PALMED: Throughput Characterization for Superscalar Architectures

Nicolas Derumigny, Théophile Bastian, Fabian Gruber, Guillaume Iooss, Christophe Guillon, Louis-Noel Pouchet, Fabrice Rastello

PALMED: Throughput Characterization for Superscalar Architectures

Nicolas Derumigny*‡, Théophile Bastian*, Fabian Gruber*, Guillaume Iooss*
Christophe Guillon†, Louis-Noël Pouchet‡, Fabrice Rastello*
* Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
† STMicroelectronics, France
‡ Colorado State University, Fort Collins, Colorado, USA

***Abstract*—In a super-scalar architecture, the scheduler dynamically assigns micro-operations (μOPs) to execution ports. The port mapping of an architecture describes how an instruction decomposes into μOPs and lists for each μOP the set of ports it *can* be mapped to. It is used by compilers and performance debugging tools to characterize the performance throughput of a sequence of instructions repeatedly executed as the core component of a loop.**

This paper introduces a dual *equivalent* representation: The resource mapping of an architecture is an abstract model where, to be executed, an instruction *must* use a set of abstract resources, themselves representing combinations of execution ports. For a given architecture, finding a port mapping is an important but difficult problem. Building a resource mapping is a more tractable problem and provides a simpler and equivalent model. This paper describes Palmed, a tool that automatically builds a resource mapping for pipelined, super-scalar, out-of-order CPU architectures. Palmed does not require hardware performance counters, and relies solely on runtime measurements.

We evaluate the pertinence of our dual representation for throughput modeling by extracting a representative set of basic-blocks from the compiled binaries of the SPEC CPU 2017 benchmarks. We compared the throughput predicted by existing machine models to that produced by Palmed, and found comparable accuracy to state-of-the art tools, achieving sub-10 % mean square error rate on this workload on Intel's Skylake microarchitecture.

***Index Terms*—performance model, port mapping, throughput, superscalar architecture, compiler, performance debugging, code selection**

I. 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 [22], [23], [37] to application level analytical formulations [36], [16]. A widely used approach for modeling the CPU of modern pipelined, super-scalar, out-of-order processors consists in decoupling the different sources of bottlenecks, such as the latency related ones (critical path, dependencies), the memory-related ones (cache behavior, bandwidth, prefetch, etc.), or the port throughput related ones (instruction execution units). Decoupled models allow to pinpoint the source of a performance bottleneck, which is critical both for compiler optimization [28], [20], kernel hand-tuning [38], [14], and performance debugging [30], [17], [21], [24], [33], [5]. In particular, the code selection step is based on ad-hoc instruction cost models, that Palmed aims at automatically generating for new architectures. Cycle-approximate simulators such as ZSim [32] or MCsimA+ [6] can also take advantage of such an instruction characterization. *This paper focuses on modeling the port throughput, that is, estimating the performance of a dependency-free loop where all memory accesses are L1-hits.*

Such modeling is usually based on the so-called port mapping of a CPU, that is the list of execution ports each instruction can be mapped to. This motivated several projects to extract information from available documentation [8], [21]. But the documentation on commercial CPUs, when available, is often vague or outright lacking information. Intel's processor manual [10], for example, does not describe all the instructions implemented by Intel cores, and for those 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 for a more exhaustive and precise instruction characterization is based on micro-benchmarks, such as those developed to characterize the memory hierarchy [9]. While characterizing the latency of instructions is quite easy [13], [18], [15], 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 the available resources and the way they are shared.

In this work, we present a fully automated, architecture-agnostic approach, fully implemented in Palmed, to automatically build a mapping between instructions and execution ports. It automatically builds a static performance model of the throughput of sets of instructions to be executed on a particular processor. While prior techniques targeting this problem have been presented, e.g. [29], [4], we make several key contributions in Palmed to improve automation, coverage, scalability, accuracy and practicality:

- We introduce a dual equivalent representation of the port mapping problem, into a conjunctive abstract resource mapping problem, facilitating the creation of specific micro-benchmarks to saturate resources.
- We present several new algorithms: to automatically generate versatile sets of saturating micro-benchmarks, for any instruction and resource; to build efficient Linear Programming optimization problems exploiting these micro-benchmark measurements; and to compute a complete resource mapping for *all* benchmarkable instructions.
- We present a complete, automated implementation in Palmed, which we evaluate against numerous other approaches including IACA [17], LLVM-mca [33], PMEvo [29] and UOPS.info [4].

This paper has the following structure. Related work is first discussed in Sec. II. Sec. III discusses the state-of-practice and presents our novel approach to automatically generate a valid port mapping. Sec. IV presents formal definitions and the equivalence between our model and the three-level mapping currently in use. Sec. V presents our architecture-agnostic approach to deduce the abstract mapping without the use of any performance counters besides elapsed CPU cycles. Sec VI extensively evaluates our approach against related work on two off-the-shelf CPU before concluding.

II. Related Work

Intel has developed a static analyzer named IACA [17] which uses its internal mapping based 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 [10], they are known to contain errors and approximations, in addition to being incomplete.

First attempts on x86 to measure the latency and throughput were led by Agner Fog [13] and Granlund [15] using hand-written microbenchmarks. 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 back-end of the LLVM compiler framework [20] is partially based on them [34]. However, Fog and Granlund's approach is tedious and error-prone, since modern CPU instruction sets have thousands of different intricate instructions. Abel and Reineke [4], [3] have tackled this problem by combining an automatic microbenchmark generator with an algorithm for port-mapping construction. Their technique relies on hardware counters for the number of μOPs executed on each execution port, only available on recent Intel CPUs. They recently started providing data on the newest generations of AMD CPUs, but by lack of necessary hardware counters, only latency and throughput are published.

OSACA [21] is an open source alternative to IACA offering 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 [16] focuses on hot loop bodies from HPC applications while also modeling caches; its mapping comes from automated benchmarks generated through Likwid [35] and hardware counters measurements. CQA [31], a static loop analyzer integrated into the MAQAO framework [12], takes a similar path while also supporting 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 hard-coded port mapping based on Fog's work and official Intel and AMD documentation.

Besides the classic port mappings, machine learning based approaches have also been used, *eg.* in Ithemal [25], to approximate the throughput of basic blocks with good accuracy. However, the resulting model is completely opaque and cannot be analyzed or used for any other purpose than the prediction of basic block throughputs. For instance, Ithemal does not report on the influence of each instruction, which is critical for manual assembly optimization.

PMEvo [29] is a tool that, like Palmed, automatically generates a set of benchmarks that it uses to build a port mapping. It produces a disjunctive tripartite model with instructions, μOPs, and ports, which is the key different with Palmed. 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 PMEvo uses internally 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, cannot represent other bottlenecks like the reorder buffer, or the non-pipelined instructions like division. More importantly, PMEvo's approach is less scalable, as handling more instructions may quickly lead to an overwhelming number of microbenchmarks, 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.

III. Motivation and Overview

A. Background

In this work, we consider a CPU as a processing 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 schedules these μOPs on a free compatible execution port, before the final *retirement* stage. Even though 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, leading to thousands of μOPs [4].

Execution ports are controllers routing μOPs to *execution units* with one or more different functional capabilities: for example, on the Intel Skylake architecture, only the 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.

The *latency* of an instruction is the number of clock cycles elapsed between two dependent computations. The latency of an instruction I can be experimentally measured by creating a micro-benchmark that executes a long chain of instances of I, each depending on the previous one.

The *throughput* of an instruction is the maximum number of instances of that instruction that can be executed in parallel in one cycle. On every recent x86 architecture, all units but the divider are fully pipelined, meaning that they can reach a maximum throughput of one μOP per cycle – even if their latency it greater than one 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: The combined throughput of a multiset[1] of instructions can be defined similarly. For example, the throughput of $\{\texttt{ADDSS}^2, \texttt{BSR}\}$, i.e. two instances of `ADDSS` and one instance of `BSR`, is equal to the number of instructions executed per cycle (IPC) by the micro-benchmark:

```
repeat:
  ADDSS %xmm1 %xmm1; ADDSS %xmm2 %xmm2; BSR %rax %rax;
  ADDSS %xmm3 %xmm3; ADDSS %xmm4 %xmm4; BSR %rbx %rbx;
  ADDSS %xmm5 %xmm5; ADDSS %xmm6 %xmm6; BSR %rcx %rcx;
  ...
```

A *resource-mapping* describes the resources used by each instruction in a way that can be used to derive the throughput for any multiset of instructions, without having to execute the corresponding micro-benchmark. Such information is crucial for manual assembly optimization to pinpoint the precise cause of slowdowns in highly optimized codes, and measure the relative usage of the peak performance of the machine.

In this work, we target the automatic construction of a resource mapping for a given CPU on which we can accurately measure elapsed cycles for a code fragment. 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].

B. Constructing a Resource Mapping

To characterize the throughput of each individual instruction, a description of the available resources and the way they are shared is needed. 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. 1a). 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 it 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 [30], [13], [15], [4].

Such approaches, while powerful and allowing a semi-automatic characterization of basic-block throughput, suffer from several limitations. First, they assume 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. Thus, it provides a performance model of an ideal architecture that does not necessarily fully match reality.

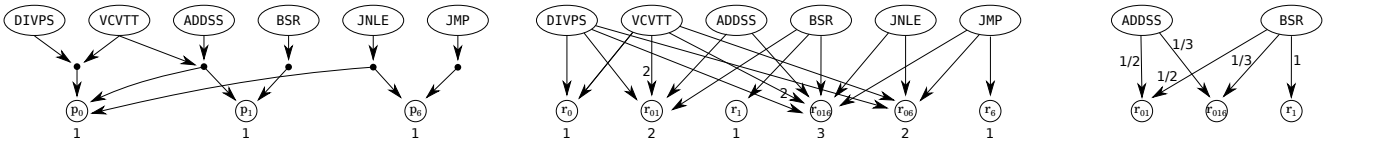
To overcome these limitations, we restrict ourselves to only using cycle measurements when building our performance model. Not relying on specialized hardware performance counters may complicate the initial model construction, but in exchange our approach is able to model resources not covered by hardware counters with relative ease. This also paves the way to significantly ease the development of modeling techniques for new CPU architectures. One of the main challenges is to generate a set of micro-benchmarks that allows the detection of all the possible resource sharing. Unfortunately, to be exhaustive, and in the absence of structural properties, this set is combinatorial: all possible mixes of instructions need to be evaluated. A simple way to reduce the set of micro-benchmarks required is to reduce the set of modeled instructions to those that are emitted by compilers [25], [29]. Another natural strategy followed by Ithemal [25] is to build micro-benchmarks from the "most executed" basic-blocks of some representative benchmarks. A third strategy, used by PMEvo [29], is to restrict micro-benchmarks to contain repetitions of only two different instructions.

Our solution is constructive and follows several successive steps that allow building a non-combinatorial number of micro-benchmarks that stresses the usage of each individual resource, thus characterizing 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, instead of a black-box only able to predict the throughput of a microkernel. One issue with the standard port-mapping approach, as used in [4], [21], [33], 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 of μOPs to ports 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

[1] A multiset is a set that can contain multiple instances of an element. As 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.



(a) Port mapping (disjunctive form) and maximum throughput of each port.

(b) Abstract resource mapping (conjunctive form) and maximum throughput of each resource.

(c) Normalized conjunctive form for ADDSS and BSR.

Fig. 1: Mappings computed for a few SKL-SP instructions.

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.

TABLE I: SUMMARY OF KEY FEATURES OF PALMED VS. RELATED WORK

	no HW counters	no manual expertise	interpretable	general
llvm-mca [33]	✗	✗	✓	✓
Ithemal [25]	✓	✗	✗	✗
IACA [17]	N/A	✗	✓	✓
uop.info [4]	✗	✗	✓	✓
PMEvo [29]	✓	✓	✓	✗
Palmed	✓	✓	✓	✓

C. Resource Mapping: Dual Representation

Our approach is based on a crucial observation: 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.

For the sake of illustration only (Palmed finds in practice a mapping for all supported instructions), we consider the Skylake instructions restricted to those that only use ports 0, 1, or 6 (denoted as p_0, p_1, and p_6). Fig. 1a shows the port mapping for six such instructions. In this example: the μOP of BSR has a single port p_1 on which it can be issued; as for instruction ADDSS, its μOP can be issued on either p_0 or p_1. Hence, BSR has a throughput of one, that is, only one instruction can be issued per cycle, whereas ADDSS has a throughput of two: two different instances of ADDSS may be executed in parallel by p_0 and p_1. The throughput of the multiset $K = \{\texttt{ADDSS}^2, \texttt{BSR}\}$, more compactly denoted by $\texttt{ADDSS}^2\texttt{BSR}$, is therefore determined by the combined throughput of resources p_0 and p_1. Indeed, in a steady state mode, the execution can saturate both resources by repeating the pattern represented in Fig 2a. In this case, there clearly does not exist any better scheduling, and the corresponding execution time for K is 3 cycles for every 6 instructions, that is, an Instruction Per Cycle (IPC) of 2. Now, if we consider the set $\texttt{ADDSS BSR}^2$, its throughput is limited by p_1. Indeed, the optimal schedule in that case would repeat the pattern represented in Fig 2b, which requires 2 cycles for 3 instructions, that is, an IPC of 1.5. More generally, the maximum throughput of a multiset on a tripartite port-mapping can be solved by expressing the minimal scheduling problem as a flow problem.

p_0	p_1
ADDSS	BSR
ADDSS	BSR
ADDSS	ADDSS

(a) $\texttt{ADDSS}^2$ BSR

p_0	p_1
ADDSS	BSR
$\emptyset$	BSR

(b) ADDSS $\texttt{BSR}^2$

Fig. 2: Disjunctive port assignment examples

The *dual representation*, advocated in this paper, corresponds to a *conjunctive* bipartite resource mapping as illustrated in Fig. 1b. In this mapping, an instruction such as ADDSS which uses one out of two possible ports p_0 and p_1 will only use the abstract resource r_{01} representing the combined load on both ports, and will use neither r_0 nor r_1. In this model, the maximum throughput of r_{01} is the sum of the throughput of p_0 and p_1, that is, 2 uses per cycle. Instructions that may only be computed on p_0 will then use r_0 *and* r_{01}, along with all other resources combining the use of p_0 with other ports such as r_{06} and r_{016}. Followingly, the average execution time of a microkernel is computed as the maximum load over all abstract resources, that is, their number of uses divided by their throughput (see Sec. IV). One can prove (see [1]) the strict equivalence between the two representations *without the need for any combinatorial explosion in the number of combined resources*. Because of this property, the trade-off offered by the conjunctive formulation (more resources for a simpler throughput computation) offers better overhaul solving complexity that former disjunctive-based approaches for real processors, hence the better scalability of Palmed. Indeed, in practice, some combined resources are not needed (e.g. r_{16} in our example) as their usage is already perfectly described by the usage of individual resources (here, r_1 and r_6).

A key 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 "instructions 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 resource 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 also represent other bottlenecks such as the instruction decoder or the reorder buffer as other abstract resources. Note that this corresponds to the user view, where the micro-ops and their execution paths are kept hidden inside the

processor. An important contribution of this paper 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.

D. Palmed: Flow of Work

Fig. 3 overviews the major steps of Palmed, which are extensively described in Sec. V. Our algorithm follows an approach similar to the one developed by uops.info: its principle is to first find a set of *basic instructions* producing only one μOP and bound to one port.

This first step can be done on Intel CPUs by measuring the μOP per cycle on each port for each instruction through performance counters.

Those basic instructions are then used to characterize the port mapping of any general instruction by artificially saturating one-by-one each individual port and measuring the effect on the usage of the other ports. The challenge addressed by Palmed is to find a mapping, even for architectures that do not have such hardware counters.

This translates in two major hardships: firstly, in our case, there is no predefined resources; secondly, there even is no simple technique to find the number of μOPs an instruction decomposes into. As illustrated by Fig. 3 the algorithm of Palmed is composed of three steps: 1. Find basic instructions; 2. Characterize a set of abstract resources (expressed as a *core mapping*) and an associated set of saturating microkernels (a single instruction might not be enough to saturate a resource); 3. Compute the resource usage of each other instruction with respect to the core mapping.

As an example, let us go back to our example: instructions using only p_0, p_1, or p_6. On Intel's Skylake microarchitecture, there exists 754 benchmarkable instructions using only these 3 ports. Quadratic benchmarking – that is, measuring the execution time of one benchmark per pair of instruction, leading to a quadratic number of measures (567762) – allows us to regroup those who have the same behavior together, leading to only 9 classes of instructions. For each class, a single instruction is used as a representative. Among those instructions, two heuristics (described in sec V-A) select the set of basic instructions, outputting `DIVPS`, `BSR`, `JMP`, `JNLE`, and `ADDSS`.

Fig. 1b shows the output of the *Core Mapping* stage in Fig. 3, in bold. In practice, abstract resources are internally named R_0, ..., R_5. For convenience we renamed them to the hardware execution ports they correspond to: for example, the abstract resource r_{01} corresponds to the combined use of port p_0 and p_1 for an optimal schedule.

The Core mapping also computes a set of saturating micro-benchmarks that individually saturate each of the individual abstract resource. Here, each basic instruction will constitute by itself a saturating micro-benchmark: `DIVPS` will saturate r_0, `BSR` will saturate r_1, `JMP` will saturate r_6, `ADDSS` will saturate r_{01}, and `JNLE` will saturate r_{06}. Note that this is not the case in general: we possibly need to combine several basic instructions together to saturate a resource. Here, the saturating micro-benchmark for resource r_{016} is composed of two basic instructions: `ADDSS` and `JNLE`. The last phase of our algorithm will, for each of the 742 remaining instructions, build a set of micro-benchmarks that combine the saturating kernels with the instruction, and compute its mapping.

IV. The Bipartite Resource Mapping

This section provides a formal presentation of the dual conjunctive formulation used by Palmed.

Definition IV.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, σ_K representing the number of repetition of the instruction K in the microbenchmark. The number of instructions executed during one loop iteration is $|K| = \sum_i \sigma_{K,i}$.*

In a classical *disjunctive* port mapping formalism, an instruction i from a microkernel K is assigned to a port (*resource* r) that is compatible. The execution time of K is determined by the resource which is used the most by its instructions in a given such assignment, and depends on the assignment picked, as presented in Sec III. Instead, we consider a *conjunctive* port mapping:

Definition IV.2 (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*, that has a (normalized) throughput of 1; $E \subset I \times \mathcal{R}$ expresses the required use of abstract resources for each instruction. An instruction i that uses a resource r ($(i,r) \in E$) always uses the same proportion (number of cycles) $\rho_{i,r} \in \mathbb{Q}^+$. 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 a resource r for $(\sigma_{K,i}\rho_{i,r})$ cycles. The number of cycles required to execute one loop iteration is:

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

One should observe that Def. IV.2 defines formally a *normalized* version where throughputs of abstract resources are set to 1. For the sake of clarity, the example in Sec. III was considering non-normalized throughputs, that is, different than 1. Going from non-normalized (as in Fig. 1b) to normalized form (as in Fig. 1c) simply relies in dividing the incoming edges of a resource by the resource's throughput before setting its throughput to 1. For example, in the non-normalized form `VCVTT` uses 2 times r_{01}, which has a throughput of 2, leading to a normalized $\rho_{\texttt{VCVTT},r_{01}}$ of 1. Similarly, $\rho_{\texttt{ADDSS},r_{016}} = 1/3$.

Definition IV.3 (Throughput)**.** *The throughput $\overline{K}$ of a microkernel K is its instruction per cycle rate (IPC), defined as:*

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

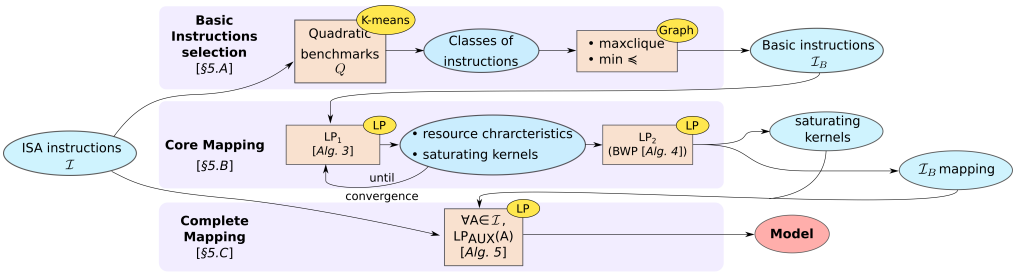


Fig. 3: High-level view of the algorithms of Palmed

Example: If $K = \texttt{ADDSS}^2\ \texttt{BSR}$, as in Fig 2a,

$$\begin{aligned} t(K) &= \max_{r\in\{r_1,r_{01},r_{016}\}} (2 \times \rho_{\texttt{ADDSS},r} + \rho_{\texttt{BSR},r}) \\ &= \max\left((r_1)\ 2\times 0+1, (r_{01})\ 2\times\frac{1}{2}+\frac{1}{2}, (r_{016})\ 2\times\frac{1}{3}+\frac{1}{3}\right) \\ &= 1.5 \\ \overline{K} &= (2+1)/1.5 = 2 \end{aligned}$$

On $K' = \texttt{ADDSS BSR}^2$, as in Fig 2b, the same computation gives $t(K') = 2$, the bottleneck being r_1; hence, $\overline{K'} = 3/2$.

The mathematical definitions, the method to build a conjunctive port mapping from a disjunctive one, and the abstract resource and the equivalence proof can be found in [1].

V. Computing Resource Mapping

As depicted in Fig. 3, our approach can be decomposed into three different steps. Sec. V-A describes the selection of *basic instructions*, a subset of instructions that map to as few resources as possible. Sec. V-B describes the computation of the *core mapping* for these basic instructions. The core mapping stays fixed for the rest of the algorithm. Along with the core mapping, we also select a saturating micro-benchmark 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. Sec. V-C describes how Palmed uses the saturating kernels and the core mapping to deduce, one by one, the resources used by the remaining instructions of the targeted architecture.

A. Basic Instructions Selection

The first step of our algorithm trims the instruction set to extract a minimal set of instructions for which the mapping will be computed. As this (core) mapping will be reused later, we need enough instructions to detect all resources, but the more instructions we have, the longer the resolution of the linear problem this mapping will take. We thus first apply two simple filters that reduce the number of basic instructions, as depicted in the first half of Algo. 1.

Low-IPC: If $\overline{a} < 1$ (measured with a microbenchmark repeating only a), then a is not consider as a candidate for basic instructions. Assuming every physical resource to have a throughput of 1, such instructions use one resource more than once. However, these low-IPC instructions are still mapped at the very last step of Palmed (see Sec. V-C).

Then, we compute, for every remaining pair of instruction (a, b), the throughput of the microkernel $a^{\overline{a}}b^{\overline{b}}$. This set of benchmarks is called *quadratic benchmarks* (see Fig. 3) as their number is quadratic with respect to the number of instructions. These quadratic benchmarks are later reused in each of the following heuristics.

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 similarly 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 uses hierarchical clustering [26].

Function *Select_basic_insts*($\mathcal{I}, n$)
  $\mathcal{I}_F := \mathcal{I}$;
  `// Remove low-IPC; compute eq. classes`
  foreach $a \in \mathcal{I}_F$ **do**
    if $\overline{a} \le 1 - \epsilon$ **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 greedier instructions`
  $\mathcal{I}_{MF} := \emptyset$;
  for $a \in \mathcal{I}_F$ *in* $\preccurlyeq_{greedier}$ *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: Set of basic instructions $\mathcal{I}_B$

Once low IPC instruction duplicates have been removed, the selection relies on two criteria (cf Algo. 1):

- **Very basic instructions:** Instructions a and b are considered *disjoint* if $a^{\overline{a}}b^{\overline{b}} = \overline{a} + \overline{b}$. The set of very basic instructions is defined as a maximal clique of disjoint

Function *Core_mapping*($\mathcal{I}_B$)
  // Characterize resources
  $\mathcal{K} := \bigcup_{(a,b)\in\mathcal{I}_B^2,\ a\neq b} \left\{a,\ a^{\overline{a}}b^{\overline{b}},\ a^M b\right\}$;
  do
    $\mathcal{G} :=$ Shape_Mapping($\mathcal{K}, \mathcal{I}_{VB}, \mathcal{I}_{MF}$)) ; // LP_1
    $\mathcal{K}_{new} := \bigcup_{r\in\mathcal{R}} \left\{\Pi_{i\in\mathcal{I}_B,\ \rho_{i,b}\geq\epsilon} i^{\overline{i}}\right\} - \mathcal{K}$;
    $\mathcal{K} := \mathcal{K} \cup \mathcal{K}_{new}$;
  until $\mathcal{K}_{new} = \emptyset$;
  $\mathcal{G} :=$ Mapping($\mathcal{K}, \mathcal{G}$) ; // LP_2
  // Find saturating kernels
  foreach $r \in \mathcal{R}$ **do**
    sat$[r] := K \in \mathcal{K}$ s.t. $\rho_{K,r} = 1$ minimizing *cons*(K);
    for $i \in \mathcal{I}_B$ *s.t.* $i \notin$ *sat*$[r]$ **do**
      $\mathcal{K} := \mathcal{K} \cup \{K_{sat}(i, r)\}$;
  return $\mathcal{K}$, *sat*, $\mathcal{G}$;

Algorithm 2: Core mapping and saturating kernels

Function *Shape_mapping*($\mathcal{K}, \mathcal{I}_{VB}, \mathcal{I}_{MF}$)
  Solve
    $\forall (i, r) \in \mathcal{I} \times \mathcal{R},\ \rho_{i,r} \in \{0, 1\}$;
    $\forall i \in \mathcal{I}_{VB}, \min_{r\in\mathcal{R}} 1 - \rho_{i,r} + \sum_{j\in\mathcal{I}_{VB}\setminus\{i\}} \rho_{j,r} = 0$;
    $\forall i \in \mathcal{I}_{MF}, \max_{r\in\mathcal{R}} \rho_{i,r} + \sum_{j\not\asymp i} \rho_{j,r} = 1 + |\{j \not\asymp i\}|$;
    foreach $k \in \mathcal{K}$ *s.t.* $\{i^\alpha \in k$ *s.t. cycles*$(i^\alpha) =$ *cycles*$(k)\} = \emptyset$ **do**
      $\max_{r\in\mathcal{R}} \sum_{i\in k} \rho_{i,r} \geq |\{i \in k\}|$;
    foreach $k \in \mathcal{K}$ *s.t.* $\{i^\alpha \in k$ *s.t. cycles*$(i^\alpha) =$ *cycles*$(k)\} \neq \emptyset$ **do**
      $\forall i^\alpha \in k$ s.t. cycles$(i^\alpha) =$ cycles(k)
        $\min_{r\in\mathcal{R}} 1 - \rho_{i,r} + \sum_{j\in k, j\neq i} \rho_{j,r} = 0$;
    Minimize $\sum_{i\in\mathcal{I}_B} \max_{r\in\mathcal{R}} \rho_{i,r}$;
  return $(\mathcal{I}, \mathcal{R}, \{\rho_{i,r}\})$;

Algorithm 3: $\texttt{LP}_1$: Shape of core mapping

Function *Mapping*($\mathcal{K}, \mathcal{G}$)
  Solve *Bipartite Weight Problem*
    $\mathcal{I} :=$ instructions($\mathcal{K}$);
    $(\rho_{i,r})_{\mathcal{I},\mathcal{R}} :=$ edges($\mathcal{G}$);
    $\forall (i, r) \in \mathcal{I} \times \mathcal{R},\ 0 \leq \rho_{i,r} \in [0, 1]$;
    $\forall (K, r) \in \mathcal{K} \times \mathcal{R}$,
      $\rho_{K,r} = \left(\sum_{i\in\mathcal{I}} \sigma_{K,i}\rho_{i,r}\right) \times \overline{K} / \left(\sum_{i\in\mathcal{I}} \sigma_{K,i}\right)$;
    $\forall (K, r) \in \mathcal{K} \times \mathcal{R},\ \rho_{K,r} \leq 1$;
    $\forall K \in \mathcal{K},\ S_K = \max_{r\in\mathcal{R}} \rho_{K,r}$;
    Minimize $\sum_{K\in\mathcal{K}} (1 - S_K)$;
  return $(\mathcal{I}, \mathcal{R}, \{\rho_{i,r}\})$;

Algorithm 4: $\texttt{LP}_2$: Bipartite Weight Problem (BWP), used in $\texttt{LP}_2$ and $\texttt{LP}_{\texttt{AUX}}$

$\mathcal{I}_B :=$ 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*
    $\forall r \in \mathcal{R},\ 0 \leq \rho_{inst,r}$;
    $\forall (K, r) \in \mathcal{K} \times \mathcal{R},\ \rho_{k,r} = \left(\sum_{i\in\mathcal{I}} \sigma_{K,i}\rho_{i,r}\right) \times \overline{k} / \left(\sum_{i\in\mathcal{I}} \sigma_{K,i}\right)$;
    $\forall (K, r) \in \mathcal{K} \times \mathcal{R},\ \rho_{K,r} \leq 1$;
    $\forall K \in \mathcal{K},\ S_K = \max_{r\in\mathcal{R}} \rho_{K,r}$;
    Minimize $\sum_{K\in\mathcal{K}} (1 - S_K)$;

Algorithm 5: $\texttt{LP}_{\texttt{AUX}}$: Complete resource mapping

instructions. This captures instructions that maps to a single resource. Indeed, two instructions that do not share any resource will have their IPC additive, thus belonging to the maximum clique of our graph.

- **Most greedier instructions:** Instruction a is considered more greedier than b ($a \preccurlyeq_{greedier} b$) if $\forall p, \overline{a^{\overline{a}}p^{\overline{p}}} \geq \overline{b^{\overline{b}}p^{\overline{p}}}$. This relation defines a pre-order, and we select the n most greedier instructions.

B. Core Mapping

The core mapping phase as described in Alg. 2 is decomposed in two steps. The objective of the first step is to build the shape of the resource mapping from the basic instructions, containing all visible resources and possible edges. In the second step, Palmed computes the values of the edges and outputs a *saturating benchmark* for every detected resource. Similarly to Abel and Reinecke's work [4], these benchmarks are reused as an indicator of the usage of their resource in the complete mapping phase (Sec. V-C).

Both of these steps use linear programming to build step-by-step a mapping that reflects accurately the measured IPC of a set of microkernels $\mathcal{K}$, and are detailed in the following two paragraphs.

Characterize resources ($\texttt{LP}_1$): The goal of the first step is to find the shape of the resource mapping, that is, the number of resources needed and the possible edges from core instructions to resources. For this, Palmed solves the following Integer Linear Programming (ILP) problem, formalized in Alg. 3, repeated until no new benchmark is added:

Objective function: Minimize the number of resources.

Constraints: From the following seed of microkernels:

1) $a \in \mathcal{I}$ alone;
2) $a^{\overline{a}}b^{\overline{b}}$, as this benchmark has and IPC of $\overline{a} + \overline{b}$ if a and b are independents or if their common resources are not dominantly used.
3) $a^M b$ (with $M = 4$ in practice – see [1] for detailed justification) to avoid the convergence of the solver to a simpler solution with fewer resources.

We derive the following constraints (in the order of Alg. 3):

– Each very basic instruction as defined in Sec. V-A is linked to at least one resource unused by other very basic instructions (line 4).
– For each greedier instruction i as defined in Sec. V-A, there exists at least one resource common to i and to all other instructions a for which $i^{\overline{i}}a^{\overline{a}} \neq \overline{i} + \overline{a}$ (line 5). This relation corresponds to the negation of the *disjoint* relation defined in Sec. V-A, that we note $\not\asymp$.
– For all other microkernels: 1) every instruction identified as saturating (that is, instructions for which the execution time of the microkernel is equal to its execution time) maps to at least a resource unused by other instructions of the microkernel (line 7); 2) if no saturating instruction is found, then there exists a resource shared by every instruction of the benchmark (line 10).

The enrichment is done as follows: for each resource found, we add a benchmark composed of every instruction using it

with a multiplicity of their IPC, splitting it in case of undesired merges. Once convergence has been reached, we expect most of existing resources and edges to be discovered: Palmed passes to the ($\texttt{LP}_2$) step to compute the value of the edges.

Bipartite Weight Problem (BWP) and Core Mapping ($\texttt{LP}_2$): The BWP is formalized in Alg. 4, and aims at finding the correct values of the edges found during the $\texttt{LP}_1$. Using the notations from Def. IV.2: $\rho_{i,r} \in \mathbb{Q}^+$ expresses the proportional usage of the resource r by instruction i, and $\overline{K}$ the average number of instructions executed each cycle when K is executed by the CPU. The proportion of a resource r that is used is thus $\rho_{K,r} = \overline{K} \cdot \left(\sum_{i \in \mathcal{I}} \sigma_{K,i} \rho_{i,r}\right) / \left(\sum_{i \in \mathcal{I}} \sigma_{K,i}\right)$, bounded by its throughput ($\rho_{K,r} \leq \rho_r = 1$). One of the resources must be the limiting factor, that is, $\exists r,\ \rho_{K,r} = 1$. However, we authorise sub-saturation of the resources, acknowledging our model does not predict accurately every microkernel, and we note $S_K = \max_r \rho_{K,r} \leq 1$. We also restrict the possible edges to the ones output by the $\texttt{LP}_1$. These constraints form our linear problem minimizing the sum of predictions error, that is $\sum_{k \in \mathcal{K}} (1 - S_k)$.

Once the mapping has been computed, for every resource r, a saturating kernel $sat[r]$ is chosen among all saturating microbenchmarks of the $\texttt{LP}_2$ (K s.t. $\rho_{K,r} = 1$, at least one necessarily exists by construction) as the one that has minimum consumption:

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

C. Complete Mapping ($\texttt{LP}_{AUX}$)

In the last step, corresponding to Algo. 5, an optimization problem is solved for each remaining instruction. The formulation of the new optimization 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 $K_{sat}(i, r) = i^{\bar{i}} sat[r]^{L * \overline{sat[r]}}$ in the set of microbenchmarks, with $L = 4$ in practise. The idea is to force the saturation of r by charging it with $sat[r]$, hence expressing the usage of r by i.

VI. Evaluation

Our evaluation section compares throughput accuracy on assembly microkernels extracted from two benchmarks suites: the SPECrate version of SPECint2017 [7] and Polybench [27].

We compare Palmed against the native execution, along with the predictions of four existing tools: IACA [17], PMEvo [29], llvm-mca [33] and the port mapping deduced from uops.info's work [4].

Our evaluation is performed on two architectures: the SKL-SP is an Intel Xeon Silver 4114 CPU at 2.20 GHz, using Debian, Linux kernel 4.19 and PAPI 6.0.0.1 to collect the execution time in cycles for each microbenchmarks, restraining to non-AVX-512 instructions. The ZEN is an AMD EPYC 7401P CPU at 2 GHz with a similar software setup. For each of these two architectures, the number of generated microbenchmarks, resources found and mapped instructions are gathered in Table II.

A. Calibration of the Model

The port mapping is computed using the algorithm presented in Section V using a list of x86 instructions extracted from Intel's XED [11]. We discard instructions which cannot be instrumented in practice, such as instruction modifying the control flow (as our microbenchmark generator cannot handle non-trivial control flow in the instrumented instructions), privileged instructions, along with instructions whose IPC is lower than 0.05, as they do not present any interest for performance prediction of throughput-limited microkernels. While benchmarking memory instructions, we ensure that every access hits the L1 cache to avoid cache-related bottlenecks, which are out of Palmed's scope. Due to the complexity of the x86 instruction set, we separate the SSE and AVX instructions from the "base ISA": we apply separately the heuristics of Sec. 1 before gathering all selected instructions in a single combined *basic instructions*' set as described in Fig. 3.

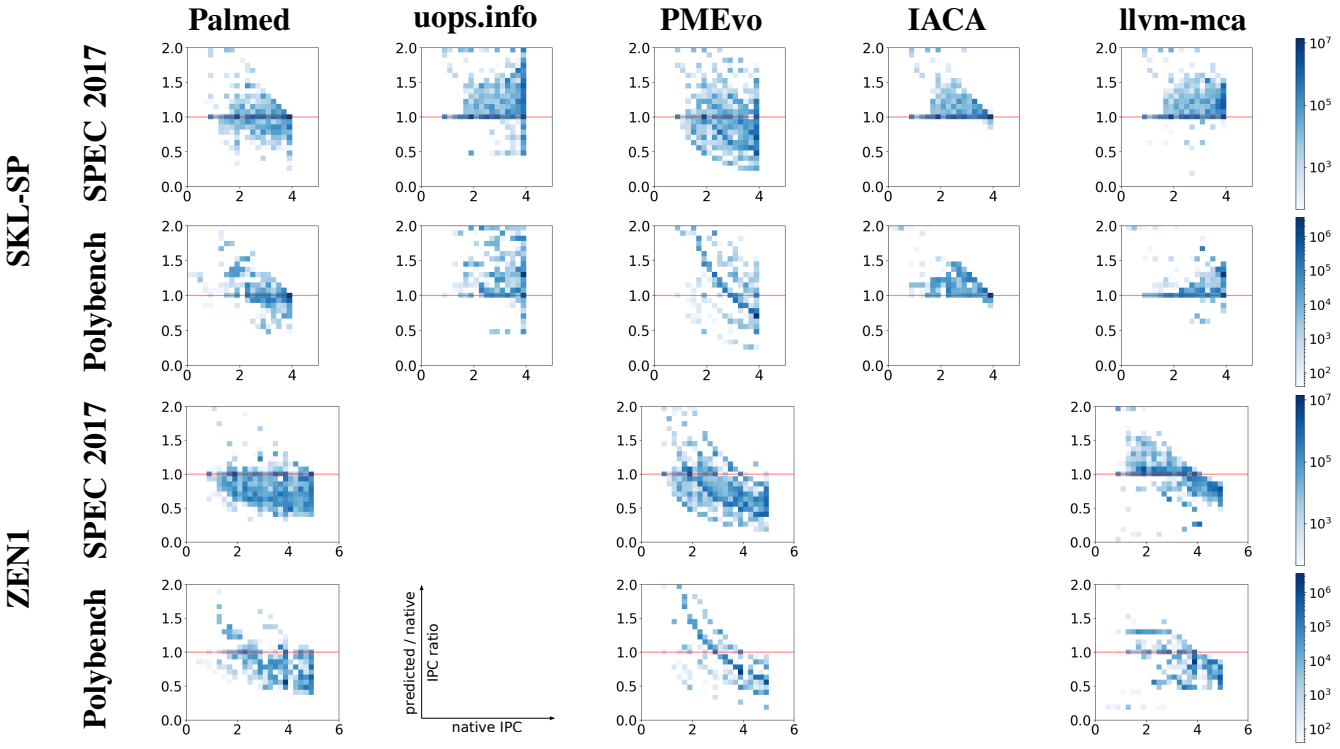
We also forbid benchmarks combining different extensions (e.g. SSE+AVX). Indeed, combinations of several vector extensions of different width are known to cause extra latency, that is, a sort of dependency from one instruction to the other (two consecutive SSE instructions would not be penalized, whereas one SSE and one AVX will). This violates our assumption that the relative order of instruction does not matter, and in practice we observed a significant degradation of the mapping without this mitigation.

Because of variations in the real-world measurements, we constrain the error rate to 0.05 for the micro-benchmark coefficient, meaning 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 $a^1 b^{20}$. Note that in the BWP defined in Algorithm 4, we use the rounded coefficients and not the ideal ones. The IPC is also rounded accordingly. Note that our microbenchmark generator is pre-constrained with these limitations; therefore we did not evaluate Palmed with another measurement back-end – although we expect similar results as we ensured to have reproducible execution times.

B. Throughput Estimations

To evaluate Palmed, the same microkernel is run: (1) natively on each CPU, with the IPC measured with `CPU_CLK_UNHALTED`; (2) using our mapping with abstract resources corresponding to the actual machine, as described in Section VI-A; (3) using Abel's work (uops.info) [4], by running a conjunctive mapping with exact compatibility and approximating the execution time by the port with the highest usage; (4) using PMEvo [29], ignoring any instruction not supported by its provided mapping; (5) using IACA [17], by inserting assembly markers around the kernel and running the tool; (6) using llvm-mca [33], by inserting markers in the assembly code generated by our back-end and running the tool with this assembly.

Unlike PMEvo and llvm-mca, UOPS and IACA do not support the ZEN1 architecture; hence the absence of data.



(a) IPC prediction profile heatmaps – predictions closer to the red line are more accurate. Predicted IPC ratio (Y) against native IPC (X)

(b) Translation block coverage (Cov.), root-mean-square error on IPC predictions (Err.) and Kendall's tau correlation coefficient (τ_K) compared to native execution

		PMD			uops.info			PMEvo			IACA			llvm-mca		
	Unit	Cov. (%)	Err. (%)	τ_K (1)	Cov. (%)	Err. (%)	τ_K (1)	Cov. (%)	Err. (%)	τ_K (1)	Cov. (%)	Err. (%)	τ_K (1)	Cov. (%)	Err. (%)	τ_K (1)
SKL-SP	**SPEC2017**	N/A	7.8	0.90	99.9	40.3	0.71	71.3	28.1	0.47	100.0	8.7	0.80	96.8	20.1	0.73
	Polybench	N/A	24.4	0.78	100.0	68.1	0.29	66.8	46.7	0.14	100.0	15.1	0.67	99.5	15.3	0.65
ZEN1	**SPEC2017**	N/A	29.9	0.68	N/A	N/A	N/A	71.3	36.5	0.43	N/A	N/A	N/A	96.8	33.4	0.75
	Polybench	N/A	32.6	0.46	N/A	N/A	N/A	66.8	38.5	0.11	N/A	N/A	N/A	99.5	28.6	0.40

Fig. 4: Accuracy of IPC predictions compared to native execution of Palmed versus uops.info, PMEvo, IACA and llvm-mca on SPEC CPU2017 and PolyBench/C 4.2

The microkernels are extracted from two well-known benchmark suites: SPECInt2017 [7] and Polybench [27]. For Polybench, we used QEMU [2] to gather the translation blocks 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 generate 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).

Fig. 4 synthesizes our results in two pieces. First, Fig. 4a displays the results as a heatmap for each basic block, comparing the predicted throughput with the measured one. A dark area at coordinate (x, y) means that the selected tool has a prediction accuracy of y for a significant number of microkernels with a real IPC of x.

Then, Table 4b synthesizes, for each tool, its error rate, aggregated over all the basic blocks of the test suite using a *Root-Mean-Square* method:

$$\text{Err}_{\text{RMS, tool}} = \sqrt{\sum_i \frac{\text{weight}_i}{\sum_j \text{weight}_j} \left(\frac{\text{IPC}_{i,\text{tool}} - \text{IPC}_{i,\text{native}}}{\text{IPC}_{i,\text{native}}} \right)^2}$$

We also provide Kendall's τ coefficient [19], a measure of the rank correlation of a predictor – that is, for each pair of basic blocks, whether a predictor predicted correctly which block had the higher IPC. The coefficient varies between -1 (full anti-correlation) and 1 (full correlation).

The same table also provides a *coverage* metric, *with respect to Palmed*. This metric characterizes the proportion of basic blocks supported by Palmed that the tool was able to process. Note that the ability to process a basic block varies from tool to tool: some work in degraded mode when meeting

instructions they cannot handle, some will crash on the basic block. For PMEvo, we ignored any instruction not supported by their mapping – degrading the quality of the result; hence, a plain error is a basic block in which *no single instruction was supported*. Although it would be fairer to other tools to measure absolute coverage – that is, the proportion of basic blocks supported by the tool, regardless of what Palmed supports –, technical limitations prevented us from doing so: running the various tools requires our back-end to generate assembly code, which can only be done for the instructions it supports.

TABLE II: MAIN FEATURES OF THE OBTAINED MAPPINGS

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	10h	8h
Gen. microbenchmarks	$\sim$ 1,000,000	$\sim$ 1,000,000
Resources found	17	17
uops' inst. supported	3313	1104
Instructions mapped	2586	2596

We compare the number of instructions supported by Palmed with the ones supported by uops.info as a baseline. As uops supports only partially AMD's architecture (providing only throughput and latencies, but no usable port mapping), less than half the instructions supported by our tool are present for this target. Contrarily, on SKL-SP, uops supports the AVX-512 extension, therefore leading to a more complete set of supported instructions. PMEvo's mapping behaves poorly in terms of coverage (see Fig. 4b), failing to support all instructions in more than 25 % of the basic blocks on any benchmark and processor tested. This behavior is due to our different compilation options, as PMEvo's supported instructions are directly collected from their SPEC2017 binaries. As a consequence, both MSE and Kendall's tau values are lower than other tools as those unsupported instructions are treated as if they took no resource at all on our IPC estimates.

Moreover, Palmed requires 2h of solving time (see Tbl. II) to map about 2500 instructions. This is between one half (SKL-SP) and one eighth (Zen) of PMEvo's solving time [4], demonstrating the scalability of Palmed with respect to the number of instructions.

In Fig. 4, we observe that Palmed performs significantly better than uops.info and PMEvo on both platforms. On Skylake, it outperforms all other tested tools in terms of Kendall's tau, and compares well with IACA and LLVM-MCA, archiving sub-10 % mean square error rate on SPEC2017. However, those two last tools use manual expertise and are tailored for a platform, whereas our tool is fully automated and generic.

On Zen1, Palmed is comparable to LLVM-MCA, but shows a greater error rate than on Intel. This is due to the internal organization of the Zen microarchitecture, which uses a separated pipeline for integer/control flow and floating point/vector operations. As Palmed tries to minimize the number of resources, this separation is not properly detected, leading to IPC predictions lower than the actual value as seen on the heatmaps on Fig. 4a.

More generally, IACA, uops.info and LLVM-MCA tend to over-estimate the IPC, which is due to their port-based approach: bottlenecks coming from neither ports nor front-end limitations are not taken into account, leading to higher IPC estimations for microkernels where other resources are bottlenecking. Contrarily, benchmarking-based approaches (Palmed and PMEvo) present both under and over approximations as they are based on real-life execution, where all bottlenecks are present. Note that Palmed, IACA, LLVM-MCA (Zen1 only) and PMEvo (Zen only) also express the front-end bottleneck: the limit on the maximal number of instructions being decoding in one cycle (no over-approximation of microkernels with high IPC), that is, a maximal IPC of 4 on SKL-SP and 5 for Zen1. Therefore, we expect Palmed (and PMEvo) to have maximal error rate on benchmarks with few instructions, case in which some undetected / wrongly detected common resource will have higher importance, whereas LLVM-MCA, uops.info and IACA will tend to be more fragile on long microkernels with possible non-port related resources – especially memory ones.

VII. CONCLUSION

We presented Palmed which automatically builds a resource mapping for CPU instructions. This allows to model not only execution port usage, but also other limiting resources, such as the front-end 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 demonstrated on one Intel and one AMD high-performance CPUs that Palmed generates automatically practical port mappings that compare favorably with state-of-the-art systems like IACA and uops.info that either use performance counters or manual expertise.

ACKNOWLEDGMENTS

The works have been funded by ECSEL-JU under the program ECSEL-Innovation Actions-2018 (ECSEL-IA) for research project CPS4EU (ID-826276) in the area Cyber-Physical Systems. This work was also supported in part by the U.S. National Science Foundation award CCF-1750399.

REFERENCES

[1] [Online]. Available: https://www.dropbox.com/s/zvsnj4wsx0fj775/PMD-full.pdf?dl=0
[2] "QEMU: the FAST! processor emulator," https://www.qemu.org.
[3] A. Abel and J. Reineke, "nanoBench: A low-overhead tool for running microbenchmarks on x86 systems," *arXiv e-prints*, vol. abs/1911.03282, 2019. [Online]. Available: http://arxiv.org/abs/1911.03282

[4] ——, "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*, I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, Eds. New York, NY, USA: ACM, April 2019, pp. 673–686. [Online]. Available: https://doi.org/10.1145/3297858.3304062

[5] ——, "Accurate throughput prediction of basic blocks on recent intel microarchitectures," 2021.

[6] J. H. Ahn, S. Li, S. O, and N. P. Jouppi, "McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling," in *2012 IEEE International Symposium on Performance Analysis of Systems and Software*. Austin, TX, USA: IEEE Computer Society, April 2013, pp. 74–85. [Online]. Available: https://doi.org/10.1109/ISPASS.2013.6557148

[7] J. Bucek, K. Lange, and J. von Kistowski, "SPEC CPU2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE 2018*, K. Wolter, W. J. Knottenbelt, A. van Hoorn, and M. Nambiar, Eds. ACM, April 2018, pp. 41–42. [Online]. Available: https://doi.org/10.1145/3185768.3185771

[8] C. Chatelet, C. Courbet, O. Sykora, and N. Paglieri. Google exegesis. [Online]. Available: https://llvm.org/docs/CommandGuide/llvm-exegesis.html

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

[10] I. Corporation. Intel 64 and ia-32 architectures optimization reference manual. [Online]. Available: https://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf

[11] ——. Intel x86 encoder decoder (intel xed). [Online]. Available: https://github.com/intelxed/xed

[12] L. Djoudi, J. Noudohouenou, and W. Jalby, "The design and architecture of MAQAOAdvisor: A live tuning guide," in *Proceedings of the 15th International Conference on High Performance Computing*, ser. HiPC 2008, P. Sadayappan, M. Parashar, R. Badrinath, and V. K. Prasanna, Eds., vol. 5374. Berlin, Heidelberg: Springer-Verlag, December 2008, pp. 42–56. [Online]. Available: https://doi.org/10.1007/978-3-540-89894-8_8

[13] A. Fog. (2020) Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, AMD and VIA CPUs. [Online]. Available: http://www.agner.org/optimize/instruction_tables.pdf

[14] F. Franchetti, T. M. Low, D. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Püschel, J. C. Hoe, and J. M. F. Moura, "SPIRAL: Extreme performance portability," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1935–1968, 2018. [Online]. Available: https://doi.org/10.1109/JPROC.2018.2873289

[15] T. Granlund. (2017) Instruction latencies and throughput for AMD and intel x86 processors. [Online]. Available: https://gmplib.org/~tege/x86-timing.pdf

[16] J. Hammer, J. Eitzinger, G. Hager, and G. Wellein, "Kerncraft: A tool for analytic performance modeling of loop kernels," in *Tools for High Performance Computing 2016*, vol. abs/1702.04653. Cham: Springer International Publishing, 2017, pp. 1–22.

[17] I. Hirsh and G. S. Intel® architecture code analyzer. [Online]. Available: https://software.intel.com/en-us/articles/intel-architecture-code-analyzer

[18] instlatx64. x86, x64 instruction latency, memory latency and cpuid dumps. [Online]. Available: http://instlatx64.atw.hu/

[19] M. G. Kendall, "A new measure of rank correlation," *Biometrika*, vol. 30, no. 1/2, pp. 81–93, 1938.

[20] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *2nd IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2004)*. San Jose, CA, USA: IEEE Computer Society, March 2004, pp. 75–88. [Online]. Available: https://doi.org/10.1109/CGO.2004.1281665

[21] J. Laukemann, J. Hammert, J. Hofmann, G. Hager, and G. Wellein, "Automated instruction stream throughput prediction for intel and AMD microarchitectures," in *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. Dallas, TX, USA: IEEE Computer Society, ACM, November 2018, pp. 121–131.

[22] G. H. Loh, S. Subramaniam, and Y. Xie, "Zesto: A cycle-level simulator for highly detailed microarchitecture exploration," in *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2009*. Boston, Massachusetts, USA: IEEE Computer Society, April 2009, pp. 53–64. [Online]. Available: https://doi.org/10.1109/ISPASS.2009.4919638

[23] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillón, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, M. Fariborz, A. F. Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. S. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and É. F. Zulian, "The gem5 simulator: Version 20.0+," 2020. [Online]. Available: https://arxiv.org/abs/2007.03152

[24] G. Marin, J. J. Dongarra, and D. Terpstra, "MIAMI: A framework for application performance diagnosis," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014*. Monterey, CA, USA: IEEE Computer Society, March 2014, pp. 158–168. [Online]. Available: https://doi.org/10.1109/ISPASS.2014.6844480

[25] C. Mendis, A. Renda, S. P. Amarasinghe, and M. Carbin, "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*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. Long Beach, California, USA: PMLR, June 2019, pp. 4505–4515. [Online]. Available: http://proceedings.mlr.press/v97/mendis19a.html

[26] F. Nielsen, *Hierarchical Clustering*, 02 2016, pp. 195–211.

[27] L.-N. Pouchet and T. Yuki, "PolyBench/C: The polyhedral benchmark suite, version 4.2," 2016, http://polybench.sf.net.

[28] G. C. Project. (1987) GNU compiler collection (gcc). [Online]. Available: https://gcc.gnu.org/

[29] F. Ritter and S. Hack, "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*, A. F. Donaldson and E. Torlak, Eds. New York, USA: ACM, June 2020, pp. 608–622. [Online]. Available: https://doi.org/10.1145/3385412.3385995

[30] A. C. Rubial, E. Oseret, J. Noudohouenou, W. Jalby, and G. Lartigue, "CQA: A code quality analyzer tool at binary level," in *21st International Conference on High Performance Computing, HiPC 2014*. Goa, India: IEEE Computer Society, December 2014, pp. 1–10. [Online]. Available: https://doi.org/10.1109/HiPC.2014.7116904

[31] ——, "CQA: A code quality analyzer tool at binary level," in *21st International Conference on High Performance Computing, HiPC 2014*. Goa, India: IEEE Computer Society, December 2014, pp. 1–10. [Online]. Available: https://doi.org/10.1109/HiPC.2014.7116904

[32] D. Sánchez and C. Kozyrakis, "ZSim: fast and accurate microarchitectural simulation of thousand-core systems," in *40th Annual International Symposium on Computer Architecture, (ISCA'13)*, A. Mendelson, Ed. New York, NY, USA: ACM, June 2013, pp. 475–486. [Online]. Available: https://doi.org/10.1145/2485922.2485963

[33] Sony Corporation and L. Project. LLVM machine code analyzer. [Online]. Available: https://llvm.org/docs/CommandGuide/llvm-mca.html

[34] C. Topper. (2018, Mar.) Update to the llvm scheduling model for intel sandy bridge, haswell, broadwell, and skylake processors. [Online]. Available: https://github.com/llvm/llvm-project/commit/cdfcf8ecda8065fda495d73ed16277668b3b56dc

[35] J. Treibig, G. Hager, and G. Wellein, "LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments," in *39th International Conference on Parallel Processing (ICPP) Workshops 2010*, W. Lee and X. Yuan, Eds. San Diego, California, USA: IEEE Computer Society, September 2010, pp. 207–216. [Online]. Available: https://doi.org/10.1109/ICPPW.2010.38

[36] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009. [Online]. Available: http://doi.acm.org/10.1145/1498765.1498785

[37] M. T. Yourst, "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator," in *2007 IEEE International Symposium on Performance Analysis of Systems and Software*. San Jose, California,

USA: IEEE Computer Society, April 2007, pp. 23–34. [Online]. Available: https://doi.org/10.1109/ISPASS.2007.363733

[38] F. G. V. Zee and R. A. van de Geijn, "BLIS: a framework for rapidly instantiating BLAS functionality," *ACM Transactions on Mathematical Software*, vol. 41, no. 3, June 2015. [Online]. Available: https://doi.org/10.1145/2764454