Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Bachelor Thesis

SUACA

A tool for performance analysis of machine programs

submitted by

Hendrik Meerkamp

submitted

July 10, 2018

Supervisor

Prof. Dr. Sebastian Hack

Advisor

Andreas Abel

Reviewers

Prof. Dr. Sebastian Hack
Prof. Dr. Jan Reineke

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,…… ……

(Datum / Date) (Unterschrift / Signature)

Abstract

When trying to highly optimize your code it is essential to know how well it fits your machine. In this work we present *Saarland University Architecture Code Analyzer* (**SUACA**), a tool that reimplements the throughput and port analysis of Intel's **IACA**. Additionally it offers various options to further investigate the code's performance as well as its bottlenecks. We will discuss what its capabilities are and how the results should be understood.

Acknowledgments

I would like to thank my advisors Prof. Sebastian Hack and Prof. Jan Reineke for the useful discussions and especially Andreas Abel for always finding time for me and all of my questions. I would also like to thank my friends Johanna Müller, Stefanie Lörsch, Kallistos Weis and Jonas Cirotzki for their proofreading.

Contents

1 Introduction

1.1 Motivation

Knowing your machine is a major advantage when trying to optimize high-performance scientific code. **IACA** (Intel Architecture Code Analyzer) [3] is a tool by Intel to analyze *x86* machine code with respect to a specific microarchitecture. However, it has some drawbacks that oftentimes prevent it from being useful in practice, mainly because it doesn't support the most recent processors. **IACA** 3.0, which was released in late 2017, supports the 4^{th} (Haswell) to the 6^{th} (Skylake) generation of Intel Core microarchitectures. Skylake was released in 2015. **IACA** 2.3 additionally supports the 2^{nd} (Sandy Bridge) and the 3^{rd} (Ivy Bridge) generation. So at the time of writing **IACA** is about three years behind and its further development remains unclear.
The second complication is that **IACA** is closed source. Its user guide [4] is the only documentation it has which provides little to no information about how it actually computes its output. As a result a user will often find himself wondering how its output fits the analyzed program.
In this work we present **SUACA** (Saarland University Architecture Code Analyzer), an open source alternative. It uses measurements provided by [1] which are parsed during runtime. This way a user does not rely on a software update of the tool as he can simply perform the measurements on his own, should we not already support his microarchitecture. At the time of writing **SUACA** supports all Intel Core microarchitectures from the 1^{st} (Nehalem) to the 8^{th} (Coffee Lake) generation, except for the server variant of Skylake.

1.2 Intel's Microarchitectures

In order to understand some the computations described in the following sections, we give a brief overview over Intel's microarchitectures. They use the *x86* instruction set. However, a single *x86* instruction will be not executed on the CPU as it is, but the instruction will be translated into a sequence of so called μops which can then be executed. Unfortunately, there is little to no official documentation about those μops, neither about the functionality of an individual one nor about their interaction with one another. From the measurements we can conclude that each microarchitecture has its own μops which makes it even harder to find reliable information.

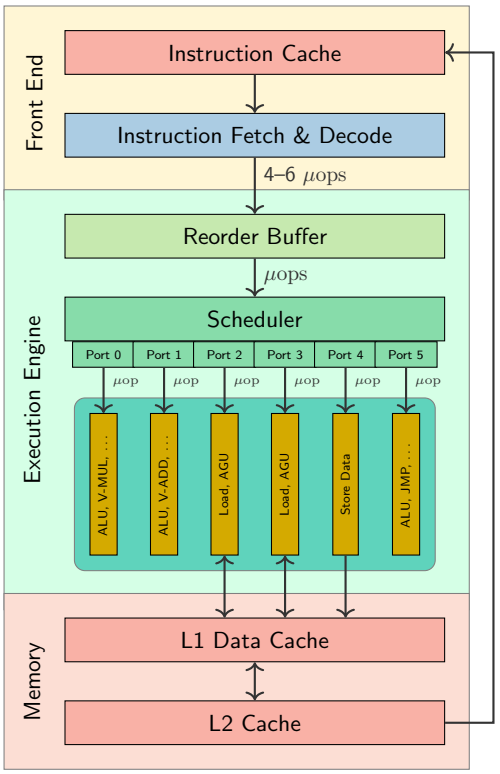

Figure 1.1: Pipeline of Intel Core CPUs (simplified). [1]

Figure 1.1 shows a sketch of a microarchitecture by Intel. We can see the front-end including the decoder unit, which is responsible for the translation of the instructions into the μops. In our simulation, we will only consider the number of μops the front-end produces each cycle which are currently 4–6 depending on the architecture. Our main interest is focused on the execution engine or more precisely on the scheduler (or reservation station) and the ports. The scheduler is responsible for the distribution of the μops over the ports. As mentioned before, a certain amount of those will be loaded into it by the front-end in each cycle. It has a maximum capacity which also depends on the specific architecture (the scheduler of the Sandy Bridge architecture we will be using for most of our examples has a capacity of 54). The most important property to observe from this figure are the ports. Each port can be seen as a pipeline that a μop can run through in order to be executed. The ports themselves hold the

actual execution units of the processor like the *ALU* or the *MULTIPLEXER*. Every port can hold a single μop per cycle and they support pipelining, which means that the port can be used again by another μop in the next cycle while others are still inside the execution units. The only exception from this is the *DIVIDER* unit which is slow at executing and can be blocked for multiple cycles. Usually it is not necessary that the μops are executed in program order. The so called out-of-order execution is possible whenever there is no dependency between the respective *x86* instructions or the μops themselves.

1.3 IACA's Analysis

In order to use **IACA** the code has to be prepared with the two markers that are defined in the *iacaMarks.h* header. As **IACA** is mostly used to analyze innermost loops of scientific code, we will show how those markers are inserted there:

```
#include "iacaMarks.h"

int main(void) {

    while (condition) {
        IACA_START
        //Some code here
    }
    IACA_END

    return 0;
}
```

When writing assembly code one can simply insert the markers that are defined in the *iacaMarks.h* header manually.
In the following we will perform a throughput analysis with **IACA** 2.3. The *throughput* is the average number of cycles needed to execute the body of the loop and is therefore the value a programmer should try to optimize.
The *latency* is the total number of cycles needed to execute a single iteration of the program. Unfortunately, the latency analysis support was dropped in **IACA** 2.2.
The analyzed program is not of particular interest here, but will be discussed later on. At the moment the focus is on how **IACA**'s output should be understood. For this purpose consider the following example:

```
Throughput Analysis Report
--------------------------
Block Throughput: 2.86 Cycles       Throughput Bottleneck: FrontEnd

Port Binding In Cycles Per Iteration:
---------------------------------------------------------------------------------------
|  Port  |  0   -  DV  |  1   |  2   -  D   |  3   -  D   |  4   |  5   |  6   |  7   |
---------------------------------------------------------------------------------------
| Cycles | 1.6    0.0  | 1.4  | 0.0    0.0  | 0.0    0.0  | 0.0  | 1.4  | 1.6  | 0.0  |
---------------------------------------------------------------------------------------

N - port number or number of cycles resource conflict caused delay, DV - Divider pipe (on port 0)
D - Data fetch pipe (on ports 2 and 3), CP - on a critical path
F - Macro Fusion with the previous instruction occurred
* - instruction micro-ops not bound to a port
^ - Micro Fusion happened
# - ESP Tracking sync uop was issued
@ - SSE instruction followed an AVX256/AVX512 instruction, dozens of cycles penalty is expected
X - instruction not supported, was not accounted in Analysis

| Num Of |                    Ports pressure in cycles                     |    |
|  Uops  |  0  - DV  |  1  |  2  -  D  |  3  -  D  |  4  |  5  |  6  |  7  |    |
---------------------------------------------------------------------------------
|   1    | 0.1       | 0.3 |           |           |     | 0.3 | 0.3 |     |    | mov rax, 0x1
|   1    | 0.9       |     |           |           |     |     | 0.1 |     |    | cmp rcx, 0x0
|   0F   |           |     |           |           |     |     |     |     |    | jnz 0x7
|   1    |           | 0.3 |           |           |     | 0.5 | 0.2 |     | CP | add rbx, rax
|   1    | 0.6       |     |           |           |     |     | 0.3 |     |    | jmp 0x5
|   1    |           | 0.4 |           |           |     | 0.3 | 0.3 |     | CP | add rbx, rax
|   1    |           | 0.3 |           |           |     | 0.3 | 0.4 |     | CP | add rbx, rbx
Total Num Of Uops: 6
```

Taking a first look, the example shows that **IACA** computed a *block throughput* of 2.86 cycles for this particular program.
The analysis also shows the bottleneck and the average port bindings. The port bindings represent the sum of the port pressure values we see in the bottom table.
A port is pressured whenever a μop is assigned to it. The pressure values themselves represent the average number of cycles the respective instruction has used each individual port. Due to port pipelining each μop only pressures the port it uses for a single cycle. Those pressure values therefore equal the number of μops that were assigned to the port and will add up to the number of μops (apart from rounding errors). The divider pipe is an exception to this, it specialties will be explained in Section 3.6.

1.4 Scope of Work

As already stated we will present a tool that is able to analyze *x86* assembler code with respect to a specific microarchitecture. Just like **IACA** our tool is able to find byte markers inside a compiled file and analyze the code in between. We are using Intel's *x86 Encoder Decoder* library [2] to disassemble said file, which allows us to support files of the *ELF*, *PECOFF* and *MACHO* format.

After disassembling, **SUACA** will perform a dependency analysis on the instructions and parse the measurement file. Finally it will perform a simulation of the code. It will not consider the actual effect of the instructions, only their latencies, dependencies and port usage. The output will be very similar to **IACA**'s and additionally **SUACA** offers some supplementary options which can be used to further investigate the given program. During all analyses the instructions are first considered in program order, although instruction reordering is still possible as we will see in Section 3.3. For several reasons, which we will are discussed in the following, our simulation will compute an estimate of the code's performance, not total numbers.
We will discuss all available options of **SUACA** in Chapter 2. In Chapter 3 we will then explain in detail how the most important parts of the simulation and the dependency analyses are implemented.

1.5 Measurements

As mentioned before a crucial part of **SUACA**'s functionality are the measurements provided by [1]. Consider this snippet from the XML-measurement-file file:

```
<instruction ... iform="ADD_LOCK_MEMv_GPRv" ...>
    <operand idx="1" r="1" type="mem" w="1" width="64"/>
    <operand idx="2" type="reg" ...>RAX,RCX,RDX,RBX,...</operand>
    <operand idx="3" type="flag" ...>OF</operand>
    <operand idx="4" type="flag" ...>SF</operand>
    ...
    <architecture name="NHM">
        <measurement port15="2" port2="1" port3="1" port4="1" total_uops="5">
            <latency cycles="19" startOp="1" targetOp="3"/>
            <latency cycles="19" startOp="1" targetOp="4"/>
            ...
        <\measurement>
    </architecture>
<\instruction>
```

We dotted out some unnecessary or redundant information. As we can see in the first line this is the information for the instruction with the *iform* "ADD_LOCK_MEMv_GPR". *iform* is an enum from the XED Library [2] that is used to identify instructions. We can extract the following information from our snippet:

- One of the *RAX, RCX, RDX, ...* registers is an operand and they have the id 2. We only need the mapping of $id \rightarrow register$ here as the xed library will tell us which operands are actually used in the analyzed programs. Similarly, the flags have their ids. The flags are the single bits of the *RFLAGS* register in *x86*.

- We have some measurements for the Intel Nehalem (NHM) microarchitecture.
- When simulating Nehalem the instruction consists of 5 μops. Two of these can use ports 1 and 5 and one each can use port 2, 3 and 4.
- As soon as the operand with id 1 is available it will take 19 cycles to compute the result for the operand with id 3.

As we do not know which μop is responsible for the computation of which operand we will ignore the *startOp* property. More precisely we do know that it takes 19 cycles to produce the result of operand 3 as soon as operand 1 is available, but we do not know which of the instruction's μops would actually perform this computation. **SUACA** will therefore always wait until all operands the instruction needs to read are available.
Most instructions have several latency items, depending on the number and kind of operands. In this case there is no information for operand 2 as the instruction will not write to those registers. However, the latency for the operand with id 4 is 19 cycles as well. Some instructions actually produce their results in a specific order. It might be the case that one operand is available after 3 cycles and another one after 5, so an instruction that only needs the first of those operands has to wait 3 cycles whereas another one that needs the second operand has to wait 5. **SUACA** can simulate this behavior as it knows which operand is causing the dependency. When simulating the whole instruction, **SUACA** takes the maximum of those values. Note that those values are always best case i.e., no port was blocked.
We can already observe a lack of information on the μops. As mentioned before we do not know which μop is responsible for which computation, so we cannot know how the above mentioned latency values come about. Probably those early results are computed by some of the μops, but as we do not know which ones we can not always precisely compute the correct latency should an instruction be delayed. If the μops of an instruction do not depend on each other it is even possible that the order of the results changes, which we also can not simulate. We also have no information about the dependencies between the μops themselves which does pose a major problem which we will further discuss in Section 3.3.

1.6 Related Work

One can find general information about **IACA** at its website [3]. The user's guide [4] gives additional information about the usage and provides some examples.

Andreas Abel [1] provides the measurements which enable us to compute our results.

Jan Laukemann [6] implemented an open source alternative to **IACA** called **OSACA** [7]. It relies on the measurements provided by Johannes Hofmann [5]. We will discuss the differences between the three tools in Chapter 4.

2 Functionality of SUACA

In this chapter we are going to explain and show the full functionality of **SUACA**. For each available analysis we will show an example run and analyze the results. As we want to compare the different runs with each other we will use the following example code for each of them:

```
1     mov rax, 1
2     cmp rcx, 0
3     jne else
4     add rbx, rax
5     jmp end
6  else:
7     add rbx, rax
8  end:
9     add rbx, rbx
```

For the sake of simplicity this code is only designed to be an example which contains two branches which both have a dependency on the previous and the following instruction.

2.1 Throughput Analysis

As mentioned above, the major use case of **IACA** is analyzing an innermost loop. While **IACA** will therefore always assume a loop and somehow determine its number of iterations, **SUACA** will give the user the option to choose them. For the following example **SUACA** considered 200 loop iterations and the Sandy Bridge microarchitecture. We will do so for all future examples except stated otherwise. Now consider the following output which will demonstrate the basic values **SUACA** computes.

```
Block throughput: 2.34 cycles
Block throughput with perfect front-end: 2.34 cycles
Block throughput with infinitely usable ports: 2.00 cycles
Block throughput without dependencies: 2.34 cycles
Microops per cycle: 2.99

Analysis for architecture: SNB

 Line  ||   Num   ||   had   || caused  ||            Used Ports
       ||   Uops  || to wait || to wait ||   0   ||   1   ||   2   ||   3   ||   4   ||   5   ||
-------------------------------------------------------------------------------------------------
   0   ||    1    ||  15.1   ||  43.2   ||  0.0  ||  1.0  ||       ||       ||       ||       || mov rax, 0x1
   1   ||    1    ||  15.8   ||  32.3   ||  0.3  ||  0.3  ||       ||       ||       ||  0.3  || cmp rcx, 0x0
   2   ||    1    ||  16.5   ||  20.8   ||       ||       ||       ||       ||       ||  1.0  || jnz 0x7
   3   ||    1    ||  15.5   ||  28.9   ||  0.7  ||  0.3  ||       ||       ||       ||       || add rbx, rax
   4   ||    1    ||  16.8   ||  20.4   ||       ||       ||       ||       ||       ||  1.0  || jmp 0x5
   5   ||    1    ||  15.2   ||  28.9   ||  0.3  ||  0.7  ||       ||       ||       ||       || add rbx, rax
   6   ||    1    ||  15.8   ||  43.9   ||  1.0  ||       ||       ||       ||       ||       || add rbx, rbx
Total number of Uops: 7
```

At the beginning, the following values are noticed:

- **Block throughput** is the average number of cycles needed to execute the program ($\frac{Total\ number\ of\ cycles}{Number\ of\ iterations}$).
- **Block throughput with perfect front-end** can be used to see if the front-end of the processor was the bottleneck of the execution. To compute this value **SUACA** will perform a full analysis of the program. However, it will assume that $number\ of\ \mu ops\ loaded\ per\ cycle = capacity\ of\ reservation\ station$. If the runtime experiences a speedup, we conclude that the front-end was indeed the bottleneck. Note that this does not ignore the maximum capacity of the scheduler.
- **Block throughput with infinitely usable ports** is computed similarly. It will perform a full analysis, but every port can be used arbitrarily in each cycle. So several μops can use the same port simultaneously. Should the runtime improve, we can come to the conclusion that one of the ports has to be the bottleneck.
- **Block throughput without dependencies** is as well akin to the aforementioned values. This time the dependencies are ignored during the simulation. An improved throughput indicates that the dependencies between instructions are the bottleneck.

When looking at the block throughput values here we can observe that our example program runs quite significantly faster with infinitely usable ports. This makes sense as only ports $0, 1$ and 5 can be used by the instructions we are using. The two jump instructions are the biggest offender since they can exclusively use port 5.

In some corner cases it might be possible that both the front-end and the ports are responsible for a decreased runtime. This can occur if every loaded instruction is directly computed (front-end bottleneck), but if the front-end was faster there would be no other port to run the additional loaded instructions on. In this case, none of the first values would differ from the normal *Block throughput*, although they are actually both part of the bottleneck.
Similar behavior can be observed with multiple combinations of the above mentioned versions of our simulation. For this reason **SUACA** gives the user the option to perform these special simulations in all possible combinations.

In the table we can observe the following columns:

- The **had to wait** column describes the average number of cycles the instruction experienced a delay from either blocked ports or register dependencies.

- The **caused to wait** column describes the average number of cycles the instruction caused a delay similar to the **had to wait** value. However, it will not track transitive dependencies. So consider a program that has a dependency chain of $A \rightarrow B \rightarrow C$ (where A, B and C are instructions of your program) and A is not fully computed. A will cause B to be delayed, resulting in an increased **caused to wait** value of A. B will then cause C to be delayed, resulting in an increased **caused to wait** value of B.

- The **Used Ports** columns describe how many cycles the respective port has been used on average. Due to the port pipelining this is equal to the number of μops that were assigned to this port in all cases except the divider pipe (see Section 3.6). If possible, **SUACA** will always assign a μop to the port that has been used the least during the analysis (out of the ports that this particular μop is able to use) in order to achieve an even distribution of the ports. A detailed description of how those are computed can be found in Section 3.3.

In **SUACA**'s output the values of the *caused to wait* column exceed those of the *had to wait* column quite significantly this is due to the fact that multiple instructions can cause a delay for a single other instruction. We will further explain this in Section 2.2.
In the *Used Ports* column we can observe that line 0 has used port 0 0.0 times. This just means that this instruction has used port 0, but to such a small amount that the rounding resulted in the 0.0 value.

2.2 Latency Analysis

The latency of a program is the number of cycles needed to execute it once. **SUACA** can be used to compute the latency by running its analysis with a single iteration.

```
Block throughput: 3.00 cycles
Block throughput with perfect front-end: 3.00 cycles
Block throughput with infinitely usable ports: 3.00 cycles
Block throughput without dependencies: 3.00 cycles
Microops per cycle: 2.33

Analysis for architecture: SNB

 Line  ||   Num   ||   had   || caused  ||           Used Ports
       ||   Uops  || to wait || to wait ||   0   ||   1   ||   2   ||   3   ||   4   ||   5   ||
-------------------------------------------------------------------------------------------------
   0   ||    1    ||         ||   1.0   ||  1.0  ||       ||       ||       ||       ||       || mov rax, 0x1
   1   ||    1    ||         ||   1.0   ||       ||  1.0  ||       ||       ||       ||       || cmp rcx, 0x0
   2   ||    1    ||   1.0   ||   1.0   ||       ||       ||       ||       ||       ||  1.0  || jnz 0x7
   3   ||    1    ||   1.0   ||   1.0   ||  1.0  ||       ||       ||       ||       ||       || add rbx, rax
   4   ||    1    ||   1.0   ||         ||       ||       ||       ||       ||       ||  1.0  || jmp 0x5
   5   ||    1    ||         ||   1.0   ||       ||  1.0  ||       ||       ||       ||       || add rbx, rax
   6   ||    1    ||   1.0   ||         ||  1.0  ||       ||       ||       ||       ||       || add rbx, rbx
Total number of Uops: 7
```

A closer look at the concrete values of this particular run reveals that the sum of the *caused to wait* column is 5, whereas the *had to wait* column only sums up to 4. This is due to the fact that both line 3 and 5 are responsible for the delay of line 6 due to the dependency via the *rbx* register. The exact dependencies can be found in Figure 2.3. The same behavior arises when an instruction cannot be executed, because all ports are blocked. When three different instructions A, B and C block the ports another instruction D would like to use the *caused to wait* values of A, B and C are increased while only D's *had to wait* value will increase.
We can also see that the instruction in line 0 has actually used port 0. This explains the 0.0 average usage value of the throughput analysis.

2.3 Control Flow Graph

The control flow graph is mainly used to compute the correct dependency graph. The CFG of our example program can be seen in Figure 2.1. The red edge only appears if the analysis runs in a loop as it represents the "back jump" to the start of the program that will not appear in a single iteration. For the sake of readability we dashed the red edge and will do so in future graphs.

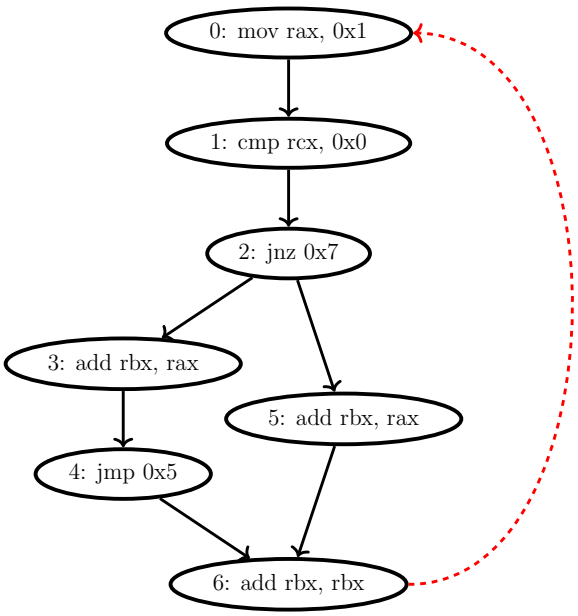

Figure 2.1: Control flow graph

2.4 Dependency Graph

The dependency graph describes all register dependencies that occur in the program. An edge from node A to node B indicates that the instruction represented by B depends on the instruction represented by A. **SUACA** will only track read-after-write dependencies, as those are the ones that can actually cause an instruction to be delayed. Whenever an instruction uses a memory address, **SUACA** will try to extract all used registers. Because we use the *XED* library [2], we can also consider the suppressed operands that cannot be seen in the code. One example for those suppressed operands is the *RFLAGS* register, but there are several examples where an instruction has to access a register that does not appear in the code itself. **SUACA** will not keep track of the stack as this would often require runtime specific information. The detailed algorithm that is used to generate this graph can be found in Section 3.1. First consider the graph shown in Figure 2.2 which will be generated in the "single loop case".

This graph was generated with the *CFG* in mind since there is no edge from node 3 to node 5. Additionally, we can observe that **SUACA** does differentiate between the different flags contained in the *RFLAGS* register as the dependence is only reasoned with the zf flag.

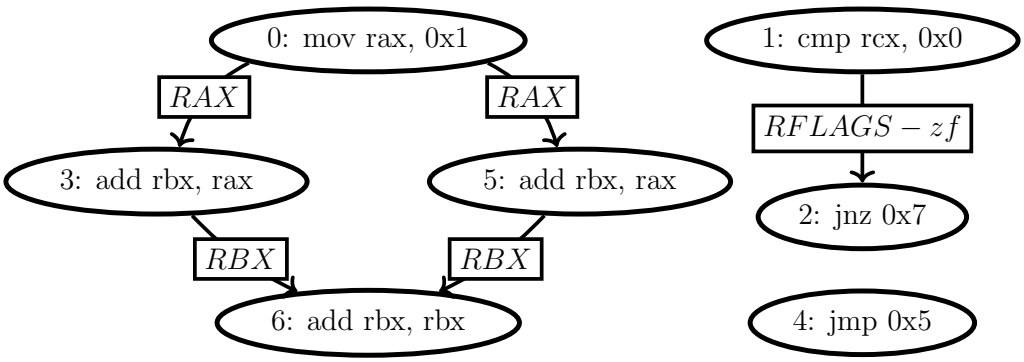

Figure 2.2: Dependency graph without loop dependencies

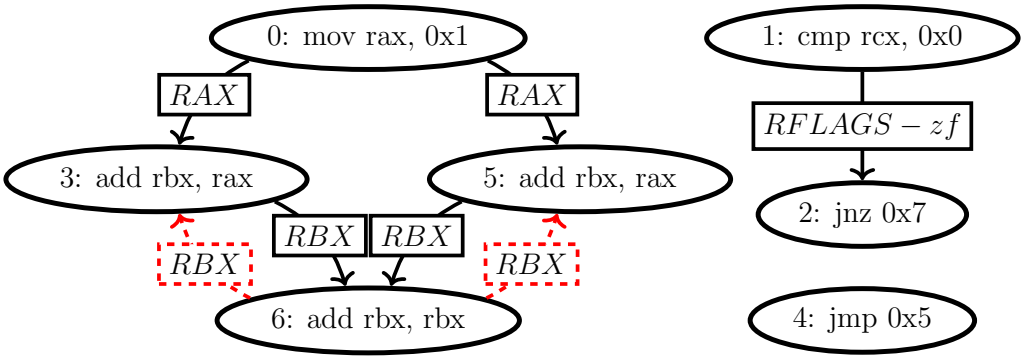

Figure 2.3: Dependency graph with loop dependencies

When **SUACA** is called with at least 2 iterations it will also track all "loop dependencies". Figure 2.3 shows the graph with those in consideration. We can see that our program has two loop dependencies.

2.5 Architecture Selection

As previously discussed, one of the big advantages of **SUACA** is that one can easily add new architectures on which the analysis can be based on. **SUACA** gives the user the ability to choose a specific microarchitecture. For the next example we will use Intel's Coffee Lake microarchitecture instead of the Sandy Bridge microarchitecture we used previously (the first two columns are left out to improve readability):

```
Block throughput: 2.00 cycles
Block throughput with perfect front-end: 2.00 cycles
Block throughput with infinitely usable ports: 2.00 cycles
Block throughput without dependencies: 1.75 cycles
Microops per cycle: 3.49

Analysis for architecture: CFL

   had   || caused  ||            Used Ports
 to wait || to wait ||   0   ||   1   ||   2   ||   3   ||   4   ||   5   ||   6   ||   7   ||
-------------------------------------------------------------------------------------------------------
   4.7   ||  15.9   ||  0.1  ||  0.8  ||       ||       ||       ||  0.1  ||  0.1  ||       || mov rax, 0x1
   5.4   ||  13.6   ||  0.2  ||  0.5  ||       ||       ||       ||  0.1  ||  0.2  ||       || cmp rcx, 0x0
   6.2   ||   9.8   ||  0.4  ||       ||       ||       ||       ||       ||  0.6  ||       || jnz 0x7
  43.2   ||  48.9   ||  0.1  ||  0.2  ||       ||       ||       ||  0.7  ||  0.0  ||       || add rbx, rax
   5.1   ||   9.1   ||  0.2  ||       ||       ||       ||       ||       ||  0.8  ||       || jmp 0x5
  42.8   ||  51.6   ||  0.7  ||  0.1  ||       ||       ||       ||  0.2  ||  0.0  ||       || add rbx, rax
  43.6   ||  90.8   ||  0.1  ||  0.2  ||       ||       ||       ||  0.7  ||  0.0  ||       || add rbx, rbx
Total number of Uops: 7
```

The most significant change marks the infinitely usable ports analysis which no longer experiences an improvement in runtime. This is due to the larger number of ports in the Coffee Lake architecture. Instead, we can now observe that the three *add* instructions, or more so their dependencies on each other, are responsible for most of the delays. We can conclude this from the *had to wait* and *caused to wait* values of these instructions and also the improved throughput with ignored dependencies.

2.6 Detailed Information

SUACA can also deliver some detailed information about one particular line. This can be useful to determine how a specific instruction causes and experiences a delay. The following table shows the result of a run on our example program with 200 iterations and details for line 0. We are using the Sandy Bridge architecture again.

```
Detailed delay information for instruction: mov rax, 0x1 in line 0

                Maximum latency: 1

                Latencies for dependencies:

                Line || 0 -> Line || Line -> 0
                ---------------------------------
                 3   ||     1     ||     0
                 5   ||     1     ||     0


                Delay caused by dependencies:

                Line || was delayed || has delayed
                ---------------------------------
                 3   ||    15.2     ||     0.0
                 5   ||    14.5     ||     0.0


                Delay caused by blocked ports:

                Port || was delayed || has delayed
                ---------------------------------
                 0   ||     0.0     ||    15.1
                 1   ||    13.5     ||    15.1
                 5   ||     0.0     ||    15.1
```

In order to get a better understanding of those values, we will split the output and explain them step by step.

```
                Maximum latency: 1

                Latencies for dependencies:

                Line || 0 -> Line || Line -> 0
                ---------------------------------
                 3   ||     1     ||     0
                 5   ||     1     ||     0
```

First we can see that this instruction has a maximum latency of one cycle. This value can differ from those of the table below as we have seen in Section 1.5.
The table itself shows the latencies **SUACA** used for the dependencies. The second column shows the delay from the analyzed line to the line given in the first column and the third row shows the delay in the other direction. In our case, lines 3 and 5 depend on line 0 (see Figure 2.3) and we can see that those lines actually have to wait one cycle for line 0 to be finished, and line 0 itself is independent of the other two.

```
Delay caused by dependencies:

Line || was delayed || has delayed
-----------------------------------
 3   ||    15.2     ||     0.0
 5   ||    14.5     ||     0.0
```

This table now shows that lines 3 and 5 actually are delayed by our analyzed line. On average, line 3 has to wait 15.2 cycles for line 0 to be finished while line 5 has to wait 14.5 cycles. Of course, both do not cause any delay on line 0 as there is no dependence.

```
Delay caused by blocked ports:

Port || was delayed || has delayed
-----------------------------------
 0   ||     0.0     ||    15.1
 1   ||    13.5     ||    15.1
 5   ||     0.0     ||    15.1
```

Finally **SUACA** outputs how much delay was caused by the ports. First consider that the *mov* instruction in line 0 can, in theory, use ports $0, 1$ and 5. In our case it causes a delay of 13.5 cycles per iteration on another instruction, because it uses port 1. It does not cause any delay on the other two ports that it might use simply because it always uses port 1 in our particular case (see Section 2.2). The second column exhibits that line 0 experiences a delay of 15.1 cycles per iteration because all three usable ports were blocked. The *mov* instruction we are considering only consists of a single μop which leads to all three of those values being identical.
More precisely, the *mov* instruction can only be delayed by blocked ports if all three of its usable ports are blocked, otherwise it would just select the free one. So all three of those blocked ports are responsible hence the three identical values.
However, this is not always the case since an instruction might consist of more than one μop. We will discuss this further in Section 3.3.

2.7 Branch Analysis

Finally, **SUACA** is able to analyze different branches. As we have seen in Figure 2.1 and Figure 2.2, the "normal" analysis already considers branches for its dependencies. However, as the effect of the instructions will be completely ignored in the simulation, the branches will not have any other effect.

The actual branch analysis will perform two simulations, one for each branch. This will always consider the first *jump* instruction and this one only. It will not acknowledge every single possible path through a program with multiple branches. We will now consider the branch analysis of our example program, with the Sandy Bridge architecture and 200 iterations:

```
Left branch analysis:

Block throughput: 2.00 cycles
Block throughput with perfect front-end: 2.00 cycles
Block throughput with infinitely usable ports: 2.00 cycles
Block throughput without dependencies: 2.00 cycles
Microops per cycle: 2.99

Analysis for architecture: SNB

 Line  ||   Num   ||   had   || caused  ||            Used Ports
       ||   Uops  || to wait || to wait ||   0   ||   1   ||   2   ||   3   ||   4   ||   5   ||
 ------------------------------------------------------------------------------------------------
   0   ||    1    ||  14.6   ||  30.1   ||  0.0  ||  1.0  ||       ||       ||       ||       || mov rax, 0x1
   1   ||    1    ||  15.6   ||  30.3   ||       ||  1.0  ||       ||       ||       ||       || cmp rcx, 0x0
   2   ||    1    ||  15.6   ||  23.8   ||       ||       ||       ||       ||       ||  1.0  || jnz 0x7
   3   ||    1    ||  15.6   ||  31.2   ||  1.0  ||       ||       ||       ||       ||       || add rbx, rax
   4   ||    1    ||  16.6   ||  22.9   ||       ||       ||       ||       ||       ||  1.0  || jmp 0x5
   6   ||    1    ||  15.7   ||  30.3   ||  1.0  ||  0.0  ||       ||       ||       ||       || add rbx, rbx
Total number of Uops: 6


Right branch analysis:

Block throughput: 2.00 cycles
Block throughput with perfect front-end: 2.00 cycles
Block throughput with infinitely usable ports: 2.00 cycles
Block throughput without dependencies: 1.67 cycles
Microops per cycle: 2.49

Analysis for architecture: SNB

 Line  ||   Num   ||   had   || caused  ||            Used Ports
       ||   Uops  || to wait || to wait ||   0   ||   1   ||   2   ||   3   ||   4   ||   5   ||
 ------------------------------------------------------------------------------------------------
   0   ||    1    ||   1.0   ||   2.8   ||  0.5  ||  0.5  ||       ||       ||       ||       || mov rax, 0x1
   1   ||    1    ||   1.5   ||   3.4   ||  0.2  ||  0.7  ||       ||       ||       ||  0.1  || cmp rcx, 0x0
   2   ||    1    ||   6.8   ||   4.7   ||       ||       ||       ||       ||       ||  1.0  || jnz 0x7
   5   ||    1    ||  40.0   ||  42.2   ||  0.5  ||  0.3  ||       ||       ||       ||  0.2  || add rbx, rax
   6   ||    1    ||  40.7   ||  42.9   ||  0.4  ||  0.2  ||       ||       ||       ||  0.4  || add rbx, rbx
Total number of Uops: 5
```

This kind of information can be useful to determine which of the two branches is more favorable for the execution. It can as well be used to detect if there are any significant differences between the two at all.
One can also use this in combination with the detailed analysis as **SUACA** will tell you the original line of all instructions in both parts of the branching analysis.

2.8 The Command Line Interface (CLI)

The CLI of **SUACA** works as followed:

suaca [*option*] *path_to_file*

where [*option*] is one or several of the following:

- *-pf* triggers the perfect front-end analysis.
- *-ip* triggers the infinite port analysis.
- *-nd* triggers the dependency free analysis.
- *-cfg* will print the control flow graph into a file called *controlflow.dot*. The format will be graphviz readable.
- *-dg* will print the dependency graph into a file called *dependency.dot*. The format will also be graphviz readable.
- *-p* triggers the "performance mode". More specifically, this will prevent the three extra analyses that are needed to generate the three additional block throughput values. In most cases however, **SUACA**'s performance bottleneck will be the parser for the measurement files which cannot be deactivated.
- *-b* triggers the branch analysis.
- *--iform* is used to print the iforms of all instructions.
- *--arch* x will consider x as the underlaying microarchitecture of the analysis. At the time of writing the available options are NHM (Nehalem), SNB (Sandy Bridge), IVB (Ivy Bridge), HSW (Haswell), BDW (Broadwell), SKL (Skylake), KBL (Kaby Lake) and CFL (Coffee Lake). The default value is SNB.
- *--loop* x will trigger the loop analysis. The default value of x is 1.
- *--detail* x will print detailed information about line x.
- *--setup* x y sets the default values for the architecture (x) and the number of iterations (y). Note that one always has to use both values.
- *--print-default* prints the default values for architecture and number of iterations.

3 Implementation

When trying to analyze a program, the user sometimes needs to fully understand how the results were calculated. Especially, if he aims to use them to improve his code. Therefore, we will discuss **SUACA**'s most important algorithms in this chapter. We will first explain the individual steps and fit them together in the end.

3.1 Dependency Analysis

3.1.1 Single Iteration

Here we want to take a look at the algorithm that computes the dependency graph.
First we want to discuss a simpler version that ignores the control flow of the program.

Algorithm 1: Dependency analysis without control flow

```
1  Function dep_analysis(instructionlist inst_list):
2      Map := map from register to line;
3      DG := Graph that has the same nodes as CFG, but no edges;
4      foreach instruction i in inst_list do
5          foreach register_operand r in operands(i) do
6              if is_read(r) then
7                  DG.add_edge(Map[r], line_of(i));
8              else
9                  Map[r] = line_of(i);
10             end
11         end
12     end
13     return DG;
14 end
```

Where

- *Map* maps each register to the last line with a write access.
- *operands(i)* returns a list of all operands of instruction i. A register that is first read then written to will be contained twice. The order will be first read then write access.
- *is_read(r)* returns true if the operand r will be read and false if it will be written to.
- *line_of(i)* returns the line of instruction i in the original program.

This algorithm will iterate over all instructions in program order. For each instruction i it will then iterate over all of its operands. For each operand it will check if it is accessed via read or write. If it is written to, the algorithm will map the register to the current line. If it is read, the algorithm will add an edge from the last write access to the current line.

The runtime of this algorithm is $\mathcal{O}(n * m)$ where n is the number of instructions and m the maximum number of operands that occur in the program.

Note that we consider every operand as a register. In practice, an operand can of course be a memory address. In this case **SUACA** will extract all registers from that address and treat them as read operands. **SUACA** does not support memory dependencies so far as we would need to keep track of the whole memory. As we have seen in Figure 2.3, **SUACA** is able to differentiate between the different flags. For readability we ignore the special case of the *RFLAGS* register here.
Now we want to take a look at the control flow sensitive algorithm that **SUACA** actually uses.

Algorithm 2: Control flow sensitive dependency analysis

```
1  Function dep_analysis_start(CFG):
2      Map := map from register to line;
3      DG := graph that has the same nodes as CFG, but no edges;
4      Node := startnode of CFG;
5      dep_analysis(CFG, DG, Map, Node);
6      return DG;
7  end
8  Function dep_analysis(CFG, DG, Father-Map, Node):
9      Map := copy of Father-Map;
10     while true do
11         foreach register_operand r in operands(instruction_of(Node))
            do
12             if is_read(r) then
13                 DG.add_edge(Map[r], line_of(i));
14             else
15                 Map[r] = line_of(i);
16             end
17         end
18         if num_successors(CFG, Node) = 0 then
19             return;
20         end
21         Node = successor(CFG, Node, 0);
22         if num_successors(CFG, Node) > 1 then
23             dep_analysis(CFG, DG, Map, successor(CFG, Node, 1));
24         end
25     end
26 end
```

Where

- *DG* has a Node for every instruction in the program. Just like the CFG.
- *instruction_of(Node)* returns the instruction that Node represents.
- *num_successors(Graph, Node)* returns the number of successors of *Node* in the Graph.
- *successor(Graph, Node, i)* returns the i^{th} successor of *Node* in the Graph.

This time we will "climb along" the *CFG*. If we never face a branch i.e., *num_successors()* never returns a value greater than 1, this algorithm will do

exactly the same as the one we have just seen.
In the case of *num_successors()* > 1 we will make another call of *dep_analysis()* on the "right branch". From this point on, there will be two analyses, one for every branch in the CFG. Each analysis has its own Map since there can be different writes on each branch. Note that we will not join the two analyses as we would need to find the first mutual descendant.
In the worst case every instruction is a branch, so we would spawn a new function for each of them. This leads to a runtime of $\mathcal{O}(n^2 * m)$.

We assume no backbranches i.e., no loops, in the program for the above mentioned algorithm. In practice, **SUACA** will simply check for each branch if it is a backbranch, and should the situation arise ignore it.

3.1.2 Multiple Iterations

When ordering **SUACA** to run the program in multiple loops we need to adjust the dependency analysis algorithm as this can cause some "loop dependencies". In order to solve this, we will simply consider the program twice. So we will append a copy of the program to itself, compute the CFG and afterwards run the above mentioned algorithm. Because we know the original length of our program, we can extract all "loop dependencies" from the resulting dependency graph.

3.2 Simulation of the Front-End

Although our main task is to simulate the scheduler we still want to consider the front-end in our analysis. Depending on the microarchitecture, the front-end is able to produce a certain amount of μops each cycle. For example, the Sandy Bridge architecture will produce at most 4 μops per cycle. However, we still have to acknowledge the capacity of the scheduler since the front-end might be faster than the execution itself. The scheduler of the Sandy Bridge architecture has a maximum capacity of 54 μops.
We will now briefly discuss how our simulation actually performs those loads.

Algorithm 3: Load instructions into scheduler

```
1 Function load_instructions(instruction_queue queue):
2 |   Waiting := first element of queue that is not fully loaded;
3 |   Loadable := max(Loads per cycle, remaining space in station);
4 |   while Loadable > 0 do
5 |   |   loaded := load_μops(Waiting, Loadable);
6 |   |   Loadable = Loadable − loaded;
7 |   end
8 end
```

Where

- *queue* is a queue of all instructions that still have to be executed.
- *Waiting* is initially set by searching for the first element in queue that has not been loaded into the scheduler by the front-end.
- *load_μops(Waiting, x)* loads x μops of *Waiting* and returns the number of μops that were actually loaded.

So it is possible that an instruction is partially (i.e., only some of its μops) loaded into the scheduler.

3.3 Choosing the Ports

In this section we are going to discuss how exactly the ports which an instruction uses are chosen. As we have seen in Section 1.5 we know of how many μops an instruction consists and which ports those μops can use. As seen in Section 3.2, an instruction can be loaded partially, which we will have to consider here.
The following algorithm contains several crucial details to the simulation. First we will see how **SUACA** tries to distribute all μops equally over all ports. It also demonstrates the exact situations in which we will execute an instruction. Lastly it explains how the *had to wait* and *caused to wait* columns we introduced in Section 2.1 are computed.

Algorithm 4: Choose ports for loaded instructions

```
1  Function choose_ports(instruction_queue queue):
2      while loaded_μops(Instruction) > 0 do
3          if not all_dependencies_resolved(Instruction) then
4              Instruction.has_to_wait++;
5              foreach Father ∈ direct_predecessors(Instruction) do
6                  Father.caused_to_wait++;
7                  Father.caused_to_wait_depedency(Instruction)++;
8                  Instruction.had_to_wait_depedency(Father)++;
9              end
10         else
11             Executable := true;
12             if not is_fully_loaded(Instruction) then
13                 Executable = false;
14             end
15             foreach μop μ ∈ loadedμops(Instruction) do
16                 Success := assign_to_ports(μ);
17                 if not Success then
18                     Executable = false;
19                 end
20             end
21             if Executable then
22                 add_to_executionlist(Instruction);
23             else
24                 Blamed := Set of instructions;
25                 foreach p ∈ blocked_ports(Instruction) do
26                     if not Blamed.contains(p.using_instruction()) then
27                         p.using_instruction().caused_to_wait++;
28                         p.using_instruction().caused_to_wait_port(p);
29                         Instruction.had_to_wait_port(p);
30                         Blamed.add(p.using_instruction());
31                     end
32                 end
33                 Instruction.has_to_wait++;
34             end
35         end
36         Instruction = queue.next(Instruction);
37     end
38 end
```

Algorithm 5: Assign μop to port

```
1 Function assign_to_ports(μop μ):
2     foreach p ∈ port_queue do
3         if μ.can_use(p) and p.is_free() then
4             p.uses(μ);
5             return true;
6         end
7     end
8     return false;
9 end
```

Algorithm 4 iterates over all instructions in program order as long as the current instruction is at least partially loaded into the scheduler. For each instruction it will first check if all of its dependencies have been resolved i.e., all predecessors in the dependency graph have finished their execution (or at least produced the needed results). If not, it cannot be executed and the delay counters have to be increased. Notice that we have separate counters for the cumulative delays and the special delays (e.g. caused_to_wait_depedency(), caused_to_wait_port(p)), which are only needed for the detailed analysis (Section 2.6). We will see the same behavior for the ports and this explains why the special delays will not always sum up to the cumulative ones.
If all dependencies have been resolved, **SUACA** will try to assign all μops of the instruction to a port. To achieve this the algorithm will iterate over all μops that have been loaded into the scheduler. Note that this will ignore all μops that have been put into a port already. So if all μops of an instruction are currently in the port pipeline this algorithm will basically just put the instruction into the execution list.
The function *assign_to_ports(μop, Instruction)* is described in Algorithm 5. This function will iterate over all ports in prior usage order and assign the μop if possible. More precisely *port_queue* contains all ports and is sorted by usage throughout the whole simulation. If possible it will assign the μop to the port and return a success. If no usable port was free it will return a fail. We can observe that this is a greedy algorithm that is obviously not optimal in regards to the distribution of all μops over the ports. However, we assume that this greedy algorithm comes close to what the schedulers are doing in reality.
If the assignment or the load of a single μop failed the flag *Executable* will be set to *false*. As we can see in line 22 the instruction will only be added to the execution list (which we will further discuss in Section 3.4) if this flag is set. This means that an instruction will not be executed as soon as a single

port was blocked. Note that an instruction can not block itself i.e., if all blocked ports were blocked by a μop of the same instruction this function will still return *true*. We did not include this special case here for the sake of readability.
We have to be this strict, because of our measurements. As we have seen in Section 1.5 those will always contain the best case for latency. The biggest problem we face here is the missing information about the μops. We do not know anything about the dependencies between them and so we do not know if there is an order in which those have to be executed, or if they can be executed simultaneously. So we have to assume a delay as soon as a single μop is denied a port although that might not actually be the case in reality. This means that we will potentially overestimate the latency of a single instruction or the whole program. It is possible though that **SUACA** will actually underestimate the latency of a program as one can note in the following examples.

Consider two instructions X and Y. X consists of two μops one of which can use port 0, and the other can use port 1. Y only consists of one μop which can use port 1. X is in front of Y in program order, both are fully loaded, not dependent on each other and have a latency of 2 cycles.
For this example we assume that the second μop (port 1) of X depends on the first (port 0). The problem is that **SUACA** does not have this information. So the simulation will do the following: In the first cycle it will assign X to ports 0 and 1. There is no port left for Y so it will not be added to the execution list. This will happen in the second cycle and as Y's latency was two cycles **SUACA** will compute a total latency of three cycles (as X was executed in the first and second).
However, in reality X will not block port 1 in the first cycle as this particular μop depends on the one that uses port 0. So in the first cycle X will only use port 0. Y can then freely use port 1. In the second cycle X can then use port 1. No port was ever blocked so there simply will be no delay, both instructions could be executed simultaneously. So the "real latency" of our example would be two cycles.

Now we will consider the same example but with switched instruction order of X and Y. In this case **SUACA** will first assign Y's μop to port 1. It will then try to assign X, but it will only be able to assign the first μop to port 0 as port 1 is blocked. As discussed before X will therefore not be added to the execution list. In the second cycle the leftover of X will be assigned to a port and the execution will start. As the latency was two cycles the simulation will stop after the third cycle.
Again this is an overestimation of the reality. Due to the dependency of the

second μop of X it does not matter that Y blocks port 1 in the first cycle. X only needs port 0 in the first cycle and in the second cycle it can then freely use port 1. Again the "real latency" of our example would be two cycles.

Finally we will construct an example were our simulation actually underestimates the throughput. We can once more use our two instructions X and Y. This time we assume that Y cannot be executed in the first cycle, because of a dependency on an arbitrary third instruction. Our simulation basically works like in our first case, except for the reason why Y cannot be executed. So it will compute a latency of three cycles for those two instructions.
In reality Y will be denied port 1 in the second cycle as it will be used by X. So the execution will start in the third cycle and end in the fourth.

Note that it is still impossible that the latency of a single instruction is underestimated as we will always simulate the execution of at least as many cycles as we measured under a best case scenario. Also an important detail is, that it is impossible for an instruction to block itself. So if multiple μops of a single instruction need to use the same port this will not cause a delay. This is again due to the measurements as the delay caused by "inner instructional port blockings" is already included in the best case runtime. We did not include this in the pseudo code above for the sake of readability.

Ultimately we will consider the rest of the algorithm starting in line 24. This part will increase the counters similarly to what we have seen at the start of the algorithm. Notable here is the function *blocked_ports(Instruction)* that will return all ports that have been blocked during the assignment phase as well as the *Blamed* set which ensures that every instructions is held responsible at most once. We need this as we want to count the number of cycles that an instruction caused a delay and not the number of blocked ports in a particular cycle.

3.4 Executing Applicable Instructions

Algorithm 6: Execute applicable instructions

```
1  Function execute_instructions(instruction_queue queue):
2      foreach I ∈ Executionlist do
3          I.executed_cycles++;
4          if I.executed_cycles = I.latency then
5              queue.remove(I);
6          end
7          inform_children_im_done(I);
8      end
9      Executionlist.clear();
10 end
```

This part is rather simple. Every instruction knows its latency and how many cycles it has been executed. This value gets increased and if the latency is hit it will be removed from the instructionqueue. The most interesting part here is the function *inform_children_im_done(Instruction)*. As we have seen in Section 1.5 the children of an instruction do not necessarily have to wait for the instruction to finish. Sometimes they only need part of the results, which are available earlier. So this function will iterate over all children and check if the execution is advanced enough and if so "release the dependency" in a way that the *all_dependencies_resolved()* function in Algorithm 5 will consider the instruction as finished. Finally we have to clear the execution list as we will fill it again in the next cycle.

3.5 Performing a Cycle

Lastly we want to briefly discuss how a whole cycle is performed. **SUACA** will run each of the three simulation algorithms we explained above. It will then free up all ports that were used during the cycle, in order to enable the port pipelining. It also passes the queue that contains all instructions that still have to be executed to the three functions. A short pseudo code representation can be found below.

Algorithm 7: Perform a whole cycle

```
1 Function perform_cycle(instruction_queue queue):
2 |   load_instructions(queue);
3 |   choose_ports(queue);
4 |   execute_instructions(queue);
5 |   foreach p ∈ Ports do
6 |   |   p.clear();
7 |   end
8 end
```

The interesting observation here is that an instruction can actually get loaded, put into a port and then executed within a single cycle, due to the order of the function calls.
This function will be executed in a loop as long as there are instructions to be executed. After said loop **SUACA** will generate its output.

3.6 The Divider Pipe

Instructions that perform a division of some kind have to use the divider pipe, which is located on port 0. **SUACA**'s output will only show the divider pipe if one of the instructions needs it. We have to consider it when choosing the ports for the instructions during algorithm 5, because the division μops cannot be pipelined. More precisely our measurements contain a "div-cycle" property for the corresponding instructions, which tells us how many cycles the divider pipe will be blocked. Because the divider pipe is located on port 0 each of those instructions has to have at least one μop that uses port 0 exclusively. **SUACA** will block the divider pipe as soon as this particular μop is assigned to port 0. As long as it is blocked all future division μops are denied port 0. After "div-cycle" many cycles the divider pipe will be available again.
We did not include this in the algorithms above for two reasons. First is readability, as the implementation of this behavior would add some otherwise unnecessary if statements. On the other hand this is a very special case as using a division is definitely not advised when you are trying to write high performance code.

4 Evaluation

In this chapter we are going to compare **SUACA** to both **IACA** and **OSACA**. We will evaluate the results and differences of the tools and show how a detailed analysis can be done with **SUACA**.

4.1 Bottleneck Analysis

Consider the following example run of **IACA** 2.3 on the Sandy Bridge architecture:

```
Throughput Analysis Report
--------------------------
Block Throughput: 2.00 Cycles       Throughput Bottleneck: FrontEnd

Port Binding In Cycles Per Iteration:
---------------------------------------------------------------------------
|  Port  |  0   -  DV  |  1   |  2   -  D   |  3   -  D   |  4   |  5   |
---------------------------------------------------------------------------
| Cycles | 2.0    0.0  | 2.0  | 0.0    0.0  | 0.0    0.0  | 0.0  | 2.0  |
---------------------------------------------------------------------------

N - port number or number of cycles resource conflict caused delay, DV - Divider pipe (on port 0)
D - Data fetch pipe (on ports 2 and 3), CP - on a critical path
F - Macro Fusion with the previous instruction occurred
* - instruction micro-ops not bound to a port
^ - Micro Fusion happened
# - ESP Tracking sync uop was issued
@ - SSE instruction followed an AVX256/AVX512 instruction, dozens of cycles penalty is expected
X - instruction not supported, was not accounted in Analysis

| Num Of |             Ports pressure in cycles              |    |
|  Uops  |  0  - DV  |  1  |  2  -  D  |  3  -  D  |  4  |  5  |    |
---------------------------------------------------------------------
|   1    | 1.0       |     |           |           |     |     | CP | mov rax, 0x6
|   1    |           | 1.0 |           |           |     |     | CP | mov rax, 0x6
|   1    |           |     |           |           |     | 1.0 | CP | mov rax, 0x6
|   1    | 1.0       |     |           |           |     |     | CP | mov rax, 0x6
|   1    |           | 1.0 |           |           |     |     | CP | mov rax, 0x6
|   1    |           |     |           |           |     | 1.0 | CP | mov rax, 0x6
Total Num Of Uops: 6
```

Note that we constructed this program exclusively for our purposes. Apart from the very first instruction the program will not have any effect. This is fine as it is designed as a toy example.
We can observe that the *mov* instruction is able to use ports 0, 1 and 5. The

front-end of the Sandy Bridge architecture can deliver up to four μops per cycle and there are no dependencies as no register is ever read.
So the clear bottleneck of this program are the three ports, because four μops get loaded but only three ports are available for their execution. Unfortunately we do not know how the bottleneck analysis of **IACA** works so we cannot argue why it believes that the front-end might be the bottleneck. Looking at the information we do have it does not make sense, though.

Now consider **SUACA**'s output of the same program with 200 iterations:

```
Block throughput: 2.00 cycles
Block throughput with perfect front-end: 2.00 cycles
Block throughput with infinitely usable ports: 1.50 cycles
Block throughput without dependencies: 2.00 cycles
Microops per cycle: 3.00

Analysis for architecture: SNB

 Line  ||   Num   ||   had   || caused  ||            Used Ports
       ||   Uops  || to wait || to wait ||   0   ||   1   ||   2   ||   3   ||   4   ||   5   ||
 ------------------------------------------------------------------------------------------------
   0   ||    1    ||  15.6   ||  46.7   ||  1.0  ||       ||       ||       ||       ||       || mov rax, 0x6
   1   ||    1    ||  15.6   ||  46.7   ||       ||  1.0  ||       ||       ||       ||       || mov rax, 0x6
   2   ||    1    ||  15.5   ||  46.7   ||       ||       ||       ||       ||       ||  1.0  || mov rax, 0x6
   3   ||    1    ||  15.6   ||  46.7   ||  1.0  ||       ||       ||       ||       ||       || mov rax, 0x6
   4   ||    1    ||  15.6   ||  46.7   ||       ||  1.0  ||       ||       ||       ||       || mov rax, 0x6
   5   ||    1    ||  15.6   ||  46.7   ||       ||       ||       ||       ||       ||  1.0  || mov rax, 0x6
Total number of Uops: 6
```

One can easily see that the port information and the block throughput is identical to **IACA**'s output. The major difference lies in the bottleneck analysis. **SUACA** tells you that the throughput improves quite drastically with infinitely usable ports. The two cycles make sense with the ports in consideration as we have three usable ports and six instructions, but when each port can be used to an infinite amount each cycle this is no longer relevant. Without the port problem we only need one and a half cycle for each iteration as we have six μops and a front-end that produces four of them each cycle.
We can also tell **SUACA** to analyze a specific line for us:

```
Detailed delay information for instruction: mov rax, 0x6 in line 0

        Maximum latency: 1

        Latencies for dependencies:
        This instruction doesn't have any dependencies!

        Delay caused by blocked ports:
         Port || was delayed || has delayed
         ----------------------------------
          0   ||    46.7     ||    15.6
          1   ||     0.0     ||    15.6
          5   ||     0.0     ||    15.6
```

This detailed analysis shows that each line suffered a delay of 15.6 cycles from each of its usable ports. This value is equal for all ports, because

this instruction only consists of a single μop and is therefore only delayed if all three ports are blocked (see algorithm 5), which will then all be held responsible. As this program does not contain any dependencies, this is also why the delay suffered from the ports equals the *had to wait* value of the original output. Additionally each instruction causes a delay on the port it used.
The *caused to wait* value in the original output is thrice as high as the *had to wait* value. This is due to the fact that there are always three instructions that are responsible for a single other instruction's delay.

Unfortunately, **OSACA** does not offer a bottleneck analysis which makes this comparison obsolete.

4.2 Complete Analysis

Now we are going to analyze the example provided by the **OSACA** thesis [6]. First consider the underlaying *C* code which represents a 2D-5pt stencil:

```
for(j = 1; j < M-1; ++j){
    #pragma vector aligned
    for(int i = 1; i < N-1; ++i){
        IACA_START
        b[j][i] = (a[j][i-1] + a[j][i+1] + a[j-1][i] + a[j+1][i]) * s;
    }
    IACA_END
}
```

In the following we will analyze the resulting machine code and compare the results of **IACA**, **OSACA** and **SUACA**. We will do so with the Ivy Bridge microarchitecture in mind. First consider **OSACA**'s analysis in Example 1. We can observe a different approach to the port bindings here. **OSACA** always tries to distribute the port pressure evenly across all ports to an extent where it does not take previous instructions into account. This basically means that every instruction will always have the same port bindings no matter how the rest of the program looks like. The idea is to give the user more information about possible bindings which are hard to observe when using **IACA**. However, this leads to an overestimation of the throughput as the pressure on port 1 is much higher than it needs to be. The *incq* and *cmpq* instructions do not need to use port 1 at all as they could instead use port 0 and even more so the very underused port 5.

```
Throughput Analysis Report
--------------------------
X - No information for this instruction in data file
" - Instruction micro-ops not bound to a port

Port Binding in Cycles Per Iteration:
-------------------------------------------------
| Port   |   0  |   1  |  2  |  3  |  4  |   5  |
-------------------------------------------------
| Cycles | 1.67 | 3.67 | 2.5 | 2.5 | 1.0 | 0.67 |
-------------------------------------------------

Ports Pressure in cycles
|   0  |   1  |   2  |   3  |   4  |   5  |
-----------------------------------------
|      |      | 0.50 | 0.50 |      |      | vmovsd  (%r14,%r15,8), %xmm2
|      | 1.00 | 0.50 | 0.50 |      |      | vaddsd  16(%r14,%r15,8), %xmm2, %xmm3
|      | 1.00 | 0.50 | 0.50 |      |      | vaddsd  8(%rax,%r15,8), %xmm3, %xmm4
|      | 1.00 | 0.50 | 0.50 |      |      | vaddsd  8(%rdx,%r15,8), %xmm4, %xmm5
| 1.00 |      |      |      |      |      | vmulsd  %xmm5, %xmm1, %xmm6
|      |      | 0.50 | 0.50 | 1.00 |      | vmovsd  %xmm6, 8(%r12,%r15,8)
| 0.33 | 0.33 |      |      |      | 0.33 | incq    %r15
| 0.33 | 0.33 |      |      |      | 0.33 | cmpq    %r13, %r15
|      |      |      |      |      |      | jb      ..B1.17
Total number of estimated throughput: 4.67
```

Example 1: OSACA run

The **OSACA** thesis states, that we can conclude that port 1 is the bottleneck of this program, which makes sense looking at the given analysis. We will now look into the program a bit further.
Example 2 displays the output of **IACA** 2.3 which shows the expected behavior regarding the overused port 1 i.e., the *incq* and *cmpq* instructions exclusively use port 5.

```
Throughput Analysis Report
--------------------------
Block Throughput: 3.00 Cycles       Throughput Bottleneck: FrontEnd

Port Binding In Cycles Per Iteration:
---------------------------------------------------------------------------
|  Port  |  0   -  DV  |  1   |  2   -  D   |  3   -  D   |  4   |  5   |
---------------------------------------------------------------------------
| Cycles | 1.0    0.0  | 3.0  | 2.5    2.0  | 2.5    2.0  | 1.0  | 2.0  |
---------------------------------------------------------------------------


| Num Of |             Ports pressure in cycles              |    |
|  Uops  |  0  - DV  |  1   |  2  -  D  |  3  -  D  |  4   |  5   |    |
-----------------------------------------------------------------------
|   1    |           |      | 1.0   1.0 |           |      |      |    | vmovsd xmm2, qword ptr [r14+r15*8]
|   2    |           | 1.0  |           | 1.0   1.0 |      |      | CP | vaddsd xmm3, xmm2, qword ptr [r14+r15*8+0x10]
|   2    |           | 1.0  | 1.0   1.0 |           |      |      | CP | vaddsd xmm4, xmm3, qword ptr [rax+r15*8+0x8]
|   2    |           | 1.0  |           | 1.0   1.0 |      |      | CP | vaddsd xmm5, xmm4, qword ptr [rdx+r15*8+0x8]
|   1    | 1.0       |      |           |           |      |      |    | vmulsd xmm6, xmm1, xmm5
|   2    |           |      | 0.5       | 0.5       | 1.0  |      |    | vmovsd qword ptr [r12+r15*8+0x8], xmm6
|   1    |           |      |           |           |      | 1.0  |    | inc r15
|   1    |           |      |           |           |      | 1.0  |    | cmp r15, r13
Total Num Of Uops: 12
```

Example 2: IACA run

Two properties of this analysis remain unclear. On the one hand, we do not know why those two instructions do not use port 0 as well. It is a possible port for both of those and it is clearly pressured less. On the other hand the distribution of ports 2 and 3 are very strict for the first four instructions and are split for the 6^{th}. **IACA** may have some internal information that would explain this behavior, but as far as we know this seems odd.

IACA declares the front-end as the bottleneck. The front-end of the Ivy Bridge architecture delivers up to four μops per cycle. The example program contains 12 μops and the throughput is exactly three cycles. So the throughput is certainly lower bounded by the front-end. However, port 1 is still pressured for three cycles. So even if the front-end was faster the throughput would not improve as it is also lower bounded by the three *vaddsd* instructions that have to use port 1.
This analysis does not show the dependencies at all. If we take a closer look at the first six instructions we can see that those have a dependency chain over the *xmm* registers. We will now consider **SUACA**'s analysis in order to take a closer look at the interactions between front-end, ports and dependencies of our program. We shortened the instructions in the output for the sake of readability.

```
Block throughput: 3.00 cycles
Block throughput with perfect front-end: 3.00 cycles
Block throughput with infinitely usable ports: 3.00 cycles
Block throughput without dependencies: 3.00 cycles
Microops per cycle: 3.99

Analysis for architecture: IVB

 Line  ||   Num   ||   had   || caused  ||            Used Ports
       ||   Uops  || to wait || to wait ||   0   ||   1   ||   2   ||   3   ||   4   ||   5   ||
 ------------------------------------------------------------------------------------------------
   0   ||    1    ||   0.3   ||   1.3   ||       ||       ||  0.5  ||  0.5  ||       ||       || vmovsd xmm2, ...
   1   ||    2    ||   2.0   ||   5.0   ||       ||  1.0  ||  0.5  ||  0.5  ||       ||       || vaddsd xmm3, ...
   2   ||    2    ||   5.0   ||   7.7   ||       ||  1.0  ||  0.5  ||  0.5  ||       ||       || vaddsd xmm4, ...
   3   ||    2    ||   7.0   ||  10.3   ||       ||  1.0  ||  0.5  ||  0.5  ||       ||       || vaddsd xmm5, ...
   4   ||    1    ||  10.0   ||  14.0   ||  1.0  ||       ||       ||       ||       ||       || vmulsd xmm6, ...
   5   ||    2    ||  14.0   ||   0.3   ||       ||       ||  0.5  ||  0.5  ||  1.0  ||       || vmovsd qword ...
   6   ||    1    ||         ||   1.0   ||  0.2  ||       ||       ||       ||       ||  0.8  || inc r15
   7   ||    1    ||   1.0   ||         ||  0.3  ||       ||       ||       ||       ||  0.7  || cmp r15, r13
Total number of Uops: 12
```

Example 3: SUACA run - 3000 iterations

For the most part this looks pretty similar to the other two analyses. We can observe that the *incq* and *cmpq* instructions are now distributed over ports 0 and 5 in a way that both of those ports experience an equal pressure and that ports 2 and 3 are evenly used just like in Example 1.
The throughput is equal to **IACA**'s output and the throughput values of the special analyses do not differ, because of the properties we explained above. We will now use **SUACA**'s runtime options to gain a better picture of the program's problems. First consider the run with only a single iteration in Example 4. We can see that the throughput (which would also be the latency in this particular case) is drastically higher and more important, that the dependencies are the biggest offender. This is due to the long dependency chain from the first to the sixth instruction. So in a single iteration of this program almost every instruction has to wait for its predecessor to be finished.

```
Block throughput: 16.00 cycles
Block throughput with perfect front-end: 16.00 cycles
Block throughput with infinitely usable ports: 16.00 cycles
Block throughput without dependencies: 6.00 cycles
Microops per cycle: 0.75

Analysis for architecture: IVB

 Line  ||   Num   ||   had   || caused  ||            Used Ports
       ||   Uops  || to wait || to wait ||   0   ||   1   ||   2   ||   3   ||   4   ||   5   ||
 ------------------------------------------------------------------------------------------------
   0   ||    1    ||         ||   1.0   ||       ||       ||  1.0  ||       ||       ||       || vmovsd xmm2, ...
   1   ||    2    ||   1.0   ||   4.0   ||       ||  1.0  ||       ||  1.0  ||       ||       || vaddsd xmm3, ...
   2   ||    2    ||   4.0   ||   6.0   ||       ||  1.0  ||  1.0  ||       ||       ||       || vaddsd xmm4, ...
   3   ||    2    ||   6.0   ||   9.0   ||       ||  1.0  ||       ||  1.0  ||       ||       || vaddsd xmm5, ...
   4   ||    1    ||   9.0   ||  13.0   ||  1.0  ||       ||       ||       ||       ||       || vmulsd xmm6, ...
   5   ||    2    ||  13.0   ||         ||       ||       ||  1.0  ||       ||  1.0  ||       || vmovsd qword ...
   6   ||    1    ||         ||   1.0   ||  1.0  ||       ||       ||       ||       ||       || inc r15
   7   ||    1    ||   1.0   ||         ||       ||       ||       ||       ||       ||  1.0  || cmp r15, r13
Total number of Uops: 12
```

Example 4: SUACA run - single iteration

However, there is only a single dependency between consecutive iterations. The *inc* instructions writes to *r15* which will be read by the following iteration multiple times, so we can simply interpret this as the start of our "single iteration dependency chain". Those chains make sense as the program is iterating over an array.

When this program runs in a loop the different iterations will run "side by side", which explains why the throughput ultimately reaches a value that is lower bounded by the front-end and port 1. This is also the reason why we chose to run Example 3 with 3000 iterations as it takes a while until the average ultimately reaches this lower bound.

Now that we know which role the dependencies play in our program we can take a closer look at the front-end and the ports. In Example 5 one can see the analysis with 200 iterations which has several interesting results in comparison to Example 3.

```
Block throughput: 3.07 cycles
Block throughput with perfect front-end: 3.06 cycles
Block throughput with infinitely usable ports: 3.06 cycles
Block throughput without dependencies: 3.02 cycles
Microops per cycle: 3.91

Analysis for architecture: IVB

 Line  ||   Num   ||   had   || caused  ||            Used Ports
       ||   Uops  || to wait || to wait ||   0   ||   1   ||   2   ||   3   ||   4   ||   5   ||
 ------------------------------------------------------------------------------------------------
   0   ||    1    ||   0.3   ||   1.3   ||       ||       ||  0.5  ||  0.5  ||       ||       || vmovsd xmm2, ...
   1   ||    2    ||   2.0   ||   5.0   ||       ||  1.0  ||  0.5  ||  0.5  ||       ||       || vaddsd xmm3, ...
   2   ||    2    ||   5.0   ||   7.7   ||       ||  1.0  ||  0.5  ||  0.5  ||       ||       || vaddsd xmm4, ...
   3   ||    2    ||   7.0   ||  10.3   ||       ||  1.0  ||  0.5  ||  0.5  ||       ||       || vaddsd xmm5, ...
   4   ||    1    ||  10.0   ||  14.0   ||  1.0  ||       ||       ||       ||       ||       || vmulsd xmm6, ...
   5   ||    2    ||  14.0   ||   0.3   ||       ||       ||  0.5  ||  0.5  ||  1.0  ||       || vmovsd qword ...
   6   ||    1    ||         ||   1.0   ||  0.2  ||       ||       ||       ||       ||  0.8  || inc r15
   7   ||    1    ||   1.0   ||         ||  0.3  ||       ||       ||       ||       ||  0.7  || cmp r15, r13
Total number of Uops: 12
```

Example 5: SUACA run - 200 iterations

We can observe that the throughput values have not reached the 3.00 mark yet, like we discussed above. But more importantly the *had to wait* and *caused to wait* values did not change at all. Usually those increase when the

number of iterations is increased. This is the case when the front-end works faster than the execution, because our simulation will only count those values for the instructions that have been loaded into the scheduler already. So if a lot of instructions are waiting inside the scheduler the *had to wait* and *caused to wait* values will be very high. As this is clearly not the case here we can conclude that (after a few iterations) the scheduler is always filled to an equal amount. This actually supports **IACA**'s claim that the front-end is the bottleneck. We argued that, should the front-end's performance improve, the ports will prevent a faster execution.
In Example 6 we can see a run of the perfect front-end analysis. We used 3000 iterations for this and all the following example runs. The only difference are the two delay values which are a bit higher.

```
Block throughput: 3.00 cycles
Microops per cycle: 4.00

Analysis for architecture: IVB

 Line  ||   Num   ||   had   || caused  ||            Used Ports
       ||   Uops  || to wait || to wait ||   0   ||   1   ||   2   ||   3   ||   4   ||   5   ||
 ------------------------------------------------------------------------------------------------
   0   ||    1    ||   1.3   ||   1.7   ||       ||       ||  0.5  ||  0.5  ||       ||       || vmovsd xmm2, ...
   1   ||    2    ||  12.0   ||  18.3   ||       ||  1.0  ||  0.5  ||  0.5  ||       ||       || vaddsd xmm3, ...
   2   ||    2    ||  15.0   ||  20.3   ||       ||  1.0  ||  0.5  ||  0.5  ||       ||       || vaddsd xmm4, ...
   3   ||    2    ||  17.0   ||  24.6   ||       ||  1.0  ||  0.5  ||  0.5  ||       ||       || vaddsd xmm5, ...
   4   ||    1    ||  20.0   ||  24.7   ||  1.0  ||       ||       ||       ||       ||       || vmulsd xmm6, ...
   5   ||    2    ||  24.7   ||   1.3   ||       ||       ||  0.5  ||  0.5  ||  1.0  ||       || vmovsd qword ...
   6   ||    1    ||   0.0   ||   1.7   ||  0.2  ||       ||       ||       ||       ||  0.8  || inc r15
   7   ||    1    ||   1.0   ||         ||  0.3  ||       ||       ||       ||       ||  0.7  || cmp r15, r13
Total number of Uops: 12
```

Example 6: SUACA run - perfect front-end

The details of line three (Example 7) show that this is actually due to dependencies for the most part.

```
Maximum latency: 3

Latencies for dependencies:
 Line || 3 -> Line || Line -> 3
---------------------------------
  2   ||     0     ||     3
  4   ||     3     ||     0
  6   ||     0     ||     1


Delay caused by dependencies:
 Line || was delayed || has delayed
-----------------------------------
  2   ||     0.0     ||    17.0
  4   ||    20.0     ||     0.0
  6   ||     0.0     ||     0.0


Delay caused by blocked ports:
 Port || was delayed || has delayed
-----------------------------------
  1   ||     4.0     ||     0.0
  2   ||     0.3     ||     0.0
  3   ||     0.3     ||     0.0
```

Example 7: Details of line 3 with perfect front-end

In order to prove our port claim we will run **SUACA** with both a perfect front-end and disabled dependencies in Example 8. We can exactly observe

the above described behavior. The throughput does not improve and the three *vaddsd* instructions are heavily delayed by port 1, whereas all the other instructions experience very little to no delay. The reason why those delay values are so small is the massive delay of the *vaddsd* instructions as those will continue to be executed for so many cycles that the average delay values of the other instructions are neglected almost completely.

```
Block throughput: 3.00 cycles
Microops per cycle: 4.00

Analysis for architecture: IVB

 Line  ||   Num   ||   had   || caused  ||            Used Ports
       ||   Uops  || to wait || to wait ||   0   ||   1   ||   2   ||   3   ||   4   ||   5   ||
 ------------------------------------------------------------------------------------------------
   0   ||    1    ||   0.2   ||   1.5   ||       ||       ||  0.5  ||  0.5  ||       ||       || vmovsd xmm2, ...
   1   ||    2    ||  48.4   ||  50.9   ||       ||  1.0  ||  0.5  ||  0.5  ||       ||       || vaddsd xmm3, ...
   2   ||    2    ||  49.4   ||  48.9   ||       ||  1.0  ||  0.5  ||  0.5  ||       ||       || vaddsd xmm4, ...
   3   ||    2    ||  49.4   ||  49.9   ||       ||  1.0  ||  0.5  ||  0.5  ||       ||       || vaddsd xmm5, ...
   4   ||    1    ||   0.0   ||   0.0   ||  1.0  ||       ||       ||       ||       ||       || vmulsd xmm6, ...
   5   ||    2    ||   0.3   ||   0.5   ||       ||       ||  0.5  ||  0.5  ||  1.0  ||       || vmovsd qword ...
   6   ||    1    ||   0.0   ||   0.0   ||       ||       ||       ||       ||       ||  1.0  || inc r15
   7   ||    1    ||   0.0   ||   0.0   ||  1.0  ||       ||       ||       ||       ||  0.0  || cmp r15, r13
Total number of Uops: 12
```

Example 8: SUACA run with pf and nd

Finally Example 9 shows what happens when only the dependencies are taken into consideration and it displays a much higher throughput that is probably only bounded by the capacity of the scheduler (54 μops in Ivy Bridge). As discussed above the program basically has no loop dependencies which leads to the results we are seeing.

```
Block throughput: 1.50 cycles
Microops per cycle: 7.98

Analysis for architecture: IVB

 Line  ||   Num   ||   had   || caused  ||            Used Ports
       ||   Uops  || to wait || to wait ||   0   ||   1   ||   2   ||   3   ||   4   ||   5   ||
 ------------------------------------------------------------------------------------------------
   0   ||    1    ||   1.0   ||   2.0   ||       ||       ||  0.5  ||  0.5  ||       ||       || vmovsd xmm2, ...
   1   ||    2    ||   2.0   ||   4.5   ||       ||  1.0  ||  0.0  ||  1.0  ||       ||       || vaddsd xmm3, ...
   2   ||    2    ||   4.5   ||   7.5   ||       ||  1.0  ||  1.0  ||  0.0  ||       ||       || vaddsd xmm4, ...
   3   ||    2    ||   7.5   ||  10.0   ||       ||  1.0  ||  0.0  ||  1.0  ||       ||       || vaddsd xmm5, ...
   4   ||    1    ||  10.0   ||  14.5   ||  1.0  ||       ||       ||       ||       ||       || vmulsd xmm6, ...
   5   ||    2    ||  14.5   ||         ||       ||       ||  1.0  ||  0.0  ||  1.0  ||       || vmovsd qword ...
   6   ||    1    ||   0.0   ||   4.0   ||  0.5  ||       ||       ||       ||       ||  0.5  || inc r15
   7   ||    1    ||   1.0   ||         ||       ||       ||       ||       ||       ||  1.0  || cmp r15, r13
Total number of Uops: 12
```

Example 9: SUACA run with pf and ip

All in all we can conclude that this program is very tricky and takes some serious thought to fully understand its issues. **SUACA** delivers various tools to gain insight into it and using **SUACA** we were able to get a good idea of the program's performance.

4.3 Flag Dependencies

For our final example we will use **IACA** 3.0 and consider the following code snippet:

```
1       adc rax, 0x1
2       adc rbx, 0x1
3       adc rcx, 0x1
4       adc rdx, 0x1
5       adc r8,  0x1
6       adc r9,  0x1
7       adc r10, 0x1
8       adc r11, 0x1
```

At first this program seems to be free of dependencies. However, the *adc* instruction is the "add with carry" instruction which reads and writes to the *cf* bit (carry flag) of the *RFLAGS* register. So there actually is a huge dependency loop that prevents any parallel execution as well as instruction reordering. Figure 4.1 shows **SUACA**'s dependency graph for this program, which of course also includes a self loop for every instruction.
Unfortunately, we do not know which dependencies **IACA** 3.0 uses exactly for its computations, but as it computes a throughput of 3.95 cycles we can conclude that it definitely does not know about the *cf* bit. This also leads to the back-end being held responsible for being the bottleneck. The full output can be seen in Example 10. We used the Skylake microarchitecture for this example.

```
Throughput Analysis Report
--------------------------
Block Throughput: 3.95 Cycles       Throughput Bottleneck: Backend
Loop Count:  22
Port Binding In Cycles Per Iteration:
-------------------------------------------------------------------------------------------------
|  Port  |   0   - DV   |   1   |   2   -  D   |   3   -  D   |   4   |   5   |   6   |   7   |
-------------------------------------------------------------------------------------------------
| Cycles |  4.0     0.0  |  0.0  |  0.0     0.0 |  0.0     0.0 |  0.0  |  0.0  |  4.0  |  0.0  |
-------------------------------------------------------------------------------------------------


| Num Of   |                    Ports pressure in cycles                         |     |
|  Uops    |  0  - DV    |  1   |  2  -  D    |  3  -  D    |  4   |  5   |  6   |  7   |
---------------------------------------------------------------------------------------------
|   1      | 1.0         |      |             |             |      |      |      |      | adc rax, 0x1
|   1      |             |      |             |             |      |      | 1.0  |      | adc rbx, 0x1
|   1      | 1.0         |      |             |             |      |      |      |      | adc rcx, 0x1
|   1      |             |      |             |             |      |      | 1.0  |      | adc rdx, 0x1
|   1      | 1.0         |      |             |             |      |      |      |      | adc r8, 0x1
|   1      |             |      |             |             |      |      | 1.0  |      | adc r9, 0x1
|   1      | 1.0         |      |             |             |      |      |      |      | adc r10, 0x1
|   1      |             |      |             |             |      |      | 1.0  |      | adc r11, 0x1
Total Num Of Uops: 8
Analysis Notes:
Backend allocation was stalled due to unavailable allocation resources.
```

Example 10: IACA 3.0's output

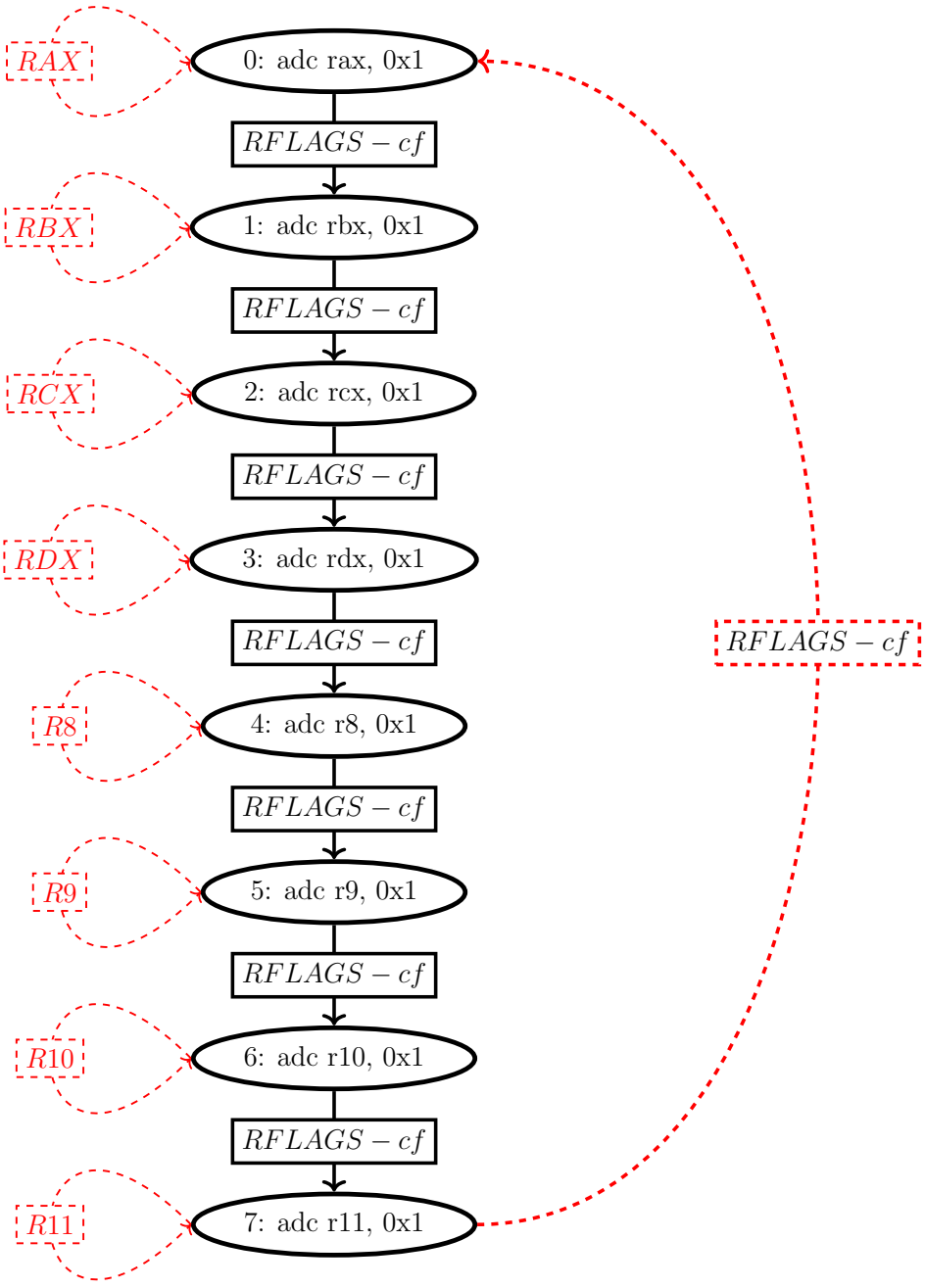

Figure 4.1: Dependency graph

As one can see in Example 11 **SUACA**'s results are identical for the port pressure, but as it does include the dependency loop it computes a throughput of 8.0 cycles. The throughput without dependencies is measured with 4.0 cycles which makes sense the example contains 8 μops and two usable ports. We have again cut the first two columns for the sake of readability.

```
Block throughput: 8.00 cycles
Block throughput with perfect front-end: 8.00 cycles
Block throughput with infinitely usable ports: 8.00 cycles
Block throughput without dependencies: 4.00 cycles
Microops per cycle: 1.00

Analysis for architecture: SKL

    had   || caused  ||           Used Ports
  to wait || to wait ||   0   ||   1   ||   2   ||   3   ||   4   ||   5   ||   6   ||   7   ||
 ---------------------------------------------------------------------------------------------------------
   92.2   || 177.5   ||  1.0  ||       ||       ||       ||       ||       ||       ||       || adc rax, 0x1
   92.3   || 177.6   ||       ||       ||       ||       ||       ||       ||  1.0  ||       || adc rbx, 0x1
   92.3   || 177.7   ||  1.0  ||       ||       ||       ||       ||       ||       ||       || adc rcx, 0x1
   92.4   || 177.8   ||       ||       ||       ||       ||       ||       ||  1.0  ||       || adc rdx, 0x1
   92.4   || 177.9   ||  1.0  ||       ||       ||       ||       ||       ||       ||       || adc r8, 0x1
   92.5   || 178.1   ||       ||       ||       ||       ||       ||       ||  1.0  ||       || adc r9, 0x1
   92.5   || 178.2   ||  1.0  ||       ||       ||       ||       ||       ||       ||       || adc r10, 0x1
   92.6   || 177.8   ||       ||       ||       ||       ||       ||       ||  1.0  ||       || adc r11, 0x1
Total number of Uops: 8
```

Example 11: SUACA's output for SKL with 200 iterations

This seems to be a problem introduced in **IACA** 3.0 as **IACA** 2.3 computes a throughput of 7.62 cycles and declares the dependency chains as the bottleneck.

5
Conclusion and Future Work

This work introduced **SUACA**, a tool that is able to compute the port bindings, latency and throughput of an *x86* assembly program. It will also give hints at the bottleneck as well as several ways to further investigate what the bottleneck might be. Read-After-Write dependencies between instructions are tracked for all register operands, including the suppressed ones and those that are used to access memory, and with respect to the control flow. It supports most currently available Intel microarchitectures and can easily be updated to support future ones.

One thing that **SUACA** does not support yet are macro and micro op fusions as well as the data fetch pipes of certain ports.
Macro op fusion usually "eliminates" the last jump instruction of a loop body by merging it into the prior instruction. This would probably require some hard coded information about the macro-fusible instructions of each microarchitecture.
IACA displays the data fetch pipes on (usually) ports 2 and 3. Those pipes are used when the respective instructions loads from memory. At the moment it remains unclear when and if those values have an impact on the throughput of a program. This is why we have not implemented it so far.
What could be considered is an improved simulation of the front-end as well as support for zero latency instructions. At the moment we are always loading the maximum number of μops into the scheduler. This is most certainly not perfectly accurate, but this improvement would require measurements of the front-end. Some instructions are so called "zero latency instructions" in *x86*. Those can be eliminated by the front-end and will therefore not affect the execution. So far **SUACA** executes them with a latency of a single cycle.
The simulation of *AMD* microarchitectures could also be considered, but it would require fundamental changes both to the measurements and **SUACA** itself.
The bottleneck analyses could also be extended. One example would be

increasing the amount of μops a certain port can handle each cycle instead of simply setting this amount to infinite for all ports.
In most microarchitectures a floating point operation followed by an integer operation on the same register (or vice versa) causes a so called "bypass delay". The exact delay depends on the specific architecture. At the time of writing **SUACA** does not consider these delays.

SUACA also has some limitations that either cannot be overcome without internal information or high additional effort. The most important one is precise information about the μops. We could remarkably improve our computations as we would eliminate the issues described in Section 3.3. Another integral part of our computations is the algorithm of the scheduler. We assumed a greedy algorithm, which might not be perfectly accurate. Implementing the correct algorithm would also bring our results closer to reality.
In some corner cases it would be possible to keep track of the memory during the dependency analysis. However, this is impossible most of the time, because a memory access usually uses a register and we would therefore need to know the actual values inside the registers. Since **SUACA** performs a static analysis, and therefore does not know the initial values of most registers, acknowledging memory dependencies becomes unattainable.

Bibliography

[1] Andreas Abel and Jan Reineke. “Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures”. Unpublished. 2018.

[2] Mark Charney. *X86 Encoder Decoder*. Apr. 13, 2018. URL: https://intelxed.github.io/ref-manual/index.html.

[3] Israel Hirsh and Gideon S. *IACA homepage*. June 4, 2018. URL: https://software.intel.com/en-us/articles/intel-architecture-code-analyzer.

[4] Israel Hirsh and Gideon S. *IACA userguide*. June 4, 2018. URL: https://software.intel.com/sites/default/files/managed/3d/23/intel-architecture-code-analyzer-3.0-users-guide.pdf.

[5] Johannes Hofmann. *ibench - Instruction Benchmarks*. 2017. URL: https://github.com/hofm/ibench.

[6] Jan Laukemann. *OSACA Thesis*. June 7, 2018. URL: https://github.com/RRZE-HPC/OSACA/blob/master/doc/Design_and_Implementation_For_a_Framework_Predicting_Instruction_Throughput.pdf.

[7] Jan Laukemann. *OSACA Website*. June 7, 2018. URL: https://github.com/RRZE-HPC/OSACA.