Institut für Programmstrukturen
und Datenorganisation (IPD)

Lehrstuhl Prof. Dr.-Ing. Snelting

Befehlsanordnung auf expliziten Abhängigkeitsgraphen

Masterarbeit von

Steffen Kromm

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuende Mitarbeiter:	M. Sc. Andreas Fried
Abgabedatum:	23. April 2019

Zusammenfassung

Die Befehlesanordnung hat die Aufgabe, die durch Abhängigkeiten teilgeordneten Befehle eines Abhängigkeitsgraphen in eine total geordnete Liste zu überführen. Die konkrete Ordnung beeinflusst dabei die Laufzeit der resultierenden Programme.
Diese Arbeit implementiert ein verbessertes Befehlsanordnungsverfahren im libFIRM Compiler.
Es werden verschiedene Optimierungsmöglichkeiten der Befehlsanordnung betrachtet und geeignete Optimierungen implementiert.
Die Evaluation der Implementierung zeigt, dass Befehlsanordnung auf Grundblöcken auf moderner Hardware nur ein geringes Optimierungspotenzial besitzt und aufwendigere Verfahren in Zukunft benötigt werden.

Instruction scheduling defines a total order on the partial order of explicit dependency graphs. Additionally, it orders the instruction in a way that optimizes the runtime of the resulting programs.
This thesis implements an improved instruction scheduler in the libFIRM compiler infrastructure.
We inspect different optimization strategies for instruction scheduling and implement suitable new optimizations in the libFIRM library.
The evaluation of implemented optimizations shows that instruction scheduling on basic block level on modern processors has only a low optimization potential and more advanced approaches are needed to achieve significant improvements.

Title image source
`https://www.flickr.com/photos/157089461@N07/27559983948`

Contents

1. Introduction

A *compiler* is a translator program that transforms source code written by a programmer in a high-level programming language into assembly code that a processor can execute.
Since high-level programming languages are designed to abstract from hardware details and be understandable for humans a compiler needs to perform optimizations in the translation process in order to create a program that is efficiently executable on a target processor.
In a program, the *instruction order* defined initially by the programmer is not always necessary for the correctness of the program. A processor can execute every instruction at any point in time, where the values that the instruction needs are available, meaning the instructions which calculate these values are executed. Any instruction order that fulfills these *instruction dependencies* is valid. In the compilation process, the original code is often converted to different representation forms, like *explicit dependency graphs*, which lose the initial total order and define the program only with the partial order of the dependencies between the instructions. The *instruction scheduler* defines a total order on the instructions of the resulting program.
In early processors, only one instruction was executed at a time. On these machines, the order of instructions did not influence the execution time of programs.
Modern processors can execute multiple instructions concurrently by exploiting the *instruction level parallelism(ILP)* of the code [1]. ILP is a term used to describe the amount of independent and therefore parallel executable instructions in the code. We use the term potential ILP to describe the maximal possible amount of instructions executed in parallel in the execution of a program. In contrast, the plain term ILP of a piece of code is used to describe the actual amount of ILP in the current order of the instructions.
Due to limited resources of the processor hardware and the dependencies between instructions, the order of instructions changes the overall execution time of a program in these modern processors.
The performance difference between the default instruction order and an optimized order can be significant [2]. Because of this, compilers implement an optimization step, the so-called instruction scheduler. This step analyses the program and reorders instructions to maximize performance.
List scheduling is a simple algorithm to solve the instruction scheduling problem. A variant of list scheduling is implemented to facilitate the instruction scheduling in the libFIRM compiler infrastructure.

This thesis analyzes different optimization strategies to instruction scheduling and especially the list scheduling approach and tests the boundaries of list scheduling algorithm. We integrate multiple variants of an adjusted selection heuristic of the instruction scheduler in the libFIRM compiler infrastructure. These adjustments are evaluated to gain new insights into the implementation of list scheduling on explicit dependency graphs and the limitations of this approach.

2. Foundations

The following chapter describes the foundations required for the understanding of this thesis.

2.1. Compiler design

A *compiler* is, in general, any piece of software that translates source code of one programming language into an equivalent code of another language. In this thesis, the examined compilers are compilers that translate source code in a high-level programming language like C into assembly code for a particular processor.
Modern compilers have a frontend-backend architecture, which uses an *intermediate representation* to abstract from different source and target languages. The *frontend* is source language dependent, while the *backend* is dependent on the target language, or in the special case of this thesis, depending on the assembly instruction set of the target processor architecture:

Intermediate representation An intermediate representation is a language which is used internally by a compiler instead of the source language. Through this intermediate representation, the compiler performs optimizations more efficiently ([3]).
Since any source language is converted into the intermediate representation and since the intermediate representation can be converted to any target language, optimizations are language-independent.

Language frontend The language frontend is the part of a compiler that parses a specific source language program and translates it into an intermediate representation program. It performs no additional optimizations.

Middle end The intermediate representation is used to make optimizations easier. The *middle end* is the part of the compilation pipeline that performs optimization on intermediate representation programs without any target platform specifics.

Architecture backend The backend takes an intermediate representation program, generates assembly code for a specific processor architecture and outputs a binary program.

Intermediate representations do not only decouple frontend and backend; they also assist in the optimization process.

2.2. SSA form

The *SSA form*[1] is a property of program code[4]. A piece of code is in SSA form if every variable is defined before it is used and has only one definition. If an assignment statement defines a variable that a previous statement has already defined, a new variable is declared instead, which is defined with the new value (see section 2.2). In the context of SSA, we do not speak of variables but only *values* which are used.

```
int a = 1;
a = a + 3;
```

Listing 2.1: Sample code without SSA form

```
int a1 = 1;
int a2 = a1 + 3;
```

Listing 2.2: Sample code converted to SSA form

A problem with SSA conversion is the case of a value varying depending on the control flow. For example, depending on which branch of an if clause is executed, the actual value of x_3 may be x_2 or x_1. Having two assignments to x_3 in each branch breaks the SSA form.
A function often referred to as Φ function solves this problem. We define a Φ function with $\Phi(x_1, x_2)$ as the value of x_3. Depending on the actual control flow, the function returns a different value. This way, the SSA form of a program can be conserved. The most significant benefit of SSA form is that it makes the data flow of the program explicit because every declared variable has a constant value after the initial definition. That way, many optimizations become implicit or easy to handle, like eliminating unused variables and code. If no statement uses a value, then the value is not used and does not need to be stored (the statement of the assignment still has to be executed if it contains side effects). Significant optimizations that are improved by using SSA form include ([5]):

- Constant propagation
- Value range propagation

[1]Single static assignment

- Sparse conditional constant propagation
- Dead code elimination
- Global value numbering
- Partial redundancy elimination
- Strength reduction
- Register allocation

As stated above, the SSA form helps with many optimization steps in the compilation process. Therefore many modern intermediate representations are designed to only be able to represent programs in SSA form. This restriction can be realized by representing the program in a graph data structure. One of these modern intermediate representations is FIRM.

2.3. FIRM

FIRM is a fully graph-based, SSA-based intermediate representation developed as part of the compiler research of the IPD at KIT ([6]).

2.3.1. Explicit dependency graphs

FIRM uses a *directed acyclic graph* (DAG) to represent the instructions of basic blocks and their dependencies. Due to this fact, the graph is called *explicit dependency graphs*. A *basic block* is in this case a piece of code that contains no jump instructions except at the end and contains no jump targets. Therefore, if a basic block is executed,

```c
int main(int argc, char** argv) {
  int a = (int) argv[2];
  int b = (int) argv[3];
  return a * (a + b);
}
```

Listing 2.3: Sample c program for which the corresponding FIRM graph was generated.

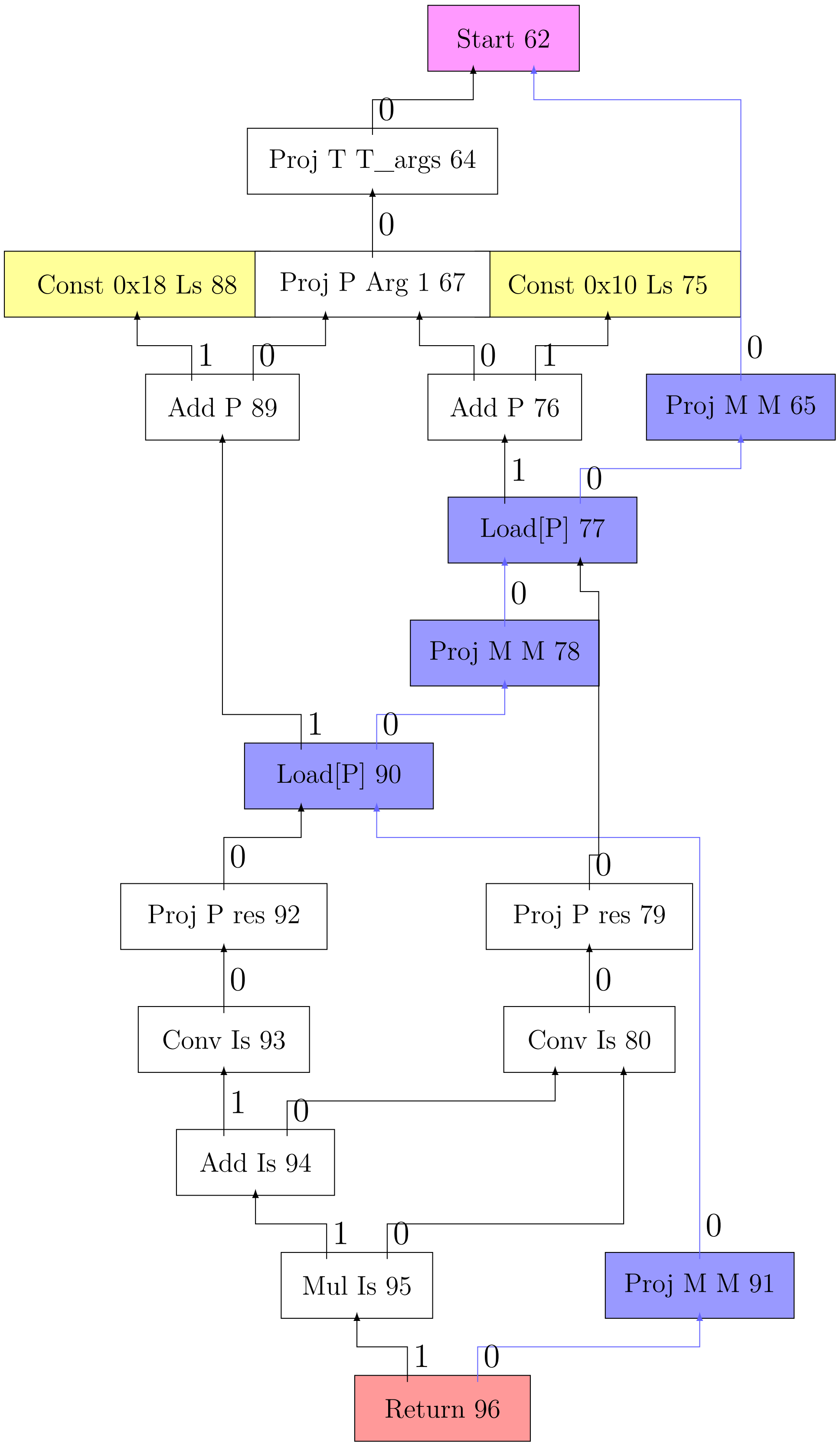

Figure 2.1.: FIRM graph of 2.3

all of its contained instruction are guaranteed to be executed. The program, on the other hand, is represented by a graph data structure with basic blocks as nodes, but this graph can contain loops. A DAG is a graph that has directed edges and does not contain cycles. Inside the basic block DAG, every node abstracts an instruction; every edge is used to represent a dependency. A directed edge that points from node A to node B means B can only start execution if A has already been executed.
Through the use of a DAG, the dependencies between instructions become explicit. Additionally, the existing FIRM nodes in combination with the DAG data structure only allow SSA form programs.

2.4. LibFIRM compiler design

LibFIRM is a compiler infrastructure that is being developed by the IPD at KIT. It consists of the core library libFIRM, which implements the SSA-based intermediate representation language FIRM, optimizations on FIRM graphs and the assembly generation for the FIRM DAG for different architectures, especially the x86 architecture.
LibFIRM is suitable for every programming language. All that is needed in order to create a compiler is a frontend that generates a FIRM graph for a given source code programming language. The most mature libFIRM frontend is cparser, which implements a frontend for the C programming language[2].
The most important steps of the code generation of the compilation process in libFIRM are the following, executed in the given order:

Instruction selection The *instruction selection* maps FIRM instructions onto the instructions of the target language. This step largely influences the following steps in the compilation because the mapping of source to target instruction is not bijective and sets the limits of available instruction level parallelism and register usage.
In the context of this thesis, the instructions are exact assembly machine instructions, which means the steps following the instruction selection must handle different instruction sets of processor architectures.

Instruction scheduling FIRM uses a explicit dependency graph to represents the instructions of a program, which defines no total order on the instructions. The instructions need to be in a total ordered list so that a processor can execute the program. The instruction scheduling performs this ordering of the instructions. It also rearranges instructions, if possible, to increase ILP and reduce register usage, to optimize the execution time of the program. If the processor executes more instructions in parallel, it also needs to store an

[2]C99

increased amount of values. This increases the amount of needed registers. If more register are needed than the hardware has available, some values need to be stored and loaded from memory instead of registers, which induces a performance penalty [7].

Register allocation The *register allocation* is the last major step of the code generation and is done after the instruction scheduling. At this point, instructions are actual assembly instructions of a real architecture (for example x86). Only the register names in the instructions are still virtual registers. The register allocation now assigns actual hardware register names to every instruction and inserts spill code if the program execution needs more registers than the hardware has available. After this final step, a processor that implements the target architecture can execute the generated assembly code.

Besides these steps, libFIRM performs many other optimization steps. The steps described above are the most relevant steps for instruction scheduling.

2.5. Instruction scheduling

Instruction scheduling refers to any optimization step in a compiler that defines a total order on the partially ordered instructions of a program while maximizing resource usage and that way optimizing the execution time of the instructions. The algorithm that defines the instruction order is called an instruction scheduler.
Modern processors implement a variety of techniques to speed up the execution of programs. The majority of these techniques exploit the ILP of the program.
The first major technique is instruction pipelining. The execution of an instruction is split into multiple steps, with each step working parallel to the others. Every one of the steps of the pipeline contains a different instruction. With this approach, the execution of instructions overlaps, and if there are n steps, then there are at most n instruction concurrently in execution at the same time.
This important, fundamental optimization technique of modern processors introduced also new dependency conflicts between instructions. All instructions that are in execution access the same hardware registers, which can lead to conflicts due to the limited number of registers.
What's more, is that data dependencies can occur if multiple instruction access the same register to access the same value.
The register pressure is the number of values a processor needs to store in registers for the currently executed instructions of an instruction pipeline. If the loaded instruction needs more registers than are physical register available, the processor must write the value of another register into the cache hierarchy or even main memory. These load and store instructions, called *register spills* are very costly due

to the latency of the memory access so that even L1 cache access can largely exceed the average latency of integer or boolean arithmetic operations, which induces a substantial performance penalty [7].
The goal of instruction scheduling, therefore, is to define a total order for the execution of instructions of a program which also has optimal performance.
With the advent of *superscalar out-of-order* and VLIW processors, which have multiple execution units and allow the concurrent execution of multiple instructions, instruction scheduling also needs to consider the ILP of the scheduled instruction order. This way, as many execution units as possible are used, improving the overall runtime due to higher ILP.
VLIW processors store multiple instruction in one super-instruction and therefore depend totally on software scheduling to create these super-instructions from the code during scheduling.
Out-of-order processors on the other hand fetch multiple instructions, the concrete amount defined by a processor specific instruction window, at every cycle. These instructions are then passed through the processor pipeline and later submitted to their corresponding execution unit. This submission is done by a hardware scheduler. This hardware scheduler can only operate on the instructions, that are fetched within the instruction window. Therefore a software scheduler can still optimize the code by scheduling on scopes higher than the instruction window size.
An instruction scheduler Ω creates an optimal schedule for a DAG G if it minimizes the following term:

$$\sum_{i=0}^{n} latency_{\Omega_G(i)} * ILP_{\Omega_G(i)} + \#spills_{\Omega_G(i)} \tag{2.1}$$

where n is the number of scheduled instructions and $\Omega_G(i)$ is the instruction, the scheduler Ω selects at the ith position in the total order from G. ILP is a factor in the interval $[0, 1]$ defined for each instruction and is used to describe the reduction of the total sum of latencies in the scheduled order defined by Ω due to parallel execution. The number of spills defines the overhead by spilling and additional memory operations that is caused by scheduling an instruction at position i.

2.6. Scheduling scope

Instruction scheduling can perform on different scopes. One is called local instruction scheduling, or basic block scheduling, because the scheduling is only done within the basic blocks of the program.
In contrast to basic block scheduling, there are also global scheduling methods. In these, more instructions than one basic block are scheduled at once. This way, the scheduler can move instructions from one basic block to another and increase the ILP

of the code further. A basic block itself is not guaranteed to be executed. Therefore the scheduled instructions are not guaranteed to be executed anymore, and the scheduler may then need to copy code if it moves an instruction outside of its original basic block. Therefore, global scheduling methods have a higher complexity.

.

2.7. Scheduling approaches

Instruction scheduling on explicit dependency graphs is the problem of converting a DAG datastructure into a total ordered list of the nodes of the DAG while maintaining the partial order defined by the edges (instruction dependencies) of the DAG.

2.7.1. List scheduling

Instruction scheduling as a special case of general scheduling is an NP-complete problem [8]. Therefore heuristic algorithms are used instead of analytical solutions. A simple heuristic algorithm for instruction scheduling is list scheduling. *List scheduling* refers to any algorithm that collects all *ready* instructions in a priority list and then selects the instruction with the highest priority from the list. The algorithm can use any heuristic to calculate the priority for each instruction.
An instruction is ready if all of its incoming instructions have been scheduled, meaning all its needed values have been scheduled.

```
Data: DAG, candidates = []
Result: schedule = []
candidates = DAG.root
while candidates not empty do
    instruction = select(candidates)
    schedule.push(instruction)
    candidates.remove(instruction)
    for successor in instruction.successors do
        if successor is ready then
            candidates.add(succesor)
        else
    end
end
```

Algorithm 1: List scheduling

List scheduling is the base for the instruction scheduler in the GCC[9] and LLVM[10] compilers.

2.7.2. Constraint programming

Instead of selecting instructions from a list based on some heuristic, which is the approach of list scheduling, another algorithm used in instruction schedulers is through constraint programming. With this, multiple constraints have been defined that limit the space of possible schedules. New constraints are added until only one valid schedule is available. In contrast to list scheduling, constraint programming schedules globally on the whole DAG instead of only inspecting the ready list at each scheduling step ([11]).

2.7.3. LibFIRM compiler instruction scheduling

LibFIRM performs the instruction scheduling with list scheduling before the register allocation on the basic block level. It uses only register pressure as the list selection priority, which means it schedule to minimize register pressure independent of the actual register usage and the potential ILP.

3. Related work

3.1. LibFIRM

This thesis is a part of the ongoing development and research effort of the IPD on libFIRM. Therefore it elaborates and builds on previous works on the libFIRM compiler infrastructure.

3.1.1. Register-pressure-aware instruction scheduling

Foremost, this thesis enhances the existing instruction scheduler of libFIRM. The used instruction scheduler in libFIRM is the so-called register pressure-aware instruction scheduler developed as part of the diploma thesis of Christoph Mallon [12].
It builds on top of the list scheduling framework developed by Sebastian Hack et al.[13]. The scheduler calculates register costs for each instruction at each point they could be scheduled and then uses this number as a selection heuristic. This way, it implements minimal register usage.

3.2. Instruction scheduling

3.2.1. Scheduling heuristics

The first research on actual instruction scheduling as defined in this thesis was done as part of research on vector machines [14]. With the advent of pipeline architectures, instruction scheduling became useful for scalar processors.
Gibbons et al. did early work on pipeline architecture optimizations and especially instruction scheduling on basic block level. They used a priority list of heuristics calculated for every instruction, which they described in [8].
Goodman et al. ([15]) developed a novel instruction scheduling technique which

combines register allocation and scheduling in one step and trying to solve the dependency problem between register allocation and instruction scheduling.

3.2.2. Global scheduling

The majority of research investigates how global level instruction scheduling can improve performance.
Havanki [16] introduced Treegion scheduling. This scheduling approach defines a treegion over the control flow graph, the graph structure that consists of basic blocks. A treegion is a tree graph of basic blocks. For a basic block, the corresponding treegion consists of all the following basic blocks that only have a single predecessor basic block.

3.3. Related research fields

In the following relevant papers on the research on compiler design and instruction scheduling are presented which are related to the content of this thesis but do not exactly match the goal of this thesis or are misleadingly confused with instruction scheduling as understood in the context of this thesis.

3.3.1. Modulo scheduling

An extensive amount of research inspects modulo scheduling and how it can be implemented effectively. Modulo scheduling is a loop unrolling technique and a special case of instruction scheduling which is not part of this thesis. Modulo scheduling is a preprocessing step for actual instruction scheduling, as it increases the size of basic blocks.
An overview of modulo scheduling is given in [17].

3.3.2. Microcode compaction

Joseph A. Fisher presented in [18] trace scheduling as a microcode compaction technique. Microcode compaction and the developed techniques represent the predecessor work to later work on instruction scheduling. [19] adopted microcode compaction algorithms for instruction scheduling on superscalar machines.

4. Design and implementation

This chapter describes the design decisions that were made to implement different instruction scheduling variants, which are subject to evaluation in this thesis.
The design and implementation process for optimizing the instruction scheduler of libFIRM was lead by the following questions:

- What optimization potential leaves the register pressure-aware scheduler?
- Which metrics can additionally be used?
- How do the metrics influence the performance?
- What optimization potential has basic block scheduling?

4.1. Requirements and limitations

A new solution has to fit in the libFIRM environment which implies a few requirements and restrictions to the solution.
The libFIRM library is microprocessor agnostic, so it has to work with abstractions over all existing architectures.
It does not handle specific microprocessor architecture implementations, like the different Intel and AMD x86 generation architectures. This makes sense since most software is distributed as binaries. As these binaries are precompiled for an architecture like x86, the actual architecture is unknown due to the unknown target hardware. To handle explicit architecture implementations or variants, changes to the overall architecture are required.
In the compilation process of libFIRM, the instruction selection is made before instruction scheduling. That means that the instruction scheduler does not handle a DAG of libFIRM node types, but every node represents an actual architecture instruction.
We implement the instruction scheduler only for the ia32 instruction set, though the steps to implement this logic can be easily replicated for other architectures.

4.2. Simplifications

Besides the imposed restrictions, we make some simplification to reduce the optimization space.
Firstly optimizations and benchmarks are implemented with a microprocessor with the Intel Skylake architecture. Architecture-specific implementations also were developed for the Skylake architecture only. Though this is a restriction, the implementation and techniques in this thesis are meant to apply to any particular microprocessor architecture. Restrictions that were imposed by the x86 architecture or the Skylake implementation are mentioned inside the following chapter.
Our solution targets superscalar out-of-order processors. We do not consider VLIW processors in our solution.

4.3. Coarse design

First, we research what possibilities are available to improve the existing instruction scheduler of libFIRM.

4.3.1. Register usage and ILP tradeoff

In libFIRM, instruction scheduling is done before register allocation. This is a relevant piece of information, as there is a significant dependency between instruction scheduling and register allocation that affects the overall performance of the program. Register allocation is the process of assigning actual hardware register names to variables and thereby define which variable is stored in which actual registers. If there are more variables active at one point in the execution than hardware register available than spilling code has to be inserted: Values have to be stored and accessed in memory via load and store operations. This memory access overhead creates a performance penalty.
The instruction scheduler rearranges instructions to maximize performance. As objective 2.1 shows, this can be achieved by either improving the ILP of the program or by reducing the register spills. By rearranging instructions for higher ILP, the lifetime of values used by these instructions is modified, which then can affect the amount of additional memory accesses that the register allocation has to add. So scheduling for higher ILP tends to increase register spills and scheduling for a minimal amount of register spills ignores the ILP of the program. So the two terms of 2.1 are often negatively correlated. A common belief is that minimizing register spills tends

to outperform higher ILP.

4.3.2. Base Algorithm

LibFIRM implements instruction scheduling with a list scheduling framework on the basic block level that is based on the thesis of Christoph Mallon [12].
We decided to investigate further the optimization potential of this instruction scheduler instead of implementing alternative approaches to list scheduling. This allowed us to concentrate on the optimization potential of list scheduling in more detail. As presented in the foundations, list scheduling selects instructions from a ready list based on a priority.
The priority selection is defined by various heuristics, which in turn are ordered in a list. If two instructions have the same heuristic value, then the next heuristic in the list is selected to resolve the tie. The last heuristic is the unique (and arbitrary) instruction index of the FIRM graph node, which guarantees a total order over the DAG and that the scheduler is deterministic.

4.4. List scheduling

4.5. Priority heuristics

A list scheduler selects ready instruction based on priority defined for every instruction at the selection time (the priority of an instruction can change depending on the previous scheduling decisions). There are two ways of how a list scheduler can perform better results. The first way is to increase the amount of instruction on which the scheduling is performed. The amount of instruction that the instruction scheduler can choose at each scheduling point is restricted by the logical dependencies of the program. This cannot be changed. Besides these dependencies that limit the scope of available instructions at each scheduling timepoint, there are a few possibilities, to maximize the scheduling scope:

Loop unrolling Loop unrolling is a compiler optimization technique, that "unrolls" the instructions of multiple iterations of a loop and puts the instructions into one single iteration. This is possible due to the partial independence of loop iterations. This way, the number of mostly independent instructions of basic blocks can be dramatically increased.

Instruction selection The instruction selection is the process of selecting what actual hardware instructions are used instead of the virtual instructions.

Global scheduling Scheduling on basic blocks simplifies scheduling because within a basic block all instructions are guaranteed to be executed. This simplification is a tradeoff with the mostly small scope of basic blocks. While the above techniques, especially loop unrolling, can increase the size of basic blocks, global scheduling techniques allow to schedule over multiple basic blocks or the whole program and therefore increase the scheduling scope immensely.

None of these possibilities were viable in the scope of this thesis. Loop unrolling and instruction selection are different optimizations in the compilation process and do only concern instruction scheduling as a supporting optimization.
Global scheduling requires too much additional effort. It was not further investigated, as this thesis explores the potential and limitations of basic block scheduling, and implementing even a simple global scheduler would require the effort of an additional thesis.
As the especially possibility to improve a list scheduler, increasing the scope is either out of the scope of the context of this thesis or not investigated due to time restrictions, the second possibility is to improve the selection heuristic of the scheduler.
In the current implementation of libFIRM, at every scheduling timepoint, the instruction with the lowest increase in register pressure is scheduled. The results of the thesis by Mallon show that this implementation partly performed worse than the previous implementation.
As stated before, it is not clear if scheduling for higher ILP yields better performance than minimizing register pressure. Even if scheduling for minimal register pressure is optimal, there are cases where two or more instructions have the same amount of register pressure increase. In these cases, the current implementation decides by the original (arbitrary) index that is assigned to every node. At least on these degrees of freedom to the total order that register pressure metric leaves scheduling for higher ILP could increase the overall performance.
To test these assumptions, we need to be able to schedule for potential ILP. An instruction scheduler that optimizes instruction level parallelism selects available instructions, so they occupy as many execution units as possible and that way execute the maximal possible amount of instructions in parallel.
An optimal solution would need the concrete amount of different execution units that are available, to schedule accordingly.
In reality, applications are normally compiled against an abstract architecture like ia32 and then distributed. Therefore, the compilation cannot depend on the concrete number of execution units that are available, because they vary between processors of the same architecture family. Also, which instructions share the same execution units is unknown for an abstract architecture like ia32.
Therefore all information to actual schedule directly for high ILP is not available.

To understand how ILP can be optimized we examine objective 2.1. As processor hardware increases capabilities, we do a theoretic experiment and inspect a system with unlimited resources. There are no register spills, so the sum of spills equals 0, which leads to:

$$\sum_{i=0}^{n} latency_{\Omega_G(i)} * ILP_{\Omega_G(i)}$$

Because this ideal system has unlimited resources, it can execute an unlimited amount of instructions in parallel. Therefore the ILP factors are either 1 for instructions on the critical path, or 0 because all other instructions run in parallel and do not add to the total latency of the program.
The length or latency of a program path is the sum of the latencies of all instructions in the path. A path is a queue of instructions, where every instruction is directly depending on the previous, and the first instruction is a root and the last instruction a leaf node. The critical path is the path with the greatest length.
If the instructions on the critical path are scheduled first, an optimal schedule for this imaginary machine is created.
Now imagine a system, which does not have unlimited execution units, but still unlimited registers.
Scheduling for the critical path can lead to long dependency chains because the instructions on the critical path are scheduled until the dependencies of the critical path instructions require to execute other instructions. This could lead to an increased amount of register pressure.
Besides that, after scheduling an instruction on the critical path, the critical path of the remaining unscheduled graph can change. At every point in time, we want to schedule the instruction that lays on the current critical path and not the global critical path. Therefore we define the maximum path latency metric for an instruction as the length of the longest path that exists from the instruction to the root node of the graph. From all ready instruction, the instruction with the highest maximum path latency is the instruction on the current critical path. Even if we do not explicitly schedule for higher ILP with this metric, any selection metric that directly increases ILP can only deviate in the selection from the maximum path latency if two instructions have a tie on the maximum path latency because a faster schedule than the critical path is not possible.
At last, we need to consider register pressure. Depending on the program, either minimizing register pressure first or selecting maximum path latency yields better performance. What every metric is chosen first, the other metric can be used on the degrees of freedom the first one leaves (instruction with the same metric value).
To calculate the maximum path latency of instruction, the instruction latency of every instruction is needed.

4.5.1. Instruction Latency

The latency of an instruction defines how many cycles the processor needs to process an instruction, that is, the time from loading the instruction into an execution unit until the results of the instruction are available to other instructions.
For a particular microprocessor, we can define the correct latency for every instruction. In general, a compiler only compiles against an architecture, without knowledge about a concrete implementation. Because of this, the latencies of instructions need to be guessed, if the architecture does not guarantee them, which is not the case in general.
For scheduling, the latencies do not need to represent the accurate cycles, but rather, the ratio between the latency values of different instructions need to be correct. This way, it is easier to generalize over all implementations of architecture, while maintaining the ability to calculate. A good estimation of latencies should consider the correct values of different microprocessors of the same architecture and weight them based on the usage of the microprocessor. This approach requires constant maintenance due to new processors and their potentially deviating latencies.
For purposes of this thesis, we didn't calculate approximated latency values like described above but rather used latency values just for the Skylake architecture. The

instruction	old libFIRM	new libFIRM
add	1	1
mul	10	4
div	25	20
fadd	4	1
fmul	4	3
fdiv	20	40
shl	1	1
and	1	1

Figure 4.1.: Table shows the latency numbers of some basic x86/x87 instructions for the current libFIRM implementation and the Agner benchmarks results[7] for the Intel Skylake architecture used as the new latency values for libFIRM.

latency values already used in libFIRM are defined for the ia32 architecture. A good measurement for the latency values specific to the Intel Skylake architecture gives Agner [7].

Difficulties

The actual latency value of instruction is even for a particular architecture like the Intel Skylake architecture not certain. There are several problems:

Load instructions The latency of load instructions is unknown due to cache misses and hits. We do not handle the varying latency of load instructions in this thesis.

Register size The latency of instruction varies depending on the size of the registers. The latency values defined in libFIRM do not differentiate between register size. We could add a constant to the latency depending on the register size, but not all instruction varies with different register sizes. Therefore we used a median between 32 and 64-bit register latency, as these instructions on these register are far more common.

Implementation in libFIRM

The used latency value is an architecture (implementation) specific value, which is assigned to every instruction.
The file *ir/be/be_t.h* contains the definition of the instruction scheduling step of the libFIRM compilation through the function **void** be_step_schedule(ir_graph ∗irg). We modify the signature of this function by adding a structure to this function as the second parameter after the DAG graph structure irg so the signature is **void** be_step_schedule(ir_graph ∗irg, instrsched_if_t ∗instrschedif). The structure instrsched_if_t is an interface that contains function definitions that every architecture implements.
To implement the architecture specific latency value, we introduce the function **unsigned** get_instruction_latency(ir_node ∗irn) as a field of instrsched_if_t that is implemented to return the architecture specific instruction latencies.
Because this property is only defined for nodes that represent instructions of the target architecture we introduce also a callback function that returns a boolean that signals if the node represents a, in our example, x86 instruction or not. We only query the latency for actual x86 instruction, and use 0 as the latency value for other nodes. The latency itself is not too useful as a metric but is used to calculate the maximum path latency metric.

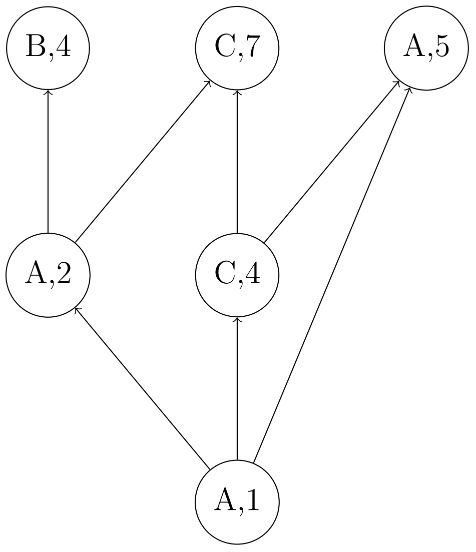

Figure 4.2.: Example graph for the maximum path latency. The graph consists of three fictive node types, A with latency 1, B with latency 2 and C with latency 3. Every graph node is labeled with the node type and its maximum path latency, separated by comma.

4.5.2. Maximum path latency

The maximum path latency metric is calculated by adding the latency of instruction and the highest maximum latency of all dependent instructions, as shown in algorithm 2. The resulting maximum path latency values are visualized by figure 4.2

```
Data: DAG, instruction I $\epsilon DAG$
Result: maximum path latency of I
if $I.dependent\_instructions.length == 0$ then
    return I.latency;
else
    return I.latency + max of I.dependent_instruction.total_path_latency
end
```

Algorithm 2: Calculate maximum path latency

Implementation in libFIRM

In order to implement maximum path latency as a heuristic in libFIRM, we use the recursive algorithm 2. The current implementation of the instruction scheduler step in libFIRM uses a struct flag_and_cost to store information for every node. We use this struct to store the calculated maximum path latency of the node. Every node is

initialized with a maximum path latency of UINT_MAX and is used as a marker for unvisited nodes. We iterate every node in the DAG and calculate their maximum path latency if their maximum path latency is UINT_MAX.
The maximum path latency calculation is implemented through the recursive function calculate_total_latency and a second function that calculates the maximum of the maximum path latencies of all successors of a given node.

```
static void calculate_total_latency(ir_node* irn)
{
  flag_and_cost *fc = get_irn_flag_and_cost(irn);
  if(!fc->no_root) {
    fc->minimal_path_latency = get_instruction_latency(irn);
  } else {
    unsigned max_succ_latency = get_max_latency_succ(irn);
    fc->minimal_path_latency =
      get_instruction_latency(irn) + max_succ_latency;
  }
}

static unsigned get_max_latency_succ(ir_node* irn)
{
  unsigned max = 0;
  foreach_irn_out(irn, index, succ) {
    flag_and_cost *fc_succ = get_irn_flag_and_cost(succ);
    if(fc_succ->minimal_path_latency == UINT_MAX)
      calculate_total_latency(succ);
    max = MAX(max, fc_succ->minimal_path_latency);
  }
  return max;
}
```

Listing 4.1: Simplified maximum path latency calculation

Listing 4.5.2 calculates the maximum path latency, but only in trivial cases. For most FIRM graphs, it will fail to calculate the correct maximum path latency or not terminate.

Handling special FIRM nodes

FIRM contains a few node types that are not translated to actual instructions, but rather are needed to realize the strict SSA-form of FIRM graphs. The nodes which need special case handling are:

Pin Pin the value of the node in the current block. No users of the Pin node can float above the Block of the Pin. The node cannot float behind this block. Often used to Pin the NoMem node.

Sync A sync node synchronizes multiple nodes that are independent of each other.

Proj A projection node extracts a value from a node that returns a tuple of values.

For these and a few other special nodes, the flag and cost struct does not exist. The flag and cost struct only exists for nodes that represent an actual instruction.
The calculation needs to be proxied to actual nodes that have flag and cost.

```
if (is_Pin(irn) || is_Sync(irn) || is_Proj(irn)) {
    foreach_irn_out(irn, index, succ) {
        if (succ == irn) continue;
        calculate_total_latency(succ);
    }
    return;
}
```

Handling graph corner cases

In actual FIRM graphs, circular dependencies can exist in the graph structure. To solve these, the maximum path latency of a node is set to 0, when it's calculation is started and then after the recursive calculation is stopped, set to its actual value. This is needed, so infinite recursions are stopped.
The complete implementation can be viewed in the appendix A.1.

4.5.3. Register pressure

With maximum path latency, we try to reduce the first ILP term of 2.1. We could also reduce the register spills to improve overall performance.
The existing instruction scheduler of libFIRM already implements a register pressure heuristic as the number of registers active, when an instruction is scheduled. This heuristic can be used to minimize register pressure.
Minimizing register pressure does not necessarily yield the best performances. Scheduling instructions for minimal register pressure, while the number of active registers is below the available number of registers is without effect, because no matter which instruction is scheduled, the memory access overhead does not change. In these cases, it makes sense to switch to maximum path latency as a scheduling metric to further increase the overall runtime performance.
The problem with this approach is missing information about what registers are used because the scheduling is performed before register allocation and therefore handles virtual register names. Therefore the scheduler schedules to reduce register pressure, even if no spills are needed.
A widespread assumption is that on modern processors business applications always have too many variables to avoid register spills, so there is always overhead, and therefore the existing register pressure metric performs optimally.

The register pressure metric is already implemented as part of the existing register pressure-aware instruction scheduler.

4.5.4. Dependencies

Maximum path latency and register pressure as metrics still can leave degrees of freedom in the schedule, where both metrics have a tie on two instructions. We look into an auxiliary metric than can be used to schedule smarter than just by the instruction index.
Every node of the program DAG represents an instruction. Also, the incoming and outgoing edges in the DAG abstract incoming and outgoing dependencies of the instruction. While scheduling an instruction, the incoming dependencies are not of interest, because all these instruction must have been scheduled already.
The outgoing dependencies, the instruction that is dependent on the complete execution of the instruction can be used in the scheduling process. A higher number of dependent instruction means that after the execution of the instruction potentially more instruction can be scheduled. If more instructions can be scheduled simultaneously, than a higher amount of potential ILP is available.
Because of this, using the number of dependent instructions can be a useful metric for instruction scheduling.

Implementation in libFIRM

The number of dependencies of instruction is equal to the number of incoming edges of a FIRM node. This number can directly be queried with the function get_irn_n_outs.

4.6. Implemented heuristic variants

We use a list of metrics to calculate the priority of a FIRM node. If two nodes have the same metric value for the first metric in the list, the second metric in the list is chosen to solve the tie and so forth.
To investigate the influence of ILP and register pressure we implement two variants, one who has to minimize register pressure as a key objective; the other one focuses on the potential ILP.

4.6.1. Variant A

. The first variant A uses lowest register pressure as its first metric and tries thereby to minimize the number of spills. On the remaining degrees of freedom, we use the highest maximum path latency to achieve a higher ILP.

1. lowest register pressure

2. highest maximum path latency
3. highest number of dependent instructions
4. index

4.6.2. Variant B

The maximum path latency schedules the instructions to maximize ILP. Ties are then schedules by register pressure, to minimize spills. Variant B reverses this order with:

1. highest maximum path latency
2. lowest register pressure
3. highest number of dependent instructions
4. index

In both cases, we use the highest number of dependent instructions as the third metric, since it only offers auxiliary function in the form of potentially increasing the number of ready instruction in the next scheduling step.

4.7. Alternative approaches

4.7.1. Machine learning

An alternative list scheduling and a static list of selection metrics we investigated in this thesis is the use of machine learning.
The idea was to train a machine learning algorithm, that outputs weights for every metric that can be used to calculate a total priority for every instruction. The algorithm would be able to recognize different code patterns and solve the scheduling better than a static list of metrics like the one used in this thesis.
We didn't choose this approach however, due to some concerns and potential problems:

Training data To create a useful prediction many training data is needed. This training data does not exist for a compiler. Because optimal scheduling depends theoretically even on particular hardware, this is an even bigger concern.

Benefit cost ratio Due to the previous concern, it seems not useful to pursue this approach, because the expected performance boost would not justify the development effort of such a solution. Global scheduling could potentially increase the gains to a degree, where it would be viable to develop. Such a solution was way beyond the scope of this thesis.

Nonetheless, the implemented static priority list of metrics of this thesis gives new insights and tests some common beliefs about the performance of scheduling metrics that were made under significantly different hardware environments.

5. Evaluation

In the following, we evaluate the different implemented variants of the instruction scheduler.
The benchmark system, which was used in the evaluation of this thesis is an Intel Core i7-6700 processor with 3.4 GHz, 4 physical cores with hyperthreading and 64 bytes instruction window.

5.1. Platform independence

The implementation is developed explicitly for the Intel Skylake architecture with its specific instruction latency values. At the moment libFIRM is not designed to handle specific processor architecture implementations, so the Skylake latencies are used in place of the overall x86 latency values. To support different processor architecture implementations, additional development effort is needed. To effectively support a new architecture, code scheme that was used to implement the latency values in the ia32 backend can easily be copied.

5.2. Performance

We use the SPEC2000INT benchmark suite[20] to compare the performance of the different variants of the selection heuristic used in the instruction scheduler. The existing register-pressure-aware scheduler is used as a base reference. Each test for each variant is sampled 50 times. We only measure the raw runtime.
We say a benchmark was significantly slower or faster than another reference if the difference of their means is at least two times the size of the biggest standard deviations of both benchmark results.
In the SPEC2000INT spec suite, variant a only showed a significant increase in performance in the test cases 175.vpr of about 0.5%. Besides that, variant also showed a significant slowdown of 1.7% in 300.twolf, 0.47% in 197.parser, 0.56% in 186.crafty and 0.45% in 176.gcc. The remaining tests showed a few improvements, all insignificant though. Overall variant a could be an improvement to the existing scheduler, but more tests are needed to validate this claim.
Variant b had similar results to variant a but showed significant improvements of 1.07% in 252.perlbmk and 0.84% in 255.vortex. It performed worse than the base with 1.66% in 300.twolf, 0.49% in 197.parser, 1.84% in 186.crafty and 0.73% in 176.gcc. It also was significantly slower in test 175.vpr than variant a.

Though variant b has greater performance gains, it also has greater slowdowns in the same test variant a showed significant slowdowns. So overall variant a seems to be preferable to variant b.
The deviations between the different variants can be explained by the different basic block sizes, register pressure and ILP of the different test programs of the SPEC2000INT test suite. So different test cases yield better results on different variants.

Test	**base**		**trivial**			**variant a**			**variant b**		
	runtime	σ	runtime	ratio	σ	runtime	ratio	σ	runtime	ratio	σ
164.gzip	59.51	0.06	59.90	0.66%	0.22	60.38	1.46%	0.41	59.92	0.69%	0.23
175.vpr	43.11	0.11	43.24	0.30%	0.10	42.90	-0.49%	0.09	43.14	0.07%	0.09
176.gcc	17.71	0.02	17.86	0.85%	0.02	17.79	0.45%	0.02	17.84	0.73%	0.02
181.mcf	20.73	0.11	20.60	-0.63%	0.11	20.72	-0.05%	0.11	20.74	0.05%	0.11
186.crafty	23.35	0.04	23.48	0.56%	0.03	23.48	0.56%	0.02	23.78	1.84%	0.02
197.parser	53.31	0.07	53.83	0.98%	0.06	53.56	0.47%	0.07	53.57	0.49%	0.09
253.perlbmk	43.73	0.02	40.75	-6.81%	0.03	43.62	-0.25%	0.03	43.26	-1.07%	0.03
254.gap	21.65	0.08	21.57	-0.37%	0.10	21.50	-0.69%	0.12	21.50	-0.69%	0.10
255.vortex	32.17	0.18	32.00	-0.53%	0.08	31.99	-0.56%	0.10	31.90	-0.84%	0.09
256.bzip2	42.89	0.12	43.50	1.42%	0.09	42.83	-0.14%	0.09	43.07	0.42%	0.12
300.twolf	59.03	0.23	59.19	0.27%	0.19	60.04	1.71%	0.21	60.01	1.66%	0.24

Table 5.1.: Table shows the base register pressure-aware scheduler, the trivial and the implemented instruction schedulers in variant a and variant b. The ratio describe the runtime change against the base scheduler. σ is the standard deviation of the corresponding mean with a sample size of 50. The standard deviation and runtime are measured in seconds.

Test	**base**		**trivial**			**maximum path latency**		
	runtime	σ	runtime	ratio	σ	runtime	ratio	σ
164.gzip	59.51	0.06	59.90	0.66%	0.22	59.95	0.74%	0.33
175.vpr	43.11	0.11	43.24	0.30%	0.10	42.98	-0.30%	0.08
176.gcc	17.71	0.02	17.86	0.85%	0.02	17.97	1.47%	0.03
181.mcf	20.73	0.11	20.60	-0.63%	0.11	20.85	0.58%	0.11
186.crafty	23.35	0.04	23.48	0.56%	0.03	23.59	1.03%	0.03
197.parser	53.31	0.07	53.83	0.98%	0.06	53.67	0.68%	0.08
253.perlbmk	43.73	0.02	40.75	-6.81%	0.03	43.86	0.30%	0.08
254.gap	21.65	0.08	21.57	-0.37%	0.10	22.20	2.54%	0.13
255.vortex	32.17	0.18	32.00	-0.53%	0.08	31.81	-1.12%	0.07
256.bzip2	42.89	0.12	43.50	1.42%	0.09	43.43	1.26%	0.11
300.twolf	59.03	0.23	59.19	0.27%	0.19	59.88	1.44%	0.20

Table 5.2.: Table shows the base register pressure-aware scheduler, the trivial and variant an instruction scheduler that only selects by highest maximum path latency. The ratio describes the runtime change against the base instruction scheduler. σ is the standard deviation of the corresponding mean with a sample size of 50. The standard deviation and runtime are measured in seconds.

5.3. Optimization potential of basic blocks

The benchmark processor is a modern out-of-order superscalar processor with a 64-byte instruction window. This means that 64 bytes of instruction are fetched with each cycle.
Due to the only slight changes in runtime in both variants a and b compared to the base reference, we suspected, that the effect of the instruction scheduler is only very limited on this processor. To further investigate this, a trivial instruction scheduler was also implemented and benchmarked, that selects instructions just by their unique index.
As the table shows, this trivial instruction scheduler performs in between the results of the base reference and the two variants.
Interestingly, the trivial scheduler performs slightly worse than variant a and b, but performs better in the test cases, where variant a and b performed better than the base reference.
Seeing the improvement of any of the different implementations, we can see, that basic block scheduling has little to no optimization potential on the target benchmark system.
We do not expect, that a more sophisticated way of calculation the priority of instruction, either with a machine learning approach or through constraint programming or another special case handling, introduces an improvement, that is exceedingly higher and justifies the implementation effort.

5.4. Effects of basic block size

We expect that the seen results depend and are caused by the size of basic blocks in the current state of the compiler.
The target system is an out-of-order processor, so on the instructions the processor fetches each cycle it automatically optimizes ILP by assigning ready instruction to available execution units.
In the previous chapter 4, we inspected the possibilities, to increase the scope of instruction scheduling. If the average size of basic blocks is below the instruction window size of the processor, optimizing for higher ILP is not worthwhile or even useless, because the hardware scheduler already optimizes on these instructions. This explains why the trivial instruction scheduler performed similarly to variant b, which used maximum path latency as a first selection heuristic. While trivial instruction scheduling performs no optimization, the optimization of maximum path latency is rendered nearly useless due to the out-of-order execution of the hardware.
If we inspect the minimization objective 2.1 of instruction scheduling, we can see that for an out-of-order machine with a large instruction window, the ILP factors $ILP_{\Omega_G(i)}$ of the latencies of all instruction are defined by the out-of-order hardware scheduler due to the automatic ILP optimization, rather than the software scheduler. So optimizing the instruction scheduling for higher ILP is rendered useless for most

basic blocks.
The only way for the software scheduler to optimize the runtime, in this case, is to reduce the register spill term of the sum. That explains, why the base reference and variant a perform the best.
The small improvements of maximum path latency scheduler in the tests 175.vpr and 255.vortex against the trivial scheduler could be explained by some basic blocks, which do indeed profit from higher ILP by software scheduling due to their size. Using maximum path latency on the remaining degrees of freedom of register pressure can then yield a small additional improvement. On the other hand in the test cases where the trivial performed significantly better than the maximum path latency scheduler we expect that the reordering by the maximum path latency increases the number of register spills more which leads to additional overhead.

5.5. Performance of maximum path latency

The maximum path latency scheduler performed for most test cases slightly worse than the variant b 5.2. Using maximum path latency as a second metric (Variant a) performed better than using maximum path latency as the first selection heuristic. Variant b performs in a few cases better than the base reference.
In table 5.2 a new scheduler that only schedules by maximum path latency was evaluated. If the assumption, that modern out-of-order processor already optimize for higher ILP on the basic block level, than this scheduler should perform similar to the trivial scheduler. Like expected this holds true for the tests 164.gzip, 176.gcc, 186.crafty, 197.parser, 255.vortex and 256.bzip2.
However, the table shows also significant deviations between the trivial and maximum path latency scheduler in the test cases 175.vpr, 181.mcf and 253.perlbmk, 254.gap. Only in test 181.mcf, 253.perlbmk, and 254.gap was the maximum path scheduler significantly slower than the trivial scheduler. It is to be noticed that in these cases the trivial scheduler had a performance increase against the base reference while the maximum path latency scheduler had a performance decrease against the base. In the tests 175.vpr, 197.parser, 255.vortex and 256.bzip2 the maximum path latency was faster than the trivial scheduler. Again, this could be explained by a larger basic block size on average in these tests.

5.6. Exceptions

The test 252.perlbmk is an exception in all benchmarks. The trivial scheduler has a significant increase of up to 6.81% in performance against the base. The other variants and the maximum path latency scheduler are also faster than the base but only around 1%. For further explanations, we looked at the benchmark results of our predecessor thesis by Christoph Mallon. His evaluation of the register pressure aware scheduler, which is our base reference, showed a decreased performance for the

252.perlbmk test. Also, the number of permutations and copy operations showed a significant increase. Only the amount of spills was reduced by around 1%.
It seems like for this test case register pressure has a small influence, so the register pressure-aware scheduler has even a negative impact on the performance. Scheduling with maximum path latency also seems to have little effect, probably due to the effects of a small average basic block size inspected in the previous section.
We can only explain the huge performance boost of the trivial scheduler as a lucky coincidence. It seems like this test is the worst case which does not profit from register pressure reduction nor maximum path latency. The trivial scheduler creates the least overhead and yields, therefore, the best performance.

6. Conclusion and outlook

6.1. Conclusion

With this thesis, we investigated how instruction scheduling can be optimized by improving the instruction scheduler of libFIRM. We focused our effort on the selection strategy of instructions of the instruction scheduler by using the existing list scheduling framework of libFIRM and basic block scoped scheduling. That way we evaluated the remaining optimization potential of this given framework.
We implemented the maximum path instruction latency as a new selection metric and evaluated its performance in comparison to other selection metrics.
We showed that basic block scheduling has low optimization potential on modern out-of-order processors.
That said, we also showed theoretically and through performance evaluation that maximum path latency is a useful instruction selection metric beside register pressure. The effectiveness of maximum path latency instruction schedulers scales with the size of the basic block due to a higher degree of potential ILP.

6.2. Further work

We do not expect that list scheduling on basic block level can be optimized and yield significantly higher performance than this thesis showed.
In the chapter "Design and implementation"(4) we inspected which possibilities are available to optimize instruction scheduling. This resulting decision tree creates a foundation for further work, which evaluates the remaining optimization possibilities. The most important of these possibilities are:

Order of instruction scheduling and the number of passes Instruction scheduling is only performed once before the register allocation. A topic of research ([21]) is a unified approach of register allocation and instruction scheduling. A unified register allocation and instruction scheduling step promises a better solution to the tradeoff between highest ILP and minimal register pressure.
Alternative it could be investigated, if multiple passes of the instruction scheduler or an instruction scheduler pass after register allocation, as Gibbons et al. [8] suggests, benefit the overall performance.

Global scheduling A huge impact on the effectiveness of list scheduling is the size of the set of scheduled instructions. A larger instruction set can have a higher amount of potential parallel (independent) instructions. Implementing

a global scheduling schema for the current list scheduler could be another great performance gain.
The priority list of heuristics may need to be adjusted because, for larger scheduled blocks, ILP-based heuristics are expected to perform better than register pressure based heuristics.
Global scheduling also induces additional overhead in the form of code spilling, because beyond basic blocks are not guaranteed to be executed, and instructions may depend on values, that vary depending on the execution path. This overhead has to be taken into account and decreases the potential benefit of global scheduling, while also increasing the implementation complexity immensely.
Global scheduling also introduces new metrics for the selection of instructions, because instructions do not have the property of assured execution like in basic blocks, and have therefore a probability of execution due to the control flow.

Constraint programming Constraint programming could be further investigated and integrated. There may be special case scheduling decisions that correspond to specific code patterns and contradict a single priority list of selection heuristics. Through constraint programming, the DAG could be transformed to solve these special cases, and then on the remaining degrees of freedom, list scheduling could be used.

6.3. Outlook

Instruction scheduling is a critical optimization step in a modern compiler. Because the latency of instructions of architecture implementations is important, scheduling has to adapt to new and better hardware.
More execution units make scheduling for higher ILP more worthwhile, while the main border of performance gains the natural dependencies in the code are, which can only be solved by writing better parallelizable code.
With higher instruction windows of out-of-order processors, scheduling for register pressure minimization seems more worthwhile, as the hardware already handles scheduling for high ILP.
Additionally, this decreases the effectiveness of basic block scheduling enormously, requiring modern instruction schedulers to use global scheduling scope and additional optimizations like loop unrolling.
Another big issue is the memory performance wall of microprocessors, which make reducing register usage as important as ILP. New architectures with bigger, less restrictive register sets induce an even higher performance boost and enable new scheduling possibilities than more execution units.

Bibliography

[1] J. E. Smith and G. S. Sohi, "The microarchitecture of superscalar processors," *Proceedings of the IEEE*, vol. 83, no. 12, pp. 1609–1624, 1995.

[2] T. J. W. I. R. C. R. Division, E. Lawler, J. Lenstra, C. Martel, B. Simons, and L. Stockmeyer, *Pipeline scheduling: A survey.* 1987.

[3] V. S. Menon, N. Glew, B. R. Murphy, A. McCreight, T. Shpeisman, A.-R. Adl-Tabatabai, and L. Petersen, "A verifiable ssa program representation for aggressive compiler optimization," in *ACM SIGPLAN Notices*, vol. 41, pp. 397–408, ACM, 2006.

[4] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson, "Practical improvements to the construction and destruction of static single assignment form," *Softw. Pract. Exper.*, vol. 28, pp. 859–881, July 1998.

[5] S. S. Muchnick, *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[6] M. Braun, S. Buchwald, and A. Zwinkau, *Firm-a graph-based intermediate representation.* KIT, Fakultät für Informatik, 2011.

[7] "The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers." `https://www.agner.org/optimize/microarchitecture.pdf`.

[8] P. B. Gibbons and S. S. Muchnick, "Efficient instruction scheduling for a pipelined architecture," in *Acm sigplan notices*, vol. 21, pp. 11–16, ACM, 1986.

[9] M. C. Rosier and T. M. Conte, "Treegion instruction scheduling in gcc," in *GCC Developers Summit*, 2006.

[10] "LLVM Documentation." `http://llvm.org/doxygen/`.

[11] K. Wilken, J. Liu, and M. Heffernan, "Optimal instruction scheduling using integer programming," in *Acm sigplan notices*, vol. 35, pp. 121–133, ACM, 2000.

[12] C. H. Mallon, "Registerdruckgewahre Befehlsanordnung." `http://www.info.uni-karlsruhe.de/papers/da_mallon.pdf`, Oct. 2008.

[13] G. Lindenmaier, "libFIRM – a library for compiler optimization research implementing FIRM," Tech. Rep. 2002-5, Sept. 2002.

[14] S. Arya, “Optimal instruction scheduling for a class of vector processors: an integer programming approach,” 1983.

[15] 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*, pp. 88–98, ACM, 2014.

[16] W. A. Havaki and Jr., “Treegion scheduling for vliw processors,” 1997.

[17] J. M. Codina, J. Llosa, and A. González, “A comparative study of modulo scheduling techniques,” in *Proceedings of the 16th international conference on Supercomputing*, pp. 97–106, ACM, 2002.

[18] J. A. Fisher, “Trace scheduling: A technique for global microcode compaction,” *IEEE transactions on computers*, no. 7, pp. 478–490, 1981.

[19] Y.-H. Shiau and C.-P. Chung, “Adoptability and effectiveness of microcode compaction algorithms in superscalar processing,” *Parallel computing*, vol. 18, no. 5, pp. 497–510, 1992.

[20] “SPEC CPU2000 Benchmark Suite.” `https://www.spec.org/cpu2000/`.

[21] S. S. Pinter, “Register allocation with instruction scheduling,” in *ACM SIGPLAN Notices*, vol. 28, pp. 248–257, ACM, 1993.

Erklärung

Hiermit erkläre ich, Steffen Kromm, dass ich die vorliegende Masterarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift

A. Code samples

A.1. LibFIRM maximum path latency implementation

```
static unsigned get_max_latency_succ(ir_node* irn)
{
  if (is_End(irn)) return 0;
  unsigned max = 0;
  foreach_irn_out(irn, index, succ) {
    ir_node *block = get_nodes_block(irn);
    if (get_nodes_block(succ) != block
      || is_End(succ) || is_Bad(succ) || succ == irn)
      continue;
    if (is_Pin(succ) || is_Sync(succ) || is_Proj(succ)) {
      max = MAX(max, get_max_latency_succ(succ));
    } else {
      flag_and_cost *fc_succ = get_irn_flag_and_cost(succ);
      calculate_total_latency(succ);
      max = MAX(max, fc_succ->maximum_path_latency);
    }
  }
  return max;
}

static void calculate_total_latency(ir_node* irn)
{
  flag_and_cost *fc     = get_irn_flag_and_cost(irn);
  if (fc && fc->maximum_path_latency != UINT_MAX) return;

  if (be_is_Keep(irn) || is_End(irn) || is_Bad(irn))
  {
    if (fc) fc->maximum_path_latency = 0;
    return;
  }
  if (is_Pin(irn) || is_Sync(irn) || is_Proj(irn)) {
    foreach_irn_out(irn, index, succ) {
      if (succ == irn) continue;
      calculate_total_latency(succ);
```

```
        }
        return;
    }

    if (!fc->no_root) {
        fc->maximum_path_latency = get_instruction_latency(irn);
    } else {
        fc->maximum_path_latency = 0;
        unsigned max_successor_latency =
            get_max_latency_succ(irn);
        fc->maximum_path_latency =
            get_instruction_latency(irn) + max_successor_latency;
    }
}
```