PMEvo: Portable Inference of Port Mappings for Out-of-Order Processors by Evolutionary Optimization

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

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

Abstract

Achieving peak performance in a computer system requires optimizations in every layer of the system, be it hardware or software. A detailed understanding of the underlying hardware, and especially the processor, is crucial to optimize software. One key criterion for the performance of a processor is its ability to exploit instruction-level parallelism. This ability is determined by the port mapping of the processor, which describes the execution units of the processor for each instruction.

Processor manufacturers usually do not share the port mappings of their microarchitectures. While approaches to automatically infer port mappings from experiments exist, they are based on processor-specific hardware performance counters that are not available on every platform.

We present PMEvo, a framework to automatically infer port mappings solely based on the measurement of the execution time of short instruction sequences. PMEvo uses an evolutionary algorithm that evaluates the fitness of candidate mappings with an analytical throughput model formulated as a linear program. Our prototype implementation infers a port mapping for Intel's Skylake architecture that predicts measured instruction throughput with an accuracy that is competitive to existing work. Furthermore, it finds port mappings for AMD's Zen+ architecture and the ARM Cortex-A72 architecture, which are out of scope of existing techniques.

CCS Concepts: • **Computer systems organization → Reduced instruction set computing**; **Complex instruction set computing**; • **General and reference → Measurement**; **Experimentation**; **Performance**; **Estimation**; **Metrics**; • **Software and its engineering →** *Compilers*; **Software performance**; • **Security and privacy → Hardware reverse engineering**; • **Computing methodologies →** *Simulation tools*; **Genetic algorithms**; • **Theory of computation → Evolutionary algorithms**.

Keywords: port mapping, evolutionary algorithm, processor reverse engineering

ACM Reference Format:
Fabian Ritter and Sebastian Hack. 2020. PMEvo: Portable Inference of Port Mappings for Out-of-Order Processors by Evolutionary Optimization. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3385412.3385995

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3385995

1 Introduction

Accurately estimating the time required to execute a given program has become increasingly complex. While advances in hardware design enable faster execution times, they make it difficult to optimize programs such that they utilize the available resources to the best possible extent. A particular cause of unforeseen performance characteristics is the exploitation of instruction level parallelism via out-of-order execution [23]. This technique enables the processor to dynamically re-order the instructions of a sequential program and execute them in parallel on a set of execution ports. Therefore, optimizing a program to achieve peak performance on a processor requires knowledge of the ports that can be used by each instruction. However, the instruction-to-port mapping, or port mapping, is usually only known to hardware manufacturers and may vary with each new hardware generation.

While approaches towards understanding the performance characteristics of processors without full insight into their internals exist, they suffer from shortcomings: Some approaches require significant manual effort [7, 12] or are restricted to validating existing port mappings [19]. Others are closely tied to microarchitecture-specific performance counters [1, 12, 13] that prevent their applicability to a wide

range of practically-relevant processors. Another line of research [20] uses machine learning to train a neural network that estimates instruction throughput. This approach is portable among microarchitectures, but the resulting black box model is hard to use for identifying concrete performance bottlenecks.

This paper proposes a solution that comes without any of these drawbacks: Experiments are automatically generated from a description of the available instructions. Performing the experiments requires only measuring the time taken for executing an instruction sequence. The result is a concise and interpretable port mapping model that existing tools can use to identify bottlenecks and to guide optimization decisions.

We achieve this with PMEvo, a framework that uses an evolutionary algorithm to find a port mapping that excels in explaining measured throughputs for automatically generated instruction sequences. These instruction sequences are designed to reveal conflicting resource requirements for pairs of instructions while exhibiting as few data dependencies as possible. For these instruction sequences, the throughput is only limited by constrained ports and therefore carries information about the port mapping.

A key component of the evolutionary algorithm is a novel bottleneck simulation algorithm to evaluate the fitness of candidate port mappings. This algorithm efficiently computes the solution of a linear program that models an optimal instruction scheduler for a given port mapping. Our novel bottleneck simulation algorithm outperforms solving the corresponding linear program for realistic port mappings by two orders of magnitude.

We evaluate PMEvo by the throughput prediction accuracy of its inferred port mappings on port-mapping-bound basic blocks for microarchitectures by Intel, AMD and ARM. PMEvo's prediction accuracy for the Intel Skylake architecture is close to existing approaches like IACA [17] and uops.info [1] that rely on stronger knowledge about the microarchitecture. For AMD and ARM, PMEvo outclasses the state-of-the-art port mapping model of llvm-mca [7].

In summary, we make the following contributions:

- An evolutionary algorithm that infers port mappings from specifically designed experiments with measured throughputs without relying on microarchitecture-specific features.
- A bottleneck simulation algorithm that allows to efficiently evaluate the fitness of port mappings for a given set of experiments.
- A prototype implementation that finds a port mapping that is competitive to related work for Intel's Skylake architecture and that is the first one to automatically find port mappings for the AMD Zen+ and the ARM Cortex-A72 microarchitectures.

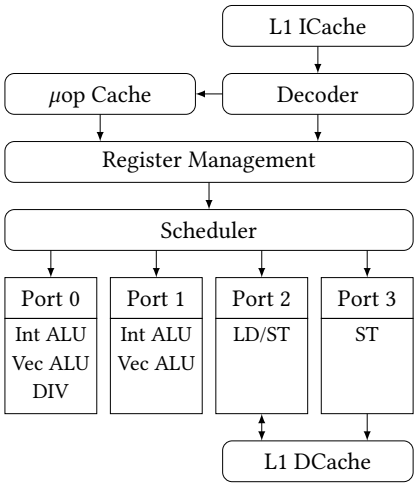

Figure 1. Simplified overview of a modern processor design (based on Figure 2-3 in the Intel Software Optimization Manual [16])

2 Background: Processor Design

Modern processors apply out-of-order execution [23].[1] This concept is based on the observation that instructions can be executed in any order as long as the results are the same as if they were executed in program order. Therefore, a processor may execute instructions in parallel and reorder them to any extent that preserves the read-after-write dependencies between the operations and the externally-visible effects.

Out-of-order execution is often combined with a scheme to decompose instructions into simpler microarchitecture-specific operations. These so-called micro-ops or μops are then subject to reordering.

Figure 1 shows the relevant parts of a microarchitecture that employs out-of-order execution and μop decomposition. Instructions are fetched and decoded from the instruction cache in program order. The decoder produces μops, which are cached for future re-use. The register management engine resolves false (write-after-read or write-after-write) dependencies by mapping the operand registers of each operation to a larger number of physical registers. A scheduler decides based on operand dependencies and resource availability when and where to execute the μops. The execution units (e.g. arithmetical units and load/store units), which execute the μops, are grouped behind ports. Often, execution units are pipelined, allowing the ports to start processing a new instruction in every cycle. Several instances of the same kind of execution unit can exist at different ports.

A key factor to the running time of a given piece of code on a processor is therefore the port mapping. It specifies how

[1] A contemporary introduction to the topic can be found e.g. in Chapter 3 of the textbook by Hennessy and Patterson [15].

instructions are decomposed into μops and which μops can be executed on which ports.[2]

Some microarchitectures, particularly those designed by Intel, provide fine-grained hardware performance counters that count the number of executed μops per port. While these greatly help at inferring port mappings, relying on them excludes all microarchitectures that do not provide similar performance counters. Therefore, we base our approach on the more portable observation of throughputs as defined in the following.[3]

Definition 1. The *throughput* $t^*(e)$ of an instruction sequence (or experiment) e on a given processor is the average number of processor cycles required to execute e in a steady state.

The execution of an experiment in an infinite loop is considered to have reached a steady state when the average number of required cycles per iteration stays constant for the remaining execution.

3 Analytical Throughput Model

Since our goal is to infer a port mapping from throughput measurements, we need to understand the connection between the port mapping of the processor and the throughput that is achieved for an experiment. The precise inner workings of processors are well-kept secrets of the manufacturers, therefore we postulate a model of how processors execute instructions with respect to a port mapping. In this section, we present a model that is supported by information provided by hardware manufacturers [2, 4, 16, 17] and related research [1, 12]. The linear programs we use in this section to define the throughput for a given port mapping are extensions of work presented by Abel and Reineke [1].

3.1 Out-of-Order Throughput Model

We start by defining a simple out-of-order execution model that does not consider the decomposition of instructions into μops. We refer to it as the *two-level model* (mapping only instructions to ports). Section 3.2 extends this model to the *three-level model*, which additionally supports decomposing instructions into μops.

Our throughput model is based on the notion of a port mapping, as defined in the following.

Definition 2. A *port mapping in the two-level model* is a bipartite graph $(\mathbf{I} \cup \mathbf{P}, M)$ with the nodes split disjointly into a set $\mathbf{I}$ of instructions and a set $\mathbf{P}$ of ports and edges $M \subseteq \mathbf{I} \times \mathbf{P}$ between these.

An edge between instruction i and port k indicates that instruction i can be executed on port k.

[2]This is called *port usage* in the work by Abel and Reineke [1].

[3]This definition is equivalent to the one by Mendis et al. [20], an extension of the instruction-wise throughput definition used by Fog [12] and Abel and Reineke [1].

Consider the port mapping shown in Figure 2. There are four instructions *mul*, *add*, *sub*, and *store* that are mapped to the three ports P_1, P_2, and P_3. The two instructions *add* and *sub* can both be executed on the same two ports P_1 and P_2, *mul* can use only one of them, P_1, and *store* has to be executed on a separate port P_3.

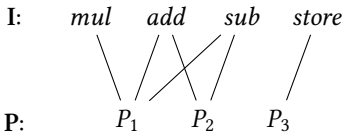

Figure 2. Example of a port mapping in the two-level model

In this model, an experiment is represented as a multiset of instructions, i.e. a function $e : \mathbf{I} \to \mathbb{N}$ that maps instructions to their number of occurrences. We abstract from the order of the instructions since we only use experiments that can be reordered freely by the scheduler. The throughput of an experiment with a given port mapping is characterized by the following definiton.

Definition 3. Given a port mapping $m := (\mathbf{I} \cup \mathbf{P}, M)$ in the two-level setting, the *throughput* $t^*_m(e)$ under m for an experiment $e : \mathbf{I} \to \mathbb{N}$ is the objective value of an optimal solution to the following linear program:

$$\begin{aligned}
&\text{minimize} && t && && \\
&\text{subject to} && \sum_{k \in \mathbf{P}} x_{ik} = e(i) && \text{for all } i \in \mathbf{I} && \text{(A)} \\
& && \sum_{i \in \mathbf{I}} x_{ik} \leq t && \text{for all } k \in \mathbf{P} && \text{(B)} \\
& && x_{ik} \geq 0 && \text{for all } (i,k) \in M && \text{(C)} \\
& && x_{ik} = 0 && \text{for all } (i,k) \notin M && \text{(D)}
\end{aligned}$$

The intuition for this linear program is that each instruction i in the experiment has the mass $e(i)$. This mass is distributed among the ports that can execute i, as required by constraint (A). The x_{ik} are real-valued variables that represent the share of the mass $e(i)$ that is executed on port k in the experiment. Constraint (B) establishes the objective t as an upper bound of the sums of mass shares on each port. The constraints (C) and (D) guarantee that the mass of an instruction i is distributed to ports that can execute i. The throughput is the maximal mass associated to any port if all mass is distributed as evenly as possible.

Example 1. Figure 3 displays a graphical interpretation of an optimal solution of the linear program for the experiment

$$e := \{add \mapsto 2, mul \mapsto 1, store \mapsto 1\}$$

under the mapping given in Figure 2.

The mass allocated to each port P_k is drawn in the corresponding bucket. The throughput of 1.5 cycles is the mass of the most occupied ports P_1 and P_2. Note that the mass

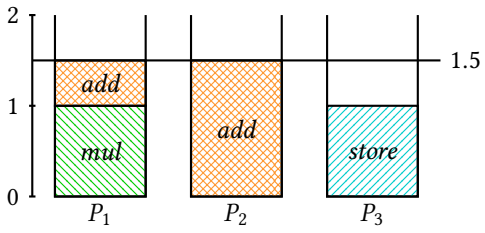

Figure 3. Visualization of an example port allocation

of the two *add* instructions is split unevenly among two ports. While these non-integer instruction portions might seem counter-intuitive, they assort with the definition of throughput as the average number of cycles to execute an experiment.

Such a throughput can be realized by executing one *add* instruction on port P_1 in every second iteration, yielding an average of 0.5 *add* instructions on P_1 per execution of the experiment.

Definition 3 relies on several assumptions:

1. The processor schedules the instructions optimally.
2. Operational units are fully pipelined, i.e. every instruction blocks exactly one port for exactly one cycle.
3. The fetch and decode units do not impose a bottleneck for the experiment.
4. There are no relevant (read-after-write) data dependencies among the instructions of the experiment.

The validity of assumptions (1) and (2) depends on the processor under test. We found these to be usually fulfilled by modern microarchitectures. An exception are complex instructions like divisions, which can block a port or an operational unit for multiple cycles.

We ensure the validity of assumptions (3) and (4) by selecting our experiments appropriately: Only sufficiently short experiments that do not hit bottlenecks in the fetch/decode stages are considered. The operands of the experiments are furthermore chosen such that writing instructions can be retired before their output register is read for the next time.

3.2 Micro-Operation Decomposition

To represent the decomposition of instructions into μops, we extend the definition of a port mapping by a layer of μops as follows.

Definition 4. A *port mapping in the three-level model* is a tripartite graph $(\mathbf{I} \cup \mathbf{U} \cup \mathbf{P}, N \cup M)$ with labeled edges $N \subseteq \mathbf{I} \times \mathbb{N} \times \mathbf{U}$ between instructions and μops as well as unlabeled edges $M \subseteq \mathbf{U} \times \mathbf{P}$ between μops and ports.

A labeled edge $(i, n, u) \in N$ means that there are n instances of the μop u in the μop decomposition of instruction i.

An example is displayed in Figure 4. Here, *add* and *sub* are implemented as one μop U_2 that can be executed on two ports P_1 and P_2. The *mul* and *store* instructions are decomposed into two μops, the former in two of the same kind, U_1, and the latter into two different ones, U_2 and U_3. The *store* instruction has a partial conflict with *add* and *sub* that cannot be represented in the two-level model.

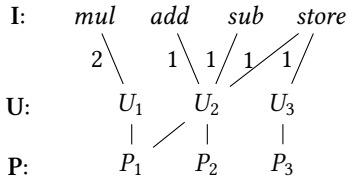

Figure 4. Example of a three-level port mapping

It is important to notice the different semantics of the layers of edges: For each instance of an instruction i, *all* corresponding μops u such that $(i, \cdot, u) \in N$ have to be executed whereas a μop u is executed on *exactly one* of the allowed ports k such that $(u, k) \in M$. The linear program from Section 3.1 can be slightly modified to compute the throughput $t^*_m(e)$ of an experiment $e : \mathbf{I} \to \mathbb{N}$ under the *three-level* port mapping $m := (\mathbf{I} \cup \mathbf{U} \cup \mathbf{P}, N \cup M)$:

$$
\begin{array}{lll}
\text{minimize} & t & \\
\text{subject to} & \displaystyle\sum_{k \in \mathbf{P}} x_{uk} = \sum_{(i,n,u) \in N} e(i) \cdot n & \text{for all } u \in \mathbf{U} \quad \text{(A)} \\
& \displaystyle\sum_{u \in \mathbf{U}} x_{uk} \le t & \text{for all } k \in \mathbf{P} \quad \text{(B)} \\
& x_{uk} \ge 0 & \text{for all } (u,k) \in M \quad \text{(C)} \\
& x_{uk} = 0 & \text{for all } (u,k) \notin M \quad \text{(D)}
\end{array}
$$

All previous occurrences of instructions are replaced by occurrences of μops except for the right-hand side of constraint (A). The right-hand side of (A) ensures that a μop u that occurs n times in the decomposition of instruction i is taken into account with its appropriate mass.

A valuable observation is that computing the throughput of an experiment $e : \mathbf{I} \to \mathbb{N}$ with a port mapping $(\mathbf{I} \cup \mathbf{U} \cup \mathbf{P}, N \cup M)$ in the three-level model can be reduced to computing throughput in the simpler two-level model: We instead compute the throughput of the experiment

$$e' = \left\{ u \mapsto \sum_{(i,n,u) \in N} e(i) \cdot n \right\}$$

with the two-level mapping $(\mathbf{U} \cup \mathbf{P}, M)$. The multiset e' contains the μops that are needed to execute e according to N. These μops are used as instructions for the two-level model.

This construction allows us to use an algorithm for the simpler two-level model to compute throughput in the three-level model.

4 The PMEvo Framework

We propose the PMEvo framework to automatically infer port mappings from throughput experiments. An overview of this framework is given in Figure 5.

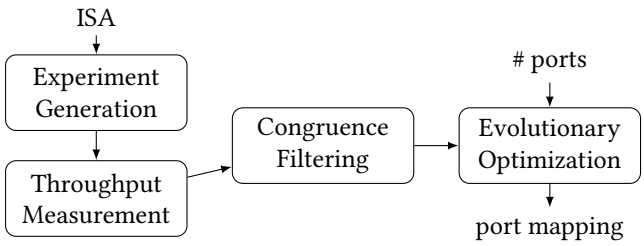

Figure 5. PMEvo framework overview

PMEvo consists of four main stages, which we describe in the following subsections: Generating relevant experiments (4.1), measuring the throughput of the experiments on a given processor (4.2), a preprocessing step that identifies congruent instructions (4.3), and evolutionary optimization (4.4).

4.1 Experiment Generation

The input of the first stage of PMEvo is a description of the instruction set architecture under test. This description is a set of instruction forms, i.e. instructions with typed placeholders for their operands. The type of the placeholder specifies the operand kind (e.g. memory operand, general purpose or vector register) and the width of the respective operand. There can be multiple instruction forms for the same operation with different operand types.

PMEvo constructs a set of experiments from this information with the following components:

1. for each instruction form i, an experiment $\{i \mapsto 1\}$ measuring its individual throughput $t^*(i)$
2. for each pair (i_A, i_B) of instruction forms, an experiment $\{i_A \mapsto 1, i_B \mapsto 1\}$
3. for each pair (i_A, i_B) of instruction forms with $t^*(i_A) > t^*(i_B)$, an experiment $\{i_A \mapsto 1, i_B \mapsto n\}$ where

$$n = \lceil t^*(i_A)/t^*(i_B) \rceil$$

Experiments with this structure lead to different outcomes depending on the port mapping: If the μops of two instruction forms i_A and i_B require the same resources, experiment (2) will result in a throughput that is the sum of the individual throughputs of i_A and i_B. In case the μops of i_A and i_B are executed by disjoint execution units, the throughput of experiment (3) will be $n \cdot t^*(i_B)$. More complex partial resource conflicts will lead to measured throughputs for these experiments that are harder to interpret manually. It is the task of the evolutionary algorithm to find a mapping that explains these throughputs.

The evolutionary algorithm is not restricted to experiments of this structure. In theory, longer experiments that combine instances of more than two different instruction forms can unveil resource conflicts that cannot be covered by these experiments. However, when exploring the experiment design space experimentally for existing processors, we did not observe benefits in port mapping quality from more complex experiments.

4.2 Throughput Measurement

The goal of this stage is to measure the throughput of the generated experiments. Our measurement method follows Definition 1: The instruction forms of the experiment are instantiated with operands while avoiding data dependencies. The resulting instruction sequence is executed in a loop such that the execution reaches a steady state.

PMEvo uses a register allocator that assigns a register from the appropriate register class to each register operand of the instruction forms. To avoid harmful dependencies, written operands are instantiated with most recently read registers and read operands with least recently written registers. Using as many different registers as available, this ensures that instructions with long latencies have enough time to complete before their results are read.

Memory operands are instantiated with a separate register containing a valid base pointer and one of several different constant offsets to avoid data dependencies on the memory.

Before operand allocation, we unroll several loop iterations. This has several benefits: It further increases the dependence distance by allowing more registers to be allocated and it avoids loop-carried dependencies. Additionally, it reduces the influence of the loop code on our time measurements. The range of loop body lengths that achieve an optimal throughput depends on the microarchitecture. We found a length of 50 instructions to be in the appropriate range for all of our evaluated architectures. With this length, the loop body will be resident in the μop cache (if the architecture has one). This avoids performance bottlenecks due to restrictions on the number of concurrently fetched and decoded instructions. The loop bound is automatically chosen to ensure that the loop runs for a specific time that guarantees steady-state execution. This time is estimated empirically for the processor under test by comparing the measurement stability for different times. For the evaluated platforms, we found a time of 10 ms to be appropriate.

To measure the throughput, we emit the instantiated loop as inline assembly into a C program wrapped with calls to `gettimeofday()` and setup code. The resulting program is compiled with a C compiler for the platform under test and executed. We compute the throughput of the experiment with the following formula:

$$t^*(e) = \frac{\text{measured time} \times \text{frequency}}{\#\text{executed instances of } e}$$

The reported throughput for an experiment is the median over multiple such measurements to accommodate for occasional fluctuations in the processor's clock frequency.

4.3 Congruence Filtering

In a processor microarchitecture, we expect that groups of instruction forms require the same execution resources. Instruction forms whose operations are implemented similarly in the processor, e.g. addition and subtraction, often lead to such groups.

PMEvo exploits these patterns to reduce the search space of the evolutionary algorithm. It partitions the set of instruction forms into congruence classes of instruction forms that are not distinguishable with the generated experiment set.

In this partitioning, two instruction forms i_A and i_B are in the same class if and only if the following conditions hold:

- i_A and i_B exhibit equal individual throughputs.
- Any two experiments $\{i_A \mapsto m, i_C \mapsto n\}$ and $\{i_B \mapsto m, i_C \mapsto n\}$ that combine these instruction forms with any other instruction form i_C exhibit equal throughputs.

For this purpose, we consider throughputs t_1 and t_2 equal (up to measurement errors) if their symmetric relative difference is limited by a user-specified constant ε, i.e. if

$$\frac{|t_1 - t_2|}{|t_1 + t_2|/2} < \varepsilon$$

For each congruence class, PMEvo selects a representative to be included in the instruction set for the evolutionary algorithm. The evolutionary algorithm then only needs to consider experiments that consist of these representatives.

4.4 Evolving Port Mappings

The core of PMEvo is an evolutionary algorithm that searches for a port mapping that accurately explains the observed throughputs for a given set of experiments. Evolutionary algorithms are a well-proven technique to approach optimization problems. They mimic concepts from natural evolution to approximatively optimize complex metrics in non-linear problem settings. We refer to the textbook by De Jong [11] for a comprehensive treatment.

Every evolutionary algorithm is centered around a representation scheme that characterizes the space of possible solutions of the optimization problem. Naturally, the scheme that we use is that of port mappings with μop decomposition as described in Section 3.2. The sets **I** of Instructions and **P** of Ports are given by the user. We identify each μop with the set of ports that can execute it and allow all non-empty subsets of **P** as μops. The width $|u| = |\{k \mid (u,k) \in M\}|$ of a μop u is the number of ports that can execute u.

PMEvo's evolutionary algorithm follows the structure in Algorithm 1. Initially, a set of p port mappings is sampled randomly to form a population. This population is iteratively refined through evolution steps. In each such step, p child mappings are generated via evolutionary operators. The resulting population of $2p$ port mappings is sorted according to the fitness metric and the best-performing p mappings are selected as the new population. The evolution terminates once the fitness of the population has converged to a single value or an iteration limit is exceeded. By selecting a value for p, the user can find a trade-off between inference time and quality of the inferred port mapping.

```
initialize population randomly
while not done do
    apply evolutionary operators
    evaluate fitness
    select new population
end
perform local search
return fittest individual
```

Algorithm 1. Structure of the evolutionary algorithm

After the evolution terminates, PMEvo employs a greedy hill-climbing algorithm to move from the found solutions to a local optimum in the space of possible port mappings. It incrementally adjusts the number n of μop occurrences for each edge $(i, n, u) \in N$ and keeps the changes to the port mapping if it is fitter than before.

In the following, we describe the components that constitute the evolutionary algorithm in detail.

Initialization. Each member of the initial population is sampled randomly from the set of possible port mappings as follows. For each instruction i, a random set of 1 to $|\mathbf{P}|$ many different μops is sampled. The number of occurrences for each of these μops u in the mapping for i is sampled from the interval $[1, \lceil t^*(i) \cdot |u| \rceil]$. The upper bound of this interval is an implication of the throughput model: An instruction with $\lceil t \cdot |u| \rceil$ instances of a μop u in its decomposition can achieve no throughput smaller than t.

Evolutionary Operators. Evolutionary operators create new individuals from existing individuals in the population. The most common operators in evolutionary algorithms are recombination and mutation.

We employ a binary recombination operator that mixes the information of two parent mappings to generate two child mappings. For each instruction i, the set of occurring μops with multiplicities is divided randomly into two parts that form the corresponding assignments for the children. This operator is applied to individuals that are sampled uniformly at random from the population.

When designing the evolutionary algorithm, we tried various random mutation strategies. Experiments showed little to no benefit over a design without a mutation operator while contributing substantial numbers of fitness computations. Therefore, we eliminated mutation operators from our

design to explore larger populations more effectively in the same execution time.

Fitness Metric. PMEvo's evolutionary algorithm approximately solves a multiobjective optimization problem (MOP) with the goal of minimizing two metrics: The average relative prediction error D_{avg} and the μop volume V. These metrics describe the quality of a port mapping $m = (\mathbf{I} \cup \mathbf{U} \cup \mathbf{P}, N \cup M)$ for a set $E \subseteq (\mathbf{I} \to \mathbb{N}) \times \mathbb{R}$ of experiments with measured throughputs as follows:

$$D_{avg}(m) = \frac{1}{|E|} \sum_{(e,t) \in E} \frac{|t^*_m(e) - t|}{t}$$

$$V(m) = \sum_{(i,n,u) \in N} n \cdot |u|$$

A low value for $D_{avg}(m)$ ensures an accurate prediction whereas a smaller μop volume indicates a more compact and therefore more interpretable mapping.

We solve the MOP through a priori scalarization, as described e.g. in Chapter 4.1 of the textbook by Miettinen [21]: We combine the objectives into a single one that is interpreted as the fitness function $F(m)$ as follows:

$$F(m) = \Lambda_1(D_{avg}(m)) + \Lambda_2(V(m))$$

Λ_1 and Λ_2 are affine transformations that are chosen in every iteration to normalize both objective metrics to the range $[0, 1000]$. They ensure that the extremal objective values of the current population are mapped to 0 and 1000, respectively, with all other objective values in between.

Combining the accuracy metric D_{avg} with a compactness metric is necessary because throughput measurements usually do not uniquely identify a single port mapping. The port mapping model is flexible enough to allow for a wide range of well-performing mappings with different characteristics. While the found compact mappings are not necessarily identical to the port mappings that are really used in the processor, they still capture the performance characteristics of the hardware as they are observable from the outside.

4.5 Efficient Bottleneck Simulation Algorithm

Practical applicability of evolutionary algorithms depends on evaluating the fitness of many candidates in as little time as possible. For a given time budget, fitness evaluation speed directly corresponds to the quality of the obtained solution. With faster fitness evaluation, more candidates for survival can be considered, resulting in superior solutions.

Therefore, a critical component of our approach is the efficient simulation of experiments under a given port mapping. Instead of directly solving the linear program from Section 3.1, we use a bottleneck simulation algorithm that computes the optimal solution of the linear program. We restrict our presentation here to port mappings in the two-level model for a more concise description. As we have observed in Section 3.2, this extends to the three-level model straightforwardly.

The bottleneck simulation algorithm implements the following characterization of the throughput $t^*_m(e)$ of an experiment e under the port mapping $m := (\mathbf{I} \cup \mathbf{P}, M)$:

$$t^*_m(e) = \max_{Q \subseteq \mathbf{P}} \frac{\sum\{e(i) \mid Ports(m,i) \subseteq Q\}}{|Q|} \tag{1}$$

$Ports(m,i) := \{k \mid (i,k) \in M\}$ denotes the set of ports that can execute an instruction i under m. This characterization is based on the observation that the throughput $t^*_m(e)$ has to be determined by a non-empty set Q^* of bottleneck ports. Each of the ports in Q^* has to execute a mass of instructions that is equal to $t^*_m(e)$. In other words, $t^*_m(e)$ is equal to the total mass of instructions that need to be executed on ports from Q^*, divided by the size of Q^*. An optimal scheduler will assign instructions that do not need to be executed on ports from Q^* to less utilized ports. For each Q, the maximized term from Equation 1 is a lower bound to $t^*_m(e)$. Consequentially, finding a maximal term gives us precisely the throughput $t^*_m(e)$. A formal proof for this equation is given in Appendix A.

Example 2. For the execution in Figure 3, Q^* is the set $\{P_1, P_2\}$. Trying to move mass from one of these ports to any other ports is either not possible (for *mul*) or causes another port from Q^* to execute more mass. P_3 on the other hand is irrelevant for the throughput of the experiment.

Our algorithmic implementation of this characterization computes the max operation in Equation 1 by enumerating all subsets of the set of ports and evaluating the corresponding term. The run-time of this algorithm is in $\Theta(2^{|\mathbf{P}|})$, which is substantially more expensive than the polynomial runtime of LP solving [6] from a complexity-theoretic point of view. Nevertheless, this algorithm is considerably faster for practical problems, as we show in Section 5.4. On the one hand, this is due to the small number of execution ports available in modern systems. Typical systems have eight (e.g. Intel Skylake [16] and ARM A72 [4]) or ten (e.g. AMD Ryzen [2]) ports available. On the other hand, thanks to the simplicity of the above algorithm, it is amenable to aggressive performance optimizations such as vectorization.

5 Evaluation

This section evaluates three aspects of our work:

- The appropriateness of the processor model as described in Section 3 and our mechanism for measuring throughput (Section 5.2).
- The quality of the inferred port mappings for three microarchitectures from different manufacturers (Section 5.3).
- The performance characteristics of the bottleneck simulation algorithm (Section 5.4).

Table 1. Evaluated processors

	SKL	ZEN	A72
Manufact.	Intel	AMD	RockChip
Processor	Core i7 6700	Ryzen 5 2600X	RK3399
Microarch.	Skylake	Zen+	Cortex-A72
# Ports	8 + DIV	10	7 + BR
Instr. Set	x86-64	x86-64	ARMv8-A
Clock Freq.	3.4 GHz	3.6 GHz	1.8 GHz
RAM	32 GB	32GB	4GB

5.1 Setup

5.1.1 Evaluated Processors. We use three devices with processors of distinct manufacturers for our evaluation, denoted as SKL, ZEN, and A72 in the following. Relevant parameters are listed in Table 1. SKL has a separate pipeline of long-running operations, marked as DIV, that has to be modeled as an additional port. One port of A72 is only used for processing branch instructions (BR). It is omitted in our model as we do not consider instructions that alter control flow. All evaluated systems have frequency scaling and flexible overclocking mechanisms (e.g. Intel Turbo Boost) disabled to facilitate reliable measurements.

A72 and ZEN are of particular interest since they do not provide the per-port performance counters that other approaches rely on [3, 5] whereas SKL gives means for a comparison to related work.

5.1.2 Considered Instructions. We select for each instruction set architecture (ISA) under test a relevant set of instruction forms. These sets are derived from the instructions that compilers emit when compiling the SPEC CPU 2017 benchmarks[8]. Our instruction forms for the ARMv8-A ISA are extracted from the instructions that GCC (version 4.9.4, flags: `-O3`) emits. For x86-64, we only extract the used instruction mnemonics from the output of Clang (version 8.0, flags: `-O3 -mavx2`) and use the machine-readable inputs of Abel and Reineke [1] to generate the corresponding instruction forms.

We exclude the following instructions from these sets:

- Branch and jump instructions, since their throughput heavily depends on the branch predictor.
- Instructions with implicitly read operands, since these cause dependencies that cannot be resolved through register allocation. Throughput for these could be measured by introducing additional dependency-breaking instructions as done by Abel and Reineke [1].
- x86 SSE instructions, since these add transition penalties when benchmarked together with AVX instructions.
- All instruction variants that operate on subregisters, to keep the run time of the evaluation bearable.
- x86 instructions that are not supported by Ithemal [20], to have a common baseline for all comparisons.

The resulting instruction descriptions contain 310 x86-64 instruction forms and 390 ARMv8-A instruction forms.

5.2 Processor Model and Measurements

In this section, we validate the practicality of the throughput model and the measurement mechanism. We compare measured throughputs with the results of a simulation according to the processor model with a ground truth port mapping from the work by Abel and Reineke [1]. Since this work only provides port usage for Intel architectures, we compare their Intel Skylake port mapping to our measurements on SKL. When performing this evaluation, we discovered two bugs in their port mapping that were acknowledged by the authors. We fixed these in the port mapping that is used for our evaluation.

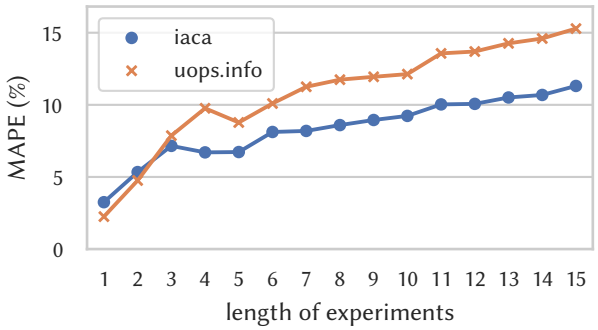

Figure 6. Mean absolute percentage error (MAPE) of simulation with the port mapping from Abel and Reineke [1] and with IACA [17] with respect to our measurements for experiments of varying length

Figure 6 shows the mean absolute percentage error for the simulation with the port mapping from Abel and Reineke with respect to our measurements for varying experiment length. For length 1, we use the set of all supported x86-64 instructions, whereas for larger lengths, we randomly sample 2,000 experiments from the set of all instruction multi-sets of the appropriate size.

For small experiment lengths, we can see a low error showing that the experiments behave as predicted by the processor model. With increasing length of experiments, the accuracy degrades. The lower prediction error of IACA [17] in Figure 6 indicates that with longer experiments, the influence of factors such as non-optimal scheduling decisions that are not covered in the throughput model (but by IACA) rises.

Overall, the error is small enough to justify the use of measurement mechanism and throughput model.

5.3 Model Predictions

Directly measuring the quality of a port mapping is hindered by the lack of ground truth for most processors. We therefore assess the inferred port mappings by their ability to accurately predict the measured throughput of port-mapping-bound experiments. For each microarchitecture, we use a different benchmark set of 40,000 experiments, which we instantiate with operands and whose throughput we measure as described in Section 4.2. These experiments are sampled uniformly at random from the set of all instruction multi-sets of size 5.

One major use case of PMEvo is to provide port mappings for performance estimation tools. Therefore, we compare the prediction accuracy of PMEvo's mappings to the modeling of port mappings in state-of-the-art performance prediction tools. To this end, we use the same benchmark sets to evaluate IACA [17] (version 3.0), llvm-mca [7] (from LLVM version 8.0.1[4]), Ithemal [20], and the port mapping provided by uops.info [1] for their respective supported platforms. Note that these benchmarks specifically stress the port-mapping aspect of these prediction tools because they do not contain any data dependencies. They are therefore not representative to evaluate the overall prediction quality of these tools on compiler-generated code.[5] Section 6 discusses the performance estimation tools we evaluate in further detail.

Of these four related approaches, only the port mapping from uops.info is directly comparable to PMEvo's results because it can only predict the throughput of instruction sequences without data dependencies. The other approaches are more general in that they can predict the throughput of arbitrary instruction sequences, but might not be attempting to provide good accuracy for dependency-free code. For example, Ithemal uses a neural network model trained via supervised learning rather than an explicit port mapping model. Being trained on collected basic blocks from entire programs where dependencies are to be expected, accurate predictions for dependency-free code might be outside of the scope of Ithemal.

For all three platforms, we ran our PMEvo prototype with a population size of 100,000 and an ε of 0.05 for congruence filtering. Table 2 gives numbers on the time required to benchmark throughputs for experiments and to infer a port mapping for all considered platforms. It further shows that the effectiveness of congruence filtering is considerable: The relevant instructions are reduced by 53% to 69%. The low number of different μops used in the inferred port mappings indicates that PMEvo developed compact representations for all three platforms. The uops.info port mapping for SKL uses 12 different μops for the same set of instructions.

Table 2. PMEvo mapping characteristics

	SKL	ZEN	A72
benchmarking time	20h	27h	74h
inference time	5h	21h	12h
insns found congruent	69%	53%	56%
number of μops	17	15	9

To provide a broad comparison of prediction accuracy, we give results for the following commonly used accuracy metrics:

- The Mean Absolute Percentage Error (MAPE) is a measure of the relative error of the simulation over measurements.
- The Pearson Correlation Coefficient (PCC) describes how closely the relation between simulation and measurements can be described by a linear equation.
- The Spearman Correlation Coefficient (SCC) is a measure of rank correlation. A high rank correlation indicates that if the measurement for one experiment is smaller than for another experiment, its simulated value is likely to be smaller as well.

The value range for PCC and SCC is $[-1, 1]$, ranging from negative correlation (-1) over no correlation (0) to maximal correlation (1).

Additionally, we visualize the prediction accuracy of our approach in comparison to related work in Figure 7 with a heat map for each pair of architecture and prediction mechanism. For each heat map, the experiments are considered as data points with measured and predicted throughput. The heat map shows the space of possible pairs of measured and predicted throughput, split into 35×35 equally sized bins. Each bin's shade represents the number of experiments that lie in it. Ideally, measurement and prediction agree, leading to experiments close to the marked diagonal line. Experiments below the diagonal indicate an under-estimation of the throughput, those above are over-estimated by the predictor.

We discuss the represented data in detail in the following sections.

5.3.1 SKL. For the Intel Skylake platform, we compare the prediction accuracy of PMEvo to all afformentioned approaches: the port mapping from uops.info, IACA, llvm-mca, and Ithemal using its publicly-available pre-trained network for the Skylake microarchitecture.

The inputs for IACA, llvm-mca, and Ithemal consist of the loop body of the experiments, unrolled to a length of ten instructions so that operand allocation can avoid loop-carried dependencies. For the entire set of experiments, we

[4] Initially, we performed these experiments on the more recent LLVM version 9.0.1 but found a severe regression in prediction accuracy on our experiments compared to version 8.0.1.

[5] We refer to the BHive project [10] for an evaluation of their accuracy for instruction sequences extracted from code generated for common benchmarks.

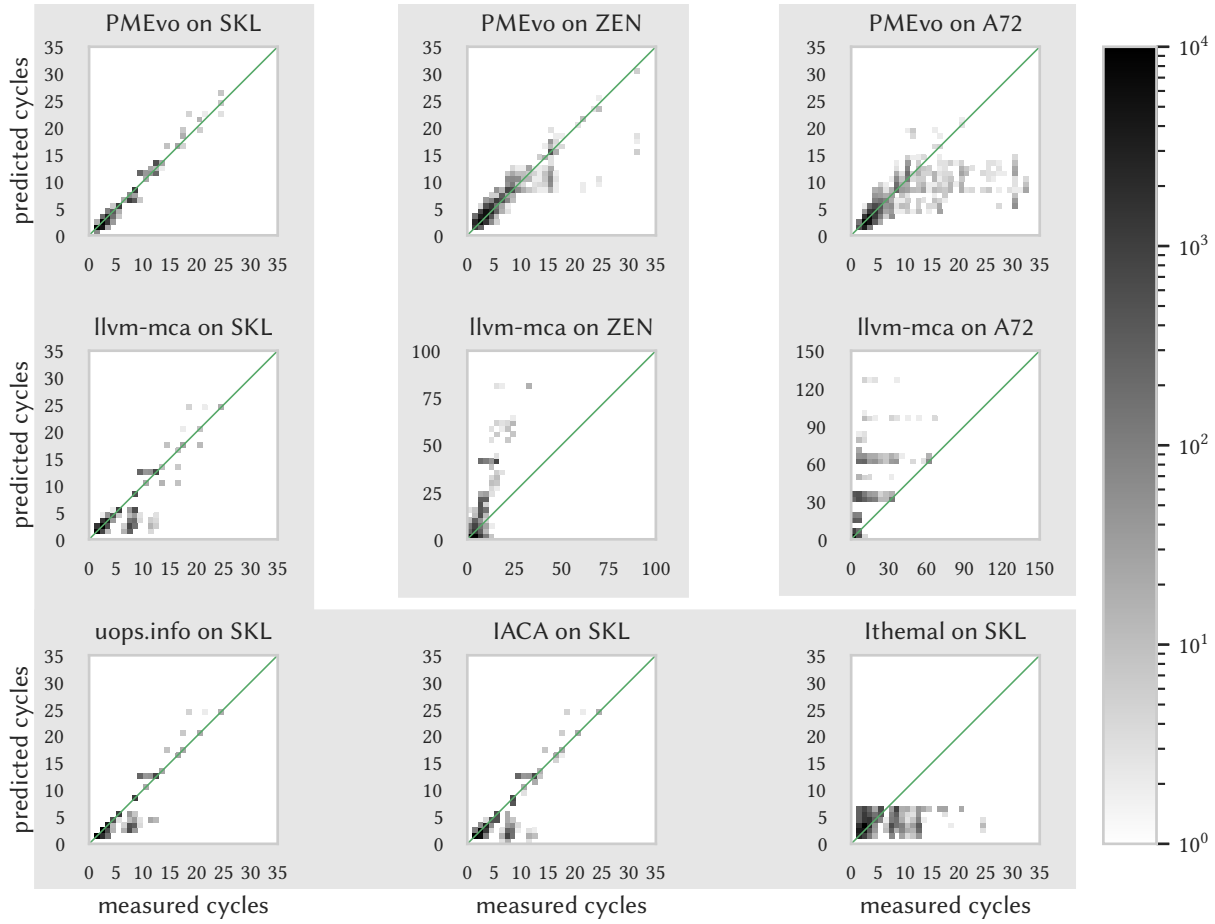

Figure 7. Prediction accuracy on *port-mapping-bound experiments*. Each heat map relates predicted and measured throughput in cycles per experiment. Points closer to the diagonal line indicate better predictions. The gray boxes group heat maps for the same platform. The experiments were set up and measured as described in Section 4.2.

report the results of the tools for this input, divided by the number of experiments in the unrolled loop body.

The accuracy metrics for the five tools under comparison are listed in Table 3.

Table 3. Prediction accuracy measures for port-mapping-bound experiments on SKL

	MAPE	Pearson CC	Spearman CC
PMEvo	14.7%	0.98	0.85
uops.info	9.3%	0.92	0.88
IACA	8.0%	0.86	0.79
llvm-mca	9.7%	0.87	0.82
Ithemal	60.6%	0.35	0.54

IACA, llvm-mca, and uops.info all predict with an average error of less than 10% with high correlation values. This impression is confirmed by the corresponding heat maps in Figure 7: Most of the experiments are close to the ideal line. They also all show a cluster of experiments below the diagonal line. These can be attributed to the family of bit test instructions (BTx), for which the measurable throughput does not agree with the throughput implied by the port usage as confirmed by the measurements of Abel and Reineke [1].

Our approach, PMEvo, has a slightly higher relative error than IACA, llvm-mca, and and uops.info, but comparable correlation coefficients. The corresponding heat map in Figure 7 shows a distribution close to the diagonal line. The BTx instructions that caused inaccuracies for the other approaches have a representation as multiple μops that map to the same ports. While differing from the real port mapping, this fits better to the observable throughputs.

For Ithemal, we observe lower correlations and a high error rate. This differs from the evaluation by Mendis et al. [20] where Ithemal exhibits superior results in these metrics in comparison to IACA.[6] As already noted, the difference in

[6]Their findings for the accuracy of IACA are consistent with the ones presented here.

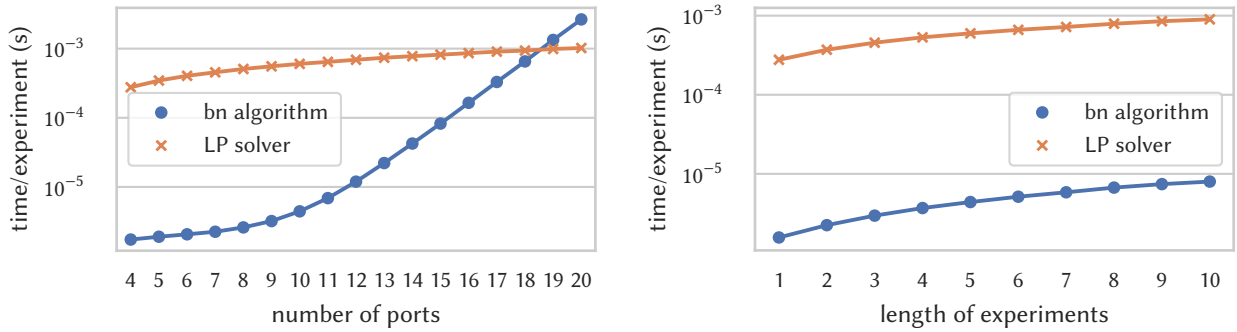

Figure 8. Execution time comparison of the bottleneck simulation algorithm and the LP solver with varying port numbers with experiments of length 4 (a) and with varying length of experiments with 10 ports (b). Both have their vertical axis in a logarithmic scale.

Table 4. Prediction accuracy measures for port-mapping-bound experiments on ZEN and A72

	MAPE	Pearson CC	Spearman CC
PMEvo (ZEN)	13.5%	0.94	0.87
llvm-mca (ZEN)	50.8%	0.86	0.54
PMEvo (A72)	21.4%	0.68	0.77
llvm-mca (A72)	65.3%	0.67	0.68

performance is likely a consequence of the different characteristics of the experiments used here and in the experimental evaluation of their paper: Ithemal is trained and validated on basic blocks emitted from a compiler for entire programs, which exhibit substantially more data dependencies than our experiments.

However, an appropriate interpretation of these results needs to be judicious: A high prediction accuracy for our experiments could have indicated a generalization of Ithemal to dependency-free code. Yet, the observed low prediction accuracy for our inputs does not allow conclusions about Ithemal's performance across real-world programs.

5.3.2 ZEN and A72. For the AMD and ARM microarchitectures, we compare PMEvo's results only to llvm-mca since the other approaches are only available for Intel architectures.

The metrics for both architectures in Table 4 show a common trend: PMEvo exhibits a considerably smaller prediction error than llvm-mca.

For ZEN, PMEvo inferred a port mapping that predicts with close to equal accuracy as its SKL mapping. With 21.4%, the prediction error of the PMEvo mapping for A72 is notably higher while correlations are lower. This observation is confirmed by the corresponding heat maps in Figure 7. PMEvo on A72 is prone to under-estimating experiments with longer running times. We attribute this to A72's less advanced out-of-order execution engine (according to the respective optimization guides [2, 4, 16]), which renders the experiments less representative for the port mapping.

In contrast to its results for SKL, llvm-mca has substantially larger prediction errors. The heat maps indicate a significant over-estimation of the throughput. One possible explanation is that these architectures are less in the focus of the developers than SKL and the respective port mapping models might not yet be as elaborate and accurate as the one for SKL. Especially for these two architectures, the models derived with PMEvo may significantly increase the accuracy of llvm-mca's throughput prediction.

5.4 Performance of the Simulation Algorithm

This section explores the performance behavior of the bottleneck simulation algorithm as presented in Section 4.5. For this purpose, we compare our optimized implementation of the bottleneck simulation algorithm to a realization of the linear program from Section 3.2 in the state-of-the-art LP solver Gurobi [14] (version 7.5.2). The running times reported for the LP version include model construction via the Gurobi C++ API as well as the actual solving.

There are two significant parameters that influence the execution time of both simulation methods: the number of ports in the microarchitecture and the length of the experiments.[7] We evaluate these parameters with randomly generated microarchitectures with the appropriate number of ports for an artificial instruction set architecture of 100 instructions. For each (number of ports, length of experiments) configuration, 128 randomly sampled experiments were simulated with each of 8 randomly sampled three-level mappings. The resulting seconds per experiment value for each pair of experiment and mapping is the arithmetic mean over 1000

[7] The number of instructions in the instruction set architecture is not relevant, since both implementations ignore instructions that do not occur in the experiment.

simulations. The points in the graph mark the median of these values for each (number of ports, length of experiments) configuration.

Influence of the Number of Ports. Figure 8a shows the results for experiments consisting of 4 instructions with a varying number of ports. For port numbers up to 10 as they occur in contemporary platforms, the bottleneck simulation algorithm outperforms the linear program by two orders of magnitude. Starting from 12 ports, the simulation time with the bottleneck simulation algorithm rises with a stronger incline. The bottleneck simulation algorithm reaches the simulation time of the LP implementation at about 18 ports. With the same inputs, the simulation time via the LP solver grows substantially slower with the number of ports. We conclude that the exponential run-time behavior of the bottleneck simulation algorithm, as explained in Section 4.5, has a negligible impact for inputs of interest.

Influence of the Length of Experiments. The experiments we use for the evolutionary algorithm are of very limited length to allow reliable execution on actual processors. Nevertheless, exploring the behavior with different lengths of experiments is worthwhile for the discussion of the bottleneck simulation algorithm. The results for varying lengths of experiments in an architecture with 10 ports are displayed in Figure 8b. Here, the bottleneck simulation algorithm consistently outperforms the LP solver by two orders of magnitude. The execution time for both methods grows sub-exponentially with the length of experiments, with an almost identical incline in the log-scale plot. This indicates that the rate at which the execution time rises with growing experiment length for the LP solver is considerably higher than for the bottleneck simulation algorithm.

6 Related Work

We divide related work into two categories: Approaches to find port mappings and work on predicting instruction throughput.

6.1 Inferring Port Mappings from Experiments

The instruction tables by Fog [12] used to be the only available source for experimentally validated information on instruction latency, throughput, and port usage. They are obtained with hand-crafted microbenchmarks that use hardware performance counters to count the number of executed cycles and the number of executed μops per port. Abel and Reineke [1] show that the reported port usage by Fog is only an under-approximation of the usable ports.

For the case that these counters are not available, Fog uses experiments that execute instructions with unknown port usage together with instructions whose port usage is known from some other resource. Observing the running time allows to identify interfering instruction combinations.

The tables include such information for a wide range of x86 microarchitectures by Intel, AMD, and VIA. The requirement to construct suitable microbenchmarks for each microarchitecture makes this approach very work-intensive.

Abel and Reineke [1] automated the process of designing microbenchmarks to measure latency, throughput, and port usage. Their algorithm to estimate port usage overcomes the imprecision of Fog's approach by using blocking instructions. The processor decomposes these instructions each into a single μop that can only be executed on a known set of ports. When executing the instruction under test with a sufficient number of blocking instructions to fully saturate a set P of ports, μops of the instruction under test will be executed on ports not in P if possible. For observing this as well as for identifying blocking instructions, they use per-port hardware performance counters as they are used by Fog [12]. While they provide throughput and latency measurements for x86 microarchitectures by Intel and AMD, they only give port mappings for the Intel platforms as only these provide all required performance counters.

Two further approaches initiated by Google are collected under the name EXEgesis. One is the EXEgesis project [13] that extracts latencies, throughputs, and port usage for Intel architectures from vendor-provided documentation. This requires automatically parsing documents that were intended for human readers: a fragile and work-intensive process. Since the provided documentation does not include all relevant information, the EXEgesis developers also created tools to infer the missing information via experiments. This led to the second project under this name, llvm-exegesis [9], a tool inside the LLVM framework [18] that automatically generates benchmarks similar to those used by Fog [12]. For measuring port usage, llvm-exegesis depends on per-port performance counters just as the two previously discussed approaches.

All of these works compare to ours in a similar way: Since they use precise hardware performance counters, they can obtain more accurate port mappings than our approach. However, our approach does not suffer from the restriction to platforms that have these performance counters, allowing us to automatically infer port mappings for x86 platforms by AMD, as well as for ARM platforms.

6.2 Work on Instruction Throughput Prediction

As port mappings are commonly used for throughput prediction, it is instructive to set the presented results in context to work from this field.

The Intel Architecture Code Analyzer (IACA) [17] models the execution of a sequence of instructions, considering factors such as port usage, operand dependencies, and instruction decoding bottlenecks. The output of IACA for a given instruction sequence includes a throughput estimation, a bottleneck resource, and the distribution of μops to ports. It is a closed-source tool provided by the processor

manufacturer Intel for some of its microarchitectures. As a consequence, IACA can make use of unpublished internal information to achieve an accurate performance prediction. Nevertheless, previous research (e.g. [1]) has shown cases where the prediction of IACA differs from the observable behavior. Since April 2019, IACA is no longer under active development.

OSACA [19] is an attempt to provide the same features as IACA, but with a non-proprietary system. They use information from port mappings for their supported architectures, a range of Intel microarchitectures as well as AMD's Zen architecture. These port mappings are extracted from sources like the tables by Fog [12] and material provided by the manufacturers. They implement means of experimentally validating this known port model via experiments, noting that experiments with multiple different instructions can uncover new details of the port mapping. Our approach systematically extends this line of work to derive new port mappings.

The llvm-mca tool [7] is also inspired by IACA. It uses knowledge from the LLVM [18] instruction scheduling models, including port usage if available, for performance prediction. These scheduling models are the result of human fine-tuning effort, proprietary knowledge contributed by processor designers, and experiments via llvm-exegesis [9].

Both, llvm-mca and OSACA, can benefit from port mappings by PMEvo for microarchitectures without available port mapping.

Ithemal [20] uses machine learning techniques for instruction throughput prediction. Similar to our approach, it only needs a specification of the instruction set architecture under test and a set of experiments labeled with measured throughputs as an input. These labeled inputs are used as training data for a hierarchical recurrent neural network based on long short-term memory (LSTM) cells.

Ithemal is trained and validated on basic blocks that are extracted from compiled benchmark programs. As a result, Ithemal captures different aspects than our approach: PMEvo focuses on experiments whose outcome is solely determined by the port mapping whereas the predictions of Ithemal are shaped by other factors such as data dependencies.

A drawback of the Ithemal approach is that the resulting processor model can only be interpreted by evaluating it on an instruction sequence. This is sufficient for certain applications like stochastic superoptimization [22]. However, in applications like the backend of an optimizing compiler, enumerating and evaluating a large set of possible instruction sequences is prohibitively expensive. A compact port mapping is more easily interpreted for constructing well-performing instruction sequences as it clearly indicates which instructions have conflicting resource requirements.

7 Conclusion

This paper presents PMEvo, a framework to infer port mappings, i.e. compact and interpretable representations of a modern processor's ability to exploit instruction-level parallelism. The inference is done by an evolutionary algorithm that optimizes port mappings to explain the instruction throughputs measured for specifically designed instruction sequences. Using a novel bottleneck simulation algorithm to evaluate the fitness of port mappings, PMEvo can explore the large search space of possible mappings effectively.

We demonstrate PMEvo's portability by inferring port mappings for three different microarchitectures, two of which are out of scope for previous automatic approaches. The high prediction accuracy of the inferred port mappings shows that PMEvo can make performance engineering tools more reliable for a wide range of hardware platforms.

A Correctness of the Bottleneck Simulation Algorithm

The proof of correctness of the bottleneck simulation algorithm presented here uses basic results from linear optimization theory. For background and proofs on these results, we refer to the textbook by Bertsimas and Tsitsiklis [6].

Let $S(m, e)$ be defined as follows:

$$S(m,e) := \left\{ \frac{\sum\{e(i) \mid Ports(m,i) \subseteq Q\}}{|Q|} \;\middle|\; Q \subseteq P \right\}$$

With this notation, Equation 1 can be written as $\hat{t}_m(e) = \max S(m,e)$. The proof proceeds by showing that the result $\hat{t}_m(e)$ of the bottleneck simulation algorithm is equal to the throughput $t^*_m(e)$ according to Definition 3 for any experiment e and any (two-level) port mapping $m := (\mathbf{I} \cup \mathbf{P}, M)$. We do so by showing that (I) $t^*_m(e)$ is included in $S(m,e)$ and that (II) each element of $S(m,e)$ is upper-bounded by $t^*_m(e)$.

I. Let s be an optimal feasible solution of the linear program. We denote the value of a variable x chosen in s by $s[x]$. Since s is optimal, there is a non-empty maximal set $Q \subseteq P$ such that for all $k \in Q$ holds that

$$\sum_{i \in \mathbf{I}} s[x_{ik}] = s[t] = t^*_m(e) \tag{2}$$

Without loss of generality, we assume that each instruction that s executes on a port in Q can only be executed on ports in Q, that is:

$$k \in Q \wedge s[x_{ik}] > 0 \Rightarrow Ports(m,i) \subseteq Q \tag{3}$$

If this is not the case for s, we can find a different solution s' with identical objective value that fulfills this constraint as follows: For every (i,k) such that $s[x_{ik}] > 0$, $\sum_{i \in \mathbf{I}} s[x_{ik}] = s[t]$, and $Q' := Ports(m,i) \cap (P \backslash Q) \neq \emptyset$, we remove a sufficiently small part of the value for x_{ik} and add

it to the value of some $x_{ik'}$ with $k' \in Q'$ such that constraint (B) is tight for neither of k and k'.[8]

By defining $J := \{i \mid Ports(m,i) \subseteq Q\}$, we identify the following equalities:

$$\sum_{i\in J} e(i) \overset{(A)}{=} \sum_{i\in J}\sum_{k\in\mathbf{P}} s[x_{ik}] \overset{(D)}{=} \sum_{i\in J}\sum_{k\in Q} s[x_{ik}] = \sum_{k\in Q}\sum_{i\in J} s[x_{ik}]$$
$$\overset{(3)}{=} \sum_{k\in Q}\sum_{i\in\mathbf{I}} s[x_{ik}] \overset{(2)}{=} \sum_{k\in Q} s[t] = t_m^*(e)\cdot|Q|$$

The equality of the leftmost term and the rightmost term proves that $t_m^*(e) \in S(m,e)$.

II. Let $Q' \subseteq \mathbf{P}$ and $t' := \sum\{e(i) \mid Ports(m,i) \subseteq Q\}/|Q|$. We assume $t' > t_m^*(e)$ and show that this leads to a contradiction, proving that $t_m^*(e)$ is an upper bound to each element of $S(m,e)$.

For this argument, we form the dual of the linear program:

$$\begin{aligned}
\text{maximize} \quad & \sum_{i\in\mathbf{I}} e(i)\cdot y_i & \\
\text{subject to} \quad & y_i - z_k \le \overline{m}_{ik} & \text{for all } i\in\mathbf{I}, k\in\mathbf{P}\\
& \sum_{k\in\mathbf{P}} z_k = 1 & \\
& z_k \ge 0 & \text{for all } k\in\mathbf{P}\\
& y_i \ge 0 & \text{for all } i\in\mathbf{I}
\end{aligned}$$

Here, the y_i and z_k are real-valued variables and $\overline{m}_{ik} = 1 \Leftrightarrow (i,k) \notin M$.

By the strong duality theorem for linear programs, an optimal solution for this dual linear program has the same objective $t_m^*(e)$ as an optimal solution for the primal linear program.

Given the assumption that $t' > t_m^*(e)$, we construct a solution s' for the dual with a higher objective value, which contradicts the strong duality theorem or the optimality of $t_m^*(e)$. The construction of s' is as follows for each $i \in \mathbf{I}$ and $k \in \mathbf{P}$:

$$\begin{aligned}
s'[z_k] &= 1/|Q'| && \text{if } k \in Q'\\
s'[y_i] &= 1/|Q'| && \text{if } Ports(m,i) \subseteq Q'
\end{aligned}$$

All other variables are set to 0. This solution fulfills all constraints and has the following objective value:

$$\sum_{i\in\mathbf{I}} e(i)\cdot y_i = \frac{\sum\{e(i) \mid Ports(m,i)\subseteq Q\}}{|Q|} = t' > t_m^*(e)$$

This proves the correctness of the bottleneck simulation algorithm. □

[8] If this was possible for all $k \in Q$, s could not be optimal.

References

[1] Andreas Abel and Jan Reineke. 2019. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. ACM, New York, NY, USA, 673–686. https://doi.org/10.1145/3297858.3304062

[2] AMD. 2017. *Software Optimization Guide for AMD Family 17h Processors.* Chapter 2, 17–44.

[3] AMD. 2019. *Processor Programming Reference for AMD Family 17h Models 01h,08h, Revision B2 Processors.* Chapter 2.1.15, 160–176.

[4] ARM. 2015. *Cortex®-A72 Software Optimization Guide.* Chapter 2, 6–7.

[5] ARM. 2016. *ARM® Cortex®-A72 MPCore Processor Technical Reference Manual (Revision r0p3).* Chapter 11.8, 428–431.

[6] Dimitris Bertsimas and John Tsitsiklis. 1997. *Introduction to Linear Optimization* (1st ed.). Athena Scientific.

[7] Andrea Di Biagio. 2018. llvm-mca: a static performance analysis tool. http://lists.llvm.org/pipermail/llvm-dev/2018-March/121490.html

[8] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering* (Berlin, Germany) *(ICPE '18)*. ACM, New York, NY, USA, 41–42. https://doi.org/10.1145/3185768.3185771

[9] Guillaume Chatelet. 2018. llvm-exegesis: Automatic Measurement of Instruction Latency/Uops. https://lists.llvm.org/pipermail/llvm-dev/2018-March/121814.html

[10] Yishen Chen, Ajay Brahmakshatriya, Charith Mendis, Alex Renda, Eric Atkinson, Ondrej Sỳkora, Saman Amarasinghe, and Michael Carbin. 2019. BHive: A Benchmark Suite and Measurement Framework for Validating x86-64 Basic Block Performance Models. In *2019 IEEE international symposium on workload characterization (IISWC). IEEE.*

[11] Kenneth A De Jong. 2006. *Evolutionary Computation: A Unified Approach.* MIT Press.

[12] Agner Fog. 2018. 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

[13] Google. 2018. EXEgesis: Automatic Measurement of Instruction Latency/Uops. https://github.com/google/EXEgesis

[14] LLC Gurobi Optimization. 2019. Gurobi Optimizer Reference Manual. http://www.gurobi.com

[15] John L Hennessy and David A Patterson. 2011. *Computer Architecture: A Quantitative Approach.* Elsevier.

[16] Intel. 2019. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* Chapter 2.2, 2–6–2–12.

[17] Intel. [n.d.]. Intel Architecture Code Analyzer. https://software.intel.com/en-us/articles/intel-architecture-code-analyzer

[18] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04).* Palo Alto, California.

[19] Jan Laukemann, Julian Hammer, Johannes Hofmann, Georg Hager, and Gerhard Wellein. 2018. 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).* IEEE, 121–131.

[20] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. 2019. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. PMLR, Long Beach, California, USA, 4505–4515.

[21] Kaisa Miettinen. 1999. *Nonlinear Multiobjective Optimization.* Vol. 12. Springer Science & Business Media.

[22] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. ACM, New York, NY, USA, 305–316. https://doi.org/10.1145/2451116.2451150

[23] Robert M Tomasulo. 1967. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development* 11, 1 (1967), 25–33.