



HAL
open science

Performance bottlenecks detection through microarchitectural sensitivity

Hugo Pompougnac, Alban Dutilleul, Christophe Guillon, Nicolas Derumigny,
Fabrice Rastello

► **To cite this version:**

Hugo Pompougnac, Alban Dutilleul, Christophe Guillon, Nicolas Derumigny, Fabrice Rastello. Performance bottlenecks detection through microarchitectural sensitivity. Institut National de Recherche en Informatique et en Automatique (INRIA). 2024, pp.1-15. hal-04796942

HAL Id: hal-04796942

<https://inria.hal.science/hal-04796942v1>

Submitted on 21 Nov 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

Performance bottlenecks detection through microarchitectural sensitivity

Hugo Pompougnac
Univ. Grenoble Alpes, Inria, CNRS,
Grenoble INP, LIG, 38000 Grenoble,
France
hugo.pompougnac@inria.fr

Alban Dutilleul
ENS Rennes
France
alban.dutilleul@ens-rennes.fr

Christophe Guillon
Univ. Grenoble Alpes, Inria, CNRS,
Grenoble INP, LIG, 38000 Grenoble,
France
christophe.guillon@inria.fr

Nicolas Derumigny
Télécom SudParis, France
derumigny.nicolas@telecom-
sudparis.eu

Fabrice Rastello
Univ. Grenoble Alpes, Inria, CNRS,
Grenoble INP, LIG, 38000 Grenoble,
France
fabrice.rastello@inria.fr

Abstract

Modern Out-of-Order (*OoO*) CPUs are complex systems with many components interleaved in non-trivial ways. Pinpointing performance bottlenecks and understanding the underlying causes of program performance issues are critical tasks to make the most of hardware resources.

We provide an in-depth overview of performance bottlenecks in recent *OoO* microarchitectures and describe the difficulties of detecting them. Techniques that *measure* resources utilization can offer a good understanding of a program's execution, but, due to the constraints inherent to Performance Monitoring Units (PMU) of CPUs, do not provide the relevant metrics for each use case.

Another approach is to rely on a performance model to *simulate* the CPU behavior. Such a model makes it possible to implement any new microarchitecture-related metric. Within this framework, we advocate for implementing modeled resources as parameters that can be varied at will to reveal performance bottlenecks. This allows a generalization of bottleneck analysis that we call *sensitivity analysis*.

We present Gus, a novel performance analysis tool that combines the advantages of sensitivity analysis and dynamic binary instrumentation within a resource-centric CPU model. We evaluate the impact of sensitivity on bottleneck analysis over a set of high-performance computing kernels.

1 Introduction

The need for compute keeps growing with massive deployment of intensive computations, whereas machines computing power remains limited by the laws of physics [22]. Thus, the optimization of compute kernels is an increasingly critical activity. Indeed, when it comes to real-time embedded systems or high-performance computing, a program's non-functional requirements (execution time, memory footprint, *etc.*) are at least as important as its functional requirements. Consequently, unexpected sources of inefficiency can be

```
1 mov    -0x10(%rsp),%rdx
2 vmovsd (%rdx,%rax,1),%xmm0
3 vaddsd 0x8(%rdx,%rax,1),%xmm0,%xmm0
4 vaddsd 0x10(%rdx,%rax,1),%xmm0,%xmm0
5 vmulsd %xmm1,%xmm0,%xmm0
6 mov    -0x18(%rsp),%rdx
7 vmovsd %xmm0,0x8(%rdx,%rax,1)
8 mov    -0x18(%rsp),%rdx
9 vmovsd -0x8(%rdx,%rax,1),%xmm0
10 vaddsd (%rdx,%rax,1),%xmm0,%xmm0
11 vaddsd 0x8(%rdx,%rax,1),%xmm0,%xmm0
12 vmulsd %xmm1,%xmm0,%xmm0
13 mov    -0x10(%rsp),%rdx
14 vmovsd %xmm0,(%rdx,%rax,1)
```

Figure 1. A computational kernel implementing the Jacobi iteration (vectorized).

seen as performance *bugs*, which need to be corrected if the program implementation is to meet its specification.

One way of guiding the optimization process is to inspect the Performance Monitoring Units (PMU) of the CPU. These hardware counters provide many measures for evaluating program performance, including the cost of execution in CPU cycles or CPU slots. Those can then be refined. For instance, the expert who optimizes a basic block (a sequence of binary or assembly instructions whose only entry point is its first instruction, and whose only exit point is its last instruction¹) reports the *throughput* of this basic block, *i.e.* its cost per iteration. Hardware counters also provide raw knowledge on the behavior of the microarchitecture's resources. Resources that have a limiting impact on a program performance are known as *bottlenecks* of this program and should be the focus of optimization efforts.

¹Strictly speaking, a basic block includes a final branch instruction, but during analysis, in the case of a loop body, it is common practice to ignore the latter[8].

In a modern Out-of-Order (OoO) CPU, the components that are often subject of bottleneckness analyses are:

- The **front-end**, which embeds the instruction decoding process and produces sequences of micro-operations (μ ops) from the input program.
- The **back-end**, or execution engine, which embeds the core’s computing units as well as the execution ports through which μ ops access the formers; the Allocator which decides which μ op should be sent to which port; and the Scheduler which retains μ ops (coming from the front-end) until their dependencies have been resolved.
- The **speculative execution**, a mechanism based on the prediction of branch conditions that executes instructions belonging to uncertain code paths. In case of mispredictions, performance are degraded.
- The **memory subsystem**, corresponding to the cache hierarchy and the off-chip memory.

Consider the assembly basic block in Fig. 1, which implements the innermost loop body of a Jacobi iteration.

The instructions on lines 7 and 14 are the only ones that do not load the Address Generation Units (AGU), while the others systematically manipulate the pointers stored in `%rsp` and `%rdx`. On Intel Skylake microarchitecture (produced until 2022 under the *Comet Lake* family), there are only two general purpose AGUs: behind ports 2 and 3. Consequently, repeated execution of this basic block is likely to saturate them. Assuming that the calculation is L1-resident, and given that the surrounding for loops (not shown here) have predictable control flow, these ports are the most likely the bottleneck, which translates to a back-end cause.

The state-of-the-art Topdown Analysis Method (TAM)[55] exploits the Performance Monitoring Units (PMUs) through hardware counters to identify bottlenecks, and is the approach used in Intel’s own performance debugging tool, VTune [29]. Detailed values of individual counters is also available for analysis through Linux `perf`[1]. In TAM, pipeline slots describe the number of μ ops a microarchitecture is capable of processing during a time interval. A slot is *filled* each time a μ op has been successfully processed (*retired*). A bottleneck is then determined as resource that causes a *sufficient number* of slots to be unused; thresholds being set heuristically.

This technique is illustrated in Fig. 2a which reproduces a subset of the `perf stat` report over the basic block from Fig. 1 in realistic conditions (being iterated thousands of times). Here, every slot is classified as unused because of the speculative execution (line 11), the front-end (line 12), the back-end (line 13), or having been filled (line 10): as expected, the back-end is detected as the bottleneck.

In most cases, TAM provides sufficient knowledge to guide optimization. However, fine-grain PMU-based analysis is known to be limited [53], as hardware counters traditionally relies on interrupt or event-based sampling. In practice, this

10	191281317	td-retiring	39,6% Retiring
11	11363246	td-bad-spec	2,4% Bad Speculation
12	17044869	td-fe-bound	3,5% Frontend Bound
13	263248545	td-be-bound	54,5% Backend Bound

(a)

Instructions	μ ops	p0	p1	p2	p3	p4
1 mov	1			1		
2 vmovsd	1				1	
3 vaddsd	2	1		1		
4 vaddsd	2	1		1		
5 vmulsd	1	1				
6 mov	1			1		
7 vmovsd	2				1	1
8 mov	1			1		
9 vmovsd	1				1	
10 vaddsd	2		1	1		
11 vaddsd	2	1			1	
12 vmulsd	1		1			
13 mov	1			1		
14 vmovsd	2				1	1

(b)

Figure 2. Extract of `perf stat` (a) and `iaca` (b) outputs when run on the basic block from Fig. 1.

means that the association from saturated resources to saturating assembly instructions is often lost, leading to tedious performance debugging.

This led to the development of a complementary approach: binary code analysis, based on the computation of metrics through a performance model at assembly level. Although these approaches produce estimations necessarily less precise than measurements, performance models offer a wider range of metrics and greater flexibility in analysis than hardware counters, and do not require any hardware support.

Intel `iaca` was the code analyzer distributed by Intel until 2019. When applied to the basic block in Fig. 1, it delivers a report including a throughput estimation, a detailed ports pressure analysis and a bottleneck analysis which outputs the most probable bottleneck cause. Fig. 2b shows a simplified transcription of the instruction-centered ports pressure table for Fig. 1. It confirms our initial intuition that most instructions use ports 2 and 3 of a Skylake architecture.

In `iaca`, bottleneck analysis essentially consists in looking for the source of probable *stalls*. Stalling occurs when a pipeline slot is empty: the corresponding hardware resources are not fed, and therefore not used during one cycle. In this way, `iaca` seeks either to identify a saturated component (front-end or back-end), or to detect dependencies between μ ops which would result in stalls in the Scheduler (latency bound). Other code analyzers (`uica`[8], `llvm-mca`[5]...) rely on different heuristics, but all embed such performance models.

We propose to generalize this approach with a method called *sensitivity*, that consists in running several simulations of the same program with different *accelerations* of CPU resources. In this framework, accelerating a resource means varying its ability to process μ ops (e.g. increasing the throughput of one port). This allows us to design a new metric: the global speed-up caused by the acceleration of a resource. When the speed-up is strictly positive, the resource is a bottleneck and should focus the optimization efforts.

Simulated resources can be considered at any level of granularity. Let us go back to the example in Fig. 1. Similarly to TAM and *iaca*, a sensitivity-based approach concludes that the back-end is bottleneck. Without any further information, a more precise report requires an expert to analyze the assembly code. But, by simply accelerating ports 2 and/or 3, sensitivity analysis can immediately conclude that the bottleneck is at the address calculation level. In this paper, we present a tool called Gus implementing such a sensitivity-oriented analysis.

Our contributions are presented as follows:

- In Section 2, we describe the challenges linked with bottleneck identification; we propose a new definition of a bottleneck based on sensitivity analysis and show how this technique generalize existing approaches.
- In Section 3, we present the code analyzer Gus. We detail its underlying performance model and how it implements sensitivity analysis.
- In Section 4, we evaluate our approach on a set of high-performance computing kernels.

2 Understanding bottlenecks

Identifying the hardware resource (resp. resources) that is (resp. are) bottleneck for running a given program on a given microarchitecture is a difficult problem. Indeed, the construction of such information implies the aggregation and weighting of estimates relating to heterogeneous and intertwined components that are already difficult to understand separately, and even more so together. In order to build a performance model capable of highlighting such bottlenecks, it is therefore important to first acquire a good understanding of the microarchitecture, even at high level.

2.1 An Out-of-Order CPU architecture

Fig. 3 is a high-level simplified representation of a pipelined modern out-of-order CPU core. During its lifecycle, each μ op passes through 4 distinct states: it is *issued* by the front-end, *dispatched* by the allocator, *executed* by the execution units during its stay in the ROB, and eventually *retired* once its effects have been committed.

Front-end The Branch Prediction Unit (BPU) guides instruction fetching with a given Program Counter (PC), potentially speculating on the outcome of a branch. The instruction fetching unit then typically reads a binary chunk

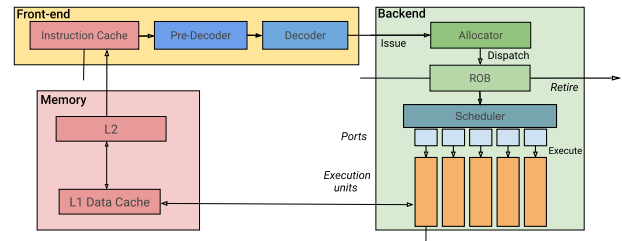


Figure 3. Simplified view of a pipelined OoO CPU core.

from the Instruction Cache (IC). The Pre-Decoder is thereafter responsible for translating this binary chunk into instruction(s), which are themselves broken down to micro-operations (μ ops) by the decoding unit. The decoded μ ops are filled into the Instruction Decode Queue (IDQ, also called Allocation Queue) while awaiting their delivery to the back-end.

Allocator/Renamer Each μ op produced by the front-end is first filled into a pipeline slot and assigned to an execution port by the Allocator[4]. Since different ports can have the same role, a given μ op can be assigned to any of these ports. This allocation mechanism is illustrated in Fig. 4, where a basic block consisting of the instruction sequence `mul sbb rol bsf rol pop mov sahf movsx sahf sbb xor` is scheduled on an Intel Skylake CPU (8 ports). For the sake of clarity, all of these instructions produces one μ op, and are assumed not to have dependencies. At the same time as allocation, registers are renamed, ensuring mapping between logical and physical registers. This operation is controlled by the Register Alias Table (RAT, also called Resource Allocation Table).

Re-Order Buffer The allocated μ ops are then used to fill the re-order buffer (ROB), which exploits instruction-level parallelism by enabling OoO execution. The ROB is a circular buffer that is filled incrementally, with μ ops being issued at the *tail* and retired from the *head*. Retiring a μ op from the ROB means that its effects on the architectural state of the pipeline are committed. The execution can be done out-of-order to increase the utilization of the execution units; two μ ops can be executed in parallel if they rely on two different and available resources and if they are not dependant on each other. However, retiring μ ops happens in-order to preserve the illusion of sequential execution. This idea is an extension of Tomasulo’s algorithm [52].

Instructions	Possible ports	Dispatched port
1 mul	p1	p1
2 sbb	p0 ∨ p0	p0
3 rol	p0 ∨ p6	p6
4 bsf	p1	p1
5 rol	p0 ∨ p6	p6
6 pop	p2 ∨ p3	p2
7 mov	p2 ∨ p3	p3
8 sahf	p0 ∨ p6	p6
9 movsx	p2 ∨ p3	p2
10 sahf	p0 ∨ p6	p0
11 sbb	p0 ∨ p6	p6
12 xor	p0 ∨ p1 ∨ p5 ∨ p6	p5

Figure 4. The allocation mechanism applied to a basic block of dependencies-free instructions (each being broken into exactly 1 μ op).

Time	ROB	μ op dispatch	μ op state
1	μ -mul	p1	D \rightarrow R
	μ -sbb	p0	D \rightarrow R
	μ -rol	p6	D \rightarrow R
	μ -bsf	p1	D (no change)
2	μ -bsf	p1	D \rightarrow R
	μ -rol	p6	D \rightarrow R
	μ -pop	p2	D \rightarrow R
	μ -mov	p3	D \rightarrow R
3	μ -sahf	p6	D \rightarrow R
	μ -movsx	p2	D \rightarrow R
	μ -sahf	p0	D \rightarrow R
	μ -sbb	p6	D (no change)
4	μ -sbb	p6	D \rightarrow R
	μ -xor	p5	D \rightarrow R
	-	-	-
	-	-	-

Figure 5. A ROB of size 4 processing the basic block in Fig. 4 along time, where μ -instr designates the μ op decoded from the instruction instr. "op state" column describes changes between D(ispatched) and R(etired) μ op state.

Fig. 5 illustrates this mechanism with a simplified ROB of 4 entries (instead of 224 in a Skylake micro-architecture), allowing 4 μ ops to be issued and retired per cycle. The latter is fed by the decoding of the basic block shown in Fig. 5:

- At cycle 1, the first 4 μ ops are dispatched. The μ ops μ -mul, μ -sbb and μ -rol are simultaneously executed by ports 1, 0 and 6. The μ op μ -bsf being mapped on the same port than μ -mul (*i.e.* the port 1), it can not be executed for now. The 3 executed μ ops are then retired.
- At cycle 2, the following 3 μ ops are dispatched. All the μ ops contained in the ROB being mapped on different ports (*i.e.* the ports 1, 6, 2 and 3), they are executed in parallel and then retired.
- At cycle 3, the following 4 μ ops are dispatched. The μ ops μ -sahf, μ -movsx and μ -sahf are simultaneously executed by ports 6, 2 and 0. The μ op μ -sbb being mapped on the

same port as μ -sahf (*i.e.* the port 6), it can not be executed for now. The 3 executed μ ops are then retired.

- At cycle 4, the last μ op of the sequence, μ -xor, is dispatched. μ -sbb and μ -xor being mapped on different ports, they are executed in parallel and finally retired.

Scheduler/Reservation Station Executing a μ op means transmitting it to the Scheduler, also known as Reservation Station. The latter holds each μ op until the execution port to which it is assigned is free and its dependencies are satisfied. Then, it is executed by its mapped port.

Execution ports and execution units Each port is linked to execution units responsible for a class of operations: arithmetic calculation and logic operations (ALU), divider, memory reads, memory writes, vector calculation, *etc.* A port processes a maximum of one μ op per cycle and is said to be saturated if it does not find itself waiting and can actually consume a new μ op as soon as it has finished with the previous one.

Memory communications During their execution, load-related μ ops are stored in Load Buffers (and store-related μ ops in Store Buffers). The execution units processing these memory-related μ ops (*e.g.* the Load-Store Units) have to wait for the data on which they operate, which induces *latencies*. Depending on the location of the required data, it must be incrementally repatriated from the farthest memory to the nearest one.

Repatriated or newly-computed data is stored in processor registers, can be vector or integer. Each kind comes with a dedicated file included within the Scheduler and which indicates their occupancy level; for example, the Vector Register File indicates which vector registers are available and which are occupied.

In addition to registers, L1 and L2 caches are core-specific, while L3 cache and RAM are system-wide unified memories.

2.2 Bottleneck analysis in state-of-the-art

Some code analyzers do not provide bottleneck analysis and are limited to throughput estimation. It is generally the case of neural-network based ones such as granite[51] and themal[38]. Others, without being black boxes, only partially model the microarchitecture and are therefore specialized for particular classes of programs. For instance, osaca[34] assumes that a basic block is never front-end bound. It produces, in addition to a port saturation analysis, a dependencies-based critical path analysis.

However, even less limited approaches, based on TAM or on code analyzers like iaca or uica, lack generality. They provide bottleneck analysis consisting in identifying under- or overexploited resources, possibly extended to include time lost due to bad speculation.

Load-based analysis: iaca, llvm-mca. The bottleneck analysis of iaca exploits the intermediate data produced

$$\begin{aligned}
 \text{Retiring} &= \frac{\text{SlotsRetired}}{\text{TotalSlots}} \\
 \text{Front End Bound} &= \frac{\text{FetchBubbles}}{\text{TotalSlots}} \\
 \text{Bad Spec} &= \frac{\text{SlotsIssued} - \text{SlotsRetired} + \text{RecoveryBubbles}}{\text{TotalSlots}} \\
 \text{Back End Bound} &= 1 - (\text{Front End Bound} + \text{Bad Spec} + \text{Retiring})
 \end{aligned}$$

Figure 6. The PMU-based formulas on which the Topdown Analysis Method relies at high level.

during a basic block throughput estimation. It first looks at the status of three simulated components belonging to the back-end: the Reservation Station (if full, the ALUs should be saturating), the Load Buffer (if full, the AGUs should be saturating) and the Vector Register File. One of these components being full, the back-end is bottleneck². Otherwise, it looks for a significant difference between the throughput prediction obtained when taking latencies into account and the throughput prediction obtained when ignoring them. If such a difference is found, latencies are considered to have a significant influence on the basic block performance: the basic block is latency-bound. Finally, if the bottleneck is neither back-end nor latency, then the front-end is bottleneck.

The `llvm-mca`[5] code analyzer, distributed as part of the LLVM project, is based on a similar approach. Like `iaca`, its bottleneck analysis engine exploits the underlying performance model used for throughput estimation. It provides indicators of back-end resources saturation and influence of dependencies-induced latencies on performance. The main difference with `iaca` is that the analyzer simply provides these values (in greater detail than `iaca`), without concluding that any of the related components is bottleneck. The conclusion is up to the expert.

Throughput-based analysis: `facile`, `uica`. These two code analyzers are built around the same performance model associating a throughput to each component of the CPU (like `iaca` and `llvm-mca`, they provide a statement of the pressure on resources at the instruction level). They differ simply in the algorithm used to make the basic block throughput estimation, which influences the bottleneck analysis. On the one hand, `uica`[8] associates the throughput of a basic block with the timestamp at which the last instruction in the (possibly iterated many times) basic block was retired. In this case, a bottleneck resource is one whose throughput exceeds a certain hard-coded proportion of the total system throughput (97% for some, 98% for others). On the other hand, `facile`[9] considers that the basic block throughput is the throughput of the slowest CPU resource. With this approach, the latter is reported as the (only) bottleneck limiting the performance.

²We have obtained this heuristic by reverse engineering the tool, but it is undocumented by Intel.

Top-down analysis Method: `perf`, `VTune`. Both `perf` and `VTune` use sampling techniques to inspect hardware counters. By default, they both collect the data required by the top-down analysis and provide a summary of the latter. At high level, it consists of classifying all the slots of an execution into 4 categories: slots lost due to bad speculation, slots lost due to back-end, slots lost due to front-end, and finally filled slots, i.e. associated with a fully executed and retired μop .

Fig. 6 shows the algorithm described in [55] to classify slots. Note that:

1. The number of filled/retiring slots is given by a dedicated counter (counter `uops_retired.retire_slots`).
2. The number of front-end bound slots is the number of “fetch bubbles”, i.e. empty slots at the front-end output causing back-end underfeeding. In practice, this means tracking μops not delivered from the IDQ to the RAT (counter `idq_uops_not_delivered.core`).
3. The number of bad speculation slots is equal to the number of issued μops (counter `uops_issued.any`) which have not been eventually retired from the ROB, plus “recovery bubbles”, i.e. empty slots produced by the Allocator because of recovery after bad speculation (counter `int_misc.recovery_cycles`, to be multiplied by 4 to convert cycles into slots).
4. All remaining slots are back-end bound.

If a resource loses more than a certain percentage of slots, it is considered bottleneck. The higher the percentage of filled slots, the more optimized the code is considered to be (assuming it is vectorized). The analysis can be carried out at a finer grained level (and, in the case of `VTune`, is pre-configured to [3]) to distinguish between branch mispredictions and machine clears, front-end bandwidth and front-end latency, different execution ports utilization, etc.. Even at this level, this approach makes it possible to identify *where* stalling occurs most (in which part of the microarchitecture), but not necessarily *when* it occurs in the sense of the Program Counter, due to the inherent inaccuracy of sampling techniques[57] and *2-* its ultimate cause. In the general case, both of these aspects require (human) additional analysis.

The ad-hoc nature of these analyses lies in the fact that they essentially rely on identifying resources that saturate enough or stalls enough. In an OoO architecture, where the exploitation of parallelism and the interdependence of resources have an essential impact on performance, such metrics are insufficient. They provide clues to guide the eye, but do not systematically identify the limiting resource, or resources, whose optimization will improve performance. We propose sensitivity analysis to address this limitation.

2.3 Why sensitivity analysis?

Fig. 7 illustrates the port pressure during the execution of the basic block in Fig. 4, assuming the simplified ROB of size 4 shown in Fig. 5. The stalling of port 0 at cycle 2 is apparent,

	1	2	3	4
p0	μ -sbb		μ -sahf	
p1	μ -mul	μ -bsf		
p2		μ -pop	μ -movsx	
p3		μ -mov		
p5				μ -xor
p6	μ -rol	μ -rol	μ -sah	μ -sbb

Figure 7. Port occupancy over time during the execution of basic block from Fig. 4 assuming each port can process at most 1 μ op per cycle.

			Acc. p0,	Acc.	Acc.
	Port	Time	p2, p3, p5	p6	p1
1	μ -mul	p1	1	1	1
2	μ -sbb	p0	1	1	1
3	μ -rol	p6	1	1	1
4	μ -bsf	p1	2	2	1
5	μ -rol	p0	2	2	2
6	μ -pop	p2	2	2	2
7	μ -mov	p3	2	2	2
8	μ -sahf	p6	3	3	2
9	μ -movsx	p2	3	3	3
10	μ -sahf	p0	3	3	3
11	μ -sbb	p6	4	4	3
12	μ -xor	p5	4	4	3
		4	4	4	3

Figure 8. The retirement timing of the sequence of μ ops produced from the basic block in Fig. 4, assuming the same 4-entries ROB as in Fig. 5 and Fig. 7. In the nominal case, all execution ports process one μ op per cycle. Accelerating a port, the latter processes two μ ops per cycle.

but beyond the fact that the size of the ROB intrinsically limits parallelism, understanding the cause of this behavior is not trivial. Identifying a bottleneck without sensitivity requires a detailed analysis of the execution and an in-depth understanding of the architecture. With an OoO architecture indeed, performance depends on the amount of parallelism the program manages to exploit, and a saturated resource does not automatically limit parallelism. Although such a metric is often a relevant clue when looking for a bottleneck, it can be ambiguous, and even misleading: in this example, the port 6, although completely saturated, does not drag down performance.

On the other hand, the sensitivity approach, which consists of successively accelerating different resources of the microarchitecture through new simulations, immediately reveals those on which performance depends directly. This is demonstrated in Fig. 8, where it can be seen that the execution of the μ op sequence costs one cycle less (3 instead of 4) when port 1 is accelerated. The latter consuming 2 μ ops per cycle, μ -mul (line 1) and μ -bsf (line 4) are executed simultaneously at the cycle 1. Consequently, the first μ -sahf

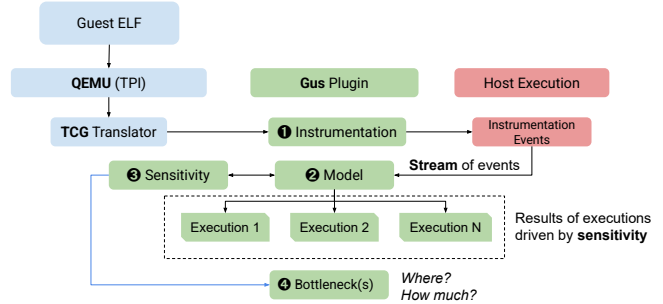


Figure 9. Overview of Gus's architecture.

(line 8) can be inserted in the ROB and executed at the cycle 2. This means that at cycle 3:

- The port 6 is no longer monopolized by the first μ -sahf (which ran at the previous cycle), allowing the second one (line 10) to run.
- An additional entry is available in the ROB, allowing the insertion (and execution) of μ -xor (line 12).

In the end, the execution costs 3 cycles instead of 4. The stalling cycle has finally been correlated to a saturation issue, but in the sense of the *momentary* saturation of port 1 at cycle 1.

3 Gus, a sensitivity-oriented code analyzer

We introduce in the following Gus, an instruction driven simulator for performance debugging. Enabling *sensitivity* analysis, Gus estimates the execution time, Instructions Per Cycle, resources occupancy (global and instruction-wise), latency for a given program.

First, we explain the design choices that led to the creation of an efficient instruction driven simulator (1). Second, we detail the underlying performance model of Gus based on abstract resources (2). Finally, we demonstrate the usefulness of sensitivity and how it overcomes the shortcomings of previous bottlenecks detection techniques (3, 4).

We give a brief overview of Gus's architecture in Fig. 9.

3.1 A dynamic binary instrumentation based frontend

The front-end of Gus is built around QEMU [13], a fast functional processor emulator. It relies on user-mode dynamic binary instrumentation within a QEMU plugin to generate the stream of events that drive the simulation.

This is mostly motivated by the fact that execution traces can become extremely large, thus storing them on disk is not usually an option for scalability reasons.

While static code analyzers [2, 8] typically work on a smaller scope, that is a single basic block, Gus thanks to its trace can work on a much larger scope chosen by the user, for instance a kernel or a whole program.

Another advantage of having a trace is that it gives Gus a lot of information about the execution of the program, such as whether a branch was taken or not, if two memory accesses alias, *etc.*. Static code analyzers don't have access to such dynamic information and thus have to make assumptions that may not hold in practice. Memory dependencies are a good example of this, as most code analyzers ignore them completely, that leads to imprecise results [44].

Instrumentation Many dynamic binary instrumentation frameworks exist, such as Pin [36], Valgrind [41], or DynamoRIO [15]. Gus is rather based on a modified version of QEMU [24]. Multiple reasons motivated this choice, that we briefly describe here.

First, contrary to other frameworks, QEMU is able to cross translate between different ISAs and application binary interfaces (ABIs). This is a major advantage for a performance debugging tool, as it allows getting interesting insights on micro-architectures that are not directly accessible to the user, or even that have scarce hardware performance counters (e.g. such as a lot of ARM or RISC-V based cores).

Second, the IR (Intermediate Representation) used by QEMU called TCG (Tiny Code Generator) IR is quite amenable to our needs of instrumentation as it abstracts away the details of the underlying ISA and ABI. While Pin [36] and DynamoRIO [41] typically introduce a much lower overhead than QEMU, they do not have such an abstraction layer as the guest code is not transformed into a very different representation.

TCG is essentially a trace based JIT [39]. Instead of translating whole functions or the program at once, TCG works on small linear sequences of guest instructions called trace blocks. These trace blocks are formed from the guest program using a very simple mechanism. Whenever the guest CPU executes an instruction that has not been translated yet a new trace block is started. This first instruction of the trace block is referred to as its entry. QEMU then parses the guest instructions following the entry until it hits a branch instruction.

The modified QEMU used by Gus has a mechanism for loading plugins, called the TCG plugin infrastructure (TPI), that can both observe and alter the translation process of TCG IR. Such plugin can inject TCG IR instructions into TBs before they get translated back to host machine code.

Gus uses this layer to add instrumentation into the guest code. We monitor multiple events such as when an instruction is executed, when a memory access occurs, when a branch is taken, *etc.* as a small example in Fig. 10 demonstrates. The number of events is kept to a minimum to reduce the overhead of the instrumentation. This instrumentation of course requires a mapping between the guest instructions and the resulting TCG IR instructions, this is possible by disassembling the guest code with the API provided by the TPI infrastructure.

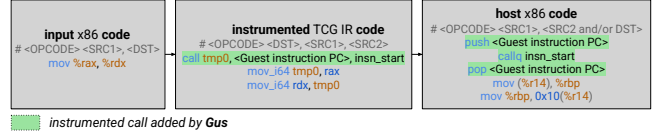


Figure 10. Different stages of a trace block in QEMU and the corresponding instrumentation generated by Gus.

3.2 An abstract resource-centric performance model

Gus's abstract performance model describes a modern out-of-order CPU. It consumes events fed by the instrumentation frontend and simulates the state of this CPU. The following components are modeled:

- A set of abstract throughput limited resources that model, amongst other things, the execution ports and the bandwidth between different levels of the cache.
- A finite-sized instruction window which models the *ROB*.
- A shadow *memory* and shadow *register file* used to track data dependencies.
- A *cache* simulator to detect cache misses.
- A *branch prediction unit* simulator.

The underlying algorithm is described in Algorithm 1. We will refer to it when describing the different components of the model.

The instruction window is at the heart of the model, and dictates when an instruction can be issued, its execution starting time t_{start} is determined by the resources it uses and data dependencies (lines 13, 15 and 17). The time it retires t_{end} is simply defined as $t_{\text{start}} + \text{instruction}_{\text{latency}}$ (line 18).

3.2.1 Abstract resources The building blocks of the model are the abstract throughput limited resources. In order to execute an instruction, one or more of these resources may be used and must be available. Otherwise, the instruction is stalled until all the required resources become available. Each resource tracks the time at which it will be available to accept another request, this is characterized by its throughput.

The timestamp at which a resource R is available to accept an instruction is denoted as $R.t_{\text{avail}}$. Every time a resource is used its t_{avail} is incremented by its inverse throughput (or *gap*) (lines 9 and 20), since it determines the minimum amount of time that must elapse between two uses of the resource.

Frontend The frontend is modeled as a single resource with a throughput of four instructions per cycle. This is an optimistic assumption that may hide some fine-grained bottlenecks, further work is needed to model the frontend more precisely, as it may have been done in other works [8].

Execution ports and functional units The classical formalism to describe the throughput and sharing of backend

resources is a port mapping, a tripartite graph, which describes how instructions decompose into μ ops and which functional units μ ops can execute on.

Gus instead uses a simpler two-level representation [19], called a resource mapping, where instructions are directly associated with a list of abstract resources. To account for μ op decomposition, a resource can appear in this list multiple times. This representation avoids the need to explicitly model the port scheduling algorithm in the backend and thus allows to achieve a lower runtime overhead.

Latencies The latencies used for execution units in our experiments are extracted from UOPS.INFO [6] and used to build a resource mapping. For bottleneck analysis, we present latencies induced by dependencies as a separate resource. The same approach can be found, for instance, in iaca. The latter helps guide optimization by highlighting the need to discover more parallelism, rather than the need to relieve the load on a hardware component.

Instruction window The out-of-order execution of instructions inside the ROB is modeled by an instruction window, that is to say a finite-sized buffer of instructions that are in-flight (issued but not retired).

The instruction window is bounded by the number of slots it has, thus if the window becomes full, no other instructions can be issued until another one retires. We denote t_{\min} as the earliest time a slot in the window will become free (line 28).

Shadow memory and register file The shadow memory and shadow register are used to detect data dependencies between instructions.

For each memory cell or register they store the time at which the data in this location will be available. The update mechanism is twofold.

First, when an instruction writes to a location, the shadow cell for that location will be set to the time when the instruction is retired (line 26). Second, when a cache miss occurs this counts as a use of all levels of the memory hierarchy up to the one where the miss occurred. The location in the shadow memory corresponding to the accessed memory location is then updated to the maximum availability of the involved resources (line 5).

Caches Gus uses a fork [23] of the Dinero IV [21] cache simulator to simulate hits and misses in the different levels of the cache hierarchy. The PLRU (Pseudo-LRU) replacement policy from this fork is used for all levels of the cache hierarchy, however cache replacement policies implemented in commercial processors are a complex topic [7] and an exhaustive modelization is out of the scope of Gus that acts mostly as a generic modern out-of-order CPU model.

Branch prediction unit simulator The branch prediction unit is composed of a branch target buffer (BTB) modeled as an associative array with an LRU replacement policy,

and accompanied by an indirect branch predictor (IBP) with a IITage scheme [49] and a conditional branch predictor (CBP) with an L-TAGE scheme [48]. Findings in the literature [45, 50, 56] show that such schemes or close derivatives are actually used in x86 commercial processors and thus good candidates for the model of Gus.

3.2.2 Core performance model To achieve a reasonable execution time, some simplifying assumptions are made to the model.

- Gus assumes that the program has regular access memory patterns with latencies that can be perfectly hidden by prefetching, similarly to what is done in the ECM [25, 27] model.
- Gus does not model either load/store queues or load-store forwarding. Thus the bandwidth between the CPU and L1 is considered infinite and not modeled as a resource.
- Gus also does not model execution pipeline hazards or operand forwarding.

This high-level model is sufficient to achieve state-of-the-art precision as we will see in subsection 4.2.

Algorithm 1: Core algorithm of Gus

```

1 function UpdateCaches(i, type)
2   foreach loc ∈ LineAccesses(i, type) do
3     l ← LowestCacheLevelHit(loc)
4     foreach l' ∈ CachesUpTo(l) do
5       shadow[loc] ← max(shadow[loc], l.tavail)
6     foreach l' ∈ CachesUpTo(l) do
7       // L1 Misses are considered free
8       if l' = l ∧ l' = L1 then
9         break
9     l.tavail ← l.tavail + l.gap
10 function Simulate()
11   foreach i ∈ T do
12     // Only update cache loads at this point as stores
13     // may use bandwidth later
14     UpdateCaches(Load)
15     t_start ← t_min
16     foreach loc ∈ i.reads do
17       t_start ← max(t_start, shadow[loc])
18     foreach res ∈ i.resources do
19       t_start ← max(t_start, res.tavail)
20     t_end ← t_start + i.latency
21     // Update execution resources
22     foreach res ∈ i.resources do
23       res.tavail ← max(t_min, res.tavail) + res.gap
24     UpdateCaches(i, Store)
25     foreach loc ∈ i.writes do
26       // Assume register renaming works perfectly
27       if loc ≡ Register then
28         shadow[loc] ← t_end
29       else
30         shadow[loc] ← max(shadow[loc], t_end)
31   window.push(t_end)
32   t_min ← window.oldest()
33   i.t_end ← t_end

```

3.3 A sensitivity-oriented code analyzer

Sensitivity analysis in the broader world Sensitivity analysis allows Gus pinpointing precisely across instructions, the source of a performance bottleneck. We briefly describe

here how it compares with other related works and how it is implemented and used in Gus.

Sensitivity analysis [28, 33], also called differential profiling [37] works by executing a program multiple times, each time varying the usage or capacity of one or more resources. Bottlenecks can then be identified by observing by how much the change for each resource impacts the overall performance, that is how sensitive performance is to a given resource, and more precisely in our case a *microarchitectural* resource.

Moreover, sensitivity analysis is an automatic approach that does not require as much expertise as other ad-hoc techniques implemented in performance debugging tools [18, 26].

While seemingly simple, this approach requires considering careful trade-offs to make it practical. Today’s hardware does not offer many ways to easily vary the capacity of resources, motivating the need for a model such as the one in Gus.

On the other hand, the precision level of the analysis is also a key factor in a model. As a rule of thumb, the more precise the analysis, the more expensive it is to run. Gus contains an abstract high-level CPU core model that is driven by reverse-engineered microarchitectural features [6, 19] of modern commercial OoO cores that allows a certain degree of generality.

Other models embedded in tools such as instruction driven simulator [46] or cycle-accurate simulators [35] usually feature a very detailed heavyweight CPU model. For example, gem5 [35] at its highest level of precision requires, on average, one year to run a single SPEC2006 benchmark [47]. A common approach to speed up simulations is to only simulate parts of a program [54] or to only simulate some aspect of the processor [17, 40] However, getting the right level of granularity is a difficult task and requires a lot of expertise.

On the other side of the spectrum, if a model is too coarse-grained, it may lead a user to focus on the wrong bottleneck. Moreover, if a tool does not give an upper-bound on the optimization potential of a bottleneck, one can end up wasting time on a bottleneck that is not worth optimizing. Hence, we believe that these two aspects are key to the usefulness of a performance debugging tool, as the expert-user time is a scarce resource.

The idea of varying microarchitectural features is not new in itself either, as it has been used by hardware designers for Microarchitecture Design Space Exploration [10, 30] to guide microarchitecture parameter tuning to explore the trade-offs amongst performance, power, and area (PPA).

DECAN [33] is a dynamic performance analysis tool based on the MAQAO [20] binary analysis and instrumentation framework. DECAN finds bottlenecks by sensitivity analysis based on binary rewriting. That is, DECAN removes or modifies instructions in a kernel and checks which how much each transformation affects overall performance. DECAN measures

the performance of its modified kernels via hardware counters. The low overhead of this approach allows it to quickly explore a large set of variants for its sensitivity analysis. The downside of DECAN’s approach is that its transformations are of course not semantic preserving and can easily introduce crashes or floating-point exceptions. Changing the semantics of a program like this might, of course, also change its performance behavior in other subtle ways, making it hard to verify or falsify the results produced by the tool.

The SAAKE system [28] is conceptually closer to Gus in its implementation of sensitivity analysis. It uses a fast symbolic execution engine that estimates the runtime of GPU programs to drive sensitivity analysis for finding bottlenecks. SAAKE input independent abstract simulation works well for the simpler microarchitectures of GPUs since they do not use out-of-order execution or speculation and handle branching control flow using predicated execution. Since SAAKE does not actually simulate the execution of instructions, there are several things it can not compute that have to be provided externally.

Sensitivity analysis in Gus A resource is any finite quantity which can have an influence on the execution time of a program. These may be quantities immediately linked to a hardware component, such as the size of the ROB, or more abstract resources, such as instructions latency. We represent each of these quantities by a real number. The sensitivity analysis implemented in Gus consists in varying the n resources represented in the model during successive execution time estimates performed on the same program p . At each iteration, the capacity c_r of a resource r is successively weighted (*accelerated*) by real numbers w_0, \dots, w_m to discover a w that minimizes the estimation function f_p . In this framework, the other resources capacities and the input program p can be seen as constants. Thus, f_p is a function from real numbers to real numbers, associating with the value of r , an estimate of duration t . The speed-up s_r obtained by weighting c_r by w is therefore calculated as follows:

$$s_{w,r} = \frac{f_p(c_r)}{f_p(wc_r)} - 1$$

Resources whose variations result in a speedup greater than 0 (or any other *ad-hoc* minimum threshold) are bottleneck resources, and resources that reach the highest speedups should be the focus of optimization efforts.

In the general case, the complexity of this algorithm is $O(n * m)$, n being the number of the model resources, and m the number of weight candidates applied to each of the latter. However, at a first glance, looking at a single weight value may be sufficient for answering where are the bottlenecks of a program before precisely quantifying their overall impact.

A visual representation of the results of the sensitivity analysis produced by Gus is shown below. We use a form of heat-map for the visualization. Each bar in it represents

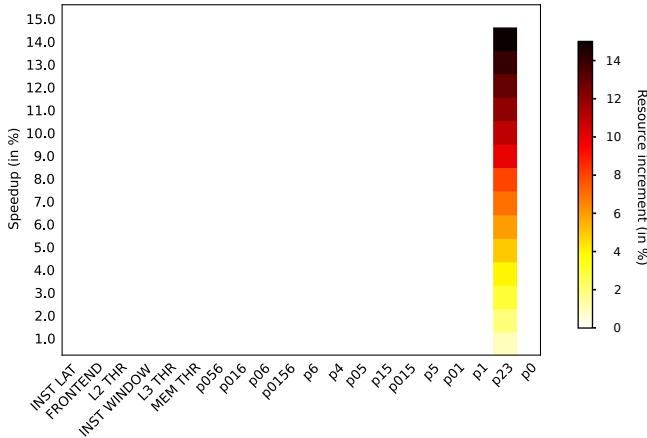


Figure 11. Sensitivity analysis report produced by Gus (Sky-lake architecture) for the Jacobi iteration method implementation discussed in Section 1.

one abstract resource. The height of each bar and its color indicate the speedup predicted by Gus if the throughput of that resource is increased.

Fig. 11 depicts the view of bottlenecks for the Jacobi example discussed in Section 1. The resource p23 represents the throughput of the combined execution ports 2 and 3. The graph shows that an increment of its throughput by 1% decreases the overall runtime of the kernel by 1%. If we increment the resource’s throughput by 2%, the runtime decreases by approximately 2%. The bar fades to black around the 13 – 14% mark, which indicates that a 15% increase in throughput produces a 13 – 14% speedup. All other resources immediately fade to white at the 0% mark. That is, according to Gus, the runtime of the kernel is not affected by increasing the throughput of any other resource. Hence, p23 is the bottleneck of this version of the Jacobi kernel.

In addition to execution ports, resources which are subject to sensitivity analysis in Gus are: the instruction latencies (INST_LAT), the front-end throughput (FRONTEND), the size of the ROB (INST_WINDOW), and the communication bandwidths between the different memory levels (L2_THR, L3_THR and MEM_THR).

Instruction-level analysis In a second step, one can look at the resource usage per instruction to get a more precise idea on where is the bottleneck in the kernel.

This is done by looking at the fine grain report shown in table 1. We see that multiple instructions use the resource p23 by loading from memory. Reducing this, for instance by applying a register tiling transformation, would reduce the pressure on this resource and thus improve the overall performance of the program.

This level of precision is as we believe particularly useful for performance debugging, as it allows a level of granularity

in the general case that is not possible with other state-of-the-art tools using for example performance counters [55].

4 Experiments

Our experiments aim first at evaluating the accuracy of the throughput estimation tools including Gus to assess the relevance of the underlying models. We believe that having a good model is a prerequisite to performing accurate bottleneck analysis. We then compare the bottleneck analysis capabilities of Gus to other state-of-the-art tools.

4.1 Benchmarking framework

Our benchmarking framework relies on CesASMe[12] and PolyBench [43].

CesASMe is a benchmarking harness designed to generate a large number of microbenchmarks from an existing benchmark suite, and to evaluate the ability of code analyzers to estimate their execution time.

Polybench is a well-known benchmark suite within the compiler community. It contains a range of thirty numerical computations kernels with static control flow, extracted from operations in various application domains (linear algebra computations, image processing, physics simulation, dynamic programming, statistics, etc.) that have attracted particular optimization efforts.

Performance models have been shown to yield higher error rates for throughput prediction on PolyBench than on SPEC2017 [19], making it a challenging dataset for such tools. In order to maximize diversity in the studied programs and thus to stress different aspects of the performance models, we use CesASMe to generate 1421 unique microbenchmarks from this initial suite using a three stages process:

1. We use C loop nest optimizers – Pluto[14] and PoCC [42] – to generate different versions of each benchmark gathered in PolyBench, each coming from a different high-level loop optimisation [16]: tiling, loop fusion, unroll and jam, etc. Some transformations have no impact on the input benchmark; for example, loop fusion applied to a program containing a single loop. In this case, the fresh version of the benchmark, identical to the former one, is discarded.
2. We extract the innermost loop (the kernel) from each of the generated benchmark and discard its surrounding loop nest to prevent it from being memory-bound (due to the memory hierarchy not being taken into account by `ica`, `uica`, ...) and to limit the execution time of `perf` and Gus. Similarly, we use `valgrind` [41] to discard benchmarks whose inner loop is not L1-resident.
3. We introduce even more combinatorics by generating different binaries from each resulting C file using `gcc` options (auto-vectorization, unrolling, etc.).

Since Gus can analyze a whole function while other code analyzers can only work on one basic block at a time, we also use the method detailed in [12] to lift the estimates they

PC	ASM	L2	p056	p016	p06	p0156	p4	p015	p01	p23	F-E	LAT/PORTS
12bb	mov rdx, qword ptr [rsp - 0x10]	0%	0%	0%	0%	0%	0%	0%	0%	10%	5%	2/p23
12c0	vmovsd xmm0, qword ptr [rdx + rax]	0%	0%	0%	0%	0%	0%	0%	0%	10%	5%	4/p23
12c5	vaddsd xmm0, xmm0, qword ptr [rdx + rax + 8]	0%	0%	7%	0%	5%	0%	7%	10%	10%	5%	4/p016 p01 p015 p0156 p23
12cb	vaddsd xmm0, xmm0, qword ptr [rdx + rax + 0x10]	0%	0%	7%	0%	5%	0%	7%	10%	10%	5%	4/p016 p01 p015 p0156 p23
12d1	vmulsd xmm0, xmm0, xmm1	0%	0%	7%	0%	5%	0%	7%	10%	0%	5%	4/p016 p01 p015 p0156
12d5	mov rdx, qword ptr [rsp - 0x18]	0%	0%	0%	0%	0%	0%	0%	0%	10%	5%	2/p23
12da	vmovsd qword ptr [rdx + rax + 8], xmm0	1%	0%	0%	0%	0%	20%	0%	0%	0%	5%	4/p4
12e0	mov rdx, qword ptr [rsp - 0x18]	0%	0%	0%	0%	0%	0%	0%	0%	10%	5%	2/p23
12e5	vmovsd xmm0, qword ptr [rdx + rax - 8]	0%	0%	0%	0%	0%	0%	0%	0%	10%	5%	4/p23
12eb	vaddsd xmm0, xmm0, qword ptr [rdx + rax]	0%	0%	7%	0%	5%	0%	7%	10%	10%	5%	4/p016 p01 p015 p0156 p23
12f0	vaddsd xmm0, xmm0, qword ptr [rdx + rax + 8]	0%	0%	7%	0%	5%	0%	7%	10%	10%	5%	4/p016 p01 p015 p0156 p23
12f6	vmulsd xmm0, xmm0, xmm1	0%	0%	7%	0%	5%	0%	7%	10%	0%	5%	4/p016 p01 p015 p0156
12fa	mov rdx, qword ptr [rsp - 0x10]	0%	0%	0%	0%	0%	0%	0%	0%	10%	5%	2/p23
12ff	vmovsd qword ptr [rdx + rax], xmm0	0%	0%	0%	0%	0%	20%	0%	0%	0%	5%	4/p4
1304	add rax, 0x18	0%	0%	0%	0%	0%	0%	0%	0%	0%	5%	1/
1308	cmp rax, rcx	0%	0%	0%	0%	5%	0%	0%	0%	0%	5%	1/p0156
130b	jne 0x12bb	0%	7%	7%	10%	5%	0%	0%	0%	0%	5%	1/p056 p016 p06 p0156

Table 1. Fine grain report analysis of resource usage by instruction by Gus on the Jacobi kernel (Skylake architecture). Resources with only zero usage across all instructions are omitted. The cells colored in orange indicate the bottleneck port.

produce. In practice, we sum up the results obtained on each basic block traversed by the program’s control flow, which provides an estimate for the whole microbenchmark.

4.2 Throughput estimation accuracy

We compare the measurements on the Skylake microarchitecture provided by the hardware counters³ and the Gus estimate with the following code analyzers: llvm-mca [5] (v13.0.1), uica [8] (commit 9cbb93), iaca [2] (v3.0-28-g1ba2cbb), ithemal [38] (commit b3c39a8).

The error distributions of each code analyzer tools compared to ground truth measurements, as well as synthetic statistical indicators are reported in Fig. 12. Gus achieves the highest MAPE and Kendall tau [31] ($\tau_K \in [0, 1]$, which measures the fraction of preserved pairwise ordering) scores. uica and iaca perform similarly, falling just behind Gus. Ithemal [38] is the overall lesser accurate tool, with a MAPE of 56.08 %.

4.3 Bottleneck analysis

We present in Table 2 a comparison of bottlenecks measured on a dual socket Intel Xeon Gold 6230R CPU (Cascade Lake generation, Skylake-X microarchitecture) by different methods over a set of micro-benchmarks extracted from our benchmarking harness. The extracted subset used a “vanilla” compilation strategy, using only the `-O3` and `-march=skylake` flags, with no transformation from any of the loop nest optimizers.

We compare several state-of-the-art tools for bottleneck analysis: code analyzers (uica [8], iaca [2]), TAMd [55] and Gus. For TAM, we used TopLev [32], an open-source

³On a Dell PowerEdge C6420 machine (Grid5000 cluster [11]), equipped with two Intel Xeon Gold 6130 CPUs (x86-64, Skylake microarchitecture).

Benchmer	MAPE	Median	Q1	Q3	τ_K
llvm-mca	32.69 %	23.43 %	10.41 %	51.62 %	0.61
uica	26.43 %	15.59 %	6.70 %	42.39 %	0.64
ithemal	56.08 %	46.20 %	20.81 %	74.30 %	0.40
iaca	27.48 %	16.11 %	6.70 %	49.15 %	0.64
Gus	20.27 %	14.72 %	5.84 %	30.41 %	0.82

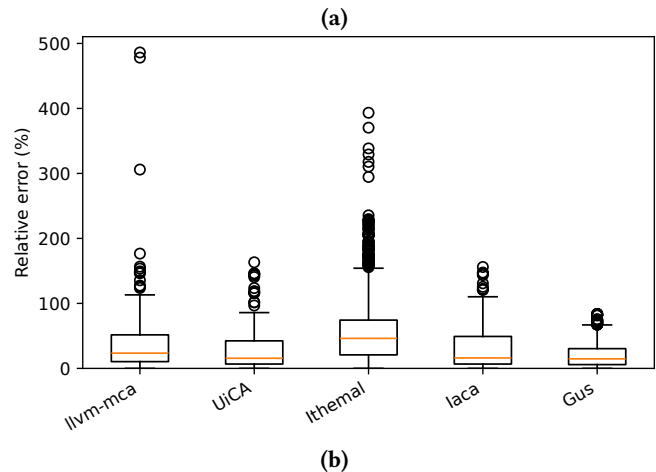


Figure 12. Statistical comparison (a) and corresponding error distribution (b) of throughput estimation tools on micro-benchmarks extracted from PolyBench [43].

implementation of TAM 4.7 for Intel CPUs that builds upon perf [1] to access hardware counters.

The perf-based approach, contrary to code analyzers like uica or iaca cannot limit the analysis to a basic block; and, contrary to Gus, cannot limit the analysis to a function call. Thus, initialization/instrumentation wrappers used by the

PolyBench suite can mislead the former. To ensure a fair comparison, we limited as much as possible this extra code.

As code analyzers typically work on a single basic block, reported bottlenecks are the ones inferred for the hottest basic block of the kernel of each benchmark. We assume that this is representative of the overall bottleneck of the benchmark as our generation approach ensures that we tackle only the loop whose execution time dominates the overall execution time.

Bottlenecks are given as is by the tools. If a tool does not provide a bottleneck for a given benchmark, they are marked as Unknown in the table. For *uica*, we explain this by the arbitrary thresholds used to infer bottlenecks (if the contribution to the total throughput of a given resource doesn't exceed a given threshold, it is not reported as a bottleneck).

We also want to emphasize that a definite automatized groundtruth for bottlenecks does not exist at the time of writing, as such our comparison focuses on the strengths and weaknesses of each tool.

The retiring (*Ret*) category is not referred as a bottleneck *per se* by the TAM, because it denotes when the pipeline is busy with typically useful operations. Non-vectorized code might have a high retiring rate where it is considered as a bottleneck, however we deal here with vectorized code where this rather indicates that no bottlenecks have been found by the automatic analysis. We see multiple such rows (*syr2k*, *correlation*, *covariance*) in the table where the TAM only reports “Retiring” as a bottleneck and whereas code analyzers refine this to a bottleneck. This category can otherwise correspond to a what's referred as “Frontend” (*F-E*) bottleneck by code analyzers or Gus.

Moreover, the TAM 4.7 does not make the distinction between what is referred as dependencies-bound (*Deps*) or latency-bound (*Lat*) by code analyzers or Gus and the more general “Backend” (*B-E*) category. This is the case for eleven rows, where both *uica* and Gus agree on this “Dependencies” bottleneck and the TAM rather reports “Backend” as a bottleneck. We are not aware of any performance counters that would allow distinguishing between these two categories, as such this shows the usefulness of code analyzers and Gus to provide more fine-grained bottlenecks in this case.

Gus and *uica* are able to provide multiple bottlenecks, this is shown in the row of *jacobi-2d* where Gus reports two bottlenecks: latency and ROB ; or in the row of *gemm* where *uica* reports two bottlenecks: decoder and predecoder. This is an inherent limitation for the underlying decision-tree used by IACA 3.0. The automated implementation of the TAM 4.7 in TopLev only reports a single critical bottleneck.

Gus is able to provide the upper-bound on the speedup that can be achieved by removing the bottleneck when accelerating the corresponding resource. This allows to assess the potential of a given optimization.

```

1 for(t6 = 1; t6 < 99; ++t6) {
2     p[t4][t6] = -c/(a*p[t4][t6-1]+b);
3     q[t4][t6] = (-d*u[t6][t4-1]+(1.0
4         + 2.0*d)*u[t6][t4] - f*u[t6][t4+1]
5         - a*q[t4][t6-1]) / (a*p[t4][t6-1]+b);
6 }

```

Figure 13. *adi* benchmark.

Benchmark	TopLev	<i>uica</i>	<i>iaca</i>	Gus
2mm	B-E (60.9%)	Deps	Deps	Lat (14.9%)
3mm	B-E (61.0%)	Deps	Deps	Lat (14.9%)
atax	B-E (61.4%)	Deps	Deps	Lat (15.0%)
bigg	B-E (61.3%)	Deps	Deps	Lat (15.0%)
doitgen	B-E (34.2%)	Ports	B-E	Frontend (0.8%), {P23, F-E, P4} (14.9%)
mvt	B-E (62.6%)	Unknown	F-E	Lat (14.9%)
gemver	Ret (77.3%)	Unknown	B-E	P23 (7.4%)
gesummv	B-E (60.5%)	Deps	Deps	Lat (15.0%)
syr2k	Ret (74.0%)	Deps	Deps	Lat (15.0%)
trmm	Ret (95.6%)	Unknown	F-E	F-E (14.9%)
symm	B-E (53.1%)	Deps	B-E	Lat (14.9%)
syrk	B-E (47.6%)	Deps	B-E	Lat (14.9%)
gemm	Ret (78.4%)	Decoder Predecoder	F-E	F-E (14.9%)
gramschmidt	B-E (55.0%)	Deps	Deps	Lat (14.9%)
cholesky	B-E (60.7%)	Deps	Deps	Lat (14.9%)
durbin	Ret (95.7%)	Decoder Predecoder	F-E	F-E (14.9%)
ludcmp	B-E (74.4%)	Deps	B-E	Lat (14.9%)
trisolv	B-E (60.8%)	Deps	Deps	Lat (15.0%)
jacobi-1d	Ret (77.7%)	Issue, LSD	B-E	Frontend (4.8%), {Lat,F-E} (15.0%)
heat-3d	Ret (79.3%)	Unknown	F-E	Lat (4.1%), ROB (4.1%), P4 (2.2%) {Lat, F-E, P01, P1, P23, P4} (15.0%)
seidel-2d	B-E (89.9%)	Deps	B-E	Lat (14.9%)
fdtd-2d	B-E (78.9%)	Ports	F-E	P23 (13.4%)
jacobi-2d	B-E (78.8%)	Unknown	B-E	Lat (7.8%), ROB (7.6%)
adi	B-E (79.5%)	Divider	B-E	Lat (14.9%)
correlation	Ret (93.5%)	Decoder Predecoder	F-E	F-E (14.9%)
covariance	Ret (78.8%)	Unknown	F-E	F-E (14.9%)
floyd-warshall	F-E (43.6%)	LSD	Deps	F-E (14.9%)
nussinov	Ret (97.3%)	LSD	Deps	F-E (14.9%)
deriche	B-E (64.7%)	Deps	B-E	Lat (14.9%)

Table 2. Comparison of bottleneck analysis tools on microbenchmarks extracted from PolyBench. Percentages in the Gus column correspond to the upper-bound on the speedup with a 15% resource acceleration (sub-1% values omitted); percentages in the TopLev column represent the proportion of slots used by the resource. Gray bottlenecks for Gus are found using sensitivity analysis on single resource, while red ones are found when varying multiple resources.

One apparent discrepancy between Gus and *uica* is shown in the *adi* benchmark, whose C code is reported in Fig. 13: *uica* reports that the divider is the bottleneck, whereas Gus reports that the latency is the critical bottleneck.

A deeper analysis of the code shows that a dependency chain is present in the loop body, indeed the previous *p* array value is used to compute the next *p* array value, the same is true for the *q* array. Static code analyzers such as *uica* are oblivious to memory dependencies [44] in this basic block, as such they cannot detect the dominant latency bottleneck. This highlights the usefulness of the dynamic nature of Gus which can easily detect this using a shadow memory.

Last but not least, we want to detail a category of benchmarks (heat-3d, dotigen, jacobi-1d) that depict a certain equilibrium between their resources. These benchmarks usually feature a very high retiring rate (nearing 80%) as reported by TopLev and are very well optimized as the balance between their resources is very good. Optimizing a single resource would not result in any significant speedup as this would not affect the overall balance between the resources. This is shown by Gus single resource sensitivity analysis, where reported bottlenecks feature a very low potential speedup ($< 5\%$). Optimizing for a set of resources, can feature a higher potential speedup, we again demonstrate this using Gus multiple resource sensitivity analysis. Uncovering this class of well-balanced benchmarks leads us to think that a more complete framework for reasoning about bottlenecks is needed, especially for intertwined resources.

5 Conclusion

Identifying performance bottlenecks in programs that need to make the most of the architecture is an increasingly critical task. We present micro-architectural mechanisms on which program performance depends, as well as the metrics and analysis methods that have been designed to guide optimization. We propose to generalize bottleneck analysis by means of microarchitectural sensitivity. It aims to automatically discover which resources influence a program's overall performance by successively accelerating each of them and observing the overall speedup generated by this variation. We present Gus, a dynamic code analyzer that implements sensitivity analysis. We evaluate both its performance model and sensitivity analysis algorithm on a set of microbenchmarks generated from the PolyBench benchmark suite, and highlight its strengths and limitations compared with existing methods based on hardware counters or performance models. Gus's performance model achieves state-of-the-art accuracy in throughput estimation over our experimental harness. As for sensitivity analysis, it allows us to enhance the results of existing approaches, especially where saturation is not sufficient to identify the one that limits parallelism.

References

- [1] Performance analysis tools for linux. <https://man7.org/linux/man-pages/man1/perf.1.html>.
- [2] Intel architecture code analyzer user's guide. <https://www.intel.com/content/dam/develop/external/us/en/documents/intel-architecture-code-analyzer-3-0\users-guide-157552.pdf>, 2017.
- [3] Intel vtune profiler performance analysis cookbook. <https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookbook/2024-0/overview.html>, 2021.
- [4] Intel 64 and ia-32 architectures optimization reference manual, 2023. <https://cdrdv2-public.intel.com/671488/248966-046A-software-optimization-manual.pdf>.
- [5] Llvmlite machine code analyzer. <https://llvm.org/docs/CommandGuide/llvm-mca.html>, 2023.
- [6] Andreas Abel and Jan Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *ASPLOS, ASPLOS '19*, pages 673–686, New York, NY, USA, 2019. ACM.
- [7] Andreas Abel and Jan Reineke. nanobench: A low-overhead tool for running microbenchmarks on x86 systems. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, August 2020.
- [8] Andreas Abel and Jan Reineke. Uica: Accurate throughput prediction of basic blocks on recent intel microarchitectures. In *Proceedings of the 36th ACM International Conference on Supercomputing, ICS '22*, New York, NY, USA, 2022. Association for Computing Machinery.
- [9] Andreas Abel, Shrey Sharma, and Jan Reineke. Facile: Fast, accurate, and interpretable basic-block throughput prediction. In *2023 IEEE International Symposium on Workload Characterization (IISWC)*, pages 87–99. IEEE Computer Society, 10 2023.
- [10] Chen Bai, Jiayi Huang, Xuechao Wei, Yuzhe Ma, Sicheng Li, Hongzhong Zheng, Bei Yu, and Yuan Xie. Archexplorer: Microarchitecture exploration via bottleneck analysis. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '23*, page 268–282, New York, NY, USA, 2023. Association for Computing Machinery.
- [11] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. Adding virtualization capabilities to the Grid'5000 testbed. In Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan, editors, *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer International Publishing, 2013.
- [12] Théophile Bastian, Hugo Pompougnac, Alban Dutilleul, and Fabrice Rastello. Cesasme and staticdeps: static detection of memory-carried dependencies for code analyzers, 2024.
- [13] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '05*, page 41, USA, 2005. USENIX Association.
- [14] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. Pluto: A practical and fully automatic polyhedral parallelizer and locality optimizer. Technical Report OSU-CISRC-10/07-TR70, The Ohio State University, October 2007.
- [15] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, CGO '03*, page 265–275, USA, 2003. IEEE Computer Society.
- [16] João M.P. Cardoso, José Gabriel F. Coutinho, and Pedro C. Diniz. Chapter 5 - source code transformations and optimizations. In João M.P. Cardoso, José Gabriel F. Coutinho, and Pedro C. Diniz, editors, *Embedded Computing for High Performance*, pages 137–183. Morgan Kaufmann, Boston, 2017.
- [17] Trevor E. Carlson, Wim Heirman, Stijn Eyerma, Ibrahim Hur, and Lieven Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.
- [18] Andres S. Charif-Rubial, Emmanuel Oseret, José Noudouhouenou, William Jalby, and Ghislain Lartigue. Cqa: A code quality analyzer tool at binary level. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10, 2014.
- [19] Nicolas Derumigny, Théophile Bastian, Fabian Gruber, Guillaume Iooss, Christophe Guillon, Louis-Noël Pouchet, and Fabrice Rastello. Palmed: Throughput characterization for superscalar architectures. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization, CGO '22*, page 106–117. IEEE Press, 2022.
- [20] Lamia Djoudi, Denis Barthou, Patrick Carribault, Christophe Lemuet, Jean-Thomas Acquaviva, and William Jalby. Maqao: Modular assembler quality analyzer and optimizer for itanium 2. In *Workshop on Explicitly Parallel Instruction Computing Techniques*, Santa Jose, California, March 2005.
- [21] Jan Edler and Mark D. Hill. Dinero iv trace-driven uniprocessor cache simulator. <https://pages.cs.wisc.edu/markhill/DineroIV/>.
- [22] Hadi Esmailzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011.
- [23] Christophe Guillon. Dinero iv with plru replacement policy support. <https://github.com/atos-tools/dineroIV>.
- [24] Christophe Guillon. Program instrumentation with qemu. In *Proceedings of the International QEMU User's Forum*, 2011.
- [25] Georg Hager, Jan Treibig, Johannes Habich, and Gerhard Wellein. Exploring performance and power properties of modern multi-core chips via simple machine models. *Concurrency and Computation: Practice and Experience*, 28(2):189–210, feb 2016.
- [26] Julian Hammer, Jan Eitzinger, Georg Hager, and Gerhard Wellein. Kerncraft: A tool for analytic performance modeling of loop kernels. In Christoph Niethammer, José Gracia, Tobias Hilbrich, Andreas Knüpfer, Michael M. Resch, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2016*, pages 1–22, Cham, 2017. Springer International Publishing.
- [27] Johannes Hofmann, Georg Hager, and Dietmar Fey. On the accuracy and usefulness of analytic energy models for contemporary multicore processors. In Rio Yokota, Michèle Weiland, David Keyes, and Carsten Trinitis, editors, *High Performance Computing*, pages 22–43, Cham, 2018. Springer International Publishing.
- [28] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, Prashant Singh Rawat, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P. Sadayappan. Gpu code optimization using abstract kernel emulation and sensitivity analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018*, page 736–751, New York, NY, USA, 2018. Association for Computing Machinery.
- [29] Intel. Vtune profiler. <https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html>, 2011.
- [30] Engin İpek, Sally A. McKee, Rich Caruana, Bronis R. de Supinski, and Martin Schulz. Efficiently exploring architectural design spaces via predictive modeling. *SIGOPS Oper. Syst. Rev.*, 40(5):195–206, oct 2006.
- [31] Maurice G Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.
- [32] Andi Kleen. pmu tools, intel pmu profiling tools. <https://github.com/andikleen/pmu-tools>.
- [33] Souad Koliaï, Zakaria Bendifallah, Mathieu Tribalat, Cédric Valensi, Jean-Thomas Acquaviva, and William Jalby. Quantifying performance bottleneck cost through differential analysis. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS '13*, page 263–272, New York, NY, USA, 2013.

Association for Computing Machinery.

- [34] Jan Laukemann and Georg Hager. Core-level performance engineering with the open-source architecture code analyzer (osaca) and the compiler explorer. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering, ICPE '23 Companion*, page 127–131, New York, NY, USA, 2023. Association for Computing Machinery.
- [35] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikanth Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhui Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kantho, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Eder F. Zulian. The gem5 simulator: Version 20.0+, 2020.
- [36] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05*, page 190–200, New York, NY, USA, 2005. Association for Computing Machinery.
- [37] Paul E. McKenney. Differential profiling. In *Proceedings of the 3rd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS '95*, page 237–241, USA, 1995. IEEE Computer Society.
- [38] Charith Mendis, Saman P. Amarasinghe, and Michael Carbin. Ithelma: Accurate, portable and fast basic block throughput estimation using deep neural networks. *CoRR*, abs/1808.07412, 2018.
- [39] James George Mitchell. *The Design and Construction of Flexible and Efficient Interactive Programming Systems*. PhD thesis, USA, 1970. AAI7104538.
- [40] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation*. A dissertation submitted for the degree of doctor of philosophy, University of Cambridge, November 2004.
- [41] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, jun 2007.
- [42] PoCC, the polyhedral compiler collection. <https://www.cs.colostate.edu/~pouchet/software/poccc/>.
- [43] Louis-Noël Pouchet and Tomofumi Yuki. PolyBench/C: The polyhedral benchmark suite, version 4.2, 2016. <http://polybench.sf.net>.
- [44] Fabian Ritter and Sebastian Hack. Anica: Analyzing inconsistencies in microarchitectural code analyzers. *Proc. ACM Program. Lang.*, 6(OOP-SLA2), oct 2022.
- [45] Erven Rohou, Bharath Narasimha Swamy, and André Seznec. Branch prediction and the performance of interpreters — don't trust folklore. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 103–114, 2015.
- [46] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture, ISCA '13*, page 475–486, New York, NY, USA, 2013. Association for Computing Machinery.
- [47] Andreas Sandberg, Nikos Nikoleris, Trevor E. Carlson, Erik Hagersten, Stefanos Kaxiras, and David Black-Schaffer. Full speed ahead: Detailed architectural simulation at near-native speed. In *2015 IEEE International Symposium on Workload Characterization*, pages 183–192, 2015.
- [48] André Seznec. The l-tage branch predictor. *J. Instr. Level Parallelism*, 9, 2007.
- [49] André Seznec. A 64-kbytes ittage indirect branch predictor. 2011.
- [50] David Suggs, Mahesh Subramony, and Dan Bouvier. The amd “zen 2” processor. *IEEE Micro*, 40(2):45–52, 2020.
- [51] O. Sykora, P. Phothilimthana, C. Mendis, and A. Yazdanbakhsh. Granite: A graph neural network model for basic block throughput estimation. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*, pages 14–26, Los Alamitos, CA, USA, nov 2022. IEEE Computer Society.
- [52] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.
- [53] Vincent M Weaver, Dan Terpstra, and Shirley Moore. Non-determinism and overcount on modern hardware performance counter implementations. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 215–224. IEEE, 2013.
- [54] R.E. Wunderlich, T.F. Wensich, B. Falsafi, and J.C. Hoe. Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, pages 84–95, 2003.
- [55] Ahmad 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.
- [56] Hosein Yavarzadeh, Mohammadkazem Taram, Shravan Narayan, Deian Stefan, and Dean Tullsen. Half&Half: Demystifying Intel's directional branch predictors for fast, secure partitioned execution. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, May 2023.
- [57] Jifei Yi, Benchao Dong, Mingkai Dong, and Haibo Chen. On the precision of precise event based sampling. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems, APSys '20*, page 98–105, New York, NY, USA, 2020. Association for Computing Machinery.