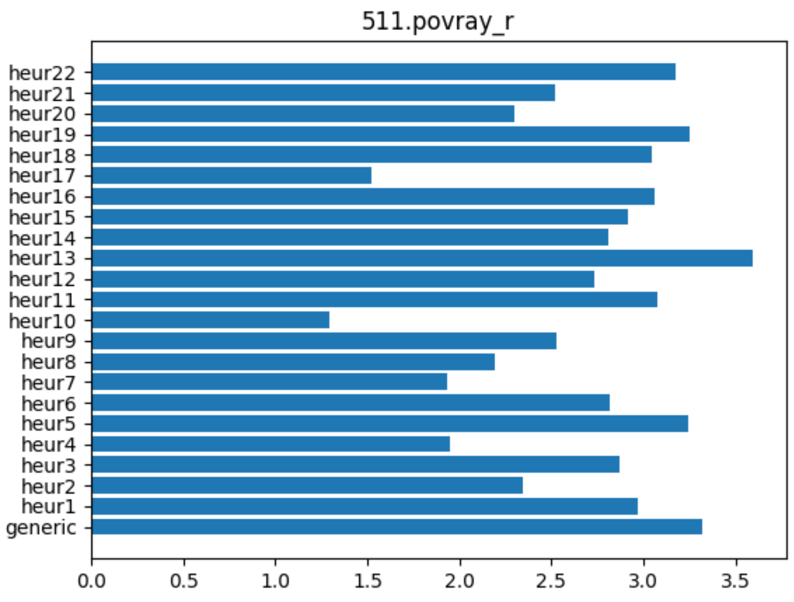


Figure D.11: 511.povray_r benchmark.

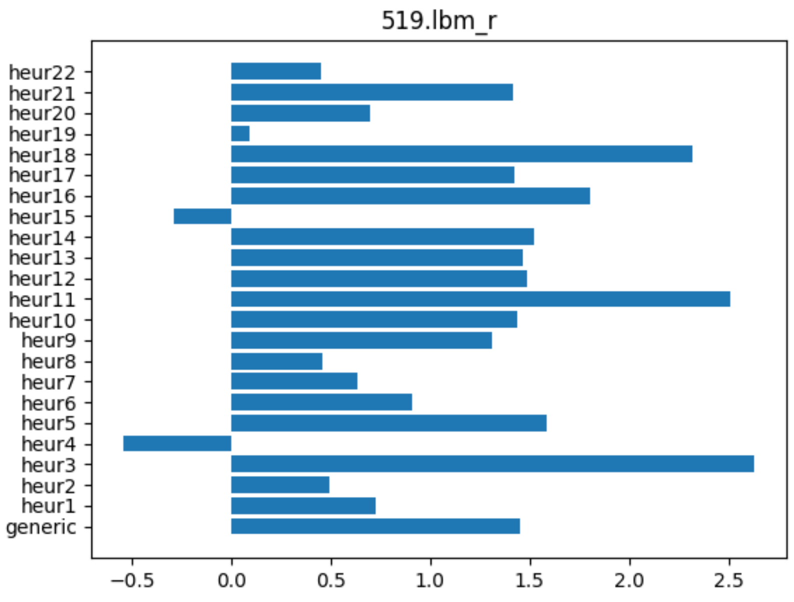


Figure D.12: 519.lbm_r benchmark.

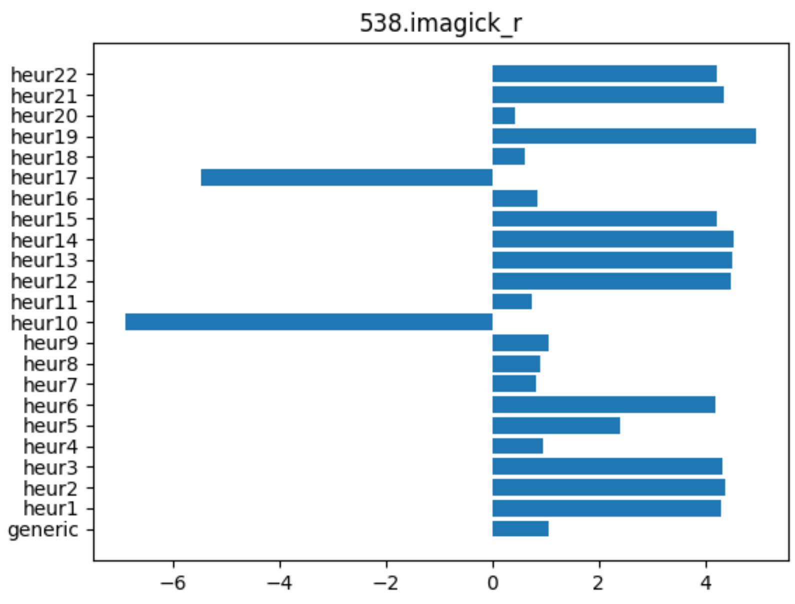


Figure D.13: 538.imagick_r benchmark.

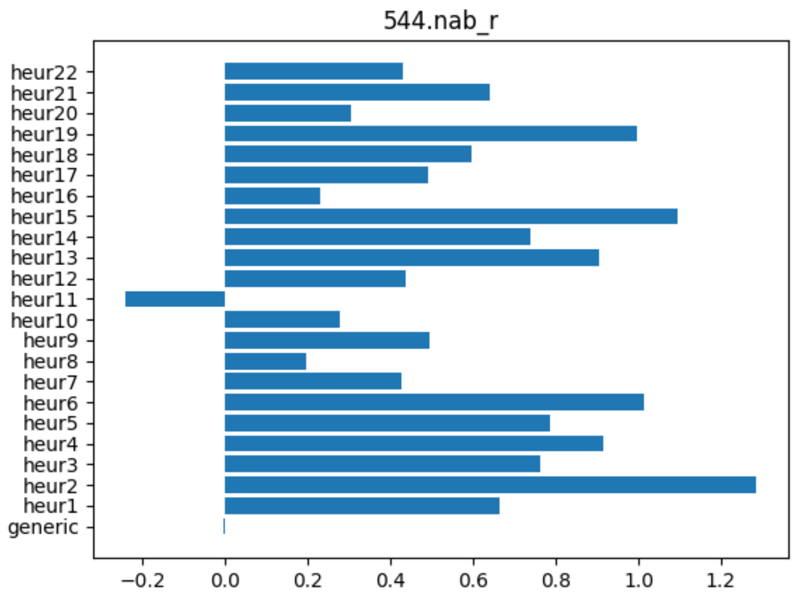


Figure D.14: 544.nab_r benchmark.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2020-11-20

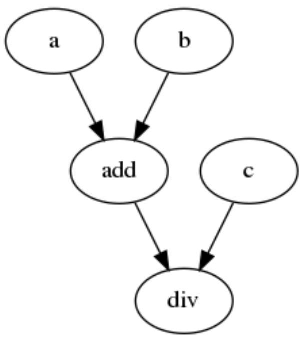
EXAMENSARBETE A Comparison of List Scheduling Heuristics in LLVM Targeting POWER8.
STUDENT Erik Samuelsson
HANDLEDARE Jonas Skeppstedt (LTH)
EXAMINATOR Flavius Gruian (LTH)

List Scheduling: Hur påverkas prestandan av de heuristik vi väljer?

POPULÄRVETENSKAPLIG SAMMANFATTNING **Erik Samuelsson**

List scheduling är en simpel, men effektiv, algoritm för schemaläggning. Som algoritm betraktad har den många användningsområden, ett av dessa är som schemaläggare av instruktioner, vilket i sin tur är ett viktigt optimeringstillfälle för kompilatorer.

En kompilators uppgift är att översätta från ett programmeringsspråk till ett annat. Den absolut viktigaste egenskapen som en kompilator besitter är onekligen att översättningen är korrekt. Är översättningen korrekt kan vi dessutom fundera på hur vi skulle kunna göra översättningen mer effektiv, mätt till exempel i exekveringstid. En kompilator är vanligtvis indelad i tre delar, *front-end*, *middle section* och *back-end*, som var för sig arbetar med en del av översättningen. Schemaläggning av instruktioner är en process som tillhör *back-end*. Det är ett viktigt optimeringstillfälle för att se till att datorns hårdvara utnyttjas på ett så bra sätt som möjligt. Den grundläggande algoritmen för att schemalägga instruktioner är *list scheduling*, den består av två delar. Först konstrueras en riktad acyklisk graf i vilken noderna representerar instruktioner, och bågar mellan noder representerar ett beroende mellan två instruktioner. Sedan schemaläggas instruktionerna genom en parvis jämförelse. De instruktioner som vid ett givet tillfälle kan schemaläggas utgörs av den mängd noder som saknar inkommande bågar. När en nod schemalagts tas den bort ur grafen. Figur 1 visar ett exempel på en graf tillhörande uttrycket `(a + b) / c`. Vi ser t.ex. att additionsinstruktionen inte kan schemaläggas före det att variablerna a och b har flyttats till varsitt register. Den intressanta delen av algoritmen, vilket också har varit fokus i detta arbete, är att bestämma vilken av två instruktioner som ska schemaläggas härnäst. Till detta utnyttjar man heuristik, för att på så sätt välja vilken instruktion man ska schemalägga.



Figur 1: Riktad acyklisk graf för: (a + b) / c.

Vid evaluering genomfördes tester både på enskilda heuristik och kombinationer av dessa. Som en del av arbetet utvecklades en ny heuristik, denna, samt resultatet av testerna kan tjäna som grund för eventuell vidareutveckling av heuristik eller av personer som är intresserad av denna typen av jämförelser. T.ex. som argumentation för eller emot en viss typ av heuristik.