PALMED: Throughput Characterization for Any Architecture

Anonymous Author(s)

Abstract

This paper describes PALMED, a tool that automatically builds a resource mapping, a performance model for pipelined, super-scalar, out-of-order CPU architectures. Resource mappings describe the execution of a program by assigning instructions in the program to abstract resources. They can be used to predict the throughput of basic blocks or as a machine model for the backend of an optimizing compiler.

PALMED does not require hardware performance counters, and relies solely on runtime measurements to construct resource mappings. This allows it to model not only execution port usage, but also other limiting resources, such as the frontend or the reorder buffer. Also, thanks to a dual representation of resource mappings, our algorithm for constructing mappings scales to large instruction sets, like that of x86.

We evaluate the algorithmic contribution of the paper in two ways. First by showing that our approach can reverse engineering an accurate resource mapping from an idealistic performance model produced by an existing port-mapping. We also evaluate the pertinence of our dual representation, as opposed to the standard port-mapping, for throughput modeling by extracting a representative set of basic-blocks from the compiled binaries of the Spec CPU 2017 benchmarks [4] and comparing the throughput predicted by existing machine models to that produced by PALMED.

Keywords: abstract simulation, sensitivity analysis, performance feedback, performance bottleneck, QEMU

1 Introduction

Performance modeling is a critical component for program optimizations, assisting compilers as well as developers in predicting the performance of code variations ahead of time. Performance models can be obtained through different approaches that span from precise and complex simulation of a hardware description [18, 19, 32] to application level analytical formulations [13, 31]. An interesting approach for modeling the CPU of modern pipelined, super-scalar, out-of-order processors trades simulation time with accuracy by separately characterizing both latency and throughput of instructions. This approach is suitable both for optimizing compilers [16, 23], but also for hand-tuning critical kernels written in assembler [11, 33]. It is used by performance-analysis tools such as CQA [25], Intel IACA [14], OSACA [17], MIAMI [20] or llvm-mca [28]. Cycle-approximate simulators such as ZSim [27] or MCsimA+ [3] can also take advantage of such an instruction characterization.

This motivated several projects to extract information from available documentation [5, 17]. But the documentation or commercial CPUs, when available, is often vague or outright missing information. Intel's processor manual [7], for example, does not describe all the instructions implemented by Intel cores, and for the instructions that are covered, it does not even provide the decomposition of individual instructions into micro operations (μOPs), nor the execution ports that these μOPs can use. Another line of work that allows more exhaustive and precise instruction characterization is based on micro-benchmarks such as those developed to characterize the memory hierarchy [6]. While characterizing the latency of instructions is quite easy [10, 12, 15], characterizing the throughput is more challenging. Indeed, on super-scalar processors, the throughput of a combination of instructions cannot be simply derived from the throughput of the individual instructions. This is because instructions compete for CPU resources, such as functional units, or execution ports, which can prevent them from executing in parallel. It is thus necessary to not only characterize the throughput of each individual instruction, but also to come up with a description of available resources and the way they are shared. The most natural way to express this sharing is through a port mapping, a tripartite graph that describes how instructions decompose to μOPs and assigns μOPs to execution ports (see Fig 2a). The goal of existing work has been to reverse-engineer such a port mapping for different CPU architectures. The first level of this mapping, from instructions to μOPs, is conjunctive, i.e., a given instruction decomposes into one or more of each of the μOPs it maps to. The second level of this mapping on the other hand is disjunctive, i.e., a μOP can choose to execute on any one of the ports maps to. Even with hardware counters that provide the number of μOPs executed per cycle and the usage of each individual port, creating such a mapping is quite challenging and requires a lot of manual effort with ad hoc solutions to handle all the cases specific to each architecture [2, 10, 12, 25].

PLDI'21, June 20 - 25, 2021, Virtual Conference

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Such approaches, while quite powerful, allowing a semi-automatic characterization of basic-block throughput, suffer from several limitations: First of all, they assumes that the architecture provides the required hardware counters; Second, they only allow modeling the throughput bottlenecks associated with port usage, and neglect other resources, such as the front-end or reorder buffer. In other words, it provides a performance model of an ideal architecture that does not necessarily fully match reality. To overcome these limitations we limit ourselves to only using cycle measurements when building our performance model. Not relying on specialized hardware performance counters may make the initial model construction more complicated, but in exchange our approach is able to model resource not covered by hardware counters with relative ease. This also makes it significantly easier to port our modeling technique to new CPU architectures.

One of the main challenges in this approach is to generate a set of micro-benchmarks that allows capturing all the possible resource sharing. Unfortunately, to be exhaustive, and in the absence of structural properties, this set is combinatorial. A simple way to reducing the set of required micro-benchmarks chosen by existing approaches [21, 24] is to reduce the set of modeled instructions to those that are emitted by compilers. Another natural strategy followed by Ithemal [21] is to build micro-benchmarks from the "most executed" basic-blocks of some representative benchmarks. A third strategy, used by PMEvo [24], is to have kernels that contain repetitions of two different instructions. Our solution is constructive and follows several steps that allows building a non-combinatorial number of micro-benchmarks that stresses the usage of each individual resource thus allowing to characterize the resource usage of *all* instructions. The second main challenge addressed by PMEvo is to build an interpretable model, that is, a resource-mapping that can be used by a compiler or a performance debugging tool, not a black-box that just predicts the throughput of a micro-kernel. The issue with the standard port-mapping, as used in [2, 17, 28], is that computing the throughput of a set of instructions requires the resolution of a flow problem. That is, given a set of micro-benchmarks, finding a mapping that best expresses the corresponding observed performances requires solving a multi-resolution linear optimization problem. This linear problem also does not scale to larger sets of benchmarks, even when restricting the micro-benchmarks to only contain up to two different instructions. PMEvo addressed this issue by using a evolutionary algorithm that approximates the result. Our approach, on the other hand, is based on a crucial observation that a dual representation exists for which computing the throughput is not a linear problem but a simple formula instead. While it takes several hours to solve the original disjunctive-port-mapping formulation, only a few minutes suffice for the corresponding conjunctive-resource-mapping formulation.

The first contribution of this paper is to provide a less intricate two-level view, that can be constructed quicker than previous works. Instead of representing the execution flow as the traditional three-level "instructions decomposed as micro-operations (micro-ops) executed by ports" model, we opt for a direct "instruction use abstract resources" model. Whereas an instruction is transformed into several micro-ops which in turn *may* be executed by different compute units; our bipartite model *strictly uses* every resources mapped to the instructions. In other words, the *or* in the mapping graph are replaced with *and*, which greatly simplifies throughput estimation. This representation may easily represents other bottlenecks such as the instruction decoder or the re-order buffer as other abstract resources. Note that this corresponds to the user view, where the micro-ops and their executions path are kept hidden inside the processor. The second contribution is to provide a constructive algorithm that provides a non-combinatorial set of representative micro-benchmarks that can be used to characterize all instructions of the architecture.

This paper has the following structure. In section 2, we present how our mapping differs from those in previous works. Section 3 gathers the formal definitions and proves the equivalence between our model and the three-level mapping currently in use. In section 4, we propose an architecture-agnostic approach in order to deduce the abstract mapping without the use of any performance counters but the one counting CPU cycles. Finally, section 5 evaluates the quality of our approach in the following two ways. First, by experimentally constructing a model following our bipartite structure from an existing three level one. Second, by estimating the execution time of micro-kernels extracted from well-know compute benchmarks.

2 Background

In this work, we consider a CPU as a complex device mainly described by the so-called "port model": Here, instructions are first fetched from memory, then decomposed into one or more *micro-operations*, also called *μOPs*. The CPU then schedules these μOPs on an available execution port, that performs the real operation. Even if some instructions such as `add %rax, %rax` translate into only a single μOP, the x86 instruction set also contains more complex instructions that translate into multiple μOPs. For example, the `wbinvd` (*Write Back and Invalidate Cache*) instruction produces as many μOPs as needed to flush every line of the cache. In practice, due to the large caches used in modern CPUs, it produces thousands of μOPs [2].

Even though execution ports play a major role in CPU performance, they are not the only source of performance bottlenecks. Indeed, throughout their execution, the μOPs travel through numerous stages, all of them being able to produce performance anomalies. After the decode stage, the

μOPs are placed in μOP queue, which can also be fed by the μOP cache. Next, the μOPs are sent to the reorder buffer, which marks the conceptual end of the CPU front-end and beginning of the backend. The *scheduler* then selects instructions and assigns them to an *execution port* with the required compute capabilities. When several instructions are assigned to the same port, the later ones are gathered in a *reservation station* and wait for their operands to be computed. Note that the operands may not correspond to actual registers or exact memory locations, as several layers of caching and register renaming may occur. When the execution port finally becomes available, the μOP may start its execution. This operation is called *issuing* an instruction, and is *out-of-order*, that is, the μOPs may be executed in a different order than the program order. For example, if a μOP is waiting for a piece of data to arrive from the RAM, the entire pipeline does not stall, and other independent instructions can be executing to hide the memory latency.

Execution ports are hardware controllers which have different functional capabilities, as they are wired to one or more *execution units*: for example, on the Skylake architecture (see Fig. 1), only port 4 may store data; and the store address must have previously been computed by an *Address Generation Unit*, available on ports 2, 3 and 7. Once executed, the instructions wait back in the reorder buffer until all preceding instructions have finished, they then write their results back to the register file and are finally discarded. On x96, except for the divider, all units are fully pipelined, meaning that they can execute one μOP per cycle. Nevertheless, this does not imply that the results of an operation are available in one cycle, as μOPs may require multiple cycles to complete.

The *latency* of an instruction is the number of clock cycles necessary between two dependent computations. For an instruction I, its latency can be experimentally measured by creating a micro-benchmark that executes a long chain of instances of I where each instance depends on the result of the preceding one. For example, assuming a 2-address mode and registers named %Ri:

```
repeat many times:
  I %R0, %R0
  I %R0, %R0
  I %R0, %R0
  ...
```

The *throughput* of an instruction is the maximum number of instances of that instructions that can be executed in parallel per cycle. For an instruction I, the throughput of I can be experimentally measured by creating a micro-benchmark that executes many non-dependent instances of I:

```
repeat many times:
  I %R0, %R0
  I %R1, %R1
  I %R2, %R2
  ...
```

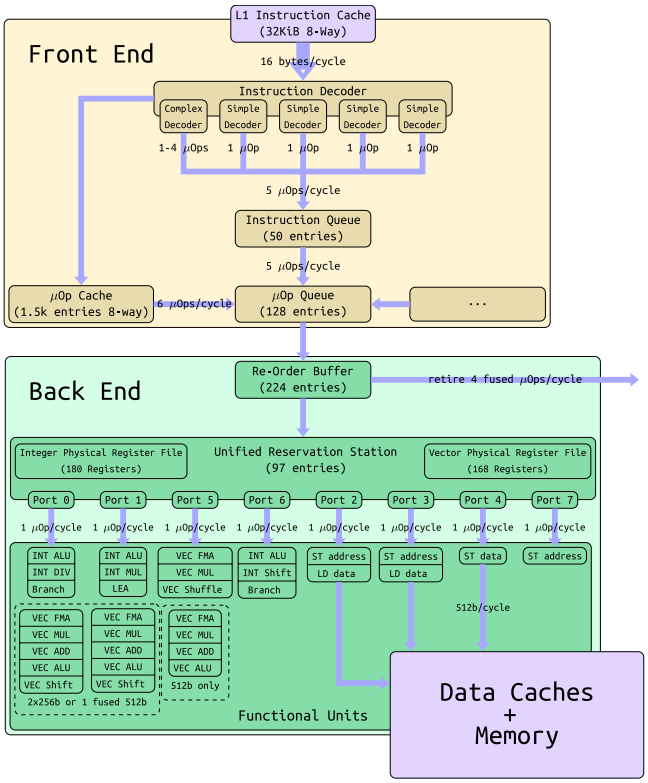

Figure 1. Intel's Skylake microarchitecture, compiled from marketing presentations and the official documentation

The combined throughput of a multiset[1] of instructions can be defined similarly. For example, the throughput of $\{I_1^2, I_2\}$, i.e. two instances of I_1 and one instance of I_2, is equal to the number of instructions executed per cycle (IPC) by the micro-benchmark:

```
repeat many times:
  I1 %R0, %R0
  I1 %R1, %R1
  I2 %R2, %R2
  ...
```

Note that PALMED only uses benchmarks that have no dependencies, that is, where all instructions can execute in parallel. Consequently the order of instructions in the benchmark does not matter [2]

A *resource-mapping* describes the resources used by each instructions in a way that can be used to derive the throughput for any multiset of instructions, without having to execute the corresponding micro-benchmark. As mentioned earlier, the standard way of representing a resource mapping is with a tripartite port-mapping as illustrated in Fig 2a. In this example: instruction I_1 decomposes into a single μOP

[1] A multiset is a set that can contain multiple instances of an element. Like with normal sets, the order of elements is not relevant

[2] We assume, like all related work we are aware of, that the CPU scheduler is able to optimally schedule these simple kernels.

v_1 that itself has a single port r_1 on which it can be issued; as for instruction I_2, it also decomposes into a single μOP v_2 that can be issued to either one of two ports, r_1 or r_2. Hence, I_1 has a throughput of one, meaning only one instruction can be issued per cycle. I_2, on the other hand has a throughput of two, meaning the two ports can be used in parallel by two different instances of I_2. The throughput of the set $\{I_1^2, I_2\}$, more compactly denoted by $I_1^2 I_2$, is determined by resource r_1 which is already saturated by I_1 alone; While I_2 could be both assigned to r_1 or to r_2, it will go to r_2. It will thus take two cycles to execute three instructions, two instances of I_1 and one instance of I_2. Hence, a throughput of $3/2 = 1.5$ instruction per cycle. Note that even if we unroll the benchmark to increase the number of instructions that could potentially execute in parallel, the IPC does not improve, since it is limited by the bottleneck in r_1.

The dual representation, advocated in this paper, corresponds to the conjunctive bipartite resource mapping as illustrated in Fig. 2b. Here, instruction I_1 uses two resources r_1, with a of throughput one, and r_{12}, with a throughput of two. To re-iterate, I_1 does not choose one the two resources to execute on, but use them both simultaneously. Instruction I_2 only uses the single resource r_{12}. The conjunctive form makes it straightforward to compute the set of resources used by the multiset $I_1^2 I_2$. r_1, with a throughput of one, is used twice by I_1, which requires two cycles. While r_{12}, with a throughput of two, is used twice by I_1 and once by I_2, requiring $3/2 = 1.5$ cycles. The bottleneck resource is thus r_1. Overall, we execute three instructions in two cycles, leading to an IPC of $3/2 = 1.5$, which is the same result produced by the tripartite model.

The following section provides the formalism that allows proving the equivalence between the bipartite and tripartite representations.

3 The bipartite resource mapping

The goal of this section is to prove the equivalence between the standard tripartite graph representation of a port mapping and the bipartite conjunctive mapping between μOPs and abstract resources promoted by this paper. This section focuses on the lower layer, from μOPs to ports/resources. To simplify the notation, this part assumes that each instruction is composed of a single μOP.

3.1 Primary definitions

Definition 3.1 (Microkernel). *A microkernel K is an* infinite loop *made up of a finite multiset of instructions, $K = I_1^{\sigma_{K,1}} I_2^{\sigma_{K,2}} \cdots I_m^{\sigma_{K,m}}$ without dependencies between instructions. The number of instructions executed during one loop iteration is $|K| = \sum_i \sigma_{K,i}$.*

Definition 3.2 (Disjunctive port mapping). *A disjunctive port mapping is a bipartite graph $(V, \mathcal{R}, E)$ where: V represents the set of μOPs; $\mathcal{R}$ represents the set of resources (corresponding to execution ports in a real-world CPU); $E \subset V \times \mathcal{R}$ expresses the possible mappings from μOPs to ports. In this original form each port $r \in \mathcal{R}$ has a throughput $\rho(r)$ of 1.*

Let $K = I_1^{\sigma_{K,1}} I_2^{\sigma_{K,2}} \cdots I_m^{\sigma_{K,m}}$ be a microkernel where each instruction is composed of a single μOP v_i.

A valid assignment represents the choice of which resources to associate with a given instance of an instruction. However, this choice might change between iterations. Thus, we represent the valid assignment as a mapping $p : \mathcal{I} \times \mathcal{R} \mapsto [0; 1]$ where $p_{i,r}$ corresponds to the frequency a given resource is chosen. We also define $R_i = \{r,\ p_{i,r} \neq 0\}$. This assignment is valid if:

$$\forall I_i \in K, \forall r \in R_i,\ (v_i, r) \in E$$

$$\forall I_i \in K,\ \sum_{r \in R_i} p_{i,r} = 1$$

The execution time *of an assignment $(p_{i,r})_{i,r}$, is:*

$$t_{end} = \max_{r \in \mathcal{R}} \sum_{i \in K} \sigma_{K,i} \cdot p_{i,r}$$

The minimal execution time over all valid assignments is denoted $t(K)$ (obtained using an optimal assignment*).*

Definition 3.3 (Conjunctive port mapping). *A conjunctive port mapping is a bipartite* weighted *graph $(I, \mathcal{R}, E, \rho_{I,\mathcal{R}})$ where: I represents the set of instructions; $\mathcal{R}$ represents the set of abstract resources; $E \subset I \times \mathcal{R}$ expresses the required use of abstract resources for each instruction;*

Each abstract resource $r \in \mathcal{R}$ has a throughput of 1; An instruction i that uses a resource r ($(i, r) \in E$) always uses the same proportion (number of cycles, possibly lower/greater than 1) $\rho_{i,r} \in \mathbb{Q}^+$; If i does not use r, then $\rho_{i,r} = 0$.

Let $K = I_1^{\sigma_{K,I_1}} I_2^{\sigma_{K,I_2}} \cdots I_m^{\sigma_{K,I_m}}$ be a microkernel. In a steady state execution of K, for each loop iteration, instruction i must use resource r ($\sigma_{K,i}\rho_{i,r}$) cycles. The number of cycles required to execute one loop iteration is:

$$t(K) = \max_{r \in \mathcal{R}} \left(\sum_{i \in K} \sigma_{K,i} \rho_{i,r} \right)$$

The throughput of K is:

$$\overline{K} = \frac{|K|}{t(K)} = \frac{\sum_{i \in K} \sigma_{K,i}}{\max_{r \in \mathcal{R}} \left(\sum_{i \in K} \sigma_{K,i} \rho_{i,r} \right)}$$

Definition 3.4 (∇-dual conjunctive port mapping). *Let $(V, \mathcal{R}, E)$ be a disjunctive port mapping. Let ∇ be a non-empty set of subsets of $\mathcal{R}$. We define its ∇-dual, a conjunctive port mapping, as $(V, \overline{\mathcal{R}}, \overline{E})$ such that:*

$$\overline{\mathcal{R}} = \left\{ \overline{r}_J,\ J \in \nabla \right\}$$
$$\overline{E} = \left\{ (v, \overline{r}_J) \text{ s.t. } \{r, (v, r) \in E\} \subseteq J \right\}$$
$$\rho(\overline{r}_J) = \textstyle\sum_{r_j \in J} \rho(r_j) = |J|$$

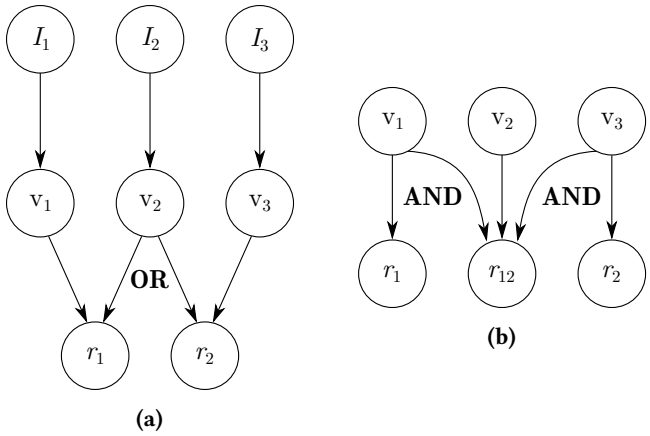

Figure 2. Example of resource mapping with three µOPs and two ports: (2a) the bottom part of this graph is a bipartite disjunctive and (2b) its bipartite conjunctive ∇-dual for $\nabla = \{\{r_1\}, \{r_2\}, \{r_1, r_2\}\}$.

Then, we can normalize this graph by adding weights to edges, and normalizing the resource throughput:

$$\rho^N_{i,\overline{r}_J} = \begin{cases} 1/\rho(\overline{r}_J) & \text{if } (i, \overline{r}_J) \in \overline{E} \\ 0 & \text{else} \end{cases}$$
$$\rho^N(\overline{r}_J) = 1$$

Example. Let p be a processor with two execution ports r_1 and r_2 and three instructions I_1, I_2 and I_3, each composed to one μOP v_1, v_2 and v_3, respectively. As illustrated in figure (2a) v_1 can be executed in one cycle on r_1, v_2 can be executed either on r_1 or r_2, v_3 can be executed on r_2. The corresponding conjunctive mapping has a combined resource r_{12} that is linked to the usage of either r_1 or r_2 by edges of weight 1/2. In this representation, every instruction using r_1 or r_2 also uses r_{12} for half the throughput, which leads to the graph illustrated in figure (2b).

3.2 Equivalence between disjunctive and conjunctive

Definition 3.5 (Saturated port set)**.** *Consider a microkernel K. Let $(p_{i,r})_{i,r}$ be a valid assignment of K for a disjunctive port mapping $(V, \mathcal{R}, E)$. The saturated port set $\mathcal{S}$ is defined as follow:*

$$\mathcal{S} = \left\{ r_s \text{ such that } t_{end} = \sum_{i \in K} \sigma_{K,i} \cdot p_{i,r} \right\}$$

Lemma 3.1 (Saturated set assumption)**.** *Let $(p_{i,r})_{i,r}$ be an valid assignment for a microkernel K in a disjunctive port mapping $(V, \mathcal{R}, E)$ and $\mathcal{S}$ its saturated set. If we have two resources r_s and r_t such that $(v, r_s) \in E \wedge (v, r_t) \in E$ and $r_s \in \mathcal{S}, r_i \notin \mathcal{S}$.*

Then, there exists a faster valid assignment for which both resources r_s and r_t are saturated.

Corollary 3.1 (Saturating assignment)**.** *Let us consider an optimal assignment $(p_{i,r})_{i,r}$ of a list of µOPs K on a disjunctive port mapping $(V, \mathcal{R}, E)$. For all $v \in V$ such that there are $(r_x, r_y) \in \mathcal{R}^2$ connected to v (i.e. $(v, r_x) \in E$ and $(v, r_y) \in E$). If $r_x \in \mathcal{S}$, then $r_y \in \mathcal{S}$. Thus:*

$$\forall i,\ [R_i \subset \mathcal{S} \Leftrightarrow \{r,\ (v_i, r) \in E\} \subset \mathcal{S}]$$

Theorem 3.1 (Equivalence of ∇-duality)**.** *Let K be a microkernel. Let $(V, \mathcal{R}, E)$ (with the set of resources $\mathcal{R}$ also denoted $\{r_j\}_j$), ∇ a set of subsets of $\mathcal{R}$, and $(V, \overline{\mathcal{R}}, \overline{E})$ (with the set of resources $\overline{\mathcal{R}}$ also denoted $\{\overline{r}_J\}_{J \in \nabla}$) be its ∇-dual.*

(i) Let $(p_{i,r})_{i,r}$ be a valid optimal assignment (i.e. of minimal execution time) of K with regard to $(V, \mathcal{R}, E)$. This assignment can be translated into its ∇-dual, with no change to its execution time. In other words, $\overline{t}(K) \leq t(K)$.

(ii) If ∇ is the set of all *subsets of $\mathcal{R}$ then $\overline{t}(K) = t(K)$.*

We have an equality if ∇ is the set of all subsets of $\mathcal{R}$. In theory, the size of this set is exponential in the number of resources. However, the proof shows that we can restrict ourselves to unions of saturated sets $\mathcal{S}$ of optimal assignments.

In practice, we build ∇ by first considering the abstract resources that directly correspond to the set of resources that a given μOP can be mapped to in the disjunctive mapping. Then, we recursively apply the following rule: if two abstract resources have a non-empty intersection, we then add their union as a new abstract resource. The intuition is that this new abstract resource introduces a new constraint on the valid assignment in the dual, corresponding to a potential saturation of these resources. We end up with a set containing less than 14 elements in our experiments.

4 Deducing the resource mapping from any CPU

Using a conjunctive resource mapping based throughput model instead of a disjunctive one has several advantages: 1. First, computing a throughput for a multiset of instructions from a disjunctive port-mapping requires the solving of a flow problem (usually expressed as an ILP [24] while a simple formula (IPC of the bottleneck resource which computation time is a simple sum) suffices for the conjunctive resource-mapping. 2. In a standard port mapping, using a tripartite conjunctive-disjunctive graph, the middle "hidden layer" has to be discovered, which greatly complicates the task of discovering the whole graph. the use of a dual conjunctive representation for the μOPs to resource mapping leads to a conjunctive-conjunctive tripartite graph that can be trivially collapsed into a conjunctive bipartite one. Discovering the bipartite conjunctive instruction to resource mapping of a CPU is clearly easier allowing the design of a constructive architecture-agnostic algorithm, which we will describe in this section.

Our approach can be decomposed into four different steps.

- Select the *basic instructions*, a subset of instructions that map to as few resources as possible;
- Compute the *core mapping* for these basic instructions. The core mapping stays fixed for the rest of the algorithm. Along with the core mapping we select a saturating microbenchmark for each resource, called the *saturating kernel*. The saturating kernel is made up of basic instructions that do not place a heavy load on other resources.
- Use the saturating kernels and the core mapping to deduce, one by one, the mapping for every remaining instruction of the targeted architecture.

The objective of the following algorithm is to create an appropriate set of microkernels; For each microkernel K, measure its throughput $\overline{K}$; For a fixed finite set of abstract resources, use operational research to deduce values for $\rho_{i,r}$ that best express the observed throughput for all the constructed microkernels; By setting the objective function of our linear program to minimize $\sum_{i,r} \rho_{i,r}$, the obtained bipartite graph is compact (minimal number of edges and resources) allowing fast (yet precise) and interpretable performance modeling.

4.1 Basic Instructions selection

The first step of our algorithm consists of trimming the instruction set to extract a minimal set of instructions for which the entire exact mapping will be computed. As this mapping will be used later, we want to be sure to have enough instructions to account for all resources, but the more instructions we have, the longer the resolution of the linear problem to find the core mapping will take. We thus first apply two simple filters that reduce the number of candidates for basic instructions.

- *Low-IPC*: if $\overline{a} < 1$ then a is ignored. The first filter simply ignores instructions with an IPC strictly less than 1: Assuming every physical resource to have a throughput of 1, such instructions use one unit more than once.
- *Equivalent classes*: if $\forall p,\ \overline{a^{\overline{a}} p^{\overline{p}}} = \overline{b^{\overline{b}} p^{\overline{p}}}$ then keep only a or b. The second filter removes duplicates, that is, if two instructions behave the same with regard to the evaluation used for our basic instruction selection, then one of them can be ignored. Obviously, on a real machine, despite all the crucial efforts to remove execution hazards, measured IPC never perfectly match and the correct criteria for selecting a representative instruction for duplicates should approximate the equality test $\forall p,\ \overline{a^{\overline{a}} p^{\overline{p}}} \approx \overline{b^{\overline{b}} p^{\overline{p}}}$. The construction of equivalence classes and associated representative instruction in this context simply uses a k-mean method on matrix $Q = \left(\overline{a^{\overline{a}} b^{\overline{b}}}\right)_{a,b}$.

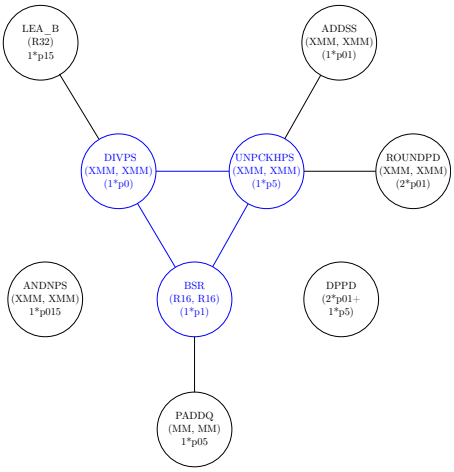

Figure 3. Disjoint graph between a few x86 instructions. Very basic instructions are defined as a maximal clique of disjoint instructions (in blue).

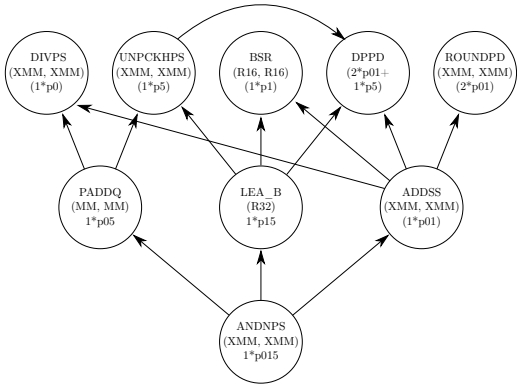

Figure 4. Frugality relation for a few x86 instructions. Basic instructions are defined as the n most frugal instructions. `ANDNPS`, `PADQ`, `LEA_B`, and `ADDSS` are the four more frugal instructions.

Once instructions with a low IPC and duplicates have been removed from the set of candidates, the selection uses two criteria:

- *Very basic instructions*: Instructions a and b are considered *disjoint* ($a \leftrightarrow b$) if $\overline{a^{\overline{a}} b^{\overline{b}}} = \overline{a} + \overline{b}$. The set of very basic instructions is defined as a maximal clique of disjoint instructions.
  The idea here is to capture instructions that produce only one μOP executed by one single physical port. Indeed, they do not share any resource so their IPC are additive when put together in a microbenchmark and they have the highest IPC compared to instructions producing twice the same μOP; thus forming the maximum clique of our graph.

A example for a subset of Skylakes' instructions, including the ports they use and their the dependency graph is shown in Figure 3. Here, $1 * p0$ expresses that instruction `DIVPS` decomposes into a single μOP that can only use port $p0$; $2 * p01$ expresses that `ROUNDPD` decomposes into two μOPs that both can use either ports $p0$ or port $p1$ (thus the use of the abstract combined port $p01$). The maximum clique, colored in blue, is composed exclusively of instructions each made up of a single μOP that map to a single port.

- *Most frugal instructions*: Instruction a is considered more frugal than b ($a \preccurlyeq b$) if $\forall p, \overline{a^{\overline{a}} p^{\overline{p}}} \geq \overline{b^{\overline{b}} p^{\overline{p}}}$. This relation defines a pre-order: The n most frugal instructions are selected (the bigger is n the more complete is the core mapping but also the more complex is the linear program).
  The idea here is to collect the instructions that use the lowest number of resources. Figure 4 shows the frugality relation for a few Skylake-X instructions. Setting $n \geq 4$ is necessary to gather the combined ports p_{015}, p_{01}, p_{15} and p_{05}, which correspond to the resources with the highest throughput.

These three steps are described in Algo. 1.

Function *Select_basic_insts*($\mathcal{I}, n$)
  $\mathcal{I}_F = \mathcal{I}$;
  `// Remove low-IPC and duplicates`
  foreach $a \in \mathcal{I}_F$ **do**
    if $\overline{a} \leq 1 - \epsilon$ **then** $\mathcal{I}_F := \mathcal{I}_F - \{a\}$;
    if $\exists b \in \mathcal{I}_F, \forall p \in \mathcal{I}, \overline{a^{\overline{a}} p^{\overline{p}}} = \overline{b^{\overline{b}} p^{\overline{p}}}$ **then**
      $\mathcal{I}_F := \mathcal{I}_F - \{a\}$
  `// Select very basic instructions`
  foreach $a \in \mathcal{I}_F$ **do**
    $Dj[a] := \left\{ b \in \mathcal{I}_F, a^{\overline{a}} b^{\overline{b}} = \overline{a} + \overline{b} \right\}$
  let $a <_{VB} b \Leftrightarrow$
   $(|Dj[a]| > |Dj[b]|) \vee \left(|Dj[a]| = |Dj[b]| \wedge \overline{a} > \overline{b} \right)$;
  $\mathcal{I}_{VB} := \emptyset$;
  for $a \in \mathcal{I}_F$ *in* $<_{VB}$ *order* **do**
    if $\mathcal{I}_{VB} \subset Dj[a]$ **then** $\mathcal{I}_{VB} := \mathcal{I}_{VB} \cup \{a\}$;
    if $|\mathcal{I}_{VB}| = n$ **then return** $\mathcal{I}_B := \mathcal{I}_{VB}$;
  `// Select most frugal instructions`
  $\mathcal{I}_{MF} := \emptyset$;
  let $a \preccurlyeq_{Frugal} b \Leftrightarrow \forall p, \overline{a^{\overline{a}} p^{\overline{p}}} \geq \overline{b^{\overline{b}} p^{\overline{p}}}$;
  for $a \in \mathcal{I}_F$ *in* $\preccurlyeq_{Frugal}$ *order* **do**
    $\mathcal{I}_{MF} := \mathcal{I}_{MF} \cup \{a\}$;
    if $|\mathcal{I}_{VB} \cup \mathcal{I}_{MF}| = n$ **then return**
     $\mathcal{I}_B := \mathcal{I}_{VB} \cup \mathcal{I}_{MF}$;
  return $\mathcal{I}_B := \mathcal{I}_{VB} \cup \mathcal{I}_{MF}$;

Algorithm 1: Finding the set of basic instructions $\mathcal{I}_B$

4.2 Core mapping

The first step before setting the core mapping is to characterize resource usage and sharing of basic instructions. This is done by finding a mapping that reflects the mesured IPC of a set of microbenchmarks $\mathcal{K}$ exclusively composed of those basic instructions (see the next paragraph): Such mapping is obtained using linear programming as described in Alg. 2, that we call the *Bipartite Weight Problem* (BWP).

Hazardous instructions. Our overall infrastructure relies on our ability to measure the throughput of any multi-set of instructions without being polluted by other execution bottlenecks such as alignment issues for the decoding that cannot be modelled by the resource mapping formalism. In theory, assuming an ideal machine that matches the port-mapping performance model, for any two instructions a and b, then three resources should be enough to model any combination $\{(i, j) \in \mathbb{N},\ a^i b^j\}$. But experiments shows that this is not the case in practice, and that one needs to accept a modelling error on the IPC of $a^i b^j$ for any i and j. An important experimental observation is that some of the instructions show more hazards than others. A first pre-processing that considers all simple instruction pairs (a, b) evaluates the minimal error $\epsilon(a, b)$ required to map those two instructions to no more than three resources. From pairwise error, a multi-set error is then defined as follow:

$$\epsilon(K) = \max_{a \in K} \left(\min_{s \in \mathcal{I}_B} \epsilon(a, s), \max_{b \in K} \epsilon(a, b) \right)$$

Bipartite Weight Problem (BWP). The notations are those defined in Def. 3.3: $\rho_{i,r} \in \mathbb{Q}^+$ expresses the usage proportion of the resource r by instruction i; For a microkernel K, each cycle an average of $\overline{K}$ instructions are executed. The proportion of resource r that is used each cycle is thus $\rho_{K,r} = \overline{K} \times \left(\sum_{i \in \mathcal{I}} \sigma_{K,i} \rho_{i,r} \right) / \left(\sum_{i \in \mathcal{I}} \sigma_{K,i} \right)$ which is bounded by its throughput ($\rho_{K,r} \leq \rho_r = 1$) . One of the resources is saturated, that is, $\exists r,\ \rho_{K,r} = 1$. These constraints form our linear problem. As we want the mapping to be as compact as possible, the objective function is set to be the minimimzation of $\sum_{i \in \mathcal{I}, r \in \mathcal{R}} \rho_{i,r}$. Observe that mechanically this objective function will lead to use as less resources as possible (only resources which have at least one none-zero edge are kept) without the need to use 0-1 nor integer variables.

Completeness of the core mapping (*LP_1*). The core mapping needs to be as complete as possible, that is, it should not miss any edge from a basic instruction to a resource. The set of microbenchmarks, which needs to be as "representative" as possible, is built by iterative enrichment: For a given set, a mapping is found using the BWP; This mapping is used to construct a new microbenchmark for each abstract resource; The process consists of a first iteration that runs until no new microbenchmark is added. The seed is made up of microbenchmarks composed of single instructions or combinations of pairs of basic instructions constructed as

Function *Mapping*($\mathcal{K}$)
  Solve *Bipartite Weight Problem*
    $\mathcal{I}$:= instructions($\mathcal{K}$);
    $\forall (i,r) \in \mathcal{I} \times \mathcal{R},\ 0 \leq \rho_{i,r}$;
    $\forall (K,r) \in \mathcal{K} \times \mathcal{R},$
      $\rho_{K,r} = \left(\sum_{i \in \mathcal{I}} \sigma_{K,i}\rho_{i,r}\right) \times \overline{K} / \left(\sum_{i \in \mathcal{I}} \sigma_{K,i}\right)$;
    $\forall (K,r) \in \mathcal{K} \times \mathcal{R},\ \rho_{K,r} \leq 1$;
    $\forall K \in \mathcal{K},\ \max_{r \in \mathcal{R}} \rho_{K,r} \geq 1 - \lambda\epsilon(K)$;
    Minimize $\sum_{i \in \mathcal{I}, r \in \mathcal{R}} \rho_{i,r}$;
  $\mathcal{R}$:= {r such that $\exists a, \rho_{a,r} \geq \epsilon$};
  $\mathcal{G} := (\mathcal{I}, \mathcal{R}, \mathcal{E})$ with $\mathcal{E} = \{(i, r, \rho_{i,r})\}$;
  return $\mathcal{G}$;

Algorithm 2: Formulation of the Bipartite Weight Problem (BWP) under linear programming. $\mathcal{G}$ is the bipartite weighted (conjunctive) graph of the resource usage of each instruction.

follows: 1. $a \in \mathcal{I}$ alone; 2. $a^{\overline{a}}b^{\overline{b}}$, as this benchmark has the following property: If a and b are independent, that is the set of resources used by a and b are disjoint, or have a cumulated usage that does not exceed $\frac{1}{\overline{a}+\overline{b}}$, then $\overline{a^{\overline{a}}b^{\overline{b}}} = \overline{a} + \overline{b}$; 3. $a^M b$ (with M big – 20 in practice) to avoid the convergence of the solver to a simpler solution with fewer resources an lower edges representing only the special conflicting case $a^{\overline{a}}b^{\overline{b}}$.

The enrichment is done as follow: for each resource found, we add a benchmark composed of every instruction using it, therefore creating others constraints relative to interdependence of instructions. Once convergence has been reached, we expect all existing resources to be discovered, and want to make sure that if an edge from an instruction i to a resource r exists it will be represented in our mapping. For this purpose we extract, for each resource r, a saturating kernel $sat[r]$ that we combine with instruction i to build a new microbenchmark.

$$K_{sat}(i,r) = i^1(sat[r])^N$$

where N is chosen bigger than $4 \times \overline{sat[r]}/\overline{i}$ [3].

Saturating kernels (LP$_2$). The saturating kernel $sat[r]$ is chosen among all saturating microbenchmarks (K s.t. $\rho_{K,r} = 1$ – at least one necessarily exists by construction) as the one that has minimum consumption

$$cons(K) = \sum_{i \in \mathcal{I},\ r \in \mathcal{R}} \rho_{i,r}$$

The algorithm for finding the core mapping is described in Algo. 3.

4.3 Finding the complete mapping (LP$_{\text{AUX}}$)

The last step, corresponding to Algo. 4 consists in solving an optimisation problem for each remaining instruction. The

[3]Proof of completness is omited by lack of space

Function *Core_mapping*($\mathcal{I}_B$)
  $\mathcal{K} := \bigcup_{(a,b) \in \mathcal{I}_B^2,\ a \neq b} \left\{a,\ a^{\overline{a}}b^{\overline{b}},\ a^M b\right\}$;
  do
    $\mathcal{G}$:= Mapping($\mathcal{K}$);
    $\mathcal{K}_{new} := \bigcup_{r \in \mathcal{R}} \left\{\Pi_{i \in \mathcal{I}_B,\ \rho_{i,b} \geq \epsilon} i^{\overline{i}}\right\} - \mathcal{K}$;
    $\mathcal{K} := \mathcal{K} \cup \mathcal{K}_{new}$;
  until $\mathcal{K}_{new} = \emptyset$;
  foreach $r \in \mathcal{R}$ **do**
    $sat[r] := K \in \mathcal{K}$ s.t. $\rho_{K,r} =$ 1 that minimizes $cons(K)$;
    for $i \in \mathcal{I}_B$ *s.t.* $i \notin sat[r]$ **do**
      $\mathcal{K} := \mathcal{K} \cup \{K_{sat}(i,r)\}$;
  $\mathcal{G}$:= Mapping($\mathcal{K}$);
  return $\mathcal{K}$, sat, $\mathcal{G}$;

Algorithm 3: Find core mapping and associated saturating kernels (LP$_1$ and LP$_2$)

formulation of the new optimisation problem is very similar to the BWP, except that the resources and the edges of the core mapping computed previously are frozen. The presence or absence of an edge from the to-be-mapped instruction i to a resource r is constrained by using $K_{sat}(i,r)$ (defined in the previous section) in the set of microbenchmarks.

$\lambda := 1$;
$\mathcal{I}_B$:= select_basic_insts($\mathcal{I}, n$);
$\mathcal{K}, sat, \mathcal{G}$:= Core_mapping($\mathcal{I}_B$);
foreach *inst* $\in \mathcal{I}$ **do**
  $\mathcal{K} := \bigcup_{r \in \mathcal{R}} K_{sat}(inst, r)$;
  $\mathcal{I} := \mathcal{I}_B \cup \{inst\}$;
  Solve *Find a solution to the following problem*
    Minimize $\sum_{r \in R} \rho_{inst,R}$;
    $\forall r \in \mathcal{R},\ 0 \leq \rho_{inst,r}$;
    $\forall (K,r) \in \mathcal{K} \times \mathcal{R},\ \rho_{k,r} =$ $\left(\sum_{i \in \mathcal{I}} \sigma_{K,i}\rho_{i,r}\right) \times \overline{k} / \left(\sum_{i \in \mathcal{I}} \sigma_{K,i}\right)$;
    $\forall (K,r) \in \mathcal{K} \times \mathcal{R},\ \rho_{K,r} \leq 1$;
    $\forall K \in \mathcal{K},\ \max_{r \in \mathcal{R}} \rho_{K,r} \geq 1 - \lambda\epsilon(K)$;
  if *No solution is found* **then**
    Launch the solver again with $\lambda := \lambda + 1$

Algorithm 4: Find a resource mapping that models the instruction throughput (LP$_{\text{AUX}}$)

5 Evaluation

Our evaluation is two-fold: first, we prove experimentally the accuracy of our mapping algorithm by obtaining the dual representation of a state-of-the art disjunctive mapping given by uops.info from Abel et al [2]. Secondly, we compare our mapping computed from real-world experiments against assembly microkernels extracted from two benchmarks suites: Polybench, and the SPEC2017 benchmark suite.

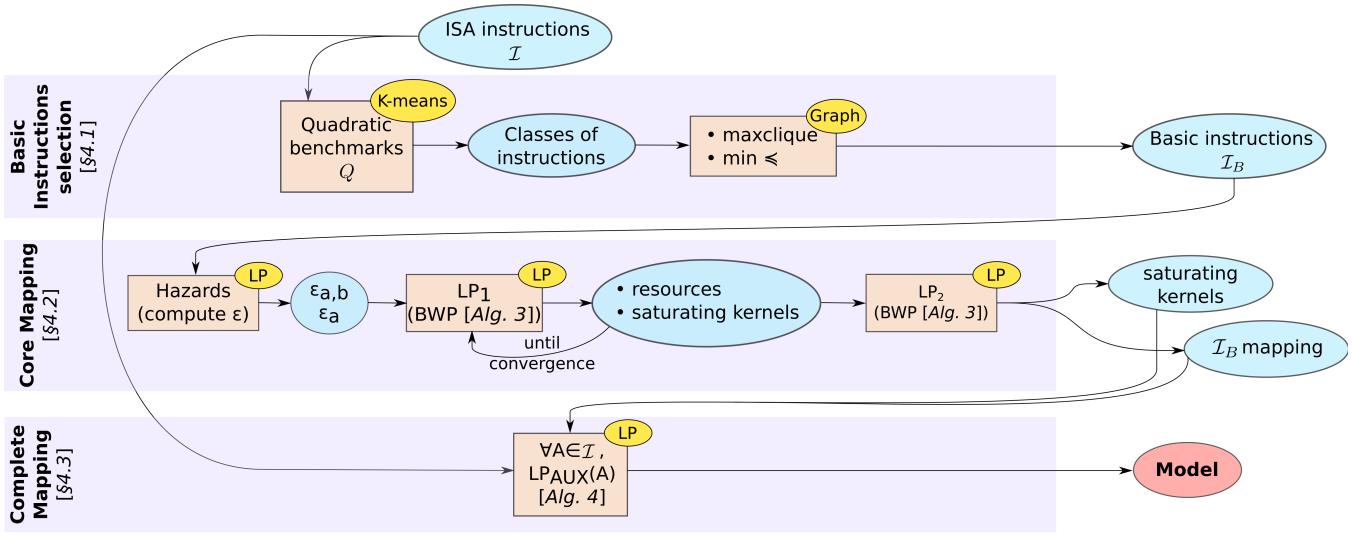

Figure 5. High-level view of the algorithms of PALMED

5.1 Retrieving a state-of-the art mapping

We want to check that we can correctly infer the minimal disjunctive mapping corresponding to the dual representation of uops.info's conjunctive mapping. Our goal is to show that our algorithm is able to find a correct mapping, given (i) an execution model matching the dual representation of uops.info's mapping, (ii) ideal microbenchmarking results, i.e. our benchmarks are simulated without any constraint on the total number of instructions per microbenchmarks and without rounding error. Given the idealized nature of the simulations, the error rate given to the ILP solver was extremely tight: we set the maximum relative error between a microbenchmark simulation (by the abstract model) and the benchmark IPC (computed from an ideal representation) to 10^{-7}.

We then apply our resource mapping algorithm, and try to find the correspondence between our abstract resources and the combined ports on every Intel microarchitecture up to Cannon Lake. The results are shown in Table 1.

Table 1. Number of detected classes for an ideal CPU simulated from uops.info's mapping

Architecture codename	Nb. of eq. classes	Silent ports	Nb. of found res.
Conroe	161	p3 / p4	8
Wolfdale	157	p3 / p4	8
Nehalem	147	p3 / p4	8
Westmere	156	p3 / p4	8
Sandy Bridge	186	None	9
Ivy Bridge	184	None	9
Haswell	218	p7	12
Broadwell	222	p7	12
Skylake	217	p7	14
Skylake-X	288	p7	14
Kaby Lake	205	p7	14
Coffee Lake	210	p7	14
Cannon Lake	242	p7	14

Silent resources. On a few architectures, some ports cannot be detected by our algorithm, as they are hidden under another resource. More generally, a *silent port* is a port p_s for which every instruction that uses this port also uses another fixed port p_m, which masks it.

Given this matter of fact, p_m will always be saturating before p_s, so *hidden ports are never bottlenecks of the execution.* It follows that their representation is not necessary to any performance model, so we do not classify their absence as errors of the mapping.

Our algorithm successfully outputs a mapping corresponding exactly to the mapping of uops.info on all tested architectures, except for the silent port. Up to the Westmere architecture, port 3 is dedicated to the generation of memory address for stores only, whereas port 4 handles the load itself. Generally, an instruction needs first to compute its address before storing anything in memory, so port 3 is hidden by port 4, but that is not always the case. Starting from Haswell, the store address generation unit was moved to port 7, and our algorithm does not output any issue related to this silent port either.

5.2 Comparison on real-world microkernels

Whereas the previous section aims at experimentally checking the expressiveness of our mapping, this section will demonstrate its practical use in real-world conditions. For this, we compare Palmed with the native execution and with the predictions from two existing tools: first, the port mapping deduced from uops.info's work and IACA [14].

5.2.1 Calibration of the model. The port mapping is computed using the algorithm presented in section 4 using a list of x86 instructions extracted from Intel's XED [8]. We discard instructions which cannot be instrumented in practice, such as instruction modifying the control flow, privileged instructions, along with instructions whose IPC is lower than 0.05, as they do not present any interest for performance predictions of throughput-limited microkernels.

Because of variations in the real-world measurements, we fix the error rate to 0.05 for the microbenchmark coefficient, which means that the number of repetitions of an instruction inside its microkernel differs by at most 5% from what the algorithm requires. For example, a benchmark $a^{\overline{a}}b^{\overline{b}}$ with $\overline{a} = 0.06$ and $\overline{b} = 1$ will be rounded to a^1b^{20}. Note that in the BWP defined in Algorithm 2, we use the rounded coefficients and not the ideal ones. The IPC is also rounded accordingly.

Table 2. Experimental environments and main features of the mappings obtained

Machine	SKL-SP	ZEN1
Processor	2x Intel Xeon Silver 4114	AMD EPYC 7401P
Cores	20	24
Benchmarking time	8h	6h
LP solving time	2h	2h
Overall time	10h30	8h30
Gen. microbenchmarks	~ 4,000,000	~ 4,000,000
Resources found	12	5
uops' inst. supported	3313	1104
Instructions mapped	2598	2592

5.2.2 Throughput estimations. To evaluate Palmed, the same microkernel is run:

1. Natively, on our machine, with the IPC measured with `CPU_CLK_UNHALTED` and `INSTRUCTION_RETIRED`.
2. Using IACA, by inserting assembly markers around the kernel and running the tool.
3. Using Abel's work [2], by running the conjunctive mapping with exact compatibility found in Section 5.1 and approximating the execution time by the abstract resource with the highest usage.
4. Using our mapping with abstract resources corresponding to the actual machine, as described in Section 5.2.1.

The microkernels are extracted from two well-known benchmark suite: SPECInt2017 [4] and Polybench [22]. For Polybench, we used QEMU to gather the translation blocs executed at runtime along with their number of executions. For SPEC, we used static binary analysis tools to extract the basic blocks along with performance counters statistics in order to recover the performance-critical section of the code, as the cost of running an emulator was too high to reproduce Polybench's setup. Overall these two benchmark suites generates thousands of basic blocks, and for each we use the various methods above to display the predicted performance of a microkernel made of the same instruction mix that is occurring in that basic block. This evaluation approach allows to generate a high variety of realistic instruction mixes (e.g., combining SIMD and address calculations for numerical kernels like in Polybench). Figure 6 display the results for each basic block/microkernel, comparing the predicted throughput with the native, measured throughput. A heatmap indicates the number of microkernels with a particular predicted IPC versus native IPC. Appendix **??** displays a similar figure, but each microkernel is weighted by the number of times it is executed in the original program, to highlight how frequently occurring basic blocks are modeled.

We evaluate two architectures: the SKL-SP is an Intel Xeon Silver 4114 CPU at 2.20GHz, using Debian, Linux kernel 4.19 and PAPI 6.0.0.1 to collect the execution time in cycle and the number of instructions for each microbenchmarks, restraining to non-AVX-512 instructions. The ZEN is an AMD EPYC 7401P CPU at 2GHz, setup similarly. This information along with execution times of the tool and other experimental results are gathered in Table 2. We compare the number of instructions supported by Palmed with the ones supported by uops.info as a baseline, but, as uops supports only partially AMD's architecture, less than half the instructions supported by our tool are present. On the contrary, uops separates every encoding of the same instructions, therefore leading to a more complete set of supported instructions on SKL-SP. Experimentally, we detect less resources on the real machine than on the simulated one, even though more bottlenecks are present. This matter of fact is explained by the error rate which allows our algorithm to merge some resources together when their are often used together. Moreover, some resources absent from the ideal mapping such as the decoding bandwidth may hide port-related resources that are never bottlenecks on real-world benchamrks. On AMD's machine, the reduced number of resources may be explained by the structure of the processor using a separated floating-point accelerator, which leads to unexpected latencies in the microbenchmarks. In Fig. 6 we observe that Palmed (left) compares very well with IACA (center), showing very similar error distributions. In particular for Polybench, which emphasizes floating point computations and SIMD instructions, the predicted versus actually measured throughput is mostly distributed along the diagonal, i.e., the error is very

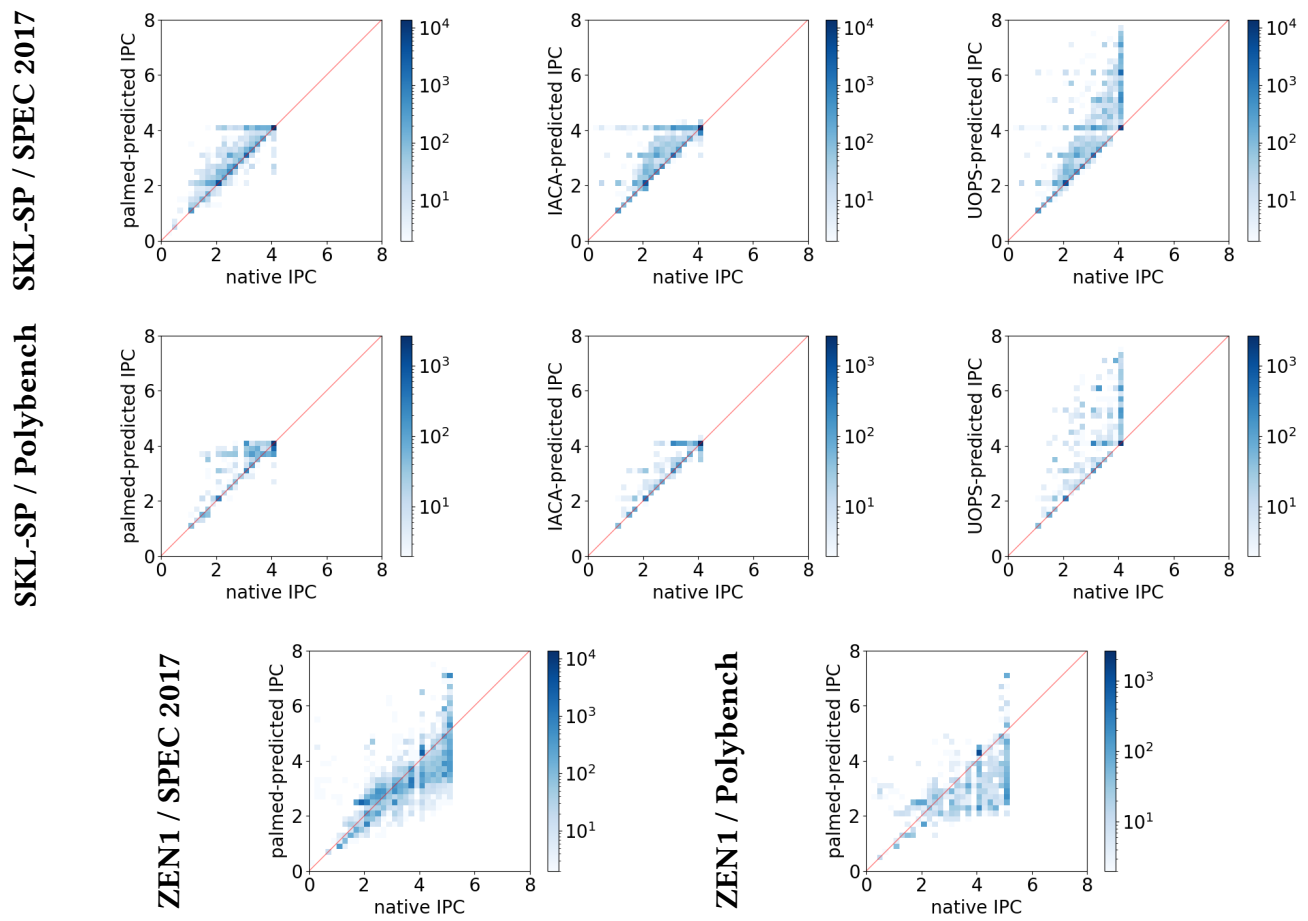

Figure 6. Accuracy of PALMED versus uops.info, IACA and native execution on SPEC CPU2017 and PolyBench/C 4.2

small. Note both can over-estimate or under-estimate the throughput, while uops (left) systematically over-estimates the throughput, and overall has a significantly higher error than IACA and Palmed. Indeed, uops' predictions are only based on port mapping, which ignore other sources of bottlenecks such as the maximum number of instructions that the processors is able to decode per cycle. In fact, Skylake-SP have 8 different hardware ports, so a mapping based solely on them may indicate an IPC up to 8, whereas real-world microbenchmarks hardly reach an IPC of 4 and never exceed this threshold, hence the need for an automated microbenchmark-driven mapping tool.

6 Related work

6.1 Port mapping detection

Intel has developed a static analyzer named IACA [14] which uses its internal mapping base on proprietary information. However, the project is closed-source and has been deprecated since April 2019. Even though some latencies are given directly in the documentation [7], they are known to contain errors and approximations, in addition to being incomplete.

First attempts to measure the latency and throughput of x86 instructions where led by Agner Fog [10] and Granlund [12] using hand-written microbenchmarks. Each benchmark measures the cycles required to repeatedly execute a single type of instruction as described in section 2. Fog also uses hardware performance counters and hand crafted benchmarks to reverse-engineers port mappings for Intel, AMD and VIA CPUs. Fog's mappings are considered by the community to be quite accurate. For example, the machine model of the x86 backend of the the LLVM compiler framework [16] is partially based on them [29].

However, Fog's and Granlund's approach using hand-written benchmark and manual analysis is tedious and error-prone, since modern CPU instruction sets have thousands of different instructions with complicated interactions. Abel and Reineke [1, 2] have tackled this problem by combining an automatic microbenchmark generator with an algorithm for port-mapping construction. Their techniques requires hardware counters that count the number of μOPs executed

on each execution port, which are only available on recent Intel CPUs. They recently also started providing data on the newest generations of AMD CPUs, but since those do not have the required hardware counters Abel and Reineke only publish instruction latencies and throughputs.

OSACA [17], is an open source alternative to IACA that offers a similar static throughput and latency estimator. It relies on automated benchmarks manually linked with publicly available documentation to infer the port mapping and the latencies of the instructions. The tool Kerncraft [13] focuses on hot loop bodies from HPC applications while also modeling caches; its mapping comes from automated benchmarks generated through Likwid [30] and hardware counters measurements. A similar path is taken by CQA [26], a static loop analyser integrated into the MAQAO framework [9] which also supports OpenMP routines. It combines dependency analysis, microbenchmarks, and a port mapping and previous manual results to offer various types of optimization advice to the user, such as vectorisation, or how to avoid port saturation. Both Kerncraft and CQA use a hardcoded port mapping based on the work of Fog and the official Intel and AMD documentation.

Besides the classic port mappings machine learning based approaches have also been used to approximate the throughput of basic blocks with good accuracy. Ithermal [21] uses a deep neural network based on LSTM as a "black box" to predict the execution time of basic blocks, trading understanding of the model for accuracy. The downside of this approach is that the resulting model is completely opaque and can not be analysed or used for any other purpose than to predict the throughput of a given basic block.

PMEvo [24] is a tool that, like Palmed, automatically generates a set of benchmarks that it uses to build a port mapping. Like other previous approaches, and unlike Palmed, it produces a tripartite model with instructions, μOPs, and ports. It does not require hardware performance counter, and only relies on runtime measurements of its benchmarks. The set of benchmarks used is determined semi-randomly using a genetic algorithm. The benchmarks themselves are simpler than those used by Palmed and contain at most two different types of instructions. The main difference between PMEvo and Palmed is that internally PMEvo uses a disjunctive bipartite resource model, instead of the conjunctive model used by Palmed. These models, while able to accurately predict the execution of pipelined instructions bottlenecked only on the execution ports, can not represent other bottlenecks like the reorder buffer, or the non-pipelined instructions like division. More importantly, PMEvo's approach to handle a large set of instructions for the mapping (i.e., all available) may lead to quickly explode the number of microbenchmarks as they are selected by evolutionaty algorithms, while our approach is focused to generate specifically microbenchmarks that saturate resources. Palmed can complete the full mapping, benchmarking included, in a few hours. Another key to this scalability is our incremental approach to handle complex instructions using a linear programming formulation to compute automatically, and optimally, the mapping.

7 Conclusion

Performance modeling for pipelined, super-scalar, out-of-order CPU architectures is notoriously difficult, in part due to the absence of accurate resource mapping information. Indeed, a starting point of CPU performance modeling is determining which instruction can be executed on which port, and at which throughput. Instructions may be executed by several resources accessible via different ports. Prior work to establish the port mapping of instructions range from browsing the usually incomplete vendor documentation to generating microbenchmarks semi-automatically to stress the CPU and measure via a variety of hardware counters the performance obtained, leading to determining the throughput of selected instructions.

In this work, we presented Palmed which automatically builds a resource mappingfor CPU instructions, without requiring specific hardware counters besides measuring instructions executed and cycles elapsed. This allows to model not only execution port usage, but also other limiting resources, such as the frontend or the reorder buffer. We presented an end-to-end approach to enable the mapping of thousands of instructions in a few hours, including microbenchmarking time. Our key contributions include the mathematically rigorous formulation of the port mapping problem as solving iteratively linear programs, enabling an incremental and scalable approach to handling thousands of instructions. We provided a method to automatically generate microbenchmarks saturating specific resources, alleviating the need for statistical sampling. We evaluated our approach and confirmed its ability to produce a port mapping with perfect accuracy for a wide range of ntel architectures in an idealized setup, and demonstrated on one Intel and one AMD high-performance CPUs our system generates automatically practical port mappings that compare favorably with systems like IACA or uops.info when evaluated on microkernels built from basic blocks in SPECInt 2017 and PolyBench/C.

References

[1] Andreas Abel and Jan Reineke. 2019. nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems. *arXiv e-prints* abs/1911.03282 (2019). arXiv:1911.03282 http://arxiv.org/abs/1911.03282

[2] 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, ASPLOS 2019*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, New York, NY, USA, 673–686. https://doi.org/10.1145/3297858.3304062

[3] Jung Ho Ahn, Sheng Li, Seongil O, and Norman P. Jouppi. 2013. McSimA+: A manycore simulator with application-level+ simulation and

detailed microarchitecture modeling. In *2012 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE Computer Society, Austin, TX, USA, 74–85. https://doi.org/10.1109/ISPASS.2013.6557148

[4] James Bucek, Klaus-Dieter Lange, and Jóakim von Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE 2018*, Katinka Wolter, William J. Knottenbelt, André van Hoorn, and Manoj Nambiar (Eds.). ACM, 41–42. https://doi.org/10.1145/3185768.3185771

[5] Chatelet Chatelet, Clement Courbet, Ondrej Sykora, and Nicolas Paglieri. [n.d.]. Google EXEgesis. https://llvm.org/docs/CommandGuide/llvm-exegesis.html

[6] C. L. Coleman and J. W. Davidson. 2001. Automatic memory hierarchy characterization. In *2001 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS*. 103–110.

[7] Intel Corporation. [n.d.]. Intel 64 and IA-32 Architectures Optimization Reference Manual. https://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf

[8] Intel Corporation. [n.d.]. Intel X86 Encoder Decoder (Intel XED). https://github.com/intelxed/xed

[9] Lamia Djoudi, Jose Noudohouenou, and William Jalby. 2008. The Design and Architecture of MAQAOAdvisor: A Live Tuning Guide. In *Proceedings of the 15th International Conference on High Performance Computing (HiPC 2008)*, P. Sadayappan, Manish Parashar, Ramamurthy Badrinath, and Viktor K. Prasanna (Eds.), Vol. 5374. Springer-Verlag, Berlin, Heidelberg, 42–56. https://doi.org/10.1007/978-3-540-89894-8_8

[10] Agner Fog. 2020. Instruction tables: Lists of instruction latencies, through-puts and micro-operation breakdowns for Intel, AMD and VIA CPUs. http://www.agner.org/optimize/instruction_tables.pdf

[11] Franz Franchetti, Tze Meng Low, Doru-Thom Popovici, Richard Michael Veras, Daniele G. Spampinato, Jeremy R. Johnson, Markus Püschel, James C. Hoe, and José M. F. Moura. 2018. SPIRAL: Extreme Performance Portability. *Proc. IEEE* 106, 11 (2018), 1935–1968. https://doi.org/10.1109/JPROC.2018.2873289

[12] Torbjörn Granlund. 2017. Instruction latencies and throughput for AMD and Intel x86 Processors. https://gmplib.org/~tege/x86-timing.pdf

[13] Julian Hammer, Jan Eitzinger, Georg Hager, and Gerhard Wellein. 2017. Kerncraft: A Tool for Analytic Performance Modeling of Loop Kernels. In *Tools for High Performance Computing 2016*, Vol. abs/1702.04653. Springer International Publishing, Cham, 1–22.

[14] Israel Hirsh and Gideon S. [n.d.]. Intel® Architecture Code Analyzer. https://software.intel.com/en-us/articles/intel-architecture-code-analyzer

[15] instlatx64. [n.d.]. x86, x64 Instruction Latency, Memory Latency and CPUID dumps. http://instlatx64.atw.hu/

[16] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2004)*. IEEE Computer Society, San Jose, CA, USA, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[17] Jan Laukemann, Julian Hammert, 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 Computer Society, ACM, Dallas, TX, USA, 121–131. https://doi.org/10.1109/PMBS.2018.8641578

[18] Gabriel H. Loh, Samantika Subramaniam, and Yuejian Xie. 2009. Zesto: A cycle-level simulator for highly detailed microarchitecture exploration. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2009*. IEEE Computer Society, Boston, Massachusetts, USA, 53–64. https://doi.org/10.1109/ISPASS.2009.4919638

[19] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jerónimo Castrillón, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Marjan Fariborz, Amin Farmahini Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc S. Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. The gem5 Simulator: Version 20.0+. arXiv:2007.03152 https://arxiv.org/abs/2007.03152

[20] Gabriel Marin, Jack J. Dongarra, and Daniel Terpstra. 2014. MIAMI: A framework for application performance diagnosis. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014*. IEEE Computer Society, Monterey, CA, USA, 158–168. https://doi.org/10.1109/ISPASS.2014.6844480

[21] Charith Mendis, Alex Renda, Saman P. 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, ICML 2019 (Proceedings of Machine Learning Research)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. PMLR, Long Beach, California, USA, 4505–4515. http://proceedings.mlr.press/v97/mendis19a.html

[22] Louis-Noël Pouchet and Tomofumi Yuki. 2016. PolyBench/C: The polyhedral benchmark suite, version 4.2. http://polybench.sf.net.

[23] GNU C Project. 1987. GNU Compiler Collection (GCC). https://gcc.gnu.org/

[24] 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 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, New York, USA, 608–622. https://doi.org/10.1145/3385412.3385995

[25] Andres Charif Rubial, Emmanuel Oseret, Jose Noudohouenou, William Jalby, and Ghislain Lartigue. 2014. CQA: A code quality analyzer tool at binary level. In *21st International Conference on High Performance Computing, HiPC 2014*. IEEE Computer Society, Goa, India, 1–10. https://doi.org/10.1109/HiPC.2014.7116904

[26] Andres Charif Rubial, Emmanuel Oseret, Jose Noudohouenou, William Jalby, and Ghislain Lartigue. 2014. CQA: A code quality analyzer tool at binary level. In *21st International Conference on High Performance Computing, HiPC 2014*. IEEE Computer Society, Goa, India, 1–10. https://doi.org/10.1109/HiPC.2014.7116904

[27] Daniel Sánchez and Christos Kozyrakis. 2013. ZSim: fast and accurate microarchitectural simulation of thousand-core systems. In *40th Annual International Symposium on Computer Architecture, (ISCA'13)*, Avi Mendelson (Ed.). ACM, New York, NY, USA, 475–486. https://doi.org/10.1145/2485922.2485963

[28] Sony Corporation and LLVM Project. [n.d.]. LLVM Machine Code Analyzer. https://llvm.org/docs/CommandGuide/llvm-mca.html

[29] Craig Topper. 2018. Update to the LLVM scheduling model for Intel Sandy Bridge, Haswell, Broadwell, and Skylake processors. https://github.com/llvm/llvm-project/commit/cdfcf8ecda8065fda495d73ed16277668b3b56dc

[30] Jan Treibig, Georg Hager, and Gerhard Wellein. 2010. LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In *39th International Conference on Parallel Processing (ICPP) Workshops 2010*, Wang-Chien Lee and Xin Yuan (Eds.). IEEE Computer Society, San Diego, California, USA, 207–216. https://doi.org/10.1109/ICPPW.2010.38

[31] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (April 2009), 65–76. https://doi.org/10.1145/1498765.1498785

[32] Matt T. Yourst. 2007. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *2007 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE Computer Society, San Jose, California, USA, 23–34. https://doi.org/10.1109/ISPASS.2007.363733

[33] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Software* 41, 3, Article 14 (June 2015), 33 pages. https://doi.org/10.1145/2764454