# Explainable Port Mapping Inference with Sparse Performance Counters for AMD's Zen Architectures

Fabian Ritter
fabian.ritter@cs.uni-saarland.de
Saarland University, Saarland Informatics Campus
Saarbrücken, Germany

Sebastian Hack
hack@cs.uni-saarland.de
Saarland University, Saarland Informatics Campus
Saarbrücken, Germany

## Abstract

Performance models are instrumental for optimizing performance-sensitive code. When modeling the use of functional units of out-of-order x86-64 CPUs, data availability varies by the manufacturer: Instruction-to-port mappings for Intel's processors are available, whereas information for AMD's designs are lacking. The reason for this disparity is that standard techniques to infer exact port mappings require hardware performance counters that AMD does not provide.

In this work, we modify the port mapping inference algorithm of the widely used uops.info project to not rely on Intel's performance counters. The modifications are based on a formal port mapping model with a counter-example-guided algorithm powered by an SMT solver. We investigate in how far AMD's processors comply with this model and where unexpected performance characteristics prevent an accurate port mapping. Our results provide valuable insights for creators of CPU performance models as well as for software developers who want to achieve peak performance on recent AMD CPUs.

**ACM Reference Format:**
Fabian Ritter and Sebastian Hack. 2024. Explainable Port Mapping Inference with Sparse Performance Counters for AMD's Zen Architectures. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3620666.3651363

## 1 Introduction

When optimizing code for peak performance on a processor microarchitecture, understanding the architecture's performance characteristics is vital. Modern processor designs use many techniques to improve overall performance that cause complex, irregular performance characteristics. As manufacturers rarely provide performance models of their designs, we need to infer such models via microbenchmarks.

In this work, we consider models for an out-of-order processor's ability to exploit instruction-level parallelism: the port mapping. The port mapping describes how instructions are decomposed into smaller operations, so-called "micro-ops" or µops, and how these µops are executed on the processor's execution ports. Port mappings are important components in the cost models of compiler backends [19, 26] and of instruction throughput predictors [3, 4, 10, 25, 27, 30].

Inferring port mappings has been the subject of recent research: uops.info [1] provides accurate port mappings for Intel's microarchitectures with microbenchmarks witnessing each instruction's port usage. Their approach however does not cover other microarchitectures like AMD's recent Zen architectures, since uops.info relies on Intel's per-port hardware performance counters. Other recent port usage inference techniques have limitations that hinder their adoption in compilers and throughput predictors: PMEvo's [29] port mappings rarely coincide with the actual microarchitecture. In contrast to the uops.info algorithm, PMEvo's evolutionary algorithm cannot provide explanatory microbenchmarks to bolster confidence in the results. Palmed [14] infers conjunctive resource mappings with good performance prediction results, but they do not map directly to the microarchitecture and therefore do not fit into existing tools.

In this paper, we present a novel port-mapping inference algorithm that does not rely on per-port performance counters and therefore supports microarchitectures that uops.info cannot handle. Our algorithm instead uses throughput measurements and a single hardware performance counter to count the total number of executed µops for a given microbenchmark. Assuming that the processor behaves according to a formal port mapping model, these measurements are sufficient to infer port mappings with explanatory microbenchmarks similar to the uops.info algorithm. The core of our technique is a counter-example-guided algorithm powered by a satisfiability-modulo-theories (SMT) solver that directly leverages the formal model.

We evaluate our approach in a case study on AMD's Zen+ architecture. We report cases where irregular performance characteristics and undocumented or wrongly documented cases hinder an accurate performance model of Zen+. Using our algorithm, we infer the most comprehensive explainable

*ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA*
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0386-7/24/04
https://doi.org/10.1145/3620666.3651363

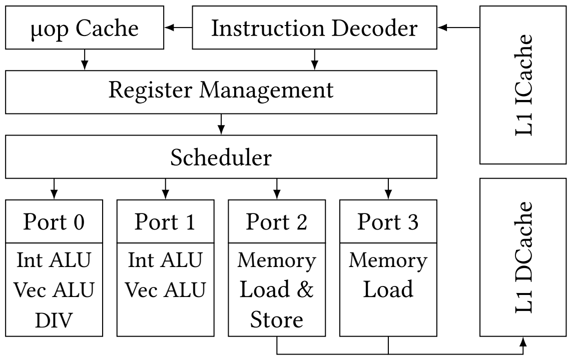


**Figure 1.** Simplified overview of a modern processor design (based on Figure 2-8 in the Intel Software Optimization Manual [22, Section 2.6]).

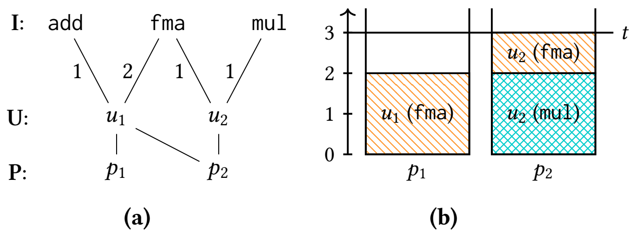


**Figure 2.** Example port mapping (a) and optimal µop distribution for [mul, mul, fma] (b). The processor executes two $u_1$ µops (for the fma instruction) and three $u_2$ µops (one for the fma instruction and one for each mul instruction) for this instruction sequence. Only port $p_2$ can handle $u_2$ while $u_1$ could be executed on either port.

port mapping for Zen+ to date. Our port mapping outperforms the state of the art in terms of throughput prediction accuracy for the supported instructions.

In summary, we provide the following contributions:

- An explainable inference algorithm for port mappings that does not require Intel's per-port performance counters and
- an implementation that we evaluate on AMD's Zen+ architecture, which was previously out of scope for explainable port mapping inference algorithms.
- The result is, to the best of our knowledge, the most comprehensive and accurate port mapping available for Zen+.
- Our case study documents numerous previously undocumented or misdocumented aspects of Zen+.

## 2 Background

### 2.1 Out-of-Order Microarchitectures

Modern microarchitectures are complex designs that combine various techniques to improve performance. Typically, they decode instructions into one or more µops and apply out-of-order execution: The µops are reordered and executed in parallel on the processor's functional units as long as their read-after-write dependencies are preserved [21, 33].

Figure 1 illustrates such a microarchitecture. Instructions are loaded from memory via the instruction cache. They are decoded into µops, which are cached for future re-use. Their register operands are translated to a larger set of microarchitectural registers. This translation eliminates "false" write-after-read and write-after-write dependencies that would otherwise limit reordering. Finally, a scheduler assigns each µop, once its operands are ready, to a port with the appropriate functional unit. Most functional units of modern processors are pipelined, allowing the ports to accept one µop per cycle. To understand how an instruction is executed, we need to know its µop decomposition and which ports can execute these µops. This information is the *port mapping*.

Processors often provide hardware performance counters: additional registers to count events in the processor. They gather statistics on how the processor executes code without affecting the execution itself. Which events can be counted depends on the microarchitecture, e.g., executed instructions and µops or cache misses. The port mapping inference algorithm of uops.info [1] depends on counting the µops that each individual port executes. Relevant microarchitectures, e.g., AMD's Zen family, do not provide these counters. Tools like nanoBench [2] and LIKWID [20, 34] can read performance counters and measure the throughput for microbenchmarks.

### 2.2 The Port Mapping Model

This work builds on a formal port mapping model introduced in previous work [1, 29]: Port mappings are tripartite graphs $M := (\mathbf{I} \uplus \mathbf{U} \uplus \mathbf{P}, F \uplus E)$ between a set $\mathbf{I}$ of instruction schemes, a set $\mathbf{U}$ of µops, and a set $\mathbf{P}$ of execution ports. Instruction schemes (or instruction forms) abstract sets of concrete instructions that differ in their operands. The x86-64 instruction set contains thousands of instruction schemes. E.g., the following scheme represents a 64-bit addition with two general purpose register operands:

```
add ⟨GPR[64]⟩, ⟨GPR[64]⟩
```

In a port mapping, the labeled edges $F \subseteq \mathbf{I} \times \mathbb{N} \times \mathbf{U}$ describe how many µops of a specific kind the instruction schemes require. The unlabeled edges $E \subseteq \mathbf{U} \times \mathbf{P}$ between µops and ports capture where each µop can be executed.

For an example, consider the port mapping in Figure 2a for the instruction schemes add, mul, and fma ("fused multiply and add") of a simplified microarchitecture. add and mul are each decomposed into a single µop: $u_1$ and $u_2$. $u_2$ has one execution port whereas $u_1$ can be executed on either $p_1$ or $p_2$. Executing the fma instruction requires processing two $u_1$ µops and one $u_2$ µop.

The port mapping model links a processor's port mapping and the throughput of dependency-free instruction sequences $e$ via the following linear program (LP):

$$\begin{aligned}
\text{min.}\quad & t \\
\text{s.t.}\quad & \sum_{k\in\mathbf{P}} x_{uk} = \sum_{(i,n,u)\in F} e(i)\cdot n && \text{for all μops } u \in \mathbf{U} && \text{(A)}\\
& \sum_{u\in\mathbf{U}} x_{uk} = p_k && \text{for all ports } k \in \mathbf{P} && \text{(B)}\\
& p_k \le t && \text{for all ports } k \in \mathbf{P} && \text{(C)}\\
& x_{uk} \ge 0 && \text{for all μops } u \in \mathbf{U}, \text{ ports } k \in \mathbf{P} && \text{(D)}\\
& x_{uk} = 0 && \text{if } (u,k) \notin E && \text{(E)}
\end{aligned}$$

The optimal objective value of this linear program is the inverse throughput $tp_M^{-1}(e)$ of $e$ with the port mapping $M$, i.e., the average number of processor cycles required for an instance of $e$ when it is executed indefinitely in a steady state.

Intuitively, each μop of the executed instructions contributes a share of "mass" corresponding to one cycle of utilization of a port. The non-negative real-valued $x_{uk}$ variables represent the mass contributed by $u$ μops that is executed on port $k$. $e(i)$ denotes how often the instruction scheme $i$ occurs in $e$. Constraint A therefore ensures that all mass is distributed to the $x_{uk}$ variables. Constraint B establishes real-valued $p_k$ variables for the total mass assigned to each port $k$ and constraint C introduces $t$ as an upper bound to the total masses of every port. Lastly, constraint E ensures that μops are only assigned to compatible ports. The requirement to minimize the upper bound $t$ guarantees that the μop mass is distributed as evenly as possible with the port mapping, achieving peak throughput. Solutions that assign non-integer μop masses to ports correspond to steady-state executions where μops go to different ports in different repetitions of the instruction sequence. This model assumes that the port mapping is the only source of throughput bottlenecks.

Figure 2b visualizes an optimal LP solution for the port mapping from Figure 2a and the instruction sequence [mul, mul, fma]. The mass for each port is collected in a corresponding bucket. The objective value $t$ is the mass of the highest bucket: three cycles. This mass is caused by the two $u_2$ μops of the mul instructions and the $u_2$ μop of the fma instruction. While the $u_1$ μops could also be executed on $p_2$, they need to be handled by port $p_1$ for an optimal distribution.

## 2.3 The uops.info Algorithm

Our work follows prior approaches [1, 29] with the goal to infer a processor's port mapping from microbenchmarks. We base our approach on a port mapping inference algorithm by Abel and Reineke [1, Section 5.1], which relies on blocking instructions: An instruction blocks a port subset $pu$ if it executes as a single μop that can use exactly the ports in $pu$.

The algorithm needs blocking instructions for the port sets of every μop. Abel and Reineke find blocking instructions by executing each instruction scheme in a steady state while counting the μops executed per port via performance counters. If there are as many μops as instructions, $i$ is a blocking instruction for the ports where its μops can be observed.

For an example, consider a 32-bit integer addition with register operands on the Intel Skylake microarchitecture:

`add ⟨GPR[32]⟩, ⟨GPR[32]⟩`

When we benchmark this instruction scheme in a steady state, Skylake's performance counters show that

1. an add instruction requires 0.25 cycles on average, i.e., four of them can execute in a single cycle,
2. the processor executes one μop per add instruction, and
3. the μops are executed in equal parts on ports 0, 1, 5, and 6 of the microarchitecture (which has a total of 8 ports).

This add instruction scheme therefore blocks the port set $\{0, 1, 5, 6\}$. Observation 3 requires per-port μop counters that are not available in AMD's processors.

```
Input: instruction under investigation $i$
1  $blkInsns \leftarrow$ pairs of (blocking insn, blocked ports),
        sorted by ascending number of blocked ports
2  $foundUops \leftarrow \{\}$
3  for $(B, pu) \in blkInsns$ do
4      $k \leftarrow$ # blocking insns $B$ sufficient to flood $pu$
5      $uops \leftarrow measureUopsOnPorts([\,k \times B, i\,], pu)$
6      $surplusUops \leftarrow uops - k$
7      for $pu', n \in foundUops$ do
8          if $pu' \subset pu$ then
9              $surplusUops \leftarrow surplusUops - n$
10     if $surplusUops > 0$ then
11         $foundUops[pu] \leftarrow surplusUops$
12 return $foundUops$
```

**Algorithm 1.** Port mapping inference for uops.info [1].

Abel and Reineke select one blocking instruction for each occurring port set and apply Algorithm 1 for each instruction scheme $i$. The multiset *foundUops* of μops, which are represented by their sets of admissible ports, for $i$ is filled throughout a run of the algorithm. The algorithm benchmarks $i$ with each blocking instruction $B$ for a set $pu$ of ports, starting with the smallest port sets and proceeding to increasing port set sizes. Each microbenchmark contains $i$ and enough copies of the considered blocking instruction $B$ such that any μop that *can* be executed on ports outside of $pu$ *is* executed on these alternative ports (ll. 4, 5). The number $k$ of blocking instruction copies must ensure that each blocked port in $pu$ receives at least as many μops as $i$ uses:

$$k \ge |pu| \cdot \mu opsOf(i) \tag{1}$$

Otherwise, ports in $pu$ might be unoccupied while µops of $i$ are issued, allowing µops of $i$ on $pu$ even though they could be executed on other ports.

When running this microbenchmark, Abel and Reineke count executed µops on ports in $pu$ via per-port performance counters. The result is the sum of $k$ and the number of µops of $i$ that only use ports in $pu$. These surplus µops include not only µops that use *all* ports of $pu$ but also those that have only a subset of $pu$ available. Because we assume a blocking instruction for the port set of each occurring µop and because the blocking instructions are sorted by ascending number of blocked ports, all µops for proper subsets of $pu$ were characterized in previous iterations of the loop. We can therefore subtract these previously characterized µops from the surplus µops (ll. 7–9) to obtain the µops of $i$ that can use any port in $pu$ (l. 11). The port usage of $i$ is fully characterized once every blocking instruction has been considered.

For example, consider a processor with the port mapping in Figure 2a. There are two blocking instructions: `mul` for the port set $\{p_2\}$ and `add` for $\{p_1, p_2\}$. For this example, to characterize the `fma` instruction, we use $k := |pu| \cdot \mu opsOf(i) = |pu| \cdot 3$ blocking instructions per benchmark.

The algorithm first benchmarks `fma` with blocking instructions for port sets of size 1, i.e., `mul`, with $k = 3$. Figure 3a shows the distribution of µops in a steady state execution: Four µops execute on the blocked port set $\{p_2\}$. Since no µops were characterized for smaller port sets, we conclude that `fma` uses $4 - k = 1$ µop that can be executed on $\{p_2\}$.

Next, `fma` is benchmarked with the `add` instruction, which blocks a port set of size 2 (i.e., $k = 6$). As shown in Figure 3b, we count nine µops on the ports $\{p_1, p_2\}$. Subtracting six blocking instructions, three surplus µops remain. One of these is explained by the $\{p_2\}$ µop found previously. The two remaining µops have the entire port set $\{p_1, p_2\}$ available. With no more blocking instructions remaining, we obtain the port usage for `fma`: $\{2 \times \{p_1, p_2\}, 1 \times \{p_2\}\}$

In practice, the uops.info implementation[1] computes the number $k$ of blocking instructions (l. 4) as follows:

$$k \leftarrow \min\big(100, \max\big(10, |pu| \cdot \mu opsOf(i),\\ 2 \cdot |pu| \cdot \max(1, \lfloor tp^{-1}([\,i\,]) \rfloor)\big)\big)$$

The resulting $k \in [10, 100]$ depends on the number $|pu|$ of blocked ports, the cycles $tp^{-1}([\,i\,])$ required to execute $i$ in a steady state, and $i$'s number $\mu opsOf(i)$ of µops. This term fulfills constraint 1 for reasonable numbers of µops. Compared to an implementation that satifisies constraint 1 tightly, we expect more resilience against measurement errors from the larger number of blocking instructions.

A key benefit of this port mapping inference algorithm is that the performed microbenchmarks serve as witnesses for the result: For each instruction $i$ and each port set $pu$, there is an experiment explaining if $i$ uses a µop for $pu$.

[1] https://github.com/andreas-abel/nanoBench/blob/faf75236cade57f7927f9ee949ebc679fc7864b7/tools/cpuBench/cpuBench.py#L3393

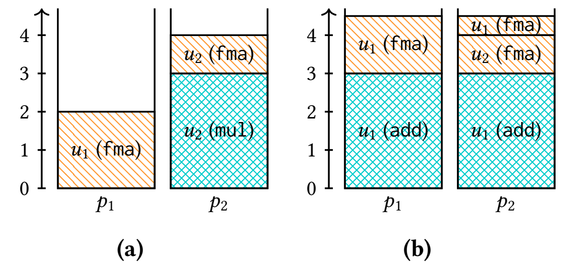


**Figure 3.** Possible steady-state distributions of µops per port in benchmarks of `fma` with (a) 3 `mul` and (b) 6 `add` blocking instructions, using the port mapping from Figure 2a.

## 3 Inferring Port Mappings Without Per-Port µop Counters

For AMD's Zen microarchitectures and several ARM designs, the uops.info algorithm is not applicable as they lack performance counters for executed µops per port. The key insight of our work is that the problems requiring per-port µop counters in the uops.info algorithm can be solved without them if we assume that the processor follows the port mapping model for some (unknown) port mapping. In the following, we provide alternatives that do not use per-port µop counters for the relevant parts of the uops.info algorithm. The only performance counter used, besides time measurements, counts the total number of µops executed for a microbenchmark.

### 3.1 Counting µops that Cannot Avoid Blocked Ports

To determine how many µops run on ports that are flooded with blocking instructions, consider two experiments:

- $e := [\,k \times B, i\,]$ consist of an instruction $i$ under investigation and $k$ blocking instructions $B$ for a port set $pu$.
- $e' := [\,k \times B\,]$ contains only the blocking instructions.

If all µops of $i$ can use unblocked ports, $tp^{-1}(e) = tp^{-1}(e')$ holds. Otherwise, each µop of $i$ that needs a port in $pu$ utilizes a flooded port for one cycle per iteration. Each such µop therefore adds $1/|pu|$ to the observed inverse throughput, i.e.:

$$tp^{-1}(e) = tp^{-1}(e') + 1/|pu| \cdot \text{µops of } i \text{ executed on } pu$$
$$\Leftrightarrow \text{µops of } i \text{ executed on } pu = \big(tp^{-1}(e) - tp^{-1}(e')\big) \cdot |pu|$$

For example, reconsider Figure 3a. The $u_2$ µop of the `fma` instruction cannot evade from the blocked port $p_2$. Compared to an execution of only the blocking instructions (3 cycles), the inverse throughput is increased by $1/|pu| = 1$ cycle.

### 3.2 Identifying Unique Blocking Instructions

We find and characterize blocking instructions as follows:

1. Count the µops when executing each instruction individually. Each instruction with only a single µop is a candidate.
2. Determine for each candidate $i$ the number of ports on which its µop can be executed, i.e., the number of instances of $i$ that can be executed per cycle. We measure this as the (non-inverse) throughput of $i$: $1/tp^{-1}([\,i\,])$.
3. Filter redundant candidates, leaving one blocking instruction per port set. Two candidates cannot be redundant if their port sets have different sizes. For two candidates $i$ and $j$ with equally sized port sets, we check for redundancy by measuring if their inverse throughputs are additive: The port sets of $i$ and $j$ are equal if
$$tp^{-1}([\,i, j\,]) = tp^{-1}([\,i\,]) + tp^{-1}([\,j\,])$$
4. Infer the port mapping of the remaining blocking instructions. Compared to the full inference problem, this concerns only few instruction schemes – e.g., the uops.info port mapping for Intel's Skylake has 12 distinct port sets that require blocking instructions – and every scheme uses only a single µop with a known number of ports. This makes the computationally expensive algorithm described in the following section practical.

## 3.3 Counter-Example Guided Port Mapping Inference

The port mapping model relates the throughput achieved for given instruction sequences with the processor's port mapping. We exploit this relation with inspiration from counter-example-guided abstraction refinement [12] and counter-example-guided inductive synthesis [31].

```
1 $Exps \leftarrow \{\}$
2 while true do
3     $m1 \leftarrow findMapping(Exps)$
4     if $m1$ = None then return None
5     $m2, newExp \leftarrow findOtherMapping(Exps, m1)$
6     if $m2$ = None then return m1
7     $cycles \leftarrow measureCycles(newExp)$
8     $Exps \leftarrow Exps \cup \{(newExp, cycles)\}$
```

**Algorithm 2.** Counter-example-guided inference.

Algorithm 2 shows the high-level structure of our counter-example-guided port mapping inference algorithm. It is centered around a set *Exps* of microbenchmarks annotated with the inverse throughput measured on the processor under investigation. In each iteration, we search a port mapping $m1$ that leads to the measured inverse throughputs in *Exps* (l. 3). If no mapping is found, the observations do not match the model: the algorithm terminates unsuccessfully (l. 4). Otherwise, we search for a different port mapping $m2$ that is also consistent with the measurements in *Exps*, but that is distinguished from $m1$ by an experiment *newExp* (l. 5). This means that $m1$ and $m2$ yield the same throughputs for *Exps*, but different throughputs for *newExp*. If no such mapping and experiment exist, $m1$ is indistinguishable by throughput measurements from the processor's actual port mapping (l. 6). Otherwise, we measure the inverse throughput of *newExp*, add it to *Exps* (ll. 7-8), and continue with the next iteration.

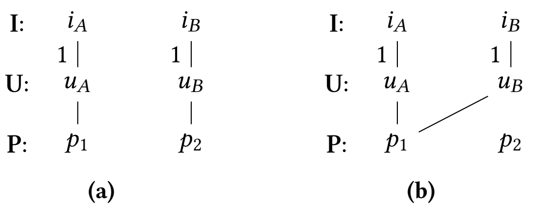


**Figure 4.** Port mappings that satisfy $\{([i_A], 1.0), ([i_B], 1.0)\}$.

For example, assume an architecture with two instructions $i_A, i_B$, each with a single µop, and two ports $p_1, p_2$. So far, we have the following measurements:
$$Exps = \{([i_A], 1.0), ([i_B], 1.0)\}$$
*findMapping*(*Exps*) might find Figure 4a as port mapping $m1$. However, this is not the only viable port mapping for these measurements: *findOtherMapping*(*Exps*, $m1$) returns a port mapping $m2$, e.g., Figure 4b. A distinguishing experiment *newExp* is $[i_A, i_B]$: With $m1$, its inverse throughput is 1.0 cycles per experiment execution while it is 2.0 cycles for $m2$.

The algorithm is guaranteed to terminate as at least one found port mapping is rejected in each iteration: No two *findOtherMapping* calls in a run can yield the same port mapping. Since there are only finitely many port mappings with only one µop per instruction, the algorithm only takes a finite number of steps. In practice, experiments usually rule out more than just a single candidate port mapping.

In the following, we derive SMT-solver-powered implementations of *findMapping* and *findOtherMapping* from the linear program in Section 2.2. The idea is to augment the LP such that the port mapping is no longer coded into the constraints, but rather represented in LP variables. We can then encode *findMapping* and *findOtherMapping* as constraints on the LP variables. An off-the-shelf solver can produce a satisfying model for the variables, from which we decode the result. Specifically, we formulate a constraint system
$$relateThroughput[enc_M, enc_e, enc_t]$$
parametrized with collections $enc_M$, $enc_e$, $enc_t$ of variables that represent a port mapping, an experiment, and the experiment's inverse throughput. We design these constraints such that a model encoding a port mapping $M$, an experiment $e$, and a number $t$ satisfies them if and only if $tp_M^{-1}(e) = t$.

**3.3.1 Encoding port mappings and experiments.** Experiments are effectively multisets of instruction schemes since their order is irrelevant in the port mapping model. We represent an experiment as a set $enc_e := \{exp[i] \mid i \in \mathbf{I}\}$ of integer-valued variables. The value of a variable $exp[i]$ is the number of occurrences of the instruction $i$ in the experiment. We constrain these variables to be non-negative.

To represent port mappings, we need variables that encode tripartite graphs between instruction schemes $\mathbf{I}$, ports $\mathbf{P}$, and an unknown set $\mathbf{U}$ of µops. Since every considered instruction scheme uses only a single µop, we chose $\mathbf{U}$ such that every instruction $i$ has its own µop $u^{(i)}$.[2] A set $enc_M := \{m[u^{(i)}, k] \mid i \in \mathbf{I}, k \in \mathbf{P}\}$ of boolean variables therefore encodes a port mapping. When a variable $m[u^{(i)}, k]$ is *True* in a model, edges connect instruction $i$ via µop $u^{(i)}$ to port $k$ in the port mapping, i.e., the µop of $i$ can be executed on $k$. E.g., the port mapping from Figure 4b corresponds to a model where $m[u^{(i_A)}, p_1]$ and $m[u^{(i_B)}, p_1]$ are set to *True* while $m[u^{(i_A)}, p_2]$ and $m[u^{(i_B)}, p_2]$ are set to *False*.

We add constraints so that each µop's number of ports fits the previous throughput measurements.

**3.3.2 Relating port mapping and throughput parametrically.** We adjust the linear program from Section 2.2 so that the inverse throughput $t$, the experiment $e$, and the port mapping $M$ occur only as free variables in the program.

***Inverse Throughput.*** The inverse throughput is present as a variable $t$ in the LP, but it is not free: Constraints A-E of the LP only assert that the experiment can be executed according to the port mapping within at most $t$ cycles and the minimization objective ensures that $t$ corresponds to an optimal µop distribution. If the value of $t$ was fixed by a constraint, the minimization objective would effectively be disabled. We therefore replace the optimization objective of the LP with more constraints that ensure optimality of the execution. We use SMT formulas in the theory of linear integer and real arithmetic (LIRA) to obtain a more intuitive formulation with logical disjunctions and implications:[3]

$$\bigvee_{k \in \mathbf{P}} q_k \tag{F}$$

$$q_k \leftrightarrow (p_k = t) \qquad \text{for all ports } k \in \mathbf{P} \tag{G}$$

$$j_u \rightarrow q_k \qquad \text{if } (u,k) \in E \tag{H}$$

$$\sum_{u \in \mathbf{U}} j_u \cdot \sum_{(i,n,u) \in F} e(i) \cdot n = \sum_{k \in \mathbf{P}} q_k \cdot t \tag{I}$$

These constraints are based on a result by Ritter and Hack [29, Section 4.5]: A distribution to ports is optimal if and only if there is a non-empty set $Q$ of bottleneck ports that are all utilized for the full number of cycles with µops that can only be executed on ports in $Q$. In the formulas, a port $k$ is in the set $Q$ of bottleneck ports iff the boolean variable $q_k$ is *True*. Constraint F asserts that $Q$ is not empty. With constraint G, we ensure that each bottleneck port is utilized for the full $t$ cycles. The boolean $j_u$ variables encode a set $J$ of µops $u$ that can only be executed on bottleneck ports in $Q$, as enforced by constraint H. Lastly, constraint I ensures that only µops from $J$ contribute to the utilization of the ports in $Q$.

***Experiment and Port Mapping.*** To introduce the experiment encoding, we use the $exp[i]$ variables instead of the fixed numbers $e(i)$ of occurrences for each instruction $i$. We replace the constraints E and I to integrate the port mapping encoding into the constraints with logical implications:

$$x_{uk} = 0 \quad \text{if } (u,k) \notin E \tag{$E$}$$
$$\rightsquigarrow \quad \neg m[u,k] \rightarrow x_{uk} = 0 \quad \text{for all } u \in \mathbf{U}, k \in \mathbf{P}$$

$$j_u \rightarrow q_k \quad \text{if } (u,k) \in E \tag{$I$}$$
$$\rightsquigarrow \quad m[u,k] \rightarrow \ (j_u \rightarrow q_k) \quad \text{for all } u \in \mathbf{U}, k \in \mathbf{P}$$

The resulting constraints form *relateThroughput*, with inverse throughput, experiment, and port mapping as free variables.

**3.3.3 *findMapping* and *findOtherMapping*.** The parametric $relateThroughput[enc_M, enc_e, enc_t]$ constraints make the functions from Algorithm 2 straightforward to implement:

*findMapping*(*Exps*) uses a single free port mapping encoding $M_{free}$. For each experiment $e$ with inverse throughput $t_e$, we assert *relateThroughput* constraints for $M_{free}$ and fresh experiment and throughput encodings that are hardwired to $e$ and $t_e$, respectively:

$$\varphi_{findMapping} := \bigwedge_{(e,t_e) \in Exps} relateThroughput[M_{free}, e, t_e]$$

The resulting conjunction ensures that the port mapping encoded in a satisfying model yields the observed inverse throughput for all experiments. We use an off-the-shelf SMT solver to check for satisfiability. If the constraints are unsatisfiable, the observed throughputs cannot be explained in the model and we find no mapping. Otherwise, we extract and return the port mapping from the values of the encoding variables in the satisfying model produced by the solver.

*findOtherMapping*$(M_1, Exps)$ also uses these constraints to require that the found port mapping satisfies the experiments, and adds more: Another mapping encoding is hardwired to the input port mapping $M_1$. For a free experiment encoding $e_{free}$, we use two more instances of the *relateThroughput* constraints to encode the inverse throughputs of both port mapping encodings in two variables $t_1, t_2$. Lastly, we assert that $t_1 \neq t_2$, i.e., the experiment distinguishes the hard-wired and the free port mapping:

$$\varphi_{findOther} := t_1 \neq t_2 \wedge relateThroughput[M_1, e_{free}, t_1]$$
$$\wedge \ relateThroughput[M_{free}, e_{free}, t_2] \wedge \varphi_{findMapping}$$

**3.3.4 Addressing Benchmarking Limitations.** When benchmarking modern processors, inexact measurements due to noise and errors are inevitable. We therefore adapt

[2]We can relax this requirement of the algorithm to arbitrary but fixed numbers of µops by introducing more µops.
[3]The multiplications in constraint I all involve either a boolean variable or a constant and are therefore encodable without undecidable theories.

the constraints: A parameter $\varepsilon$ constrains the maximal difference between measured and modeled cycles per instruction (CPI) of the experiments.[4] The following constraint encodes equality of the measured and modeled inverse throughputs $t_e$ and $enc_t$:

$$|enc_t - t_e| < \varepsilon \cdot |exp|$$

When asserting that the modeled inverse throughputs of the two port mappings in *findOtherMapping* are different, no observed value may be considered equal to both modeled inverse throughputs. Otherwise a found experiment might not rule out any of the candidate mappings. This can be guaranteed if the modeled CPIs differ by at least $2 \cdot \varepsilon$:

$$|t_1 - t_2| > 2 \cdot \varepsilon \cdot |exp|$$

Moreover, *findOtherMapping* can produce excessively large microbenchmarks that hit bottlenecks outside the port mapping, e.g., the cache boundaries. We therefore follow a stratified approach: First, we only search distinguishing experiments with a single instruction and increase this bound once no more experiments are found. If increasing the bound yields no experiments, we run *findOtherMapping* without bound. If this produces no experiment either, the algorithm terminates; otherwise it continues with a larger bound. As a result, the benchmark experiments have minimal size without sacrificing the completeness of the algorithm.

## 3.4 Handling Pipeline Bottlenecks

The port mapping model assumes that throughput is only limited by the availability of functional units. In practice, that is not the case. All modern processors that we are aware of, including recent designs by Intel and AMD, cannot sustain a full utilization of all ports. The culprit is often the decoding frontend (including the caches) or the instruction retirement rate. When a bottleneck limits the execution to at most $R_{max}$ instructions per cycle, experiments that are faster according to the port mapping model are slowed to meet the limit.

Such bottlenecks can affect the correctness of our algorithm. The checks for equivalence of blocking instructions (Section 3.2) and for evading µops (Section 3.1) measure if an experiment utilizes more than a certain number $n$ of ports. These checks fail if there is no gap between $R_{max}$ and the largest such $n$, i.e., the maximal port set size of any µop. We need to check this requirement when applying the algorithm.

The counter-example-guided inference algorithm in Section 3.3 needs an adjustment for such bottlenecks: We change *relateThroughput*$[enc_M, enc_e, enc_t]$ such that $enc_t$ is the maximum of the number $tp_M^{-1}(e)$ of cycles according to the model and the peak inverse throughput $|e|/R_{max}$ at the bottleneck. Some theoretically distinguishable port mappings become indistinguishable with this adjustment.

[4]i.e., inverse throughput divided by the length of the experiment.

## 3.5 Supported Microarchitectures

Our algorithm has the following requirements:

- We need to measure the number of cycles required to execute a piece of code. Such functionality is commonplace in contemporary Intel, AMD, and ARM microarchitectures.
- There needs to be a counter for the total number of µops executed for a piece of code. Recent Intel Core architectures support this, and AMD's Zen, Zen+, and Zen2 are documented to support this as well.[5]
- The processor's throughput bottleneck should not be hit when executing only instructions of the same kind. AMD's Zen-family microarchitectures (up to Zen4) satisfy this requirement [15], most Intel Core architectures violate it.

According to available documentation, examples of contemporary microarchitectures as of 2024 that satisfy these requirements include:

- AMD's Zen, Zen+, and Zen2 microarchitectures: They sustain a throughput of 5 instructions per cycle (IPC) [15, Chapter 22] and individual µops have up to 4 ports available. The "Retired Uops" counter [6, Section 2.1.15.4.5] is documented to count µops.[6]
- Intel's Golden Cove microarchitecture, which is used in their Sapphire Rapids and Alder Lake processors: It sustains 6 IPC [15, Section 13.1] and its µops have up to 5 ports available [1]. Golden Cove has a `UOPS_EXECUTED.THREAD` performance counter [23].
- Fujitsu's A64FX microarchitecture: Its decoder can issue 4 instructions per cycle and µops can use up to 3 ports [17, Chapter 2]. This architecture provides a `UOP_SPEC` performance counter [18, p. 16].
- ARM's Neoverse V2: Its decoder sustains a throughput of 8 IPC while µops can use up to 6 ports [8, Section 2.1]. Its `OP_RETIRED` counter [9, Table 18-1] counts executed µops.
- Apple's M1: According to results by Dougall Johnson [24], its performance core sustains 8 IPC and each µop has up to 6 ports available. Johnson uses an an undocumented performance counter to count µops.

Additionally, as in the original uops.info algorithm, we assume that there is a blocking instruction for most µops of the microarchitecture. Where this requirement is not met, replacement instructions with a throughput dominated by the respective µop need to be specified manually. In the following case study, we show that, contrary to official documentation, such µops occur in AMD's Zen+ microarchitecture.

## 4 Case Study: The AMD Zen+ Architecture

We evaluate our port mapping inference algorithm with the AMD Zen+ microarchitecture. This allows us to use

[5]See Section 4.1 for more on this and subsequent Zen microarchitectures.

[6]Our experiments indicate that this performance counter behaves like the "Retired Ops" counter of Zen3 and Zen4 [7, Section 2.1.15.4.5]. With a throughput of 6 IPC and up to 4 ports per µop [15, Chapters 23–24], Zen3 and Zen4 could therefore be handled similarly as the previous Zen generations.

PMEvo [29] and Palmed [14] as points of comparison in Section 4.5. Since Zen+ does not have full per-port μop counters, the original uops.info algorithm is not applicable. We also compare our results to the available documentation for Zen+:

- AMD's Software Optimization Guide (SOG) [5] describes the microarchitecture and documents instruction latencies and throughputs, if they are microcoded, and, for simple instructions, their execution units.
- Agner Fog's microarchitecture guide [15] analyzes the similar Zen architecture based on manual microbenchmarks.
- Fog [16] and uops.info [1] provide tables with measured latencies, throughputs, and numbers of μops of individual instructions. They include the port usage of floating point (FP)/vector instructions since per-port performance counters for these units are available [6, Section 2.1.15.4.1].
- WikiChip collects information on Zen and Zen+, in part from AMD's marketing resources [35, 36].

Our test system has an AMD Ryzen 5 2600X processor and 32 GB of RAM. It runs the port mapping inference algorithm and automatically performs microbenchmarks when required. Simultaneous multi-threading and frequency scaling are disabled. We measure inverse throughput and retired μops with a technique similar to nanoBench [2], taking the median over 11 repeated microbenchmark runs.[7] We consider two throughput measurements equal if the implied cycles per instruction (CPI) differ by at most $\varepsilon = 0.02$. This value allows us to distinguish if experiments use five ports (0.20 CPI) or four ports (0.25 CPI). Zen+ meets our bottleneck requirement: Five blocking instructions can be executed per cycle [15, Section 22.21] and μops have at most four ports.

We take the x86-64 instruction schemes from uops.info and remove control flow and system instructions as well as instructions with known input-dependent performance characteristics. For FP and vector operations, we only consider instructions from the AVX and AVX2 instruction set extensions. This gives us 2,980 instruction schemes. We further reduce this set of instruction schemes when the need arises throughout the stages of the algorithm.

## 4.1 Identifying Blocking Instruction Candidates

The first algorithm stage benchmarks every single instruction under investigation individually. Instructions that are executed with a single μop are blocking instructions.

**4.1.1 Counting μops.** Compared to AMD's documentation, we measure unexpected numbers of μops, e.g., for this instruction scheme:

```
add ⟨GPR[32]⟩, ⟨MEM[32]⟩
```

It loads a value from memory, adds it to the value of a register, and writes the result into the same register. According to AMD's SOG [5, Table 1], this instruction uses two μops: one that loads and one that adds. However, the "Retired Uops" counter only increases by one. We observe the same for any instruction with memory operands. uops.info and Fog's tables, which also rely on this performance counter, agree with our observations. While our inquiry with AMD's support remains unanswered, there is evidence that the "Retired Uops" performance counter, `PMCx0C1`, counts *macro-ops* instead of μops: The observed values are consistent with the SOG's macro-op numbers and AMD's documentation for the more recent Zen 3 and 4 microarchitectures [7, Section 2.1.15.4.5] documents this identifier for counting macro-ops.

AMD's macro-ops are a representation between x86-64 instructions and the μops that are executed by the execution units [5, Section 2.3]. Many instructions are implemented with a single macro-op, whereas, e.g., 256-bit-wide vector operations use two narrower macro-ops. Complex instructions are microcoded with a greater number of macro-ops.

To the best of our knowledge, there is no detailed published information on how macro-ops are decomposed into μops and no suitable performance counter for experimental characterization. Since our algorithm requires a count of μops, we postulate a macro-op-to-μop correspondence in the Zen+ microarchitecture, based on AMD's SOG [5, Section 2.3]: Let $n$ be the number of macro-ops observed when executing a basic block $bb$. We obtain the μop count by adding

- 1 for each memory operand with a width of at most 128 bits (excluding "load effective address" and loading `mov`s),
- 2 for each memory operand with a width of 256 bits (as they are implemented as two 128-bit operations).

For one case, we deviate from the SOG: It claims that `mov`s which *store* to memory do not require an additional μop. This contradicts our observations:

- A store-`mov` together with four simple register-additions takes 1.25 cycles. Therefore, it has a μop that is restricted to the four ALU ports.
- A `vmovapd` vector-register-to-memory store (documented with a store μop and one to deliver the stored data) together with the four additions takes only 1.0 cycles. Hence, no μops of this instruction are restricted to the ALU ports.
- A storing `mov` with a storing `vmovapd` leads to an inverse throughput of 2 cycles. These instructions therefore interfere, i.e., a μop of the `mov` instruction uses a port that the `vmovapd` instruction also needs.

Hence, the storing `mov` instruction has a μop that is restricted to one or more ALU ports and one for a non-ALU port. Therefore, similar to Intel architectures [1, Section 5.1.1], there is no proper blocking instruction for memory store μops.

**4.1.2 Problematic Instructions.** For several instruction schemes, we observe breaks in the algorithm's assumptions:

- `nop`s and 32 or 64-bit-wide register-to-register `mov`s use no ports: The processor resolves such `mov`s via register renaming [15, Section 22.13] and implements `nop`s without μops. No port mapping is necessary for these cases.

[7] We use the `PMCx0C1` ("Retired Uops") counter [6, Section 2.1.15.4.5].

- Some FP instructions execute slower than the port mapping model permits, e.g., divisions, square-root computations and approximate reciprocals.
- A mov of a 64-bit immediate into a GPR causes unreliable measurements. As these constants are unusual in the ISA, they use special handling in the hardware [5, Section 2.9].
- We cannot measure instructions that modify operands that are hardwired or restricted to ah–dh registers without observing effects of data dependencies.

We exclude all these instruction schemes, leaving 2,323 remaining schemes. Of these, 691 are identified as blocking instruction candidates.

## 4.2 Filtering Equivalent Blocking Instructions

Next, we run microbenchmarks for pairs of blocking instruction candidates with equally-sized port sets to check if they are equivalent. We encounter further problematic instructions in these experiments: Conditional move instructions, AES de/encryption operations, numerical conversions of the vcvt* family, and double-precision FP multiplications cause unstable measurements when benchmarked with other instructions. FP/vector operations with three read operands, like fused multiply-and-add instructions and some vector blending operations, do not fit the port mapping model either in Zen+. While these operations can execute on two of the four ports of the FP unit, they use data lines of a third port [5, Section 2.11]. This third port meanwhile has to idle, which we observe as contradicting equivalence information. We exclude these instructions from the following steps.

This leaves us with 1,887 instruction schemes in total, with 563 blocking instruction candidates. Of these candidates, 13 are identified as unique blocking instructions. Table 1 shows them with the number of candidates per equivalence class.

This selection is consistent with uops.info: If we found two candidates equivalent and if they are covered by uops.info, then their reported port usages are equal. uops.info does not cover 266 of our 563 candidates. AMD's tables do not agree for 33 instruction schemes. E.g., they document the following xmm vector comparisons with the same two ports:

vpcmp**gtq** vpcmp**eqq** vpcmp**gtb**

In our measurements, only the second (testing equality for $2 \times 64$-bit integers) has two ports, whereas the first and third have one and three ports available (greater-than tests for $2 \times 64$-bit and $16 \times 8$-bit integers, respectively). Fog's table and uops.info agree with our measurements; this appears to be an error in AMD's documentation.

## 4.3 Computing a Mapping for the Blocking Instructions

Here, we compute a port mapping for the blocking instructions with the counter-example-guided inference algorithm. We use z3 [13] (V.4.12.1) as SMT solver and select $\varepsilon = 0.02$. Following the SOG [5], we use a set of 10 ports.

**Table 1.** Identified blocking instruction classes for AMD Zen+. Representants were selected manually for clarity.

| # Ports | Instruction Scheme | # Equiv. | Description |
|---|---|---|---|
| 4 | add ⟨GPR[32]⟩, ⟨GPR[32]⟩ | 242 | ALU ops |
| | vpor ⟨XMM⟩, ⟨XMM⟩, ⟨XMM⟩ | 21 | logical vector ops |
| 3 | vpaddd ⟨XMM⟩, ⟨XMM⟩, ⟨XMM⟩ | 30 | vector int. arith. |
| 2 | vminps ⟨XMM⟩, ⟨XMM⟩, ⟨XMM⟩ | 143 | FP compare, mul. |
| | vbroadcastss ⟨XMM⟩, ⟨XMM⟩ | 50 | vector layouting |
| | vpaddsw ⟨XMM⟩, ⟨XMM⟩, ⟨XMM⟩ | 17 | saturating vec. ops |
| | vaddps ⟨XMM⟩, ⟨XMM⟩, ⟨XMM⟩ | 10 | FP additions |
| | mov ⟨GPR[32]⟩, ⟨MEM[32]⟩ | 6 | memory loads |
| 1 | vpslld ⟨XMM⟩, ⟨XMM⟩, ⟨XMM⟩ | 27 | vector shifts |
| | vpmuldq ⟨XMM⟩, ⟨XMM⟩, ⟨XMM⟩ | 10 | elaborate vec. mul. |
| | imul ⟨GPR[32]⟩, ⟨GPR[32]⟩ | 9 | integer mul. |
| | vroundps ⟨XMM⟩, ⟨XMM⟩, ⟨IMM[8]⟩ | 4 | vector rounding |
| | vmovd ⟨XMM⟩, ⟨GPR[32]⟩ | 2 | vector-to-GPR mov |

As there are no proper blocking instructions for store operations, we add "improper" blocking instructions manually:

- mov ⟨MEM[32]⟩, ⟨GPR[32]⟩, which stores a 32-bit value from a general purpose register into memory, and
- vmovapd ⟨MEM[128]⟩, ⟨XMM⟩, which stores a 128-bit value from a vector register into memory.

While we expect to use only the mov instruction in place of a blocking instruction for the store μop, both are required to infer that the store μop does not use an ALU instruction, cf. Section 4.1. We augment the SMT formulas such that the improper blocking instructions use exactly two μops and one of their μops is equal to one with a proper blocking instruction. These constraints avoid prohibitively long execution times.

For three blocking instructions, the generated experiments exhibit throughputs outside of the port mapping model:

- The imul scheme for scalar integer multiplications, e.g., when combined with four additions:

  4× add ⟨GPR[32]⟩, ⟨GPR[32]⟩
  1× imul ⟨GPR[32]⟩, ⟨GPR[32]⟩

  Since add has four ports and imul is restricted to one, two inverse throughputs are possible in the port mapping model: 1.25 cycles, if imul uses a port of the add instruction or 1.0 cycles, if their ports are disjoint. While AMD's SOG [5, Section 2.10.2] indicates the former, we measure ca. 1.5 cycles for this experiment, matching neither case.
- vpmuldq, which represents uncommon vector multiplication operations,[8] leads to experiments that run slower than their port usage would imply. This deviation from the modeled throughputs would require a larger $\varepsilon$, reducing the accuracy for other instructions.

[8]This specific instruction multiplies the 32-bit integers at even-numbered lanes in the source registers without overflows into a vector of 64-bit integers.

**Table 2.** Documented and inferred port usage of the blocking instructions for Zen+. Inferred ports were renamed to ease comparison.

| Instruction Scheme | Doc. Ports | Inferred Ports |
|---|---|---|
| `add ⟨GPR[32]⟩, ⟨GPR[32]⟩` | ALU | [6,7,8,9] |
| `vpor ⟨XMM⟩, ⟨XMM⟩, ⟨XMM⟩` | FP 0,1,2,3 | [0,1,2,3] |
| `vpaddd ⟨XMM⟩, ⟨XMM⟩, ⟨XMM⟩` | FP 0,1,3 | [0,1,3] |
| `vminps ⟨XMM⟩, ⟨XMM⟩, ⟨XMM⟩` | FP 0,1 | [0,1] |
| `vbroadcastss ⟨XMM⟩, ⟨XMM⟩` | FP 1,2 | [1,2] |
| `vpaddsw ⟨XMM⟩, ⟨XMM⟩, ⟨XMM⟩` | FP 0,3 | [0,3] |
| `vaddps ⟨XMM⟩, ⟨XMM⟩, ⟨XMM⟩` | FP 2,3 | [2,3] |
| `mov ⟨GPR[32]⟩, ⟨MEM[32]⟩` | AGU | [4,5] |
| `vpslld ⟨XMM⟩, ⟨XMM⟩, ⟨XMM⟩` | FP 2 | [2] |
| `vroundps ⟨XMM⟩, ⟨XMM⟩, ⟨IMM[8]⟩` | FP 3 | [3] |
| `mov ⟨MEM[32]⟩, ⟨GPR[32]⟩` | AGU | [5] + [6,7,8,9] |
| `vmovapd ⟨MEM[128]⟩, ⟨XMM⟩` | FP 2 | [5] + [2] |

- For vmovd, we observe inconsistent resource conflicts when combined with different instructions. As this instruction scheme is untypical in that it transfers data between vector registers and the GPRs, its throughput might depend on resources outside of the port mapping model.

These instructions cause UNSAT results in the *findMapping* method. We exclude them and instructions with the same mnemonics (as we expect them to share aspects of the problematic instructions) from this investigation.

In three runs with the remaining blocking instructions, the algorithm terminated within 12–20 hours after generating 55–59 experiments with up to five instructions. Table 2 shows the inferred port mapping and the documented port usage. For vector and FP instructions, where documented port usages are available, our port mapping is equivalent.

Results for the add blocking instruction differ across repeated algorithm runs in whether a port is shared with the FP instructions: Besides the mapping in Table 2, "[6,7,8,9]", variants like "[0,6,7,8]" and "[1,6,7,8]" that use FP ports are possible. These variants are indistinguishable with the throughput bottleneck of five instructions per cycle. Which result we get depends on choices of the SMT solver. This ambiguity would be resolved with a less tight bottleneck or with blocking instructions for the individual FP ports or fine-grained subsets of the ALU ports. We use "[6,7,8,9]" in the rest of the algorithm as it is consistent with the documentation.

The results for the improper blocking instructions (at the bottom of the table) are consistent with the expectations: They have a μop (presumably for storing to memory) for port 5 in common. vmovapd has an additional μop for port 2, which uops.info reports as its port usage. For mov, the additional μop is an ALU μop, matching our observations from Section 4.1.

### 4.4 Computing the Remaining Port Mapping

Finally, the algorithm benchmarks the remaining instructions against the suite of blocking instructions. To combat unstable measurements, we run this part of the algorithm three times and only report the port usage for an instruction if at least two of the runs agree. We use `mov ⟨MEM[32]⟩, ⟨GPR[32]⟩` to block the store port 5.

The results follow regular patterns for most instructions:

- 256-bit wide AVX instructions use μops of the same kinds as the 128-bit variants, only with twice the number, e.g.:

  `vpcmpeqq ⟨XMM⟩, ⟨XMM⟩, ⟨XMM⟩` $\rightsquigarrow 1 \times [0,3]$
  `vpcmpeqq ⟨YMM⟩, ⟨YMM⟩, ⟨YMM⟩` $\rightsquigarrow 2 \times [0,3]$

- Instruction schemes with a read memory operand differ from their register-only counterparts by one load μop (two for double-pumped 256-bit AVX instructions), e.g.:

  `add ⟨GPR[32]⟩, ⟨GPR[32]⟩` $\rightsquigarrow [6,7,8,9]$
  `add ⟨GPR[32]⟩, ⟨MEM[32]⟩` $\rightsquigarrow [6,7,8,9] + [4,5]$

  This follows our postulated macro-op decomposition.

- Simple scalar instructions with a read and written memory operand use an ALU μop and a store μop:

  `add ⟨GPR[32]⟩, ⟨GPR[32]⟩` $\rightsquigarrow [6,7,8,9]$
  `add ⟨MEM[32]⟩, ⟨GPR[32]⟩` $\rightsquigarrow [6,7,8,9] + [5]$

  In contrast to Intel architectures, Zen+ has no separate μops for the two memory operations in read-modify-write instructions. As an exception, operations on ≤ 32 bit use an additional μop on the address generation units [4,5].

Overall, 70% of the remaining 1,819 considered instruction schemes fall into this category.

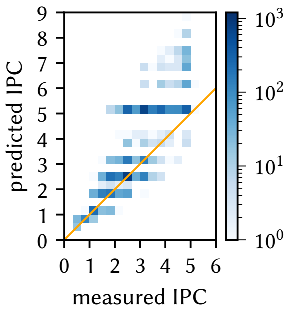
For complex instructions, we find unexpected results, e.g.:

`bsf ⟨GPR[64]⟩, ⟨MEM[64]⟩`
$\rightsquigarrow 9 \times [6,7,8,9] + [4,5] + 9 \times [0,1,2,3]$
`vphaddw ⟨XMM⟩, ⟨XMM⟩, ⟨XMM⟩`
$\rightsquigarrow [0,1,2,3] + [0,1,3] + 2 \times [1,2] + 4 \times [6,7,8,9]$

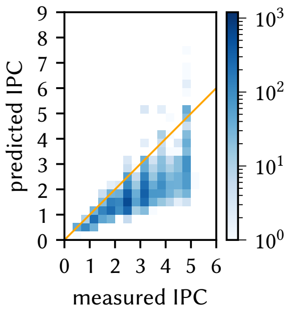
The former is a *bit scan forward* instruction, which finds the position of the least significant bit set in its read (memory) operand. The latter is a horizontal vector addition. Their inferred port usages are unexpected in two ways: They contain more μops than reported by the performance counter (8+1 counted and adjusted for a memory operand for bsf and 4 counted for vphaddw) and they include μops for unlikely ports. For the scalar integer operation bsf, we do not expect vector/FP ports $[0,1,2,3]$, whereas the vector operation vphaddw is unlikely to use the scalar ALU ports $[6,7,8,9]$. We suspect these to be spurious observations caused by the processor's microcode sequencer (MS). For instructions with many μops, the processor's instruction decoder only emits an entry point address for the MS ROM. The MS then emits the relevant operations. Our observations match a MS that emits four operations per cycle while stalling the remaining frontend. Rather than μops that cannot execute on unblocked

| | MAPE | PCC | $\tau_K$ |
|---|---|---|---|
| PMEvo | 28.0% | 0.83 | 0.72 |
| Palmed | 35.2% | 0.79 | 0.66 |
| Ours | 6.6% | 0.96 | 0.90 |

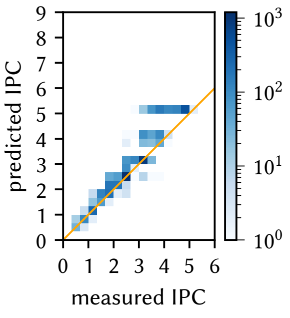
**(a)** Accuracy Metrics



**(b)** PMEvo



**(c)** Palmed



**(d)** ours

**Figure 5.** IPC prediction accuracy for Zen+ in metrics (a) and as heatmaps of predicted vs. measured IPC per model (b-d).

ports, we measure the overhead of this bottleneck. This occurs for 8% of the 1,819 considered instruction schemes.

For 7% of the instruction schemes, e.g., for bit shift operations on vector registers, the experiments yield throughputs that are unstable or outside the port mapping model.

This last stage of the algorithm takes 8–10 hours. The last two stages of the algorithm dominate the running time of the algorithm, with a total of 20–28 hours. Overall, we inferred a port mapping for 1,700 of the initial 2,980 instruction schemes. uops.info has no port mapping for 1,142 (67%) of these 1,700 supported instruction schemes.

### 4.5 Prediction Accuracy – Port Mapping

We evaluate our Zen+ port mapping quantitatively by comparing its throughput prediction accuracy against PMEvo [29] and Palmed [14].[9] As port mappings model only the use of functional units, we focus on instruction sequences whose throughput is not limited by data dependencies.

To predict the throughput of an experiment $e$ with our mapping, we solve the LP from Section 2.2 for the number $t$ of cycles of an optimal execution w.r.t. the port mapping. If this number is faster than the bottleneck of 5 IPC allows, we report an inverse throughput of $5/|e|$ cycles, and $t$ otherwise. For PMEvo, we combine the available implementation with the measurement setup used for our case study and infer a new port mapping. We seed the population of its evolutionary algorithm with 50,000 random port mappings and let it run until evolution converges after ca. 59 hours.[10] For Palmed, we use the most recent available model for the Zen architecture. To keep benchmarking times for PMEvo manageable, we restrict this evaluation to instruction schemes that occur in compiled binaries for the SPEC CPU2017 benchmarks [11] and are covered by our mapping.[11] From the resulting 577 instruction schemes, we generate 5,000 dependency-free basic blocks, each consisting of five randomly sampled instructions. We benchmark their throughput in instructions per cycles (IPC) on the Zen+ hardware.

Figure 5a shows the IPC prediction accuracy in terms of mean absolute percentage error (MAPE), Pearson's correlation coefficient (PCC), and Kendall's $\tau_K$ for each tool. A high PCC indicates a linear correlation of predictions and measurements whereas a high $\tau_K$ implies that sorting the instruction sequences by predicted or measured IPC leads to similar rankings. Both metrics can range between -1 and 1.

The predictions of PMEvo and Palmed share a similar level of accuracy, with significant correlations but rather high errors of 28–35%. Our model is substantially more accurate with an error of 6.7% and very strong linear and rank correlations. The heatmaps in Figure 5 quantify each tool's prediction accuracy in more detail. They group the basic blocks into buckets based on the IPC we observed in the benchmarks and the predictions of each model. Buckets are displayed darker the more basic blocks they contain; the closer the darker buckets are to the diagonal line (orange) the closer are predictions and observations. The heatmap for our model, Figure 5d, is notably closer to the diagonal than PMEvo's and Palmed's.

The structure of PMEvo's mapping differs substantially from ours: There, most instructions have only a single kind of µop in their port usage. Our explainable approach captures structures of the microarchitecture that PMEvo does not resolve. As shown in Figure 5c, Palmed's resource model usually predicts slower executions than what we measure in microbenchmarks. As Palmed depends on assumptions in its measurement infrastructure, we cannot evaluate whether

[9]See Section 5 for a conceptual comparison to these approaches.

[10]Following the paper, we do not adjust PMEvo's predictions for the IPC bottleneck. Adjusting only causes minor differences in the metrics.

[11]The Palmed model includes data for almost all of the instruction schemes we extracted from uops.info.

its model would be more consistent with our throughput measurements if it used our microbenchmarking setup.

## 5 Related Work

Aside from uops.info [1], which the previous sections discuss extensively, two other works in the field of port mapping inference are comparable. PMEvo [29] has even weaker requirements on performance counters than this approach, using only time measurements. This flexibility comes at the cost of explainability: PMEvo uses an evolutionary algorithm to optimize candidate port mappings such that they accurately model the performance of a fixed set of microbenchmarks. In contrast to our approach, there is no tangible justification for the entries of port mappings found by PMEvo. PMEvo uses a heuristically chosen set of benchmarks where complex interactions between instructions might not be represented.

Like our approach, Palmed [14] takes inspiration from the uops.info algorithm: They proceed in two phases, first finding a core mapping for a small instruction set and then inferring models for all other instructions based on the core mapping. Rather than identifying blocking instructions with a µop counter, they select basic instructions for their core mapping heuristically based on throughput benchmarks. Instead of a traditional port mapping, this core mapping (as well as the final result of Palmed) is a *conjunctive mapping* that captures the stress that each instruction puts on various abstract resources of the processor. Palmed uses (integer) linear programming to construct a set of abstract resources that represent possible bottlenecks in the execution of core mapping instructions. They further generate for each resource a kernel of basic instructions that saturates the resource, similar to how blocking instructions flood their corresponding set of ports. Palmed then benchmarks every instruction that is not in the core mapping individually with the saturating kernels and uses a linear program to compute the pressure the instruction puts on the corresponding abstract resources.

Besides port sets, Palmed's resource model inherently represents other potential bottlenecks like the maximal execution rate of the frontend, which our approach needs to treat explicitly. However, conjunctive mappings are challenging to integrate with existing performance models since the inferred abstract resources have no clear correspondence to documented aspects of the microarchitecture.

Approaches like Ithemal [28] and Granite [32] use machine learning to infer instruction-level throughput models. Among other factors, they model effects of the utilization of the processor's functional units. However, as black-box models, they provide less insight into how instructions are executed compared to throughput predictors like CQA [30], llvm-mca [10, 27], OSACA [25], uiCA [3], or Facile [4] with an explicit port mapping as we infer it.

## 6 Conclusion

We have shown that per-port µop counters are not necessary for a uops.info-style port mapping inference algorithm. If the processor under investigation follows the port mapping model, we can infer the port usage of instructions efficiently.

Our study of AMD's Zen+ microarchitecture indicates that the approach is practical for a large portion of the processor's instructions. However, there are practical hindrances like throughput bottlenecks in parts of the processor, misdocumented performance counters, and complex micro-coded or non-pipelined instructions. Nevertheless, we uncovered details of the Zen+ microarchitecture that have, to the best of our knowledge, not been previously documented. We inferred the first explainable port mapping for over 1,000 instruction schemes on Zen+ that were out of scope for previous work and demonstrated its ability to accurately model performance characteristics of the microarchitecture.

## A Artifact

### A.1 Abstract

This work is accompanied by an artifact that includes our prototype implmentation of the proposed port mapping inference algorithm as well as the data sets and results of the case study in Section 4. In particular, the provided results include human-readable and machine-readable representations of the inferred Zen+ port mapping, the PMEvo and Palmed models used as points of comparison, and the raw data for Figure 5. We provide the artifact as a public repository on Github and as an archived virtual machine image bundled with all software dependencies that can be run using Vagrant and VirtualBox.

### A.2 Artifact check-list (meta-information)

- **Algorithm:** Port Mapping Inference
- **Run-time environment:** Python 3, Linux, VirtualBox
- **Hardware:** x86-64
- **Disk space required:** less than 10 GB
- **Publicly available:** https://github.com/cdl-saarland/pmtestbench
- **Code licenses:** MIT
- **Archived:** https://zenodo.org/doi/10.5281/zenodo.10794887

### A.3 Description

**A.3.1 How to access.** The artifact can be accessed from the URLs listed in the above check-list. The source code and data is available via the Github URL whereas the virtual machine image that bundles source code and data with the necessary dependencies is available at the archive URL.

**A.3.2 Hardware dependencies.** The virtual machine image is built for x86-64 processors.

### A.4 Installation

The virtual machine image comes with installation instructions in the artifact_usage.md file. The source code has installation instructions in its README.md file.

### A.5 Evaluation and expected results

The virtual machine image comes with a suggested workflow for evaluating the artifact in the artifact_usage.md file. This workflow includes reproducing the heatmaps of Figure 5.

## References

[1] Andreas Abel and Jan Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 673–686. ACM, 2019.

[2] 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)*, pages 34–46. IEEE, 2020.

[3] Andreas Abel and Jan Reineke. uiCA: Accurate throughput prediction of basic blocks on recent intel microarchitectures. In Lawrence Rauchwerger, Kirk W. Cameron, Dimitrios S. Nikolopoulos, and Dionisios N. Pnevmatikatos, editors, *ICS '22: 2022 International Conference on Supercomputing, Virtual Event, June 28 - 30, 2022*, pages 33:1–33:14. ACM, 2022.

[4] Andreas Abel, Shrey Sharma, and Jan Reineke. Facile: Fast, accurate, and interpretable basic-block throughput prediction. In *IEEE International Symposium on Workload Characterization, IISWC 2023, Ghent, Belgium, October 1-3, 2023*, pages 87–99. IEEE, 2023.

[5] AMD. *Software Optimization Guide for AMD Family 17h Processors*. 2017.

[6] AMD. *Processor Programming Reference for AMD Family 17h Models 01h,08h, Revision B2 Processors*. 2019.

[7] AMD. *Processor Programming Reference (PPR) for AMD Family 19h Model 21h, Revision B0 Processors*. 2021.

[8] ARM. ARM Neoverse V2 core software optimization guide. https://developer.arm.com/documentation/PJDOC-466751330-593177/r0p2/, 2022. Revision r0p2, Accessed: March 1, 2024.

[9] ARM. ARM Neoverse V2 core technical reference manual. https://developer.arm.com/documentation/102375/0002/, 2022. Revision r0p2, Accessed: March 1, 2024.

[10] Andrea Di Biagio. llvm-mca: A static performance analysis tool. https://lists.llvm.org/pipermail/llvm-dev/2018-March/121490.html, 2018. Accessed: 2023-08-22.

[11] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 41–42. ACM, 2018.

[12] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[13] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[14] Nicolas Derumigny, Théophile Bastian, Fabian Gruber, Guillaume Iooss, Christophe Guillon, Louis-Noël Pouchet, and Fabrice Rastello. PALMED: throughput characterization for superscalar architectures. In Jae W. Lee, Sebastian Hack, and Tatiana Shpeisman, editors, *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2022, Seoul, Korea, Republic of, April 2-6, 2022*, pages 106–117. IEEE, 2022.

[15] Agner Fog. The microarchitecture of Intel, AMD, and VIA CPUs. https://www.agner.org/optimize/microarchitecture.pdf, 2022. Accessed: 2023-08-22.

[16] Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. https://www.agner.org/optimize/instruction_tables.pdf, 2023. Accessed: 2023-08-22.

[17] Fujitsu. A64FX microarchitecture manual. https://github.com/fujitsu/A64FX/blob/master/doc/A64FX_Microarchitecture_Manual_en_1.8.1.pdf, 2022. Version 1.8.1, Accessed: March 1, 2024.

[18] Fujitsu. A64FX pmu events. https://github.com/fujitsu/A64FX/blob/master/doc/A64FX_PMU_Events_v1.3.pdf, 2022. Version 1.3, Accessed: March 1, 2024.

[19] GCC. GCC - scheduling model for AMD Zen microarchitectures. https://github.com/gcc-mirror/gcc/blob/15b83b69ca99d97643075776ba94f2dd1f05b46e/gcc/config/i386/znver.md, 2023. Accessed: 2023-08-22.

[20] Thomas Gruber, Jan Eitzinger, Georg Hager, and Gerhard Wellein. LIKWID, November 2023. https://doi.org/10.5281/zenodo.10105559.

[21] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach – 6th Edition*. Elsevier, 2017.

[22] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. 2023.

[23] Intel. Performance monitoring events for 12th and 13th generation intel core processors. https://perfmon-events.intel.com/ahybrid.html, 2024. Accessed: March 1, 2024.

[24] Dougall Johnson. Apple M1 microarchitecture research. https://dougallj.github.io/applecpu/firestorm.html, 2021. Accessed: 2023-08-22.

[25] Jan Laukemann, Julian Hammer, Johannes Hofmann, Georg Hager, and Gerhard Wellein. Automated instruction stream throughput prediction for Intel and AMD microarchitectures. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 121–131. IEEE, 2018.

[26] LLVM. LLVM - scheduling model for AMD Zen microarchitectures. https://github.com/llvm/llvm-project/blob/4eb1f1fab35d0f386b458bf1da4396bbeb00b04f/llvm/lib/Target/X86/X86ScheduleZnver1.td, 2023. Accessed: 2023-08-22.

[27] LLVM. llvm-mca - LLVM machine code analyzer (command guide). https://llvm.org/docs/CommandGuide/llvm-mca.html, 2023. Accessed: 2023-08-22.

[28] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 4505–4515, Long Beach, California, USA, 09–15 Jun 2019. PMLR.

[29] Fabian Ritter and Sebastian Hack. PMEvo: Portable inference of port mappings for out-of-order processors by evolutionary optimization. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 608–622. ACM, 2020.

[30] Andres Charif Rubial, Emmanuel Oseret, Jose Noudohouenou, William Jalby, and Ghislain Lartigue. CQA: A code quality analyzer tool at

binary level. In *21st International Conference on High Performance Computing, HiPC 2014, Goa, India, December 17-20, 2014*, pages 1–10. IEEE Computer Society, 2014.

[31] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In John Paul Shen and Margaret Martonosi, editors, *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 404–415. ACM, 2006.

[32] Ondrej Sýkora, Phitchaya Mangpo Phothilimthana, Charith Mendis, and Amir Yazdanbakhsh. GRANITE: A graph neural network model for basic block throughput estimation. pages 14–26, 2022.

[33] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

[34] Jan Treibig, Georg Hager, and Gerhard Wellein. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In Wang-Chien Lee and Xin Yuan, editors, *39th International Conference on Parallel Processing, ICPP Workshops 2010, San Diego, California, USA, 13-16 September 2010*, pages 207–216. IEEE Computer Society, 2010.

[35] WikiChip. Zen - microarchitectures - AMD. https://en.wikichip.org/wiki/amd/microarchitectures/zen, 2023. Accessed: 2023-08-22.

[36] WikiChip. Zen+ - microarchitectures - AMD. https://en.wikichip.org/wiki/amd/microarchitectures/zen%2B, 2023. Accessed: 2023-08-22.