TÉCNICO LISBOA

Exploring processor frontend capabilities via micro-benchmarking

Rafael Abrantes Forte

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Doctor Aleksandar Ilic
Doctor Leonel Augusto Pires Seabra de Sousa

Examination Committee

President: Doctor Teresa Maria Sá Ferreira Vazão Vasques
Supervisor: Doctor Aleksandar Ilic
Vogal: Doctor João Nuno de Oliveira e Silva

January 2021

Abstract

Nowadays, processor companies aim at achieving higher performing and more efficient processor architectures, which is not an easy task as we reach the limits of Moore's Law. To keep up with the market demands, processors micro-architectures are increasing in complexity, which makes it harder for application developers and other professionals to identify the factors that affect application efficiency and performance. In order to accurately characterize and improve application performance, it is necessary to rely on models and tools that provide useful insights on how an application is performing on the micro-processor. These models use micro benchmarking and hardware counters in order to obtain their the metrics, to guarantee more accurate results. Even though there are plenty of state-of-the art models used to analyse application performance, most of them focus on the Back End portion the the micro-architecture. Consequently, the Front End of the micro-architecture, which is responsible for the decoding and issuing of instructions, has been overlooked. Which is an issue considering how it can impact and bottleneck application performance. In this Thesis a new methodology of micro-benchmarking to assess Front End (FE) limitations is proposed, in order to provide useful insights on their impact in application performance. To achieve this goal, a set of micro benchmarks is designed using hardware counters and tested on the Intel Skylake micro-architecture to assess the Front End components and their capabilities. Finally a set of benchmarks are created and used to validate the methodology proposed in different scenarios.

Keywords

Performance; Efficiency; Micro Benchmarking; Front End; MITE; DSB; Application Characterization; Integer Applications.

Resumo

Nos dias de hoje as empresas de processadores estão em constante competição para criarem os processadores mais eficientes e com melhor performance possível, sendo cada vez mais dificíl à medida que nos aproximamos do limite da lei de Moore. Para acompanharem as exigências do mercado, as micro arquitecturas dos processadores estão cada vez mais complexas, o que dificulta o trabalho de encontrar os factores que afectam a eficiência e a performance que é da responsabilidade de engenheiros de software e outros proffisionais. De modo a caracterizar e optimizar aplicações é necessário recorrer a modelos e ferramentas que forneçam informações uteís no que toca à performance da aplicação em determinado micro-processador. Estes modelos utilizam micro benchmarking e hardware counters para obterem as suas métricas, e garantirem resultados mais precisos. Apesar de existirem vários modelos para analisar a performance de aplicações, a maior parte deles foca-se sobretudo no Back End das micro-arquitecturas. Consequentemente, o Front End das micro-arquitecturas, que é responsável pela descodificação e envio de instrucões para o Back End, tem sido esquecido. O que é problemático considerando o impacto que este pode ter na performance de aplicações. O objectivo desta tese é propor uma nova metodologia para o micro benchmarking do Front End que avalie as suas limitações, de modo a fornecer informações úteis sobre os seus impactos na performance de aplicações. Para atingir este objectivo foram criadas micro benchmarks, recorrendo a hardware counters, e executadas num processador Intel Skylake, para obter resultados relativos às capacidades do Front End da micro-arquitetura e das suas componentes. Por fim foram criadas benchmarks para validar a metodologia proposta em diversos cenários.

Palavras Chave

Performance; Eficiência; Micro Benchmarking; Front End; MITE; DSB; Caracterização de aplicações; Aplicações de Inteiros.

Contents

1 Introduction 1

1.1 Motivation . . . 2

1.2 Objectives . . . 3

1.3 Main Contributions . . . 3

1.4 Outline . . . 4

2 Background and State of the art 5

2.1 Intel Core Micro-Architecture . . . 6

2.1.1 Intel Skylake micro-architecture . . . 6

2.1.2 Front End . . . 7

2.1.3 Back End . . . 10

2.2 Micro Benchmarking . . . 12

2.2.1 Performance Models Based on Benchmarking and Hardware Counters . . . 13

2.3 State of the Art Approaches . . . 17

2.4 Open Challenges . . . 20

2.5 Summary . . . 23

3 Micro Benchmarking of Intel Skylake Front End 25

3.1 Micro Benchmarking Tool . . . 26

3.2 Front End Micro benchmarks . . . 27

3.2.1 MITE Benchmarks . . . 28

3.2.2 DSB Benchmarks . . . 31

3.2.3 Bottleneck Prediction . . . 32

3.3 Validation Tests . . . 33

3.4 Summary . . . 34

4 Experimental Results 35

4.1 Front End Micro-benchmarking Result . . . 36

4.1.1 MITE Micro Benchmark Results . . . 36

4.1.2 DSB Micro Benchmark Results . . . 41

4.1.3 Predicting bottlenecks and performance . . . 45

4.2 Summary . . . 55

5 Conclusions **56**

5.1 Conclusions and Future Work 57

5.1.1 Future Works 57

List of Figures

2.1 Skylake SP Pipeline. [1] 7

2.2 Skylake SP Front End. [2] 8

2.3 Skylake SP Back End. [2] 11

2.4 Original Roofline Model in OpteronX2. [3] 14

2.5 Memory Traffic CARM Vs RM. [4] 14

2.6 Example CARM Model. [5] 15

2.7 Top Down Hierarchy 17

2.8 SPEC CPU2017 benchmarks 20

2.9 Roofline 600_perlbench_s 22

2.10 Top Down perlbench_s - s_regmatch 22

2.11 s_regmatch detailed Top Down 22

3.1 Benchmarking tool layout 26

4.1 Results for $P_{Overall\_MITE}$ 37

4.2 Results for P_{Only_MITE} 37

4.3 Uops Coming from MITE 39

4.4 Overall Throughput - $P_{Overall}$ 40

4.5 Overall Bandwidth 41

4.6 Results for $P_{Overall\_DSB}$ - 2B, 3B, 4B and 5B Instructions 42

4.7 Results for $P_{Overall\_DSB}$ - 6B, 7B, 8B, 9B and 10B Instructions 42

4.8 Results for P_{Only_DSB} 42

4.9 Uops issued by DSB - 2B-5B Instructions 44

4.10 Uops issued by DSB-6B-10B Instructions 44

4.11 Uops issued by DSB per Total instructions - 2B-5B Instructions 44

4.12 Uops issued by DSB per Total instructions - 6B-10B Instructions 44

4.13 Blocks of Random Instructions - Throughput blocks 0 and 1 46

4.14 Blocks of Random Instructions - Throughput blocks 2 and 3 47

4.15 Blocks of Random Instructions - Error Margin 47

4.16 Blocks of Random Instructions - Not considering retirement bottleneck 48

4.17 Throughput Instructions with 4 ports 49

4.18 Error Margin - Instructions with 4 ports 50

4.19 Throughput Instructions with 3 ports 50
4.20 Error Margin - Instructions with 3 ports 51
4.21 Load Data L1 - Throughput 52
4.22 Load Data L1 - Error Margin 53
4.23 Load Data L2 - Throughput 53
4.24 Load Data L3 - Throughput 54
4.25 Load Data DRAM - Throughput 54
4.26 Store Data L1 - Throughput 55

List of Tables

2.1 State of the art works 18

3.1 Hardware Counters MITE micro benchmarking [6] 31
3.2 Hardware Counters DSB micro benchmarking [6] 32

4.1 Machine Specifications 36
4.2 Blocks Instruction Details. 45
4.3 Instructions Second Set of Tests 48
4.4 Instructions Third set tests 52

List of Acronyms

CARM	Cache-Aware Roofline Model
ORM	Original Roofline Model
LLC	Last Level Cache
OOO	Out-Of-Order
AVX512	Advanced Vector Instructions 512
IDQ	Instruction Decode Queue
MSROM	Micro-Code Store Read Only Memory
DSB	Decoded Stream Buffer
BPU	Branch Prediction Unit
LSD	Loop Stream Detector
ROB	Re-Order Buffer
RAT	Register Alias Table
CPI	Clockticks per Instruction
FE	Front End
BE	Back End
MITE	Micro-Instruction Translation Engine
IQ	Instruction Queue
ITLB	Instruction TLB
L1 ICache	L1 Instruction Cache
BOB	Branch Order Buffer
EU	Execution Units
Flops	Floating Point Operations per Second
AI	Arithmetic Intensity

1 Introduction

Contents

1.1 Motivation . . . 2
1.2 Objectives . . . 3
1.3 Main Contributions . . . 3
1.4 Outline . . . 4

Over the course of the last decade micro-processors were target of several micro-architectural enhancements in order to keep up with the increasing performance demands. However, these enhancements have also contributed to the increasing complexity of the underlying hardware. For example, current modern multi-core systems contain a memory hierarchy with several memory levels, and support a wide range of function units. For this reason, achieving an efficient execution of applications is modern systems is a demanding task. This leads to software developers having to search for the best optimization techniques in a broad design space.

In order to accurately pinpoint the architecture components that prevent the application to achieve higher performance, it is necessary to assess the performance limits of micro-architectures. However, this is not always an easy task due to the multitude and complexity of the components inside the micro-architecture, many of which work in conjunction, thus making it more difficult to evaluate their limits. The assessment of micro-architecture limits can be performed through micro-benchmarks, that are designed to exercise different components in the core pipeline. This is an important process because it gives the realistic performance limits, different from the theoretical maximums typically found in the data sheets. To validate the micro-benchmarks it is also essential to rely on hardware counters. These registers included in the micro-architectures can not only be used to obtain a large number of micro-architecture related metrics, but also to verify the correctness of micro-benchmarks.

The added value of micro-benchmarks make them popular among several state-of-art works, that rely on them to obtain their metrics. For example, the Cache-Aware Roofline Model (CARM) [4] uses micro-benchmarks in order to obtain micro-architecture metrics, such as memory bandwidths and computational throughput of instructions, which are further combined to provide insights regarding applications bottlenecks and optimization guidelines. Other methods and tools also rely on hardware counter based metrics to give the idea of the main performance limiters of applications, e.g., the Top Down Method [7], which focuses on the utilization of different pipeline components when determining the performance bottlenecks.

1.1 Motivation

Until now processors have followed the Moore's Law, duplicating the number of transistors per chip every two years, and that has allowed computational power of processors to grow at a fast pace. However, Moore's Law is reaching it's limits since transistors can not physically continue to get smaller, putting more pressure on the increase of efficiency and performance. The continue enhancement of micro-architectures also comes with an increase of their complexity, turning this process of improvement more expensive and slow.

Due to their complexity it is hard to analyse application performance without an adequate performance model. Nowadays there are many tools and performance models available, each one using a slightly different approach to uncover micro-architecture limitations. However, the majority of these models focus only on one part of the micro-architecture, the Back End. Models such as CARM, provide great insights regarding application performance bottlenecks, but are lacking any information regarding Front End bottlenecks.

While many important scientific applications over the last decade might have been mostly Back End bottlenecked, due to the great amount of floating point operations they performed, in current days more and more applications are becoming limited by the Front End. For example, artificial intelligence and neural network applications are made of a big number of integer instructions and branches, which lead them to be bottlenecked by the Front End. Since these type of applications continue to evolve, there is a need to include the Front End in current application performance models.

To tackle these issues, the main objective of this Thesis is to provide a micro benchmarking methodology of the Front End of current micro-architectures to assess its limitations in a wide range of scenarios. This will allow to better understand the components of the Front End and how they can impact application performance.

1.2 Objectives

To accomplish this task, the following objectives are established:

- Developing a micro-benchmarking methodology that exercises the diverse components of the Front End under different execution scenarios;
- Proposing the minimum set of performance counters present in Intel processors that allow to derive the metrics necessary to benchmark the main Front End components;
- In-depth micro-benchmarking of the performance limits of the Front End of an Intel Skylake micro-architecture when fetching instructions from different memory levels and when issuing micro operations through different hardware resources;
- Predicting the performance bottlenecks of several benchmarks that mimic the characteristics of real-world applications with different instruction sizes and accessing different memory levels for both data and instruction fetching.

1.3 Main Contributions

In this Thesis, a micro-benchmarking methodology is proposed in order to evaluate the performance limits of the Front End under different scenarios. This methodology consists on assessing different Front End components related to the decoding and issuing of micro operations. The methodology considers a wide range of scenarios, from accessing instructions from different memory levels to varying the instruction sizes. To asses each component individually, new performance metrics were designed related to each component that provide new insights regarding Front End performance. The hardware counters used provide a good base of Front End related metrics useful for benchmarking several Front End components and understanding their limitations.

Furthermore, by relying on this methodology, the Front End of an Intel Skylake processor is evaluated in order to assess its limitations when fetching and decoding instructions from multiple memory levels and when using different Front End components. Besides this, it was also validated for some kernels that simulate the characteristics of real applications. The low errors obtained allow to conclude that the method is accurate and can be used in performance models that lack information regarding Front End bottlenecks. Improving the quality of the performance analysis by incorporating both Front End and Back End limitations.

1.4 Outline

This thesis is organized as follows:

- **Chapter 2 - Background and State of the art :** presents a summary on the state of the art. This chapter contains four sections. The first will provide detailed information on the Skylake micro-architecture, the multi-core processor used in this thesis. The next section highlights the importance of micro benchmarking and its use in state of the art works. It will also provide a description of models based on micro benchmarking and hardware counters that are used in today's tools for application performance analysism such as CARM [4] and Top Down method[7]. After presenting these models, the following section briefly introduce several state-of-the-art works related to micro benchmarking and performance analysis. Finally the open challenges that we propose to tackle are presented and discussed.

- **Chapter 3 -Micro Benchmarking:** In this chapter the tool used for micro benchmarking is presented, followed by a description of the proposed methodology for micro-benchmarking the Front End, as well as a methodology to validate the results of our method.

- **Chapter 4 - Experimental Results:** In this chapter the results obtained from the micro-benchmarks are presented alongside a critical discussion and some conclusions about their outcome. After presenting the results we present a different approach to calculate Front End bottlenecks, and execute a series of tests in order to validate this new approach.

- **Chapter 5 - Conclusions and Future Work:** In this final chapter, conclusions on the performed worked are given along with some future work suggestions.

2

Background and State of the art

Contents

2.1 Intel Core Micro-Architecture 6
2.1.1 Intel Skylake micro-architecture 6
2.1.2 Front End 7
2.1.3 Back End 10
2.2 Micro Benchmarking 12
2.2.1 Performance Models Based on Benchmarking and Hardware Counters 13
2.3 State of the Art Approaches 17
2.4 Open Challenges 20
2.5 Summary 23

The work on this Thesis mainly focuses on micro benchmarking the different hardware components involved in the fetching and decoding of instructions in modern Out-Of-Order (OOO) computing systems. While these hardware resouces can significantly impact application performance they are usually not considered by different methods and models that aim at modeling and evaluate application performance. To address this issue it is essential to have a solid background on the micro-architecture of current multi-core processors, as well as the usability of micro-benchmarking on the state-of-the-art models and tools used for performance modeling in order to derive a useful and accurate micro-benchmarking methodology.

With this aim, this chapter introduces one of the most recent Intel micro-architectures, Skylake, providing a detailed overview on the different components of the processor, such as memory hierarchy and pipeline stages. After covering this subject in detail, the state-of-the-art solutions that make use of micro benchmarking and hardware counters for performance modeling are presented and discussed, in particular the CARM [4] and the Top-Down method [7]. Following the introduction of the two main solutions currently used, the next section provides a brief overview of other tools used for application performance modeling, such as Hardware counters and static code analyzers. We will then present state of the art works that highlight the usefulness and importance of micro benchmarking in evaluating different systems performance. To close out the chapter we will discuss the open challenges we intend to tackle with this thesis work, and end with a brief summary.

2.1 Intel Core Micro-Architecture

Intel is one of the biggest companies in the world regarding micro-processors manufacturing. It has been in this market since their first release, the x86 processor, in 1978. Since then, their processors have come a long way in terms of computational capabilities and efficiency by mostly following Moore's Law when improving the manufacturing process. Together with the enhancements introduced across micro-architectures, this resulted in an improved performance throughout the years, at the cost of increased complexity.

In order to understand why these changes have such a big impact in the modeling of application performance, the micro-architectures should be studied and well known , not only to be aware of which component can be influencing the performance of the application but also to discover possible extensions that can be integrated in current state-of-the-art models. In the scope of this thesis, the core pipeline of Skylake based micro-architecture is considered, providing an overview regarding the main hardware resources that significantly impact application performance.

2.1.1 Intel Skylake micro-architecture

The Intel Skylake micro-architecture was launched by Intel in August 2015. It is based on a 14nm transistor technology, succeeding the Broadwell micro-architecture. Although 5 years have passed most Intel processors found in home computers and industries servers have micro-architectures based on Intel Skylake. Thus the core pipeline is very similar apart of some minor improvements, such as support for AVX512 instructions, higher number of cores and a new mesh interconnect, resulting in higher performances. For this work two different processors were used, an Intel i7 6700K [8] and an Intel Xeon Gold

6140 (Skylake-SP [9, 10]). The next paragraphs will explore the last machine (the Xeon Gold 6140).

Considering the similarity between the core pipelines of Skylake processors we will only present the pipeline of Intel Skylake-SP, shown in Figure 2.1. In general, the core pipeline can be divided in two main sub-systems, i.e., Front End (FE) and Back End (BE). These parts are separated by the Instruction Decode Queue (IDQ), which will be described in later on this chapter. It is important that the FE can provide a steady stream of decoded instructions to the BE to avoid starving it. For example, if the IDQ is left empty and the BE scheduler does not have instructions to send to the execution units, the pipeline will be stalled until instructions are available again, resulting in a loss of performance.

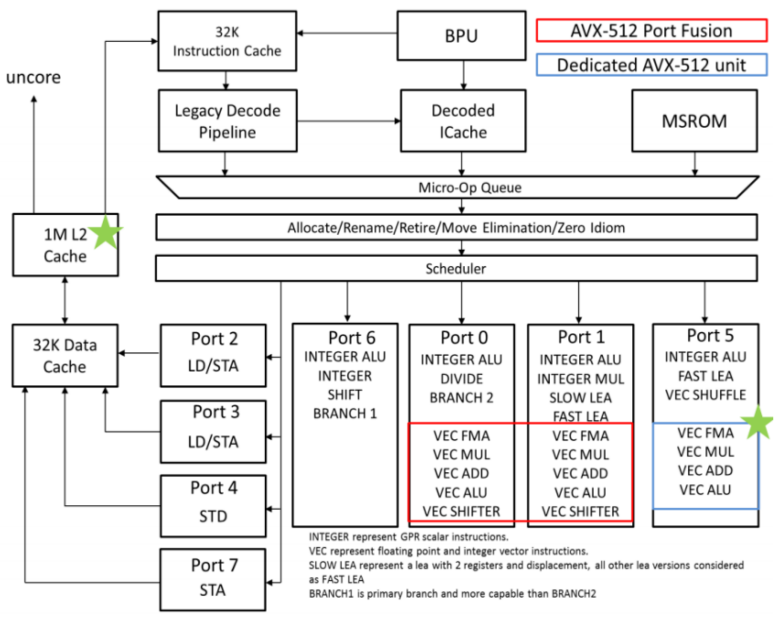

32K Instruction Cache
BPU
AVX-512 Port Fusion
Dedicated AVX-512 unit
uncore
Legacy Decode Pipeline
Decoded ICache
MSROM
Micro-Op Queue
Allocate/Rename/Retire/Move Elimination/Zero Idiom
Scheduler
1M L2 Cache
32K Data Cache
Port 2 LD/STA
Port 3 LD/STA
Port 4 STD
Port 7 STA
Port 6 INTEGER ALU INTEGER SHIFT BRANCH 1
Port 0 INTEGER ALU DIVIDE BRANCH 2 VEC FMA VEC MUL VEC ADD VEC ALU VEC SHIFTER
Port 1 INTEGER ALU INTEGER MUL SLOW LEA FAST LEA VEC FMA VEC MUL VEC ADD VEC ALU VEC SHIFTER
Port 5 INTEGER ALU FAST LEA VEC SHUFFLE VEC FMA VEC MUL VEC ADD VEC ALU
INTEGER represent GPR scalar instructions.
VEC represent floating point and integer vector instructions.
SLOW LEA represent a lea with 2 registers and displacement, all other lea versions considered as FAST LEA
BRANCH1 is primary branch and more capable than BRANCH2

Figure 2.1: Skylake SP Pipeline. [1]

2.1.2 Front End

The FE of Intel Skylake-SP is presented in Figure 2.2. This part of the core pipeline is responsible for fetching and decoding the instructions into micro-operations. Since it is a complex system that contains multiple components which can limit the performance of applications, to correctly identify which component impacts application performance, it is crucial to understand how each component works and their limitations. The FE, presented in Figure 2.2 [2], is responsible for fetching and decoding the instructions into micro-operations. There are three paths available to decode x86 instructions into micro operations: the Micro-Instruction Translation Engine (MITE), the Decoded Stream Buffer (DSB) and the Micro-Code Store Read Only Memory (MSROM) , all of them send decoded micro operations to an allocation queue called IDQ.

Regarding the MITE, the x86 instructions are fetched from the L1 Instruction Cache (32 KiB 8-Way associative) in a 16 byte window to the pre-decode component. The fetched instructions have variable length ranging from 1 byte to 15 bytes, depending on the instruction. For example, a simple instruction that does not use registers will not have any bytes with registers information and will consequently have less bytes than an instruction that uses two different registers. In order to facilitate the work on the rest

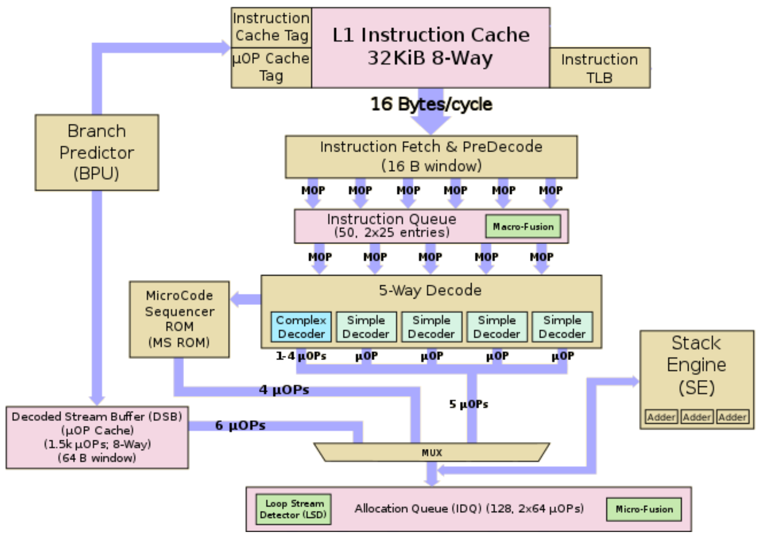

Instruction Cache Tag
µOP Cache Tag
L1 Instruction Cache 32KiB 8-Way
Instruction TLB
16 Bytes/cycle
Branch Predictor (BPU)
Instruction Fetch & PreDecode (16 B window)
MOP
Instruction Queue (50, 2x25 entries)
Macro-Fusion
MicroCode Sequencer ROM (MS ROM)
5-Way Decode
Complex Decoder
Simple Decoder
1-4 µOPs
µOP
4 µOPs
5 µOPs
6 µOPs
Stack Engine (SE)
Adder
Decoded Stream Buffer (DSB) (µOP Cache) (1.5k µOPs; 8-Way) (64 B window)
MUX
Loop Stream Detector (LSD)
Allocation Queue (IDQ) (128, 2x64 µOPs)
Micro-Fusion

Figure 2.2: Skylake SP Front End. [2]

of the pipeline, the length of each instruction is detected at this pre-decode stage. These pre-decoded instructions, typically called macro operations, are sent to the Instruction Queue (IQ) at a maximum rate of 6 macro operations per cycle. This queue has 25 entries per thread and can perform macro operation fusion, i.e., fuse two macro operations into one complex macro operation (for example, by combining one compare with one jump instruction).

With this information we can already see one theoretical bottleneck for applications. Knowing that the maximum bytes fetched from the L1 is 16 bytes per cycle, the bigger the size of the instructions the lower the throughput of macro operations, implying that fewer instructions are being feed to the rest of the pipeline. In this scenario, if the instructions are long enough, the Pre-Decoder stage may affect the overall performance of the micro processor. The IQ can help alleviate these problems by serving as a buffer, for example, if the IQ is holding 10 macro operations and in the next 2 cycles the Pre-Decoder sends 2 and 3 macro operations respectively to the IQ, the Decoder will still receive from the IQ 5 macro operations on the next cycle, followed by another 5 macro operations the cycle after, and finally 5 (2+3) macro operations in the last cycle. In this example the IQ was holding enough macro operations so that when the throughput of the Pre-Decoder dropped it did not influence the throughput of the IQ to the Decoder. The IQ can send macro instructions to the instruction decoder at a maximum rate of 5 macro instructions per cycle.

The macro-operations contained in the IQ are sent to the instruction decoder. This component is responsible for decoding the macro operations into micro operations that have fixed length and can be interpreted by the BE. The decoder contains 4 Simple decoders, that can decode 1 macro instruction per cycle, and 1 Complex decoder for macro instructions that result in more than 1 micro operation. The complex decoder can decode one macro operation into a maximum of 4 micro operations per cycle. When the complex decoder is performing macro-operation decoding, there is one less simple decoder active per each extra micro operation the complex decoder outputs. For this reason, the instruction decoder is able to deliver a maximum of 5 micro operations per cycle to the IDQ. However, when macro operations correspond to more than 4 micro operations, these instructions are decoded by the MSROM. In this case the instruction decoder is deactivated and the macro operations are decoded and delivered to the IDQ by the MSROM, at a maximum rate of 4 micro operations per cycle. The instruction decoder will remain deactivated until the MSROM finishes decoding.

Besides the MITE and MSROM, micro operations can also be delivered to the IDQ through the DSB, which works as a L0 instruction cache. It contains 32 sets, 8-ways and it is inclusive to the L1 instruction cache. It stores the last micro operations decoded and issued by the MITE, with a maximum capacity of 1536 micro operations. Hence, when micro operations are delivered by the DSB to the IDQ, all the instruction decoding stages previously mentioned are avoided, decreasing the possibility of execution bottlenecks. The micro operations are stored along 256 lines, each line holding anywhere from 1 micro operation to 6 micro operations. The DSB lines are divided in groups with a maximum of 6 lines, each corresponding to code blocks aligned to 64B. In the scenario that a 64B block contains more than 36 micro operations, i.e., the maximum number of micro operations that can fit in 6 DSB lines, none of the micro operations of the block are stored in the DSB. Whenever the last micro operation of the current block has some of its bytes in the next block, the DSB waits for the next block and puts this instruction in the last line of the initial block. The filling strategy of the DSB has the disadvantage of some lines ending up partially filled with less than 6 micro instructions. Since the DSB can deliver one line per cycle, its maximum throughput corresponds to the maximum number of micro operations that can be stored in a single line, i.e., 6 micro operations per cycle. We can say that the DSB throughput is directly related to the fill ratio of its lines, and a code that leaves a lot of lines partially filled can result in a negative impact on the performance. The DSB not only allows to avoid all the decoding process, but can also improves the throughput significantly, specially for bigger instructions, since their throughput is usually limited by the MITE 16B window. For these instructions the DSB can be very useful to keep a high FE throughput, has long as the code being fetched fits in the DSB.

The IDQ is the queue that receives all the micro instructions, coming from either the MITE, the DSB or the MSROM, and can hold up a total of 64 micro operations per thread. The instructions emitted to the IDQ are analysed by the Stack Engine in search for operations that change the stack pointer. When it finds one of these instructions, such as PUSH and POP, it uses one of its three dedicated adders to increase or decrease the stack pointer, alleviating work from the Back End. Finally, micro operations are ready to be sent to the BE at a maximum rate of 6 micro instructions per cycle. The IDQ also contains the Loop Stream Detector (LSD) which detects loops of instructions that fit inide the IDQ. When it is activated, the LSD locks the loop, streaming always the same sequence of micro operations directly from the IDQ, avoiding fetching and decoding of instructions. While this is happening the rest of the front end is disabled. The streaming of micro operations stops when it reaches a branch miss predict. The LSD is turned off in Intel Skylake-SP.

The component that decides from which path (MITE, DSB or MSROM) the instructions are fetched is the Branch Prediction Unit (BPU). This hardware component predicts the next instructions that belong to the correct stream of instructions, even before a branch true path is known. This usually leads to a big increase in performance, since current BPUs have very good prediction ratios. However, in the case of applications that have a instruction stream that cannot be predicted by the BPU, the overall performance can be severely impacted by the BPU, since every time a branch prediction misses the core pipeline needs to be flushed and cleared, which represents a big overhead of cycles. Besides the instruction decoding and issuing process, fetching instructions from the memory subsystem can also lead to severe application bottlenecks, especially when considering the high complexity of current memory subsystems containing several memory levels and TLBs.

Furthermore, the memory subsystem of the Intel Skylake-SP micro-architecture contains three memory levels that can be used to store instructions, namely: L1 Instruction, L2 and L3. The L1 Instruction Cache (L1 ICache), which is 8-way set associative and can store a maximum of 32KiB, has a maximum bandwidth of 16 bytes per cycle. The L2 Cache is a 1 MiB 16-way cache and, unlike the L1 ICache, it is shared between instructions and data. The bandwidth between this cache and the L1 ICache is 64 bytes per cycle, which should be enough to avoid performance losses, since the MITE only fetches 16 bytes per cycle. Unfortunately there is usually a penalty in performance when fetching instructions from the L2, caused by the latency of bringing the instructions from the L2 to the L1 ICache. The last level cache, the L3 cache, has a maximum bandwidth of 64 bytes per cycle between itself and the L2 and it stores 1.375MiB (per Core) in a 11-way configuration. Considering the latency penalty of accessing this cache, we should expect to see a drop in performance when accessing the L3. Although it is not part of the micro-processor there is another memory level worth mentioning, the DRAM. This level can have different configurations with much bigger sizes but it will always have a big negative impact in the application performance since its bandwidth is lower than the caches with much bigger latency penalties. Other system memory components, such as hardrives and SSDs, offer even worse performance, and are almost never used by an application.

Besides the different memory levels, the TLBs can also play an important role when limiting application performance. The Instruction TLB (ITLB) in Skylake-SP is similar to a 8-way cache that facilitates the translation of virtual addresses to physical addresses. It can hold 128 entries for pages of 4KB, which means it can hold all the pages of a code with a maximum of 512KB of instructions. If a page is not present in the ITLB the processor will spend a lot of cycles translating the virtual address which will lead to a loss in performance. This should only happen if we have an application with a huge code size and/or we have a great number of jumps that hit different pages.

2.1.3 Back End

The BE, illustrated in Figure 2.3 [2], is the OOO part of the processor where the instructions are executed. To attain an efficient OOO execution, the BE relies on several components, such as, executions units and register tables. These components have limitations that can become the bottleneck of applications. The first component of the BE, the one that receives micro instructions from the FE, is the Re-Order Buffer (ROB)

The ROB receives up to 6 micro instructions per cycle from the IDQ and can perform 3 different optimizations, namely: "Move Elimination", "Ones Idioms" and "Zeroing Idioms" . These optimizations are performed before any renaming or execution happens, in order to prevent the waste of resources to execute micro operations whose the result can be predetermined. While "Move Elimination" focus on eliminating register to register moves, "Ones Idioms" and "Zeroing Idioms" optimize instructions in which the result would be either all ones or all zeros respectively, for example a XOR of a register with itself (Zeroing Idiom). After the optimizations are completed micro-architecture registers are mapped onto the physical registers, available scheduler ports are determined and register naming is performed through the Register Alias Table (RAT), that can rename up to 4 micro operations per cycle. The renaming of registers is important to identify data dependencies and data sources, so that the micro-processor can perform optimizations, like the forwarding of operands, and avoid pipeline stalls. The ROB also interacts with the Branch Order Buffer (BOB) which guarantees that in the case of a miss speculation the processor can invalidate its state and role back to a previous valid state. The BOB can hold up to 48 micro instructions that are in the same order has they were originally fetched. From the ROB, the micro operations are delivered to the Scheduler.

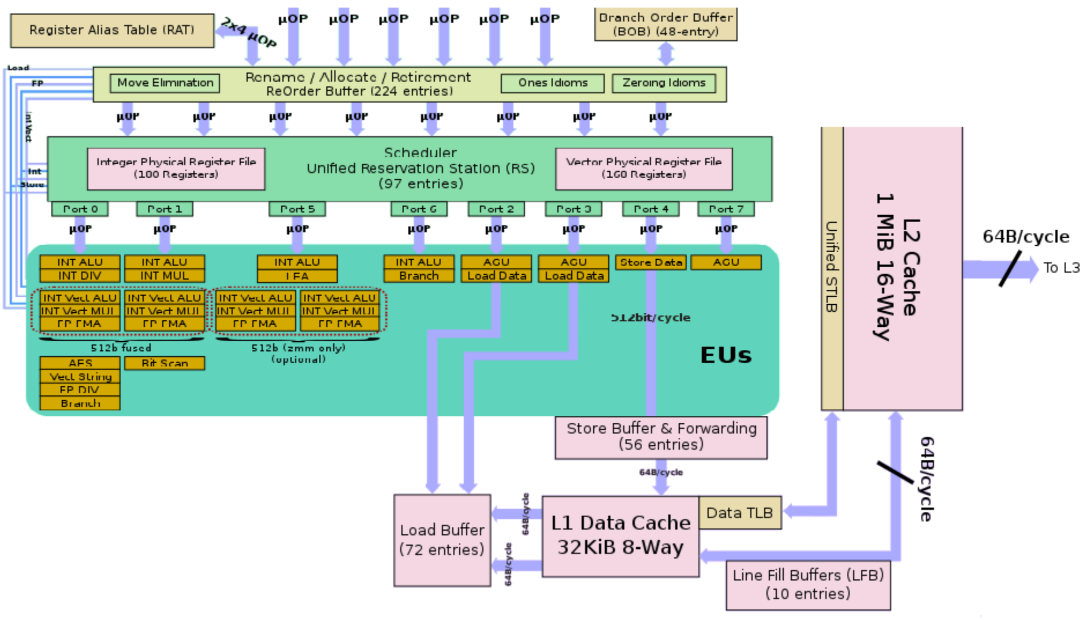

Register Alias Table (RAT)
2x4 µOP
µOP
Branch Order Buffer (BOB) (48-entry)
Load
FP
Int Vect
Int
Store
Move Elimination
Rename / Allocate / Retirement ReOrder Buffer (224 entries)
Ones Idioms
Zeroing Idioms
Scheduler Unified Reservation Station (RS) (97 entries)
Integer Physical Register File (180 Registers)
Vector Physical Register File (168 Registers)
Port 0
Port 1
Port 5
Port 6
Port 2
Port 3
Port 4
Port 7
INT ALU
INT DIV
INT MUL
LEA
Branch
AGU
Load Data
Store Data
INT Vect ALU
INT Vect MUL
FP FMA
512b fused
512b (zmm only) (optional)
AES
Vect String
FP DIV
Bit Scan
512bit/cycle
EUs
Store Buffer & Forwarding (56 entries)
64B/cycle
Load Buffer (72 entries)
L1 Data Cache 32KiB 8-Way
Data TLB
Line Fill Buffers (LFB) (10 entries)
Unified STLB
L2 Cache 1 MiB 16-Way
64B/cycle
To L3

Figure 2.3: Skylake SP Back End. [2]

The scheduler can receive up to 8 micro instructions per cycle from the ROB and has 97 entries (shared by two threads), 180 integer registers and 168 vector registers. It will hold a micro instruction until all the operands are available for the operation and the Execution Units (EU)s needed are free. Once a micro instruction is ready for execution, the scheduler sends it to the respective EU, through one of its 8 ports. The Scheduler can output one micro instruction per cycle per each of its ports.

A new EU was added in Skylake-SP micro-architecture that is able to perform Advanced Vector Instructions 512 (AVX512) instructions. These instructions can increase the performance of the Intel Skylake-SP up to 2x the performance of previous Intel Skylake architectures that only supported at most AVX instructions. This comes from the ability of the new EU to perform the same vector operations at the same rate but with registers with double the size. The scheduler sends this type of instructions to port 5, that has a dedicated AVX512 execution unit, or to port 0, where two execution units able to perform calculations with 256 bits registers (one on port 0 and one on port 1) are combined to compute the 512 bit operation. For integer operations the scheduler ports that link to integer units are the ports 0, 1, 5 and 6, to perform memory operations the scheduler can dispatch memory instructions to any of its following ports: 2, 3, 4 and 7. While port 2 and 3 are reserved for load operations, port 4 is used to perform store operations and port 7 reserved for store address calculation. After executing any instruction in the EUs the BE is capable of retiring them at a rate of 4 micro operations per cycle, i.e., a maximum Clockticks per Instruction (CPI) of 0,25 [11, 12].

The memory sub-system of the BE, can be observed in Figure 2.3 [2], is responsible for feeding both the loads and stores instructions. The memory sub-system is capable of sustaining two memory reads and one memory write per clock cycle since it has two available ports for loading instructions (ports 2 and 3), and one for writing(port 4). Each memory operation can fetch a maximum of 512-bits. The memory hierarchy, similar to the FE, is divided in 3 levels: a private L1 Data cache with 32KB (8-Way associative), which can perform two loads (2 x 64B) and a store per cycle (64B); the L2 cache, shared with the FE, with 1 MiB (16-Way associative) with a bandwidth of 64B per cycle; and a L3 non-inclusive shared cache between the cores with a size of 1,375 MB per core (the size of the L3 is related to the number of cores) with a bandwidth of 64B per cycle. With these specifications in mind, it is possible to calculate the maximum rate that integers can be loaded and stored for L1 cache. For example, a double word integer occupies 32 bits, using vector instructions and AVX-512 registers 32 integers can be loaded and 16 can be stored per cycle. However, for the remaining memory levels, the sustainable memory bandwidth can only be obtained through an accurate micro-architecture benchmarking.

All these micro architecture characteristics should be known before start micro benchmarking and evaluating application performance. This knowledge allows for a better use and understanding of the components and grant a more comprehensive and critical view of the micro benchmarks results.

2.2 Micro Benchmarking

Micro benchmarking is the process through which we can experimentally obtain the characteristics of an architecture and its components, from their limitations to their capacities and performance. By evaluating the performance of a hardware component under diverse execution scenarios it is possible to uncover its impact on the application performance. Due to the ability of micro-benchmarking methodologies to accurately expose the micro-architectural bottlenecks that contribute to reduce application performance, several performance models, such as CARM are derived based on these methodologies. Moreover, to assess the characteristics of real-hardware, micro-benchmarking are also usually validated by relying on hardware counters in-built on current processors. These hardware counters are not only essential to verify the accuracy of micro-benchmarking, but can also be used to construct accurate performance models that provide a set of metrics to hinting which components limit application performance. Other areas unrelated to micro-processors also take advantage of micro-benchmarking to analyse their platforms and tools. Such is the case for areas like cloud computing, software development, GPUs, etc. The broad use of micro-benchmarking is a good indicator of their usefulness and importance.

As mentioned previously, one important instrument widely used, including in micro-benchmarking, are hardware counters. These are registers incorporated in the micro processor that store values related to many different metrics related to hardware activities, from the number of clock cycles to the power consumption. They allow for a low level analysis (obtaining micro-architecture metrics) yet they can differ from processor to processor. It requires the programmer to initialize them in the code with the correct set of parameters that change depending on the processor (even if from the same company) and are usually found in a document provided by the micro processor company [6]. After initializing the counters they have to be read from and then meaningful metrics have to be produced from those readings, since most of the counters alone do not provide great insights.

2.2.1 Performance Models Based on Benchmarking and Hardware Counters

There are several state-of-the-art models that can be used to predict and analyse the performance of applications in modern processors, each of them have a distinct approach [4, 7, 13, 14]. The most adopted models rely on micro-benchmarking and hardware counters. One of the most popular solutions that rely on micro-benchmarking and hardware counters are the roofline modeling approaches, in particular ORM and CARM. In the next pages we will present these two models.

Original Roofline Model (ORM) [3] relates the maximum floating point performance of the processor, F_p in Floating Point Operations per Second (Flops), with the maximum DRAM sustainable bandwidth, B_D in β_D/s (β_D refers to the bytes transferred between DRAM and Last Level Cache (LLC)), and with the operational intensity of the application, I in Flops/β_D. I gives a sense for how much DRAM bandwidth the application will need. ORM considers that the application spends most of its time performing computational operations or memory transfers and that these overlap in time due to OOO nature of modern processors.

Due to its principles, ORM contains two regions, i.e., the compute bound and memory bound regions, as it can be observed in Figure 2.4. The compute region is limited by a horizontal roof which corresponds to F_p, while the performance on the memory region is bounded by the maximum bandwidth between L3 cache and DRAM, represented by the slanted roof in Figure 2.4. The point in the graph where the two roofs intercept is called the ridge point. This point provides some insight on the overall performance of the computer. If it is too far to the right, it means that in order to achieve F_p an application has to have very high I which can be difficult to program, if it is far to the left means that almost every application will be able to achieve peak performance.

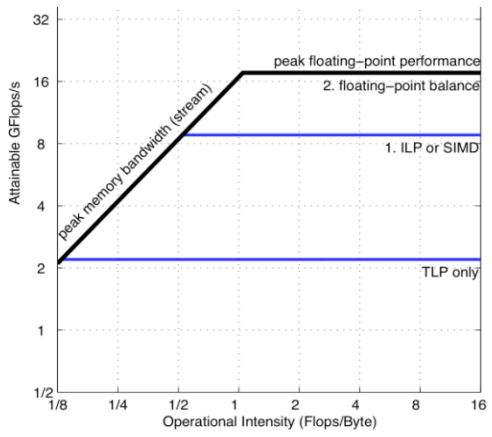


Figure 2.4: Original Roofline Model in OpteronX2. [3]

By characterizing applications in this model, it is possible to extract useful information for software developers to improve application performance. The position of the application on the graph tells the programmer what is the maximum performance their application could get, and if it that performance would be memory bounded (application is under the memory roof) or compute bounded (application under computational roof). It also provides a sense of where should the developer focus to improve the application, if the application has a very high I but is far away from the computational roof, the programmer should spend most of its efforts on improving the code to allow better resource utilization, for example, vectorizing the application code to increase computational performance. On the other hand, if the application has a low I and it is far away from the memory roof, the programmer should invest more time in improving accesses to memory, for example by doing software pre-fetching.

Since the model only considers the memory traffic between L3 cache and DRAM, it is not ideal because applications access data from different memory levels and can be bounded at different memory levels. One solution is the utilization of a different roofline modeling approach, i.e., the Cache-Aware Roofline Model (CARM), which considers all the memory levels contained in the memory hierarchy of modern processors.

Differently from ORM, CARM evaluates both memory bandwidth and floating point performance from the core point of view. For this reason it accounts for all data transfers, not only accesses to the DRAM, and provides the different values of bandwidths ($B_{L1\to c}$, $B_{L2\to c}$, $B_{LLC\to c}$, $B_{D\to c}$). Figure 2.5 [4] illustrates this difference between ORM and CARM.

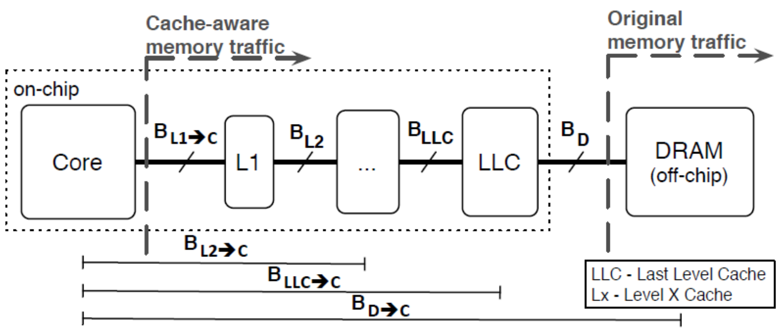


Figure 2.5: Memory Traffic CARM Vs RM. [4]

Another difference between ORM and CARM is the way intensity is defined. CARM considers the Arithmetic Intensity (AI) and is dependent on the number of floating point operations ϕ, and the number of bytes transferred β, AI($\frac{\phi}{\beta}$), seen from core point of view. With a different application intensity, and different bandwidths (B_y), the CARM has a maximum attainable performance, $F_{a,y}(AI)$, given by

$$F_{a,y}(AI) = min(B_y AI, F_p) \tag{2.1}$$

With this new equation for $F_{a,y}(AI)$, the CARM provides performance limits (roofs) for each memory level, as is demonstrated in Figure 2.6. This contrasts with the ORM that only contains one roof in the memory region as we can see in Figure 2.4. It is possible to apply ORM to other memory levels by using different bandwidth instead of B_D but it would require to construct and analyze the model many times, one for each memory level.

The point were the curves intercept is called ridge point, just as in the ORM, and provides the same insights as discussed in previous paragraphs. The horizontal roofs of ORM and CARM are identical, since they are related to computational bounds.

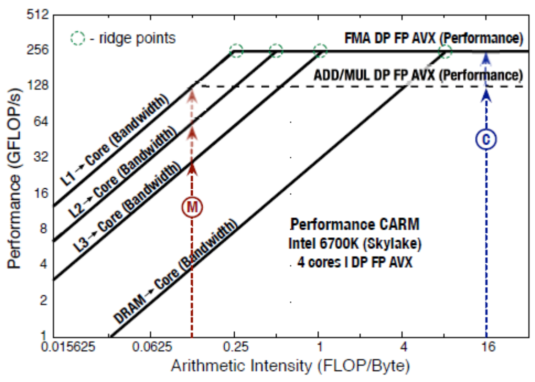


Figure 2.6: Example CARM Model. [5]

There are other differences between ORM and CARM that are should be pointed out. First, due to how ORM calculates its operational intensity, I, accounting only for bytes transferred between the LLC and the DRAM, when the problem size grows (increasing the number of iterations) it affects the I of the application, possibly moving the application to a different bound region. In CARM, application AI remains constant when the problem grows in size, thanks to the way it is calculated (accounting for all data transfers). Secondly, since in ORM changes in problem size can shift the application to a different region, it is possible that by analysing the application ORM the user concludes that there are optimizations that can be made, when in reality there is not, the application is just in the wrong zone.

Both ORM and CARM need to consider a multitude of limitations inherent to a micro-architecture, from memory bandwidth to the number of scheduler ports. CARM uses micro benchmarks to obtain the bandwidths for the different levels of the memory hierarchy and the maximum computational throughput for different instructions. It is with these values that its roofs are calculated and plotted, and since every micro-architecture is unique, with components of different specifications, the model need to execute a set of micro benchmarks on every processor before creating the model an evaluate the application performance. The CARM model has already been incorporated in Intel Advisor. Intel Advisor is an Intel high-performance framework that provides insight into code vector optimization, memory access patterns, thread prototyping, flow graph analysis and Roofline analysis [15]. It provides CARM modeling for applications along with some guidelines on which part of the code is affecting performance the most and how should it be optimized. The CARM chart itself provides useful information such as: values of the different computational roofs, bandwidths from different levels memories, applications performance and application arithmetic intensity. These provide the user insights on the amount of performance being lost.

Besides models based on micro benchmarking, state-of-the-art methods that rely on performance counters can also be a viable mechanism to identify the main bottlenecks that limit application performance. This is the case of Top Down method [7], which uses a wide set of performance counters presented in modern processors to identify the possible bottlenecks that affect application execution. To perform this task, this method provides an in-depth and hierarchical structure, which decouples application execution time in several nodes, each representing a potential bottleneck.

At the top of the hierarchy there are 4 main nodes. These nodes will be flagged if they represent a bottleneck for the application, so that the user knows what path of the hierarchy to follow in order to get more details regarding the bottlenecks. The hierarchic view of this method is displayed in Figure 2.7 The top four nodes are:

- Frontend Bound - Highlights performance issues at the initial stage of the pipeline, the Front End. The rate that the front end feeds instructions to the back end can be a major performance problem. The Top Down method divides frontend bound in two other subcategories: the fetch latency and the fetch bandwidth. The first relates to performance bottlenecks caused by cache misses, like a instruction cache miss. The last refers to performance bottlenecks caused by inefficiency in the instruction decoders.

- Bad Speculation - Reflects time wasted when a branch misprediction occurs, including the time the processor was executing operations of the wrong path (that have to be discarded) and the time the processor takes to recover to a stage before the miss prediction. High values in this domain should be considered a red flag by the user, since the amount of time lost to perform a flush of the pipeline is huge. Bad speculation divides into two subcategories: branch miss predictions, performance lost due to wrong predictions, and Machine Clears which requires also a pipeline flush.

- Retiring - This node represents the time spent retiring micro operations. A high percentage of application time spent in this node is what we would want. High percentages of retiring means the processor is working at his the maximum, and it is mostly bounded by the capacity of the micro-architecture to retire instructions. A high retiring value also means that if the number of operations per instruction can be improved (for example by vectorizing the code) the performance can be improved, and so, this node has sub-nodes that allow the user to have a sense of the type of computations being performed (scalar or vector) and if could be improved.

- Backend Bound - This node divides into two big sub-nodes: the core bound, and the memory bound. The memory bound represents time spent in the pipeline performing memory accesses (or waiting for memory accesses). The memory node is then divided in smaller nodes that represent different memory levels, from L1 cache to external memory. Each memory level has its sub-nodes that provide information regarding latency issues, for example when memory accesses are sporadic, or related to bandwidth, when the memory has not enough sustainable bandwidth. The Core bound represents time spent in the execution units of the pipeline, the sub-nodes provide information port utilization.

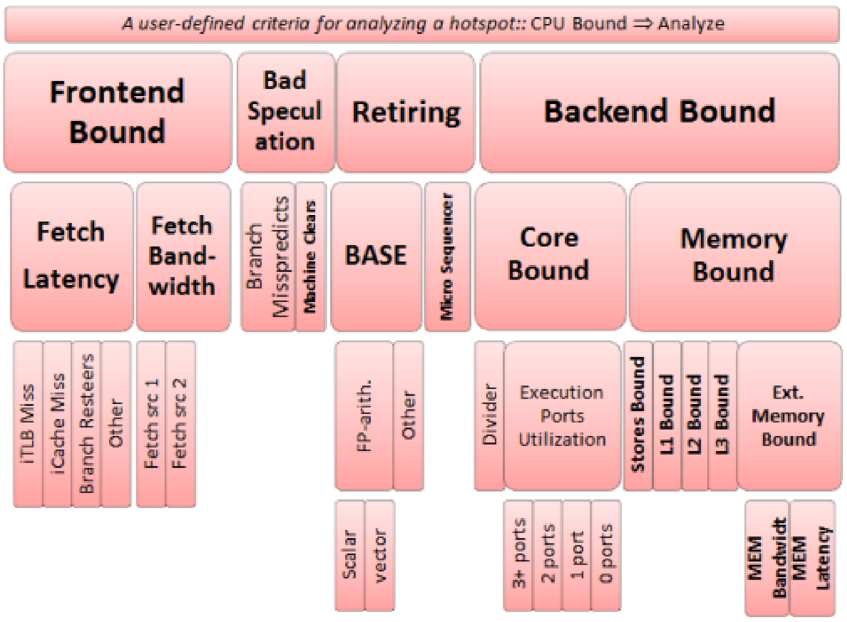


Figure 2.7: Top Down Hierarchy

When using this method to analyse application performance is important to compare only the categories in the same hierarchical level and from the same group. For example, the user can compare Fetch Latency with Fetch Bandwidth, but it should not compare Fetch Latency with Core Bound or with ICache Miss. The Top Down Method can be used to analyse applications in Intel VTune. Intel VTune is an Intel product which implements the Top Down Method, providing the user with a simple graph with the metrics related to it. This means the user can see in a simple way, the percentage of the application that is Frontend Bound, Backend Bound, etc, and can even go into more detail down the hierarchical path to discover what is causing damage to the application performance. VTune is also a helpful tool for memory access analysis, threading performance analysis, software sampling, etc [16].

2.3 State of the Art Approaches

Despite its undeniable value for understanding of microprocessors and their behaviour, micro benchmarking is also used in other areas. There are several state of the art works that utilize benchmarks in order to evaluate and/or improve their systems. In the next table some of this works are mentioned and briefly detailed: Micro benchmarking is very useful to the understanding of micro-processors and how they function in different cases, but this is not the only use case for micro benchmarks. There are several state of the art works that utilize benchmarks in order to evaluate and/or improve their systems. In table 2.1 some state of the art works mentioned.

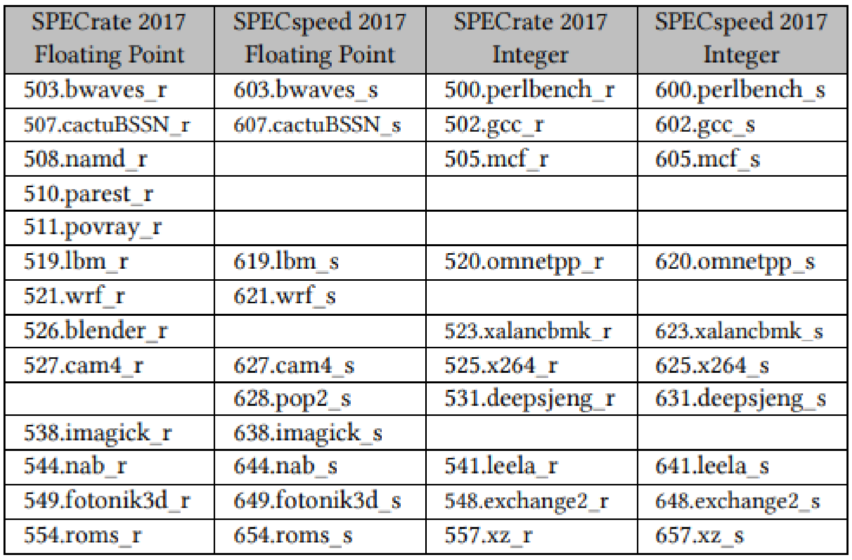
Table 2.1: State of the art works

Article/Source	Year	Use Case
[17]	2018	Micro Benchmarking used to construct faithful models of the latency, throughput and port usage of x86 Instructions.
[18]	2010	Micro benchmarks and benchmarks used to model application performance in Virtualized Environments
[19], [20]	2017, 2018	Micro benchmarks and Roofline used to gather important information on a supercomputer and create guidelines for performance optimizations
[21]	2010	Micro benchmarks used to model application performance in GPUs
[22]	2018	Micro benchmarks used to model application performance on Cloud services
[23]	2010	Micro benchmarks used to provide insights of applications performance on 3 supercomputers
[24]	2017	A set of benchmarks representative of real world applications used to evaluate processor performance
[25], [26]	2009, 2016	Mechanistic performance model focused on development of micro-architectures
[27]	-	A performance analysis tool to statically measure performance of machine code

There are multiple works where micro benchmarks are used as the foundation of the work developed thanks to the detail information micro benchmarks can provide. For example, in [17] the authors create a series of micro benchmarks to characterize the latency, throughput and port usage of instructions on Intel micro-architectures. Some of the micro benchmarks results shined a light on information not publicly available, while other provided more accurate values for some metrics. This information can then be used by other tools, such as performance-analysis tools, to predict, explain and optimize application performance. The work presented in [18] focuses on application performance in virtual environments. For these environments architecture-specific and performance-counters based models are not the best option for analysing application performance, due to resource competition by other applications virtual machines (VM) and the lack of information the developer may have regarding VM applications, operating systems, etc. To overcome this issue, this work uses a set of micro-benchmarks to gather information on the memory system of a virtualized application and its relation with other components such as I/Os. On other works, namely [20] and [19], micro-benchmarking was used to uncover key micro-architectural specifications of China's SW26010 processor that is used on the TaihuLight supercomputer. The results of this work provided important information for performance optimization and modeling on this supercomputer, such as: instruction latencies, the issue order of instructions (in order vs out of order), static and dynamic routing, different bandwidths, etc. There are also works related to GPU performance that take advantage of micro-benchmarking, such as [21]. Unfortunately micro-benchmarking techniques used for CPUs do not translate to GPUs architectures due to the high level of parallelism in them. In order to create an application performance model for GPUs, the authors of this work design a suite of micro-benchmarks in order to gather the exact values of GPU architectural parameters and other features that vary depending on program characteristics. With this new suite they highlighted some GPUs bottlenecks and provided not only better insights on the GPUs components but also some optimization guidelines. The next work mentioned in table 2.1 is on the area of cloud application modeling [22]. This work focused on developing a methodology using micro-benchmarks to profile applications and predict their performance on cloud services. The methodology was then tested with scientific applications, with positive results, achieving a prediction error inferior to 10%. It is also highlighted the importance of using benchmark-based metrics instead of specification-based to improve estimation accuracy. The work done in [23] uses micro-benchmarks to compare application performance in three different supercomputers: Intrepid, Ranger and Jaguar. The results from micro benchmarking does not only allow a comparison of the supercomputers regarding characteristics like latencies, bandwidths, etc. but also provides important insights regarding application performance. Some models, such as the Interval Model described in papers [25] and [26], do not use values obtained through benchmarks and instead use theoretical or estimated values. These two models in particular focus on providing a helpful insight into the processor performance, focusing more on how changing some aspects of the processor, like the pipeline width, can influence overall performance. In this case by not having to run simulations or a ton of micro benchmarks they provide information in a much shorter time period, which help architects make multiple experiments with different configurations. The trade-off is that typically they have a bigger margin of error, so it is not the best approach to evaluate applications performance and their bottlenecks. There is also work been done with other tools that provide alternative options for evaluating performance. For example the LLVM-MCA (low level virtual machine - machine code analyzer) [27]. This performance analysis tool not only provides estimations of code execution time in a specific target processor but also helps identifying bottlenecks and performance issues.

For all the importance that micro benchmarks have in today's works, there is also need to have standard benchmarks publicly available to test different platforms, processors, components, etc. in order to enable comparisons among them. Nowadays there is a vast number of applications used around the scientific world which can be very different from each other, from the size of the problem to the type of operations they perform. In order to evaluate processor performance the benchmarks used should be representative of the applications seen in the real world. This is the case of SPEC CPU benchmarks which were created with the objective of emulating present and future real life applications. The last iteration launched is the SPEC 2017 [24], launched in 2017. It has a total of 43 benchmarks, divided in four suites: SPEC speed integer, SPEC speed floating point, SPEC rate integer and SPEC rate floating point. A list of all the benchmarks per suite can be seen in 2.8

All the benchmarks have three sets of input data: test, train and ref. The test is the smallest input and has the lowest execution time. Typically this input only server as a quick test to confirm all requirements are in order. The train set is the middle data set, which can be used to to provide information on bottlenecks and performance problems. The ref set has the largest input, making it the



SPECrate 2017 Floating Point	SPECspeed 2017 Floating Point	SPECrate 2017 Integer	SPECspeed 2017 Integer
503.bwaves_r	603.bwaves_s	500.perlbench_r	600.perlbench_s
507.cactuBSSN_r	607.cactuBSSN_s	502.gcc_r	602.gcc_s
508.namd_r		505.mcf_r	605.mcf_s
510.parest_r			
511.povray_r			
519.lbm_r	619.lbm_s	520.omnetpp_r	620.omnetpp_s
521.wrf_r	621.wrf_s		
526.blender_r		523.xalancbmk_r	623.xalancbmk_s
527.cam4_r	627.cam4_s	525.x264_r	625.x264_s
	628.pop2_s	531.deepsjeng_r	631.deepsjeng_s
538.imagick_r	638.imagick_s		
544.nab_r	644.nab_s	541.leela_r	641.leela_s
549.fotonik3d_r	649.fotonik3d_s	548.exchange2_r	648.exchange2_s
554.roms_r	654.roms_s	557.xz_r	657.xz_s

Figure 2.8: SPEC CPU2017 benchmarks

one that consumes more time to run, and it is used to analyse processor performance, being the only set acceptable for submitting official results. To run all the benchmark suites with the ref data set it takes a great amount of time run, and some of these benchmarks might provide similar insights onto processor performance. In order to reduce the time running the benchmarks, there is some work being done in order to obtain the subset of benchmarks that offer the best representative subset [28, 29]. There are also state of the art works focused on creating more workloads for the benchmarks [30], allowing to minimize the risk of the hidden learning problem, where the target of the evaluation is optimized to these data sets and will have better performance results that may not correspond to a better machine in terms of performance.

A lot of work is done in characterizing the SPEC benchmarks [28, 29, 31, 32]. This work provides useful information on the type of operations Integer applications are most prone to, where are the bottlenecks usually situated and what are the focus points that affect performance. From these works some conclusions can be taken about the Integer applications: around 35% of operations are memory operations; around 35% of operations are computational operations; 10% of operations are Stores; around 18% of operations are Branches (considerably more than FP applications); integer applications are mostly bounded by FE stalls and Memory Bound stalls. These analysis of SPEC CPU Integer benchmarks demonstrate the need to characterize and model performance problems related to the FE, especially for integer applications.

2.4 Open Challenges

So far we have presented different approaches and tools used to analyze and characterize application performance, which allow a developer to discover the bottlenecks of an application and optimize it for the processor where it is running. However there is a crucial piece missing, a detailed analysis of the FE components and what limitations it imposes on the overall application performance.

The Top Down method takes into account FE problems based on FE stalls emitted, but it does not show the affect those stalls have in the performance or even what performance we could achieve if we improved our FE performance. On the other hand we have CARM that does a very good job in showing the developer where are the bottlenecks of the application and the impacts on the performance, but that does not take into consideration the limitations and problems of the FE, which means that using CARM on applications with problems on the FE will not provide useful information regarding optimization. To better justify these claims, lets look at example of an application bounded by the FE.

To find such application we tested some SPEC CPU Integer Speed benchmarks utilizing the train input set, and analyzed the application using Intel VTune to obtain the Top Down method, and Intel Advisor to obtain a CARM chart. From the applications tested we pick the one that better illustrated a FE bound condition with very low BE problems in order to only focus on FE problems and avoid mixing both. After an overview of all tested applications we picked the 600_Perlbench_s to present and discuss. This application, according to experiments performed in [28], has around 44% memory instructions, 18% Branch instructions and 38% Computational instructions. Even though these results do not come from the same machine they can expected to be approximated to what we would obtain since the architecture of our machine (Skylake SP) is very similar to the one used in the paper. With these percentages of instructions, at first look, we can expect the application to be computational bound or memory bound, since there is a big percentage of these two types of instructions. The percentage of branches instructions is also high enough to consider it a possible issue that can affect performance, if the application has a lot of mispredictions.

Looking at the roofline in Figure 2.9 we see the 5 bigger functions/loops of the application and where the overall application would stand (represented by a black cross). From this we can infer that the application is bounded by memory (being in the memory zone and close to memory roofs), more specifically by the DRAM and L3. For a Top Down analysis we picked one of the biggest functions, namely the function s_regmatch, not only because, according to CARM, it has the most impact in overall performance, but also due to the fact that, according to Top Down, it is mostly FE bounded. The values of the Top Down method in Figure 2.10 and 2.11 tell us that the function is limited by the FE, and that it also loses a considerable amount of performance due to bad speculation. It is worth nothing that not all subcategories contain values due to the low information gathered from those subcategories that are not enough to provide a confident result. If we perform a critical analysis of these results we see some of the models weaker points. In CARM we would assume the memory is the the main bottleneck, since s_regmatch is in the memory zone under the L2 roof, and would not point us to problems in the FE at all (the model does not take these into account). On the other hand we have the Top Down which points to problems in the FE latency and bandwidth but it does not show what that relates to in terms of performance lost or what performance we could achieve if we made the correct optimizations. This is important information since an application can be bottlenecked by a specific component but there may not be any optimization possible, for example if the component is working at maximum throughput.

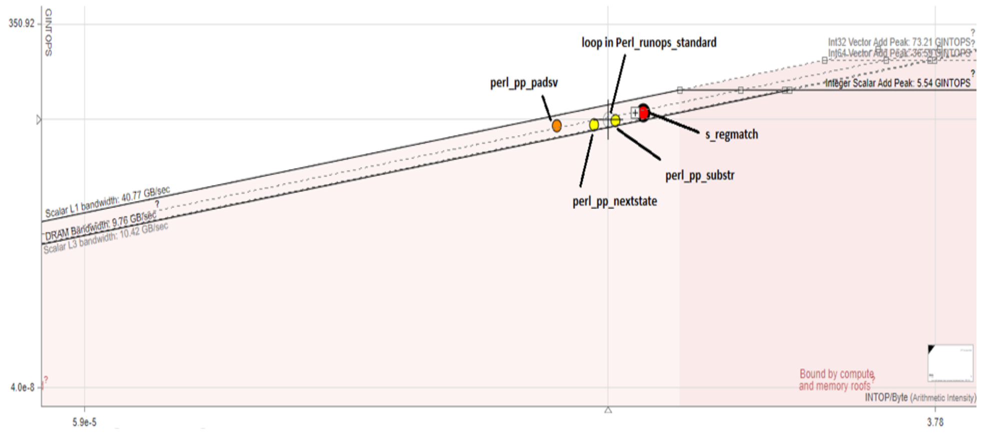


Figure 2.9: Roofline 600_perlbench_s

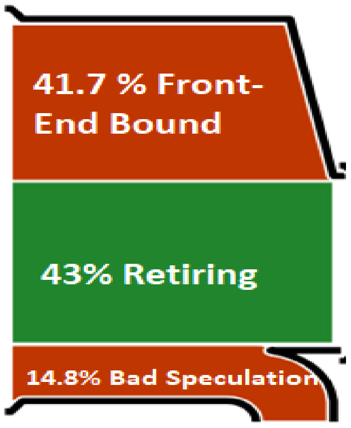
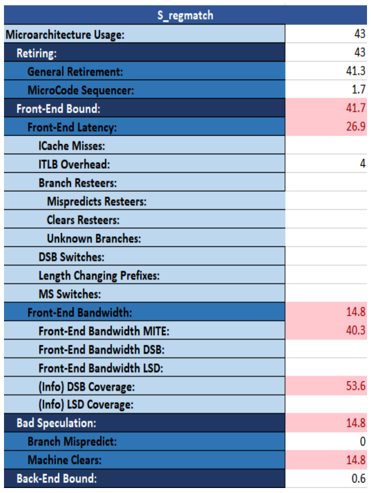


Figure 2.10: Top Down perlbench_s - s_regmatch



S_regmatch	
Microarchitecture Usage:	43
Retiring:	43
General Retirement:	41.3
MicroCode Sequencer:	1.7
Front-End Bound:	41.7
Front-End Latency:	26.9
ICache Misses:	
ITLB Overhead:	4
Branch Resteers:	
Mispredicts Resteers:	
Clears Resteers:	
Unknown Branches:	
DSB Switches:	
Length Changing Prefixes:	
MS Switches:	
Front-End Bandwidth:	14.8
Front-End Bandwidth MITE:	40.3
Front-End Bandwidth DSB:	
Front-End Bandwidth LSD:	
(Info) DSB Coverage:	53.6
(Info) LSD Coverage:	
Bad Speculation:	14.8
Branch Mispredict:	0
Machine Clears:	14.8
Back-End Bound:	0.6

Figure 2.11: s_regmatch detailed Top Down

Looking at the subcategories of the Top Down method in Figure 2.11 we can see different components of the FE with the potential to impact the overall performance, namely: ICache, ITLB, BPU, DSB, MS and MITE. All of these deserve to be investigated in detail in order to fully model the FE and create a model that incorporates both BE and FE bottlenecks.

In this thesis we will work on micro benchmarking and understanding the behaviours and limitations of the MITE, DSB and ICache. To highlight some of these overlooked components and how they can impact application performance lets take a deeper look at the MITE, DSB and ICache with some examples.

At the beginning of this chapter we described the MITE in great detail, and talked about the 16B window at the start of the pre-decode stage. The size of this window can actually be a bottleneck for some applications. For example, with big instructions, like 8B instructions, the throughput of the pre-decode stage will be 2 operations per cycle, which can drastically reduce the overall throughput of the micro-processor, since this can retire a maximum of 4 micro operations per cycle. On the other hand, if the instruction size is small enough to fit more than 6 instructions in one window, for example 7 instructions, the pre-decoder will output 6 macro operations in one cycle and 1 macro operation in an extra cycle, limiting the throughput to 3.5 ($\frac{7}{2}$) macro operations per cycle. Therefore also limiting the overall throughput to 3.5 micro operations per cycle. To complicate things further, there can also be a negative impact on performance whenever the last instruction in the 16B window is splitted between the current and the next window. For example, a code made of 3B instructions would be able to fetch 5.33 instructions per window. However, the pre-decoder would output 5 macro instructions in the first cycle and then spend and an extra cycle to fetch the remaining bytes of the split instruction. This behaviour would affect the following windows, with the second window having 2 less bytes to decode and resulting in a similar behaviour, outputting 4 macro operations in the first cycle and 1 macro operation in an extra cycle. Finally in the third window, now with 1 less byte to decode, this behaviour would reset and 5 macro operation would be sent to the IQ in one single cycle. In total we would get 16 macro operations in 5 cycles which gives a throughput of 3.2 macro operations per cycle, instead of 5.33 ($\frac{16}{3}$) or 2.66 ($\frac{\frac{16}{3}}{2}$).

Regarding the DSB it is important to understand its limits and how it can be utilized to improve application performance. To give an example, if we look at instructions of 8B, the same size we used early for the MITE example that had a maximum throughput of 2 micro operations per cycle, in the DSB we would have 8 micro instructions ($\frac{64}{8}$), that would use 2 lines of the DSB, one filled with 6 micro instructions an the other with only 2, resulting in a throughput of 4 micro instructions per cycle. Just by using the DSB we could double our maximum performance. Another possible scenario where we could be loosing application performance is if the DSB lines are not being filled to their maximum. Sometimes this might be inevitable, but other times the developer might be able to shuffle instructions around to maximize the DSB fill ratio. One last component that is not receiving the proper attention and that can severely affect application performance is the ICache and all the memory hierarchy for instructions. Despite the fact that a lot of work has been done regarding the impact of memory accesses on application performance, all of it focus just on the Data side and not on the Instruction side. For code that works with a lot of data and a small number of instructions this should be enough, but when we start to have applications with a big number of instructions the accesses to memory to fetch instructions might become a bottleneck to the application.

All of these details are mostly overlooked by today's tools, and deserve to be studied and incorporated in new solutions. On this thesis we pretend to perform a detailed micro benchmarking of the FE of the Skylake micro architecture in order to gain a deep knowledge on its limitations. With this knowledge we should be able to better understand how an application is limited by the FE, achieve some early predictions of performance accounting for FE bottlenecks, and produce important results and conclusions to serve as base for a future performance model able to incorporate both FE and BE problems.

2.5 Summary

This chapter provided a deep analysis on the processor micro architecture, Skylake, that will be used in this thesis work, such as maximum number of instructions retired per cycle and the width of the FE, to allow a critical view of the results obtained and conclusions. In the next section importance of micro-benchmarking and hardware counters is discussed, followed by a brief description of state-of-the-art models that utilize micro-benchmarks and hardware counter, such as CARM and the Top Down Method, as long with the modern performance analysis tools that have implemented them.

After presenting the models, several state-of-the-art works on micro-benchmarking are presented, alongside some modern tools for performance analysis. To end the chapter we discussed the open challenges this thesis proposes to tackle, namely the lack of micro benchmarking methodology to asses FE limitations and its impact on overall performance.

3 Micro Benchmarking of Intel Skylake Front End

Contents

3.1 Micro Benchmarking Tool **26**
3.2 Front End Micro benchmarks **27**
3.2.1 MITE Benchmarks 28
3.2.2 DSB Benchmarks 31
3.2.3 Bottleneck Prediction 32
3.3 Validation Tests **33**
3.4 Summary **34**

The fetching and decoding of instructions in current out-of-order processors involves diverse hardware resources with distinct capabilities, that affect application performance differently. For this reason, optimizing applications whose execution is highly dependent on the FE performance is a challenging task.

To tackle this issue, it is crucial to evaluate the performance upper-bounds of the FE under diverse execution scenarios, and relate them to application execution. With this aim, micro-benchmarking methods can be used to accurately assess the performance limits of the diverse FE components. This experimental evaluation also allows to assess the realistic capabilities of a real systerm, which may not correspond to the theoretical upper-bounds indicated in data sheets.

In this Chapter, the micro-benchmarking tools and methodology used to assess the performance limits of the Intel Skylake FE are introduced and deeply analyzed. The proposed methodology consists on a set of micro-benchmarks that stress the different components in the FE of the micro-architecture (MITE, DSB, etc), allowing to assess their performance upper-bounds under different execution scenrarios. This task is performed through the utilization of a set of metrics based on hardware counters. Finally, a validation methodology is also proposed in this Chapter aiming at demonstrate the usability and accuracy of the micro-benchmarking when used to predict the overall performance of the system.

3.1 Micro Benchmarking Tool

In order to derive the metrics to experimentally obtain the performance upper-bounds of the FE components, it is necessary to access a set of hardware counters available in current processors. In particular for Intel Skylake, each hardware counter is represented by a register, i.e., a Model Specific Register (MSR), that can be identified by its unique address. This unique address is always used when it is necessary to read the counter value, or when the counter is modified. To perform a read or a write on a MSR specific assembly instructions must be used, i.e., rdmsr, to read the counter, and wrmsr, to configure the counter. However, both the reading and configuration of the counters can only be performed in kernel-space. Thus, to access the MSRs, a separate kernel module needs to be incorporated in the micro-benchmarks in order to access the counters from the user space. To solve this issue we used the benchmarking tool illustrated in Figure 3.1, which provides an interface between the user-space and kernel-space through a set of system calls.

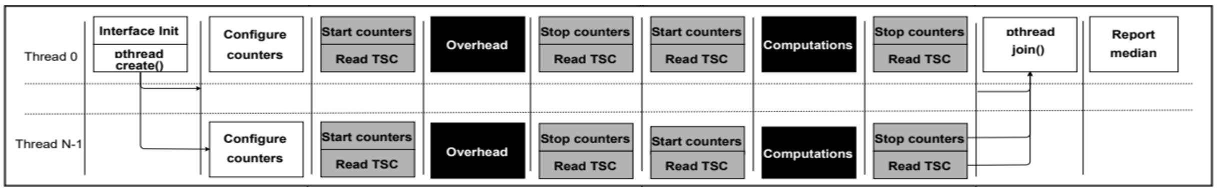


Figure 3.1: Benchmarking tool layout

As it can be observed in Figure 3.1, the tool starts by initializing the interface between user-space and kernel-space. After the interface initialization, the threads are launched by using the function pthread_create from the phtreads interface. Then, each thread configures the counters necessary to obtain the measurements required to derive the metrics used to evaluate each component (e.g. throughput, bandwidth, etc). This includes fixed counters that are not configured by the user, such as the Time Stamp Counter (TSC) which measures the number of elapsed clock cycles. At this point, the tool creates the MSR configuration in the user side and uses the system calls and assembly instructions to send the configuration to the kernel-space, along with its unique address and command (read/write).

To configure the counters it is necessary to enable them through the IA32_PERF_GLOBAL_CTRL MSR [6]. The first 8 bits of this MSR enable the general purpose counters, while the bits from 32 to 34 enable the fixed counters, therefore we need to set all these bits to 1 so that we can access both types of counters. After enabling the counters we configure the general purpose counters. This is done by using the respective IA32_PERFEVTSEL MSR [6]. The first 8 bits of this MSR (0-7) correspond to the event select of our desired counter and the next 8 bits (8-15) correspond to its unit mask. In the case where our counter needs to define a counter mask this is also done on this MSR. With all the configuration done, the counters can be read from the respective IA32_PMC MSR [6]. The machine used for our micro benchmarking supports 4 general purpose counters per core, in hyper-threading mode, or 8 without hyper-threading on.

3.2 Front End Micro benchmarks

In this section the methodology used for micro benchmarking is presented and discussed. To benchmark the FE components 4 general purpose counters and 2 fixed counters are configured in every micro-benchmark. While the general purpose counters vary based on the tested component, the fixed counters CPU_CLK_UNHALTED.THREAD (to measure the number of cycles), and INST_RETIRED.ANY (to measure the number of instructions retired) are configured for all the FE micro-benchmarks. Furthermore, to ensure that the micro-benchmarks were exercising each component as expected, the counters IDQ_UOPS_NOT _DELIVERED.CYCLES_FE_WAS_OK (measures the cycles where FE issues 4 micro operations or is stalled by the BE), IDQ.ALL_MITE_CYCLES_4_UOPS (measure the cycles where MITE issues 4 micro operations) and IDQ.ALL_DSB_CYCLES_4_UOPS (measure the cycles where DSB issues 4 micro operations) were also measured in order to confirm the origin of the performance issues. Through the fixed hardware counters, we define the throughput as the number of micro operations per cycle, which is given by Equation (3.1):

$$P = \frac{\#micro\ operations}{\#cycles} = \frac{\#instructions \times (\frac{\#micro\ operations}{\#instructions})}{\#cycles}, \tag{3.1}$$

where #micro operations is the number of micro operations, #cycles is the amount of elapsed cycles and #instructions is the number of instructions retired.

As referred in the analysis of the Intel Skylake micro-architecture (Section 2.1.2), the IDQ is able to receive micro operations from three different sources: MITE, DSB and MSROM. This implies that the total number of execution cycles can be estimated by adding together the number of cycles where each of the decoding paths (MITE, DSB, MSROM) is issuing and the number of cycles all decoding paths are stalling for instructions (assuming there are no stalls from the BE), which leads to equation (3.2):

$$Cycles = MITE\_Cycles + DSB\_Cycles + MSROM\_Cycles + FE\_Stalls, \tag{3.2}$$

where MITE_Cycles is the amount of cycles where the MITE is issuing micro operations, DSB_Cycles is the amount of cycles where the DSB is issuing micro operations, MSROM_Cycles is the amount of cycles where the MSROM is issuing micro operations and FE_Cycles is the amount of cycles where all components (MITE, DSB, MSROM) are stalling for instructions.

For the proposed micro-benchmarking methodology, all the tested instructions have one micro operation per instruction, allowing us to calculate the throughput by using only the number of cycles and the number of instructions retired. Other approaches, such as CARM, take in consideration the number of computations per operation, but since the FE works at the micro-operation granularity, the amount of operations performed by each instruction does not affect its throughput. Its throughput is only affected by instructions based on their size and complexity, not by the operations they will execute.

3.2.1 MITE Benchmarks

The micro benchmark code used to test the limitations of the MITE has the structure illustrated in Figure 3.1. The code contains two main loops: the outer loop, and the inner loop. The outer loop ensures that every benchmark runs during a pre-defined amount of time. To achieve this, the code is first executed a small number of times in order to calculate how long it takes to run a single micro-benchmark iteration. After knowing how long it takes to execute it once, the number of iterations of the outer loop is calculated in order to achieve an execution time equal (or at least very close) to the pre-defined time duration. This way the outer loop guarantees small benchmarks run enough times that any sporadic error that may occur in one iteration is attenuated. Otherwise the results of a small benchmark could be influenced by a sporadic event that occur during the small time window it was running. For big benchmarks the outer loop has the number of iterations reduced in order to save time, since in a big time window sporadic errors should not have great impact in the results. The inner loop focus on attenuating the impact of the first run of the benchmarks. The first time instructions are being executed they are not stored in either the DSB or the caches, to mitigate this situation our inner loop has a fixed value of 10 iteration, decreasing the weight of the first iteration. When parsing the results of these micro benchmarks all the iteration values are considered in order to obtain the results regarding a single code execution.

Algorithm 3.1 Micro benchmark structure

```
for Outer_Loop_iterations do
    for Inner_Loop_iterations do
        NOP_Instruction
        NOP_Instruction
        .
        .
        .
    end
end
```

Inside the inner loop only one instruction per benchmark is used, meaning to test two different instruction, for example, 2B NOP and 3B NOP, two different benchmarks are created. To micro benchmark the MITE, it is crucial to minimize the BE interference in the measurements. In order to mitigate this issue the proposed MITE micro benchmarks only contain NOP instructions. While these instructions are still decoded by the FE and issued to the BE, NOPs are practically ignored by the latter, since they are immediately retired without interacting with other BE components besides the ROB. Therefore providing information regarding FE performance. With this approach it is guaranteed that any performance limitations identified through the measurements are related uniquely to the FE, with the exception of the micro-architecture limitation of 4 micro operations per cycle.

In order to test the limitations of the MITE under different execution scenarios, different instruction sizes are considered, through the utilization of NOP instructions with sizes ranging from 2B to 10B. With this methodology, it is possible to assess the throughput and performance of the MITE for a wide range of instruction sizes, which can be used to predict application performance based on the average instruction size. Moreover, to evaluate how the accesses to different memory levels impacts the MITE performance, the code size of the benchmark varies according to the tested memory level. For example, to test L1 instruction cache, the code size changes from a few bytes to 32KB. In order to reach all three caches, L1 Icache, L2 and L3, the code size ranges from 100B to 4MB.

To analyse the MITE performance and later on predict FE bottlenecks two different metrics are proposed. These are: the overall MITE throughput ($P_{Overall\_MITE}$) and the MITE throughput (P_{Only_MITE}), calculated through equations (3.3) and (3.4) respectively:

$$P_{Overall\_MITE} = \frac{\#MITE\_Uops}{\#Cycles}, \tag{3.3}$$

$$P_{Only\_MITE} = \frac{\#MITE\_Uops}{\#MITE\_Cycles}, \tag{3.4}$$

where #MITE_Uops is the number of micro operations issued by the MITE, #Cycles is the amount of elapsed cycles and #MITE_cycles is the amount of cycles where the MITE is issuing micro operations.

Although at first glance these metrics may seem almost identical they provide useful and unique insights about the FE. Based on equation (3.2) and on the micro-architecture details discussed in chapter 2, there should be situations where P_{Only_MITE} and $P_{Overall\_MITE}$ are equal, and situations where they diverge. In order to discuss these situations let us first assume that there are no FE_Stalls. When this occurs, all execution cycles are associated with the issuing of instructions by one of the FE components. With no FE_Stalls, three different scenarios can occur: 1) all instructions are being issued by the MITE, leading the P_{Only_MITE} and $P_{Overall\_MITE}$ to be equal; 2) none of the instructions are issued by the MITE, leading to a $P_{Overall\_MITE} = 0$ and a P_{Only_MITE} with an unreliable value (caused by residual values in the measurements); 3) the micro-operations are issued by the MITE and other decoding components, which would make the $P_{Overall\_MITE}$ lower than the P_{Only_MITE} due to the number of total cycles being bigger than the number of MITE cycles.

Other possibilities emerge when we assume the existence of cycles linked to FE_Stalls. For those same three situations it is expected to obtain a worst $P_{Overall\_MITE}$, consequence of an increase of the number of total cycles, and the P_{Only_MITE} to have a similar behavior as before, since the MITE_Cycles do not account for FE_Stalls. After validating that P_{Only_MITE} is not affected by FE_Stalls, we can use it to calculate and predict MITE bottlenecks, provided we also have the number and size of instructions issued by the MITE.

In order to micro benchmark the L1 instruction cache and the rest of the memory system, the micro benchmarks use the structure as for the MITE micro benchmarks. Considering that for code sizes bigger than the L1 ICache the only component issuing instructions is the MITE, any memory bottlenecks will affect both MITE and FE performances equally. However, MITE performance and FE performance are not the same metric, since they only behave equally once the code is outside the L1 ICache. The purpose of micro benchmarking the memory system is to analyse how accesses to different memory levels can impact the FE performance. Hence, these benchmarks will begin with a small number of instructions that will steadily increase until the total code size reaches 16MB, ensuring the code goes from fitting in the L1 to fitting in the DRAM, passing by all memory levels in between. Since only the overall FE performance is being analysed, when evaluating the instruction caches, only the fixed hardware counters are needed to obtain our metrics.

Since the MITE is directly connected to the L1 instruction cache, FE problems caused by memory accesses should start to appear only when instructions no longer fit inside the L1 and start being fetched from the L2. At this point only the MITE is decoding instructions, consequently the overall throughput ($P_{Overall}$) can be calculated through equation (3.5). The $P_{Overall}$ is expected to behave inline with $P_{Overall\_MITE}$ after the L1, and has a combination of $P_{Overall\_MITE}$ and $P_{Overall\_DSB}$ while the code fits inside the L1. Besides the $P_{Overall}$ it is possible to calculate the bandwidth of each memory level, from the point of view of the FE, (B_{mem_level}). This is calculated based on the code size and the total number of cycles, resulting in Equation (3.6). With the bandwidth values it is possible to calculate the maximum throughput of the FE for different size instructions, when the only limitation factor is the memory.

$$P_{Overall} = \frac{\#InstructionsRetired}{\#Cycles} = \frac{\#MicroOperations}{\#Cycles} = \frac{\#MITE\_Uops}{\#Cycles}, \tag{3.5}$$

$$B\ (bytes/cycle) = \frac{CodeSize(bytes)}{\#Cycles} = \frac{\#Instructions \times Instruction\_Size}{\#Cycles}, \tag{3.6}$$

where #InstructionsRetired is the number of instruction retired by the micro-processor, #Micro Operations is the number of micro operations, #MITE_Uops is the number of micro operations issued by the MITE, #Cycles is the amount of elapsed cycles, CodeSize is the size of the code and Instruction_Size is the size of the instruction used in the benchmark.

In order to derive the metrics necessary to evaluate MITE and FE performance (P_{Only_MITE}, $P_{Overall\_MITE}$, $P_{Overall}$ and B), it is necessary to rely on the set of counters presented in Table 3.1, where $IDQ.MITE\_UOPS = \#MITE\_Uops$, $IDQ.MITE\_Cycles = \#MITE\_Cycles$, $INST\_RETIRED.ANY = \#Instructions = \#InstructionsRetired$ and $CPU\_CLK\_UNHALTED.THREAD = \#Cycles$.

Table 3.1: Hardware Counters MITE micro benchmarking [6]

IDQ.MITE_UOPS	Counts the number of micro operations that are delivered to the Instruction Decode Queue (IDQ) from the MITE path. When instructions are being issued to the BE through MITE there are no micro operations issued by the DSB
IDQ.MITE_CYCLES	Counts the number of cycles which micro operations are delivered to the IDQ through the MITE path
INST_RETIRED.ANY	Counts the number of all instructions retired. Since we use instructions with one micro operation per instructions, this can be used as micro instructions retired
CPU_CLK_UNHALTED. THREAD	Counts the number of core cycles while the thread in not in a halt state. If core frequency is constant it can be used to obtain an approximated elapsed time

3.2.2 DSB Benchmarks

The DSB, as it was discussed in chapter 2, works as a cache of decoded micro operations that is inclusive of the L1 instruction cache. With this in mind the micro benchmarks were slightly altered. While they follow the exact same micro-benchmark structure presented in Figure 3.1, it is only considered a maximum code size of 32KB, since the DSB only stores instructions contained in the L1 instruction cache. Similarly to the MITE tests, DSB micro-benchmarks are composed of instructions with the same size. Furthermore, in order to test different size instructions, multiple benchmark sets are tested, each set containing NOP instructions of different sizes ranging from 2B to 10B. By knowing exactly what instructions are being tested it is possible to predict how the DSB is filled and the moment where it starts to discard instructions. Due to the use of NOP instructions to avoid bottlenecks outside of the FE, it is expected the DSB to be limited by the micro-architecture retiring limit of 4 micro instructions per cycle.

In order to analyse DSB performance two different metrics are calculated, the overall DSB throughput ($P_{Overall\_DSB}$) and the DSB throughput (P_{Only_DSB}), which are calculated through equations (3.7) and (3.8) respectively.

$$P_{Overall\_DSB} = \frac{\#DSB\_Uops}{\#Cycles}, \tag{3.7}$$

$$P_{Only\_DSB} = \frac{\#DSB\_Uops}{\#DSB\_Cycles}, \tag{3.8}$$

where #DSB_Uops is the number of micro operations issued by the DSB, #Cycles is the amount of elapsed cycles and #DSB_cycles is the amount of cycles where the DSB is issuing micro operations.

Just as it was the case for the MITE metrics, there are situations where these two DSB metrics should have the same values, and situations where they are expected to differ. Since all instructions fit inside the L1 instructions cache, FE stalls are very unlikely to occur. In this scenario, there are three possible outcomes: 1) all instructions are issued by the DSB; 2) instructions are issued by the DSB and also by the MITE; 3) no instructions are being issued by the DSB.

In these benchmarks the DSB is only expected to issue 0 instructions when the code size reaches 32KB. At that point it is expected that $P_{Overall\_DSB} = 0$, just as it happened for $P_{Overall\_MITE}$ when the MITE issued no instructions. The differences between the DSB and MITE metrics occur in the two remaining situations. When the DSB is issuing all instructions, the $P_{Overall\_DSB}$ can be either equal or lower than the P_{Only_DSB}, contrary to what happened in MITE. This occurs mainly due to the micro-architecture limitation of retiring 4 micro operations per cycle. For those instructions where the DSB is able to issue more than 4 instructions per cycle, the $P_{Overall\_DSB}$ will be lower than the P_{Only_DSB}. According to the DSB behaviour described in chapter 2, this is expected to happen for the benchmarks with instructions of 2B, 3B, 4B, 5B, 6B and 7B, since these instructions fill the DSB lines with an average number of micro operation bigger than 4, and the DSB outputs on line per cycle, the throughput is superior to 4. When both the DSB and the MITE are issuing instructions, the $P_{Overall\_DSB}$ is expected to be lower than the P_{Only_DSB} since the throughput of the DSB should always be higher than the throughput of the MITE when comparing the same instruction size.

For the DSB micro benchmarks the hardware counters needed to obtain our metrics are briefly detailed in table 3.2.

Table 3.2: Hardware Counters DSB micro benchmarking [6]

IDQ.DSB_UOPS	Counts the number of micro operations that are delivered to the Instruction Decode Queue (IDQ) from the DSB. When instructions are being issued to the BE through DSB there are no micro operations issued by the MITE
IDQ.DSB_CYCLES	Counts the number of cycles which micro operations are delivered to the IDQ through the DSB path
INST_RETIRED.ANY	Counts the number of all instructions retired. Since we use instructions with one micro operation per instructions, this can be used as micro instructions retired
CPU_CLK_UNHALTED. THREAD	Counts the number of core cycles while the thread in not in a halt state. If core frequency is constant it can be used to obtain an approximated elapsed time

Compared to the hardware counters necessary to evaluate the MITE, DSB evaluation requires two different general purpose MSRs, namely: IDQ.DSB_UOPS (measure number of micro operations issued by DSB) and IDQ.DSB_CYCLES (measure number of cycle where the DSB is issuing). These are used on the DSB metrics as $\#DSB\_Uops$ and $\#DSB\_Cycles$ respectively.

3.2.3 Bottleneck Prediction

Our proposed method to calculate the FE bottlenecks of an application is based on the number of instructions, the instructions sizes, the percentage of each instruction and the metrics presented in the previous sections, namely the memory bandwidths, the MITE throughput and the DSB throughput. Since applications are composed by a mixture of instructions, for our method to combine the expected throughput of different instructions, we adapted our performance metric and arrived to Equation (3.9).

$$P_a = \frac{\theta}{T} = \frac{\sum_i \theta_i}{\sum_i \frac{\theta_i}{P_i}} = \frac{\sum_i R_i^\theta * \#\theta_i}{\sum_i \frac{R_i^\theta * \#\theta_i}{P_i}} = \frac{1}{\sum_i \frac{R_i^\theta}{P_i}}, \tag{3.9}$$

In equation (3.9) our performance P_a is defined as the total amount of instructions, θ, divided by the total time, T. Developing the equation further we can define the total amount of instructions as the sum of all types of instructions, $\sum_i \theta_i$, and the total time corresponds to the sum of all θ_i divided by the maximum attainable performance of each instruction, P_i. In order to account for the percentage of each instruction in the code, we redefine θ_i as the the ratio of each instruction, R_i^θ, times the number of micro instructions per instruction, $\#\theta_i$. Since in our tests all instructions have one micro instruction per instruction, we can simplify the equation once more, giving us the final form we see in equation (3.9). For the value of R_i^θ we will use the results obtained in our micro benchmarking, more specifically R_i^θ will be the lowest bottleneck between DSB+MITE throughput and memory bandwidth, both calculated according to the size of the instruction, code size and the memory level where the instruction is fetched.

3.3 Validation Tests

After micro benchmarking the FE and obtaining the metrics needed for the proposed method, a series of benchmarks is developed in order to validate our method under different execution scenarios. In this section the methodology used for these benchmarks is going to be presented and discussed.

The first validation test focus on evaluating the proposed method in situations where the code is composed by a mix of instructions. This validation test inherits the same structure used in the previous micro benchmarks. Besides the structure, the amount of instructions inside the inner loop also follows the same trend, starting with a low number of instructions and adding instructions up until the code reaches the DRAM (16MB). The difference between this validation test and the micro benchmarks are in the instructions used to fill the inner loop. While the validation test still only contains NOPs,in order to avoid BE issues, the inner loop contains several blocks composed by 15 NOPs of different sizes, The sizes of each NOP are randomly selected prior benchmark execution. By repeating the same block throughout the test, the percentage of each instruction in the test is equal to its percentage in the block. Using this method we test how well our predictions fare against a more typical application while easily obtaining the average instruction size and instruction percentages.

With the goal of evaluating our method against logic and computational instructions, the second validation test once again follows the structure of our previous benchmarks by placing logical and computational instructions in the inner loop, instead of NOPs. The selected instruction is repeated in the inner loop, with each benchmark adding more instructions until the code reaches the DRAM (16MB). In order to minimize BE issues each instruction uses only one register which can not be used again by the following 4 instructions, avoiding data dependencies. For this test it is also taken into account the number of ports available in the BE to each instruction when predicting performance.

Finally, our last validation test will evaluate our method against memory operations, i.e. loads and stores. Unlike our previous benchmarks, this test has a slightly different structure, which is displayed in Figure 3.2. In order to evaluate the impact on the FE when also fetching data from the memory subsystem, this benchmark contains memory operations fetching data from specific memory levels while varying the code size. In order to accomplish this task, the micro-benchmark contains one additional loop when compared to the tests previously introduced, and a vector with the necessary size. This extra loop, displayed in Figure 3.2, ensures the instructions are fetching data from the memory level being tested. By copying the address of the first vector element into a pointer (rax), different elements of the vector can be accessed. In order to access different vector elements, an offset value is added the the pointer, corresponding to the size of X number of elements. This offsets the pointer that is then used in a load or store to access the vector element corresponding to that offset. By adding an offset value to the the pointer (rax) before iterating the loop, in the next iteration the same instructions will access a different vector element. In order to reach the DRAM, the new loop has to iterate enough times so that the offset can reach vector elements on the desired memory level. The number of instruction inside this loop depends on the code size, for small code sizes the loop has to perform more iterations to reach the memory level intended. Since the number of instructions needed to reach a certain data memory level is not always multiple of the number of instructions needed to reach the desired code size, there will sometimes be instructions outside the third loop to ensure both the code size and data size intended are reached.

Algorithm 3.2 Benchmark structure - Third Loop

```
for Outer_Loop_iterations do
    for Inner_Loop_iterations do
        for Third_Loop_iterations do
            Memory_Instruction 0(rax) , (register)
            Memory_Instruction 8(rax) , (register)
            Memory_Instruction 16(rax) , (register)
            .
            .
            Memory_Instruction 2048(rax) , (register)
            Add 2048, (rax)
        end
        Memory_Instruction 0(rax) , (register)
        Memory_Instruction 8(rax) , (register)
        .
        .
    end
end
```

3.4 Summary

In this chapter the tool used for micro benchmarking the Front End was introduced and explained. A new methodology for micro benchmarking the Front End was proposed along with the Front End related metrics that provide crucial information to our method. The hardware counters necessary to obtain our metrics were highlighted and the structure of the proposed micro benchmarks was presented along with some important details regarding the micro benchmarks. Finally a methodology to validate our method was presented and discussed.

4 Experimental Results

Contents

4.1 Front End Micro-benchmarking Result 36
4.1.1 MITE Micro Benchmark Results 36
4.1.2 DSB Micro Benchmark Results 41
4.1.3 Predicting bottlenecks and performance 45
4.2 Summary 55

In the Chapter 3 the micro-benchmarks necessary to evaluate the different hardware resources contained in the FE of current OOO micro-architectures were presented. As it was stated, in order to fully uncover the limits of each component, the micro-benchmarks need to be carefully designed in order to exercise each one of the resources.

In this chapter the experimental results obtained by relying on the proposed micro-benchmarking methodology are presented and analyzed. Futhermore, the results obtained through micro-architecture benchmarking are used to predict the performance of the validation tests presented in Chapter 3, which better mimic the characteristics of real-world applications.

4.1 Front End Micro-benchmarking Result

In this section we will start by presenting the execution setup used to run the micro-benchmarks. Then the results of the micro benchmarking are presented and discussed in the following order: MITE, DSB and Instruction Cache. We will briefly analyse our metrics and compare the results with our expectations, which were detailed in chapter 3. At the end of each component section we will highlight how the results will be used to predict performance bottlenecks.

For the following results we used a machine with an Intel Core I7 6700K. The system specifics are presented in the table 4.1. This machine has its frequency fixed at the base frequency and every benchmark runs in single-threaded mode, with the thread bounded to a single core to avoid context switching overheads.

Table 4.1: Machine Specifications

CPU	Intel Core i7 6700K, based on Skylake micro architecture 6th Intel Processor generation built on 14 nm technology
Frequency	Maximum base frequency of 4.00 GHz, maximum turbo frequency of 4.2 GHz
Cores	4 cores and 8 threads available
OS	Linux CentOS 7.3
RAM	Total of 32 GB, DDR4 frequency of 2.4 GHz
ISA	Intel SSE, Intel AVX2
Memory Hierarchy	L1 ICache (32KB) - L2 (256KB) - L3 (8MB) - DRAM (32GB)

4.1.1 MITE Micro Benchmark Results

The MITE micro benchmarking focus on evaluating the MITE throughput for diverse instruction sizes, and fetched from different memory levels. In chapter 3 the methodology used for these benchmarks was presented in detail along with two MITE metrics: the overall MITE throughput ($P_{Overall\_MITE}$), and the MITE throughput (P_{Only_MITE}), calculated through equations (3.3) and (3.4) respectively. The results obtained for these two metrics can be observed in Figures 4.1 ($P_{Overall\_MITE}$) and 4.2 (P_{Only_MITE}).

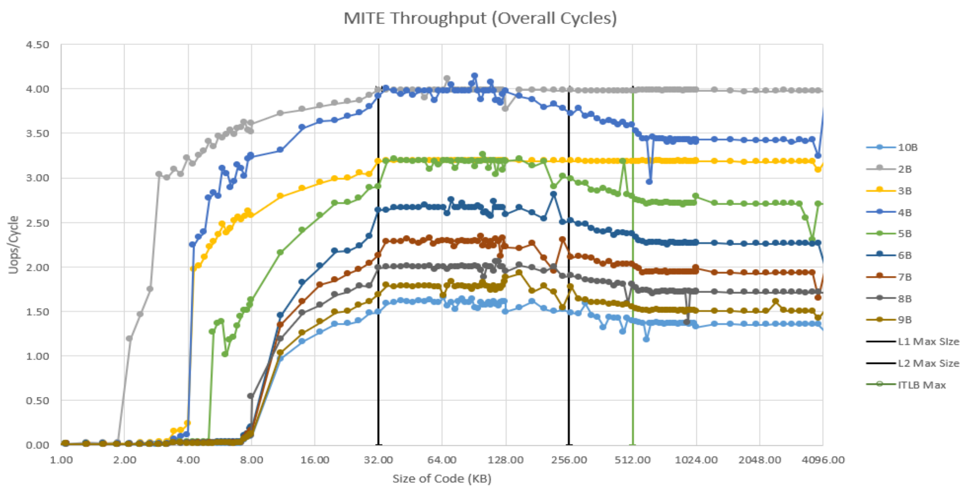


Figure 4.1: Results for $P_{Overall\_MITE}$

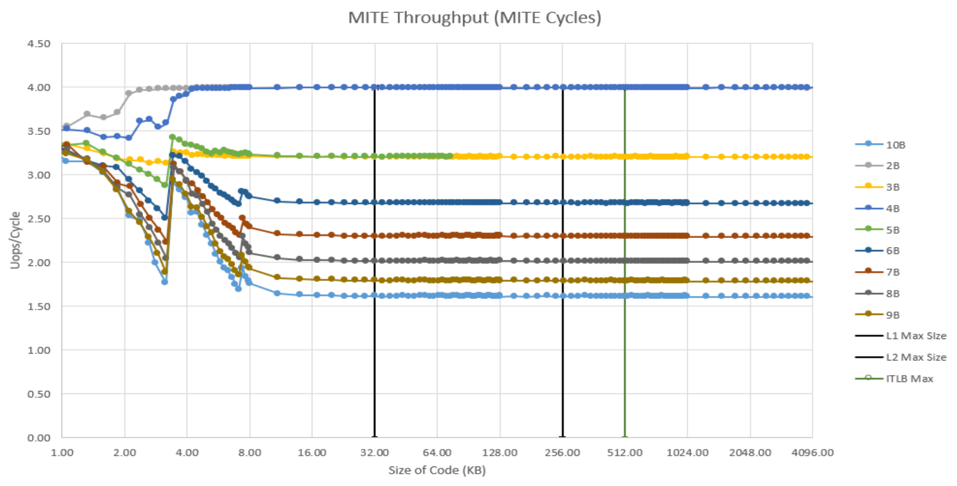


Figure 4.2: Results for P_{Only_MITE}

In Figure 4.1 the $P_{Overall\_MITE}$ depends significantly on the code size. While the code size is lower than 8KB, $P_{Overall\_MITE}$ value is always close to 0 until it increases drastically. The drastic increase happens at different code sizes for different instruction sizes, for example, while for all instructions bigger than 5B (6B, 7B, 8B and 9B and 10B) this occurs at around 8KB, for 2B instructions this occurs earlier, at around 2 KB. This behaviour is related to instruction size because it depends on the moment the DSB gets fully filled. Since the DSB stores micro operations and not instructions, the fact smaller instructions produce a larger amount of micro operations for the same size, i.e., 8KB of 2B instructions is equal to 4096 micro operations while for 8B instructions is only 1024 micro operations, justifies why the DSB gets full quicker for smaller instructions (in terms of code size).

With this information three conclusions can be made: the DSB is issuing instructions while the code is in the L1; for small codes the DSB is issuing all the instructions, justifying why $P_{Overall\_MITE}$ is 0 until the point where the DSB gets full; the initial spike in $P_{Overall\_MITE}$ implies that some instructions previously issued by the DSB start being issued by the MITE, otherwise $P_{Overall\_MITE}$ would increase more gradually, since what changes drastically at those points and consequently changes $P_{Overall\_MITE}$, is the number of micro operations issued by the MITE. The code size does not change drastically, meaning the big increase in micro operations issued by the MITE is at the cost of a decrease on micro operations issued by the DSB.

Once the code size surpasses the L1 instruction cache sized (32KB) and fetches instructions from the L2 cache the $P_{Overall\_MITE}$ stays constant until reaching the limits of the L2 This constant value depends on instruction size, due to the MITE 16B window and the MITE behaviour discussed in detail in chapter 2. For example, the results show for 10B instructions a value around 1.6, which is the result obtained by dividing the 16B window for 10B (the instruction size). For some instruction sizes this value can not be calculated only by dividing the 16B window by the instruction size. For example, for 3B instructions this constant value seen in 4.1 is around 3.2, and not 5.33, because how 3B instructions fit in the MITE window, this was explained in more detail in section 2. We might have expected to see a slight performance drop when the code size passes from the L1 to the L2, since the L2 is expected to have lower bandwidth and higher latency penalties than the L1, but has we can see Figure 4.1, there is no noticeable drop for any of the instructions tested. This behaviour indicates that any performance loss when accessing instructions in the L2 is not due to the L2 bandwidth, meaning there are no apparent penalties in terms of $P_{Overall\_MITE}$ in fetching instructions from the L2.

When reaching the end of the L2 (256KB) we see an expected performance drop, that can be attributed to the smaller bandwidth and higher latency penalties of the L3. $P_{Overall\_MITE}$ continues to decrease until all the code is inside L3, at which point $P_{Overall\_MITE}$ achieves another constant region. This behaviour is in line with what was previously discussed in chapter 3, i.e., the $P_{Overall\_MITE}$ decreases from L2 to L3 due to the FEstalls that occur due to the instruction fetching from a higher latency memory level, except for 2B and 3B instructions. This effect occurs for these specific instruction types since their reduced size allows each cache line of 64B to contain enough instructions to hide L3 latency. Since instructions are fetched from the memory subsystem in 16B window, having enough instructions on those 16B for the MITE to take more cycles decoding them can help hide the L3 latency. While the MITE is still busy decoding instructions the next block of instructions is being fetched from the L3. The decrease of $P_{Overall\_MITE}$ on L3 is related to the smaller bandwidth and higher latency penalties. These limitations will be discussed latter in this section when we present and discuss the impacts of memory accesses on the FE performance.

Knowing the MITE is not the only component issuing code, it is necessary to look at P_{Only_MITE} (Figure 4.2) so we can later combine both the MITE and DSB performance to obtain an overall performance. When analysing these results, it is possible to observe an unexpected behaviour until the code size reaches 8KB. The behaviour is caused by residual instructions issued by the MITE which are not the focus of our tests, producing the unexpected results.

This is confirmed by the results presented in Figure 4.3, which shows the amount of micro operations issued by the MITE according to the code size. As it can be observed, the moment where the MITE starts to issue instructions from our loop the values of P_{Only_MITE} achieve a constant value, for example, the 10B results show a P_{Only_MITE} of 1.6 after the code size reaches 8KB. Comparing with Figure 4.3, 8KB is the same code size where 10B instructions start seeing an increase of the number of micro operations issued by the MITE. The same happens for the other instructions, to give another example, for 2B instructions the number of MITE instructions start increasing near 2KB, and at the same code size P_{Only_MITE} reaches its constant value of 4. P_{Only_MITE} continues to stay constant even when the code enters the L3. This allow us to confirm our suspicions that P_{Only_MITE} is not affected by accesses to higher latency caches, has it was expected since P_{Only_MITE} is calculated based on the number of cycles the MITE is issuing instructions and not on the overall number of cycles, which would contain cycles where the MITE is waiting for the instruction to arrive from the caches. This allow the of P_{Only_MITE} values to predict FE bottlenecks, especially when not considering memory limitations.

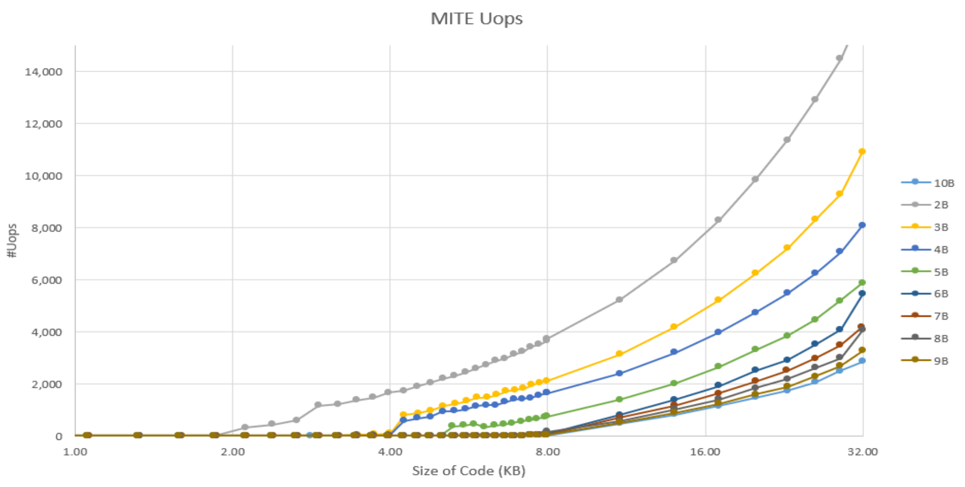


Figure 4.3: Uops Coming from MITE

To evaluate the impact of the memory subsystem on the FE performance, the maximum code size was extended to 16MB, in order to include the impacts of the DRAM. The results of these benchmarks that focus on the evaluation of the memory subsystem when fetching different instructions are presented in Figures 4.4 ($P_{Overall}$) and 4.5 (B - overall bandwidth).

Regarding the $P_{Overall}$, its initial values are expected to be similar to $P_{Overall\_DSB}$ for small code sizes, up until the MITE starts issuing instructions, where the expected decrease in $P_{Overall}$ occurs, declining until it reaches the end of L1 instruction cache. At this point the value of $P_{Overall}$ equals the values seen before for the $P_{Overall\_MITE}$. From L2 to DRAM, $P_{Overall}$ behaves like $P_{Overall\_MITE}$, suffering performance losses whenever it reaches a new memory level, as we can see, for example, in instructions of 5B, where both $P_{Overall}$ (Figure 4.4) and $P_{Overall\_MITE}$ (Figure 3.3) values are 3.2, when the code is inside the L2, and both decreasing to 2.7 after the code reaches the L3. The only exceptions to this behaviour are the 2B and 3B instructions for reasons that were already discussed in the MITE analysis, maintaining the same $P_{Overall}$ value of 4 and 3.2 respectively, through L2 and L3.

Regarding the bandwidth results on Figure 4.5, where the bandwidth, calculated through equation (3.6), is presented, it is possible to observe that L1 bandwidth depends on the instruction size. For example, for 2B attains 8 bytes per cycle while for 10B achieves values from 34 to 16 bytes per cycle. These results can be explain by the use of the DSB to issue instructions. Since DSB is able to issue big instruction at a higher rate than the MITE, the overall bandwidth achieves higher values, with the smaller bandwidths corresponding to the smaller instructions. When entering the L2 cache the DSB stops issuing instructions and we see our bandwidth stabilize at 16B per cycle, which corresponds to the window size of MITE. The only exceptions are the L2 bandwidths of 2B and 3B instructions. While all other instructions (from 4B to 10B) achieve a bandwidth of 16 bytes per cycle (4.5), the 2B and 3B have a lower bandwidth of 8 and 9.55 bytes per cycle, respectively.

When fetching these instructions, each 16B decoding windows contains more than 5 instructions, resulting in extra cycles to decode them, decreasing their performance as well as their overall bandwidth. Therefore the bottleneck for these instructions continues to be the MITE path and not the memories accesses.

When the code size reaches the L3 cache, the bandwidth for 2B and 3B instructions remains the same while all the other instructions see their bandwidths drop to around 13.6 bytes per cycle. Finally, when reaching the DRAM all instructions are affected, even the 2B and 3B instructions, with their bandwidth falling to around 3.5 bytes/cycle. The DRAM is the first memory level to affect both the 2B and 3B instructions since its the only situation where the memory bandwidth limitation is lower than the limitation of the MITE, making it the performance bottleneck for these instructions.

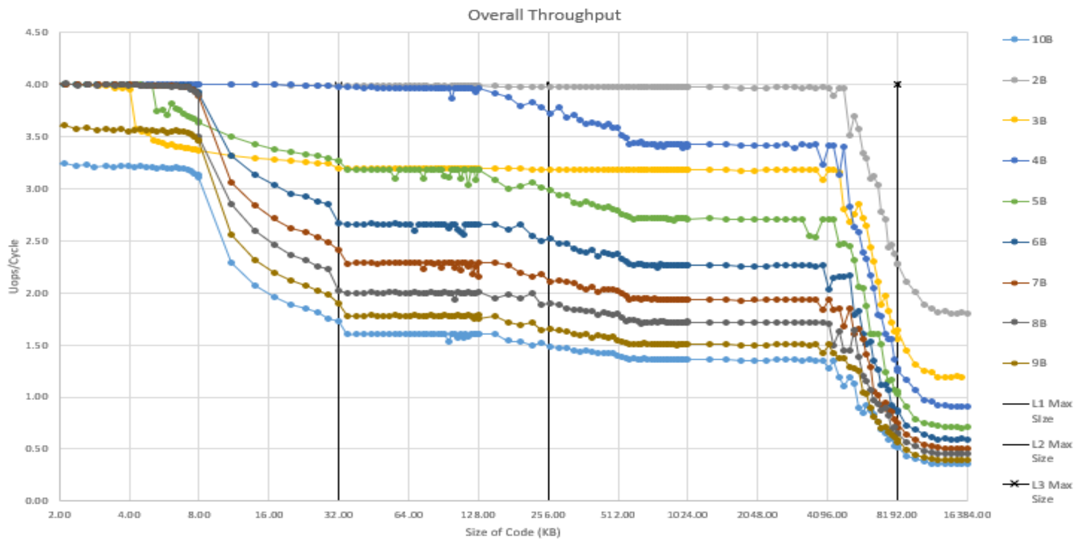


Figure 4.4: Overall Throughput - $P_{Overall}$

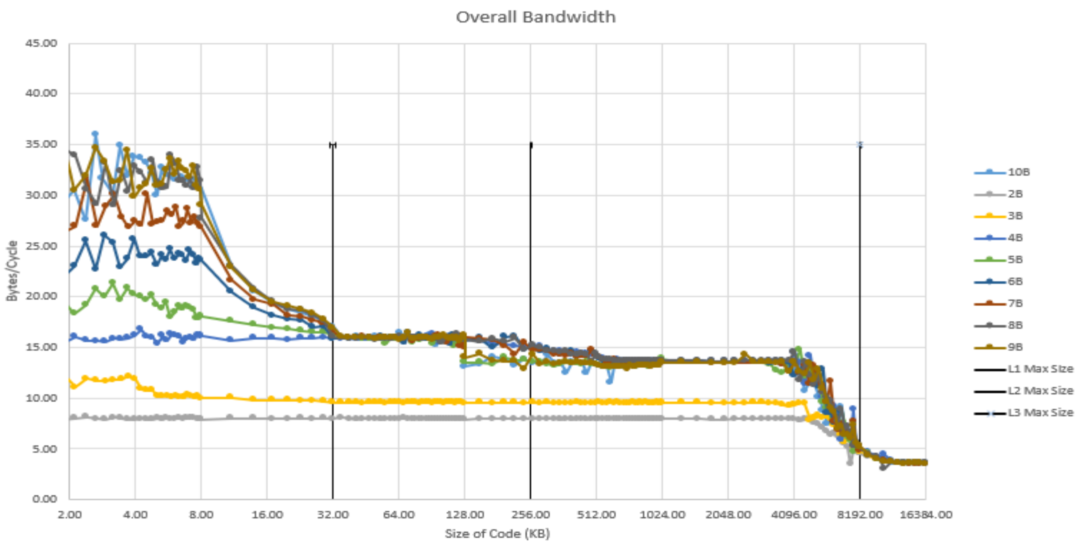


Figure 4.5: Overall Bandwidth

With these results we can attribute a bandwidth value for the different levels of memory which will be use to predict performance and application bottlenecks. Since the L2 does not seem to have an impact in the FE performance, due to the bottlenecks related to the 16B windows of MITE, the method proposed in this Thesis only considers the bandwidth values for the L3 cache (B_{L3} = 13.6 bytes/cycle) and DRAM (B_{DRAM} = 3.5 bytes/cycle). From B_{L3} and B_{DRAM} it is possible to obtain the expected throughput, for situations where an application is only bottlenecked by memory, just based on the instruction sizes, with the exception of 2B and 3B instructions inside the L3. For example, for instructions of 7B, and with no problems on the BE we would expect their throughput when coming from L3 and DRAM to be:

$$P_{7B-L3} = \frac{B_{L3}}{7B/Instr.} = \frac{13.6}{7} = 1.94\ Instr./cycle, \tag{4.1}$$

$$P_{7B-DRAM} = \frac{B_{DRAM}}{7B/Instr.} = \frac{3.5}{7} = 0.5\ Instr./cycle. \tag{4.2}$$

4.1.2 DSB Micro Benchmark Results

By following the methodology for the DSB micro-benchmarks presented in Chapter 3, it is possible to obtain the measurements necessary to calculate the overall DSB throughput ($P_{Overall\_DSB}$) and the DSB throughput (P_{Only_DSB}). Similarly to the MITE benchmarking, the DSB study focus on evaluating its performance for different instruction sizes. The obtained results are presented in Figures 4.6, 4.7 ($P_{Overall\_DSB}$), and 4.8 (P_{Only_DSB}).

As it can be observed in Figures 4.6 and 4.7 the $P_{Overall\_DSB}$ is clearly limited by the maximum retirement rate of instructions for sizes between 2B and 7B, since for these sizes the DSB is expected to output more than 4 micro operations per cycle, as it was discussed in chapter 2. For example, 4B instructions are expected to be issued by the DSB at a rate of 5.33 micro operations per cycle. This would

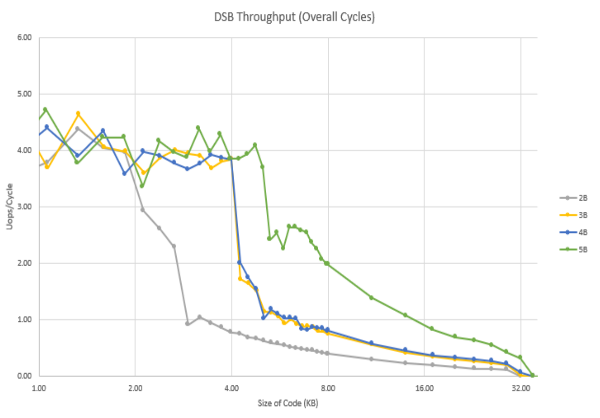


Figure 4.6: Results for $P_{Overall\_DSB}$ - 2B, 3B, 4B and 5B Instructions

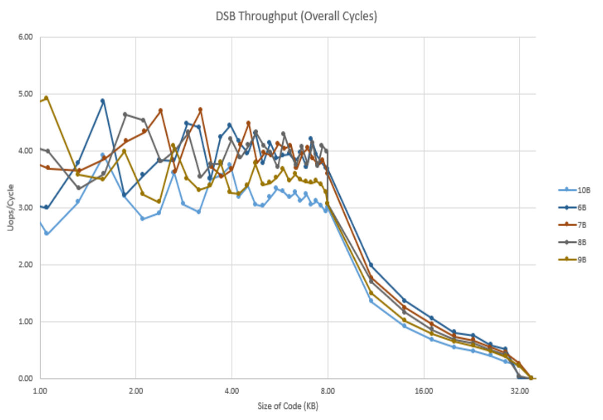


Figure 4.7: Results for $P_{Overall\_DSB}$ - 6B, 7B, 8B, 9B and 10B Instructions

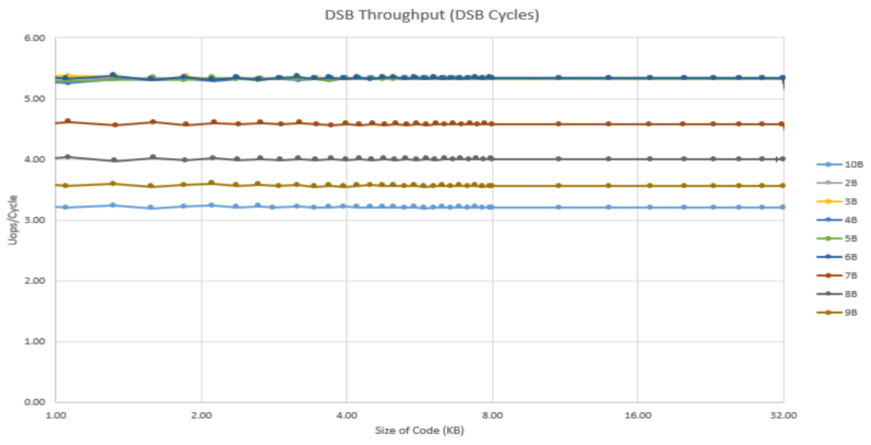


Figure 4.8: Results for P_{Only_DSB}

be the expected throughput for small code sizes where the MITE is not issuing instructions. Although for smaller code sizes the results are less stable (mainly due to the reduced code size). For the remaining instructions (8B, 9B and 10B) $P_{Overall\_DSB}$ aligns with the expectations. These instructions are limited by how they fill the DSB lines, for example, for the 10B instructions the value of $P_{Overall\_DSB}$ before the MITE starts issuing instructions (before code size reaches 8KB) is around 3.2 (Figure 4.7), which matches with the DSB behaviour explained in section 2.

As the code size increases, there is a reduction on $P_{Overall\_DSB}$, due to the issuing of instructions by the MITE. The moment when the MITE starts issuing micro operations $P_{Overall\_DSB}$ has a drastic drop, for example, for 2B instructions this drop happens at around 2KB of code size (Figure 4.6), which we previously concluded to be point where the MITE starts issuing micro operations (Figure 4.3). This drastic drop implies that some micro operations previously issued by the DSB start being issued by the MITE, otherwise $P_{Overall\_DSB}$ would decrease more gradually. In this case, the number of DSB micro operations drop significantly while the number of cycles continues to steadily increase, making those big $P_{Overall\_DSB}$ drops. The reason why theses drops occur at different code sizes for different instruction sizes was already discussed in the results of MITE, but to summarize, smaller instructions have a higher number of micro operations for the same code size and the drop occurs when to the DSB reaches its maximum storage capacity. Since the DSB stores micro operations and not instructions, the code size where the DSB gets full depends on the size of the instructions, for example, while 10B instructions fill the DSB at around 8KB of code, for 4B instructions this happens around 4KB. After the MITE starts issuing micro operations, $P_{Overall\_DSB}$ steadily decreases as can be seen, for example, for 4B instructions when the code size surpasses 4KB (Figure 4.6). The difference between $P_{Overall\_DSB}$ of different instruction sizes is related to how the different instructions fit in DSB lines, and to how many instructions fit in that code size, i.e., 8KB of code being 4096 instructions of 2B while only being 1024 instructions of 8B. Upon reaching 32KB of code size the instructions will no longer be in the L1 cache, and $P_{Overall\_DSB}$ drops to 0 as expected since all micro operations are issued by the MITE..

In Figure 4.8 we have the results of P_{Only_DSB} for the different instruction sizes. Differently from $P_{Overall\_DSB}$, the results for the P_{Only_DSB} are constant for the entire range of code size. This is expected since it is calculated based on the number of cycles the DSB is issuing micro operations and all the code fits in L1 ICache . Furthermore, this throughput also depends on the instruction size. For example, with instructions from 2B to 7B having a P_{Only_DSB} of 5.33 and bigger instructions having P_{Only_DSB} of 4 or lower due to their sizes. As discussed in Chapter 2, these values are expected since the DSB structure is based on lines that can have from 1 to 6 micro operations, depending on how the instructions fit in each line.

Moreover, in order to accurately use P_{Only_DSB} and P_{Only_MITE} to predict the FE performance, it is crucial to uncover the number of micro operations at which point the MITE starts issuing and how many micro operations the DSB issues before and after this point. With this aim, the number of micro operations issued by the DSB for different code sizes are obtained and presented in Figures 4.9 and 4.10.

The number of DSB micro operations issued after reaching its limits do not vary equally for any instruction sizes from 2B to 10B. Since the DSB is not holding instructions but decoded micro operations, and that the fill rate of a DSB line can change from 1 micro operation to 6 depending on how the micro operations fit the first time they are decoded, the amount of micro operations per each block of 64B depends on the instruction size. Thus, for bigger instructions, it is necessary to achieve a higher code size in order to attain the maximum number of micro operations that can be stored in the DSB.

Before the MITE starts to issue micro operations the DSB will be issuing all the micro operations. This can be seen by the steady increase on the number of DSB micro operations, seen in Figures 4.9 and 4.10, for the smaller code sizes. For example, in 4B instructions and 8B instructions before the code size reaches 4KB and 8KB respectively. After the MITE starts to issue, we see the number of DSB micro operations drop to a constant value, which will depend on the size of the instructions, for example, in 4B instructions the DSB continues to issue around 400 micro operations while for 3B instructions it continues to issue around 620 micro operations.

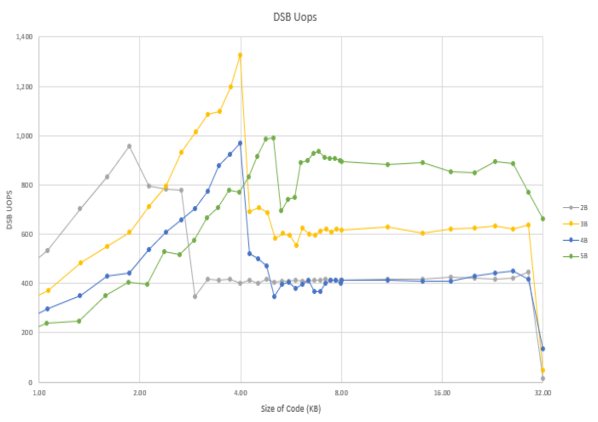


Figure 4.9: Uops issued by DSB - 2B-5B Instructions

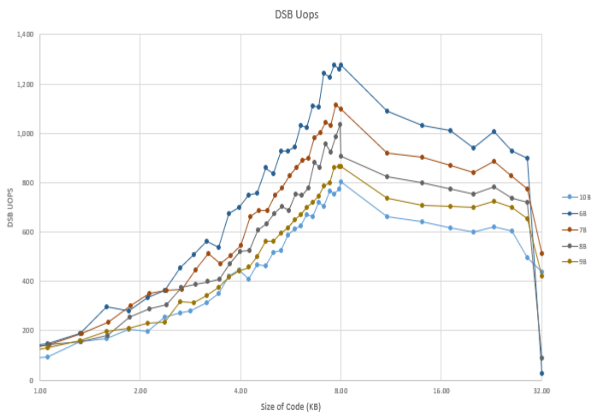


Figure 4.10: Uops issued by DSB-6B-10B Instructions

This behavior can also be observed when representing the variation of amount of micro operations served by the DSB with the number of total instructions, which can be observed in Figures 4.11 and 4.12.

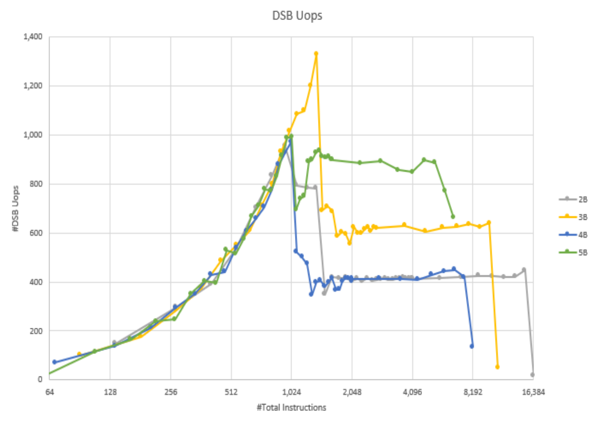


Figure 4.11: Uops issued by DSB per Total instructions - 2B-5B Instructions

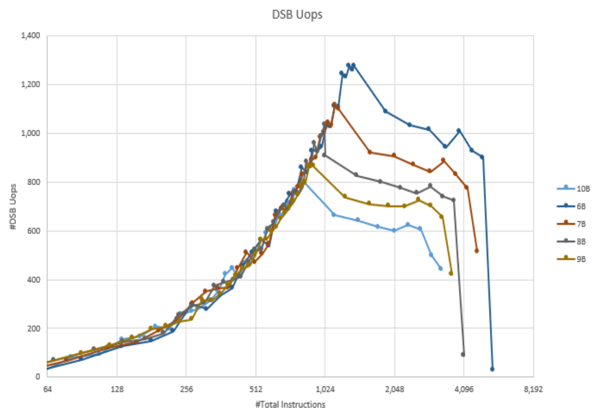


Figure 4.12: Uops issued by DSB per Total instructions - 6B-10B Instructions

In both Figures 4.11 and 4.12 it is clear that once the drop of instructions happens the DSB keeps issuing approximately the same amount of instructions until the code exceeds the L1 cache. Moreover, for some instruction sizes there is two regions of instructions. For example, the 2B instruction has a constant region of 780 micro operations for code sizes between 2KB and 3.5KB. After 3.5KB, it drops for another constant region of 400 micro operations. This effect occurs due to switches between MITE and DSB, which results in FE stalls. In order to reduce the impact of this penalty, the FE is designed to avoid switching between MITE and DSB frequently, which explains the existence of this constant regions. Hence, in certain scenarios the DSB may contain the necessary micro operations but the FE continues to issuing instructions from the MITE.

The proposed method will use the results presented in Figures 4.11 and 4.12 to calculate the point where MITE starts issuing instructions, in order to estimate the number of micro operations coming both DSB or MITE. With the number of instructions coming from these components and their respective throughput, the FE performance can be calculated.

4.1.3 Predicting bottlenecks and performance

In order to validate the approach proposed in this Thesis, three set of tests were developed to evaluate our predictions by following the validation methodology presented in Chapter 3. The first set of tests aims at validating our prediction when the code is composed by a mixture of instructions with different sizes. Following this methodology we created 4 different blocks of 15 instructions randomly chosen. These blocks have different average instruction sizes and instruction percentages. The average instruction sizes are : 4.13B for block 0, 5.67B for block 1, 6.2B for block 2 and 4.67B for block 3. A summary of these blocks that includes the percentage of each instructions is presented in in Table 4.2:

Table 4.2: Blocks Instruction Details.

Block	Average Instruction Size [B]	2B (%)	3B (%)	4B (%)	5B (%)	6B (%)	7B (%)	8B (%)	9B (%)	10B (%)
0	4.13	20	13.33	20	26.67	20	0	0	0	0
1	5.67	6.67	20	6.67	13.33	20	13.33	6.67	0	13.33
2	6.2	6.67	13.33	26.67	0	0	13.33	6.67	20	13.33
3	4.67	6.67	20	26.67	6.67	26.67	13.33	0	0	0

However, based on the experimental results obtained for the DSB and MITE, the prediction approach proposed in Chapter 3 needs to be slightly modified. The first approach to predict the bottlenecks of these tests takes into account the percentages of each instruction and their maximum attainable performance. While for most of code sizes this methods provides accurate predictions, when the codes fit in the L1 instruction cache, the prediction error increased significantly. This increased error results mainly from the DSB utilization, since the instructions order have a big impact on the DSB fill rate, and consequently on its throughput. Hence, using the percentage of instructions as described in Section 3.3, leads to inaccurate results. As an alternative, when predicting DSB performance for a mix of different instruction sizes, the prediction method for DSB prioritizes the average instruction size. The modified approach only considers the percentages of two instructions to achieve a given average instruction size. The predicted performance is calculated based on the maximum attainable throughput of these two instruction sizes: the size equal to the average size rounded up and the size equal to the average rounded down. For example, if our average instruction size is 4.7B we will use the maximum throughput of 4B and 5B, with ratios of 0.3 and 0.7 respectively. This results in a smaller margin of error when predicting performance.

With this approach, it is possible to obtain the performance predictions presented in Figures 4.13 and 4.14. Both Figures 4.13 and 4.14 represent the overall throughput of each benchmark, with our predictions plotted in solid lines. The initial part of all predictions, when the code in inside the L1 instruction cache, is based on MITE and DSB maximum attainable throughputs. After surpassing the L1 limit, our predictions are calculated through the average instruction size and the bandwidth values of each memory level. For blocks 0 and 1, the performance predicted inside the L2 is 3.87, and 2.82 respectively, and has it shows in Figure 4.13, they both are close to the results obtained. The same happens for blocks 2 and 3, that have a predicted performance of 2.58 and 3.42 respectively. The predictions continue to be very close to the results throughout L3 and DRAM for all 4 blocks. The error between prediction and experimental results can be observed in Figure 4.15. For all the 4 benchmarks the prediction error is low after the code size is bigger than L1 cache capacity. The error is almost 0% for blocks 1, 2 and 3, and under 5% for block 0. Since the predictions only change value after crossing the limits of a memory level, and in a real scenario the application performance starts decreasing before reaching the limit of the memory level, and only becomes constant again when inside the next memory level, the error values spike at two points. These points, as can be identified in Figure 4.15, are at code size 256KB and 8192, which correspond to the limits of L2 and L3 respectively. The results when the code fits in the L1 are more unpredictable. While block 0 and block 3 have good predictions and consequently low errors along all the L1, blocks 1 and 2 have worst predictions when the DSB is filled and the MITE starts issuing instructions. Without knowing how all instructions fit in the DSB it gets extremely hard to calculate a bottleneck more in line with the results. It is important to remember that our bottleneck estimations are a best case scenario, which means our goal is to estimate the bottlenecks of the FE and see how far the application is from this value. This provides useful insights to the developer of how much performance the application is leaving on the table, and that can possibly be gained back by optimizing the code.

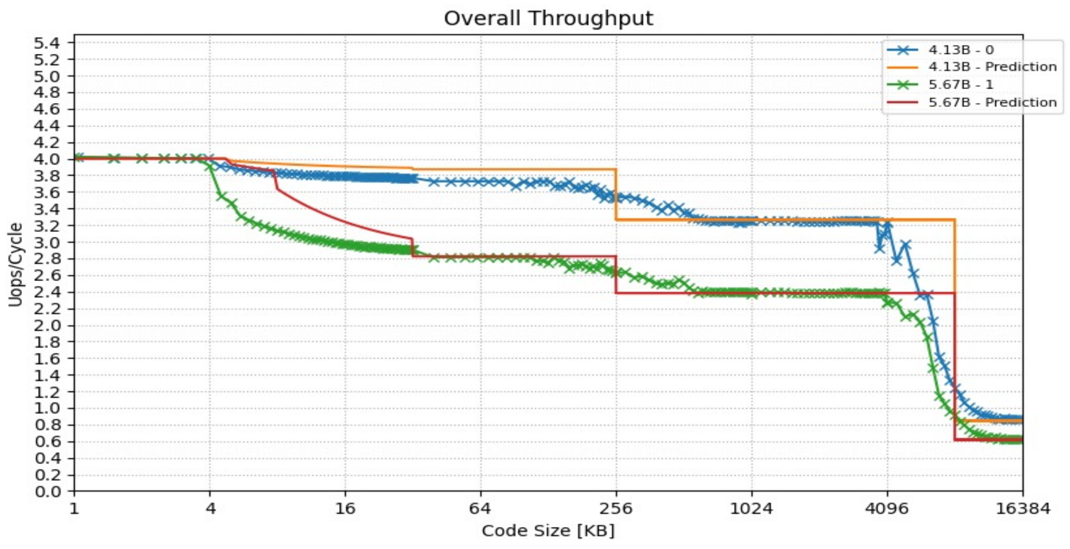


Figure 4.13: Blocks of Random Instructions - Throughput blocks 0 and 1

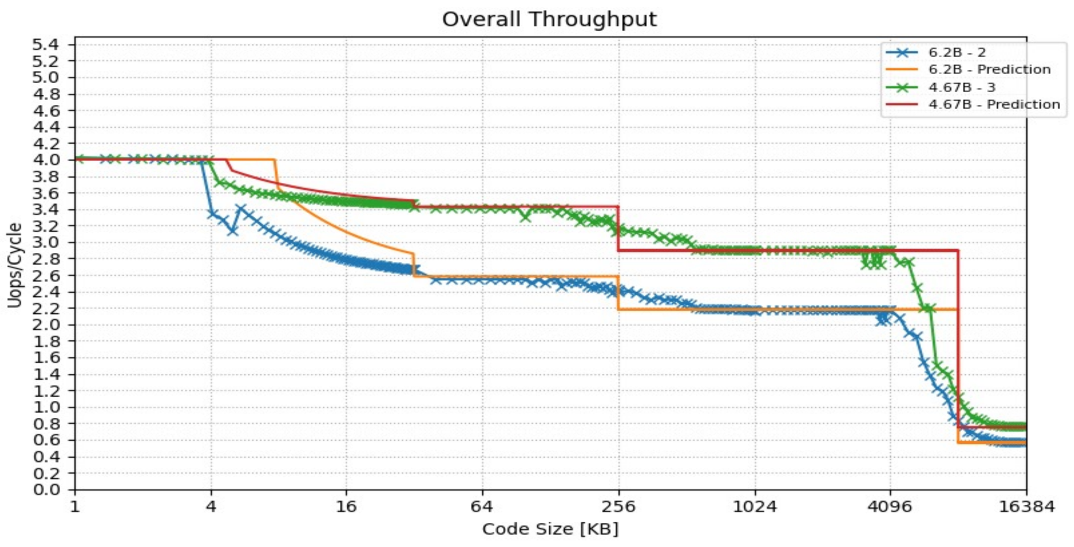


Figure 4.14: Blocks of Random Instructions - Throughput blocks 2 and 3

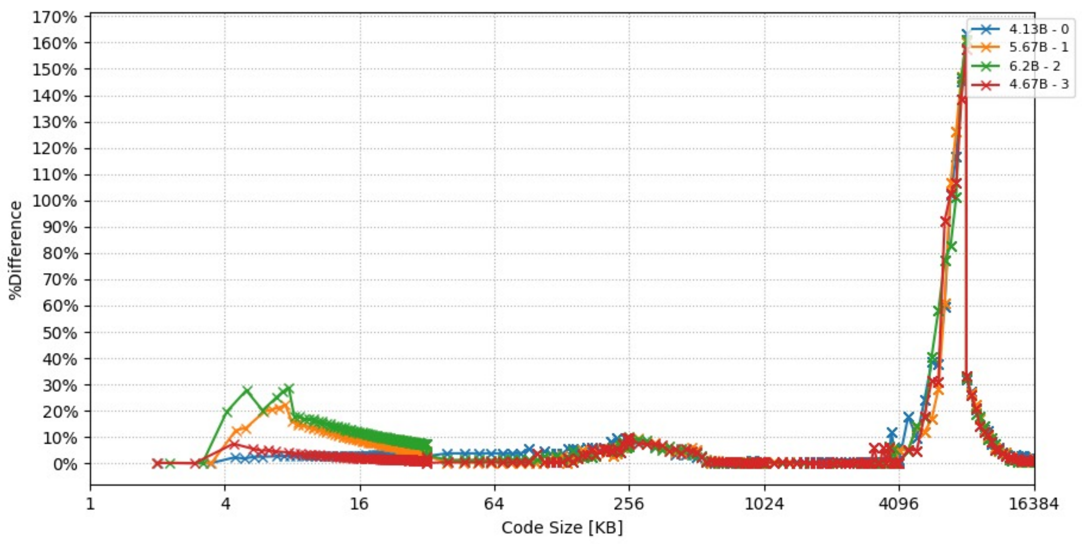


Figure 4.15: Blocks of Random Instructions - Error Margin

These first set of tests used NOP instructions to avoid having the BE limit our results in any way. However, it is still necessary to consider the maximum retirement rate of the micro-architecture, i.e., 4 micro operations per cycle. In the results presented in Figures 4.13 and 4.14 this bottleneck was taken into account, which consequentially made all the predictions start at 4 micro instructions per cycle. To truly consider only the FE limitations it would be necessary to neglect the maximum retirement rate, which would increase the error of the predictions when the code fits in L1 cache, as it can be observed in Figure 4.16. In this scenario, presenting the predictions seen in Figure 4.16 would be misleading, suggesting that there is a big performance loss that could possibly be avoided.

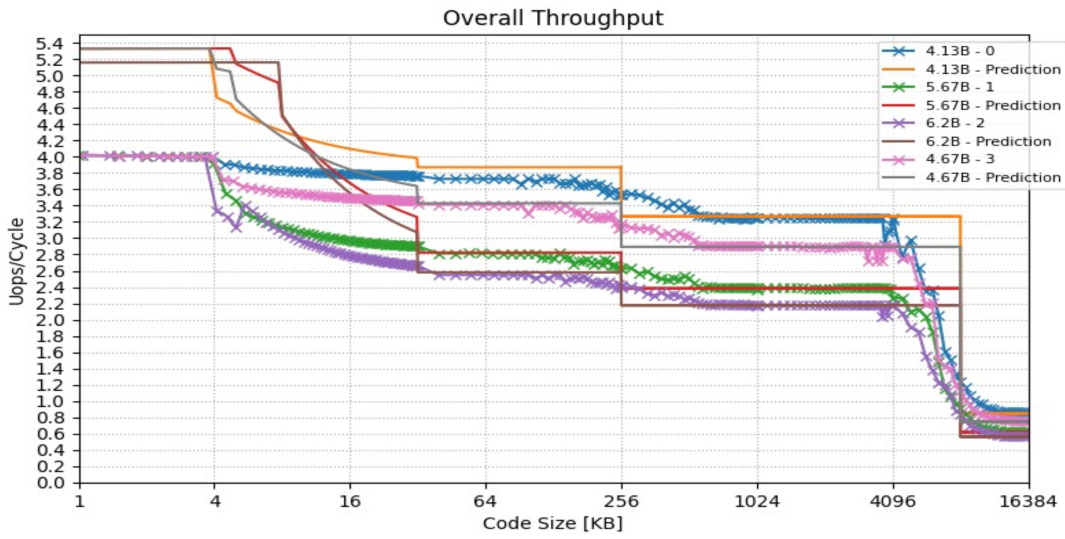


Figure 4.16: Blocks of Random Instructions - Not considering retirement bottleneck

The second set of tests validates the proposed method for logical and computational instructions. Following the methodology discussed in chapter 3, the second set of tests is comprised of computational and logic instructions with registers of three different sizes, 8B, 16B and 32B, and no accesses to the memory hierarchy to fetch Data. The instructions were selected: one logical instruction that executes a XOR in regular registers, two arithmetic instructions, one that performs additions with regular registers and one that performs vector addition with both xmm and ymm registers. The instructions chosen have some details, such as number of ports available and size, summarized in table 4.3. For these results we will continue to account for the maximum retirement rate, and for the number of ports available to each instruction. Since each test will only use one type of instructions, the performance of this instruction is limited by the number of ports available to execute that instruction. For example, if the number of ports available for a given instruction is 2, the instruction will never be able to achieve throughput superior to 2 micro instructions per cycle. Taking this limitation into account will provide a better insight on performance issues.

Table 4.3: Instructions Second Set of Tests

Instruction	Size [B]	Number of Ports	Registers	Size Data Elements [B]
ADD	3 / 4	4	r0 - r15	8
XOR	3 / 4	4	r0 - r15	8
VPADDW	4 / 5	3	xmm0 - xmm15	16
VPADDW	4 / 5	3	ymm0 - ymm15	32

The results obtained for this set of tests are present in Figures 4.17 and 4.18 (instructions with 4 execution ports) and in Figures 4.19 and 4.20 (instructions with 3 execution ports). The first group of instructions, composed of XOR and ADD , utilizes registers with 8 bytes of data. Both ADD and XOR instructions have two different sizes according to their operands. When these instructions only use registers, their size is equal to 3B; when their operands are a register and an immediate, XOR and ADD occupy 4B. As it can be observed in Figures 4.17 and 4.18, the predictions of the performance of these instructions are accurate. With the errors displayed in Figure 4.18, always near 0%, except for the two zones already explained, where the limits of the L2 and L3 are reached and a spike in the error value occurs. The results on Figure 4.17 also demonstrate that the throughput of 3B XOR and ADD are first limited by the maximum retirement rate of the micro-architecture, and after the code size surpasses 4KB, the FE throughput is the main performance limiter, attaining a throughput of 3.2 inside the L2 and L3, and a throughput of 1.2 when it reaches the DRAM. While for both 4B XOR and ADD, the FE only becomes the bottleneck upon reaching the L2. The second group of instructions is composed of instructions VPADDW that use either YMM or XMM registers. Depending on the registers used, the instruction size varies. For registers between xmm_0 and xmm_7 or ymm_0 and ymm_7 the instruction size is 4B, while for the registers xmm_8 and xmm_{15} or ymm_8 and ymm_{15} the size of the instructions increases to 5B. Once again our predictions in Figure 4.19 are close to the experimental measurements, which leads to a very small error as we can see in Figure 4.20. Once again with the error values near 0% except at 256KB and 8192KB, when the limits of L2 and L3, respectively, are reached. On these tests the throughput is limited by the number of ports available, limiting the throughput of all instructions to 3 as we can see in Figure 4.19, up until the code size reaches the L3 cache, where the throughput decreases to 2.7 for the 5B instructions, while the 4B instructions maintain their throughput. At this point the results are bottlenecked by the FE, more precisely by the accesses to the memory. When the instructions reach the DRAM their throughput massively drops.

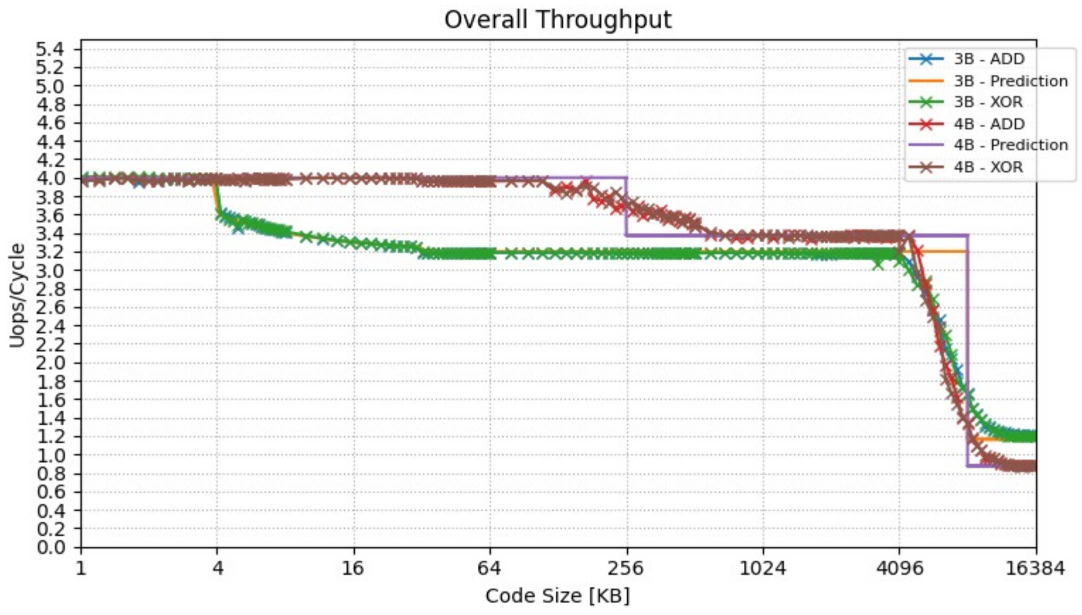


Figure 4.17: Throughput Instructions with 4 ports

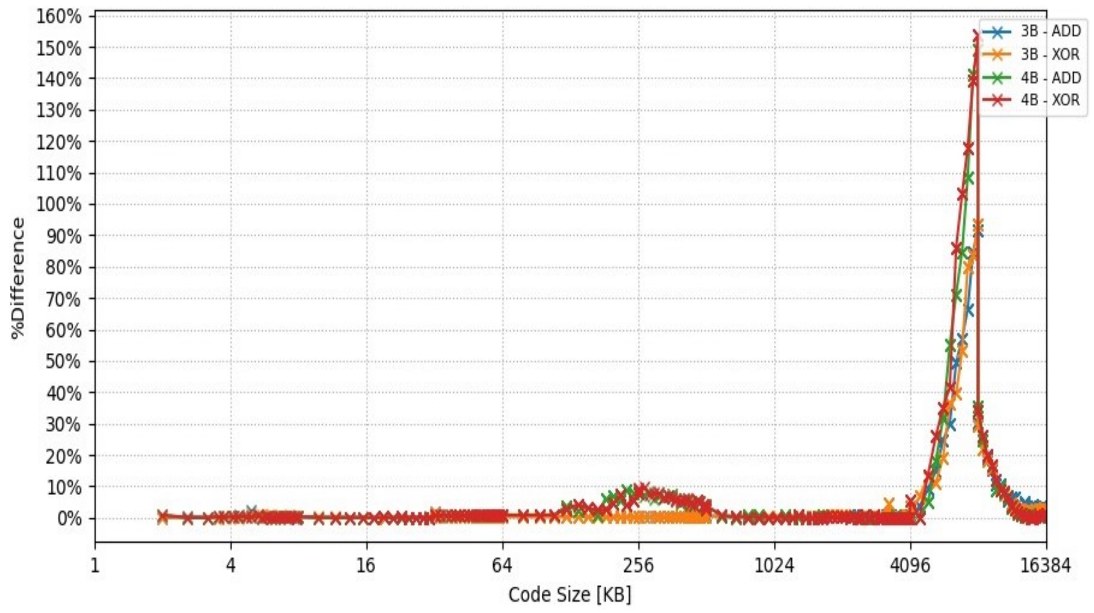


Figure 4.18: Error Margin - Instructions with 4 ports

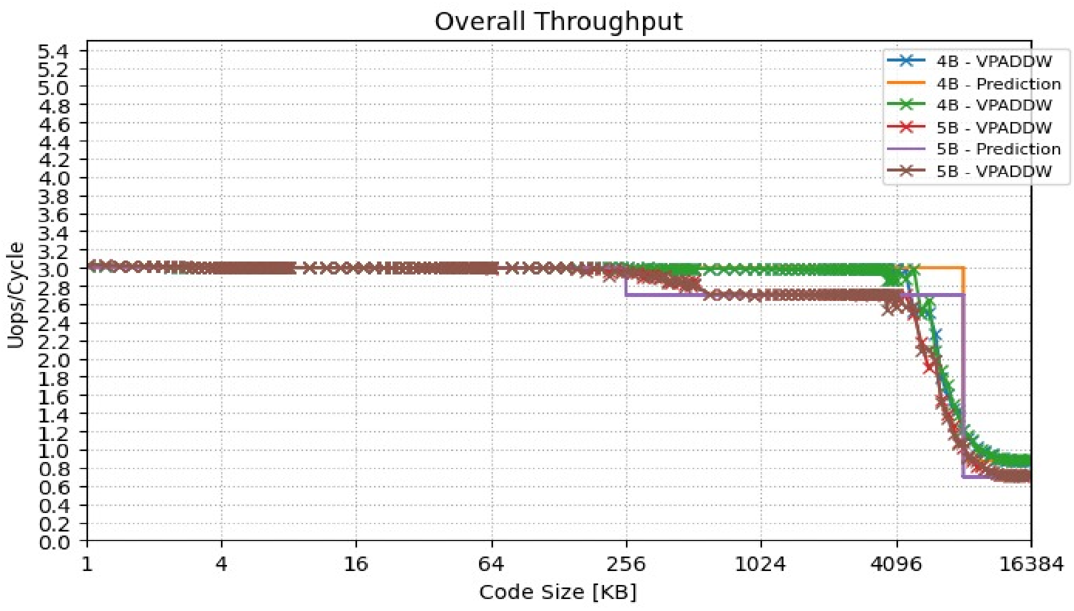


Figure 4.19: Throughput Instructions with 3 ports

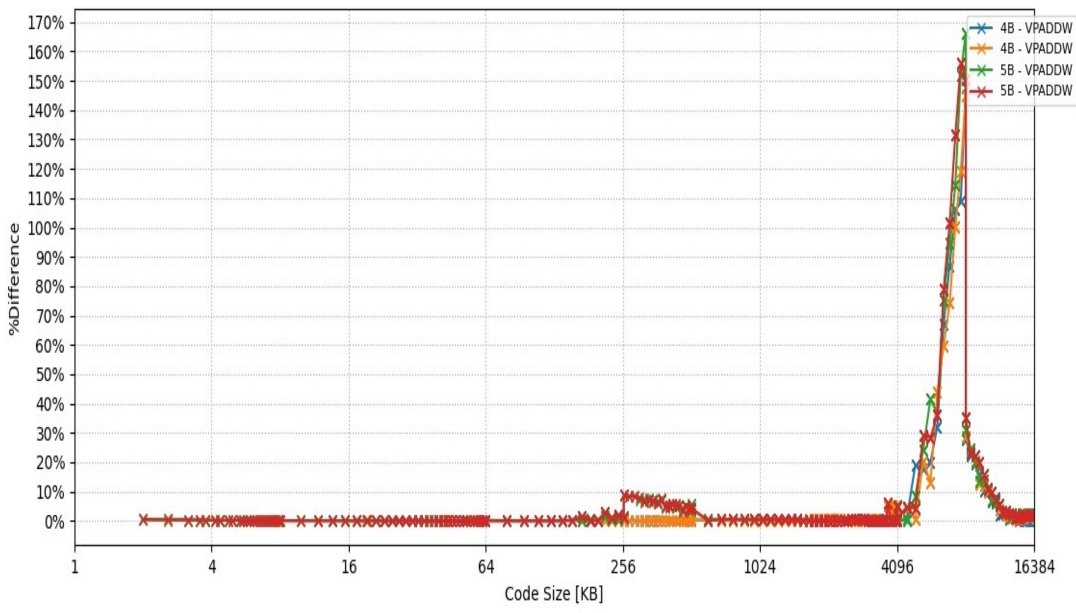


Figure 4.20: Error Margin - Instructions with 3 ports

These results illustrate the accuracy and usability of the proposed approach to predict the performance of logical and computational instructions when limited by the FE throughput. With this set of tests it was possible to assess the importance of considering both FE limitations and BE limitations. By only considering limitations from certain components the results provide misleading guidelines to software developers when optimizing applications. A good model should incorporate both FE and BE bottlenecks, either by calculating an overall bottleneck based on both, or by highlighting the multiple bottlenecks in place.

The third and last set of tests focuse on evaluating the proposed method against memory instructions, i.e loads and stores. These tests are composed by the instructions MOV, VMOVAPD and VMOVDQA, following the methodology presented in chapter 3. A summary of the instructions used is displayed in Table 4.4, where it is detailed the size of each instruction, the number of ports available and the size of the operators for each instruction. To simplify prediction calculations it is considered that the size of all MOV instructions is 7B and all VMOVAPD and VMOVDQA instructions to have a size of 8B. We can safely make this approximation since only a very small number of instructions will not be 7B or 8B.

This set of benchmarks considers 5 distinct execution scenarios, namely: load instructions with data coming from L1 Data cache, L2, L3 and DRAM; and store instructions with data coming from L1 Data cache. For all the benchmarks, the code size varied from a few hundreds of bytes up to 16 MB. The results obtained for this set of tests are presented in Figures 4.21 to 4.26.

With the data inside the L1 Data cache (Figure 4.21) the throughput of all instructions is limited to 2 micro operations per cycle due to port limitations. This remains the limitation factor until the code size reaches 256KB, where the throughput of 8B instructions decreases to 1.7 due to FE limitations. Upon reaching the DRAM, both 7B and 8B instructions are bottlenecked by the FE and have their throughput decrease to around 0.54 uops/cycle. As it can be observed in Figure 4.22, when data is served by L1 data cache the error results, are under 5% throughout all memory levels. With the same exception as before,

Table 4.4: Instructions Third set tests

Instruction	Size [B]	Number of Ports	Registers	Size Data Elements [B]
MOV	7	2 (Loads) - 1 (Stores)	r0 - r15	8
VMOVAPD	8	2 (Loads) - 1 (Stores)	xmm0 - xmm15	16
VMOVAPD	8	2 (Loads) - 1 (Stores)	ymm0 - ymm15	32
VMOVDQA	8	2 (Loads) - 1 (Stores)	xmm0 - xmm15	16
VMOVDQA	8	2 (Loads) - 1 (Stores)	ymm0 - ymm15	32

at the limits of L2 and L3.

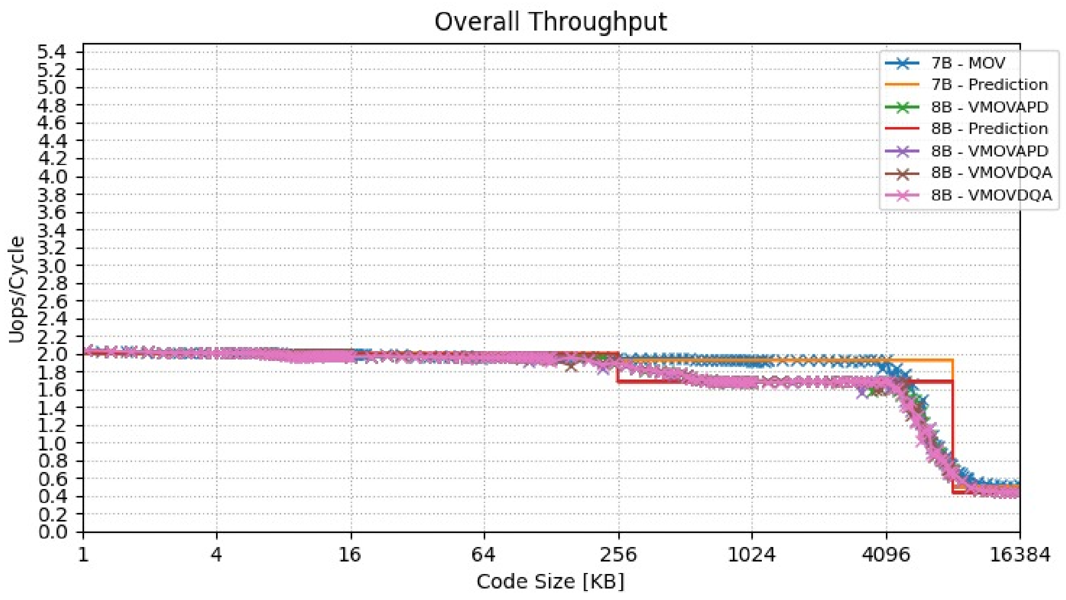


Figure 4.21: Load Data L1 - Throughput

For the load test with data served by the L2 cache. In Figure 4.23, it is possible to observe that the test is limited at 1 uops/cycle. This is undoubtedly a BE limitation, caused by the fetching of data from the L2 cache. This limitation continues to bottleneck the test up until the code size reaches the DRAM, only then the test starts being bottlenecked by the FE, more precisely by the memory accesses performed by the FE, which decreases the throughput of all instructions to 0.5 uops/cycle. Similar effects occur for the L3 and DRAM evaluations, presented in Figures 4.24 and 4.25, respectively. For smaller code sizes, the L3 Data test throughput is limited by the BE, attaining a performance of 0.5 uops/cycle when using YMM registers and 0.65 uops/cycle when using XMM and/or general purpose registers. Once the code only fits in DRAM, the FE becomes the main performance limiter, decreasing the performance of all instructions to around 0.4 uops/cycle. For DRAM Data, the performance is never limited by the FE, since the BE bandwidth is always lower than FE performance, and it attains a throughput of 0.2

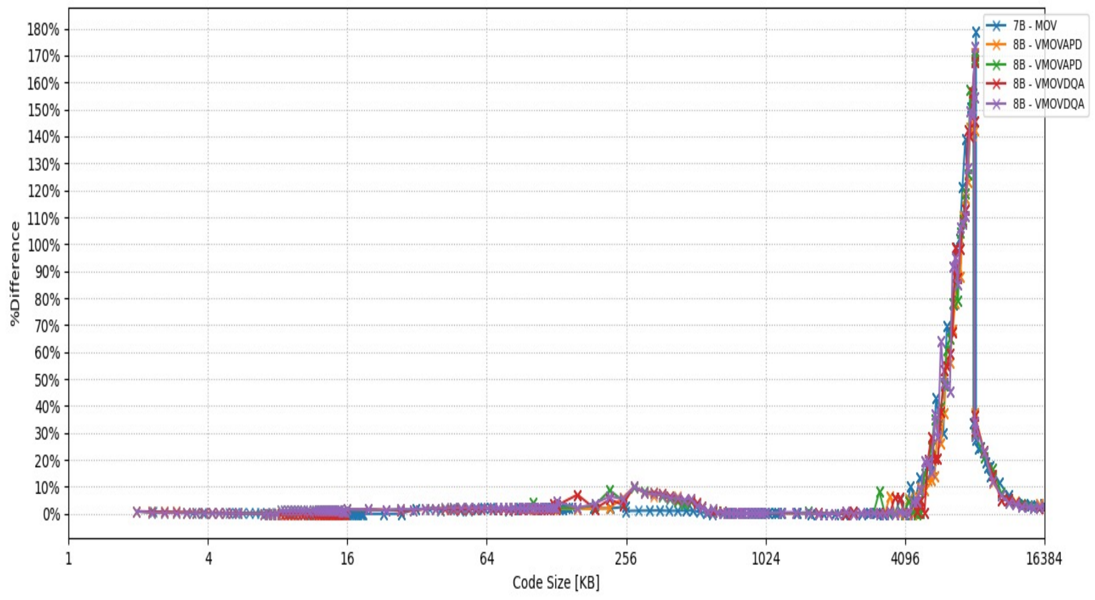


Figure 4.22: Load Data L1 - Error Margin

uops/cycle when using YMM registers and 0.11 uops/cycle when using XMM and/or general purpose registers.

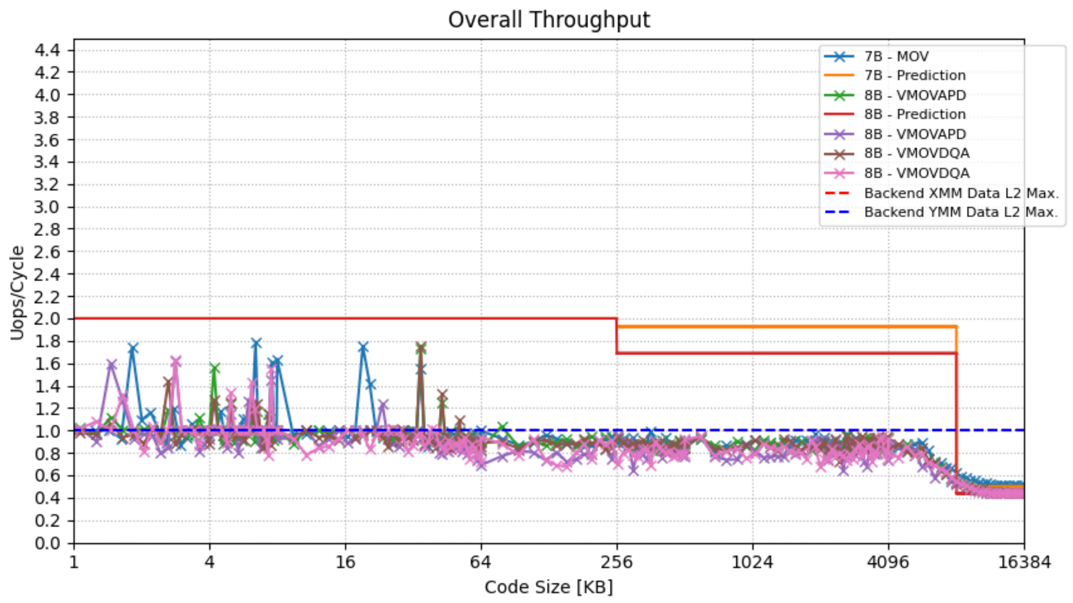


Figure 4.23: Load Data L2 - Throughput

For the store instructions, the micro architecture only supports one micro operation per cycle (1 execution port). Hence, the maximum attainable performance of this test is equal to be maximum performance supported by the BE. For the store instructions only the L1 test is presented since the remaining tests have a behavior similar to the loads tests. The results obtained for the store benchmark are presented in Figure 4.26. As it can be observed, when the code size is small, the performance of the stores is limited by the number of ports that support store instructions. Once the code size surpasses

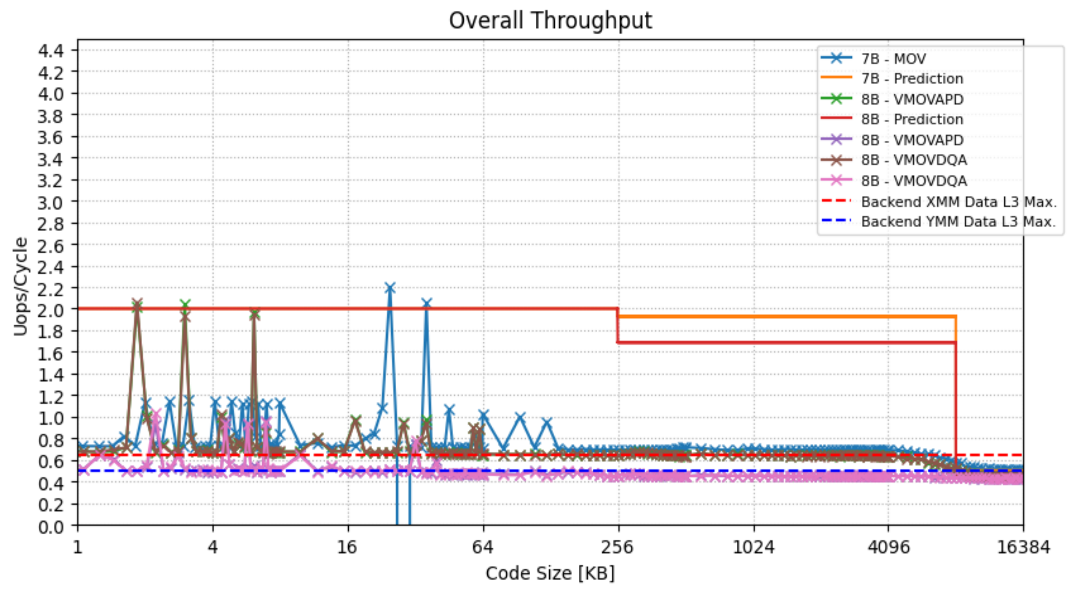


Figure 4.24: Load Data L3 - Throughput

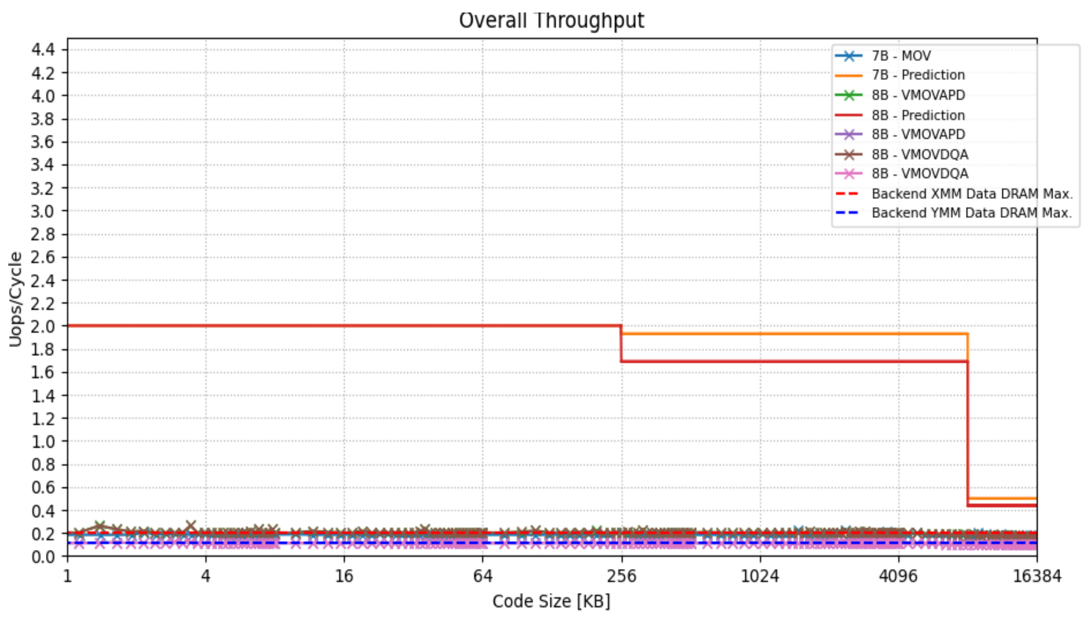


Figure 4.25: Load Data DRAM - Throughput

L1 instruction cache capacity, it occurs a small decrease of the throughput of the stores to around 0.9 uops/cycle. While this performance decrease was not expected in the predictions (1 uops/cycle), this effect ikely occurs due to the unified storage of data and instructions after L1 I cache. Differently from the loads, the stores are written-back to the memory after the update in the cache, caused by the instructions filling the L2. For this reason, the FE may provoke evictions in the caches, reducing stores performance. Moreover, as it can be observed in Figure 4.26, after the code size surpasses L3 capacity, the FE directly limits stores performance, attaining a throughput of around 0.4 uops/cycle.

This experimental evaluation demonstrates that the proposed approach is able to pinpoint execution

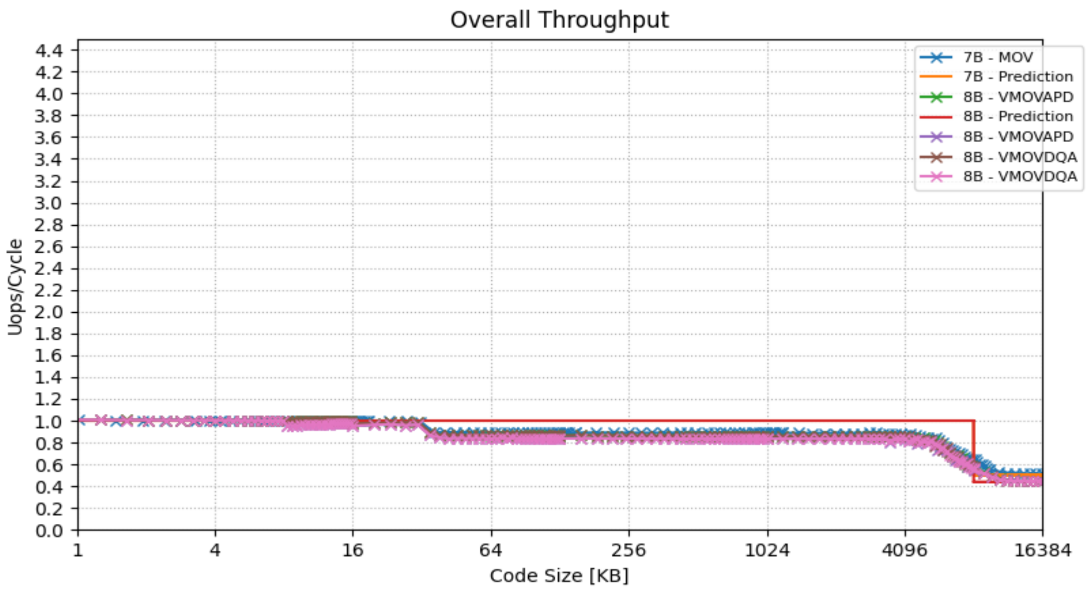


Figure 4.26: Store Data L1 - Throughput

scenarios where application performance is limited by the FE, especially for big code sizes. These tests also highlighted the importance of considering both the FE and BE bottlenecks, while focusing on a single micro-architectural sub-system can results in misleading guidelines. The proposed approach is also accurate for diverse instruction types, from NOPs to computational and memory instructions, with all of the results providing some useful information regarding FE bottlenecks and insights.

4.2 Summary

In this section the execution setup was briefly introduced. The results of the micro benchmarking of the Front End components was presented and discussed in depth. From the results, the limitations of each component were measured and highlighted, and were later used to predict the performance of applications limited by the FE.

Finally three sets of tests were used to validate the results of the proposed method. Each test focused on different scenarios. The first test aim was to evaluate the proposed test against an application with a mix of different instructions. The second test evaluated the proposed method against logical and computational instructions. The last test was used to validate the proposed method against memory instructions. The proposed method presented small errors in all validation tests, and by doing so, proved to be accurate and useful in predicting the performance of different type of applications bottleneck by the FE.

5 Conclusions

Contents

5.1 **Conclusions and Future Work** **57**
5.1.1 Future Works 57

5.1 Conclusions and Future Work

Over the last years, micro-processor companies have been continuously pushing the boundaries of technology, to keep improving the performance of micro-processors. However, the enhancements introduced in the micro-architectures also contributed to significantly increase their complexity. For this reason, the optimization of an application is far from being a trivial task, especially when considering the high system complexity coupled with the distinct capabilities and characteristics of each hardware component. Hence, software developers strive for tools that analyse application performance and provide insightful information regarding application bottlenecks.

With this aim, it is crucial to assess the performance upper-bounds of the different hardware components contained in the core pipeline of micro-architectures. One of the best approaches to assess the micro-architecture limitations is through micro-benchmarks designed to exercise differently each of the components. Through carefully designed micro-benchmarks we can obtain important performance metrics, that can be used to pinpoint the execution bottlenecks. There are several models that utilize micro-benchmarking to perform application performance analysis. Unfortunately, none of them assess FE limitations, leaving all of its components out of the bottleneck analysis.

In order to tackle this issue, this Thesis proposed a new micro-benchmarking methodology that exercises multiple FE components under different execution scenarios. This new methodology provides a new set of metrics linked to the FE components in order to calculate FE bottlenecks and predict application performance. To obtain these metrics, a minimum set of hardware counters was proposed. This set of counters provide useful insights into multiple FE metrics, and can be used to micro-benchmark several FE components.

The proposed methodology was used to perform an in-depth micro-benchmarking of the Intel Skylake FE in order to assess its performance limits. The results provided useful information regarding the limitations of FE components (MITE, DSB and memory subsystem), such as MITE throughput and memory bandwidths.. The results obtained from the micro-benchamarking were also used to predict the performance of applications limited by the FE. Finally to validate the proposed method, it was created a set of benchmarks that mimic the characteristics of real-world applications. The proposed methodology presented small errors in all of the validations tests, proving to be a capable method of prediction the performance of applications bottlenecked by the FE

5.1.1 Future Works

The proposed methodology focuses on three components of the FE that can limit application performance. These are the MITE, the DSB and the memory subsytem. As it was highlighted in chapter 2, there are two more FE components that impact application performance and which are not considered in this work.

The first component is the ITLB. The ITLB can become a bottleneck for applications with big code sizes, or applications with big jumps (superior to page sizes). When the translation of the address pages is not inside the ITLB, the whole FE will have to stall until the translation process is completed, which can take a big amount of cycles. Further research should be done regarding the ITLB and its impact in application performance. The second component is the BPU. The BPU can heavily impact application performance if it misses the prediction of a lot of branches. This is not something unusual, specially in Artificial Intelligence applications. Therefore it should be researched in order to assess its limitations and impact on application performance.

References

[1] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, September 2019.

[2] Skylake SP Micro-Architecture Specifications Wiki. https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server).

[3] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. Commun. ACM, 52(4):65–76, April 2009.

[4] A. Ilic, F. Pratas, and L. Sousa. Cache-aware roofline model: Upgrading the loft. IEEE Computer Architecture Letters, 13(1):21–24, 2014.

[5] D. Marques, H. Duarte, A. Ilic, L. Sousa, R. Belenov, P. Thierry, and Z. A. Matveev. Performance analysis with cache-aware roofline model in intel advisor. In 2017 International Conference on High Performance Computing Simulation (HPCS), pages 898–907, 2017.

[6] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, May 2019.

[7] A. Yasin. A top-down method for performance analysis and counters architecture. In 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 35–44, 2014.

[8] Skylake Micro-Architecture Specifications Wiki. https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client).

[9] S. M. Tam, H. Muljono, M. Huang, S. Iyer, K. Royneogi, N. Satti, R. Qureshi, W. Chen, T. Wang, H. Hsieh, S. Vora, and E. Wang. Skylake-sp: A 14nm 28-core xeon® processor. In 2018 IEEE International Solid - State Circuits Conference - (ISSCC), pages 34–36, 2018.

[10] Skylake SP Specifications. https://www.nas.nasa.gov/hecc/support/kb/skylake-processors_550.html.

[11] Agner Fog Forum - Skylake Performance. https://www.agner.org/optimize/blog/read.php?i=415.

[12] Agner Fog Forum - Skylake-X Performance. https://www.agner.org/optimize/blog/read.php?i=962.

[13] Tuomas Koskela, Zakhar Matveev, Charlene Yang, Adetokunbo Adedoyin, Roman Belenov, Philippe Thierry, Zhengji Zhao, Rahulkumar Gayatri, Hongzhang Shan, Leonid Oliker, Jack Deslippe, Ron Green, and Samuel Williams. A novel multi-level integrated roofline model approach for performance

characterization. In Rio Yokota, Michèle Weiland, David Keyes, and Carsten Trinitis, editors, High Performance Computing, pages 226–245, Cham, 2018. Springer International Publishing.

[14] A. Ilic, F. Pratas, and L. Sousa. Beyond the roofline: Cache-aware power and energy-efficiency modeling for multi-cores. IEEE Transactions on Computers, 66(1):52–58, 2017.

[15] Intel Advisor User Guide. https://software.intel.com/en-us/get-started-with-advisor.

[16] Intel VTune User Guide. https://software.intel.com/en-us/vtune-help.

[17] Andreas Abel and Jan Reineke. uops.info. Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Apr 2019.

[18] S. Kundu, R. Rangaswami, K. Dutta, and M. Zhao. Application performance modeling in a virtualized environment. In HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture, pages 1–10, 2010.

[19] Z. Xu, J. Lin, and S. Matsuoka. Benchmarking sw26010 many-core processor. In 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pages 743–752, 2017.

[20] James Lin, Zhigeng Xu, Linjin Cai, Akira Nukada, and Satoshi Matsuoka. Evaluating the sw26010 many-core processor with a micro-benchmark suite for performance optimizations. Parallel Computing, 77:128 – 143, 2018.

[21] R. Taylor and X. Li. A micro-benchmark suite for amd gpus. In 2010 39th International Conference on Parallel Processing Workshops, pages 387–396, 2010.

[22] J. Scheuner and P. Leitner. Estimating cloud application performance based on micro-benchmark profiling. In 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pages 90–97, 2018.

[23] Abhinav Bhatelé, Lukasz Wesolowski, Eric Bohm, Edgar Solomonik, and Laxmikant V. Kalé. Understanding application performance via micro-benchmarks on three large supercomputers: Intrepid, ranger and jaguar. The International Journal of High Performance Computing Applications, 24(4):411–427, 2010.

[24] SPEC 2017 official website. https://www.spec.org/cpu2017/.

[25] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A mechanistic performance model for superscalar out-of-order processors. ACM Trans. Comput. Syst., 27(2), May 2009.

[26] S. Van den Steen, S. Eyerman, S. De Pestel, M. Mechri, T. E. Carlson, D. Black-Schaffer, E. Hagersten, and L. Eeckhout. Analytical processor performance and power modeling using micro-architecture independent characteristics. IEEE Transactions on Computers, 65(12):3537–3551, 2016.

[27] LLVM-MCA. https://llvm.org/docs/CommandGuide/llvm-mca.html.

[28] R. Panda, S. Song, J. Dean, and L. K. John. Wait of a decade: Did spec cpu 2017 broaden the performance horizon? In 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 271–282, 2018.

[29] A. Limaye and T. Adegbija. A workload characterization of the spec cpu2017 benchmark suite. In 2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 149–158, 2018.

[30] J. N. Amaral, E. Borin, D. R. Ashley, C. Benedicto, E. Colp, J. H. S. Hoffmam, M. Karpoff, E. Ochoa, M. Redshaw, and R. E. Rodrigues. The alberta workloads for the spec cpu 2017 benchmark suite. In 2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 159–168, 2018.

[31] Ranjan Hebbar S R and Aleksandar Milenković. Spec cpu2017: Performance, event, and energy characterization on the core i7-8700k. In Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, ICPE '19, page 111–118, New York, NY, USA, 2019. Association for Computing Machinery.

[32] Sarabjeet Singh and Manu Awasthi. Memory centric characterization and analysis of spec cpu2017 suite. Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, Apr 2019.