Exploring processor Frontend capabilities via micro-benchmarking

Rafael Forte
Instituto Superior Técnico, Universidade de Lisboa
Lisbon, Portugal
Email: rafael.forte@tecnico.ulisboa.pt

EXTENDED ABSTRACT

***Abstract* - Nowadays, processor companies are constantly innovating to achieve higher performing and more efficient processors. As advances are made in micro-architectures, their complexity keeps increasing, making it harder for application developers to identify the factors that affect application efficiency and performance. In order characterize and improve applications it is necessary to have tools that provide useful insights on how an application is performing on the micro-processor. Such tools use models based on micro benchmarking and hardware counters to asses the micro-architecture limitations. Even though there are performance models utilizing micro-benchmarking, all of them are overlooking an important part of the micro-architecture, the Front End. To tackle this issue, this Thesis proposes a new methodology of micro-benchmarking to assess Front End limitations, in order to provide useful insights on their impact in application performance.**

I. Introduction

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, this contributed to increase the 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. In order to overcome the problems in achieving higher application performances 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 making it more difficult to evaluate their limits. The assessment of their limits can be performed through micro-benchmarks, that are designed to exercise the different components in the core pipeline of micro-architectures. This is an important process because it gives the realistic performance limits, different from theoretical found in the data sheets. 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 carm [1] uses micro-benchmarks in order to obtain micro-architecture metrics, such as memory bandwidths and computational throughput of instructions. Unfortunately, the current tools available to analyse application performance do not provide insightful information regarding one critical micro-architecture component, the Front End. In order to tackle this issue, this Thesis proposes a micro benchmarking methodology of the Front End of current micro-architectures, which allows to assess the impact of the different Front End components in the overall performance.

The remaining of this paper is organized as follows: Section II provides an overview on the Intel Skylake micro-architecture, the multi-core processor used in this thesis, with more emphasis on its Front End. In section III the importance of micro benchmarking is highlighted and some performance models based on micro benchmarking and hardware counter are briefly presented. Section IV presents and analyses the proposed micro benchmarking methodology. In Section V the experimental results of the micro benchmarking of the Intel Skylake Front End are presented and discussed. Section VI describes other related works. Finally section VII draws conclusions regarding the experimental results and insights gained in previous sections.

II. Intel Skylake-SP Micro-Architecture

The Intel Skylake micro-architecture was launched by Intel in August 2015. Although 5 years have passed most Intel processors found in home computers and industries servers have micro-architectures based on Intel Skylake, for example, the Intel Skylake-SP launched in 2017. Thus the core pipeline of Intel current processors is very similar apart of some minor improvements. In the next paragraphs core pipeline of the Intel Skylake-SP is divided into two main subsystems, i.e., the Front End (FE) and the Back End (BE), and briefly presented.

A. Front End (FE)

The FE of Intel Skylake-SP is presented in Figure 1 . 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.

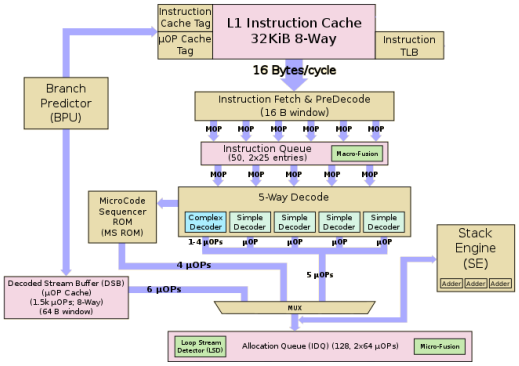

Fig. 1: Intel Skylake-SP Front End [2]

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 named Instruction Decode Queue (IDQ). Regarding the MITE path, 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. 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. With this information there is already 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. After reaching the IQ, macro operations are sent to the Instruction Decoder at a maximum rate of 5 macro operations per cycle. The instruction decoder is responsible for decoding the macro operations into micro operations that have fixed length and can be interpreted by the BE. It can decode instructions and sent them to the IDQ at a rate of 5 micro operations per cycle. 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.

Besides the MITE and the 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. 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 is directly related to the fill ratio of its lines. The DSB is specially useful to improve throughput of bigger instructions, since their throughput is usually limited by the MITE 16B window. After reaching the IDQ, micro operations are sent to the BE at a maximum rate of 6 micro instructions per cycle. Another component of the FE that can impact performance is the Branch Predict Unit (BPU). This component is responsible for deciding from which path (MITE, DSB or MSROM) the instructions are fetched. The BPU also 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, 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 Cache (L1 ICache), L2 and L3. The 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. 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.

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.

B. Back End (BE)

The BE, illustrated in Figure 2, is the out-of-order (OOO) part of the processor where the instructions are executed. To attain an efficient OOO execution, the BE relies on several components. 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).

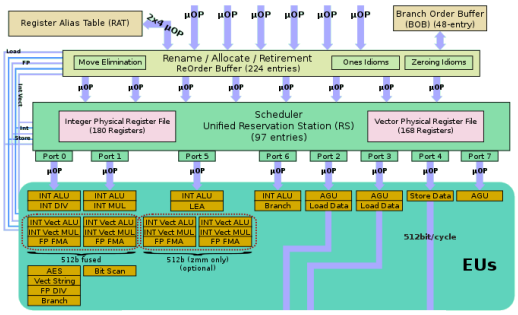

Fig. 2: Intel Skylake-SP Back End [2]

The ROB receives up to 6 micro instructions per cycle from the IDQ. At this stage, 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). The ROB also interacts with the Branch Order Buffer (BOB) which guarantees that in the case of a branch miss speculation the processor can invalidate its state and role back to a previous valid state. From the ROB micro operations are delivered to the Scheduler. The scheduler holds micro instructions until all the operands are available for the operation and the Execution Units (EUs) necessary are free. Once a micro instruction is ready for execution, the scheduler sends it to the respective EU. The memory sub-system connected to the BE 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). The memory hierarchy is shared with the FE, with the exception of a private L1 Data cache with 32KB (8-Way associative), which can perform two loads (2 x 64B) and a store per cycle (64B).

All these micro architecture characteristics should be known before developing a micro benchmarking methodology to assess micro-architecture limitations. 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.

III. 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. These micro-benchmarks rely on hardware counters built on current processors to assess the characteristics of micro-architecture components and to validate their results. 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 [3–6]. The most adopted models rely on micro-benchmarking and hardware counters. The following paragraphs will briefly present two performance models based on Micro benchmarking and/or hardware counters.

A. Cache-Aware Roofline Model (CARM)

One of the most popular solutions that rely on micro benchmarking and hardware counters are the roofline modeling approaches, in particular CARM. CARM evaluates both memory bandwidth and floating point performance from the core point of view, accounting for all data transfers. It relays on micro benchmarking to obtain the different values of computational performance and memory bandwidths. CARM defines an application metric, Arithmetic Intensity (AI), as the number of floating point operations (ϕ) divided by the number of bytes transferred (β) from the core point of view. Combining the different memory bandwidths (B_y) with this metric (AI) and the maximum floating point performance of the processor, F_p in flops, CARM calculates a maximum attainable performance, $F_{a,y}(AI)$, given by Equation 1.

$$F_{a,y}(AI) = min(B_y AI, F_p) \quad (1)$$

With this new equation for $F_{a,y}(AI)$, the CARM provides performance limits (roofs) for each memory level and for different computational instructions, as is demonstrated in Figure 3. 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 a 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 AI 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. On the other hand, if the application has a low AI and it is far away from the memory roof, the programmer should invest more time in improving accesses to memory. Furthermore, the point where the curves intercept is called ridge point, and also 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 AI 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.

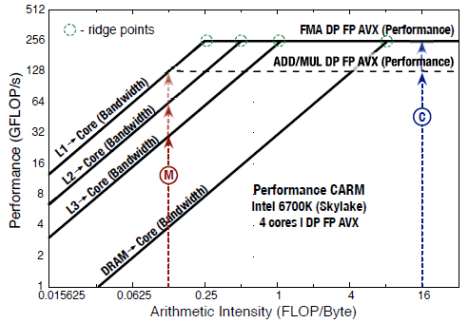

Fig. 3: Example CARM Model. [7]

The CARM model has already been incorporated in Intel Advisor, which is an Intel high-performance framework that provides insight into code vector optimization, memory access patterns, thread prototyping, flow graph analysis and Roofline analysis [8]. Unfortunately, this model does not account for any limitations on the FE, which can provide misleading information regarding applications bottlenecked by the FE.

B. Top Down Method

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 [6], 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 4.

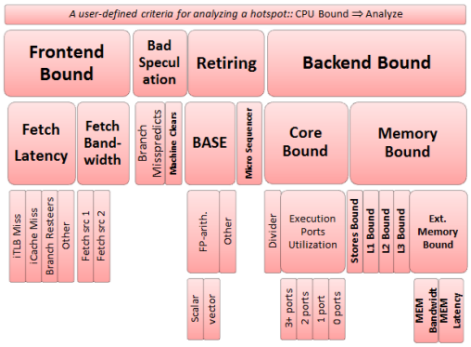

Fig. 4: Top Down Hierarchy

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.
- 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.
- Backend Bound - This node divides represents time spent performing memory accesses (or waiting for memory accesses) and time spent in the execution units of the pipeline.

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, it is fine to compare Fetch Latency with Fetch Bandwidth, but Fetch Latency can not be compared with Core Bound or with ICache Miss. The Top Down Method is already

implemented 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. Although the Top Down considers FE problems based on FE stalls emitted, 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.

In order to assess the FE problems in a way that can provide useful information regarding how its limitations are affecting application performance, a new micro benchmarking methodology is presented and discussed in the next section.

IV. Micro Benchmarking Methodology

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.

A. Micro Benchmarking Tool

To solve this issue the benchmarking tool illustrated in Figure 5 was used, which provides an interface between the user-space and kernel-space through a set of system calls.

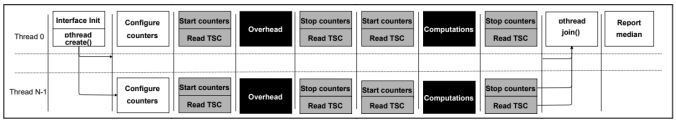

Fig. 5: Benchmarking tool layout

As it can be observed in Figure 5, 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). 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 [9]. 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 [9]. 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 [9].

B. Front End Micro Benchmarks

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 2:

$$P = \frac{\#instructions \times (\frac{\#micro\ operations}{\#instructions})}{\#cycles}, \qquad (2)$$

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

Since there are three different paths in the FE to decode and issue instructions (MITE, DSB and MSROM) the total number of execution cycles can be estimated by adding together the number of cycles where each of the decoding paths 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):

$$\begin{aligned} Cycles = MITE\_Cycles + DSB\_Cycles + \\ MSROM\_Cycles + FE\_Stalls, \end{aligned} \qquad (3)$$

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 FE components 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.

1) MITE and Memory Subsystem: The micro benchmark code used to test the limitations of the MITE has the structure illustrated in Figure 6. 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.

Algorithm 1 Micro benchmark structure

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

Fig. 6: Micro benchmark structure

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. 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 minimize the BE interference in the measurements, the MITE micro benchmarks only contain NOP instructions. 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. 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.

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 (4) and (5) respectively:

$$P_{Overall\_MITE} = \frac{\#MITE\_Uops}{\#Cycles}, \tag{4}$$

$$P_{Only\_MITE} = \frac{\#MITE\_Uops}{\#MITE\_Cycles}, \tag{5}$$

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. This metrics will be used to calculate and predict FE 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. 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. Considering 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 (6). 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 (7). 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.

$$P_{Overall} = \frac{\#Inst.Retired}{\#Cycles} = \frac{\#MITE\_Uops}{\#Cycles}, \tag{6}$$

$$B\ (bytes/cycle) = \frac{CodeSize}{\#Cycles} = \frac{\#Inst. \times Inst.\_Size}{\#Cycles}, \tag{7}$$

2) DSB: The DSB micro-benchmarks follow the exact same micro-benchmark structure presented in Figure 6, but 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. 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 (8) and (9) respectively.

$$P_{Overall\_DSB} = \frac{\#DSB\_Uops}{\#Cycles}, \tag{8}$$

$$P_{Only\_DSB} = \frac{\#DSB\_Uops}{\#DSB\_Cycles}, \tag{9}$$

where #DSB_Uops is the number of micro operations issued by the DSB and #DSB_cycles is the amount of cycles where the DSB is issuing micro operations. This metrics will be used to calculate and predict FE bottlenecks, provided we also have the number and size of instructions issued by the DSB.

C. 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 previously, 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 (10).

$$P_a = \frac{\theta}{T} = \frac{\sum_i \theta_i}{\sum_i \frac{\theta_i}{P_i}} = \frac{\sum_i R_i^\theta * \#\theta_i}{\sum_i \frac{R_i^\theta * \#\theta_i}{P_i}} = \frac{1}{\sum_i \frac{R_i^\theta}{P_i}}, \tag{10}$$

In equation (10) 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 (10). 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.

After micro benchmarking the FE and obtaining the metrics needed for the proposed method, a series of 3 benchmarks is developed in order to validate our method under different execution scenarios.

1) Validation Test 1: 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, with the only difference being the instructions used to fill the inner loop will be a mix of randomly chosen NOPs.

2) Validation Test 2: The second validation test focus on evaluating the proposed method in situations where the code is composed by logic and computational instructions. It follows the structure of the previous benchmarks by placing logical/computational instructions in the inner loop, instead of NOPs.

3) Validation Test 3: The third validation test will evaluate the proposed method against memory operations, i.e. loads and stores. Unlike the previous benchmarks, this test has a slightly different structure, with and extra loop inside the inner loop, to ensure the instructions are fetching data from the memory level being tested.

V. Results of Front End Micro Benchmarking

In this section the execution setup and the experimental results are presented and discussed in detail. To obtain the results of our micro-benchmarks a machine with an Intel Core I7 6700K was used. This machine had its frequency fixed at the base frequency and every benchmark runs in single-threaded mode.

A. MITE and Memory Results

The results obtained for the MITE metrics $P_{Overall\_MITE}$ and P_{Only_MITE} can be observed in Figures 7 and 8 respectively. In Figure 7 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. With the results seen in Figure 7 prior to 32KB, 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.

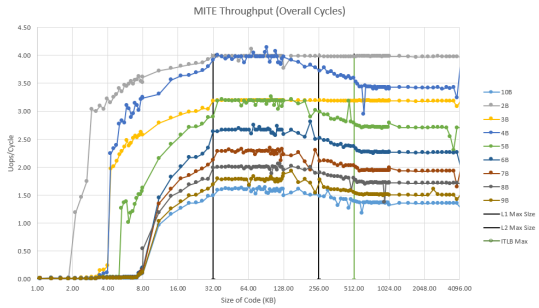

Fig. 7: Results for $P_{Overall\_MITE}$

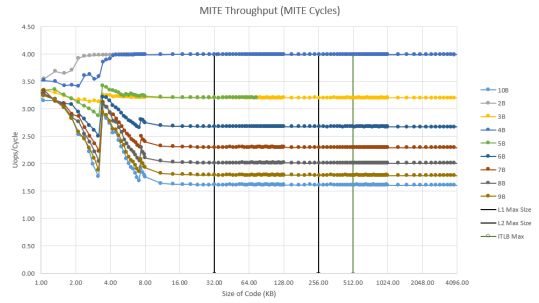

Fig. 8: Results for P_{Only_MITE}

Once the code size surpasses the L1 instruction cache size (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. 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 7 is around 3.2, and not 5.33, because how 3B instructions fit in the MITE window. We might have expected to see a slight performance drop when the code size passes from the L1 to the L2 but has it can be seen Figure 7, there is no noticeable drop for any of the instructions tested, 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, except for 2B and 3B instructions. This exceptions occur for these specific instruction types since their reduced size allows each cache line of 64B to contain enough instructions to hide L3 latency.

Knowing the MITE is not the only component issuing code, it is necessary to look at P_{Only_MITE} (Figure 8) 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. After surpassing this point, P_{Only_MITE} achieves a constant value for each instruction size and maintains this throughout all caches. The constant value is related to the 16B window and the size of each instruction, for example, 10B instructions will have a P_{Only_MITE} of $\frac{16}{10} = 1.6$. This confirms our suspicions that P_{Only_MITE} is not affected by accesses to higher latency caches, allowing the of P_{Only_MITE} to be used to predict FE bottlenecks.

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 9 ($P_{Overall}$) and 10 (B - overall bandwidth). Regarding $P_{Overall}$, its initial values are the result of a mixture of instructions coming from the DSB and from the MITE, up until reaching the end of the L1 Instruction cache where 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 9) and $P_{Overall\_MITE}$ (Figure 4) 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 due to being small instructions, and providing enough instructions in a 16B window to hide the L3 penalties.

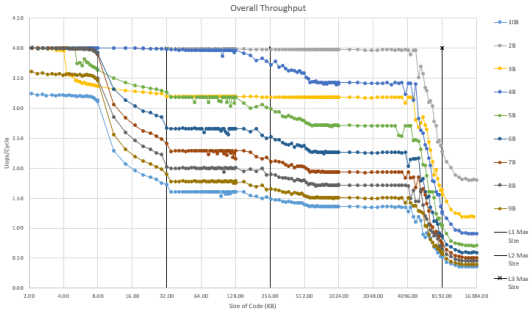

Fig. 9: Overall Throughput - $P_{Overall}$

Regarding the bandwidth results on Figure 10, where the bandwidth, calculated through equation 7, 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.

When entering the L2 cache the DSB stops issuing instruc-

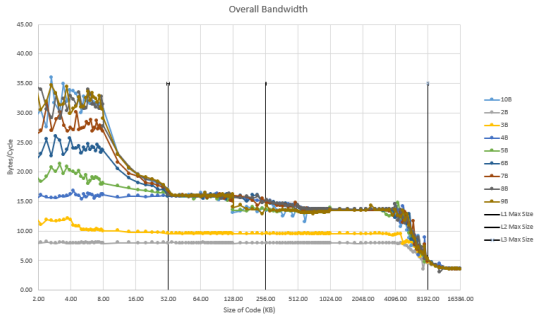

Fig. 10: Overall Bandwidth

tions 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. 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. 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 paper only considers the bandwidth values for the L3 cache ($B_{L3} = 13.6$ bytes/cycle) and DRAM ($B_{DRAM} = 3.5$ bytes/cycle).

B. DSB Results

While the results for $P_{Overall\_DSB}$ were obtained, these are not presented here due to space constraints. The results for the P_{Only_DSB}, displayed in Figure11, 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. 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 variation of amount of micro operations served by the DSB with the number of total instructions is displayed in Figure 12. Due to space constraints only the results for instructions of 2B to 5B are shown.

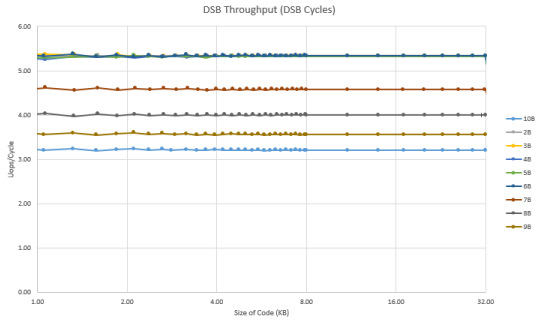

Fig. 11: Results for P_{Only_DSB}

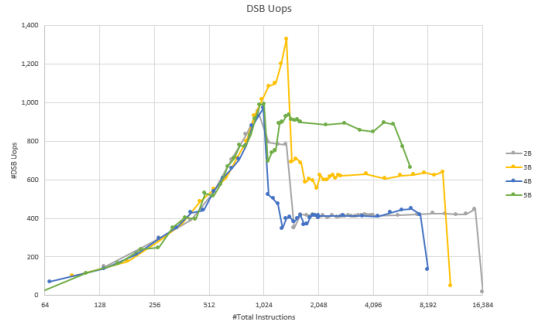

Fig. 12: #Uops issued by DSB per Total instructions - 2B-5B Instructions

In Figure 12 it is clear that once the drop of instructions happens, due to the DSB reaching its limits, 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. The proposed method will use the results presented in Figure 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.

C. Validation Tests

To validate the approach proposed in this Thesis, the three validation tests mentioned earlier where developed and tested. The results of the proposed method in all three tests were very positive, with the predictions having error values almost

always under 5%, but due to space constraints, only the results for some instructions of the the second validation test are presented. The second set of tests validates the proposed method for logical and computational instructions. The results presented in Figure 13 used instructions VPADDW, with registers of both xmm and ymm type, and sizes of 4B and 5B dependent on the registers at use. This instruction only has 3 executions units in the BE, making its maximum throughput limited to 3 uops/cycle, as we can see in Figure 13, up until the code size reaches the L3 cache (256KB), 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 drops to around 0.7. As it can be seen in Figure 13, the results follow almost perfectly the method's prediction. This happen in all the validation tests, which illustrates the accuracy and usability of the proposed approach to predict the performance of applications limited by the FE.

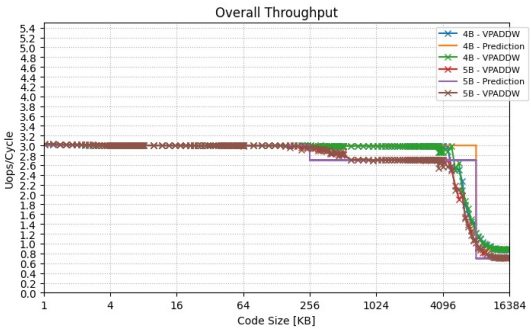

Fig. 13: Throughput Instructions VPADDW

VI. Related Work

Besides the CARM model and the Top Down Method which were already presented, there are several other state of the art works that utilize benchmarks and hardware counters in order to evaluate and/or improve their systems. From works focused on creating series of micro benchmarks to characterize the latency, throughput and port usage of instructions on Intel micro-architectures [10], to works that utilize micro-benchmarks in order to analyse application performance in different systems like Virtual Environments[11] or in the Cloud [12]. In some works, micro-benchmarking is used to uncover key micro-architectural specifications of micro-architectures, as it was the case for the Chinese supercomputer TaihuLights [13, 14]. More recently micro-benchmarking has also been used to study and assess GPUs [15] limitations.

VII. Conclusions

Over the last years the advances made in processor micro-architectures have significantly increased their complexity. This increase the difficulty of optimizing application performance. In order to provide insightful information regarding application performance, is necessary to assess the limitations of the underlying hardware. To do so, many state of the art works, including performance models, rely on micro benchmarking. Unfortunately is an essential part of the micro-architecture that has been overlooked, the FE. This Thesis tackled this issue by researching a modern micro-architecture, the Intel Skylake, and by proposing a micro-benchmarking methodology of the FE that exercises multiple FE components under different execution scenarios. This new methodology provides a new set of metrics linked to the FE components that were used to calculate FE bottlenecks and predict application performance. Finally the proposed methodology was validated with a set of benchmarks that mimic real life applications. The small errors in all of the validations tests, proved the accuracy and usability of this approach in predicting the performance of applications bottlenecked by the FE.

VIII. References

References

[1] L. S. Aleksandar Ilic, Frederico Pratas, "Cache-aware roofline model: Upgrading the loft," Apr. 2013.

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

[3] C. Y. A. A. Tuomas Koskela, Zakhar Matveev, "A novel multi-level integrated roofline model approach for performance characterization," May 2018.

[4] L. S. Aleksandar Ilic, Frederico Pratas, "Beyond the roofline: Cache-aware power and energy-efficiency modeling for multi-cores," June 2016.

[5] D. P. Samuel Williams, Andrew Waterman, "Roofline: An insightful visual performance model for floating-point programs and multicore architectures," 2009.

[6] A. Yasin, "A top-down method for performance analysis and counters architecture," June 2014.

[7] A. I. L. S. Diogo Marques, Helder Duarte, "Performance analysis with cache-aware roofline model in intel advisor," Sept. 2017.

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

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

[10] J. R. Andreas Abel, "uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures," Oct. 2018.

[11] K. D. M. Z. Sajib Kundu, Raju Rangaswami, "Application performance modeling in a virtualized environment," Jan. 2010.

[12] P. L. Joel Scheuner, "Estimating cloud application performance based on micro-benchmark profiling," July 2018.

[13] S. M. L. C. A. N. Zhigeng Xu, James Lin, "Evaluating the sw26010 many-core processor with a micro-benchmark suite for performance optimizations," Sept. 2018.

[14] S. M. Zhigeng Xu, James Lin, "Benchmarking sw26010 many-core processor," June 2017.

[15] X. L. Ryan Taylor, "A micro-benchmark suite for amd gpus," Sept. 2010.