Professur für
Höchstleistungsrechnen
Friedrich-Alexander-Universität
Erlangen-Nürnberg

BACHELOR THESIS

Design and Implementation of a Framework for Predicting Instruction Throughput

Jan Laukemann

Erlangen, January 11, 2018

Examiner:	Prof. Dr. Gerhard Wellein
Advisor:	Julian Hammer

Eidesstattliche Erklärung / Statutory Declaration

Erlangen, January 11, 2018

Jan Laukemann

Zusammenfassung

Für das Aufstellen eines Performance Modells oder anderweitiger Optimierung von, speziell wissenschaftlicher, Software, ist eine Analyse der in-core Ausführungszeit meist unerlässlich. Diese kann je nach verwendeter Microarchitektur des Prozessors unterschiedlich ausfallen. Deshalb soll in dieser Arbeit ein architekturabhängiges Open-Source Analysetool *OSACA* (Open-Source Architecture Analyzer) implementiert werden, welches die Durchsatzanalyse des Intel-eigenen Tools *IACA* (Intel Archcitecture Code Analyzer) beherrscht. Es wird gezeigt, dass OSACA die durchschnittliche Belegung einzelner Ports in der Ausführungspipeline und eine gesamte Durchsatzanalyse beherrscht und zusätzlich das semi-automatische Messen von Durchsatz und Latenz einzelner Instruktion unterstützt.

Abstract

For creating performance models or the optimization of, mostly scientific, software, it is essential to run analyses for in-core execution time, which are dependent on the micro architecture of the chip. Therefore in this work we present a tool named *Open-Source Architecture Code Analyzer (OSACA)* which is meant to recreate the throughput analysis functionalities of *IACA* (Intel Architecture Code Analyzer). We can show, that OSACA predicts the average port pressure in the execution pipeline for specific instruction forms and the total port binding for identifying bottlenecks. Furthermore, it is able to semi-automatically measure throughput and latency values of instructions forms to integrate them in the database.

Acknowledgement

I would like to thank my advisors Julian Hammer and Georg Hager for always having time for my problems and discussions, their patient guidance and useful critiques, while having busy schedules as usual, especially Julian for sharing his experience in Python software projects and Georg for having a keen eye for details. I would also like to thank Johannes Hofmann for his support with ibench and the helpful discussions whenever possible.

CONTENTS

1 INTRODUCTION

In the last years in this century, computers not only became more powerful and faster, but in the same time the complexity of processor architectures grew tremendously. In order to optimize a modern program for a specific architecture, insight knowledge of the executing processor is almost indispensable. One way to predict the behavior of a CPU for a kernel is to create a *performance model*. They provide the user with information about memory traffic and execution time and allow to predict a bottleneck to see which component affects the application performance at most.

1.1 Motivation

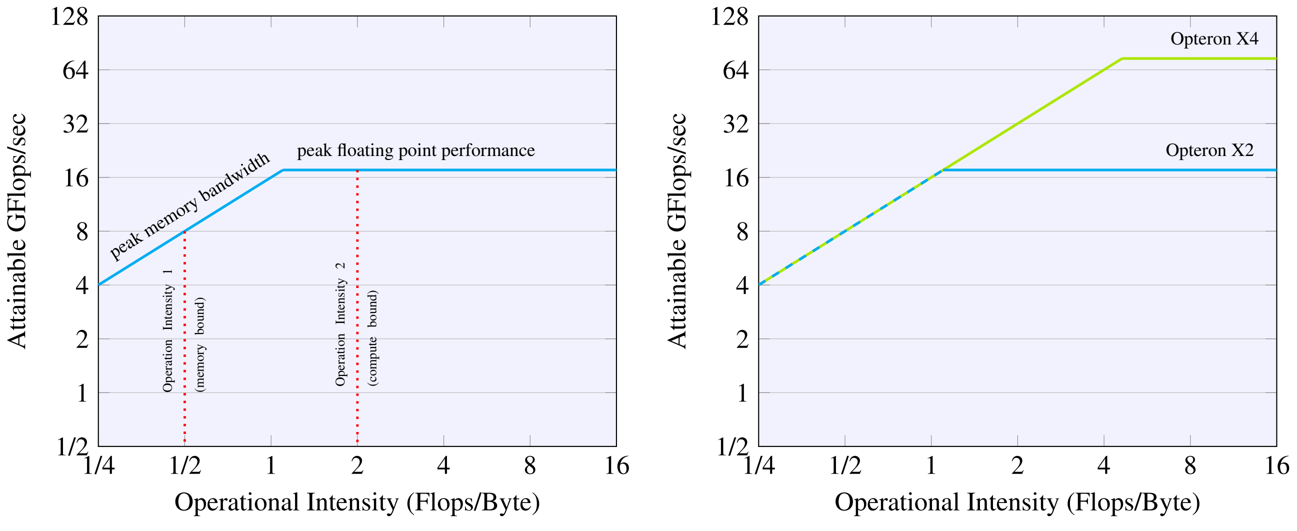
Most software in scientific computing is organized in loops. Therefore it is often useful to create a performance model of a loop kernel. To get an impression of the currently used ways for kernel analysis, two analytic performance models will be described briefly:

The *roofline model* [1] in its simplest form puts processor performance and off-chip memory traffic in a relation. For this the term *arithmetic intensity* is introduced. Arithmetic intensity or *operational intensity* can be described as the ratio of *work* W, which is the number of floating point operations executed in a certain kernel, and the *memory traffic* Q, which is the number of bytes of memory needed in the very same kernel. The arithmetic intensity I can then be written as [2]

$$I = \frac{W}{Q}$$

It is the number of operations per byte, thus, true arithmetic intensity is always dependent of both the chip architecture and the executed kernel [1]. The roofline model represents the maximum attainable floating point performance:

$$P = min \begin{cases} P_{peak} \\ \hline BW_{measured} \times I \end{cases}$$



(a) **(b)**

Figure 1.1: Roofline model for (a) AMD Opteron X2 and (b) Opteron X2 vs. Opteron X4 [1]

Its roof lines result from both the peak memory bandwidth $B_{measured}$ and the overall peak performance P_{peak} and determine the upper bounds of the model. Figure 1.1a shows an example of a standard roofline model. In the optimal case a program — represented as a single point in the graph — can be found on the roof line limited by the peak floating point performance, but normally a naive code kernel is bounded by another, not in the model plotted roof line. A simple way of identifying the current performance of a kernel is to analyze the optimal in-core throughput. Figure 1.1b outlines the difference between a 2.25 GHz AMD Opteron X2 with two cores and a 2.3 GHz AMD Opteron X4 with four cores. Due to the fact that they are placed on the same socket, both processors use the same DRAM interface and share the same peak memory bandwidth as upper bound. Nevertheless, the Opteron X4 can achieve more than four times the peak performance of the X2, because it not only has twice as many cores, but also can issue twice the number of SSE2 instructions per cycle [1].

A different, more complex approach for performance modeling is the *Execution-Cache-Memory (ECM) model* [3]. It takes the same input as the roofline model plus the data transfer times in the cache hierarchy and a more accurate in-core execution model. It then predicts execution time in CPU clock cycles. There are rules when parts of the core execution time overlaps with the transfer time as shown in [4] and the ECM model considers these additional information. The total model prediction for a single core can be described as

$$T_{ECM} = f(T_{nOL} + T_{data}, T_{OL}) \tag{1.1}$$

with T_{data} as the transfer time, T_{OL} as the part of the core execution that overlaps with the transfer time, and T_{nOL} as the part that does not [3]. For x86 microarchitectures till Intel Broadwell (BDW) function $f()$ can be replaced with $max()$ [4]. Therefore the model prediction for Intel BDW is the maximum of the overlapping time and the sum of all other contributions.

Assuming out-of-order execution within the execution units, no overlap between any data transfer up to L1 cache and perfect overlap between all other instructions and data transfer, a scaling limit, thus, the number of cores up to which the performance increases linear with the number of nodes, can be defined by:

$$n_s = \frac{\text{\# cycles per iteration overall}}{\text{\# cycles per iteration at the bottleneck}}$$

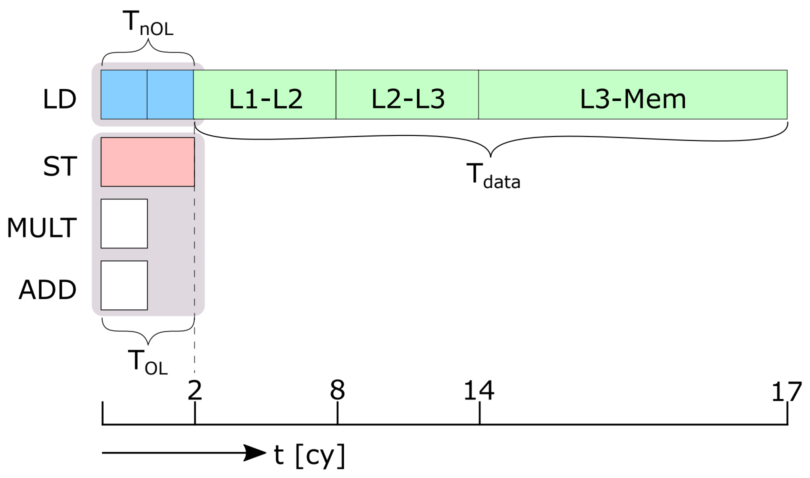


Figure 1.2: Single-core ECM model for the DAXPY loop kernel on Intel SNB using AVX [4].

For better understanding the ECM model will be applied to the DAXPY loop on an Intel Sandy Bridge (SNB) core:

```
for(i=0; i<N; ++i)
    a[i] = s * b[i] + a[i]
```

The loop body consists of two loads, one store, one multiply and one add instruction. This defines our work unit. A timeline diagram of the execution is shown in Figure 1.2. It is easy to see the bottleneck of DAXPY is in the load pipeline. The different contributions to the ECM model are written as $\{T_{OL} \parallel T_{nOL} | \, T_{L1L2} | \, T_{L2L3} | \, T_{L3Mem}\}$ which for the DAXPY would result in $\{2 \parallel 2 \mid 6 \mid 6 \mid 13\}$ cy. The ECM model on SNB is constructed according to Equation 1.1, so, e.g., the prediction for data out of L2 cache would be $T_{L1L2} = max(2, 2{+}6)$ cy $= 8$ cy [4]. Therefore the total prediction is $\{2 \rceil 6 \rceil 6 \rceil 13\}$ cy with "$\rceil$" as delimiter indicating the results of the algorithm as shown above for L2 cache. The model calculates the saturation assumption as:

$$n_s = \left\lceil \frac{T_{ECM}^{mem}}{T_{L3Mem}} \right\rceil = \left\lceil \frac{27}{13} \right\rceil = 3$$

Beyond the accurate calculation of data traffic, one crucial element of an analytic performance model is the in-core execution time, quantified by T_{OL} and T_{nOL}. In simple cases like the DAXPY loop above, this is easy to do by hand. However, due to the intricacy of the core architecture and the ubiquity of bottlenecks on the core level, complex loop bodies often evade manual analysis. Intel provides a free tool called Intel Architecture Code Analyzer (IACA) [5]. For a given, marked innermost loop body it provides an analysis of optimal instruction throughput and the binding of the instructions to the processor ports. It also identifies the critical path. IACA requires a piece of byte code to work on, which the user can provide by inserting special byte sequences, so-called *markers*, in the assembly code. This can be difficult especially for large code or if developers are not familiar with assembly. Optionally they may insert inline-assembly markers in high-level code, but this has proven to inhibit certain compiler optimizations as correct vectorization. Furthermore, as IACA is a tool owned by Intel, it only supports Intel Architectures, currently Sandy Bridge, Ivy Bridge, Haswell, Broadwell, Skylake and Skylake X. Unfortunately, Intel dropped its support for latency analysis in Version 2.2 (Dec 2016), so a user has no chance to get information about the latency of single instructions[1], therefore a detailed critical path analysis is not possible. Moreover, the future development path of IACA is unclear, as new versions are deployed irregularly and there is no official information about the prospective plans for the tool.

To overcome these issues and provide a solid basis for other tools such as Kerncraft [6], an open-source replacement for IACA should be developed. The goal of the present thesis is to create an initial implementation of Open Source Architecture Code Analyzer (OSACA) with the aim to support the throughput

[1]So called "micro-ops" or "μops" in Intel literature.

Listing 1.1: Evaluation output of the 20 most common instruction forms out of 163 in HPCG and STREAM benchmarks.

```
Number of       mnemonic
calls

1168            movslq          MEM(offset(base, index, scale))  GPR64
887             vmovsd          MEM((base, index, scale))  XMM128
620             vmovhpd         MEM((base, index, scale))  XMM128  XMM128
591             xor             GPR32  GPR32
565             mov             MEM(offset(base))  GPR64
543             add             IMD  GPR64
478             vmulsd          MEM((base, index, scale))  XMM128  XMM128
472             jb              LBL
431             mov             MEM(offset(base))  GPR32
424             cmp             GPR64  GPR64
419             movslq          MEM((base, index, scale))  GPR64
416             movslq          GPR32  GPR64
408             vsubsd          XMM128  XMM128  XMM128
353             mov             IMD  GPR32
337             vmovsd          MEM(offset(base))  XMM128
323             cmp             GPR32  GPR32
308             vinsertf128     IMD  XMM128  YMM256  YMM256
307             jne             LBL
270             jmpq            LBL
269             vaddsd          XMM128  XMM128  XMM128
```

analysis mode of IACA and provide a high level code marker option. Future work will extend OSACA to support various current CPU architectures as well as latency analysis mode.

1.2 Scope of Work

The scope of work can be separated into three main tasks: (i) The automatic extraction of relevant instructions using assembly or high level code markers, (ii) the automatic generation of benchmark files for measuring throughput and latency for a yet unknown instruction form, and (iii) the computation of the throughput analysis for a given kernel using the throughput values and manually inserted port occupations. Only Intel architectures are targeted for now, since these are very well documented and allow direct comparison with IACA reference results.

It is important to differentiate not only between single instructions, but also between the operands of instructions. For example an Intel Ivy Bridge (IVB) processor has a reciprocal throughput of $0.5\,\frac{\text{cy}}{\text{instr}}$ for `MOV r64,m64`, which equates to a load instruction, while `MOV m64, r64`, commonly known as store instruction, has a reciprocal throughput of $1\,\frac{\text{cy}}{\text{instr}}$. Therefore the term *instruction form* is introduced. An instruction form describes the combination of an assembly instruction mnemonic and its operand types in a specific order. This is particularly relevant on Intel architectures because of the intricacy of the instruction set and the use of the same mnemonic for many different operand types and orders.

Modern microprocessors have extremely large instruction sets. The detailed performance characteristics of individual instructions are often insufficiently documented for use in performance modelling, and even if they are, the data is not available in electronic (and parseable) form. In order to generate a set of relevant machine instructions that the tool OSACA should support, a semi-automatic “instruction profiling” approach was chosen: Benchmark programs or real applications are compiled and “typical” instruction forms are subsequently extracted from the assembly output. As an example, Listing 1.1 shows the output of the 20 most common instruction forms extracted from the profiler. Even though comparison and jump instructions can be found in the listing, they are not considered in the final analysis, because they normally carry no weight during a large number of loop iterations in an inner loop.

To extract the marked kernel, two different approaches were considered. As one possible option, a user can insert byte markers as known from IACA into the assembly code. For convenience OSACA supports the same byte marker as IACA, as presented below:

```
movl   $111, %ebx
.byte  100,103,144
    # ..LABEL:
        # Some code
        # ...
        # conditional jump to ..LABEL
movl   $222, %ebx
.byte  100,103,144
```

For this type of marking it is necessary to find the kernel in the assembly code, either by hand or by using Kerncraft. The Kerncraft IACA marker insertion function is integrated in OSACA; further information can be found in Section 1.3. In order to avoid handling assembly code, it is possible to insert a marker in the high level code. It is not advisable to insert inline-assembler, because additional bytes interfere with compiler optimizations such as SIMD vectorization and unrolling. Alternatively, the comment-based marker "`//STARTLOOP`" must be put one line before the loop header, and the loop code must be indented consistently. This means the marker and the head must have the same indentation level while the whole loop body needs to be more indented than the code before and after. The indentation is necessary, due to the fact the formatting of high-level code will be kept while interleaving it with assembly instructions. For instance, a valid OSACA marker can have a form as follows:

```
int i = 0;
while(i < M){
        int j = 0;
        //STARTLOOP
        while(j < N){
                //do work
        }
}
```

An automatic analysis of ELF files compiled with `-g` and also of assembly files is therefore possible.

After identifying the individual instruction forms, OSACA automatically generates an assembly benchmark file for missing instruction forms. For this, it must consider data dependencies and the necessary initialization of registers and memory addresses. Ibench [7], a tool for measuring instruction latency and throughput, which is discussed in Section 1.3, imports this benchmark test and measures the needed latency and throughput values. For valid results, the size of the loop for executing the instruction to be measured needs to be customizable, because instructions with a short execution time in a comparatively small loop body may lead to wrong values du to the overhead at the start and the end of the loop. Given this throughput and latency information as standard ibench output, OSACA reads the file and checks every measured value to add if it is reasonable.

Having a file with a marked kernel, OSACA extracts the relevant loop code and provides a throughput prediction for steady-state, throughput-limited execution with port bindings. By design, it assumes that all data for execution lies in the L1 cache and all instructions of the loop body are in the instruction cache. The associated data file of the corresponding core architecture not only contains the throughput and latency values, but also the average pressure of each operation per port, thus, the average amount of cycles the port is reserved for the instruction form during a steady-state execution. The instruction form `ADD xmm0, xmm1`, which has a reciprocal throughput of `1.00` cy can be executed in the port model on port 0, 1 or 5, would be assigned with a `0.33` on port 0, 1 and 5 and a `0` otherwise. Furthermore, the analysis contains the overall port binding per iteration through which one can pinpoint a possible bottleneck by checking the functions of the execution unit at the port with the most workload.

Finally the last task of this work was to make the code available for further collaboration and freely accessible as an open source project.

1.3 Related Work

General information about IACA by Israel Hirsh and Gideon S. can be found at the official IACA homepage [5]. The User's Guide [8] provides an overview of the installation of the tool, functionalities, supported platforms and processors and presents various examples.

Narayanan et al. [9] introduced a tool named *Pbound* for automatically generating an upper bound performance estimation. It further creates parametrized memory and computational metrics, as information about execution resources, from C/C++ source code for a simplified performance model [6]. The functionality of Pbound is entirely covered by Kerncraft and IACA/OSACA. It only provides a crude model for in-core runtime prediction based on benchmarks.

Johannes Hofmann [7] developed the *ibench* tool for measuring latency and maximum throughput of single assembly instructions embedded in a C program. OSACA relies on ibench as it provides the framework for integrating measured latency and throughput values.

Hammer et al. [6] describe the *Kerncraft* tool[2] for predicting the single-core performance and scaling behavior of loops using analytic performance models (Roofline or ECM model). It also provides the functionality to heuristically detect the innermost loop in an assembly file and automatically inserts IACA markers around it. Based on the IACA output and the data traffic analysis it then constructs analytic Roofline and ECM models. It is planned to replace IACA with OSACA in kerncraft as soon as OSACA is sufficiently mature.

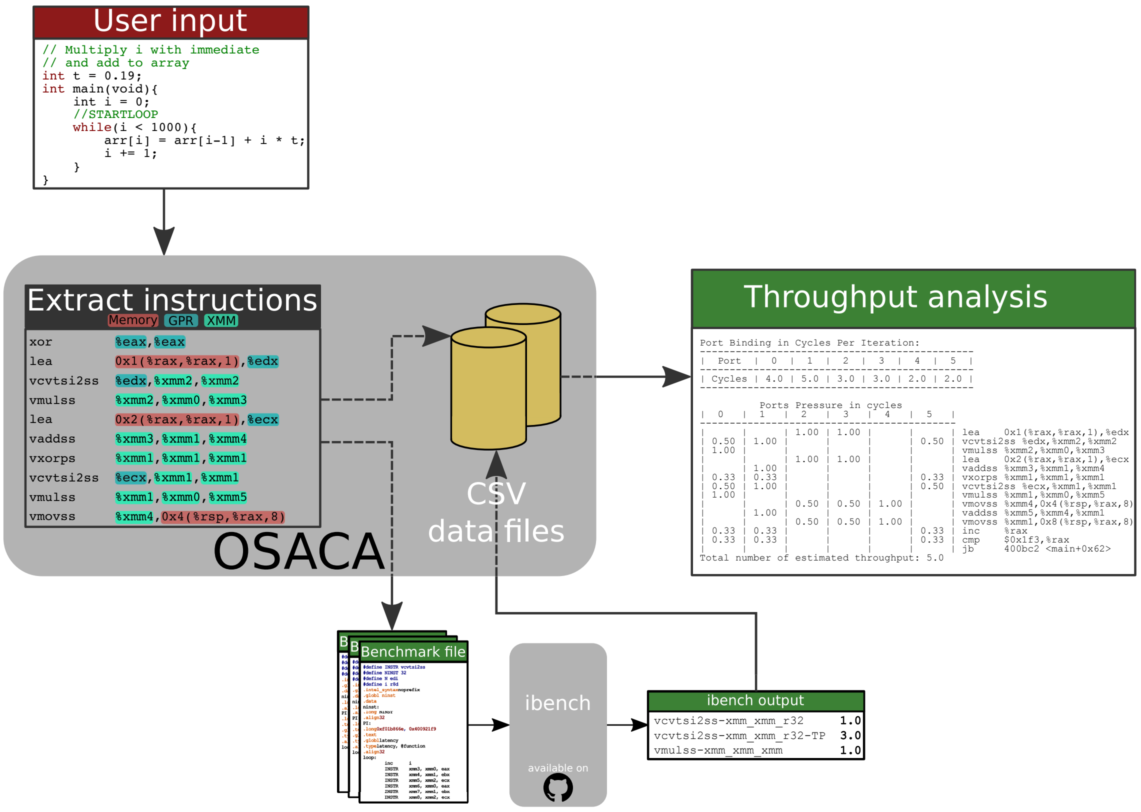
1.4 Results

A structural overview of the design of OSACA is presented in Figure 1.3. Given a marked kernel within source code, OSACA extracts the instructions form of the loop and analyzes the throughput by the provided information in its data file. In case of new instruction forms, OSACA automatically generates benchmark files which act as input for ibench runs. Subsequently OSACA uses the output of ibench for inserting the new measured throughput and latency values for the previously unknown instruction forms. It calculates the average throughput per active execution unit for each instruction and returns a tabular output on the terminal together with the overall port binding and a throughput estimation of the loop.

At the time of writing, OSACA supports the following functionalities:

- **Throughput analysis**
  As main functionality of OSACA this process starts by default. It is necessary to specify the core architecture by the flag `--arch ARCH`, where ARCH may be `SNB, IVB, HSW, BDW` or `SKL`. The optional flag `--iaca` defines if OSACA needs to search for the IACA byte markers or the OSACA marker in the chosen file. With an additional, optional `--tp-list`, OSACA adds a simple list of all kernel instruction forms together with their reciprocal throughput to the output. This is helpful if there is no further information about the port binding of a particular instruction form.
- **Including new measurements into the data file**
  Running OSACA with the flag `-i` or `--include-ibench` and a specified microarchitecture, it takes the values given in an ibench output file and checks them for sanity. If a value is not in the data file already, it will be added, otherwise OSACA prints out a warning message and keeps the old value in the data file. If a value does not pass the validation, a warning message is shown, however, OSACA will keep working with the new value. This is an advantage over IACA because it allows the user not only to enlarge the data files of OSACA, but even to create a base of information for a not yet implemented microarchitecture.
- **Inserting IACA markers** Using the `-m` or `--insert-marker` flags for a given file, OSACA calls the Kerncraft module for identifying and marking the inner-loop block. More information about how this is done can be found in Appendix C of [10].

[2]The current version of Kerncraft is available at `https://github.com/RRZE-HPC/Kerncraft`



```
// Multiply i with immediate
// and add to array
int t = 0.19;
int main(void){
    int i = 0;
    //STARTLOOP
    while(i < 1000){
        arr[i] = arr[i-1] + i * t;
        i += 1;
    }
}
```

```
xor         %eax,%eax
lea         0x1(%rax,%rax,1),%edx
vcvtsi2ss   %edx,%xmm2,%xmm2
vmulss      %xmm2,%xmm0,%xmm3
lea         0x2(%rax,%rax,1),%ecx
vaddss      %xmm3,%xmm1,%xmm4
vxorps      %xmm1,%xmm1,%xmm1
vcvtsi2ss   %ecx,%xmm1,%xmm1
vmulss      %xmm1,%xmm0,%xmm5
vmovss      %xmm4,0x4(%rsp,%rax,8)
```

```
Port Binding in Cycles Per Iteration:
-------------------------------------------------
|  Port  |  0  |  1  |  2  |  3  |  4  |  5  |
-------------------------------------------------
| Cycles | 4.0 | 5.0 | 3.0 | 3.0 | 2.0 | 2.0 |
-------------------------------------------------

          Ports Pressure in cycles
|   0   |   1   |   2   |   3   |   4   |   5   |
-------------------------------------------------
|       |       | 1.00  | 1.00  |       |       | lea    0x1(%rax,%rax,1),%edx
| 0.50  | 1.00  |       |       |       | 0.50  | vcvtsi2ss %edx,%xmm2,%xmm2
| 1.00  |       |       |       |       |       | vmulss %xmm2,%xmm0,%xmm3
|       |       | 1.00  | 1.00  |       |       | lea    0x2(%rax,%rax,1),%ecx
|       | 1.00  |       |       |       |       | vaddss %xmm3,%xmm1,%xmm4
| 0.33  | 0.33  |       |       |       | 0.33  | vxorps %xmm1,%xmm1,%xmm1
| 0.50  | 1.00  |       |       |       | 0.50  | vcvtsi2ss %ecx,%xmm1,%xmm1
| 1.00  |       |       |       |       |       | vmulss %xmm1,%xmm0,%xmm5
|       |       | 0.50  | 0.50  | 1.00  |       | vmovss %xmm4,0x4(%rsp,%rax,8)
|       | 1.00  |       |       |       |       | vaddss %xmm5,%xmm4,%xmm1
|       |       | 0.50  | 0.50  | 1.00  |       | vmovss %xmm1,0x8(%rsp,%rax,8)
| 0.33  | 0.33  |       |       |       | 0.33  | inc    %rax
| 0.33  | 0.33  |       |       |       | 0.33  | cmp    $0x1f3,%rax
|       |       |       |       |       |       | jb     400bc2 <main+0x62>
Total number of estimated throughput: 5.0
```

```
vcvtsi2ss-xmm_xmm_r32       1.0
vcvtsi2ss-xmm_xmm_r32-TP    3.0
vmulss-xmm_xmm_xmm          1.0
```

Figure 1.3: Structural design of OSACA.

For clarifying the functionality of OSACA a sample kernel is analyzed for an Intel IVB core.

```
double a[n], b[N];
double s;

//STARTLOOP
for(int i = 0; i < N; ++i)
        a[i] = s * b[i];
```

The example above shows a simple scalar multiplication of a vector b and a floating-point number s. The result is written to vector a. After including the OSACA marker "`//STARTLOOP`" and compiling the source, one can start the analysis typing `osaca --arch IVB path/to/file` in the command line. Optionally, one can create the assembly code out of the file, identify and mark the kernel of interest and run OSACA with the additional `--iaca` flag.

In Listing 1.2 the resulting output of the run can be seen. It shows the whole kernel together with the average port pressure and the overall port binding. While the actual loop kernel including `a[i] = s * b[i]` as well as the increment of the index variable and the compare and jump instruction is represented in lines 20–32 there are two more instructions in before, since the marker is placed before the for-loop and therefore the analyzed kernel consists also of the beginning loop control instructions. Especially for large loop bodies this behavior can be neglected.

OSACA estimates a block throughput of `6.0 cy` and most of the instructions are bound to port `2` and `3`, which lets us assume a bottleneck for load operations. Note that for now, no automatic port allocation of instruction forms is implemented and must be added in the data file by hand.

Listing 1.2: OSACA output for analysis of scalar multiplication kernel.

```
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 | 2.33 | 1.33 | 6.0 | 6.0 | 3.0 | 1.33 |
-------------------------------------------------


          Ports Pressure in cycles
|  0   |  1   |  2   |  3   |  4   |  5   |
-------------------------------------------
|      |      | 0.50 | 0.50 | 1.00 |      | movl   $0x0,-0x24(%rbp)
|      |      |      |      |      |      | jmp    10b <scale+0x10b>
|      |      | 0.50 | 0.50 |      |      | mov    -0x48(%rbp),%rax
|      |      | 0.50 | 0.50 |      |      | mov    -0x24(%rbp),%edx
| 0.33 | 0.33 |      |      |      | 0.33 | movslq %edx,%rdx
|      |      | 0.50 | 0.50 |      |      | vmovsd (%rax,%rdx,8),%xmm0
| 1.00 |      | 0.50 | 0.50 |      |      | vmulsd -0x50(%rbp),%xmm0,%xmm0
|      |      | 0.50 | 0.50 |      |      | mov    -0x38(%rbp),%rax
|      |      | 0.50 | 0.50 |      |      | mov    -0x24(%rbp),%edx
| 0.33 | 0.33 |      |      |      | 0.33 | movslq %edx,%rdx
|      |      | 0.50 | 0.50 | 1.00 |      | vmovsd %xmm0,(%rax,%rdx,8)
| 0.33 | 0.33 | 1.00 | 1.00 | 1.00 | 0.33 | addl   $0x1,-0x24(%rbp)
|      |      | 0.50 | 0.50 |      |      | mov    -0x24(%rbp),%eax
| 0.33 | 0.33 | 0.50 | 0.50 |      | 0.33 | cmp    -0x54(%rbp),%eax
|      |      |      |      |      |      | jl     e4 <scale+0xe4>
Total number of estimated throughput: 6.0
```

1.5 Outline

This thesis is organized as follows: In Chapter 2 a simplified core architecture, which forms the basis of OSACA, is described. Chapter 3 explains the general design and algorithmic details of the throughput analysis tool. Chapter 4 compares the results of OSACA to IACA and goes into detail about the differences between the two tools. In the final Chapter 5 further challenges and future work are discussed.

2 BACKGROUND

In this Chapter we give an short overview of modern microarchitectures and the simplified core architecture used in OSACA.

2.1 Modern Microarchitectures

Modern computer architectures are still exclusively based on the von Neumann architecture [12]. Many additions have been made, but the main components that we need to consider are: the *Arithmetic Logic Unit (ALU)* responsible for calculations, the *Control Unit (CU)* taking care of control flow of the program and the *memory* supplying the CU and ALU with input data and storing output data [10]. In the following section some of the most important functionalities in modern core architectures will be described.

Branch Prediction

Nowadays processors execute instructions in pipelined steps. Due to the distribution of execution, the overall throughput of instructions can be increased. This means also the right *branch prediction* of conditional program sequences gets more into focus, because the scheduling of not executed code leads to stalls in the pipeline and bad performance. Efficient code tries to avoid conditional branches and jumps, but at least in loops, which are ubiquitous in code, an end-of-loop conditional branch is indispensable. Here the prediction that a branch is going to be always taken can produce quite good results, because typically loops are iterated a large number of times until the program leaves the loop. Therefore, for performance analysis, the end-of-loop condition can be ignored.

Out-of-order Execution

Out-of-order execution denominates the ability of a microprocessor to execute instructions in a different order than their appearance in the machine code.

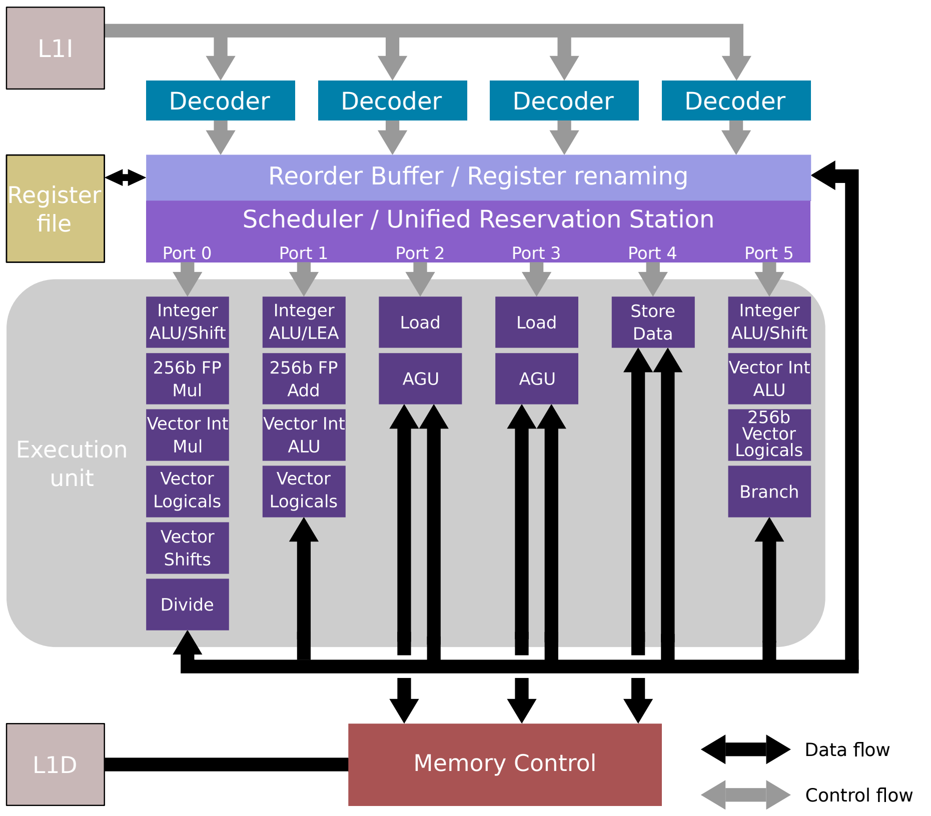


Figure 2.1: Execution pipeline and ports of a single Intel Sandy Bridge microarchitecture (based on [10] and [14]).

In case of otherwise forced stalls of the pipeline, e.g. if instruction operands can not made available in register within a needed time slot because of the memory hierarchy is too slow to keep up with processor speed, out-of-order execution can avoid these idle times. Thus, not only throughput can be improved but also it is easier for compilers to arrange machine code for optimal performance [13]. For instance, the Intel SNB architecture has six ports: Port `0`, `1` and `5` are responsible for arithmetic operations, `2` and `3` for loading data and address generation and port `4` handles store operations. An overview of this architecture is shown in Figure 2.1. The reorder, rename and schedule units allow an out-of-order execution by identifying the independence between operations and parallel queuing and assignment to the execution ports.

Macro-Op Fusion

In specific cases modern processors are able to merge two adjacent instructions to execute them within one instead of two cycles. This feature is called *macro-op fusion* or *macrofusion* and happens before decoding. Fused instructions can represent more work within less time, free up execution units, save pipeline bandwidth and consequently save power. A macrofusion can only happen once each cycle and is limited to a few defined combinations, thus: The first part of the macrofusion always has to be done on a flag-modifying instruction (e.g., `cmp` or `inc`) and second part has to be a conditional jump instruction. An overview of all macro-fusible instructions for Intel SNB is shown in Table 2.1.

2.2 Simplified Core Architecture

For throughput analysis, OSACA assumes a simplified core architecture: The underlying CPU is based on the port model as, e.g., in Figure 2.1. All instructions of the kernel to analyze fit in the instruction cache and all data used in the loop body come from the L1 cache. All predictions rely on a steady-state-execution,

Instructions	**TEST**	**AND**	**CMP**	**ADD**	**SUB**	**INC**	**DEC**
JO/JNO	✓	✓	✗	✗	✗	✗	✗
JC/JB/JAE/JNB	✓	✓	✓	✓	✓	✗	✗
JE/JZ/JNE/JNZ	✓	✓	✓	✓	✓	✓	✓
JNA/JBE/JA/JNBE	✓	✓	✓	✓	✓	✗	✗
JS/JNS/JP/JPE/JNP/JPO	✓	✓	✗	✗	✗	✗	✗
JL/JNGE/JGE/JNL/JLE/JNG/JG/JNLE	✓	✓	✓	✓	✓	✓	✓

Table 2.1: Macro-Fusible Instructions in Intel Microarchitecture Code Name Sandy Bridge [15].

thus, there is no warm-up effect. It is also assumed that the loop has enough iterations so that startup and wind-down effects can be neglected. Since OSACA only analyzes the loop body, we can assume every end-of-loop branch is taken, hence, there is a perfect branch prediction and the reciprocal throughput and latency value of every jump operation is assigned with 0 cy. As a consequence, OSACA does not consider explicit macrofusion. Finally the CPU can handle out-of-order execution, also across multiple loop iterations.

3 IMPLEMENTATION

This chapter details the implementation of the *Open Source Architecture Code Analyzer* (OSACA) tool. The source code and example inputs can be found on GitHub[1] and are licensed under the GNU Affero General Public License version 3 (AGPLv3) [16]. The command `osaca` allows access to the throughput analysis and all other functionalities. For any throughput analysis a file with the marked kernel and a data file structured as CSV containing the port occupation and the measured throughput and latency values of the instruction forms for a specific microarchitecture, as well as a constant defining the microarchitecture for the current run (e.g. `"SKL"`) need to be provided. The usage and all command line arguments are explained in Section 3.5

3.1 Structure

Internally OSACA consists of the following components: The command line interface (CLI), the identification and instruction fetching of the marked kernel, the generation and integration of ibench input and output files for measuring throughput and latency of instruction forms and the computation of the port pressure for the given kernel. The CLI is provided by the `osaca.py` program, the main entry point for users. The identification of the marked kernel and the fetching and generalization of each single instruction form for further analysis is also done by `osaca.py`, as well as the integration of measured execution times of instruction forms provided by ibench into the data file. The generation of benchmark loop files as input for ibench takes place in `testcase.py`. Finally, `eu_sched.py` provides the scheduling of the instruction forms to calculate the port pressure and the command line output.

Additionally OSACA accesses to the `iaca.py` module from Kerncraft to provide the functionality to heuristically detect the innermost loop and insert IACA markers. For identifying the correct architecture-dependent register ressources and the generalization of instruction operands, `param.py` provides the needed structure. It consists of the superclass `Parameter` and its subclasses `Register` and `MemAddr`.

[1]https://github.com/RRZE-HPC/OSACA

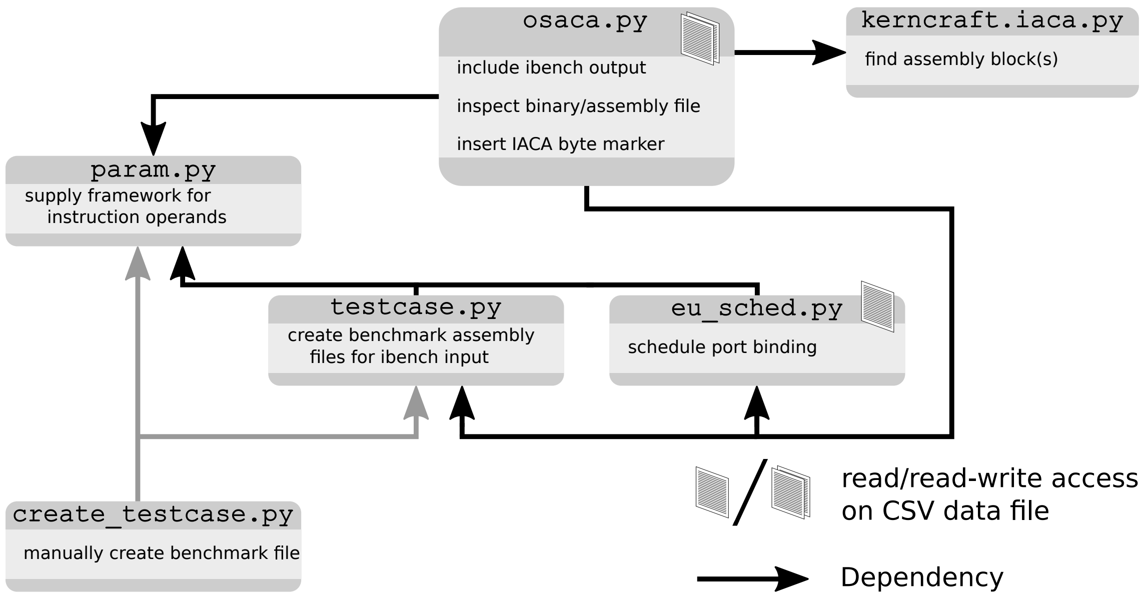

osaca.py
include ibench output
inspect binary/assembly file
insert IACA byte marker
kerncraft.iaca.py
find assembly block(s)
param.py
supply framework for instruction operands
testcase.py
create benchmark assembly files for ibench input
eu_sched.py
schedule port binding
create_testcase.py
manually create benchmark file
read/read-write access on CSV data file
Dependency

Figure 3.1: Internal structure of the packages of OSACA. Gray arrows represent dependencies from outside the OSACA package.

A general overview of the internal structure and dependencies of OSACA can be found in Figure 3.1. The file `create_testcase.py` is not callable by the main program and represents a template for manually creating benchmark assembly files by the user if needed.

3.2 Throughput Analysis

For the throughput analysis OSACA needs either an assembly file or an Executable and Linking Format (ELF) file [17], which includes executables or (shared) objects. In the latter case, for identifying the OSACA marker, the source must be compiled with the compiler flag `-g` for additional debugging information. If the kernel is marked with the IACA byte markers, nothing else here is required. Furthermore, for printing the throughput analysis, the program needs an architecture-specific CSV file including for each instruction form a throughput and latency value and the average port pressure.

For extracting the OSACA-marked kernel, `osaca.py` goes through the interleaved assembly and original source code, generated by `objdump`, and searches for the start marker. An extract of the interleaved code for the scalar multiplication kernel from Section 1.4 can be seen in Listing 3.1. The interleaved high level code is colored in blue, while editorial comments are in red.

Because of the same indentation of high level code in the original file and objdump output, it is important that all code of the loop body is more indented than the outer code and that the OSACA marker has the same indentation as the loop head. It cannot be assumed that objdump prints the whole loop block as consecutive text, thus, a complete iteration through the file is necessary.

If IACA byte markers are used, the search for the start and end marker is done using regular expressions.

Listing 3.1: Extract of the "`objdump -S`" output of the scalar multiplication kernel in Section 1.4. For brevity, the loop body is shown as compact code in this Listing.

```
  d7:   48 89 45 b8             mov    %rax,-0x48(%rbp)
    double s;

    //STARTLOOP
    for(int i=0; i<N; ++i){
  db:   c7 45 dc 00 00 00 00    movl   $0x0,-0x24(%rbp)
  e2:   eb 27                   jmp    10b <scale+0x10b>
            a[i] = s * b[i];
  e4:   48 8b 45 b8             mov    -0x48(%rbp),%rax
### loop body
 102:   c5 fb 11 04 d0          vmovsd %xmm0,(%rax,%rdx,8)
    for(int i=0; i<N; ++i){
 107:   83 45 dc 01             addl   $0x1,-0x24(%rbp)
 10b:   8b 45 dc                mov    -0x24(%rbp),%eax
 10e:   3b 45 ac                cmp    -0x54(%rbp),%eax
 111:   7c d1                   jl     e4 <scale+0xe4>
 113:   48 89 cc                mov    %rcx,%rsp
    }
}
 116:   90                      nop
```

Every line of assembly is inspected and the operands of the instructions are generalized for comparing the instruction form with the entries in the data file. For creating the throughput analysis, the instruction forms are given to `eu_sched.py`, which looks up the values for the given instruction form in the data file. If no port binding is found, even when there is a throughput value stored in the data file, the line with the related instruction form is left blank. In this case the user can get an additional output of the throughput values of all instruction forms occurring in the kernel to manually inspect the values and predict manually the port pressure of the instruction forms that miss information. After the insertion of port binding information, another OSACA analysis will provide the throughput information of the instruction form with the recently added port binding in the output.

3.3 Generating and Including Measurements

To ensure a convenient throughput analysis, OSACA provides a set of instruction forms in the architecture-dependent data files, which will be extended over time. For setting up an initial set of instruction forms, a program for automatically compiling and instruction form fetching from common benchmarks was created. Since we want to focus on applications from scientific computing, two application benchmarks were selected to start with: HPCG [18], which is a preconditioned CG solver and now recognized as complementing the LINPACK benchmark, and STREAM [19] for memory bandwidth. A profiling run of HPCG with gprof[2] showed that 95 % of the total runtime is executed in the methods `ComputeSYMGS` (66 %) and `ComputeSPMV` (29 %), which are responsible for the symmetric Gauss-Seidel method and the sparse matrix vector multiplication[3]. Therefore only these two methods were evaluated. The whole STREAM benchmark consists only of one file, so in that case the complete benchmark was considered for evaluation. To get a high variety in instruction forms both benchmarks were compiled with different compilers and different compiler flags. A summary of all options can be found in Table 3.1. The flag `-g` was always used throughout compiling to include debugging information in the generated binary.

If an instruction form is not found in the data file while fetching the instruction forms out of the marked kernel and if no benchmark file already exists, OSACA creates a test case which functions as ibench input. As a result, the instruction form is marked with an "`X`" in the output.

[2]More information in [20]

[3] Profiling was done for a 60 seconds HPCG run on a single Intel Xeon E5-2660 v2 @ 2.2 GHz (IVB) node with a problem size of `NX=NY=NZ=104`.

Benchmark	Compiler	Version	Compiler Flags	
HPCG STREAM	gcc	{4.9.3 | 5.4.0 | 6.1.0}	`-fargument-noalias -fopenmp {-O0|-O1|-O2|-O3}`	`-march={sandybridge| ivybridge|haswell| broadwell|skylake}`
	icc	{2013SP1.3 | 2016.3 | 2017.1}	`-fno-alias -fopenmp {-O0|-O1|-O2|-O3}`	`{-xAVX|-xCORE-AVX-I| -xCORE-AVX2|-xCORE-AVX512}`
	clang	{3.6 | 3.8}	`-fno-assume-sane-operator-new -fopenmp {-O0|-O1|-O2|-O3}`	

Table 3.1: The different compilers and compiler flags used for building the HPCG and STREAM benchmarks. The operator "|" in this context is used as *exlusive or*.

The full generation of the benchmark file is done by `testcase.py`. It inspects the given instruction form and provides both a throughput and a latency benchmark file for measurements. Depending on the operands of the instruction, it allocates memory and prepares general purpose or SIMD registers for use. To avoid invalid results for instructions with a short execution time, for which the execution of the loop control sequences may falsify the measurements, the number of instructions per iteration must be configurable. By default OSACA creates benchmark files with 32 instructions in the loop body.

For instance, in Listing 3.2, which shows the scalar multiplication example already seen in Section 1.4, in the fifth to last line containing `addl $0x1, -0x24(%rbp)` one can see an "X" in front of the instruction form and no port occupation. This means either there are no measured values for this instruction form or no port binding is provided in the data file. OSACA then automatically creates two benchmark assembly files (`add-mem_imd.S` for latency and `add-mem_imd-TP.S` for throughput) in the benchmark folder, if they not already exist there.

With the given file one can now run ibench to get the throughput value for `addl` with a memory address and an immediate as operands. Mind that the assembly file, which is used for ibench, is written in Intel syntax. So for a valid run instruction "`addl`" must be changed to "`add`" manually, because "`addl`" is only a correct mnemonic for AT&T syntax. Executing ibench on a IVB core may give an output like this:

```
Using frequency 2.20GHz.
add-mem_imd-TP: 1.023 (clock cycles)    [DEBUG - result: 1.000000]
add-mem_imd:    6.050 (clock cycles)    [DEBUG - result: 1.000000]
```

The debug output as resulting value of register `xmm0` is an additional validation information depending on the executed instruction form meant for the user and is not considered by OSACA. The ibench output information can be included by OSACA running the program with the flag `--include-ibench` or just `-i` and the specific microarchitecture defined by `--arch`. For now, no automatic port allocation of instruction forms is implemented, so for consideration in the ports pressure table, one must add the port occupation by hand. After doing this, knowing that the inserted instruction form must be assigned always to Port `2`, `3` and `4` and additionally to either `0`, `1` or `5`[4] [11], another OSACA run returns an output shown in Listing 3.3.

If one wants to measure the throughput and latency of an specific instruction form without marking a kernel, the template `create_testcase.py` is provided in the git repository. In this a user can type in the instruction of interest, its operands and the number of instructions per loop to manually create a benchmark file for throughput and latency measurements via ibench.

[4] A valid port assignment in the CSV data file would look like: `addl-mem_imd,1.0,6.0,"(0.33,0.33,1.00,1.00,1.00,0.33)"`

Listing 3.2: OSACA output for analysis of scalar multiplication with missing instruction form. Some lines are shortened for formatting reasons.

```
Throughput Analysis Report
--------------------------
X - No information for this instruction in data file


Port Binding in Cycles Per Iteration:
----------------------------------------------
|  Port  |  0  |  1  |  2  |  3  |  4  |  5  |
----------------------------------------------
| Cycles | 2.0 | 1.0 | 5.0 | 5.0 | 2.0 | 1.0 |
----------------------------------------------


          Ports Pressure in cycles
|  0   |  1   |  2   |  3   |  4   |  5   |
-------------------------------------------
|      |      | 0.50 | 0.50 | 1.00 |      | movl   $0x0,-0x24(%rbp)
|      |      |      |      |      |      | jmp    10b <scale+0x10b>
|      |      | 0.50 | 0.50 |      |      | mov    -0x48(%rbp),%rax
|      |      | 0.50 | 0.50 |      |      | mov    -0x24(%rbp),%edx
| 0.33 | 0.33 |      |      |      | 0.33 | movslq %edx,%rdx
|      |      | 0.50 | 0.50 |      |      | vmovsd (%rax,%rdx,8),...
| 1.00 |      | 0.50 | 0.50 |      |      | vmulsd -0x50(%rbp),...
|      |      | 0.50 | 0.50 |      |      | mov    -0x38(%rbp),%rax
|      |      | 0.50 | 0.50 |      |      | mov    -0x24(%rbp),%edx
| 0.33 | 0.33 |      |      |      | 0.33 | movslq %edx,%rdx
|      |      | 0.50 | 0.50 | 1.00 |      | vmovsd %xmm0,...
|      |      |      |      |      |      | X addl $0x1,-0x24(%rbp)
|      |      | 0.50 | 0.50 |      |      | mov    -0x24(%rbp),%eax
| 0.33 | 0.33 | 0.50 | 0.50 |      | 0.33 | cmp    -0x54(%rbp),%eax
|      |      |      |      |      |      | jl     e4 <scale+0xe4>
Total number of estimated throughput: 5.0
```

Listing 3.3: OSACA output for analysis of scalar multiplication with all instruction forms. Some lines are shortened for formatting reasons.

```
Throughput Analysis Report
--------------------------
X - No information for this instruction in data file


Port Binding in Cycles Per Iteration:
-------------------------------------------------
|  Port  |   0  |   1  |  2  |  3  |  4  |   5  |
-------------------------------------------------
| Cycles | 2.33 | 1.33 | 6.0 | 6.0 | 3.0 | 1.33 |
-------------------------------------------------


          Ports Pressure in cycles
|  0   |  1   |  2   |  3   |  4   |  5   |
-------------------------------------------
|      |      | 0.50 | 0.50 | 1.00 |      | movl   $0x0,-0x24(%rbp)
|      |      |      |      |      |      | jmp    10b <scale+0x10b>
|      |      | 0.50 | 0.50 |      |      | mov    -0x48(%rbp),%rax
|      |      | 0.50 | 0.50 |      |      | mov    -0x24(%rbp),%edx
| 0.33 | 0.33 |      |      |      | 0.33 | movslq %edx,%rdx
|      |      | 0.50 | 0.50 |      |      | vmovsd (%rax,%rdx,8),...
| 1.00 |      | 0.50 | 0.50 |      |      | vmulsd -0x50(%rbp),...
|      |      | 0.50 | 0.50 |      |      | mov    -0x38(%rbp),%rax
|      |      | 0.50 | 0.50 |      |      | mov    -0x24(%rbp),%edx
| 0.33 | 0.33 |      |      |      | 0.33 | movslq %edx,%rdx
|      |      | 0.50 | 0.50 | 1.00 |      | vmovsd %xmm0,...
| 0.33 | 0.33 | 1.00 | 1.00 | 1.00 | 0.33 | addl $0x1,-0x24(%rbp)
|      |      | 0.50 | 0.50 |      |      | mov    -0x24(%rbp),%eax
| 0.33 | 0.33 | 0.50 | 0.50 |      | 0.33 | cmp    -0x54(%rbp),%eax
|      |      |      |      |      |      | jl     e4 <scale+0xe4>
Total number of estimated throughput: 6.0
```

Mind that the python file is not part of the OSACA package, but is dependent on `params.py` and `testcase.py`, therefore in these the import of modules within the OSACA package has to be changed from

```
from osaca.[pyfile] import [class(es)]
```

to:

```
from [pyfile] import [class(es)]
```

After measuring the throughput and latency values of an instruction form via ibench, which will be discussed in Section 3.5 in detail, OSACA can incorporate the results into its data file. This is done by `osaca.py`.

For every measured value in the output file (latency and throughput) OSACA checks the data file corresponding to the defined microarchitecture for the instruction form. If there is already an entry for the instruction form and the value to check was defined earlier, it prints out both values for the user to compare, but does not change the entry. If there is no value for the given measurement, OSACA first checks it for sanity. Since every instruction must execute in an integral number of latency cycles, this is done by checking every latency value in clock cycles against integers. For the validation of throughput values it is furthermore necessary to check them against a reasonable number of reciprocals, in case the instruction may be dispatched to multiple execution ports in parallel.

Because OSACA cannot yet detect the port binding of an instruction form, yet, initially a new instruction is "bound" to port `-1` until the user manually inserts a valid port binding.

3.4 Inserting IACA Markers

For heuristically detecting the innermost loop body OSACA calls the appropriate Kerncraft function, for which the pypi package must be installed. OSACA runs the IACA instrumentation, which tries to idg see Appendix C in [10] or the github repository[5].

[5] `https://github.com/RRZE-HPC/Kerncraft`

3.5 Usage

Besides the `create_testcase` template OSACA consists only of the `osaca` command. To use `osaca` for throughput analyses or updating of the data file, the user must provide either a kernel code file or an ibench output file, somehow or other an architecture data file is necessary. The required structure of the kernel code file is described in Section 3.2 and an example usage of both the IACA byte marker and the OSACA marker can be found in the Listings in section 1.2. By default, architecture data files for common Intel architectures (SNB, IVB, Haswell (HSW), BDW, Skylake (SKL)) are provided by OSACA and have to be in CSV format with four entify blocks by searching for a high number of packed (or vectorized) instructions in between labels and conditional jump instructions. OSACA uses the manual mode of this tool to give the user the possibility to interactively select the correct block. For more information about Kerncraft's assembly block markincolumns including the unique instruction form name (`instr`), a throughput (`TP`) and a latency (`LT`) value and the port binding (`ports`). E.g., a correct entry in the data file for Intel IVB for the instruction

```
vmulsd xmm0, xmm0, qword ptr [rbp-0x50]
```

— which can be assigned either to port `0` and `2` or port `0` and `3` — would look like this:

```
vmulsd-xmm_xmm_mem,1.0,5.0,"(1.0,0.0,0.5,0.5,0.0,0.0)"
```

In order to get correct benchmark results, it is highly recommended to run ibench on an otherwise idle node. For the correct function of ibench the benchmark files from OSACA must be placed in a subdirectory of `src` in the root of the ibench repository, so it can create a folder with the subdirectory's name and the shared objects. For running the tests, the frequencies of all cores must set to a constant value[6] and this has to be supplied as an argument together with the directory of the shared objects to ibench, e.g.: "`./ibench ./AVX 2.2`" for running ibench in the directory `AVX` with a core frequency of 2.2 GHz.

In the following we will have a look at `osaca`'s command line interface and how to use it. The CLI accepts the following format:

```
osaca [-h] [-V] [--arch ARCH] [--tp-list] [-i | --iaca | -m] FILEPATH
```

- `-h` or `--help` prints out the help message.
- `-V` or `--version` shows the program's version number.
- `ARCH` needs to be replaced with the wished architecture abbreviation. This flag is necessary for the throughput analysis (default function) and the inclusion of an ibench output (`-i`). Possible options are `SNB`, `IVB`, `HSW`, `BDW` or `SKL` for the latest Intel processor generations.
- While in the throughput analysis mode, one can add `--tp-list` for printing the additional throughput list of the kernel or `--iaca` for letting OSACA to know it has to search for IACA binary markers.
- `-i` or `--include-ibench` starts the integration of ibench output into the CSV data file determined by `ARCH`.
- With the flag `-m` or `--insert-marker` OSACA calls the Kerncraft module for the interactively insertion of IACA marker in suggested assembly blocks.
- `FILEPATH` is the path to the file to work with and is always necessary

Example runs can be seen in Section 1.4 and Chapter 4.

[6] The `likwid-setFrequencies` tool from the LIKWID tool suite [21] is a convenient way to set the clock frequency.

4 Evaluation

For evaluating the functionalities of the developed tool, in the following chapter a sample kernel will be analyzed with the OSACA throughput analysis and compared to the performance analysis gained with IACA.

4.1 Throughput Analysis with OSACA and IACA

As source code we use a basic stencil code. Stencils are widely used in scientific computing for numerical solvers and apply an update function to all elements of an n-dimensional array by referencing neighboring elements, with potential for spatial and temporal locality. Below we present a basic 2D-5pt stencil code with OSACA marker included:

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

The code is compiled twice by the Intel Compiler (ICC) version 17.0.5.239 with the flags

```
-fno-alias -O3 -fopenmp -xCORE-AVX-I -g
```

and additionally `-c` and `-S`, respectively, to create an object file and an assembly output for the Intel IVB architecture. The directive `#pragma vector aligned` is added to ensure the compiler uses aligned data movement instructions for all array references when vectorizing.

A naive run with OSACA searching for the high level code marker (`osaca --arch IVB`) returns the output shown in Listing 4.1. Clearly these 72 instructions picture not a single loop body for the stencil, but a lot more.

Listing 4.1: Output of the 2D-5pt stencil OSACA throughput analysis with OSACA marker.

```
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 | 16.67 | 19.17 | 8.5 | 7.0 | 8.0 | 11.67 |
-----------------------------------------------------


          Ports Pressure in cycles
|  0   |  1   |  2   |  3   |  4   |  5   |
-------------------------------------------
| 0.50 | 0.50 |      |      |      |      | lea     (%r15,%rcx,8),%r11
| 0.50 | 0.50 |      |      |      |      | lea     (%r14,%rcx,8),%rdx
| 0.33 | 0.33 |      |      |      | 0.33 | add     $0xfffffffe,%edi
| 0.33 | 0.33 |      |      |      | 0.33 | mov     %rcx,%r9
| 0.33 | 0.33 |      |      |      | 0.33 | mov     %edi,%eax
| 0.33 | 0.33 |      |      |      | 0.33 | and     $0xfffffff0,%eax
|      | 0.50 |      |      |      | 0.50 | shl     $0x4,%r9
| 0.33 | 0.33 |      |      |      | 0.33 | movslq %eax,%rax
| 0.33 | 0.33 |      |      |      | 0.33 | add     %r14,%r9
| 0.33 | 0.33 |      |      |      | 0.33 | movslq %edi,%rdi
|      |      | 0.50 | 0.50 | 1.00 |      | mov     %rax,-0x50(%rbp)
|      |      | 0.50 | 0.50 | 1.00 |      | mov     %rdi,-0x58(%rbp)
| 0.33 | 0.33 |      |      |      | 0.33 | cmp     $0x2,%rcx
|      |      |      |      |      |      | jle     196 <jacobi2D5pt+0x196>
| 0.33 | 0.33 |      |      |      | 0.33 | cmp     $0x10,%edi
|      |      |      |      |      |      | jl      20b <jacobi2D5pt+0x20b>
|      |      | 0.50 | 0.50 |      |      | mov     -0x48(%rbp),%r14d
| 0.33 | 0.33 |      |      |      | 0.33 | xor     %edx,%edx
|      |      | 0.50 | 0.50 |      |      | mov     -0x50(%rbp),%r12
| 0.50 | 0.50 |      |      |      |      | lea     (%r11,%r8,1),%rax
|      |      | 0.50 | 0.50 | 1.00 |      | vmovupd %ymm0,0x8(%rax,%rdx,8)
|      |      | 0.50 | 0.50 | 1.00 |      | vmovupd %ymm0,0x28(%rax,%rdx,8)
|      |      | 0.50 | 0.50 | 1.00 |      | vmovupd %ymm0,0x48(%rax,%rdx,8)
|      |      | 0.50 | 0.50 | 1.00 |      | vmovupd %ymm0,0x68(%rax,%rdx,8)
| 0.33 | 0.33 |      |      |      | 0.33 | add     $0x10,%rdx
| 0.33 | 0.33 |      |      |      | 0.33 | cmp     %r12,%rdx
|      |      |      |      |      |      | jb      d9 <jacobi2D5pt+0xd9>
| 0.50 | 0.50 |      |      |      |      | lea     0x1(%r14),%eax
| 0.33 | 0.33 |      |      |      | 0.33 | cmp     %edi,%eax
|      |      |      |      |      |      | ja      196 <jacobi2D5pt+0x196>
| 0.33 | 0.33 |      |      |      | 0.33 | movslq %r14d,%r14
|      |      | 0.50 | 0.50 |      |      | mov     -0x58(%rbp),%r13
| 0.33 | 0.33 |      |      |      | 0.33 | sub     %r14,%r13
| 0.33 | 0.33 |      |      |      | 0.33 | cmp     $0x4,%r13
|      |      |      |      |      |      | jl      203 <jacobi2D5pt+0x203>
| 0.33 | 0.33 |      |      |      | 0.33 | mov     %r13d,%r15d
| 0.50 | 0.50 |      |      |      |      | lea     (%r11,%r8,1),%rax
| 0.33 | 0.33 |      |      |      | 0.33 | and     $0xfffffffc,%r15d
| 0.33 | 0.33 |      |      |      | 0.33 | xor     %edx,%edx
| 0.33 | 0.33 |      |      |      | 0.33 | movslq %r15d,%r15
| 0.50 | 0.50 |      |      |      |      | lea     (%rax,%r14,8),%rax
|      |      | 0.50 | 0.50 | 1.00 |      | vmovupd %ymm0,0x8(%rax,%rdx,8)
| 0.33 | 0.33 |      |      |      | 0.33 | add     $0x4,%rdx
| 0.33 | 0.33 |      |      |      | 0.33 | cmp     %r15,%rdx
|      |      |      |      |      |      | jb      12e <jacobi2D5pt+0x12e>
| 0.33 | 0.33 |      |      |      | 0.33 | cmp     %r13,%r15
|      |      |      |      |      |      | jae     196 <jacobi2D5pt+0x196>
|      |      | 0.50 | 0.50 |      |      | mov     -0x38(%rbp),%rax
| 0.50 | 0.50 |      |      |      |      | lea     (%r11,%r8,1),%r12
|      |      | 0.50 | 0.50 |      |      | mov     -0x40(%rbp),%rsi
```

```
    | 0.50 | 0.50 |      |      |      |      | lea    (%r9,%r8,1),%rdx
    | 0.50 | 0.50 |      |      |      |      | lea    (%r12,%r14,8),%r12
70  | 0.33 | 0.33 |      |      |      | 0.33 | add    %r8,%rax
    | 0.33 | 0.33 |      |      |      | 0.33 | add    %r8,%rsi
    | 0.50 | 0.50 |      |      |      |      | lea    (%rdx,%r14,8),%rdx
    | 0.50 | 0.50 |      |      |      |      | lea    (%rax,%r14,8),%rax
    | 0.50 | 0.50 |      |      |      |      | lea    (%rsi,%r14,8),%r14
75  |      |      | 0.50 | 0.50 |      |      | vmovsd (%r14,%r15,8),%xmm2
    |      | 1.00 | 0.50 |      |      | 0.50 | vaddsd 0x10(%r14,%r15,8),%xmm2,%xmm3
    |      | 1.00 | 0.50 |      |      | 0.50 | vaddsd 0x8(%rax,%r15,8),%xmm3,%xmm4
    |      | 1.00 | 0.50 |      |      | 0.50 | vaddsd 0x8(%rdx,%r15,8),%xmm4,%xmm5
    | 1.00 |      |      |      |      |      | vmulsd %xmm5,%xmm1,%xmm6
80  |      |      | 0.50 | 0.50 | 1.00 |      | vmovsd %xmm6,0x8(%r12,%r15,8)
    | 0.33 | 0.33 |      |      |      | 0.33 | inc    %r15
    | 0.33 | 0.33 |      |      |      | 0.33 | cmp    %r13,%r15
    |      |      |      |      |      |      | jb     168 <jacobi2D5pt+0x168>
    | 0.33 | 0.33 |      |      |      | 0.33 | xor    %r15d,%r15d
85  |      |      |      |      |      |      | jmpq   13d <jacobi2D5pt+0x13d>
    | 0.33 | 0.33 |      |      |      | 0.33 | xor    %r14d,%r14d
    |      |      |      |      |      |      | jmpq   fa <jacobi2D5pt+0xfa>
    |      |      |      |      |      |      | * nopl   (%rax)
    |      |      |      |      |      |      | * nopw   %cs:0x0(%rax,%rax,1)
90  Total number of estimated throughput: 19.17
```

However, no loop unrolling happened, because we still only have one iteration with its expected three additions and one multiplication in lines 76–79. This is because `objdump` does not always list the source code at the correct place(s) in the disassembled object code. It assigns the OSACA start marker to too many code sequences, so the kernel enlarges erroneously. The same behavior can be observed using the GNU Debugger (`gdb`) for code interleaving.

Therefore OSACA provides a built-in function to call the automatic Kerncraft IACA marker. As seen in the interactive Kerncraft output below, the manual insertion of OSACA calls with `osaca -m` suggests to choose block 2 as loop body, due to the highest number of used AVX registers:

```
Blocks found in assembly file:
   block     | OPs | pck. | AVX || Registers |    YMM    |    XMM    |    GP    ||ptr.inc|
-----------+-----+------+-----++-----------+-----------+-----------+---------++-------|
 0  ..B1.8  |   8 |    0 |   0 ||  15 (  4) |   4 ( 1) |   0 ( 0) | 11 ( 3) ||   128 |
 1 ..B1.13 |   5 |    0 |   0 ||   6 (  4) |   1 ( 1) |   0 ( 0) |  5 ( 3) ||    32 |
 2 ..B1.17 |  12 |    0 |   0 ||  28 ( 13) |   1 ( 1) |  11 ( 6) | 16 ( 6) ||  None |
Choose block to be marked [2]:
```

This corresponds lines 75–83 in Listing 4.1.

Kerncraft then automatically inserts the IACA byte markers. To run OSACA again, it is not necessary to recompile the assembly code, because OSACA can work with IACA marked assembly code as well. The command for starting the throughput analysis is `osaca --arch IVB --iaca` for Intel IVB with IACA markers. The throughput analysis now prints out the expected kernel, as shown in Listing 4.2.

To explain the work between OSACA and ibench on an example, we assume `vmulsd` in the marked kernel is not in the data file yet, so first it would look like:

```
                Ports Pressure in cycles
       |  0   |  1   |  2   |  3   |  4   |  5   |
       -------------------------------------------
       ...
       |      |      |      |      |      |      | X vmulsd    %xmm5, %xmm1, %xmm6
       ...
```

If OSACA does not have any information about the instruction form at all or about the port binding of it, it prints out a blank line, marked with an "X". In beforehand, while fetching the instruction forms, OSACA creates two assembly files for benchmarking both the latency and the throughput of the instruction form. To get valid measurements, one has to copy the files from the `benchmark` folder to the `src` folder of ibench, compile them via the Makefile to create shared object files and run ibench on a exclusively allocated core with a fixed in-core frequency.

Listing 4.2: Output of the 2D-5pt stencil OSACA throughput analysis with IACA markers.

```
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
```

Listing 4.3: Output of the 2D-5pt stencil IACA throughput analysis. Some lines are shortened for formatting reasons.

```
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  |
-------------------------------------------------------------------------

N - port number or number of cycles resource conflict caused delay, DV - Divider pipe
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   1.0 |           |     |     |    | vmovsd xmm2, ...
|   2    |           | 1.0 |           | 1.0   1.0 |     |     | CP | vaddsd xmm3, ...
|   2    |           | 1.0 | 1.0   1.0 |           |     |     | CP | vaddsd xmm4, ...
|   2    |           | 1.0 |           | 1.0   1.0 |     |     | CP | vaddsd xmm5, ...
|   1    | 1.0       |     |           |           |     |     |    | vmulsd xmm6, ...
|   2    |           |     | 0.5       | 0.5       | 1.0 |     |    | vmovsd qword ...
|   1    |           |     |           |           |     | 1.0 |    | inc r15
|   1    |           |     |           |           |     | 1.0 |    | cmp r15, r13
|   0F   |           |     |           |           |     |     |    | jb 0xffffffff...
Total Num Of Uops: 12
```

A successful run will output two values similar to the Listing below, where we ran ibench on an Intel IVB with 2.2 GHz core frequency:

```
Using frequency 2.20GHz.
vmulsd-xmm_xmm_xmm:     1.004 (clock cycles)    [DEBUG - result: 1.000000]
vmulsd-xmm_xmm_xmm-TP:  5.015 (clock cycles)    [DEBUG - result: 1.000000]
```

In order to add these values to the data file, one has to call osaca with the `-i` flag and a specific microarchitecture defined by `--arch`, e.g. `IVB`. After manually adding the port binding for `vmulsd-xmm_xmm_xmm` in the CSV file[1], the throughput analysis of the instruction form looks as in Listing 4.2.

The total reciprocal throughput is predicted as `3.67 cy`. Since the vectorized add of scalar double-precision floating point values (`VADDSD`) always needs port `1` for execution and therefore never can be scheduled to another functional unit for all three additions, it is sure to say the bottleneck for this stencil on a single core and all data in the L1 cache is port `1`.

For comparison we analyze the exact same IACA-marked kernel with Intel's IACA. For this, another compilation of the assembly source is necessary for creating an object file including the IACA markers (`icc -c`). The command `iaca.sh -arch IVB` analyzes the given file. IACA's output is shown in Listing 4.3.

IACA prints out some more information, like the total number of micro operations for each instruction form, the divider pipe on port `0` to illustrate the availability of the port for floating-point division or the data fetch pipes for ports `2` and `3`, which allow the Address Generation Units (AGUs) to get freed after 1 cycle for 256-bit load operations keeping the port busy for 2 cycles. Furthermore, e.g., IACA marks macro fusions with an "`F`" after the second fused instruction or "`@`", if an SSE instruction follows an AVX instruction, for which the upper 128 bits of the YMM register have to be stored which results in a massive delay. Finally, IACA gives a small clue about the critical path for the kernel, which is marked with a "`CP`" and returns a prediction about the assumed bottleneck, such as front-end, port number, divider unit or long dependency chains.

One of the most remarkable differences is the diverse representation of port bindings: IACA schedules an instruction form, that can be executed on different ports equally, sometimes fully, sometimes equally distributed and even imbalanced between ports, e.g.:

```
        | Num Of |              Ports pressure in cycles              |     |
        |  Uops  |  0  - DV  |  1  |  2  -  D  |  3  -  D  |  4  |  5  |     |
        ---------------------------------------------------------------------
        |   1    | 0.9       |     |           |           |     | 0.1 |     | inc rax
```

Here IACA predicts that only every tenth `inc` instruction gets scheduled to port `5`, while the rest is executed on port `0`. Port `1`, which is able to handle `inc` as well, is left out completely. We speculate that IACA has some additional internal information about the instruction scheduling, therefore it can not be reproduced without further investigation or publicly unavailable architecture information. .

OSACA, in contrast, always schedules an instruction form equally on all available ports for execution. This can lead to a different maximum throughput estimation, as in Listings 4.2 and 4.3. While IACA only predicts pressure of `3 cy` on port `1` and assumes that `inc r15` and `cmp r15,r13` are executed always on port `5`, OSACA considers that every third operation of both will be executed on port `1`, due to the possibility of executing these instructions either on port `0`, `1` or `5` [11]. Therefore, the total inverse block throughput is estimated as `3.67 cy` instead of IACA's `3.0 cy`.

Another difference between OSACA and IACA can be seen when using the OSACA marker for a kernel. When running `objdump` for interleaving the assembly and high level source code, it includes the loop initialization into the marked kernel. For instance, this can be observed in Listing 3.3. The `jmp` instruction in the next to last line points to the `mov -0x48(%rbp),%rax` in the third line of the kernel. Therefore, in a strict sense, the innermost loop only contains the instructions form the third to the second to last line. Especially for large kernels this small number of instructions can be neglected for throughput analysis, but the user should be aware that IACA markers are a more precise way of indicating the loop code.

[1] `vmulsd` with `xmm` registers as operands only runs on port `0`, therefore a correct port binding in the data file would be: `vmulsd-xmm_xmm_xmm,1.0,5.0,"(1.0,0,0,0,0,0)"`.

Furthermore IACA provides in its version 2.3 the function to create a graphical trace including in-depth information about different operation stages inside the processor (Allocate - Source ready - Dispatched - Execute - Writeback - Retire - Idle) to identify bottlenecks and pressure points.

If an instruction form is not supported by any of the two tools, an immediate analysis is not possible for any of them, naturally. OSACA automatically creates an assembly benchmark file for measuring throughput and latency of the missing instruction form via ibench. After integration of the given output of ibench, it is able to show a throughput analysis of the instruction in the context of the kernel, which is not possible with IACA. An example instruction which is not included in IACA is `xgetbv`[2]. OSACA's data file therefore is able to grow dynamically and can even be extended on other chip architectures. Also the benchmark files produced during instruction fetching are not bound to the current microarchitecture and are usable for various microarchitectures with Intel assembly syntax.

4.2 IACA versus OSACA

At the moment OSACA is not able to reproduce all the functionalities of the IACA tool. Nevertheless OSACA provides a throughput analysis of a marked kernel, both for IACA byte markers and OSACA high level code marker. If one is not able at all to identify the inner-most loop for setting IACA byte markers, it is a good approach to use the OSACA high level code marker for finding the kernel or at least the surrounding area of it.

Currently OSACA only supports Intel processors with Sandy Bridge or later microarchitecture, due to their good documentation and the ability to compare results to IACA. Furthermore, ibench only supports Intel syntax for now.

OSACA can create benchmark files for unsupported instructions on-the-fly and therefore allows to add new measured values semi-automatically. Thus, it can dynamically update and enlarge the data files for supported microarchitectures.

In contrast to IACA's not always comprehensible scheduling of single instructions to the ports, OSACA's port binding is designed to always distribute the instructions equally to all available ports, which allows the user to get an more detailed view of possible bindings.

[2]Further information to this instruction can be found on http://www.felixcloutier.com/x86/XGETBV.html.

5 Conclusion and Future Work

We have shown that an automatic throughput analysis of loop kernels based on semi-automatic benchmark measurements is possible and allows a detailed view on in-core loop performance on a given hardware architecture. The OSACA tool can extract loop kernels and their instruction forms out of a given, labeled assembly code or object file and automatically creates benchmark files for instructions forms that are not in the data files yet. After measuring instruction throughput and latency via the benchmark tool ibench, OSACA automatically integrates the results and checks for validity. In the current version the user must still provide information about the port(s) that each instruction can be assigned to. With full knowledge about the instructions involved, OSACA can then produce a prediction of best-case (full throughput) in-core execution time for each loop kernel. Hence, it may function as a replacement for the closed-source IACA (Intel Architecture Code Analyzer) tool provided by Intel.

In the nearest future it is planned to make OSACA available as a listed python package on the Python Package Index (pypi) [22]. In order to automate the prediction process further, an automatic determination of the port binding for each newly added instruction form will be implemented. One possible way to achieve this is to add likwid [21] instrumentation to ibench to count the micro-operations executed on the units behind every port. This will only be possible on architectures with a port model, such as all current Intel multicore CPUs.

Another future challenge is the support of other x86 and non-x86 microarchitectures. While other x86 microarchitectures, e.g., by AMD, will be less problematic, due to the similar or identical assembly syntax, it is an open question how challenging a support of all functionality on non-x86 microarchitectures will be.

In order to overcome irregular behavior with `objdump` while extracting kernels with high level OSACA marker, we will keep searching for either already freely available modules to interleave assembly and high level code with debugging information or implement this functionality by ourselves.

As we observed different behavior depending not only on the type, but also on the value of operands in an instruction form, it is planned to provide a set of "special" operands for instruction forms in the benchmark files. This can be predefined values as `NaN`, `0`, `INTEGER.MAX` or `INTEGER.MIN`, as well as user-defined operands if needed. Additionally, not only special immediate, but also special register operands are planned to be supported, to cover unusual behavior, e.g. for the higher 128 bit of a register in comparison to the lower 128. Furthermore, a more efficient way to identify suitable register ranges must be found so that any instruction in the data file can be combined with any (allowed) register. In the current version of OSACA, the range of usable registers is hard-coded.

For supporting predictions about the worst-case execution path of a loop kernel, a different future challenge is to enable OSACA to identify dependency chains and therefore the critical path. This feature was dropped for Intel's IACA since version 2.2. This kind of analysis will enable "worst-case/best-case" type of modeling, where the actual execution time is expected to lie between the two extremal predictions.

Finally, it is intended to replace IACA in the Kerncraft performance analysis tool as soon as OSACA supports all needed functionality, in order to overcome the dependency to Intel architectures and proprietary software components.

Bibliography

[1] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4):65–76, April 2009. ISSN 0001-0782. doi: 10.1145/1498765.1498785. URL `http://doi.acm.org/10.1145/1498765.1498785`.

[2] G. Ofenbeck, R. Steinmann, V. Caparros, D. G. Spampinato, and M. Püschel. Applying the roofline model. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 76–85, March 2014. doi: 10.1109/ISPASS.2014.6844463.

[3] Johannes Hofmann, Jan Eitzinger, and Dietmar Fey. Execution-Cache-Memory Performance Model: Introduction and Validation. *CoRR*, abs/1509.03118, 2015. URL `http://arxiv.org/abs/1509.03118`.

[4] Holger Stengel, Jan Treibig, Georg Hager, and Gerhard Wellein. Quantifying Performance Bottlenecks of Stencil Computations Using the Execution-Cache-Memory Model. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 207–216, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3559-1. doi: 10.1145/2751205.2751240. URL `http://doi.acm.org/10.1145/2751205.2751240`.

[5] Gideon S. Israel Hirsh. Intel® Architecture Code Analyzer. URL `https://software.intel.com/en-us/articles/intel-architecture-code-analyzer`.

[6] Julian Hammer, Jan Eitzinger, Georg Hager, and Gerhard Wellein. Kerncraft: A Tool for Analytic Performance Modeling of Loop Kernels. *CoRR*, abs/1702.04653, 2017. URL `http://arxiv.org/abs/1702.04653`.

[7] Johannes Hofmann. ibench - Instruction Benchmarks, 2017. URL `https://github.com/hofm/ibench`.

[8] *Intel® Architecture Code Analyzer User's Manual.* Intel Corporation, 2.3 edition, 03 2017. URL `https://software.intel.com/sites/default/files/managed/29/78/intel-architecture-code-analyzer-2.3-users-guide.pdf`.

[9] Sri Hari Krishna Narayanan, Boyana Norris, and Paul D. Hovland. Generating Performance Bounds from Source Code.

[10] Julian Hammer. Automatic Loop Kernel Analysis and Performance Modeling, 07 2015. URL `https://github.com/RRZE-HPC/kerncraft/blob/master/doc/masterthesis-2015.pdf`.

[11] Agner Fog. 4. Instruction Tables, 1997-2017. URL `http://www.agner.org/optimize/instruction_tables.pdf`.

[12] John von Neumann. First Draft of a Report on the EDVAC. *IEEE Ann. Hist. Comput.*, 15(4):27–75, October 1993. ISSN 1058-6180. doi: 10.1109/85.238389. URL `http://dx.doi.org/10.1109/85.238389`.

[13] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers.* Chapman & Hall/CRC Computational Science. CRC Press, 2010. ISBN 9781439811931. URL `https://www.crcpress.com/Introduction-to-High-Performance-Computing-for-Scientists-and-Engineers/Hager-Wellein/p/book/9781439811924`.

[14] Anand Lal Shimpi. Intel's Haswell Architecture Analyzed: Building a New PC and a New Intel, 10 2012. URL https://www.anandtech.com/show/6355/intels-haswell-architecture/8.

[15] *Intel 64 and IA-32 Architecture Optimization Reference Manual*. Intel Coporation, 6 2016. URL https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[16] Inc. Free Software Foundation. GNU Affero General Public License, 2007. URL http://www.gnu.org/licenses/agpl-3.0.html.

[17] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification. Technical report, 05 1995.

[18] Jack Dongarra, Michael A. Heroux, and Piotr Luszczek. HPCG Benchmark: a New Metric for Ranking High Performance Computing Systems. Technical Report ut-eecs-15-736, 01-2015 2015. URL http://www.eecs.utk.edu/resources/library/file/1047/ut-eecs-15-736.pdf.

[19] John McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. 09 1991. URL http://cs.virginia.edu/stream.

[20] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A Call Graph Execution Profiler. *SIGPLAN Not.*, 17(6):120–126, June 1982. ISSN 0362-1340. doi: 10.1145/872726.806987. URL http://doi.acm.org/10.1145/872726.806987.

[21] Like I knew what I Am Doing - Lightweight performance tool. URL https://github.com/RRZE-HPC/likwid.

[22] PyPI - the Python Package Index. URL https://pypi.python.org/pypi.