WIP: A Second Look at Port Assignment on Intel CPUs

Yarin Oziel* Tomer Laor* Shlomi Levy* Clémentine Maurice† Yossi Oren*
Thomas Rokicki‡ Gabriel Scalosub*

1 Introduction

Modern Intel CPUs use a superscalar, out-of-order execution (OoOE) pipeline that maximizes instruction throughput through instruction-level parallelism (ILP). A critical component affecting system performance is the process in which µ-ops are mapped to execution ports, which lays at the core of the ability to perform OoOE. Significant effort has been undertaken in the attempt to reverse-engineer and model this process [3, 5, 6].

Our work revisits this question from a *security* perspective, focusing on potential vulnerabilities stemming from µ-op port assignment and execution. Using carefully designed code gadgets, we expose behaviors that contradict state-of-the-art models.

A key observation, which underlies our entire work, is that *all proposed models fail to capture significant aspects of µ-op port assignment and execution*. For most Intel architectures the µ-op port assignment policy exhibits significant irregularities and dynamics, that are inconsistent with state-of-the-art models.

In this Work In Progress, we ask: *Which undocumented behaviors in Intel's port assignment algorithm affect instruction scheduling?* By extending the micro-benchmarks used by Abel and Reineke, we identify corner cases that strongly deviates from the common case, where the CPU adopts at least three different port assignment strategies. These depend not only on the instruction and the microarchitectural generation, but, surprisingly, also on immediate operands.

To make these findings fully accessible, we release an interactive online database showcasing all our experiments and results: https://uops-again.info/. We envision this resource as a long-term reference point for researchers exploring Intel port assignment.

*Ben-Gurion University of the Negev
†Univ. Lille, CNRS, Inria
‡Univ Rennes, CNRS, CentraleSupélec, Inria

2 Background

The execution pipeline of modern Intel CPUs is broadly divided into two stages: (i) the *front end*, which is responsible for fetching and decoding instructions into micro-operations (µ-ops), and (ii) the *back end*, which handles the actual execution of µ-ops. System performance is critically influenced by the mapping of µ-ops to execution ports, a process that constitutes the foundation of effective out-of-order execution. Intel's official documentation confirms that its CPU has multiple execution ports, and describes the functional execution units found in each port, as well as common representative instructions related to each execution unit [4]. It does not, however, describe the port usage of all instructions, nor does it describe the exact strategy used by the renamer/allocator to assign µ-ops to ports, especially in the case where two or more ports with matching execution units are available. To close this gap, multiple researchers, most notably Abel and Reineke, published an extensive body of work focused on reverse engineering the Intel Core and understanding its internal behavior [1–3]. However, these models focus on common execution cases, and fail to capture corner cases that may be exploited in security contexts.

3 Experiments and Contributions

Our microbenchmarks consist of small code blocks comprising an `LFENCE` instruction, followed by a pair of two other instructions. For example, to analyze the "`LFENCE; STC; ADD R64, R64`" code block, we invok nanoBench [2] with the following command:[1]

```
sudo ./kernel-nanoBench.sh -no_norm -n_meas 1
  -warm_up_count 10
  -config configs/cfg_port_0156_only.txt
  -asm "LFENCE; STC; ADD RAX, RBX"
  -unroll 120
```

[1]We use a configuration file `cfg_port_0156_only.txt`, which only extracts usage counters of ports 0, 1, 5, and 6.

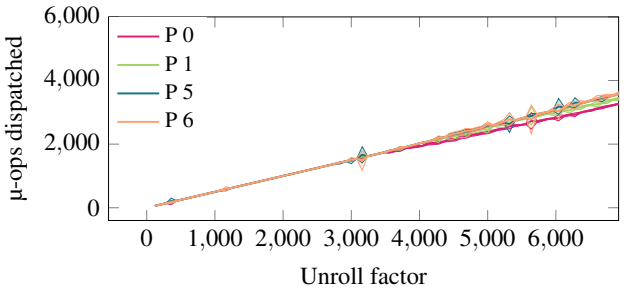

Figure 1: Common-case port assignment algorithm, for the "`CBW; CBW`" code block (without `LFENCE`).

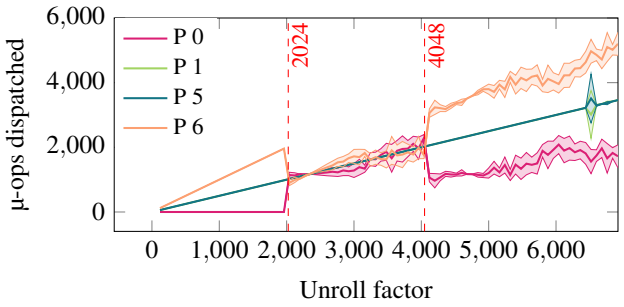

Figure 2: Three different port assignment regions, for the "`LFENCE; CBW; CBW`" code block.

We measure the µ-ops dispatched to each port across increasing unroll factors, with unroll values ranging from 100 to 6980, in increments of 20. Each unroll factor is repeated 10 times, and we take the mean and standard deviation of the µ-ops dispatched to each port.

Our experiments show that Intel's port assignment policies can diverge significantly from the well-documented "least-loaded eligible port" model, illustrated in Figure 1. Using carefully crafted two-instruction microbenchmarks preceded by an `LFENCE`, we consistently observed dynamic scheduling policies. Instead of a fixed distribution across eligible ports, the port assignment changes as the unroll factor increases, producing distinct regions separated by cutoffs. As illustrated in Figure 2 for the "`LFENCE; CBW; CBW`" snippet, the port scheduler employs three different strategies depending on the number of loop iterations. At lower unroll factors, one sparsest port is strongly preferred. After a first cutoff, the allocation becomes approximately uniform across all eligible ports, albeit noisy. At a second cutoff, the scheduler shifts again, favoring a different subset of ports. The second cutoff's unroll factor is twice the first's unroll factor. These dynamics are not isolated: we observed similar cutoff-based transitions across multiple instructions and instruction pairs, and in some cases, the behavior also depends on the *order of instructions* in the block or on *immediate values used in operands*. We believe that this might serve as a new microarchitectural attack surface which can be harnessed towards implementing, e.g., covert channels, fingerprinting, etc. Importantly, the observed cutoffs are consistent and reproducible across multiple runs, but differ between CPU generations. These findings show that static eligibility sets cannot fully describe port assignment. Instead, the allocator follows multiple hidden policies, switching between them in ways not accounted for by existing models.

4 Discussion

This work in progress uncovers undocumented irregularities in Intel's µ-op scheduling policies. Major open questions remain: the microarchitectural origin of the observed cutoff thresholds, and how widespread these corner cases are across real workloads and CPU generations. Our findings call for refined models to support better compilers, performance tools and benchmarks. From a security perspective, this line of research raises compelling offensive opportunities. Our observations motivate a focused search for *gadgets* in real victim code whose port-assignment behavior might change with secret-dependent data.

Acknowledgments. This research was supported by Israel Science Foundation grant 229/24. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

[1] Andreas Abel and Jan Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on Intel microarchitectures. In *ASPLOS*, 2019.

[2] Andreas Abel and Jan Reineke. nanoBench: A low-overhead tool for running microbenchmarks on x86 systems. In *ISPASS*, 2020.

[3] Andreas Abel and Jan Reineke. uiCA: Accurate throughput prediction of basic blocks on recent Intel microarchitectures. In *ICS*, 2022.

[4] Intel Corp. Intel® 64 and IA-32 architectures optimization reference manual, 2023. https://www.intel.com/content/www/us/en/content-details/671488/.

[5] J. Laukemann, J. Hammer, J. Hofmann, G. Hager, and G. Wellein. Automated instruction stream throughput prediction for Intel and AMD microarchitectures. In *PMBS*, pages 121–131, 2018.

[6] Fabian Ritter and Sebastian Hack. PMEvo: Portable inference of port mappings for out-of-order processors by evolutionary optimization. In *PLDI*, 2020.