THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Université Grenoble Alpes

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Informatique
Unité de recherche : Centre de recherche Inria de l'Université Grenoble Alpes

Vers une caractérisation automatique du comportement micro-architectural pour la modélisation des performances de noyaux de calcul : une analyse des micro-architectures Cortex A72 et Intel

Towards automatic characterization of microarchitectural behaviour for performance modeling of computing kernels: an analysis of the Cortex A72 and Intel microarchitectures

Présentée par :

Théophile BASTIAN

Direction de thèse :

Fabrice RASTELLO — Directeur de thèse
DIRECTEUR DE RECHERCHE, INRIA CENTRE GRENOBLE-RHONE-ALPES

Rapporteurs :

KEVIN MARTIN
PROFESSEUR DES UNIVERSITES, UNIVERSITE BRETAGNE SUD - LORIENT VANNES
GABRIEL RODRIGUEZ
ASSOCIATE PROFESSOR, UNIVERSIDAD DA CORUÑA

Thèse soutenue publiquement le **9 décembre 2024**, devant le jury composé de :

LAURENCE PIERRE, — Présidente
PROFESSEUR DES UNIVERSITES, UNIVERSITE GRENOBLE ALPES
FABRICE RASTELLO, — Directeur de thèse
DIRECTEUR DE RECHERCHE, CENTRE INRIA UNIVERSITE GRENOBLE ALPES
KEVIN MARTIN, — Rapporteur
PROFESSEUR DES UNIVERSITES, UNIVERSITE BRETAGNE SUD - LORIENT VANNES
GABRIEL RODRIGUEZ, — Rapporteur
ASSOCIATE PROFESSOR, UNIVERSIDAD DA CORUÑA
GAËL THOMAS, — Examinateur
DIRECTEUR DE RECHERCHE, CENTRE INRIA DE SACLAY

Version `6b0539a`

Résumé

Qu'il s'agisse de calculs massifs distribués sur plusieurs baies, de calculs en environnement contraint — comme de l'embarqué ou de l'*edge computing* — ou encore de tentatives de réduire l'empreinte écologique d'un programme fréquemment utilisé, de nombreux cas d'usage justifient l'optimisation poussée d'un programme. Celle-ci s'arrête souvent à l'optimisation de haut niveau (algorithmique, parallélisme, …), mais il est possible de la pousser jusqu'à une optimisation bas-niveau, s'intéressant à l'assembleur généré en regard de la microarchitecture du processeur précis utilisé.

Une telle optimisation demande une compréhension fine des aspects à la fois logiciels et matériels en jeu, et n'est bien souvent accessible qu'aux experts du domaine. Les *code analyzers* (analyseurs de code), cependant, permettent d'abaisser le niveau d'expertise nécessaire pour accomplir de telles optimisations, en automatisant une partie du travail de compréhension des problèmes de performance rencontrés. Ces mêmes outils permettent également aux experts d'être plus efficaces dans leur travail.

Dans ce manuscrit, nous étudierons les principaux facteurs limitants de la performance d'un processeur, sur lesquels la précision des outils de l'état de l'art est bien souvent inégale. Nous apportons, sur chacun de ces facteurs limitants, une contribution nouvelle : automatisation de l'obtention d'un modèle du *backend*, étude manuelle du *frontend* en vue de l'automatisation de son modèle, et extraction automatique des dépendances *à travers la mémoire* d'un noyau de calcul. Nous apportons également une étude systématique et automatisée des performances de prédiction de différents *code analyzers* de l'état de l'art.

Abstract

Be it massively distributed computation over multiple server racks, constrained computation — such as in embedded environments or in *edge computing* —, or still an attempt to reduce the ecological footprint of a frequently-run program, many use-cases make it relevant to deeply optimize a program. This optimisation is often limited to high-level optimisation — choice of algorithms, parallel computing, … Yet, it is possible to carry it further to low-level optimisations, by inspecting the generated assembly with respect to the microarchitecture of the specific microprocessor used to fine-tune it.

Such an optimisation level requires a very detailed understanding of both the software and hardware aspects implied, and is most often the realm of experts. *Code analyzers*, however, are tools that help lowering the expertise threshold required to perform such optimisations by automating away a portion of the work required to understand the source of the encountered performance problems. The same tools are also useful to experts, as they help them be more efficient in their work.

In this manuscript, we study the main performance bottlenecks of a processor, on which the state of the art does not perform consistently. For each of these bottlenecks, we contribute to the state of the art. We work on automating the obtention of a model of the processor's *backend*; we manually study the processor's *frontend*, hoping to set a milestone towards the automation of the obtention of such models; we provide a tool to automatically extract a computation kernel's *memory-carried* dependencies. We also provide a systematic, automated and fully-tooled study of the prediction accuracy of various state-of-the-art code analyzers.

Remerciements

Cette thèse n'aurait très certainement pas été la même sans la présence à mes côtés et l'aide précieuse de nombreuses personnes, que je souhaite remercier chaleureusement ici.

À commencer, bien évidemment, par Fabrice, sans qui ce manuscrit n'aurait tout bonnement pas existé, et qui m'a accompagné dans mon parcours, de la découverte de ce domaine à la soutenance de cette thèse.

Merci également au jury, et encore plus aux deux rapporteurs, Kevin Martin et Gabriel Rodriguez, de s'être plongés dans ce manuscrit. Merci pour vos commentaires positifs et encourageants, mais aussi pour vos retours critiques qui me permettent d'améliorer ce travail.

Plus largement, c'est aussi toute l'équipe CORSE que je souhaite remercier, tant pour les échanges scientifiques constructifs que nous avons pu avoir que pour les conversations plus légères de tous les jours qui ont fait du labo un endroit agréable [1]. Parmi vous, merci tout particulièrement à Nicolas D. pour notre collaboration sur `Palmed` que nous avons su, malgré les confinements et couvre-feux de cette étrange période, rentre motivante et productive. Merci aussi à Hugo P., Valentin T. et Christophe G., avec qui j'ai le plus directement collaboré. Merci encore à Imma P., qui sait si efficacement rendre la charge administrative transparente à nos yeux. Travailler avec vous tous et toutes a été un plaisir.

Mes remerciements vont aussi à la Ville d'Échirolles, et tout particulièrement à Philippe et Nicolas, qui m'ont permis de finir ma thèse dans de bonnes conditions malgré mon emploi à temps plein.

Toutes ces courbes et ces modèles n'auraient pas pu voir le jour sans l'exploitation forcenée de quelques pauvres processeurs, qui, je n'en doute pas, espéraient plutôt couler de doux jours dans un serveur web. Merci donc à Pinocchio, Dracula, les Dahus de Grid5000, et mes sincères excuses à Corse-rpi0 pour les pics de fièvre que je lui ai causés [2].

Il va également sans dire que la présence et le soutien de mes proches de tous horizons m'a été précieux au cours de ces années, et je vous remercie chaleureusement toutes et tous. Au risque d'oublier des gens, je citerai pêle-mêle mes amis de l'ENS et de l'Arcoloc, les choristes et amis des Rainbow et, plus tard, du CUG et les personnes rencontrées à Grésille. Et, surtout, un merci tout particulier à Sarah, Nina et à ma mère, Rosine.

Merci à vous toutes et tous.

1. Même s'il y manquait du vrai café.
2. Et mon manque d'inspiration pour son baptême.

Contents

Notations **8**

Introduction **9**

1 Foundations **12**

1.1 A dive into processors' microarchitecture 12
1.1.1 High-level abstraction of processors 12
1.1.2 Microarchitectures 13
1.2 Kernel optimization and code analyzers 16
1.2.1 Code analyzers 17
1.2.2 Examples with `llvm-mca` 18
1.2.3 Definitions 23
1.3 State of the art 28
1.3.1 Manufacturer-sourced data 28
1.3.2 Third-party instruction data 28
1.3.3 Code analyzers and their models 29

2 `Palmed`: automatically modelling the backend **31**

2.1 Resource models 31
2.1.1 Usual representation: tripartite disjunctive graph 31
2.1.2 Dual representation: conjunctive resource mapping 33
2.2 Palmed design 34
2.3 Measuring a kernel's throughput: `Pipedream` 35
2.4 Finding basic blocks to evaluate `Palmed` 36
2.4.1 Benchmark suites 37
2.4.2 Manually extracting basic blocks 37
2.4.3 Automating basic block extraction 37
2.5 Evaluating `Palmed` 39
2.5.1 Evaluation harness 39
2.5.2 Metrics extracted 40
2.5.3 Results 40
2.6 Other contributions 42

3 Beyond ports: manually modelling the A72 frontend **44**

3.1 Necessity to go beyond ports 45
3.2 The Cortex A72 CPU 45
3.3 Manually modelling the A72 frontend 47
3.3.1 Finding micro-operation count for each instruction 47
3.3.2 Bubbles in the pipeline 49
3.4 Evaluation on Palmed 53
3.5 A parametric model for future works of automatic frontend model generation . 53

4 A more systematic approach to throughput prediction performance analysis: `CesASMe` 57
4.1 Re-defining the execution time of a kernel 58
4.2 Related works 59
4.3 Generating microbenchmarks 60
4.3.1 Benchmark suite 60
4.3.2 C-to-C loop nest optimizers 61
4.3.3 Constraining utility 61
4.3.4 C-to-binary compiler 61
4.4 Benchmarking harness 61
4.4.1 Basic block extraction 62
4.4.2 Throughput predictions and measures 62
4.4.3 Prediction lifting and filtering 62
4.5 Experimental setup and evaluation 63
4.5.1 Experimental environment 63
4.5.2 Comparability of the results 63
4.5.3 Relevance and representativity (bottleneck analysis) 65
4.5.4 Carbon footprint 65
4.6 Results analysis 66
4.6.1 Throughput results 66
4.6.2 Understanding `BHive`'s results 66
4.6.3 Bottleneck prediction 68
4.6.4 Impact of dependency-boundness 69

5 Static extraction of memory-carried dependencies 73
5.1 Types of dependencies 73
5.2 A baseline: dynamic dependencies detection with `valgrind` 75
5.2.1 Valgrind 75
5.2.2 Depsim 75
5.3 Static dependencies detection 76
5.3.1 Far-reaching dependencies do not impact performance 77
5.4 Staticdeps 79
5.4.1 The `staticdeps` heuristic 79
5.4.2 Practical implementation 81
5.4.3 Limitations 81
5.5 Evaluation 82
5.5.1 Comparison to `depsim` results 82
5.5.2 Enriching `uiCA`'s model 84
5.5.3 Analysis speed 86

6 Wrapping it all up 88
6.1 Critical path model 88
6.2 Evaluation 88
6.3 Towards a modular approach? 90

Conclusion 91

Bibliography 94

Notations

Throughout this whole document, the following non-standard notations are used.

Notation	Meaning	(See also)
$\overline{\mathcal{K}}$	Reciprocal throughput of $\mathcal{K}$, in cycles per occurrence of $\mathcal{K}$.	§1.2.3
$\overline{\mathcal{K}}^{M(n)}$	Measured reciprocal throughput of $\mathcal{K}$, over n iterations of $\mathcal{K}$. When there is no ambiguity and n is sufficiently large, we often write $\overline{\mathcal{K}}$ instead.	§1.2.3
$\overline{\mathcal{K}}^{\mathbf{B}}$	Reciprocal throughput of $\mathcal{K}$ if it was only limited by the CPU's backend.	§3.3.1
$\overline{\mathcal{K}}^{\mathbf{F}}$	Reciprocal throughput of $\mathcal{K}$ if it was only limited by the CPU's frontend.	§3.3.1
$C(\mathcal{K})$	Number of cycles of a kernel $\mathcal{K}$.	§1.2.3
$\mathcal{K}^n$	$\mathcal{K}$ repeated n times.	§1.2.3
IPC($\mathcal{K}$)	Instructions Per Cycle in the execution of the kernel $\mathcal{K}$, in steady state, averaged.	§1.2.3
$\#_\mu i$	Number of μOPs the instruction i is decoded into. This can be extended to a kernel: $\#_\mu \mathcal{K}$.	§3.3.1
τ_K	Kendall's τ coefficient of correlation.	§2.5.2, [Ken38]

Introduction

Developing new features and fixing problems are often regarded as the major parts of the development cycle of a program. However, performance optimization might be just as crucial for compute-intensive software. On small-scale applications, it improves usability by reducing, or even hiding, the waiting time the user must endure between operations, or by allowing heavier workloads to be processed without needing larger resources or in constrained embedded hardware environments. On large-scale applications, that may run for an extended period of time, or may be run on whole clusters, optimization is a cost-effective path, as it allows the same workload to be run on smaller clusters, for reduced periods of time.

The most significant optimisation gains come from "high-level" algorithmic changes, such as computing on multiple cores instead of sequentially, caching already computed results, reimplementing a function to run asymptotically in $\mathcal{O}\left(n \cdot \log(n)\right)$ instead of $\mathcal{O}\left(n^2\right)$ or avoiding the copy of large data structures. However, when a software is already well-optimized from these perspectives, the impact of low-level considerations, stemming from the hardware implementation of the machine itself, cannot be neglected anymore. A common example of such impacts is the iteration of a large matrix either row-major or column-major:

```
sum ← 0
for row < MAX_ROW do
    for col < MAX_COLUMN do
        sum ← sum + matrix[row][col]
```

```
sum ← 0
for col < MAX_COLUMN do
    for row < MAX_ROW do
        sum ← sum + matrix[row][col]
```

While both programs are performing the exact same computation, the left one iterates on rows first, or *row-major*, while the right one iterates on columns first, or *column-major*. The latter, on large matrices, will cause frequent cache misses, and was measured to run up to about six times slower than the former [Bas23].

This, however, is still an optimization that holds for the vast majority of CPUs. In many cases, transformations targeting a specific microarchitecture can be very beneficial. For instance, Uday Bondhugula found out that manual tuning, through many techniques and tools, of a general matrix multiplication could multiply its throughput by roughly 13.5 compared to `gcc -O3`, or even be 130 times faster than `clang -O3` [Bon20]. This kind of optimizations, however, requires manual effort, and a deep expert knowledge both in optimization techniques and on the specific architecture targeted. These techniques are only worth applying on the parts of a program that are most executed — usually called the *hottest* parts —, loop bodies that are iterated enough times to be assumed infinite. Such loop bodies are called *computation kernels*, with which this whole manuscript will be concerned.

Developers are used to *functional debugging*, the practice of tracking the root cause of an unexpected bad functional behaviour. Akin to it is *performance debugging*, the practice of tracking the root cause of a performance below expectations. Just as functional debugging can be carried in a variety of ways, from guessing and inserting print instructions to sophisticated tools such as `gdb`, performance debugging can be carried with different tools. Crude timing measures and profiling can point to a general part of the program or hint an issue; reading *hardware counters* — metrics reported by the CPU — can lead to a better understanding, and

may confirm or invalidate an hypothesis. Other tools still, *code analyzers*, analyze the assembly code and, in the light of a built-in hardware model, strive to provide a performance analysis.

An exact modelling of the processor would require a cycle-accurate simulator, reproducing the precise behaviour of the silicon, allowing one to observe any desired metric. Such a simulator, however, would be prohibitively slow, and is not available on most architectures anyway, as processors are not usually open hardware and the manufacturer regards their implementation as industrial secret. Code analyzers thus resort to approximated, higher-level models of varied kinds. Tools based on such models, as opposed to measures or hardware counters sampling, may not always be precise and faithful. They can, however, inspect at will their inner model state, and derive more advanced metrics or hypotheses, for instance by predicting which resource might be overloaded and slow the whole computation.

In this thesis, we explore the three major aspects that work towards a code analyzer's accuracy: a *backend model*, a *frontend model* and a *dependencies model*. We propose contributions to strengthen them, as well as to automate the underlying models' synthesis. We focus on *static* code analyzers, that derive metrics, including runtime predictions, from an assembly code or assembled binary without executing it.

The first chapter introduces the foundations of this manuscript, describing the microarchitectural notions on which our analyses will be based, and exploring the current state of the art.

The chapter 2 introduces `Palmed`, a benchmarks-based tool automatically synthesizing a model of a CPU's backend. Although the theoretical core of `Palmed` is not my own work, I made major contributions to other aspects of the tool. The chapter also presents the foundations and methodologies `Palmed` shares with the following parts.

In chapter 3, we explore the frontend aspects of static code analyzers. This chapter focuses on the manual study of the Cortex A72 processor, and proposes a static model of its frontend. We finally reflect on the generalization of our manual approach into an automated frontend modelling tool, akin to `Palmed`.

Chapter 4 makes an extensive study of the state-of-the-art code analyzers' strengths and shortcomings. To this end, we introduce a fully-tooled approach in two parts: first, a benchmark-generation procedure, yielding thousands of benchmarks relevant in the context of our approach; then, a benchmarking harness evaluating code analyzers on these benchmarks. We find that most state-of-the-art code analyzers struggle to correctly account for some types of data dependencies.

Further building on our findings, chapter 5 introduces `staticdeps`, an accurate heuristic-based tool to statically extract data dependencies from an assembly computation kernel. We extend `uiCA`, a state-of-the-art code analyzer, with `staticdeps` predictions, and evaluate the enhancement of its accuracy.

Throughout this manuscript, we explore notions that are transversal to the hardware blocks the chapters lay out.

Most of our approaches work towards an *automated, microarchitecture-independent* tooling. While fine-grained, accurate code analysis is directly concerned with the underlying hardware and its specific implementation, we strive to write tooling that has the least dependency towards vendor-specific interfaces. In practice, this rules out most uses of hardware counters, which depend greatly on the manufacturer, or even the specific chip considered. As some CPUs expose only very bare hardware counters, we see this commitment as an opportunity to develop methodologies able to model these processors.

This is particularly true of `Palmed`, in chapter 2, whose goal is to model a processor's backend resources without resorting to its vendor-specific hardware counters. Our frontend study, in chapter 3, also follows this strategy by focusing on a processor whose hardware counters give

little to no insight on its frontend. While this goal is less relevant to `staticdeps`, we rely on external libraries to abstract the underlying architecture.

Our methodologies are, whenever relevant, *benchmarks- and experiments-driven*, in a bottom-up style, placing real hardware at the center. In this spirit, `Palmed` is based solely on benchmarks, discarding entirely the manufacturer's documentation. Our model of the Cortex A72 frontend is based both on measures and documentation, yet it strives to be a case study from which future works can generalize, to automatically synthesize frontend models in a benchmarks-based fashion. One of the goals of our survey of the state of the art, in chapter 4, is to identify through experiments the shortcomings that are most crucial to address in order to strengthen static code analyzers.

Finally, against the extent of the ecological and climatic crises we are facing, as assessed among others by the IPCC [Con23], we believe that every field and discipline should strive for a positive impact or, at the very least, to reduce as much as possible its negative impact. Our very modest contribution to this end, throughout this thesis, is to commit ourselves to computations as *frugal* as possible: run computation-heavy experiments as least as possible; avoid running multiple times the same experiment, but cache results instead when this is feasible; etc. This commitment partly motivated us to implement a results database in `Palmed`, to compute only once each benchmark. As our experiments in chapter 4 take many hours to yield a result, we at least evaluate their carbon impact.

We believe it noteworthy, however, to point out that although this thesis is concerned with tools that help optimize large computation workloads, *optimization does not lead to frugality.* In most cases, Jevons paradox — also called rebound effect — makes it instead more likely to lead to an increased absolute usage of computational resources [Jev66; YM16].

Chapter 1

Foundations

Code analyzers lay at the boundary of program analysis and microarchitectural knowledge. In order to understand their internal models, and how they derive performance metrics from them, notions on both of those worlds are needed.

This first chapter aims to lay the foundations for this manuscript. To this end, we first go over a coarse-grained view of the microarchitectural details of the parts of a modern processor relevant to our works. We then introduce notions and metrics on program analysis that will be used throughout this thesis. Finally, we summarize the current state of the art in the field of code analyzers, and relevant neighbouring topics.

1.1 A dive into processors' microarchitecture

A modern computer can roughly be broken down into a number of functional parts: a processor, a general-purpose computation unit; accelerators, such as GPUs, computation units specialized on specific tasks; memory, both volatile but fast (RAM) and persistent but slower (SSD, HDD); hardware specialized for interfacing, such as networks cards or USB controllers; power supplies, responsible for providing smoothed, adequate electric power to the previous components.

This manuscript will largely focus on the processor. While some of the techniques described here might possibly be used for accelerators, we did not experiment in this direction, nor are we aware of efforts in this direction.

1.1.1 High-level abstraction of processors

A processor, in its coarsest view, is simply a piece of hardware that can be fed with a flow of instructions, which will, each after the other, modify the machine's internal state.

The processor's state, the available instructions themselves and their effect on the state are defined by an *Instruction Set Architecture*, or ISA; such as x86-64 or A64 (ARM's ISA). More generally, the ISA defines how software will interact with a given processor, including the registers available to the programmer, the instructions' semantics — broadly speaking, as these are often informal —, etc. These instructions are represented, at a human-readable level, by *assembly code*, such as `add (%rax), %rbx` in x86-64. Assembly code is then transcribed, or *assembled*, to a binary representation in order to be fed to the processor — for instance, `0x480318` for the previous instruction. This instruction computes the sum of the value held at memory address `%rax` and of the value `%rbx`, but it does not, strictly speaking, *return* or *produce* a result: instead, its stores the result of the computation in register `%rbx`, altering the machine's state.

This state, generally, is composed of a small number of *registers*, small pieces of memory on which the processor can directly operate — to perform arithmetic operations, index the main memory, etc. It is also composed of the whole memory hierarchy, including the persistent

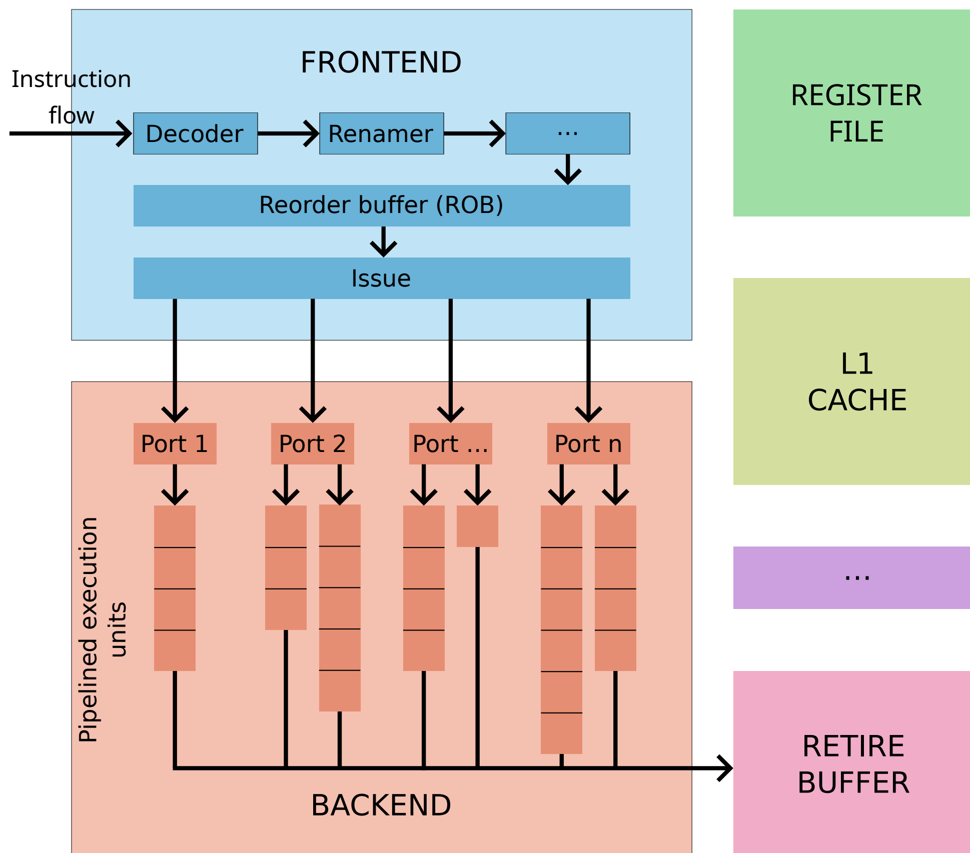


Figure 1.1 – Simplified and generalized global representation of a CPU microarchitecture

memory, the main memory (usually RAM) and the hierarchy of caches between the processor and the main memory. This state can also be extended to encompass external effects, such as networks communication, peripherals, etc.

The way an ISA is implemented, in order for the instructions to alter the state as specified, is called a *microarchitecture*. Many microarchitectures can implement the same ISA, as it is the case for instance with the x86-64 ISA, implemented both by Intel and AMD, each with multiple generations, which translates into multiple microarchitectures. It is thus frequent for ISAs to have many extensions, which each microarchitecture may or may not implement.

1.1.2 Microarchitectures

While many different ISAs are available and used, and even many more microarchitectures are industrially implemented and widely distributed, some generalities still hold for the vast majority of processors found in commercial or server-grade computers. Such a generic view is obviously an approximation and will miss many details and specificities; it should, however, be sufficient for the purposes of this manuscript.

A microarchitecture can be broken down into a few functional blocks, shown in Figure 1.1, roughly amounting to a *frontend*, a *backend*, a *register file*, multiple *data caches* and a *retire buffer*.

Frontend and backend. The frontend is responsible for fetching the flow of instruction bytes to be executed, break it down into operations executable by the backend and issue them

to execution units. The backend, in turn, is responsible for the actual computations made by the processor.

As such, the frontend can be seen as a manager for the backend: the latter actually executes the work, while the former ensures that work is made available to it, orchestrates its execution and scheduling, and ensures each "worker" in the backend is assigned tasks within its skill set.

Register file. The register file holds the processor's registers, small amounts of fast memory directly built into the processor's cores, on which computations are made.

Data caches. The cache hierarchy (usually L1, L2 and L3) caches data rows from the main memory, whose access latency would slow computation down by several orders of magnitude if it was accessed directly. Usually, the L1 cache resides directly in the computation core, while the L2 and L3 caches are shared between multiple cores.

An instruction's walk through the processor

Several CPU cycles may pass from the moment an instruction is first fetched by the processor, until the time this instruction is considered completed and discarded. Let us follow the path of one such instruction through the processor.

The CPU frontend constantly fetches a flow of instruction bytes. This flow must first be broken down into a sequence of instructions. While on some ISAs, each instruction is made of a constant amount of bytes — *eg.* ARM —, this is not always the case: for instance, x84-64 instructions can be as short as one byte, while the ISA only limits an instruction to 15 bytes [23c]. This task is performed by the *decoder*, which usually outputs a flow of *micro-operations*, or μOPs.

Some microarchitectures rely on complex decoding phases, first splitting instructions into *macro-operations*, to be split again into μOPs further down the line. Part of this decoding may also be cached, *eg.* to optimize loop decoding, where the same sequence of instructions will be decoded many times.

Microarchitectures typically implement more physical registers in their register file than the ISA exposes to the programmer. The CPU takes advantage of those additional registers by including a *renamer* in the frontend, to which the just-decoded operations are fed. The renamer maps the ISA-defined registers used explicitly in instructions to concrete registers in the register file. As long as enough concrete registers are available, this phase eliminates certain categories of data dependencies; this aspect is explored briefly below, and later in chapter 5.

Depending on the microarchitecture, the decoded operations — be they macro- or micro-operations at this stage — may undergo several more phases, specific to each processor.

Typically, however, μOPs will eventually be fed into a *Reorder Buffer*, or ROB. Today, most consumer- or server-grade CPUs are *out-of-order*, with effects detailed below; the reorder buffer makes this possible. The μOPs may wait for a few cycles in this reorder buffer, before being pulled by the *issuer*.

Finally, the μOPs are *issued* to the backend towards *execution ports*. Each port usually processes at most one μOP per CPU cycle, and acts as a sort of gateway towards the actual execution units of the processor.

Each execution port may be (and usually is) connected to multiple different execution units: for instance, Intel Skylake's port 6 is responsible for both branch μOPs and integer arithmetics; while ARM's Cortex A72 has a single port for both memory loads and stores.

In most cases, execution units are *fully pipelined*, meaning that while processing a single μOP takes multiple cycles, the unit is able to start processing a new μOP every cycle: multiple μOPs are thus being processed, at different stages, during each cycle, akin to a factory's assembly line.

Finally, when a μOP has been entirely processed and exits its processing unit's pipeline, it is committed to the *retire buffer*, marking the μOP as complete.

Dependencies handling

In this flow of μOPs, some are dependent on the result computed by a previous μOP — or, rather more precisely, await the change of state induced by a previous μOP. If, for instance, two successive identical μOPs compute `%r10` $\leftarrow$ `%r10` $+$ `%r11`, the second instance must wait for the completion of the first one, as the value of `%r10` after the execution of the latter is not known before its completion.

The μOPs that depend on a previous μOP are not *issued* until the latter is marked as completed by entering the retire buffer [1].

Since computation units are pipelined, they reach their best efficiency only when μOPs can be fed to them in a constant flow. Yet, as such, a dependency may block the computation entirely until its dependent result is computed, throttling down the CPU's performance.

The *renamer* helps relieving this dependency pressure when the dependency can be broken by simply renaming one of the registers. We detail this later on chapter 5, but such dependencies may be *eg. write-after-read*: if `%r11` $\leftarrow$ `%r10` is followed by `%r10` $\leftarrow$ `%r12`, then the latter must wait for the former's completion, as it would else overwrite `%r10`, which is read by the former. However, the second instruction may be *renamed* to write to $\texttt{\%r10}_{\text{alt}}$ instead — also renaming every subsequent read to the same value —, thus avoiding the dependency.

Out-of-order vs. in-order processors

When computation is stalled by a dependency, it may however be possible to issue immediately a μOP which comes later in the instruction stream, but depends only on results already available.

For this reason, many processors are now *out-of-order*, while processors issuing μOPs strictly in their original order are called *in-order*. Out-of-order microarchitectures feature a *reorder buffer*, from which instructions are picked to be issued. The reorder buffer acts as a sliding window of microarchitecturally-fixed size over decoded μOPs, from which the oldest μOP whose dependencies are satisfied will be executed. Thus, out-of-order CPUs are only able to execute operations out of order as long as the μOP to be executed is not too far ahead from the oldest μOP awaiting to be issued — specifically, not more than the size of the reorder buffer ahead.

It is also important to note that out-of-order processors are only out-of-order *from a certain point on*: a substantial part of the processor's frontend is typically still in-order.

Hardware counters

Many processors provide *hardware counters*, to help (low-level) programmers understand how their code is executed. The counters available widely depend on each specific processor. The majority of processors, however, offer counters to determine the number of elapsed cycles between two instructions, as well as the number of retired instructions. Some processors further offer counters for the number of cache misses and hits on the various caches, or even the number of μOPs executed on a specific port.

While access to these counters is vendor-dependant, abstraction layers are available: for instance, the Linux kernel abstracts these counters through the `perf` interface, while `PAPI` further attempts to unify similar counters from different vendors under a common name.

1. Some processors, however, introduce "shortcuts" when a μOP can yield a result before its full completion. In such cases, while the μOP depended on is not yet complete and retired, the dependant μOP can still be issued.

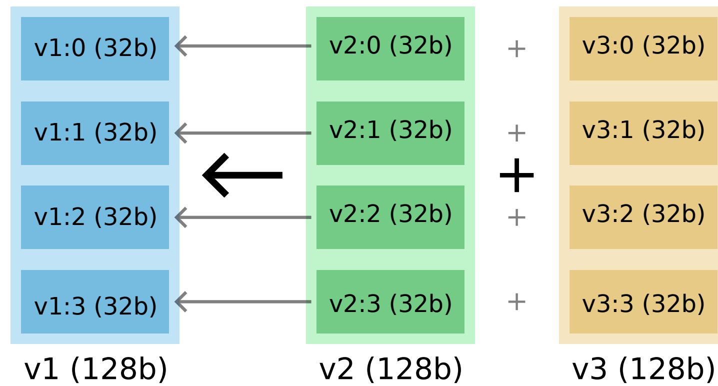


Figure 1.2 – Example of SIMD 4×32 bits add instruction on 128 bits

SIMD operations

Processors operate at a given *word size*, fixed by the ISA — typically 32 or 64 bits nowadays, even though embedded processors might operate at lower word sizes.

Some instructions, however, operate on chunks of multiple words at once. These instructions are called *vector instructions*, or *SIMD* for Single Instruction, Multiple Data. A SIMD "add" instruction may, for instance, add two chunks of 128 bits, treated each as four integers of 32 bits bundled together, as illustrated in Figure 1.2.

Such instructions present clear efficiency advantages. If the processor is able to handle one such instruction every cycle — even if it is pipelined for multiple cycles —, it multiplies by its number of vector elements the processor's throughput, making it able to process *eg.* four add operations per cycle instead of one, as long as the data is arranged in memory in an appropriate way. Some processors, however, are not able to process the full vector instruction at once, by lack of backend units — it may, for instance, only process two 32-bits adds at once, making the processor able to execute only one such instruction per two cycles. Even in this case, there are clear efficiency benefits: while there is no real gain in the backend, the frontend has only one instruction to decode, rename, etc., greatly alleviating frontend pressure. This is for instance the case of the implementation of the RISC-V [Wat+11] vector extension, supporting up to 256 double-precision floats in a single operation, while the hardware supports far less in one cycle [Man+23; Man23].

1.2 Kernel optimization and code analyzers

Optimizing a program, in most contexts, mainly means optimizing it from an algorithmic point of view — using efficient data structures, running some computations in parallel on multiple cores, etc. As pointed out in our introduction, though, optimizations close to the machine's microarchitecture can yield large efficiency benefits, sometimes up to two orders of magnitude [Bon20]. These optimizations, however, are difficult to carry for multiple reasons: they depend on the specific machine on which the code is run; they require deep expert knowledge; they are most often manual, requiring expert time — and thus making them expensive.

Such optimizations are, however, routinely used in some domains. Scientific computation — such as ocean simulation, weather forecast, … — often rely on the same operations, implemented by low-level libraries optimized in such a way, such as OpenBLAS [Xia; Wan+13] or Intel's MKL [Int03], implementing low-level math operations, such as linear algebra. Machine learning applications, on the other hand, may typically be trained for extensive periods of time, on many cores and accelerators, on a well-defined hardware, with small portions of code being executed many times on different data; as such, they are very well suited for such specific and low-level optimizations.

When optimizing those short fragments of code whose efficiency is critical, or *computation kernels*, insights on what limits the code's performance, or *performance bottlenecks*, are precious to the expert. These insights can be gained by reading the processor's hardware counters, described above in section 1.1.2, typically accurate but of limited versatility. Specialized profilers, such as Intel's VTune [Inta], integrate these counters with profiling to derive further performance metrics at runtime.

1.2.1 Code analyzers

Another approach is to rely on *code analyzers*, pieces of software that analyze a code fragment — typically at assembly or binary level —, and provide insights on its performance metrics on a given hardware. Code analyzers thus work statically, that is, without executing the code.

Common hypotheses. Code analyzers operate under a set of common hypotheses, derived from the typical intended usage.

The kernel analyzed is expected to be the body of a loop, or nest of loops, that should be iterated many times enough to be approximated by an infinite loop. The kernel will further be analyzed under the assumption that it is in *steady-state*, and will thus ignore startup or border effects occurring in extremal cases. As the kernels analyzed are those worth optimizing manually, it is reasonable to assume that they will be executed many times, and focus on their steady-state.

The kernel is further assumed to be *L1-resident*, that is, to work only on data that resides in the L1 cache. This assumption is reasonable in two ways. First, if data must be fetched from farther caches, or even the main memory, these fetch operations will be multiple orders of magnitude slower than the computation being analyzed, making it useless to optimize this kernel for CPU efficiency — the expert should, in this case, focus instead on data locality, prefetching, etc. Second, code analyzers typically focus only on the CPU itself, and ignore memory effects. This hypothesis formalizes this focus; code analyzers metrics are thus to be regarded *assuming the CPU is the bottleneck*.

Code analyzers also disregard control flow, and thus assume the code to be *straight-line code*: the kernel analyzed is considered as a sequence of instructions without influence on the control flow, executed in order, and jumping unconditionally back to the first instruction after the last — or, more accurately, the last jump is always assumed taken, and any control flow instruction in the middle is assumed not taken, while their computational cost is accounted for.

Metrics produced. The insights they provide as an output vary with the code analyzer used. All of them are able to predict either the throughput or reciprocal throughput — defined below — of the kernel studied, that is, how many cycles one iteration of the loop takes, in average and in steady-state. Although throughput can already be measured at runtime with hardware counters, a static estimation — if reliable — is already an improvement, as a static analyzer is typically faster than running the actual program under profiling.

Each code analyzer relies on a model, or a collection of models, of the hardware on which it provides analyzes. Depending on what is, or is not modelled by a specific code analyzer, it may further extract any available and relevant metric from its model: whether the frontend is saturated, which computation units from the backend are stressed and by which precise instructions, when the CPU stalls and why, etc. Code analyzers may further point towards the resources that are limiting the kernel's performance, or *bottlenecks*.

Static vs. dynamic analyzers. Tools analyzing code, and code analyzers among them, are generally either performing *static* or *dynamic* analyses. Static analyzers work on the program itself, be it source code, assembly or any representation, without running it; while dynamic

analyzers run the analyzed program, keeping it under scrutiny through either instrumentation, monitoring or any relevant technique. Some analyzers mix both strategies to further refine their analyses. As a general rule of thumb, dynamic analyzers are typically more accurate, being able to study the actual execution trace (or traces) of the program, but are significantly slower due to instrumentation's large overhead and focus more on the general, average case than on edge cases.

As most code analyzers are static, this manuscript largely focuses on static analysis. The only dynamic code analyzer we are aware of is `Gus`, described more thoroughly in section 1.3 later, trading heavily run time to gain in accuracy, especially regarding data dependencies that may not be easily obtained otherwise.

Input formats used. The analyzers studied in this manuscript all take as input either assembly code, or assembled binaries.

In the case of assembly code, as for instance with `llvm-mca`, analyzers take either a short assembly snippet, treated as straight-line code and analyzed as such; or longer pieces of assembly, part or parts of which being marked for analysis by surrounding assembly comments.

In the case of assembled binaries, as all analyzers were run on Linux, executables or object files are ELF files. Some analyzers work on sections of the file defined by user-provided offsets in the binary, while others require the presence of *`IACA` markers* around the code portion or portions to be analyzed. Those markers, introduced by `IACA` as C-level preprocessor statements, consist in the following x86 assembly snippets:

```
1 mov ebx, 111
2 db 0x64, 0x67, 0x90
```

`IACA` start marker

```
1 mov ebx, 222
2 db 0x64, 0x67, 0x90
```

`IACA` end marker

On UNIX-based operating systems, the standard format for assembled binaries — either object files (`.o`) or executables — is ELF [Com+95]. Such files are organized in sections, the assembled instructions themselves being found in the `.text` section — the rest holding metadata, program data (strings, icons, …), debugging information, etc. When an ELF is loaded to memory for execution, each segment may be *mapped* to a portion of the address space. For instance, if the `.text` section has 1024 bytes, starting at offset 4096 of the ELF file itself, it may be mapped at virtual address `0x454000`; as such, the byte that could be read from the program by dereferencing address `0x454010` would be the 16th byte from the `.text` section, that is, the byte at offset 4112 in the ELF file.

Throughout the ELF file, *symbols* are defined as references, or pointers, to specific offsets or chunks in the file. This mechanism is used, among others, to refer to the program's function. For instance, a symbol `main` may be defined, that would point to the offset of the first byte of the `main` function, and may also hold its total number of bytes.

Both these mechanisms can be used to identify, without `IACA` markers or the like, a section of ELF file to be analyzed: an offset and size in the `.text` section can be provided (which can be found with tools like `objdump`), or a symbol name can be provided, if an entire function is to be analyzed.

1.2.2 Examples with `llvm-mca`

We have now covered enough of the theoretical background to introduce code analyzers in a concrete way, through examples of their usage. For this purpose, we use `llvm-mca`, one of the state-of-the-art code analyzers.

Due to its relative simplicity — at least compared to *eg.* Intel's x86-64 implementations —, we will base the following examples on ARM's Cortex A72, which we introduce in depth later in chapter 3. No specific knowledge of this microarchitecture is required to understand the following examples; for our purposes, if suffices to say that:

— the A72 has a single load port, a single store port and two integer arithmetics ports;
— the xN registers are 64-bits registers;
— the ldr instruction (**load register**) loads a value from memory into a register;
— the str instruction (**store register**) stores the value of a register to memory;
— the add instruction adds integer values from its two last operands and stores the result in the first.

Simple example: a single load. We first start by running `llvm-mca` on a single load operation: `ldr x1, [x2]`.

```
1 $ echo 'ldr x1,[x2]' | llvm-mca --march=aarch64 --mcpu=cortex-a72 -
2 Iterations:        100
3 Instructions:      100
4 Total Cycles:      106
5 Total uOps:        100
6
7 Dispatch Width:    3
8 uOps Per Cycle:    0.94
9 IPC:               0.94
10 Block RThroughput: 1.0
11
12
13 Instruction Info:
14 [1]: #uOps
15 [2]: Latency
16 [3]: RThroughput
17 [4]: MayLoad
18 [5]: MayStore
19 [6]: HasSideEffects (U)
20
21 [1]    [2]    [3]    [4]    [5]    [6]    Instructions:
22  1      4     1.00    *                   ldr   x1, [x2]
23
24
25 Resources:
26 [0]   - A57UnitB
27 [1.0] - A57UnitI
28 [1.1] - A57UnitI
29 [2]   - A57UnitL
30 [3]   - A57UnitM
31 [4]   - A57UnitS
32 [5]   - A57UnitW
33 [6]   - A57UnitX
34
35
36 Resource pressure per iteration:
37 [0]    [1.0]  [1.1]  [2]    [3]    [4]    [5]    [6]
38  -      -      -     1.00    -      -      -      -
39
40 Resource pressure by instruction:
41 [0]    [1.0]  [1.1]  [2]    [3]    [4]    [5]    [6]    Instructions:
42  -      -      -     1.00    -      -      -      -     ldr     x1, [x2]
```

The first rows (2-10) are high-level metrics. `llvm-mca` works by simulating the execution of the kernel — here, 100 times, as seen row 2 —. This simple kernel contains only one instruction, which breaks down into a single μOP. Iterating it takes 106 cycles instead of the expected 100 cycles, as this execution is *not* in steady-state, but accounts for the cycles from the decoding of the first instruction to the retirement of the last.

Row 7 indicates that each cycle, the frontend can issue at most 3 μOPs. The next two rows are simple ratios. Row 10 is the block's *reverse throughput*, which we will note $\overline{\mathcal{K}}$ and formalize

later in section 1.2.3, but is roughly defined as the number of cycles a single iteration of the kernel takes.

The next section, *instruction info*, lists data about the instructions present.

Finally, the last section, *resources*, breaks down individual instructions into load incurred on execution ports, first aggregating it by full iteration of the kernel, then instruction by instruction. The maximal load of each port is normalized to 1, which amounts to say that it is expressed in number of cycles required to process the load.

Here, the only pressure is 1 on the port labeled `[2]`, that is, the load port. Thus, the kernel cannot complete in less than a full cycle, as it takes up all load resources available.

The timeline mode. Another useful view that can be displayed by `llvm-mca` is its timeline mode, enabled by passing an extra `--timeline` flag. In the previous example, it further outputs:

```
Timeline view:
                    012345
Index     0123456789

[0,0]     DeeeeER   .    .   ldr         x1, [x2]
[1,0]     D=eeeeER  .    .   ldr         x1, [x2]
[2,0]     D==eeeeER .    .   ldr         x1, [x2]
[3,0]     .D==eeeeER.    .   ldr         x1, [x2]
[4,0]     .D===eeeeER    .   ldr         x1, [x2]
[5,0]     .D====eeeeER   .   ldr         x1, [x2]
[6,0]     . D====eeeeER  .   ldr         x1, [x2]
[7,0]     . D=====eeeeER .   ldr         x1, [x2]
[8,0]     . D======eeeeER.   ldr         x1, [x2]
[9,0]     .  D======eeeeER   ldr         x1, [x2]
```

which indicates, for each instruction, the timeline of its execution. Here, `D` stands for decode, `e` for being executed — in the pipeline —, `E` for last cycle of its execution — leaving the pipeline —, `R` for retiring. When an instruction is decoded and waiting to be dispatched to execution, an `=` is shown.

The identifier at the beginning of each row indicates the kernel iteration number, and the instruction within.

Here, we can better understand the 106 cycles seen earlier: it takes a first cycle to decode the first instruction, the instruction remains in the pipeline for 5 cycles, and must finally be retired. In steady-state, however, the instruction would be already decoded (while a previous instruction was being executed), the retirement would also be taking place while another instruction executes, and the pipeline would be accepting new instructions for four of these five cycles. We can thus avoid using up 6 of those 106 cycles in steady-state, taking us back to the expected 100 cycles.

Single integer add. If we substitute this load operation with an integer add operation, we find a reverse throughput halved:

```
$ echo 'add x1,x2,x3' | llvm-mca --march=aarch64 --mcpu=cortex-a72 -
Iterations:        100
Instructions:      100
Total Cycles:      53
Total uOps:        100

Dispatch Width:    3
uOps Per Cycle:    1.89
IPC:               1.89
Block RThroughput: 0.5

[...]

```

```
[1.0] - A57UnitI
[1.1] - A57UnitI

[...]

Resource pressure by instruction:
[0]    [1.0]  [1.1]  [2]    [3]    [4]    [5]    [6]    Instructions:
 -     0.50   0.50    -      -      -      -      -     add     x1, x2, x3

Timeline view:
Index     01234567

[0,0]     DeER . .   add        x1, x2, x3
[1,0]     DeER . .   add        x1, x2, x3
[2,0]     D=eER. .   add        x1, x2, x3
[3,0]     .DeER. .   add        x1, x2, x3
[4,0]     .D=eER .   add        x1, x2, x3
[5,0]     .D=eER .   add        x1, x2, x3
[6,0]     . D=eER.   add        x1, x2, x3
[7,0]     . D=eER.   add        x1, x2, x3
[8,0]     . D==eER   add        x1, x2, x3
[9,0]     .  D=eER   add        x1, x2, x3
```

Indeed, as we have two integer arithmetics unit, two adds may be executed in parallel, as can be seen in the timeline view.

Load and two adds. If we combine those two instructions in a kernel with a single load and two adds, we obtain a kernel that still fits in the execution ports in a single cycle. `llvm-mca` confirms this:

```
$ echo -e 'ldr x1,[x2]
add x3,x4,x5
add x6,x7,x8' | llvm-mca --march=aarch64 --mcpu=cortex-a72 -

Iterations:        100
Instructions:      300
Total Cycles:      106
Total uOps:        300

Dispatch Width:    3
uOps Per Cycle:    2.83
IPC:               2.83
Block RThroughput: 1.0

[...]

Resource pressure per iteration:
[0]    [1.0]  [1.1]  [2]    [3]    [4]    [5]    [6]
 -     1.00   1.00   1.00    -      -      -      -

Resource pressure by instruction:
[0]    [1.0]  [1.1]  [2]    [3]    [4]    [5]    [6]    Instructions:
 -      -      -     1.00    -      -      -      -     ldr     x1, [x2]
 -      -     1.00    -      -      -      -      -     add     x3, x4, x5
 -     1.00    -      -      -      -      -      -     add     x6, x7, x8
```

We can indeed see that an iteration fully utilizes the three ports, but still fits: the kernel still manages to have a reverse throughput of 1.

Three adds. A kernel of three adds, however, will not be able to run in a single cycle:

```
$ echo -e 'add x1,x2,x3
add x4,x5,x6
add x7,x8,x9' | llvm-mca --march=aarch64 --mcpu=cortex-a72 --timeline -

Iterations:        100
Instructions:      300
Total Cycles:      153
Total uOps:        300

Dispatch Width:    3
uOps Per Cycle:    1.96
IPC:               1.96
Block RThroughput: 1.5

[...]

Resource pressure per iteration:
[0]    [1.0]  [1.1]  [2]    [3]    [4]    [5]    [6]
 -     1.50   1.50    -      -      -      -      -

Resource pressure by instruction:
[0]    [1.0]  [1.1]  [2]    [3]    [4]    [5]    [6]    Instructions:
 -     0.50   0.50    -      -      -      -      -     add     x1, x2, x3
 -     0.50   0.50    -      -      -      -      -     add     x4, x5, x6
 -     0.50   0.50    -      -      -      -      -     add     x7, x8, x9


Timeline view:
                    01234567
Index     0123456789

[0,0]     DeER .    .    . .   add      x1, x2, x3
[0,1]     DeER .    .    . .   add      x4, x5, x6
[0,2]     D=eER.    .    . .   add      x7, x8, x9
[1,0]     .DeER.    .    . .   add      x1, x2, x3
[1,1]     .D=eER    .    . .   add      x4, x5, x6
[1,2]     .D=eER    .    . .   add      x7, x8, x9
[...]
```

The resource pressure by iteration view confirms that we exceed the integer arithmetic capacity of the processor for a single cycle. This is correctly reflected in the timeline view: the instruction `[0,2]` starts executing only at cycle 3, along with `[1,0]`.

Load, store and two adds. A kernel of one load, two adds and one store should, ports-wise, fit in a single cycle. However, `llvm-mca` finds for this kernel a reverse throughput of 1.3:

```
$ echo -e 'ldr x1,[x2]
add x3,x4,x5
add x6,x7,x8
str x9,[x10]' | llvm-mca --march=aarch64 --mcpu=cortex-a72 --timeline -

Iterations:        100
Instructions:      400
Total Cycles:      139
Total uOps:        400

Dispatch Width:    3
uOps Per Cycle:    2.88
IPC:               2.88
Block RThroughput: 1.3

```

```
16 [...]
17
18 Resource pressure per iteration:
19 [0]     [1.0]   [1.1]   [2]     [3]     [4]     [5]     [6]
20  -      1.00    1.00    1.00     -      1.00     -       -
21
22 Resource pressure by instruction:
23 [0]     [1.0]   [1.1]   [2]     [3]     [4]     [5]     [6]     Instructions:
24  -       -       -      1.00     -       -       -       -      ldr     x1, [x2]
25  -       -      1.00     -       -       -       -       -      add     x3, x4, x5
26  -      1.00     -       -       -       -       -       -      add     x6, x7, x8
27  -       -       -       -       -      1.00     -       -      str     x9, [x10]
28
29
30 Timeline view:
31                     012345678
32 Index     0123456789
33
34 [0,0]     DeeeeER   .    .  .   ldr     x1, [x2]
35 [0,1]     DeE---R   .    .  .   add     x3, x4, x5
36 [0,2]     DeE---R   .    .  .   add     x6, x7, x8
37 [0,3]     .DeE--R   .    .  .   str     x9, [x10]
38 [1,0]     .DeeeeER  .    .  .   ldr     x1, [x2]
39 [1,1]     .DeE---R  .    .  .   add     x3, x4, x5
40 [1,2]     . DeE--R  .    .  .   add     x6, x7, x8
41 [1,3]     . DeE--R  .    .  .   str     x9, [x10]
```

While the resource pressure views confirm that the ports are able to handle this kernel in a single cycle, the timeline shows that it is in fact the frontend that stalls the computation. As only three instructions may be decoded and issued per cycle, the backend is not fed with enough instructions per cycle to reach a reverse throughput of 1.

1.2.3 Definitions

Throughput and reciprocal throughput

Given a kernel $\mathcal{K}$ of straight-line assembly code, we have referred to $\overline{\mathcal{K}}$ as the *reciprocal throughput* of $\mathcal{K}$, that is, how many cycles $\mathcal{K}$ will require to complete its execution in steady-state. We define this notion here more formally.

> **Notation** ($\mathcal{K}^n$)
>
> Given a kernel $\mathcal{K}$ and a positive integer $n \in \mathbb{N}^*$, we note $\mathcal{K}^n$ the kernel $\mathcal{K}$ repeated n times, that is, the instructions of $\mathcal{K}$ concatenated n times.

> **Definition** ($C(\mathcal{K})$)
>
> The *number of cycles* of a kernel $\mathcal{K}$ is defined, *in steady-state*, as the number of elapsed cycles from the moment the first instruction of $\mathcal{K}$ starts to be decoded to the moment the last instruction of $\mathcal{K}$ is issued.
>
> We note $C(\mathcal{K})$ the number of cycles of $\mathcal{K}$.
>
> We extend this definition so that $C(\emptyset) = 0$; however, care must be taken that, as we work in steady-state, this $\emptyset$ must be *in the context of a given kernel* (*ie.* we run $\mathcal{K}$ until steady-state is reached, then consider how many cycles it takes to execute 0 further instructions). This context is clarified by noting $C\left(\mathcal{K}^0\right)$.

Due to the pipelined nature of execution units, this means that the same instruction of each iteration of $\mathcal{K}$ will be retired — *ie.* yield its result — every steady-state execution time. For this reason, the execution time is measured until the last instruction is issued, not retired.

Lemma (*Periodicity of* $C\left(\mathcal{K}^{n+1}\right)-C\left(\mathcal{K}^{n}\right)$)

Given a kernel $\mathcal{K}$, the sequence $\left(C\left(\mathcal{K}^{n+1}\right)-C\left(\mathcal{K}^{n}\right)\right)_{n\in\mathbb{N}}$ is periodic, that is, there exists $p \in \mathbb{N}^*$ such that

$$\forall n \in \mathbb{N}, C\left(\mathcal{K}^{n+1}\right)-C\left(\mathcal{K}^{n}\right)=C\left(\mathcal{K}^{n+p+1}\right)-C\left(\mathcal{K}^{n+p}\right)$$

We note this period $\mathcal{P}(\mathcal{K})$.

Proof. The number of CPU resources that can be shared between instructions in a processor is finite (and relatively small, usually on the order of magnitude of 10). These resources are typically the number of μOPs issued for each port in the current cycle, the number of decoded instructions, total number of issued μOPs this cycle and such.

For each of these resources, their number of possible states is also finite (and also small). Thus, the total number of possible states of a processor at the end of a kernel iteration cannot be higher than the combination of those states.

For a given kernel $\mathcal{K}$, We note $\sigma(\mathcal{K})$ the CPU state reached after executing $\mathcal{K}$, in steady-state.

Given a kernel $\mathcal{K}$, the set $\{\sigma(\mathcal{K}^n), n \in \mathbb{N}\}$ is a subset of the total set of possible states of the processor, and is thus finite — and, in all realistic cases, is usually way smaller than the full set, given that only a portion of those resources are used by a kernel.

We further note that, for all $n \in \mathbb{N}, \sigma(\mathcal{K}^{n+1})$ is function of only the processor considered, $\mathcal{K}$ and $\sigma(\mathcal{K}^n)$: indeed, a steady-state for $\mathcal{K}^n$ is also a steady-state for $\mathcal{K}^{n+1}$ and, knowing $\sigma(\mathcal{K}^n)$, the execution can be continued for the following $\mathcal{K}$, reaching $\sigma(\mathcal{K}^{n+1})$.

Thus, by the pigeon-hole principle, there exists $p \in \mathbb{N}$ such that $\sigma(\mathcal{K}) = \sigma(\mathcal{K}^{p+1})$. By induction, as each state depends only on the previous one, we thus obtain that $(\sigma(\mathcal{K}^n))_n$ is periodic of period p. As we consider only the execution's steady state, the sequence is periodic from rank 0.

As the number of cycles needed to execute $\mathcal{K}$ only depend on the initial state of the processor, we thus have

$$\forall n \in \mathbb{N}, C\left(\mathcal{K}^{n+1}\right)-C\left(\mathcal{K}^{n}\right)=C\left(\mathcal{K}^{n+p+1}\right)-C\left(\mathcal{K}^{n+p}\right)$$

□

Definition (*Reciprocal throughput of a kernel*)

The *reciprocal throughput* of a kernel $\mathcal{K}$, noted $\overline{\mathcal{K}}$ and measured in *cycles per iteration*, is also called the steady-state execution time of a kernel.

We note $p = \mathcal{P}(\mathcal{K}) \in \mathbb{N}^*$ the period of $C\left(\mathcal{K}^{n+1}\right)-C\left(\mathcal{K}^{n}\right)$ (by the above lemma), and define

$$\overline{\mathcal{K}} = \frac{C\left(\mathcal{K}^{p}\right)}{p}$$

We define this as the average on a whole period because subsequent kernel iterations may “share” a cycle.

Example

Let $\mathcal{K}$ be a kernel of three instructions, and assume that a given processor can only issue two instructions per cycle, but has no other bottleneck for $\mathcal{K}$. Then, $C(\mathcal{K}) = 2$, as three instructions cannot be issued in a single cycle; yet $C\left(\mathcal{K}^2\right) = 3$, as six instructions can be issued in only three cycles. In this case, the period p is clearly 2. Thus, in this case, $\overline{\mathcal{K}} = 1.5$.

Remark

As $C(\mathcal{K})$ depends on the microarchitecture of the processor considered, the throughput $\overline{\mathcal{K}}$ of a kernel $\mathcal{K}$ implicitly depends on the processor considered.

Lemma

Let $\mathcal{K}$ be a kernel and $p = \mathcal{P}(\mathcal{K})$. For all $n \in \mathbb{N}$ such that $n = kp + r$, with $k, r \in \mathbb{N}$, $1 \leq r \leq p$,

$$C\left(\mathcal{K}^n\right) = kC\left(\mathcal{K}^p\right) + C\left(\mathcal{K}^r\right)$$

Proof. From the previous lemma instantiated with $n = 0$, we have

$$\begin{aligned} & C\left(\mathcal{K}^1\right) - C\left(\mathcal{K}^0\right) = C\left(\mathcal{K}^{p+1}\right) - C\left(\mathcal{K}^p\right) \\ \iff & C\left(\mathcal{K}^p\right) = C\left(\mathcal{K}^{p+1}\right) - C\left(\mathcal{K}^1\right) \end{aligned}$$

and thus by induction, $\forall m \in \mathbb{N}, C\left(\mathcal{K}^{m+p}\right) - C\left(\mathcal{K}^m\right) = C\left(\mathcal{K}^p\right)$.

Thus, if $k = 0$, the property is trivial. If $k = 1$, it is a direct application of the above:

$$C\left(\mathcal{K}^{p+r}\right) = C\left(\mathcal{K}^p\right) + C\left(\mathcal{K}^r\right)$$

We prove by induction the cases for $k > 1$. □

Lemma

Given a kernel $\mathcal{K}$,

$$\frac{C(\mathcal{K}^n)}{n} \xrightarrow[n \to \infty]{} \overline{\mathcal{K}}$$

Furthermore, this convergence is linear:

$$\left| \frac{C\left(\mathcal{K}^n\right)}{n} - \overline{\mathcal{K}} \right| = \mathcal{O}\left(\frac{1}{n}\right)$$

Proof. Let $n \in \mathbb{N}^*$ and $p = \mathcal{P}(\mathcal{K}) \in \mathbb{N}^*$ the periodicity by the above lemma.

Let $k, r \in \mathbb{N}^*$ such that $n = kp + r$, $1 \leq r \leq p$.

$$
\begin{aligned}
C(\mathcal{K}^n) &= k \cdot C(\mathcal{K}^p) + C(\mathcal{K}^r) && \textit{(by lemma)} \\
&= kp\frac{C(\mathcal{K}^p)}{p} + C(\mathcal{K}^r) \\
&= kp\overline{\mathcal{K}} + C(\mathcal{K}^r) \\
\implies \left|C(\mathcal{K}^n) - n\overline{\mathcal{K}}\right| &= \left|kp\overline{\mathcal{K}} + C(\mathcal{K}^r) - (kp+r)\overline{\mathcal{K}}\right| \\
&= \left|C(\mathcal{K}^r) - r\overline{\mathcal{K}}\right| \\
&\leq C(\mathcal{K}^r) + r\overline{\mathcal{K}} && \textit{(all is positive)} \\
&\leq \left(\max_{m \leq p} C(\mathcal{K}^m)\right) + p\overline{\mathcal{K}}
\end{aligned}
$$

This last right-hand expression is independent of n, which we note M. Dividing by n, we obtain

$$
\left|\frac{C(\mathcal{K}^n)}{n} - \overline{\mathcal{K}}\right| \leq \frac{M}{n}
$$

from which both results follow. □

Throughout this manuscript, we mostly use reciprocal throughput as a metric, as we find it more relevant from an optimisation point of view — an opinion we detail in chapter 4. However, the *throughput* of a kernel is most widely used in the literature in its stead.

Definition (*Throughput of a kernel*)

The *throughput* of a kernel $\mathcal{K}$, measured in *instructions per cycle*, or IPC, is defined as the number of instructions in $\mathcal{K}$, divided by the steady-state execution time of $\mathcal{K}$:

$$
\text{IPC}(\mathcal{K}) = \frac{|\mathcal{K}|}{\overline{\mathcal{K}}}
$$

In the literature or in analyzers' reports, the throughput of a kernel is often referred to as its *IPC* (its unit).

Notation (*Experimental measure of* $\overline{\mathcal{K}}$)

We note $\overline{\mathcal{K}}^{M(n)}$ the experimental measure of $\mathcal{K}$, realized by:

- sampling the hardware counter of total number of instructions retired and the counter of total number of cycles elapsed,
- executing $\mathcal{K}^n$,
- sampling again the same counters, and noting respectively $\Delta_n\text{ret}$ and $\Delta_n C$ their differences,
- noting $\overline{\mathcal{K}}^{M(n)} = \dfrac{\Delta_n C \cdot |\mathcal{K}|}{\Delta_n\text{ret}}$, where $|\mathcal{K}|$ is the number of instructions in $\mathcal{K}$.

Lemma

For any kernel $\mathcal{K}$, $\overline{\mathcal{K}}^{M(n)} \xrightarrow[n\to\infty]{} \overline{\mathcal{K}}$.

Proof. For an integer number of kernel iterations n, ${}^{\Delta_n \text{ret}}/_{|\mathcal{K}|} = n$. While measurement errors may make $\Delta_n$ret fluctuate slightly, this fluctuation will be below a constant threshold.

$$\left| \frac{\Delta_n \text{ret}}{|\mathcal{K}|} - n \right| \leq E_{\text{ret}}$$

The same way, and due to the pipelining effects we noted below the definition of $\overline{\mathcal{K}}$,

$$|\Delta_n C - C(\mathcal{K}^n)| \leq E_C$$

with E_C a constant.

As those errors are constant, while other quantities are linear, we thus have

$$\overline{\mathcal{K}}^{M(n)} = \frac{\Delta_n C}{{}^{\Delta_n ret}/_{|\mathcal{K}|}} \xrightarrow[n \to \infty]{} \frac{C(\mathcal{K}^n)}{n}$$

and, composing limits with the previous lemma, we thus obtain

$$\overline{\mathcal{K}}^{M(n)} \xrightarrow[n \to \infty]{} \overline{\mathcal{K}}$$

□

Given this property, we will use $\overline{\mathcal{K}}$ to refer to $\overline{\mathcal{K}}^{M(n)}$ for large values of n in this manuscript whenever it is clear that this value is a measure.

Basic block of an assembly-level program

Code analyzers are meant to analyze sections of straight-line code, that is, portions of code which do not contain control flow. As such, it is convenient to split the program into *basic blocks*, that is, portions of straight-line code linked to other basic blocks to reflect control flow. We define this notion here formally, to use it soundly in the following chapters of this manuscript.

Notation

For the purposes of this section,

— we formalize a segment of assembly code as a sequence of instructions;

— we confuse an instruction with its address.

An instruction is said to be a *flow-altering instruction* if this instruction may alter the normal control flow of the program. This is typically true of jumps (conditional or unconditional), function calls, function returns, ...

An address is said to be a *jump site* if any flow-altering instruction in the considered sequence may alter control to this address (and this address is not the natural flow of the program, *eg.* in the case of a conditional jump).

Definition (*Basic block decomposition*)

Consider a sequence of assembly code A. We note the J_A the set of jump sites of A, F_A the set of flow-altering instructions of A. As each element of those sets is the address of an instruction, we note F_A^+ the set of addresses of instructions *directly following* an instruction from F_A — note that, as instructions may be longer than one byte, it is not sufficient to increase by 1 each address from F_A.

We note $S_A = J_A \cup F_A^+$. We split the instructions from A into $BB(A)$, the set of segments that begin either at the beginning of A, or at instructions from S_A — less formally, we split A at each point from S_A, including each boundary in the following segment.

The members of $BB(A)$ are the *basic blocks* of A, and are segments of code which, by construction, will always be executed as straight-line code, and whose execution will always begin from their first instruction.

Remark

This definition gives a direct algorithm to split a segment of assembly code into basic blocks, as long as we have access to a semantics of the considered assembly that indicates whether an instruction is flow-altering, and if so, what are its possible jump sites.

1.3 State of the art

Performance models for CPUs have been previously studied, and applied to static code performance analysis.

1.3.1 Manufacturer-sourced data

Manufacturers of CPUs are expected to offer optimisation data for software compiled for their processors. This data may be used by compilers authors, within highly-optimized libraries or in the optimisation process of critical sections of programs that require very high performance.

Intel provides its *Intel® 64 and IA-32 Architectures Optimization Reference Manual* [23b], regularly updated, whose nearly 1,000 pages give relevant details to Intel's microarchitectures, such as block diagrams, pipelines, ports available, etc. It further gives data tables with throughput and latencies for some instructions. While the manual provides a huge collection of important insights — from the optimisation perspective — on their microarchitectures, it lacks exhaustive and (conveniently) machine-parsable data tables and does not detail port usages of each instruction.

ARM typically releases optimisation manuals that are way more complete for its microarchitectures, such as the Cortex A72 optimisation manual [15].

AMD, since 2020, releases lengthy and complete optimisation manuals for its microarchitecture. For instance, the Zen4 optimisation manual [23d] contains both detailed insights on the processor's workflow and ports, and a spreadsheet of about 3,400 x86 instructions — with operands variants broken down — and their port usage, throughput and latencies. Such an effort, which certainly translates to a non-negligible financial cost to the company, showcases the importance and recent expectations on such documents.

As a part of its EXEgesis project [Goo], Google made an effort to parse Intel's microarchitecture manuals, resulting in a machine-usable data source of instruction details. The extracted data has since then been contributed to the llvm compiler's data model. The project, however, is no longer developed.

1.3.2 Third-party instruction data

The lack, for many microarchitectures, of reliable, exhaustive and machine-usable data for individual instructions has driven academics to independently obtain this data from an experimental approach.

Since 1996, Agner Fog has been maintaining tables of values useful for optimisation purposes for x86 instructions [Fog20]. These tables, still maintained and updated today, are often considered very accurate. They are the result of benchmarking scripts developed by the author, subject to manual — and thus tedious, given the size of microarchitectures — analysis, and are mainly conducted through hardware counters measurements. The main issue, however, is that those tables are generated through the use of hand-picked instructions and benchmarks, depending on specific hardware counters and features specific to some CPU manufacturers. As such, while these tables are very helpful on the supported CPUs for x86, the method does not scale to the abundance of CPUs on which such tables may be useful — for instance, ARM processors, embedded platforms, etc.

Following the work of Agner Fog, Andreas Abel and Jan Reineke have designed the `uops.info` framework [AR19], striving to automate the previous methodology. Their work, providing data tables for the vast majority of instructions on many recent Intel microarchitectures, has been recently enhanced to also support AMD architectures.

The `uops.info` approach, detailed in their article, consists in finding so-called *blocking instructions* for each port which, used in combination of the instruction to be benchmarked and port-specific hardware counters, yield a detailed analysis of the port usage of each instruction — and even its break-down into μOPs. This makes for an accurate and robust approach, but also limits it to microarchitectures offering such counters, and requires a manual analysis of each microarchitecture to be supported in order to find a fitting set of blocking instructions. Although we have no theoretical guarantee of the existence of such instructions, this should never be a problem, as all pragmatic microarchitecture design will lead to their existence.

1.3.3 Code analyzers and their models

Going further than data extraction at the individual instruction level, academics and industrials interested in this domain now mostly work on code analyzers, as described in section 1.2 above. Each such tool embeds a model — or collection of models — on which its inference is based, and whose definition, embedded data and obtention method varies from tool to tool. These tools often use, to some extent, the data on individual instructions obtained either from the manufacturer or the third-party efforts mentioned above.

The Intel Architecture Code Analyzer (`IACA`) [Intb], released by Intel, is a fully-closed source analyzer able to analyze assembly code for Intel microarchitectures only. It draws on Intel's own knowledge of their microarchitectures to make accurate predictions. This accuracy made it very helpful to experts aiming to do performance debugging on supported microarchitectures. Yet, being closed-source and relying on data that is partially unavailable to the public, the model is not totally satisfactory to academics or engineers trying to understand specific performance results. It also makes it vulnerable to deprecation, as the community is unable to *fork* the project — and indeed, `IACA` has been discontinued by Intel in 2019. Thus, `IACA` does not support recent microarchitectures, and its binary was recently removed from official download pages.

In the meantime, the LLVM Machine Code Analyzer — or `llvm-mca` — was developed as an internal tool at Sony, and was proposed for inclusion in `llvm` in 2018 [Bia18]. This code analyzer is based on the data tables that `llvm` — a compiler — has to maintain for each microarchitecture in order to produce optimized code. The project has since then evolved to be fairly accurate, as seen in the experiments later presented in this manuscript. It is the alternative Intel offers to `IACA` subsequently to its deprecation.

Another model, `Osaca`, was developed by Jan Laukemann *et al.* starting in 2017 [Lau+18; Lau+19]. Its development stemmed from the lack (at the time) of an open-source — and thus, open-model — alternative to IACA. As a data source, `Osaca` makes use of Agner Fog's data

tables or `uops.info`. It still lacks, however, a good model of frontend and data dependencies, making it less performant than other code analyzers in our experiments later in this manuscript.

Taking another approach entirely, `Ithemal` is a machine-learning-based code analyzer striving to predict the reciprocal throughput of a given kernel. The necessity of its training resulted in the development of `BHive`, a benchmark suite of kernels extracted from real-life programs and libraries, along with a profiler measuring the runtime, in CPU cycles, of a basic block isolated from its context. This approach, in our experiments, was significantly less accurate than those not based on machine learning. In our opinion, its main issue, however, is to be a *black-box model*: given a kernel, it is only able to predict its reverse throughput. Doing so, even with perfect accuracy, does not explain the source of a performance problem: the model is unable to help detecting which resource is the performance bottleneck of a kernel; in other words, it quantifies a potential issue, but does not help in *explaining* it — or debugging it.

In yet another approach, `PMEvo` [RH20] uses genetic algorithms to infer, from scratch and in a benchmarks-oriented approach, a port-mapping of the processor it is running on. It is, to the best of our knowledge, the first tool striving to compute a port-mapping model in a fully-automated way, as `Palmed` does (see chapter 2 later), although through a completely different methodology. As detailed in `Palmed`'s article [Der+22], it however suffers from a lack of scalability: as generating a port-mapping for the few thousands of x86-64 instructions would be extremely time-consuming with this approach, the authors limit the evaluation of their tool to around 300 most common instructions.

Abel and Reineke, the authors of `uops.info`, recently released `uiCA` [AR22], a code analyzer for Intel microarchitectures based on `uops.info` tables on one hand as a port model, and on manual reverse-engineering through the use of hardware counters to model the frontend and pipelines. We found this tool to be very accurate (see experiments later in this manuscript), with results comparable with `llvm-mca`. Its source code — under free software license — is self-contained and reasonably concise (about 2,000 lines of Python for the main part), making it a good basis and baseline for experiments. It is, however, closely tied by design to Intel microarchitectures, or microarchitectures very close to Intel's ones.

Chapter 2

`Palmed`: automatically modelling the backend

The state-of-the-art tools presented in section 1.3 are already capable of good microarchitectural analysis and predictions in many aspects. One thing, however, that we found lacking, was a generic method to obtain a model for a given microarchitecture. Indeed, while *eg.* `IACA` and `uops.info` are performant and quite exhaustive models of Intel's x86-64 implementations, they are restricted to Intel CPUs — and few others for `uops.info`. These models were, at least up to a point, handcrafted. While `IACA` is based on insider's knowledge from Intel (and thus would not work for *eg.* AMD), `uops.info`'s method is based on specific hardware counters and handpicked instructions with specific properties.

While these methods provide great models, they only apply to the covered CPUs. In the meantime, many new CPUs are released, some for commercial applications, some others for specific domains, fostering less attention from microarchitecture specialists. For those CPUs not covered by these specific models, a generic method to extract a model from running benchmarks on a processor could be beneficial to performance debuggers.

While `PMEvo` [RH20] laid a first milestone in this direction (see section 1.3), their methodology struggles to scale to the full instruction set of a CPU, as we show in this chapter. To this end, Nicolas Derumigny designed `Palmed` during his PhD work. Although the theoretical work at the core of `Palmed` is his, I contributed significantly to this project during the first period of my own PhD.

In this chapter, sections 2.1 through 2.3 describe `Palmed`, and present what is mostly not my own work, but introduce important concepts for this manuscript. Sections 2.4 and later describe my own work on this project.

2.1 Resource models

2.1.1 Usual representation: tripartite disjunctive graph

As we saw earlier in section 1.1, the behaviour of a CPU's backend can be, throughput-wise, characterized by the behaviour of its ports. Thus, a throughput model of the backend consists in a mapping of the ISA's instructions to execution ports of the backend, called a *port mapping*.

The mapping, however, is not direct: we also saw in section 1.1 that instructions are themselves broken down into a number of micro-operations (μOPs), which all have to be executed. Each of those μOPs are then scheduled on one of the compatible execution ports of the CPU. A port mapping, thus, is actually a tripartite graph: a first layer mapping instructions to μOPs, followed by a second layer mapping μOPs to ports. In Figure 2.1, we show such a port mapping for a few x86-64 instructions on the SKL-SP microarchitecture. The `uops.info`

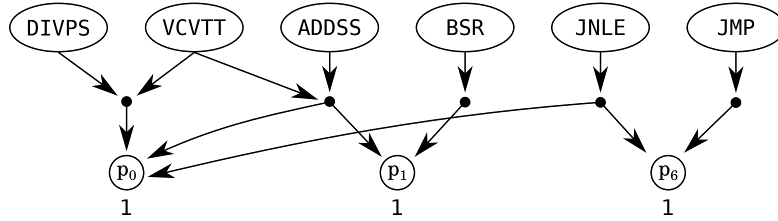


Figure 2.1 – Port mapping and maximum port throughput for a few SKL-SP instructions.

framework [AR19], for instance, produces such a model: each instruction's mapping is described as a string, *eg.* `VCVTT` [1] is described as `1*p0+1*p01`.

The two layers of such a model play a very different role. Indeed, the top layer (instructions to μOPs) can be seen as an *and*, or *conjunctive* layer: an instruction is decomposed into each of its μOPs, which must all be executed for the instruction to be completed. The bottom layer (μOPs to ports), however, can be seen as an *or*, or *disjunctive* layer: a μOP must be executed on *one* of those ports, each able to execute this μOP. This can be seen in the example from `uops.info` above: `VCVTT` is decomposed into two μOPs, the first necessarily executed on port 0, the second on port either 0 or 1.

We also saw that on modern CPUs, ports and computation units are most of the time fully-pipelined; that is, each port can execute a μOP each cycle, even through actually executing a μOP may take multiple cycles. Thus, instruction latencies are not needed to compute the throughput of a kernel without dependencies in steady-state, and a port mapping is sufficient.

As some μOPs are compatible with multiple ports, the number of cycles required to run one occurrence of a kernel is not trivial. An assignment, for a given kernel, of its constitutive μOPs to ports, is a *schedule* — the number of cycles taken by a kernel given a fixed schedule is well-defined. The throughput of a kernel is defined as the throughput under an optimal schedule for this kernel.

Example (*Kernel throughputs with port mappings*)

The kernel $\mathcal{K}_1 = \texttt{DIVPS} + \texttt{BSR} + \texttt{JMP}$ can complete in one cycle: $\overline{\mathcal{K}_1} = 1$. Indeed, according to the port mapping in Figure 2.1, each of those instructions is decoded into a single μOP, each compatible with a single, distinct port. Thus, the three instructions can be issued in parallel in one cycle.

The same goes for $\mathcal{K}_2 = \texttt{ADDSS} + \texttt{BSR}$, although it is a bit less trivial. Both instructions decode to a single μOP. `BSR` can only be executed by port p_1, while `ADDSS` can be executed either by port p_0 or p_1: by picking p_0, both instructions can be executed in a single cycle in steady state, hence $\overline{\mathcal{K}_2} = 1$.

The kernel $\mathcal{K}_3 = \texttt{ADDSS} + 2 \times \texttt{BSR}$, however, needs at least two cycles to be executed: `BSR` can only be executed on port p_1, which can execute at most one μOP per cycle. $\overline{\mathcal{K}_3} = 2$.

The instruction `ADDSS` alone, however, can be executed twice per cycle: once on p_0 and once on p_1. The kernel $\mathcal{K}_4 = 2 \times \texttt{ADDSS} + \texttt{BSR}$ can thus be executed in 1.5 cycles in average: $\overline{\mathcal{K}_4} = 1.5$.

The following tables present an optimal schedule for each kernel $\mathcal{K}_2, \mathcal{K}_3, \mathcal{K}_4$. Each row represents a cycle.

1. The precise variant is `VCVTTSD2SI (R32, XMM)`

$\mathcal{K}_2$	
p_0	p_1
ADDSS	BSR
ADDSS	BSR
⋮	

$\mathcal{K}_3$	
p_0	p_1
ADDSS	BSR
$\emptyset$	BSR
ADDSS	BSR
$\emptyset$	BSR
⋮	

$\mathcal{K}_4$	
p_0	p_1
ADDSS	BSR
ADDSS	BSR
ADDSS	ADDSS
⋮	

Finding the throughput of the kernels presented above is easy enough, as the kernels involve few μOPs compatible with many ports. However, in the general case, finding an optimal schedule becomes more complicated; in fact, it can be expressed as a flow problem — described in Section 3.5.1 of Fabian Gruber's PhD thesis [Gru19].

2.1.2 Dual representation: conjunctive resource mapping

The method behind `Palmed` is based on the observation that a port mapping admits a dual representation, where the bottom layer is not expressed as an "or", but also as an "and".

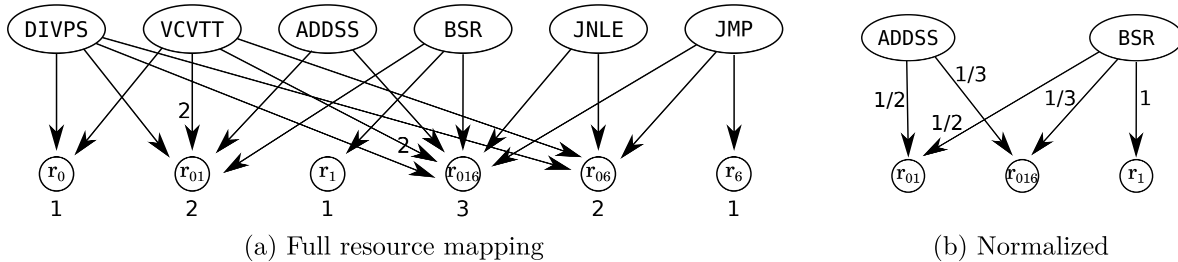
In this dual model, an instruction such as `ADDSS` does not use *either* p_0 or p_1, but instead uses once the combined resource r_{01}, which has a throughput of 2. Instructions such as `BSR`, using only p_1, are using *both* r_1 and r_{01}. In Figure 2.2a, we present the resource mapping equivalent to the port mapping presented in Figure 2.1. As both top and bottom layers are now conjunctive, we remove altogether the intermediate nodes: an instruction directly consumes resources. We then normalize this graph to resources with a unitary throughput by dividing each edge's weight by its corresponding resource throughput. The normalized mapping for `ADDSS` and `BSR` is presented in Figure 2.2b.

The construction of this dual model, and its equivalence to the original, disjunctive model is detailed in the extended version of the full article on `Palmed` [Der+22].

Finding the throughput of a kernel with this conjunctive representation does not require the solving of an optimisation problem. The number of cycles required by a kernel is simply the maximum load over all resources.

More formally, we note $\rho_{i,r}$ the weight of the edge between instruction i and resource r, and $\mathcal{R}$ the set of resources. We consider a kernel $\mathcal{K} = \sum_{i \in \mathcal{K}} \sigma_{i,\mathcal{K}} \times i$. Then, in steady-state, the backend would need $\overline{\mathcal{K}}$ cycles to execute $\mathcal{K}$, and

$$\overline{\mathcal{K}} = \max_{r \in \mathcal{R}} \left(\sum_{i \in \mathcal{K}} \sigma_{i,\mathcal{K}} \times \rho_{i,r} \right) \tag{2.1}$$



(a) Full resource mapping

(b) Normalized

Figure 2.2 – Abstract resource mapping (conjunctive form) and maximum resource throughput for a few SKL-SP instructions.

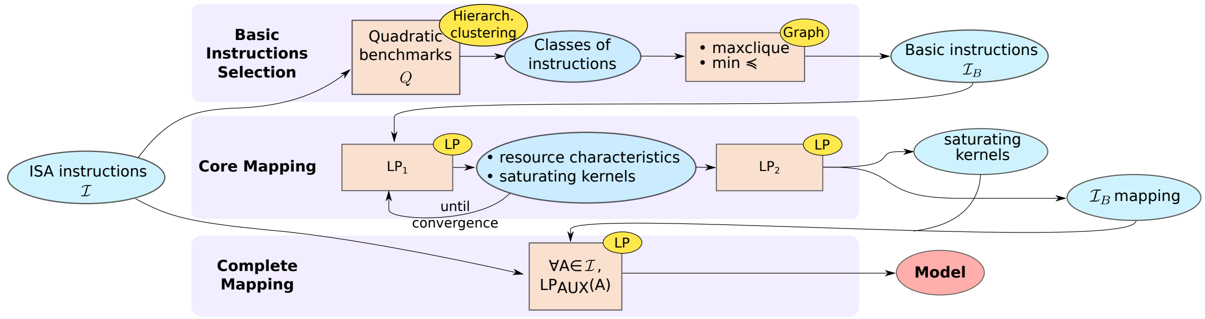


Figure 2.3 – High-level view of Palmed's architecture

Example

The throughputs of the previous kernels can be computed using the conjunctive resource model instead.

$\mathcal{K}_2$

	r_0	r_1	r_{01}
ADDSS			1/2
BSR		1	1/2
Total	0	**1**	**1**

$\implies \overline{\mathcal{K}_2} = 1$

$\mathcal{K}_3$

	r_0	r_1	r_{01}
ADDSS			1/2
2×BSR		2	1
Total	0	**2**	1.5

$\implies \overline{\mathcal{K}_3} = 2$

$\mathcal{K}_4$

	r_0	r_1	r_{01}
2×ADDSS			1
BSR		1	1/2
Total	0	1	**1.5**

$\implies \overline{\mathcal{K}_4} = 1.5$

The drawback of this conjunctive model, however, is that it generates a theoretically combinatorial number of new resources. This, however, does not happen in practice: a combined resource is only necessary if at least one μOP is supported by this set of combined ports. On real processors, ports are not random, but instead have a well-defined set of functions, *eg.* arithmetics, memory access, etc. Thus, only a very limited number of combined resources are necessary.

2.2 Palmed design

Palmed is a tool aiming to construct a resource mapping for a CPU, in a fully automated way, based on the execution of well-chosen benchmarks. As its goal is to construct a resource mapping, its only concern is backend throughput — in particular, dependencies are entirely ignored. In-order effects are not modelled either; in fact, Palmed defines a kernel as a multiset of instructions, discarding instructions ordering at once.

The general idea behind Palmed is that, as we saw above, the execution time of a kernel is described by a resource model through Equation 2.1. We can, however, reverse the problem: if we measure $\overline{\mathcal{K}}$, the only unknown parameters in Equation 2.1 become the $\rho_{i,r}$; that is, the weight of the edges in the resource model for the CPU under scrutiny. Given enough, well-chosen couples $\left(\mathcal{K}, \overline{\mathcal{K}}\right)$, it should then be possible to solve the system for the $\rho_{i,r}$ coefficients, thus building a resource model.

This section does not detail entirely Palmed, but rather coarsely describes the general approach; the full methodology can be found in the full article [Der+22]. Its main steps and components are sketched in Figure 2.3.

Palmed starts off with a list of instructions available in the ISA, $\mathcal{I}$, as well as a description of their legal parameters. This list can be obtained using a decompiler.

The first block, Basic Instructions Selection, benchmarks every couple of instructions — a

step we call *quadratic benchmarks*. These quadratic benchmarks are used to group together instructions into *classes* of instructions that behave identically from the backend's point of view. Formally, the classes are built as equivalence classes for the relation $\sim$:

$$a \sim b \iff \forall i \in \mathcal{I}, \overline{a+i} = \overline{b+i}$$

To accommodate for measurement imprecisions and fluctuations, this strict equality is in practice relaxed; the classes are obtained by hierarchical clustering [War63], splitting the tree into classes by maximizing the silhouette [Rou87]. This clustering into classes is reused later in chapter 3.

The first block then finishes by applying heuristics to select *basic* instructions, that is, instructions that stress as few resources as possible, with the highest possible throughput. These instructions can later be combined with others to detect whether they stress a resource.

The second block, Core Mapping, builds benchmarks against these basic instructions to discover, for each resource r, a kernel — that should be as simple as possible — that saturates it: adding any instruction that uses r to this kernel should increase its execution time. These saturating kernels are discovered with successive Linear Programming (LP) passes, using the Gurobi Optimizer [Gur].

These kernels are then used in a final block, Complete Mapping, to find the $\rho_{i,r}$ coefficients for every instruction, constituting the final model.

2.3 Measuring a kernel's throughput: `Pipedream`

To build a mapping of a CPU, Palmed fundamentally depends on the ability to measure the execution time $\overline{\mathcal{K}}$ of a kernel $\mathcal{K}$. However, as we saw above, Palmed defines a kernel as a multiset of instructions, and makes hypotheses on the measures accordingly. Specifically, this measure should reflect only out-of-order behaviours, without any dependency between instructions. The behaviour should be purely the CPU's; specifically, no memory effect should be accounted for, and thus, data should be L1-resident. Finally, even if hardware counters could be used to provide such a metric, they should be ignored, as Palmed aims to be as universal as possible and should avoid depending on vendor-specific counters.

For this purpose, we use `Pipedream` as a benchmarking backend, a tool initially written by Fabian Gruber and described in his PhD thesis [Gru19], with further developments from Nicolas Derumigny and Christophe Guillon. Originally written for a broader purpose, `Pipedream` is able to generate assembly code to benchmark a multiset of instructions, fitting the constraints mentioned above. The generated assembly uses the following high-level shape:

```
for NUM_MEASURES:
    measure_start()
    for NUM_ITER:
        kernel
        kernel
        ...
        kernel
    measure_stop()
```

The kernel is unrolled enough times so that the body of the innermost loop has at least `UNROLL_SIZE` instructions; and `NUM_ITER` is defined so that `UNROLL_SIZE` $\times$ `NUM_ITER` $\geq$ `TOTAL_INSN`. `UNROLL_SIZE`, `TOTAL_INSN` and `NUM_MEASURES` are parameters of the benchmark generation.

`Pipedream` must be able to distinguish between variants of instructions with the same mnemonic — *eg.* `mov` — but different operand kinds, altering the semantics and performance of the instruction — such as a `mov` loading from memory versus a `mov` between registers. To this end, `Pipedream` represents instructions fully qualified with their operands' kind — this can be seen as a process akin to C++'s name mangling.

As `Pipedream` gets a multiset of instructions as a kernel, these instructions' arguments must be instantiated to turn them into actual assembly code — that is, turn *eg.* `ADD_GPR64i64_IMMi8` (the x86-64 variant of `add` taking as arguments a general purpose regsiter of 64 bits and an immediate of 8 bits) into `addq $0x10, %rdx`. There is no particular issue to instantiate immediates; however, allocating registers and memory operands requires some care, as no dependency must be created between instructions.

Register operands. For each type of register of the ISA (general purpose, vector, …), the registers are split into a read and a write pool. The read pool only contains the maximum number of registers of this type needed by one instruction of the kernel; while the write pool contains the rest. The registers are then allocated for each instruction as follows.

— The read operands are allocated registers from the read pool without specific care, as read-after-read does not create a dependency between two instructions.

— The written operands are allocated registers from the write pool in a round-robin fashion, to maximize the distance between two reuses of the same register. Indeed, on some architectures, write-after-write dependencies do not allow full parallelism; while on some others, some instructions' operands are both read and written, resulting in read-after-write dependencies.

Memory operands. At startup, a memory arena small enough to fit into the L1 cache is allocated. This arena is again split into a read and a write pool.

— In direct register addressing mode — *eg.* `movq %rax, (%rbx)` in x86-64 —, the same address is always reused (one for reads and one for writes).

— In base-index-displacement mode, the same base address is always reused; displacements are used in a round-robin fashion [2].

These two allocation policies are meant to ensure that, whenever possible, no dependency will be created between two instructions: a dependency should only appear when write-after-write dependencies matter, and not enough registers of a kind are available in the architecture — in this case, the same register may be reused too early.

To finally ensure that data is always L1-resident, warm-up rounds are performed before actually measuring the inner loop. As the memory area is small enough, and no other memory access is made during the measure, the memory area is subsequently L1-resident. This also has the effect to warm up the branch predictor.

Finally, the generated assembly is assembled into a shared library object, and invoked with a lightweight C harness. Using the `PAPI` [Muc+99] library, it measures and records two standard and omnipresent hardware events: the number of elapsed cycles (`PAPI_TOT_CYC`) and the number of completed instructions (`PAPI_TOT_INS`).

2.4 Finding basic blocks to evaluate `Palmed`

In the context of all that is described above, my main task in the environment of `Palmed` was to build a system able to evaluate a produced mapping on a given architecture.

Some tools, such as `PMEvo` [RH20], use randomly-sampled basic blocks for their evaluation. However, random generation may yield basic blocks that are not representative of the various workloads our model might be used on. Thus, while arbitrarily or randomly generated microbenchmarks were well suited to the data acquisition phase needed to generate the model, the kernels on which the model would be evaluated could not be arbitrary, and must instead come from real-world programs.

2. On ARM, a displacement of 0 is always used, resulting in the same accesses as direct register addressing.

2.4.1 Benchmark suites

Models generated by `Palmed` are meant to be used on basic blocks that are computationally intensive — so that the backend is actually the relevant resource to monitor, compared to *eg.* frontend- or input/output-bound code —, running in steady-state — that is, which is the body of a loop long enough to be reasonably considered infinite for performance modelling purposes. The basic blocks used to evaluate `Palmed` should thus be reasonably close from these criteria.

For this reason, we evaluate `Palmed` on basic blocks extracted from two well-known benchmark suites: Polybench and SPEC CPU 2017.

Polybench is a suite of benchmarks built out of 30 kernels of numerical computation [PY16]. Its benchmarks are domain-specific and centered around scientific computation, mathematical computation, image processing, etc. As the computation kernels are clearly identifiable in the source code, extracting the relevant basic blocks is easy, and fits well for our purpose. It is written in C language. Although it is not under a free/libre software license, it is free to use and open-source.

SPEC CPU 2017 is a suite of benchmarks meant to be CPU intensive [BLK18]. It is composed of both integer and floating-point based benchmarks, extracted from (mainly open source) real-world software, such as `gcc`, `imagemagick`, … Its main purpose is to obtain metrics and compare CPUs on a unified workload; it is however commonly used throughout the literature to evaluate compilers, optimizers, code analyzers, etc. It is split into four variants: integer and floating-point, combined with speed — time to perform a single task — and rate — throughput for performing a flow of tasks. Most benchmarks exist in both speed and rate mode. The SPEC suite is under a paid license, and cannot be redistributed, which makes peer-review and replication of experiments — *eg.* for artifact review — complicated.

2.4.2 Manually extracting basic blocks

The first approach that we used to extract basic blocks from the two benchmark suites introduced above, for the evaluation included in our article for `Palmed` [Der+22], was very manual. We use different — though similar — approaches for Polybench and SPEC.

In the case of Polybench, we compile multiple versions of each benchmark (`-O2`, `-O3` and tiled using the Pluto optimizer [BRS07]). We then use `QEMU` [QEM] to extract *translation blocks* — very akin to basic blocks — and an occurrence count for each of those. We finally select the basic blocks that have enough occurrences to be body loops.

In the case of SPEC, we replace `QEMU` with the Linux `perf` profiler, as individual benchmarks of the suite are heavier than Polybench benchmarks, making `QEMU`'s instrumentation overhead impractical. While `perf` provides us with occurrence statistics, it does not chunk the program into basic blocks; we use an external disassembler and heuristics on the instructions to do this chunking. We describe both aspects — profiling and chunking — with more details below.

Altogether, this method generates, for x86-64 processors, 13 778 SPEC-based and 2 664 polybench-based basic blocks.

2.4.3 Automating basic block extraction

This manual method, however, has multiple drawbacks. It is, obviously, tedious to manually compile and run a benchmark suite, then extract basic blocks using two collections of scripts depending on which suite is used. It is also impractical that the two benchmark suites are scrapped using very similar, yet different techniques: as the basic blocks are not chunked using the same code, they might have slightly different properties.

Most importantly, this manual extraction is not reproducible. This comes with two problems.

— If the dataset was to be lost, or if another researcher wanted to reproduce our results, the exact same dataset could not be identically recreated. The same general procedure could be followed again, but code and scripts would have to be re-written, manually typed and undocumented shell lines re-written, etc. Most importantly, the re-extracted basic blocks may well be slightly different.
— The same consideration applies to porting the dataset to another ISA. Indeed, as the dataset consists of assembly-level basic-blocks, it cannot be transferred to another ISA: it has to be re-generated from source-level benchmarks. This poses the same problems as the first point.

This second point particularly motivated us to automate the basic block extraction procedure when `Palmed` — and the underlying `Pipedream` — were extended to produce mappings for ARM processors.

Our automated extraction tool, `benchsuite-bb`, is able to extract basic blocks from Polybench and SPEC. Although we do not use it to evaluate `Palmed`, it also supports the extraction of basic blocks from Rodinia [Che+09], a benchmark suite targeted towards heterogeneous computing, and exhibiting various usual kernels, such as K-means, backpropagation, BFS, …

For the most part, `benchsuite-bb` implements the manual approach used for SPEC. On top of an abstraction layer meant to unify the interface to all benchmark suites, it executes the various compiled binaries while profiling them through `perf`, and chunks the relevant parts into basic blocks using a disassembler.

Profiling with `perf`. The `perf` profiler [Lin] is part of the Linux kernel. It works by sampling the current program counter (as well as the stack, if requested, to obtain a stack trace) upon either event occurrences, such as number of elapsed CPU cycles, context switches, cache misses, …, or simply at a fixed, user-defined time frequency.

In our case, we use this second mode to uniformly sample the program counter across a run. We recover the output of the profiling as a *raw trace* with `perf report -D`.

ELF natigation: `pyelftools` and `capstone`. To trace this program counter samplings back to basic blocks, we then need to chunk the relevant sections of the ELF binary down to basic blocks. For this, we use two tools: `pyelftools` and `capstone`.

The `pyelftools` Python library is able to parse and decode many informations contained in an ELF file. In our case, it allows us to find the `.text` section of the input binary, search for symbols, find the symbol containing a given program counter, extract the raw assembled bytes between two addresses, etc.

The `capstone` disassembler, on the other hand, allows to disassemble a portion of assembled binary back to assembly. It supports many ISAs, among which x86-64 and ARM, the two ISAs we investigate in this manuscript. It is able to extract relevant details out of an instruction: which operands, registers, … it uses; which broader group of instruction it belongs to; etc. These groups of instructions, in our case, are particularly useful, as it allows us to find control flow instructions without writing code specific to an ISA. These control-altering instructions are jumps, calls and returns. We are also able to trace a (relative) jump to its jump site, enabling us later to have a finer definition of basic blocks.

Extracting basic blocks. We describe the basic block extraction, given the `perf`-provided list of sampled program counters, in Algorithm 1. For each program counter, we find the ELF symbol it belongs to, and decompose this whole symbol into basic blocks — we memoize this step to do it only once per symbol. We then bissect the basic block corresponding to the current PC from the list of obtained basic blocks to count the occurrences of each block.

```
function bbsOfSymbol(symbol)                        ▷ Memoized, computed only once per symbol
    instructions ← disassemble(bytesFor(symbol))      ▷ Uses both pyelftools and capstone
    flowSites ← ∅
    jumpSites ← ∅
    for instr ∈ instructions do
        if isControlFlow(instr) then
            flowSites ← flowSites ∪ {next(instr).addr}
            if isJump(instr) then
                jumpSites ← jumpSites ∪ {instr.jump_addr}
    return instructions.splitAt(flowSites ∪ jumpSites)

function bbsOfPcs(pcs)
    occurences ← {}
    for pc ∈ pcs do
        bbs ← bbsOfSymbol(symbolOfPc(pc))
        bb ← bissect(pc, bbs)
        occurrences[bb] + +
    return occurrences
```

Algorithm 1 – Basic block extraction procedure, given a `perf`-obtain list of program counters.

To split a symbol into basic blocks, we follow the procedure introduced by our formal definition in section 1.2.3. We determine using `capstone` its set of *flow sites* and *jump sites*. The former is the set of addresses just after a control flow instruction, while the latter is the set of addresses to which jump instructions may jump. We then split the straight-line code of the symbol using the union of both sets as boundaries.

Altogether, on x86-64, we retrieve 2 499 basic blocks on Polybench, and 13 383 basic blocks on SPEC. The structure, dataset and runtime of both benchmark suite, however, makes it significantly harder to gather real “kernels” on SPEC: while 724 of the Polybench basic blocks were sampled more than 1 000 times, only 75 were so sampled in SPEC, and 668 were sampled more than 100 times.

2.5 Evaluating `Palmed`

Evaluating `Palmed` on the previously gathered basic blocks now requires, on one hand, to define evaluation metrics and, on the other hand, an evaluation harness to collect the throughput predictions from `Palmed` and the other considered code analyzers, from which metrics will be derived.

2.5.1 Evaluation harness

We implement into `Palmed` an evaluation harness to evaluate it both against native measurement and other code analyzers.

We first strip each basic block gathered of its dependencies to fall into the use-case of `Palmed` using `Pipedream`, as we did previously. This yields assembly code that can be run and measured natively. The body of the most nested loop can also be used as an assembly basic block for other code analyzers. However, as `Pipedream` does not support some instructions (control flow, x86-64 divisions, …), those are stripped from the original kernel, which might denature the original basic block.

To evaluate `Palmed`, the same kernel’s run time is measured:

1. natively on each CPU, using the `Pipedream` harness to measure its execution time;

2. using the resource mapping `Palmed` produced on the evaluation machine;
3. using the `uops.info` [AR19] port mapping, converted to its equivalent conjunctive resource mapping [3];
4. using `PMEvo` [RH20], ignoring any instruction not supported by its provided mapping;
5. using `IACA` [Intb], by inserting assembly markers around the kernel and running the tool;
6. using `llvm-mca` [SL], by inserting markers in the `Pipedream`-generated assembly code and running the tool.

The raw results are saved (as a Python `pickle` file) for reuse and archival.

2.5.2 Metrics extracted

As `Palmed` internally works with Instructions Per Cycle (IPC) metrics, and as all these tools are also able to provide results in IPC, the most natural metric to evaluate is the error on the predicted IPC. We measure this as a Root-Mean-Square (RMS) error over all basic blocks considered, weighted by each basic block's measured occurrences:

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

This error metric measures the relative deviation of predictions with respect to a baseline. However, depending on how this prediction is used, the relative *ordering* of predictions — that is, which basic block is faster — might be more important. For instance, a compiler might use such models for code selection; here, the goal would not be to predict the performance of the kernel selected, but to accurately pick the fastest.

For this, we also provide Kendall's τ coefficient [Ken38]. This coefficient varies between -1 (full anti-correlation) and 1 (full correlation), and measures how many pairs of basic blocks (i, j) were correctly ordered by a tool, that is, whether

$$\text{IPC}_{i,\text{native}} \leq \text{IPC}_{j,\text{native}} \iff \text{IPC}_{i,\text{tool}} \leq \text{IPC}_{j,\text{tool}}$$

Finally, we also provide a *coverage* metric for each tool; that is, which proportion of basic blocks it was able to process.

The definition of *able to process*, however, varies from tool to tool. For `IACA` and `llvm-mca`, this means that the analyzer crashed or ended without yielding a result. For `uops.info`, this means that one of the instructions of the basic block is absent from the port mapping. `PMEvo`, however, is evaluated in degraded mode when instructions are not mapped, simply ignoring them; it is considered as failed only when *no instruction at all* in the basic block was present in the model.

This notion of coverage is partial towards `Palmed`. As we use `Pipedream` as a baseline measurement, instructions that cannot be benchmarked by `Pipedream` are pruned from the benchmarks. Hence, `Palmed` has a 100 % coverage *by construction* — which does not mean that is supports all the instructions found in the original basic blocks, but only that our methodology is unable to process basic blocks unsupported by Pipedream.

2.5.3 Results

We run the evaluation harness on two different machines:

3. When this evaluation was made, `uiCA` [AR22] was not yet published. Since `Palmed` only provides a resource mapping, but no frontend, the comparison to `uops.info` is fair.

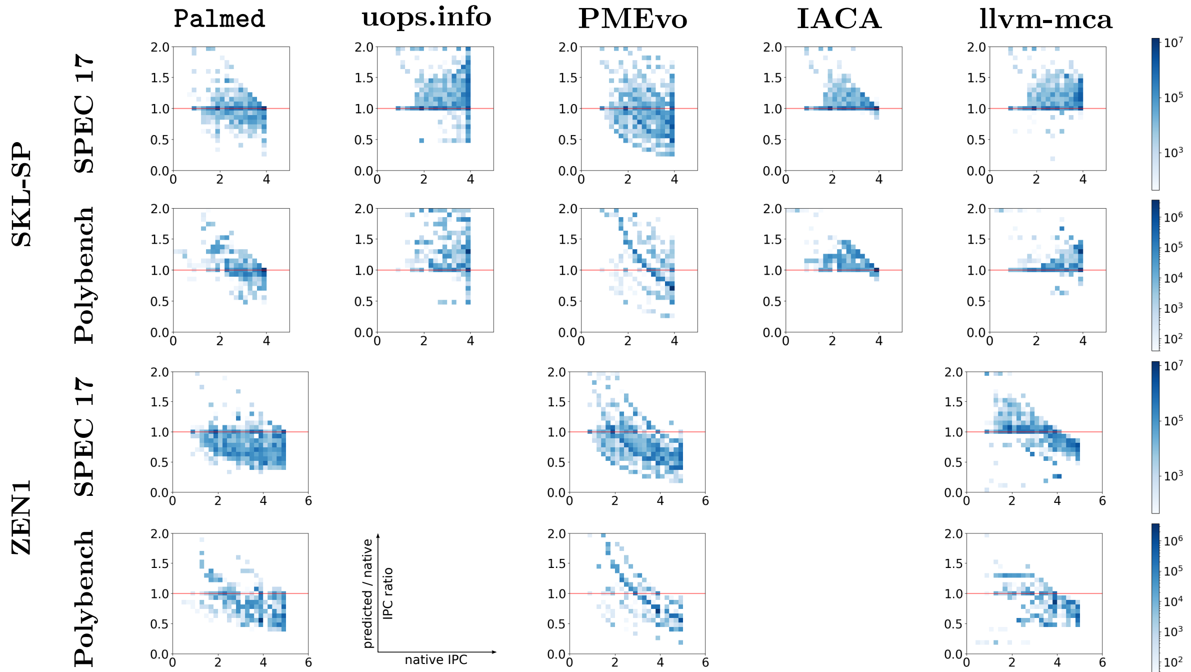


Figure 2.4 – IPC prediction profile heatmaps — predictions closer to the red line are more accurate. Predicted IPC ratio (Y) against native IPC (X)

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

Table 2.1 – Accuracy of IPC predictions compared to `Pipedream`-based native executions on SPEC 2017 and Polybench

— an x86-64 Intel `SKL-SP`-based machine, with two Intel Xeon Silver 4114 CPU, totalling 20 cores;
— an x86-64 AMD `ZEN1`-based machine, with a single AMD EPYC 7401P CPU with 24 cores.

As `IACA` only supports Intel CPUs, and `uops.info` only supports x86-64 machines and gives only very rough information for `ZEN` architectures — without port mapping —, these two tools were only tested on the `SKL-SP` machine.

The evaluation metrics for all three architecture and all five tools are presented in Table 2.1. We further represent IPC prediction accuracy as heatmaps in Figure 2.4. 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 measured IPC of x. The closer a prediction is to the red horizontal line, the more accurate it is.

These results are analyzed in the full article [Der+22].

2.6 Other contributions

Using a database to enhance reproducibility and usability. `Palmed`'s method is driven by a large number of `Pipedream` benchmarks. For instance, generating a mapping for an x86-64 machine requires the execution of about 10^6 benchmarks on the CPU.

Each of these measures takes time: the multiset of instructions must be transformed into an assembly code, including the register mapping phrase; this assembly must be assembled and linked into an ELF file; and finally, the benchmark must be actually executed, with multiple warm-up rounds and multiple measures. On average, on the `SKL-SP` CPU, each benchmark requires half to two-thirds of a second on a single core. The whole benchmarking phase, on the `SKL-SP` processor, roughly took eight hours.

As `Palmed` relies on the Gurobi optimizer, which is itself non-deterministic, `Palmed` cannot be made truly reproducible. However, the slight fluctuations in measured cycles between two executions of a benchmark are also a major source of non-determinism in the execution of Palmed.

For both these reasons, we implemented into `Palmed` a database-backed storage of measurements. Whenever `Palmed` needs to measure a kernel, it will first try to find a corresponding measure in the database; if the measure does not exist yet, it will be run, then stored in database.

For each measure, we further store for context: the time and date at which the measure was made; the machine on which the measure was made; how many times the measure was repeated; how many warm-up rounds were performed; how many instructions were in the unrolled loop; how many instructions were executed per repetition in total; the parameters for `Pipedream`'s assembly generation procedure; how the final result was aggregated from the repeated measures; the variance of the set of measures; how many CPU cores were active when the measure was made; which CPU core was used for this measure; whether the kernel's scheduler was set to FIFO mode.

We believe that, as a whole, the use of a database increases the usability of `Palmed`: it is faster if some measures were already made in the past and recovers better upon error.

This also gives us a better confidence towards our results: we can easily archive and backup our experimental data, and we can easily trace the origin of a measure if needed. We can also reuse the exact same measures between two runs of `Palmed`, to ensure that the results are as consistent as possible.

General engineering contributions. Apart from purely scientific contributions, we worked on improving `Palmed` as a whole, from the engineering point of view: code quality; reliable parallel measurements; recovery upon error; logging; … These improvements amount to about a hundred merge-requests between Nicolas Derumigny and myself.

Chapter 3

Beyond ports: manually modelling the A72 frontend

The usual reverse-engineering methods for CPU models usually make abundant use of hardware counters — and legitimately so, as they are the natural and accurate way to obtain insight on the internals of a CPU. Such methods include, among others, the optimisation guides from Agner Fog [Fog20], as well as `uops.info` [AR19] and `uiCA`'s [AR22] approach to respectively model the CPU's back- and front-end. In chapter 2, we introduced Palmed, whose main goal is to automatically produce port-mappings of CPUs without assuming the presence of specific hardware counters.

The ARM architectures occupy a growing space in the global computing ecosystem. They are already pervasive among the embedded and mobile devices, with most mobile phones featuring an ARM CPU [Haa23]. Processors based on ARM are emerging in datacenters and supercomputers: the Fugaku supercomputer — considered the fastest supercomputer in the world by the TOP500 ranking [Fuj23] — runs on ARM-based CPUs [Mat21], the MareNostrum 4 supercomputer has an ARM-based cluster [Bar20].

Yet, the ARM ecosystem is still lacking in performance debugging tooling. While `llvm-mca` supports ARM, it is one of the only few: `IACA`, made by Intel, is not supported — and will never be, as it is end-of-life —; `uiCA` is focused on Intel architectures, and cannot be easily ported as it heavily relies on reverse engineering specific to Intel, and enabled by specific hardware counters; Intel `VTune`, a commonly used profiling performance analysis tool, supports only x86-64.

In this context, modelling an ARM CPU — the Cortex A72 — with `Palmed` seemed to be an important goal, especially meaningful as this particular CPU only has very few hardware counters. However, it yielded only mixed results, as we will see in section 3.4.

In this chapter, we show that a major cause of imprecision in these results is the absence in `Palmed` of a frontend model. We manually model the Cortex A72 frontend to compare a raw `Palmed`-generated model, to one naively augmented with a frontend model.

While this chapter only documents a manual approach, we view it as a preliminary work towards an automation of the synthesis of a model that stems from benchmarks data, in the same way that `Palmed` synthesises a backend model. In this direction, we propose in section 3.5 a generic, parametric frontend that, we expect, could be used with good results on many architectures. We also offer methodologies that we expect to be able to automatically fill some of the parameters of this model for an arbitrary architecture.

3.1 Necessity to go beyond ports

The resource models produced by `Palmed` are mainly concerned with the backend of the CPUs modeled. However, the importance of the frontend in the accuracy of a model's prediction cannot be ignored. Its effect can be clearly seen in the evaluation heatmaps of various code analyzers in Figure 2.4. Each heatmap has a clear-cut limit on the horizontal axis: independently of the benchmark's content, it is impossible to reach more than a given number of instructions per cycle for a given processor — 4 instructions for the `SKL-SP`, 5 for the `ZEN1`. This limit is imposed by the frontend.

Some analyzers, such as `Palmed` and `IACA`, model this limit: the heatmap shows that the predicted IPC will not surpass this limit. The other three analyzers studied, however, do not model this limit; for instance, `uops.info` has a high density of benchmarks predicted at 8 instructions per cycle on SPEC2017 on the `SKL-SP` CPU, while the native measurement yielded only 4 instructions per cycle. The same effect is visible on `PMEvo` and `llvm-mca` heatmaps.

> **Example** (*High back-end throughput on `SKL-SP`*)
>
> On the `SKL-SP` microarchitecture, assuming an infinitely large frontend, a number of instructions per cycle higher than 4 is easy to reach.
>
> According to `uops.info` data, a 64-bits integer `addq` is processed with a single μOP, dispatched on port 0, 1, 5 or 6. In the meantime, a simple form 64 bits register store to a direct register-held address — *eg.* a `movq %rax, (%rbx)` — is also processed with a single μOP, dispatched on port 2 or 3.
>
> Thus, backend-wise, the kernel $4 \times \texttt{addq} + 2 \times \texttt{mov}$ has a throughput of 6 instructions per cycle. However, in reality, this kernel would be frontend-bound, with a theoretical maximum throughput of 4 instructions per cycle — in fact, a `Pipedream` measure only yields 3 instructions per cycle.

To account for this, `Palmed` tries to detect an additional resource, apart from the backend ports and combined ports, on which every μOP incurs a load. This allows `Palmed` to avoid large errors on frontend-bound kernels.

The approach is, however, far from perfect. The clearest reason for this is is that the frontend, both on x86-64 and ARM architectures, works in-order, while `Palmed` inherently models kernels as multisets of instructions, thus completely ignoring ordering. This resource model is purely linear: an instruction incurs a load on the frontend resource in a fully commutative way, independently of the previous instructions executed this cycle and of many other effects.

The article introducing `uiCA` [AR22] explores this question in detail for x86-64 Intel architectures. The authors, having previously developed `uops.info`, discuss the importance of a correct modelling of the frontend to accurately predict throughput. Their approach, based on the exploration and reverse-engineering of the crucial parts of the frontend, showcases many important and non-trivial aspects of frontends usually neglected, such as the switching between the decoders and μOP-cache as source of instructions — which cannot be linearly modelled.

3.2 The Cortex A72 CPU

The Cortex A72 [ARM] is a CPU based on the ARMv8-A ISA — the first ARM ISA to implement Aarch64, the 64-bits ARM extension. It is an out-of-order CPU, with Neon SIMD support. The CPU is designed as a general-purpose, high-performance core for low-power applications.

The Raspberry Pi 4 uses a 4-cores A72 CPU, implemented by Broadcom as BCM2711; it is thus easy to have access to an A72 to run experiments.

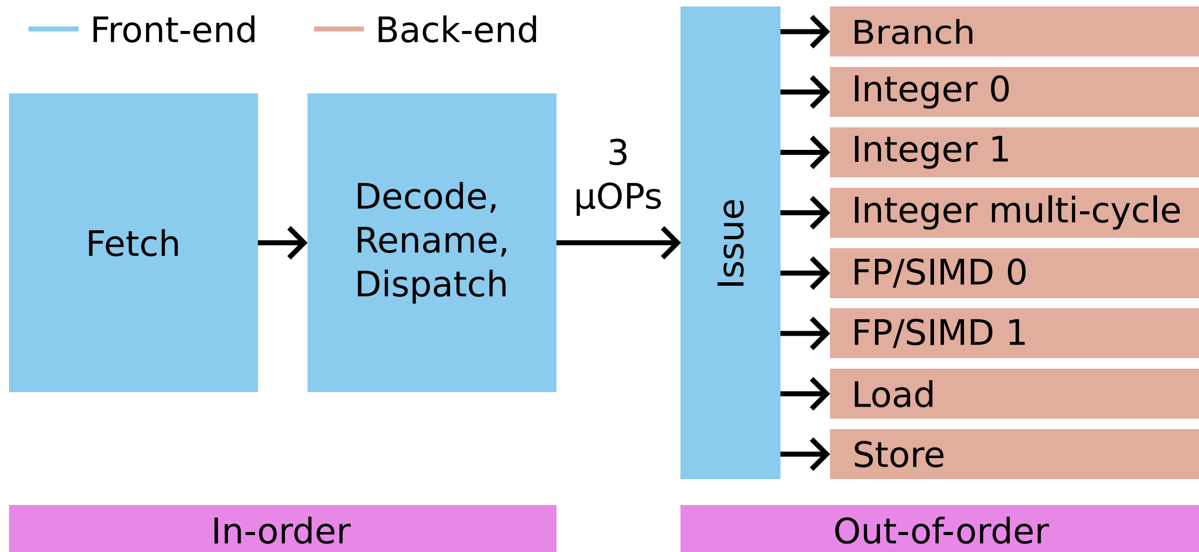


Figure 3.1 – Simplified overview of the Cortex A72 pipeline

Backend. As can be seen in Figure 3.1 (adapted from the software optimization guide for the Cortex A72, published by ARM [15]), the Cortex A72 has eight execution ports:

- a branch port (branch instructions, equivalent to x86 jumps);
- two identical integer ports (integer arithmetic operation), noted `Int01`;
- an integer multi-cycle port (complex integer operations, *eg.* divisions), noted `IntM`;
- two nearly-identical floating point and SIMD ports, noted `FP0` and `FP1`, or `FP01` to denote both. They are mostly identical, with slight specializations: *eg.* only port `FP0` can do SIMD multiplication, while only port `FP1` can do floating point comparisons);
- a load port, noted `Ld`;
- a store port, noted `St`.

Frontend. The Cortex A72 frontend can only decode three instructions and dispatch three μOPs per cycle [15]. Intel's `SKL-SP`, which we considered before, has a frontend that bottlenecks at four μOPs per cycle [Fog16]. This difference of one μOP per cycle is actually meaningful, as this means that only three of the eight backend ports can be used each cycle.

> **Example** (*2nd order polynomial evaluation*)
>
> Consider a kernel evaluating the 2nd order polynomial expression for different values of x:
>
> $$\begin{aligned} P[i] &= aX[i]^2 + bX[i] + c \\ &= (aX[i] + b) \times X[i] + c \end{aligned}$$
>
> which directly translates to four operations: load $X[i]$, two floating point multiply-add, store the result $P[i]$. The backend, having a load port, two SIMD ports and a store port, can execute one iteration of such a kernel every cycle; in steady-state, out-of-order execution can lift the latency-induced pressure. However, as the frontend bottlenecks at three μOPs per cycle, this kernel does not fit in a single cycle.

Lack of hardware counters. The Cortex A72 only features a very limited set of specialized hardware counters. While the CPU is able to report the number of elapsed cycles, retired instructions, branch misses and various metrics on cache misses, it does not report any event

regarding macro- or micro-operations, dispatching or issuing to specific ports. This makes it, as pointed before, a particularly relevant target for `Palmed`.

3.3 Manually modelling the A72 frontend

Our objective is now to manually construct a frontend model of the Cortex A72. We strive, however, to remain as close to an algorithmic methodology as possible: while our model's structure is manually crafted, its data should come from experiments that can be later automated.

3.3.1 Finding micro-operation count for each instruction

As we saw in section 3.2, the Cortex A72's frontend can only dispatch three μOPs per cycle. The first important data to collect, thus, is the number of μOPs each instruction is decoded into.

To that end, the optimisation manual [15] helps, but is not thorough enough: for each instruction, it lists the ports on which load is incurred, which sets a lower bound to the number of μOPs the instruction is decomposed into. This approach, however, is not really satisfying. First, because it cannot be reproduced for another architecture whose optimisation manual is not as detailed, cannot be automated, and fully trusts the manufacturer. Second, because if an instruction loads *eg.* the integer ports, it may have a single or multiple μOPs executed on the integer ports; the manual is only helpful to some extent to determine this.

We instead use an approach akin to `Palmed`'s saturating kernels, itself inspired by Agner Fog's method to identify ports in the absence of hardware counters [Fog20]. To this end, we assume the availability of a port mapping for the backend — in the case of the Cortex A72, we use `Palmed`'s output, sometimes manually confronted with the software optimisation guide [15]; `uops.info` could also be used on the architectures it models.

The `Palmed` resource mapping we use as a basis is composed of 1 975 instructions. To make this more manageable in a semi-automated method, we reuse the instruction classes provided by `Palmed`, introduced in section 2.2, as instructions in the same class are mapped to the same resources, and thus are decomposed into the same μOPs; this results in only 98 classes of instructions.

Basic instructions. We use `Palmed`'s mapping to hand-pick *basic instructions*: for each port, we select one instruction which decodes into a single μOP executed by this port. We use the following instructions, in `Pipedream`'s notation:

- integer 0/1: `ADC_RD_X_RN_X_RM_X`, *eg.* `adc x0, x1, x2`;
- integer multi-cycle: `MUL_RD_W_RN_W_RM_W`, *eg.* `mul w0, w1, w2`;
- load: `LDR_RT_X_ADDR_REGOFF`, *eg.* `ldr x0, [x1, x2]`;
- store: `STR_RT_X_ADDR_REGOFF`, *eg.* `str x0, [x1, x2]`;
- FP/SIMD 0: `FRINTA_FD_D_FN_D`, *eg.* `frinta d0, d1` (floating-point rounding to integral);
- FP/SIMD 1: `FCMP_FN_D_FM_D`, *eg.* `fcmp d0, d1` (floating-point comparison);
- FP/SIMD 0/1: `FMIN_FD_D_FN_D_FM_D`, *eg.* `fmin d0, d1, d1` (floating-point minimum);
- (Branch: no instruction, as they are unsupported by `Pipedream`).

As the integer ports are not specialized, a single basic instruction is sufficient for both of them. The FP/SIMD ports are slightly specialized (see section 3.2), we thus use three basic instructions: one that stresses each of them independently, and one that stresses both without distinction.

For each of these ports, we note $\mathcal{B}_{\texttt{p}}$ the basic instruction for port `p`; *eg.*, $\mathcal{B}_{\texttt{Int01}}$ is `ADC_RD_X_RN_X_RM_X`.

Counting the micro-ops of an instruction. There are three main sources of bottleneck for a kernel $\mathcal{K}$: backend, frontend and dependencies. When measuring the execution time with `Pipedream`, we eliminate (as far as possible) the dependencies, leaving us with only backend and frontend. We note $\overline{\mathcal{K}}^{\mathbf{F}}$ the execution time of $\mathcal{K}$ if it was only limited by its frontend, and $\overline{\mathcal{K}}^{\mathbf{B}}$ the execution time of $\mathcal{K}$ if it was only limited by its backend. If we consider a kernel $\mathcal{K}$ that is simple enough to exhibit a purely linear frontend behaviour — that is, the frontend's throughput is a linear function of the number of μOPs in the kernel —, we then know that either $\overline{\mathcal{K}} = \overline{\mathcal{K}}^{\mathbf{F}}$ or $\overline{\mathcal{K}} = \overline{\mathcal{K}}^{\mathbf{B}}$.

For a given instruction i and for a certain $k \in \mathbb{N}$, we then construct a kernel $\mathcal{K}_k$ such that:

(i) $\mathcal{K}_k$ is composed of the instruction i, followed by k basic instructions;

(ii) the kernel $\mathcal{K}_k$ is simple enough to exhibit this purely linear frontend behaviour;

(iii) $\overline{\mathcal{K}_k}^{\mathbf{B}} \leq \overline{\mathcal{K}_k}^{\mathbf{F}}$.

We denote by $\#_\mu\mathcal{K}$ the number of μOPs in kernel $\mathcal{K}$. Under the condition (ii), we have for any $k \in \mathbb{N}$

$$\begin{aligned}
\overline{\mathcal{K}_k}^{\mathbf{F}} &= \frac{\#_\mu\left(\mathcal{K}_k\right)}{3} && \text{for the A72} \\
&= \frac{\#_\mu i + k}{3} && \text{by condition (i)} \\
&\geq \frac{k+1}{3}
\end{aligned}$$

We pick $k_0 := 3\left\lceil\bar{\imath}\right\rceil - 1$. Thus, we have $\left\lceil\bar{\imath}\right\rceil \leq \overline{\mathcal{K}_{k_0}}^{\mathbf{F}} \leq \overline{\mathcal{K}_{k_0}}$. Condition (iii) can then be relaxed as $\overline{\mathcal{K}_{k_0}}^{\mathbf{B}} \leq \left\lceil\bar{\imath}\right\rceil$, which we know to be true if the load from $\mathcal{K}_{k_0}$ on each port does not exceed $\left\lceil\bar{\imath}\right\rceil$ (as execution takes at least this number of cycles).

We build $\mathcal{K}_{k_0}$ by adding basic instructions to i, using the port mapping to pick basic instructions that do not load a port over $\left\lceil\bar{\imath}\right\rceil$. This is always possible, as we can load independently seven ports (leaving out the branch port), while each instruction can load at most three ports by cycle it takes to execute — each μOP is executed by a single port, and only three μOPs can be dispatched per cycle —, leaving four ports under-loaded. We build $\mathcal{K}_{k_0+1}$ the same way, still not loading a port over $\left\lceil\bar{\imath}\right\rceil$; in particular, we still have $\overline{\mathcal{K}_{k_0+1}}^{\mathbf{B}} \leq \left\lceil\bar{\imath}\right\rceil \leq \overline{\mathcal{K}_{k_0+1}}^{\mathbf{F}}$. To ensure that condition (ii) is valid, as we will see later in section 3.3.2, we spread as much as possible instructions loading the same port: for instance, $i + \mathcal{B}_{\texttt{Int01}} + \mathcal{B}_{\texttt{FP01}} + \mathcal{B}_{\texttt{Int01}}$ is preferred over $i + 2 \times \mathcal{B}_{\texttt{Int01}} + \mathcal{B}_{\texttt{FP01}}$.

Unless condition (ii) is not met or our ports model is incorrect for this instruction, we should measure $\left\lceil\bar{\imath}\right\rceil \leq \overline{\mathcal{K}_{k_0}}$ and $\overline{\mathcal{K}_{k_0}} + 1/3 = \overline{\mathcal{K}_{k_0+1}}$. For instructions i where it is not the case, increasing k_0 by 3 or using other basic instructions eventually yielded satisfying measures. Finally, we obtain

$$\#_\mu i = 3\overline{\mathcal{K}_{k_0}} - k_0$$

Applying this procedure manually on each instruction class provides us with a model mapping each supported instruction of the ISA to its μOP count.

> **Example** (*μOP count measure:* `ADC_RD_X_RN_X_RM_X`)
>
> We measure the μOP-count of $i =$ `ADC_RD_X_RN_X_RM_X`, our basic instruction for the integer port.
>
> We measure $\bar{\imath} = 0.51 \simeq 1/2$ cycle; hence, we consider $\mathcal{K}_2$ and $\mathcal{K}_3$. Our mapping indicates that this instruction loads only the `Int01` port with a load of $1/2$.

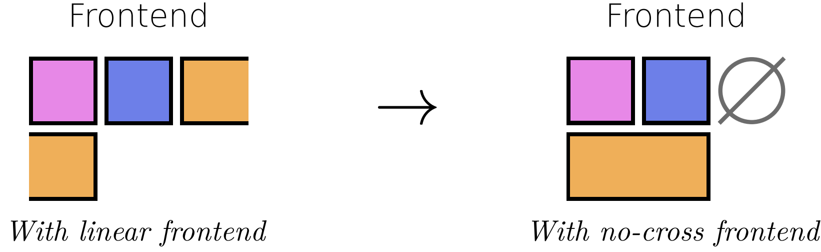


Figure 3.2 – Illustration of the no-cross frontend model. Rows represent CPU cycles.

We select *eg.* $\mathcal{K}_2 = i + 2 \times \mathcal{B}_{\texttt{FP01}}$ and $\mathcal{K}_3 = i + \mathcal{B}_{\texttt{FP01}} + \mathcal{B}_{\texttt{Ld}} + \mathcal{B}_{\texttt{FP01}}$.
We measure
— $\overline{\mathcal{K}_2} = 1.01 \simeq 1$ cycle
— $\overline{\mathcal{K}_3} = 1.35 \simeq 1\,{}^1\!/\!_3$ cycles
which is consistent. We conclude that, as expected, $\#_\mu i = 3\overline{\mathcal{K}_2} - 2 = 3 - 2 = 1$.

Example (*µOP count measure:* `ADDV_FD_H_VN_V_8H`)

We measure the μOP-count of $i =$ `ADDV_FD_H_VN_V_8H`, the SIMD “add across vector” operation on a vector of eight sixteen-bits operands.

We measure $\bar{\imath} = 1.01 \simeq 1$ cycle; hence, we consider $\mathcal{K}_2$ and $\mathcal{K}_3$. Our mapping indicates that this instruction loads the `FP1` port with a load of 1, and the `FP01` port with a load of 1 [a].

We select *eg.* $\mathcal{K}_2 = i + 2 \times \mathcal{B}_{\texttt{Int01}}$ and $\mathcal{K}_3 = i + \mathcal{B}_{\texttt{Int01}} + \mathcal{B}_{\texttt{Ld}} + \mathcal{B}_{\texttt{Int01}}$.
We measure
— $\overline{\mathcal{K}_2} = 1.35 \simeq 1\,{}^1\!/\!_3$ cycles
— $\overline{\mathcal{K}_3} = 1.68 \simeq 1\,{}^2\!/\!_3$ cycles
which is consistent. We conclude that $\#_\mu i = 3\overline{\mathcal{K}_2} - 2 = 4 - 2 = 2$.

a. The `FP01` port has a throughput of 2, hence a load of 1 means two μOPs. As there is already a μOP loading the `FP1` port, which also loads the combined port `FP01`, this can be understood as one μOP on `FP1` exclusively, plus one on either `FP0` or `FP1`.

3.3.2 Bubbles in the pipeline

The frontend, however, does not always exhibit a purely linear behaviour. We consider for instance the kernel $\mathcal{K} =$ `ADDV_FD_H_VN_V_8H` $+3 \times \mathcal{B}_{\texttt{Int01}}$; for the rest of this chapter, we refer to `ADDV_FD_H_VN_V_8H` as simply `ADDV` when not stated otherwise.

Backend-wise, `ADDV` fully loads `FP1` and `FP01`, while $\mathcal{B}_{\texttt{Int01}}$ half-loads `Int01`. The port most loaded by $\mathcal{K}$ is thus `Int01`, with a load of $1\,{}^1\!/\!_2$. We then expect $\overline{\mathcal{K}}^{\mathbf{B}} = 1\,{}^1\!/\!_2$.

Frontend-wise, `ADDV` decomposes into two μOPs, while $\mathcal{B}_{\texttt{Int01}}$ decomposes into a single μOPs; thus, $\#_\mu\mathcal{K} = 5$. We then expect $\overline{\mathcal{K}}^{\mathbf{F}} = 1\,{}^2\!/\!_3$.

As the frontend dominates the backend, we expect $\overline{\mathcal{K}} = \overline{\mathcal{K}}^{\mathbf{F}} = 1\,{}^2\!/\!_3$. However, in reality, we measure $\overline{\mathcal{K}} = 2.01 \simeq 2$ cycles.

From then on, we strive to find a model that could reliably predict, given a kernel, how many cycles it requires to execute, frontend-wise, in a steady-state.

No-cross model

On the x86-64 architectures they analyzed, `uiCA`'s authors find that the CPU's predecoder might cause an instruction's μOPs to be postponed to the next cycle if it is pre-decoded across

a cycle boundary [AR22, §4.1].

We hypothesize that the same kind of effect could postpone an instruction's μOPs until the next cycle if its μOPs would cross a cycle boundary otherwise. This behaviour is illustrated in Figure 3.2, with a kernel composed of three instructions: the first two each decode to a single μOP, while the third one decodes to two μOPs. In this figure, each row represents a CPU cycle, while each square represents a μOP-slot in the frontend; there are thus at most three squares in each row. In the no-cross case (right), the constraint forced the third instruction to start its decoding at the beginning of the second cycle, leaving a "bubble" in the frontend on the first cycle.

This model explains the $\mathcal{K} = \texttt{ADDV} + 3 \times \mathcal{B}_{\texttt{Int01}}$ example introduced above, as depicted in Figure 3.3, where $\mathcal{K}$ is represented twice, to ensure that the steady-state was reached. Here, the frontend indeed requires two full cycles to issue $\mathcal{K}$, which is consistent with our measure.

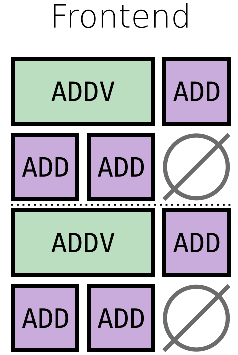
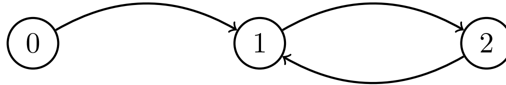


Figure 3.3 – No-cross frontend for $\texttt{ADDV} + 3 \times \mathcal{B}_{\texttt{Int01}}$

The notion of steady-state is, in the general case, not as straightforward: it is well possible that, after executing the kernel once, the second iteration of the kernel does not begin at the cycle boundary. The state of the model, however, is entirely defined by the number $s \in \{0, 1, 2\}$ of μOPs already decoded this cycle. Thus, if at the end of a full execution of a kernel, s is equal to a state previously encountered at the end of a kernel, k kernel iterations before, steady-state was reached for this portion: we know that further executing the kernel k times will bring us again to the same state. The steady-state execution time, frontend-wise, of a kernel is then the number of elapsed cycles between the beginning and end of the steady-state pattern (as the start and end state are the same), divided by the number of kernel repetitions inside the pattern.

The no-cross model is formalized by the `next_state` function defined in Listing 1 in Python.

There are two main phases when repeatedly applying the `next_state` function. Consider the following example of a graph representation of the `next_state` function, ignoring the `cycles_started` return value:



When repeatedly applied starting from 0, the `next_state` function will yield the sequence $0, 1, 2, 1, 2, 1, 2, \ldots$. The first iteration brings us to state 1, which belongs to the steady-state; starting from there, the next iterations will loop through the steady-state.

In the general case, the model iterates the `next_state` function starting from state 0 until a previously-encountered state is reached — this requires at most three iterations. At this point, steady-state is reached. The function is further iterated until the same state is encountered again — also requiring at most three iterations —. The number of elapsed cycles during this second phase, divided by the number of iterations of the function, is returned as the predicted steady-state execution time of the kernel, frontend-wise.

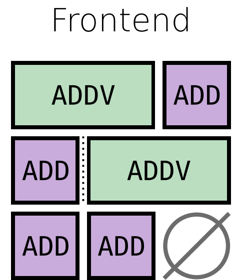


Figure 3.4 – No-cross frontend for $\texttt{ADDV} + 2 \times \mathcal{B}_{\texttt{Int01}}$

This model, however, is not satisfactory in many cases. For instance, the kernel $\mathcal{K}' = \texttt{ADDV} + 2 \times \mathcal{B}_{\texttt{Int01}}$ is predicted to run in 1.5 cycles, as depicted in Figure 3.4; however, a `Pipedream` measure yields $\overline{\mathcal{K}'} = 1.35 \simeq 1\,{}^{1}\!/_{3}$ cycles.

```python
State = type([0, 1, 2])  # incorrect Python for clarity


def next_state(cur_state: State, kernel: list[int]) -> tuple[State, int]:
    """Returns a tuple (next_state, cycles_elapsed). An instruction in the
    kernel is represented by its number of micro-ops."""

    cycles_started = 0

    for insn_uops in kernel:
        if cur_state > 0 and cur_state + insn_uops > 3:
            # Start a new cycle instead, creating bubbles
            cur_state = insn_uops % 3
            cycles_started += 1 + (insn_uops // 3)
        else:
            # General "linear" case
            n_state = cur_state + insn_uops
            cur_state = n_state % 3
            cycles_started += n_state // 3

    # Normalize the state: if the first instruction of the next kernel
    # would not fit in this cycle, initialize the next cycle.
    if cur_state > 0 and cur_state + kernel[0] > 3:
        cur_state = 0
        cycles_started += 1

    return (cur_state, cycles_started)
```

Listing 1 – Implementation of the `next_state` function for the no-cross frontend model

Dispatch-queues model

The software optimisation guide, however, details additional dispatch constraints in Section 4.1 [15]. While it confirms the dispatch constraint of at most three μOPs per cycle, it also introduces more specific constraints. There are six dispatch pipelines, that each bottleneck at less than three μOPs dispatched each cycle:

Pipeline	Related ports	μOPs/cyc.
Branch	Branch	1
Int	Int01	2
IntM	IntM	2
FP0	FP0	1
FP1	FP1	1
LdSt	Ld, St	2

These dispatch constraints could also explain the $\mathcal{K} = \texttt{ADDV} + 3 \times \mathcal{B}_{\texttt{Int01}}$ measured run time, as detailed in Figure 3.5: the frontend enters steady state on the fourth cycle and, on the fifth (and every second cycle from then on), can only execute two μOPs, as the third one would be a third μOP loading the `Int` dispatch queue, which can only dispatch two μOPs per cycle. As this part of the CPU is in-order, the frontend is stalled until the next cycle, leaving a dispatch bubble. In steady-state, the dispatch-queues model thus predicts $\overline{\mathcal{K}}^{\mathbf{F}} = 2$ cycles, which is consistent with our measure.

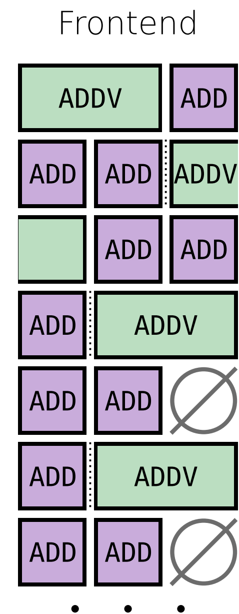


Figure 3.5 – Disp. queues frontend for $\texttt{ADDV} + 3 \times \mathcal{B}_{\texttt{Int01}}$

This model also explains the $\mathcal{K}' = \texttt{ADDV} + 2 \times \mathcal{B}_{\texttt{Int01}}$ measured time, as detailed in Figure 3.6: the dispatch-queues constraints do not force any bubble in the dispatch pipeline. The model

thus predicts $\overline{\mathcal{K}'}^{\mathbf{F}} = 1\,^1\!/_3$, which is consistent with our measure.

Finding a dispatch model. This model, however, cannot be deduced straightforwardly from our previous μOPs model: each μOP needs to further be mapped to a dispatch queue.

The model yielded by `Palmed` is not usable as-is to deduce the dispatch queues used by each instruction: the detected resources are not a perfect match for the CPU, and some resources tend to be duplicated due to measurement artifacts. For instance, the `Int01` resource might be duplicated into r_0 and r_1, with some integer operations loading r_0, some loading r_1, and a majority loading both — while it makes for a reasonably good throughput model, it would require extra cleaning work to be used as a dispatch model. It is, however, a good basis for manual mapping: for instance, the `Ld` and `St` ports are one-to-one matches, and allow to automatically map all load and store operations.

We generate a base dispatch model from the Palmed model, and manually cross-check each class of instructions using the optimisation manual, with `Pipedream` measures in some cases to further clarify.

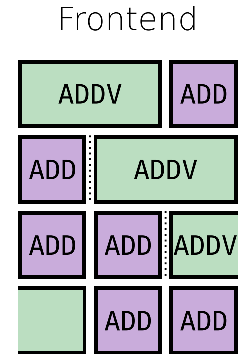


Figure 3.6 – Disp. queues frontend for `ADDV` $+ 2 \times \mathcal{B}_{\texttt{Int01}}$

This method trivially works for most instructions, which are built out of a single μOP and for which we find a single relevant dispatch queue. However, instructions where this is not the case require specific processing. For an instruction i, with $u = \#_\mu i$ μOPs and found to belong to d distinct dispatch queues, we break down the following cases.

- If $d > u$, our μOPs model is wrong for this instruction, as each μOP belongs to a single dispatch queue. This did not happen in our model.
- If $d = 1, u > 1$, each μOP belongs to the same dispatch queue.
- If $d = u > 1$, each μOP belongs to its own dispatch queue. For now, our model orders those μOPs arbitrarily. However, the order of those μOPs might be important due to dispatch constraints, and would require specific investigation with kernels meant to stress the dispatch queues, assuming a certain μOP order.
- If $1 < d < u$, we do not have enough data to determine how many μOPs belong to each queue; this would require further measurements. We do not support those instructions, as they represent only 35 instructions out of the 1749 instructions supported by our `Palmed`-generated backend model.

Due to the separation of `FP0` and `FP1` in two different dispatch queues, this model also borrows the abstract resources' formalism in a simple case: it actually models seven queues, including `FP0`, `FP1` and `FP01`, the two former with a maximal load of 1 and the latter with a maximal load of 2. Any μOP loading queues `FP0` or `FP1` also load the `FP01` queue likewise.

Implementing the dispatch model. A key observation to implement this model is that, as with the no-cross model, the state of the model at the end of a kernel occurrence is still only determined by the number of μOPs already dispatched in the current cycle. Indeed, since the dispatcher is in-order, at the end of a kernel occurrence, the last μOPs dispatched will always be the same in steady-state, as the last instructions are the few last of the kernel.

On account of this observation, the general structure of the no-cross implementation remains correct: at most three kernel iterations to reach steady-state, and at most three more to find a fixed point. The `next_state` function is adapted to account for the dispatch queues limit.

			llvm-mca	Palmed with frontend…			
				none	linear	no-cross	disp. queues
SPEC	Cov.	(%)	100.0	N/A	97.21	97.21	97.16
	Err.	(%)	9.0	20.1	6.2	6.3	4.6
	τ_K	(1)	0.83	0.88	0.91	0.91	0.93
Polybench	Cov.	(%)	100.00	N/A	99.33	99.33	99.33
	Err.	(%)	13.9	12.6	8.1	8.1	8.0
	τ_K	(1)	0.47	0.82	0.88	0.88	0.90

Table 3.1 – Comparative accuracy of IPC predictions with different frontend models on the Cortex A72

3.4 Evaluation on Palmed

To evaluate the gain brought by each frontend model, we plug them successively on top of the `Palmed` backend model. The number of cycles for a kernel $\mathcal{K}$ is then predicted as the maximum between the backend-predicted time and the frontend-predicted time.

We evaluate four models: `Palmed`'s backend alone, `Palmed` with a purely linear frontend, based on our modeled number of μOPs for each instruction, `Palmed` with the no-cross frontend, and finally `Palmed` with the dispatch-queues frontend. The results of each model are reported in Table 3.1, to which we add `llvm-mca`'s results as a basis for comparison with the state-of-the-art.

As expected, the error is greatly reduced with the addition of any reasonable frontend model — especially on the SPEC benchmark suite. Using the dispatch-queues model, which models more accurately the frontend, further reduces significantly the error rate on SPEC by 1.6 points, without significantly increasing the τ_K coefficient. On Polybench, however, the gains brought by the dispatch-queues model are very modest — only 0.1 point.

In all cases, `Palmed` with a frontend model performs significantly better than `llvm-mca` on the Cortex A72.

3.5 A parametric model for future works of automatic frontend model generation

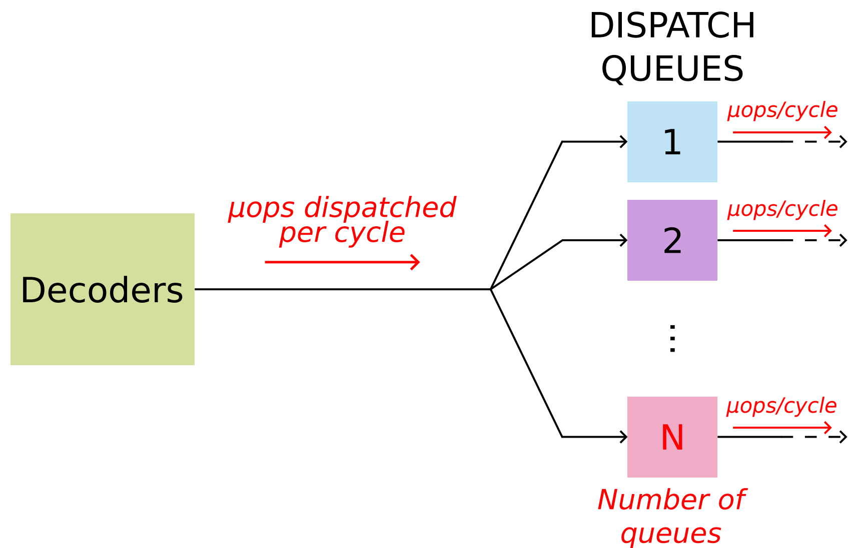
While this chapter was solely centered on the Cortex A72, we believe that this study paves the way for an automated frontend model synthesis akin to `Palmed`. This synthesis should be fully-automated; stem solely from benchmarking data and a description of the ISA; and should avoid the use of any specific hardware counter.

As a scaffold for such a future work, we propose the parametric model in Figure 3.7. Some of its parameters should be possible to obtain with the methods used in this chapter, while for some others, new methods must be devised.

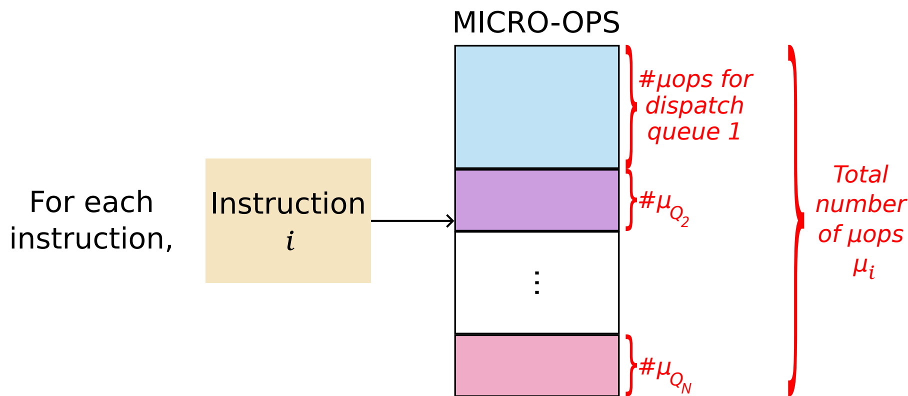
Such a model would probably be unable to account for “unusual” frontend bottlenecks — at least not at the level of detail that *eg.* `uiCA` authors gather for Intel frontends [AR22]. This level of detail, however, is possible exactly because the authors’ restricted their scope to microarchitectures that share a lot of similarity, coming from the same manufacturer. Assessing the extent of the loss of precision of an automatically-generated model, and its gain of precision *wrt.* a model without frontend, remains to be done.

Our model introduces a limited number of parameters, depicted in red italics in Figure 3.7. It is composed of two parts: a model of the frontend in itself, describing architectural parameters; and insights about each instruction. Its parameters are:

— the number of μOPs that can be dispatched overall per cycle;



(a) Frontend model



(b) Instruction model

Figure 3.7 – A generic parametric model of a processor's frontend. In red italics, the parameters which must be discovered for each architecture.

- the number of distinct dispatch queues of the processor (*eg.* memory operations, integer operations, ...);
- for each of those queues, the number of μOPs it can dispatch per cycle;
- for each instruction i,
  - its total number of μOPs μ_i;
  - the number of μOPs that get dispatched to each individual queue (summing up to μ_i).

The first step in modeling a processor's frontend should certainly be to characterize the number of μOPs that can be dispatched in a cycle. We assume that a model of the backend is known — by taking for instance a model generated by `Palmed`, using tables from `uops.info` or any other mean. To the best of our knowledge, we can safely further assume that instructions that load a single backend port only once are also composed of a single μOP. Generating a few combinations of a diversity of those and measuring their effective throughput — making sure using the backend model that the latter is not the bottleneck — and keeping the maximal throughput reached should provide a good value.

In this chapter, we obtained the number of dispatch queues and their respective throughput by reading the official documentation. Automating this part remains to be addressed to obtain an automatic model. It should be possible to make these parameters apparent by identifying "simple" instructions that conflict further than the main dispatch limitation and combining them.

The core of the model presented in this chapter is the discovery, for each instruction, of its μOP count. Still assuming the knowledge of a backend model, the method described in subsection 3.3.1 should be generic enough to be used on any processor. The basic instructions may be easily selected using the backend model — we assume their existence in most microarchitectures, as pragmatic concerns guide the ports design. Counting the μOPs of an instruction thus follows, using only elapsed cycles counters.

This method assumes that $\overline{\mathcal{K}}^{\mathbf{F}}$ bottlenecks on a global dispatch queue for $\mathcal{K}$, and not specific dispatch queues. This must be ensured by selecting well-chosen kernels — for instance, on the A72, care must be taken to interleave instructions corresponding to diverse enough dispatch pipelines.

Finally, the break-down of each instruction's μOPs into their respective dispatch queues should follow from the backend model, as each dispatch queue is tied to a subset of backend ports.

The question of complex decoders. While the ARM ISA is composed of instructions of fixed length, making decoding easier, such is not always the case. The x86 ISA, for one, uses instructions that vary in length from one to fifteen bytes [23c]. Larger instructions may prove to be a huge frontend slowdown, especially when such instructions cross an instruction cache line boundary [AR22].

Processors implementing ISAs subject to decoding bottleneck typically also feature a decoded μOP cache, or *decoded stream buffer* (DSB). The typical hit rate of this cache is about 80% [23c, Section B.5.7.2; Ren+21]. However, code analyzers are concerned with loops and, more generally, hot code portions. Under such conditions, we expect this cache, once hot in steady-state, to be very close to a 100% hit rate. In this case, only the dispatch throughput will be limiting, and modeling the decoding bottlenecks becomes irrelevant.

Points of vigilance and limitations. This parametric model aims to be a compromise between simplicity of automation and good accuracy. Experimentation may prove that it lacks some important features to be accurate. Depending on the architecture targeted, the following points should be investigated if the model does not reach the expected accuracy.

- We introduced just above the DSB (μOP cache). This model considers that the DSB will never be the cause of a bottleneck and that, instead, the number of dispatched μOPs per cycle will always bottleneck before. This might not be true, as DSBs are complex in themselves already [AR22].
- Intel CPUs use a Loop Stream Detector (LSD) to keep in the decode queue a whole loop's body of μOPs if the frontend detects that a small enough loop is repeated [AR22; Ren+21]. In this case, μOPs are repeatedly streamed from the decode queue, without even the necessity to hit a cache. We are unaware of similar features in other commercial processors. In embedded programming, however, *hardware loops* — which are set up explicitly by the programmer — achieve, among others, the same goal [SRO04; Kav07; TJ01].
- The *branch predictor* of a CPU is responsible for guessing, before the actual logic is computed, whether a conditional jump will be taken. A misprediction forces the frontend

to re-populate its queues with instructions from the branch actually taken and typically stalls the pipeline for several cycles [ESE06]. Our model, however, does not include a branch predictor for much the same reason that it does not include complex decoder: in steady-state, in a hot code portion, we expect the branch predictor to always predict correctly.

— In reality, there is an intermediary step between instructions and μOPs: macro-ops. Although it serves a designing and semantic purpose, we omit this step in the current model as — we believe — it is of little importance to predict performance.

— On x86 architectures at least, common pairs of micro- or macro-operations may be "fused" into a single one, up to various parts of the pipeline, to save space in some queues or artificially boost dispatch limitations. This mechanism is implemented in Intel architectures, and to some extent in AMD architectures since Zen [23b, §3.4.2; AR22; Vis+21]. This may make some kernels seem to "bypass" dispatch limits.

Chapter 4

A more systematic approach to throughput prediction performance analysis: CesASMe

In the previous chapters, we focused on two of the main bottleneck factors for computation kernels: chapter 2 investigated the backend aspect of throughput prediction, while chapter 3 dived into the frontend aspects.

Throughout those two chapters, we entirely left out another crucial factor: dependencies, and the latency they induce between instructions. We managed to do so, because our baseline of native execution was `Pipedream` measures, *designed* to suppress any dependency.

However, state-of-the-art tools strive to provide an estimation of the execution time $\overline{\mathcal{K}}$ of a given kernel $\mathcal{K}$ that is *as precise as possible*, and as such, cannot neglect this third major bottleneck. An exact throughput prediction would require a cycle-accurate simulator of the processor, based on microarchitectural data that is most often not publicly available, and would be prohibitively slow in any case. These tools thus each solve in their own way the challenge of modeling complex CPUs while remaining simple enough to yield a prediction in a reasonable time, ending up with different models. For instance, on the following x86-64 basic block computing a general matrix multiplication,

```
1 movsd (%rcx, %rax), %xmm0
2 mulsd %xmm1, %xmm0
3 addsd (%rdx, %rax), %xmm0
4 movsd %xmm0, (%rdx, %rax)
5 addq $8, %rax
6 cmpq $0x2260, %rax
7 jne 0x16e0
```

`llvm-mca` predicts 1.5 cycles, `IACA` and `Ithemal` predict 2 cycles, while `uiCA` predicts 3 cycles. One may wonder which tool is correct.

In this chapter, we take a step back from our previous contributions, and assess more generally the landscape of code analyzers. What are the key bottlenecks to account for if one aims to predict the execution time of a kernel correctly? Are some of these badly accounted for by state-of-the-art code analyzers? This chapter, by conducting a broad experimental analysis of these tools, strives to answer these questions.

In section 4.1, we investigate how a kernel's execution time may be measured if we want to correctly account for its dependencies. We advocate for the measurement of the total execution time of a computation kernel in its original context, coupled with a precise measure of its number of iterations to normalize the measure.

We then present a fully-tooled solution to evaluate and compare the diversity of static

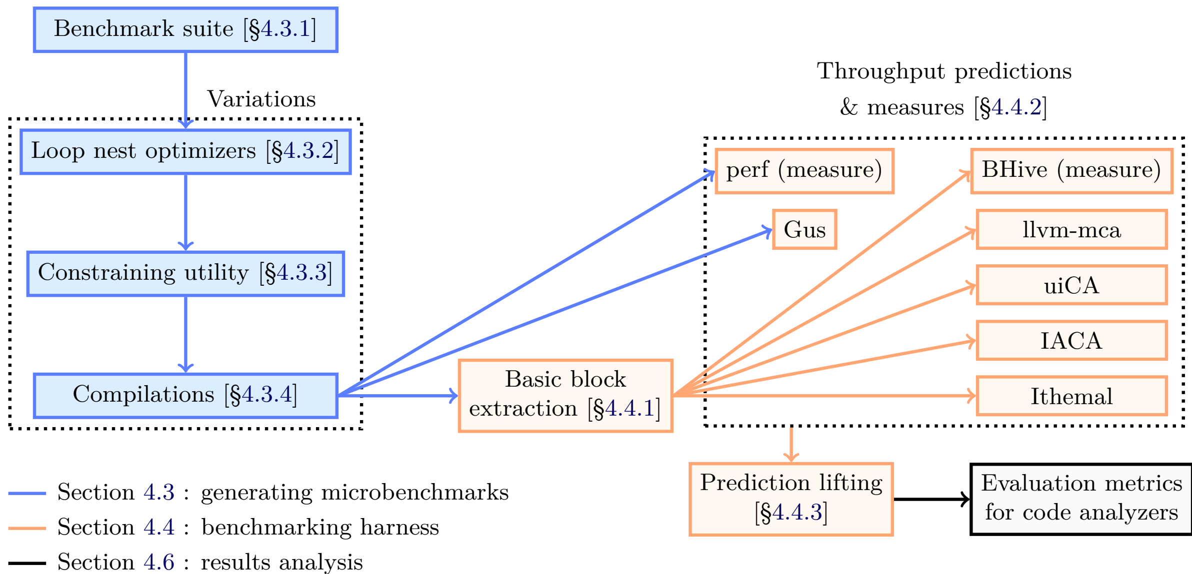


Figure 4.1 – Our analysis and measurement environment.

throughput predictors. Our tool, CesASMe, solves two main issues in this direction. In Section 4.3, we describe how CesASMe generates a wide variety of computation kernels stressing different parameters of the architecture, and thus of the predictors' models, while staying close to representative workloads. To achieve this, we use Polybench [PY16], a C-level benchmark suite that we already introduced for Palmed in section 2.4. Polybench is composed of benchmarks representative of scientific computation workloads, that we combine with a variety of optimisations, including polyhedral loop transformations.

In Section 4.4, we describe how CesASMe is able to evaluate throughput predictors on this set of benchmarks by lifting their predictions to a total number of cycles that can be compared to a hardware counters-based measure. A high-level view of CesASMe is shown in Figure 4.1.

In Section 4.5, we detail our experimental setup and assess our methodology. In Section 4.6, we compare the predictors' results and analyze the results of CesASMe. In addition to statistical studies, we use CesASMe's results to investigate analyzers' flaws. We show that code analyzers do not always correctly model data dependencies through memory accesses, substantially impacting their precision.

4.1 Re-defining the execution time of a kernel

We saw above that state-of-the-art code analyzers disagreed by up to 100 % on the execution time of a relatively simple kernel. The obvious solution to assess their predictions is to compare them to an actual measure. However, accounting for dependencies at the scale of a basic block makes this *actual measure* not as trivially defined as it would seem. Take for instance the following kernel:

```
1 mov (%rax, %rcx, 1), %r10
2 mov %r10, (%rbx, %rcx, 1)
3 add $8, %rcx
```

At first, it looks like an array copy from location %rax to %rbx. Yet, if before the loop, %rbx is initialized to %rax+8, there is a read-after-write dependency between the first instruction and the second instruction at the previous iteration; which makes the throughput drop significantly. As we shall see in Section 4.6.2, *without proper context, a basic block's throughput is not well-defined.*

To recover the context of each basic block, we reason instead at the scale of a C source code.

This makes the measures unambiguous: one can use hardware counters to measure the elapsed cycles during a loop nest. This requires a suite of benchmarks, in C, that both is representative of the domain studied, and wide enough to have a good coverage of the domain. However, this is not in itself sufficient to evaluate static tools: on the preceding matrix multiplication kernel, counters report 80,059 elapsed cycles — for the total loop. This number compares hardly to `llvm-mca`, `IACA`, `Ithemal`, and `uiCA` basic block-level predictions seen above.

A common practice to make these numbers comparable is to renormalize them to instructions per cycles (IPC). Here, `llvm-mca` reports an IPC of $7/1.5 = 4.67$, `IACA` and `Ithemal` report an IPC of $7/2 = 3.5$, and `uiCA` reports an IPC of $7/3 = 2.3$. In this case, the measured IPC is 3.45, which is closest to `IACA` and `Ithemal`. Yet, IPC is a metric for microarchitectural load, and *tells nothing about a kernel's efficiency*. Indeed, the static number of instructions is affected by many compiler passes, such as scalar evolution, strength reduction, register allocation, instruction selection… Thus, when comparing two compiled versions of the same code, IPC alone does not necessarily point to the most efficient version. For instance, a kernel using SIMD instructions will use fewer instructions than one using only scalars, and thus exhibit a lower or constant IPC; yet, its performance will unquestionably increase.

The total cycles elapsed to solve a given problem, on the other hand, is a sound metric of the efficiency of an implementation. We thus instead *lift* the predictions at basic-block level to a total number of cycles. In simple cases, this simply means multiplying the block-level prediction by the number of loop iterations; however, this bound might not generally be known. More importantly, the compiler may apply any number of transformations: unrolling, for instance, changes this number. Control flow may also be complicated by code versioning.

Instead of guessing this final number of iterations at the assembly level, a sounder alternative is to measure it on the final binary. In section 4.4, we present our solution to do so, using `gdb` to instrument an execution of the binary.

4.2 Related works

Another comparative study: `AnICA`. The `AnICA` framework [RH22] also attempts to comparatively evaluate various throughput predictors by finding examples on which they are inaccurate. `AnICA` starts with randomly generated assembly snippets fed to various code analyzers. Once it finds a snippet on which (some) code analyzers yield unsatisfying results, it refines it through a process derived from abstract interpretation to reach a more general category of input, *eg.* "a load to a SIMD register followed by a SIMD arithmetic operation".

A dynamic code analyzer: `Gus`. So far, this manuscript was mostly concerned with static code analyzers. Throughput prediction tools, however, are not all static. `Gus` is a dynamic tool first introduced in Fabian Gruber's PhD thesis [Gru19]. It leverages `QEMU`'s instrumentation capabilities to dynamically predict the throughput of user-defined regions of interest in whole program. In these regions, it instruments every instruction, memory access, … in order to retrieve the exact events occurring through the program's execution. `Gus` then leverages throughput, latency and microarchitectural models to analyze resource usage and produce an accurate theoretical elapsed cycles prediction.

Its main strength, however, resides in its *sensitivity analysis* capabilities: by applying an arbitrary factor to some parts of the model (*eg.* latencies, arithmetics port, …), it is possible to investigate the impact of a specific resource on the final execution time of a region of interest. It can also accurately determine if a resource is actually a bottleneck for a region, *ie.* if increasing this resource's capabilities would reduce the execution time. The output of `Gus` on a region of interest provides a very detailed insight on each instruction's resource consumption and its contribution to the final execution time. As a dynamic analysis tool, it is also able to extract the dependencies an instruction exhibits on a real run.

The main downside of `Gus`, however, is its slowness. As most dynamic tools, it suffers from a heavy slowdown compared to a native execution of the binary, oftentimes about 100× slower. While it remains a precious tool to the user willing to deeply optimize an execution kernel, it makes `Gus` highly impractical to run on a large collection of execution kernels.

An isolated basic-block profiler: `BHive`. In section 4.1 above, we advocated for measuring a basic block's execution time *in-context*. The `BHive` profiler [Che+19], initially written by `Ithemal`'s authors [MAC18] to provide their model with sufficient — and sufficiently accurate — training data, takes an orthogonal approach to basic block throughput measurement. By mapping memory at any address accessed by a basic block, it can effectively run and measure arbitrary code without context, often — but not always, as we discuss later — yielding good results.

4.3 Generating microbenchmarks

Our framework aims to generate *microbenchmarks* relevant to a specific domain. A microbenchmark is a code that is as simplified as possible to expose the behaviour under consideration. The specified computations should be representative of the considered domain, and at the same time they should stress the different aspects of the target architecture — which is modeled by code analyzers.

In practice, a microbenchmark's *computational kernel* is a simple `for` loop, whose body contains no loops and whose bounds are statically known. A *measure* is a number of repetitions n of this computational kernel, n being a user-specified parameter. The measure may be repeated an arbitrary number of times to improve stability.

Furthermore, such a microbenchmark should be a function whose computation happens without leaving the L1 cache. This requirement helps measurements and analyses to be undisturbed by memory accesses, but it is also a matter of comparability. Indeed, most of the static analyzers make the assumption that the code under consideration is L1-resident; if it is not, their results are meaningless, and can not be compared with an actual measurement.

The generation of such microbenchmarks is achieved through four distinct components, whose parameter variations are specified in configuration files : a benchmark suite, C-to-C loop nest optimizers, a constraining utility and a C-to-binary compiler.

4.3.1 Benchmark suite

Our first component is an initial set of benchmarks which materializes the human expertise we intend to exploit for the generation of relevant codes. The considered suite must embed computation kernels delimited by ad-hoc `#pragma`s, whose arrays are accessed directly (no indirections) and whose loops are affine. These constraints are necessary to ensure that the microkernelification phase, presented below, generates segfault-free code.

In this case, we use Polybench [PY16], a suite of 30 benchmarks for polyhedral compilation — of which we use only 26. The `nussinov`, `ludcmp` and `deriche` benchmarks are removed because they are incompatible with PoCC (introduced below). The `lu` benchmark is left out as its execution alone takes longer than all others together, making its dynamic analysis (*eg.* with `Gus`) impractical. In addition to the importance of linear algebra within Polybench, one of its important features is that it does not include computational kernels with conditional control flow (*eg.* `if-then-else`) — however, it does includes conditional data flow, using the ternary conditional operator of C.

4.3.2 C-to-C loop nest optimizers

Loop nest optimizers transform an initial benchmark in different ways (generate different *versions* of the same benchmark), varying the stress on resources of the target architecture, and by extension the models on which the static analyzers are based.

In this case, we chose to use the PLUTO [BRS07] and PoCC [Pou09] polyhedral compilers, to easily access common loop nest optimizations : register tiling, tiling, skewing, vectorization/simdization, loop unrolling, loop permutation, loop fusion. These transformations are meant to maximize variety within the initial benchmark suite. Eventually, the generated benchmarks are expected to highlight the impact on performance of the resulting behaviours. For instance, *skewing* introduces non-trivial pointer arithmetics, increasing the pressure on address computation units ; *loop unrolling*, among many things, opens the way to register promotion, which exposes dependencies and alleviates load-store units ; *vectorization* stresses SIMD units and decreases pressure on the front-end ; and so on.

4.3.3 Constraining utility

A constraining utility transforms the code in order to respect an arbitrary number of non-functional properties. In this case, we apply a pass of *microkernelification*: we extract a computational kernel from the arbitrarily deep and arbitrarily long loop nest generated by the previous component. The loop chosen to form the microkernel is the one considered to be the *hottest*; the *hotness* of a loop being obtained by multiplying the number of arithmetic operations it contains by the number of times it is iterated. This metric allows us to prioritize the parts of the code that have the greatest impact on performance.

At this point, the resulting code can compute a different result from the initial code; for instance, the composition of tiling and kernelification reduces the number of loop iterations. Indeed, our framework is not meant to preserve the functional semantics of the benchmarks. Our goal is only to generate codes that are relevant from the point of view of performance analysis.

4.3.4 C-to-binary compiler

A C-to-binary compiler varies binary optimization options by enabling/disabling auto-vectorization, extended instruction sets, *etc.* We use `gcc`.

Eventually, the relevance of the microbenchmarks set generated using this approach derives not only from initial benchmark suite and the relevance of the transformations chosen at each stage, but also from the combinatorial explosion generated by the composition of the four stages. In our experimental setup, this yields up to 144 microbenchmarks per benchmark of the original suite.

4.4 Benchmarking harness

To compare full-kernel cycle measurements to throughput predictions on individual basic blocks, we lift predictions by adding the weighted basic block predictions:

$$\text{lifted_pred}(\mathcal{K}) = \sum_{b \in \text{BBs}(\mathcal{K})} \text{occurences}(b) \times \text{pred}(b)$$

Our benchmarking harness works in three successive stages. It first extracts the basic blocks constituting a computation kernel, and instruments it to retrieve their respective occurrences in the original context. It then runs all the studied tools on each basic block, while also

running measures on the whole computation kernel. Finally, the block-level results are lifted to kernel-level results thanks to the occurrences previously measured.

4.4.1 Basic block extraction

Using the Capstone disassembler [QC], we split the assembly code at each control flow instruction (jump, call, return, …) and each jump site, as in Algorithm 1 from subsection 2.4.3.

To accurately obtain the occurrences of each basic block in the whole kernel's computation, we then instrument it with `gdb` by placing a break point at each basic block's first instruction in order to count the occurrences of each basic block between two calls to the `perf` counters [1]. While this instrumentation takes about 50 to 100× more time than a regular run, it can safely be run in parallel, as the performance results are discarded.

4.4.2 Throughput predictions and measures

The harness leverages a variety of tools: actual CPU measurement; the `BHive` basic block profiler [Che+19]; `llvm-mca` [SL], `uiCA` [AR22] and `IACA` [Intb], which leverage microarchitectural models to predict a block's throughput; `Ithemal` [MAC18], a machine learning model; and `Gus` [Gru19], a dynamic analyzer based on `QEMU` that works at the whole binary level.

The execution time of the full kernel is measured using Linux `perf` [Lin] CPU counters around the full computation kernel. The measure is repeated four times and the smallest is kept; this ensures that the cache is warm and compensates for context switching or other measurement artifacts. `Gus` instruments the whole function body. The other tools included all work at basic block level; these are run on each basic block of each benchmark.

We emphasize the importance, throughout the whole evaluation chain, to keep the exact same assembled binary. Indeed, recompiling the kernel from source *cannot* be assumed to produce the same assembly kernel. This is even more important in the presence of slight changes: for instance, inserting `IACA` markers at the C-level — as is intended — around the kernel *might* change the compiled kernel, if only for alignment regions. We argue that, in the case of `IACA` markers, the problem is even more critical, as those markers prevent a binary from being run by overwriting registers with arbitrary values. This forces a user to run and measure a version which is different from the analyzed one. In our harness, we circumvent this issue by adding markers directly at the assembly level, editing the already compiled version. Our `gdb` instrumentation procedure also respects this principle of single-compilation. As `QEMU` breaks the `perf` interface, we have to run `Gus` with a preloaded stub shared library to be able to instrument binaries containing calls to `perf`.

4.4.3 Prediction lifting and filtering

We finally lift single basic block predictions to a whole-kernel cycle prediction by summing the block-level results, weighted by the occurrences of the basic block in the original context (formula above). If an analyzer fails on one of the basic blocks of a benchmark, the whole benchmark is discarded for this analyzer.

In the presence of complex control flow, *eg.* with conditionals inside loops, our approach based on basic block occurrences is arguably less precise than an approach based on paths occurrences, as we have less information available — for instance, whether a branch is taken with a regular pattern, whether we have constraints on register values, etc. We however chose this block-based approach, as most throughput prediction tools work a basic block-level, and are thus readily available and can be directly plugged into our harness.

1. We assume the program under analysis to be deterministic.

Finally, we control the proportion of cache misses in the program's execution using `Cachegrind` [NS03] and `Gus`; programs that have more than 15 % of cache misses on a warm cache are not considered L1-resident and are discarded.

4.5 Experimental setup and evaluation

Running the harness described above provides us with 3500 benchmarks — after filtering out non-L1-resident benchmarks —, on which each throughput predictor is run. Before analyzing these results in Section 4.6, we evaluate the relevance of the methodology presented in Section 4.4 to make the tools' predictions comparable to baseline hardware counter measures.

4.5.1 Experimental environment

The experiments presented in this chapter, unless stated otherwise, were all realized on a Dell PowerEdge C6420 machine, from the *Dahu* cluster of Grid5000 in Grenoble [Bal+13]. The server is equipped with 192 GB of DDR4 SDRAM — only a small fraction of which was used — and two Intel Xeon Gold 6130 CPUs (x86-64, Skylake microarchitecture) with 16 cores each.

The experiments themselves were run inside a Docker environment based on Debian Bullseye. Care was taken to disable hyperthreading to improve measurements stability. For tools whose output is based on a direct measurement (`perf`, `BHive`), the benchmarks were run sequentially on a single core with no experiments on the other cores. No such care was taken for `Gus` as, although based on a dynamic run, its prediction is purely function of recorded program events and not of program measures. All other tools were run in parallel.

We use `llvm-mca v13.0.1`, `IACA v3.0-28-g1ba2cbb`, `BHive` at commit `5f1d500`, `uiCA` at commit `9cbbe93`, `Gus` at commit `87463c9`, `Ithemal` at commit `b3c39a8`.

4.5.2 Comparability of the results

We define the relative error of a time prediction C_{pred} (in cycles) with respect to a baseline C_{baseline} as

$$\text{err} = \frac{\left| C_{\text{pred}} - C_{\text{baseline}} \right|}{C_{\text{baseline}}}$$

We assess the comparability of the whole benchmark, measured with `perf`, to lifted block-based results by measuring the statistical distribution of the relative error of two series: the predictions made by `BHive`, and the series of the best block-based prediction for each benchmark.

We single out `BHive` as it is the only tool able to *measure* — instead of predicting — an isolated basic block's timing. This, however, is not sufficient: as discussed later in Section 4.6.2, `BHive` is not able to yield a result for about 40 % of the benchmarks, and is subject to large errors in some cases. For this purpose, we also consider, for each benchmark, the best block-based prediction: we argue that if, for most benchmarks, at least one of these predictors is able to yield a satisfyingly accurate result, then the lifting methodology is sound in practice.

The result of this analysis is presented in Table 4.1 and in Figure 4.2. The results are in a range compatible with common results of the field, as seen *eg.* in [AR22] reporting Mean Absolute Percentage Error (MAPE, corresponding to the "Average" row) of about 10-15 % in many cases. While lifted `BHive`'s average error is driven high by large errors on certain benchmarks, investigated later in this article, its median error is still comparable to the errors of state-of-the-art tools. From this, we conclude that lifted cycle measures and predictions are consistent with whole-benchmark measures; and consequently, lifted predictions can reasonably be compared to one another.

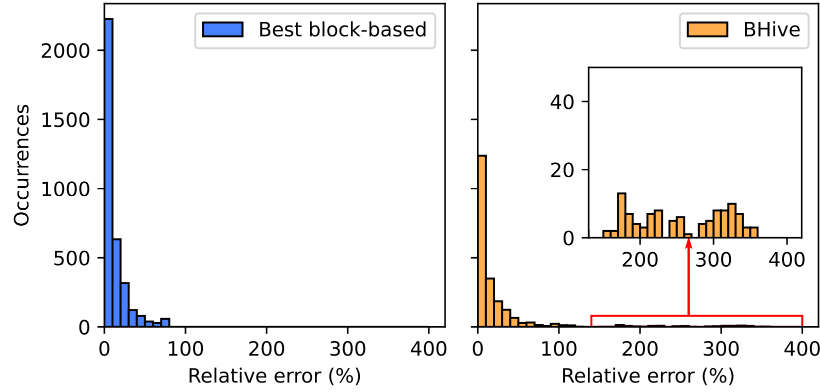


Figure 4.2 – Relative error distribution *wrt.* `perf`

	Best block-based	BHive
Datapoints	3500	2198
Errors	0	1302
	(0 %)	(37.20 %)
Average (%)	11.60	27.95
Median (%)	5.81	7.78
Q1 (%)	1.99	3.01
Q3 (%)	15.41	23.01

Table 4.1 – Relative error statistics *wrt.* `perf`

Polybench benchmark	**Frontend**			**Ports**			**Dependencies**		
	yes	**no**	**disagr.**	**yes**	**no**	**disagr.**	**yes**	**no**	**disagr.**
2mm	34	61	25.8 %	25	13	70.3 %	18	29	63.3 %
3mm	44	61	18.0 %	30	13	66.4 %	23	37	53.1 %
atax	13	72	41.0 %	25	17	70.8 %	23	30	63.2 %
bicg	19	59	45.8 %	25	25	65.3 %	21	37	59.7 %
doitgen	51	25	40.6 %	36	30	48.4 %	17	22	69.5 %
mvt	27	53	33.3 %	9	18	77.5 %	7	32	67.5 %
gemver	62	13	39.5 %	2	48	59.7 %	1	28	76.6 %
gesummv	16	69	41.0 %	17	23	72.2 %	24	28	63.9 %
syr2k	51	37	38.9 %	8	42	65.3 %	19	34	63.2 %
trmm	69	27	25.0 %	16	30	64.1 %	15	30	64.8 %
symm	0	121	11.0 %	5	20	81.6 %	9	5	89.7 %
syrk	54	46	30.6 %	12	42	62.5 %	20	48	52.8 %
gemm	42	41	42.4 %	30	41	50.7 %	16	57	49.3 %
gramschmidt	48	52	21.9 %	16	20	71.9 %	24	39	50.8 %
cholesky	24	72	33.3 %	0	19	86.8 %	5	14	86.8 %
durbin	49	52	29.9 %	0	65	54.9 %	2	39	71.5 %
trisolv	53	84	4.9 %	6	22	80.6 %	4	16	86.1 %
jacobi-1d	18	78	33.3 %	66	9	47.9 %	0	13	91.0 %
heat-3d	32	8	72.2 %	26	0	81.9 %	0	0	100.0 %
seidel-2d	0	112	22.2 %	32	0	77.8 %	0	0	100.0 %
fdtd-2d	52	22	47.1 %	20	41	56.4 %	0	40	71.4 %
jacobi-2d	6	31	73.6 %	24	61	39.3 %	0	44	68.6 %
adi	12	76	21.4 %	40	0	64.3 %	0	0	100.0 %
correlation	18	36	51.8 %	19	30	56.2 %	23	45	39.3 %
covariance	39	36	37.5 %	4	34	68.3 %	19	53	40.0 %
floyd-warshall	74	16	29.7 %	16	24	68.8 %	20	8	78.1 %
Total	907	1360	35.2 %	509	687	65.8 %	310	728	70.3 %

Table 4.2 – Bottleneck reports from the studied tools

4.5.3 Relevance and representativity (bottleneck analysis)

The results provided by our harness are only relevant to evaluate the parts of the tools' models that are stressed by the benchmarks generated; it is hence critical that our benchmark generation procedure in Section 4.3 yields diverse results. This should be true by construction, as the various polyhedral compilation techniques used stress different parts of the microarchitecture.

To assess this, we study the generated benchmarks' bottlenecks, *ie.* architectural resources on which a release of pressure improves execution time. Note that a saturated resource is not necessarily a bottleneck: a code that uses *eg.* 100 % of the arithmetics units available for computations outside of the critical path, at a point where a chain of dependencies is blocking, will not run faster if the arithmetics operations are removed; hence, hardware counters alone are not sufficient to find bottlenecks.

However, some static analyzers report the bottlenecks they detect. To unify their results and keep things simple, we study three general kinds of bottlenecks.

— *Frontend:* the CPU's frontend is not able to issue micro-operations to the backend fast enough. `IACA` and `uiCA` are able to detect this.

— *Ports:* at least one of the backend ports has too much work; reducing its pressure would accelerate the computation. `llvm-mca`, `IACA` and `uiCA` are able to detect this.

— *Dependencies:* there is a chain of data dependencies slowing down the computation. `llvm-mca`, `IACA` and `uiCA` are able to detect this.

For each source benchmark from Polybench and each type of bottleneck, we report in Table 4.2 the number of derived benchmarks on which all the tools agree that the bottleneck is present or absent. We also report the proportion of cases in which the tools failed to agree. We analyze those results later in Section 4.6.3.

As we have no source of truth indicating whether a bottleneck is effectively present in a microbenchmark, we adopt a conservative approach, and consider only the subset of the microbenchmarks on which the tools agree on the status of all three resources; for those, we have a good confidence on the bottlenecks reported. Obviously, this approach is limited, because it excludes microbenchmarks that might be worth considering, and is most probably subject to selection bias.

Of the 3,500 microbenchmarks we have generated, 261 (7.5 %) are the subject of the above-mentioned consensus. This sample is made up of microbenchmarks generated from 21 benchmarks — *ie.* for 5 benchmarks, none of the derived microbenchmarks reached a consensus among the tools —, yielding a wide variety of calculations, including floating-point arithmetic, pointer arithmetic or Boolean arithmetic. Of these, 200 (76.6 %) are bottlenecked on the CPU front-end, 19 (7,3 %) on back-end ports, and 81 (31.0 %) on latency introduced by dependencies. As mentioned above, this distribution probably does not transcribe the distribution among the 3,500 original benchmarks, as the 261 were not uniformly sampled. However, we argue that, as all categories are represented in the sample, the initial hypothesis that the generated benchmarks are diverse and representative is confirmed — thanks to the transformations described in Section 4.3.

4.5.4 Carbon footprint

Generating and running the full suite of benchmarks required about 30h of continuous computation on a single machine. During the experiments, the power supply units reported a near-constant consumption of about 350W. The carbon intensity of the power grid in France, where the experiment was run, at the time of the experiments, was of about 29gCO_2eq/kWh [Ele].

The electricity consumed directly by the server thus amounts to about 10.50 kWh. Assuming a Power Usage Efficiency of 1.5, the total electricity consumption roughly amounts to 15.75 kWh, or about 450 gCO_2eq.

Bencher	Datapoints	Failures (Count)	(%)	MAPE (%)	Median (%)	Q1 (%)	Q3 (%)	K_τ	Time (CPU·h)
BHive	2198	1302	(37.20)	27.95	7.78	3.01	23.01	0.81	1.37
llvm-mca	3500	0	(0.00)	36.71	27.80	12.92	59.80	0.57	0.96
UiCA	3500	0	(0.00)	29.59	18.26	7.11	52.99	0.58	2.12
Ithemal	3500	0	(0.00)	57.04	48.70	22.92	75.69	0.39	0.38
Iaca	3500	0	(0.00)	30.23	18.51	7.13	57.18	0.59	1.31
Gus	3500	0	(0.00)	20.37	15.01	7.82	30.59	0.82	188.04

Table 4.3 – Statistical analysis of overall results

A carbon footprint estimate of the machine's manufacture itself was conducted by the manufacturer [Del19]. Additionally accounting for the extra 160 GB of DDR4 SDRAM [Gup+22], the hardware manufacturing, transport and end-of-life is evaluated to 1,266 kgCO_2eq. In 2023, this computation cluster's usage rate was 35 %. Assuming 6 years of product life, 30h of usage represents about 2,050 gCO_2eq. The whole experiment thus amounts to 2.5 kgCO_2eq.

4.6 Results analysis

The raw complete output from our benchmarking harness — roughly speaking, a large table with, for each benchmark, a cycle measurement, cycle count for each throughput analyzer, the resulting relative error, and a synthesis of the bottlenecks reported by each tool — enables many analyses that, we believe, could be useful both to throughput analysis tool developers and users. Tool designers can draw insights on their tool's best strengths and weaknesses, and work towards improving them with a clearer vision. Users can gain a better understanding of which tool is more suited for each situation.

4.6.1 Throughput results

The error distribution of the relative errors, for each tool, is presented as a box plot in Figure 4.3. Statistical indicators are also given in Table 4.3. We also give, for each tool, its Kendall's tau indicator [Ken38] that we used earlier in chapter 2 and chapter 3.

These results are, overall, significantly worse than what each tool's article presents. We attribute this difference mostly to the specificities of Polybench: being composed of computation kernels, it intrinsically stresses the CPU more than basic blocks extracted out of *eg.* the Spec benchmark suite. This difference is clearly reflected in the experimental section of Palmed in chapter 2: the accuracy of most tools is worse on Polybench than on Spec, often by more than a factor of two.

As `BHive` and `Ithemal` do not support control flow instructions (*eg.* `jump` instructions), those had to be removed from the blocks before analysis. While none of these tools, apart from `Gus` — which is dynamic —, is able to account for branching costs, these two analyzers were also unable to account for the front- and backend cost of the control flow instructions themselves as well — corresponding to the TP_U mode introduced by `uiCA` [AR22], while others measure TP_L.

4.6.2 Understanding `BHive`'s results

The error distribution of `BHive` against `perf`, plotted right in Figure 4.2, puts forward irregularities in `BHive`'s results. Since `BHive` is based on measures — instead of predictions — through hardware counters, an excellent accuracy is expected. Its lack of support for control flow instructions can be held accountable for a portion of this accuracy drop; our lifting method, based on block occurrences instead of paths, can explain another portion. We also find that `BHive` fails to produce a result in about 40 % of the kernels explored — which means that, for

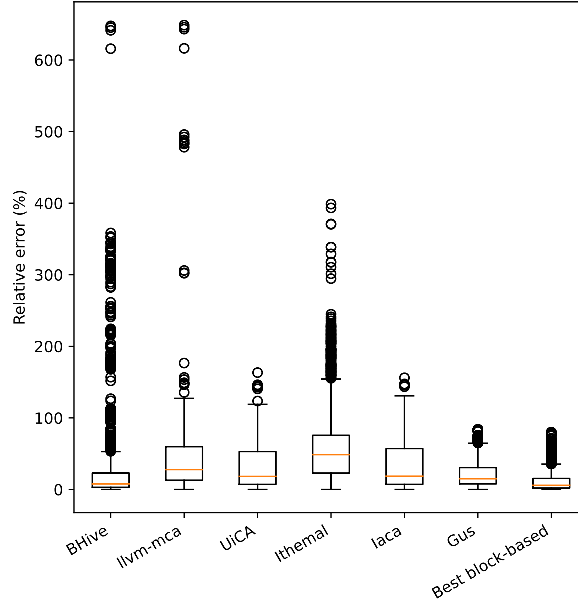


Figure 4.3 – Statistical distribution of relative errors

those cases, `BHive` failed to produce a result on at least one of the constituent basic blocks. In fact, this is due to the difficulties we mentioned in section 4.1 earlier, related to the need to reconstruct the context of each basic block *ex nihilo*.

The basis of `BHive`'s method is to run the code to be measured, unrolled a number of times depending on the code size, with all memory pages but the code unmapped. As the code tries to access memory, it will raise segfaults, caught by `BHive`'s harness, which allocates a single shared-memory page, filled with a repeated constant, that it will map wherever segfaults occur before restarting the program. The main causes of `BHive` failure are bad code behaviour (*eg.* control flow not reaching the exit point of the measure if a bad jump is inserted), too many segfaults to be handled, or a segfault that occurs even after mapping a page at the problematic address.

The registers are also initialized, at the beginning of the measurement, to the fixed constant `0x2324000`. We show through two examples that this initial value can be of crucial importance.

The following experiments are executed on an Intel(R) Xeon(R) Gold 6230R CPU (Cascade Lake), with hyperthreading disabled.

Imprecise analysis. We consider the following x86-64 kernel.

```
1    vmulsd (%rax), %xmm3, %xmm0
2    vmovsd %xmm0, (%r10)
```

Note here that the `vmulsd out, in1, in2` instruction is the scalar double-precision float multiplication of values from `in1` and `in2`, storing the result in `out`; while `vmovsd out, in` is a simple `mov` operation from `in` to `out` operating on double-precision floats in `%xmm` registers.

When executed with all the general purpose registers initialized to the default constant, `BHive` reports 9 cycles per iteration, since `%rax` and `%r10` hold the same value, inducing a read-after-write dependency between the two instructions. If, however, `BHive` is tweaked to initialize `%r10` to a value that aliases (*wrt.* physical addresses) with the value in `%rax`, *eg.* between `0x10000` and `0x10007` (inclusive), it reports 19 cycles per iteration instead; while a value between `0x10008` and `0x1009f` (inclusive) yields the expected 1 cycle — except for values in `0x10039-0x1003f` and `0x10079-0x1007f`, yielding 2 cycles as the store crosses a cache line boundary.

Tool	Ports		Dependencies	
llvm-mca	567	(24.6 %)	1032	(41.9 %)
uiCA	516	(22.4 %)	530	(21.5 %)
IACA	1221	(53.0 %)	900	(36.6 %)

Table 4.4 – Diverging bottleneck prediction per tool

In the same way, the value used to initialize the shared memory page can influence the results whenever it gets loaded into registers.

Failed analysis. Some memory accesses will always result in an error; for instance, on Linux, it is impossible to `mmap` at an address lower than `/proc/sys/vm/mmap_min_addr`, defaulting to `0x10000`. Thus, with equal initial values for all registers, the following kernel would fail, since the second operation attempts to load at address 0:

```
1    subq %r11, %r10
2    movq (%r10), %rax
```

Such errors can occur in more circumvoluted ways. The following x86-64 kernel, for instance, is extracted from a version of the `durbin` kernel[2].

```
1    vmovsd 0x10(%r8, %rcx), %xmm6
2    subl %eax, %esi
3    movslq %esi, %rsi
4    vfmadd231sd -8(%r9, %rsi, 8), \
5        %xmm6, %xmm0
```

Here, `BHive` fails to measure the kernel when run with the general purpose registers initialized to the default constant at the 2nd occurrence of the unrolled loop body, failing to recover from an error at the `vfmadd231sd` instruction with the `mmap` strategy. Indeed, after the first iteration the value in `%rsi` becomes null, then negative at the second iteration; thus, the second occurrence of the last instruction fetches at address `0xffffffff0a03ff8`, which is in kernel space. This microkernel can be benchmarked with BHive *eg.* by initializing `%rax` to 1.

Some other microkernels fail in a similar way when trying to access addresses that are not a virtual address in *canonical form* space for x86-64 with 48 bits virtual addresses, as defined in Section 3.3.7.1 of Intel's Software Developer's Manual [23c] and Section 5.3.1 of the AMD64 Architecture Programmer's Manual [23a]. Others still fail with accesses relative to the instruction pointer, as `BHive` read-protects the unrolled microkernel's instructions page.

4.6.3 Bottleneck prediction

We introduced in Section 4.5.3 earlier that some of the tools studied are also able to report suspected bottlenecks for the evaluated program, whose results are presented in Table 4.2. This feature might be even more useful than raw throughput predictions to the users of these tools willing to optimize their program, as they strongly hint towards what needs to be enhanced.

In the majority of the cases studied, the tools are not able to agree on the presence or absence of a type of bottleneck. Although it might seem that the tools are performing better on frontend bottleneck detection, it must be recalled that only two tools (versus three in the other cases) are reporting frontend bottlenecks, thus making it more likely for them to agree.

The Table 4.4, in turn, breaks down the cases on which three tools disagree into the number of times one tool makes a diverging prediction — *ie.* the tool predicts differently than the two others. In the case of ports, `IACA` is responsible for half of the divergences — which is not

2. `durbin.pocc.noopt.default.unroll8.MEDIUM.kernel21.s` in the full results

Bencher	Datapoints	Failures	(%)	MAPE	Median	Q1	Q3	K_τ
BHive	1365	1023	(42.84 %)	34.07 %	8.62 %	4.30 %	24.25 %	0.76
llvm-mca	2388	0	(0.00 %)	27.06 %	21.04 %	9.69 %	32.73 %	0.79
UiCA	2388	0	(0.00 %)	18.42 %	11.96 %	5.42 %	23.32 %	0.80
Ithemal	2388	0	(0.00 %)	62.66 %	53.84 %	24.12 %	81.95 %	0.40
Iaca	2388	0	(0.00 %)	17.55 %	12.17 %	4.64 %	22.35 %	0.82
Gus	2388	0	(0.00 %)	23.18 %	20.23 %	8.78 %	32.73 %	0.83

Table 4.5 – Statistical analysis of overall results, without latency bound through memory-carried dependencies rows

sufficient to conclude that the prediction of the other tools is correct. In the case of dependencies, however, there is no clear outlier, even though `uiCA` seems to fare better than others.

In no case one tool seems to be responsible for the vast majority of disagreements, which could hint towards it failing to predict correctly this bottleneck. In the absence of a source of truth indicating whether a bottleneck is effectively present, and with no clear-cut result for (a subset of) tool predictions, we cannot conclude on the quality of the predictions from each tool for each kind of bottleneck.

4.6.4 Impact of dependency-boundness

An overview of the full results table hints towards two main tendencies: on a significant number of rows, the static tools — thus leaving `Gus` and `BHive` apart —, excepted `Ithemal`, often yield comparatively bad throughput predictions *together*; and many of these rows are those using the `O1` and `O1autovect` compilation setting (`gcc` with `-O1`, plus vectorisation options for the latter).

To confirm the first observation, we look at the 30 % worst benchmarks — in terms of MAPE relative to `perf` — for `llvm-mca`, `uiCA` and `IACA` — yielding 1050 rows each. All of these share 869 rows (82.8 %), which we call *jointly bad rows*.

Among these 869 jointly bad rows, we further find that respectively 342 (39.4 %) and 337 (38.8 %) are compiled using the `O1` and `O1autovect`, totalling to 679 (78.1 %) of `O1`-based rows, against 129 (14.8 %) for `default` and 61 (7.0 %) for `O3nosimd`. This result is significant enough to be used as a hint to investigate the issue.

Insofar as our approach maintains a strong link between the basic blocks studied and the source codes from which they are extracted, it is possible to identify the high-level characteristics of the concerned microbenchmarks. In the overwhelming majority (97.5 %) of those jointly bad rows, the tools predicted fewer cycles than measured, meaning that a bottleneck is either missed or underestimated. Manual investigation of a few simple benchmarks (no polyhedral transformation applied, `O1` mode, not unrolled) further hints towards dependencies: for instance, the `gemver` benchmark, which is *not* among the badly predicted benchmarks, has this kernel:

```
1 for(c3)
2     A[c1][c3] += u1[c1] * v1[c3]
3                + u2[c1] * v2[c3];
```

while the `atax` benchmark, which is among the badly predicted ones, has this kernel:

```
1 for(c3)
2     tmp[c1] += A[c1][c3] * x[c3];
```

The first one exhibits no obvious dependency-boundness, while the second, accumulating on `tmp[c1]` (independent of the iteration variable) lacks in instruction-level parallelism. Among the simple benchmarks (as described above), 8 are in the badly predicted list, all of which exhibit a read-after-write data dependency to the preceding iteration.

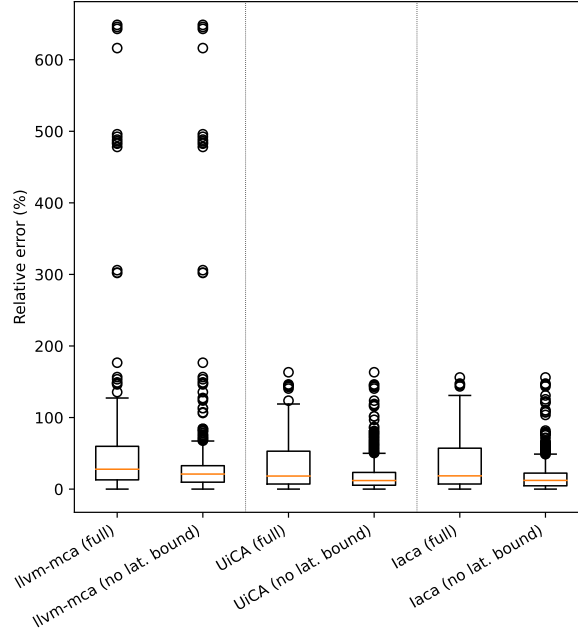


Figure 4.4 – Statistical distribution of relative errors, with and without pruning latency bound through memory-carried dependencies rows

Looking at the assembly code generated for those in `O1` modes, it appears that the dependencies exhibited at the C level are compiled to *memory-carried* dependencies: the read-after-write happens for a given memory address, instead of for a register. This kind of dependency, prone to aliasing and dependent on the values of the registers, is hard to infer for a static tool and is not supported by the analyzers under scrutiny in the general case; it could thus reasonably explain the results observed.

There is no easy way, however, to know for certain which of the 3500 benchmarks are latency bound: no hardware counter reports this. We investigate this further using `Gus`'s sensitivity analysis: in complement of the "normal" throughput estimation of `Gus`, we run it a second time, disabling the accounting for latency through memory dependencies. By construction, this second measurement should be either very close to the first one, or significantly below. We then assume a benchmark to be latency bound due to memory-carried dependencies when it is at least 40 % faster when this latency is disabled; there are 1112 (31.8 %) such benchmarks.

Of the 869 jointly bad rows, 745 (85.7 %) are declared latency bound through memory-carried dependencies by `Gus`. We conclude that the main reason for these jointly badly predicted benchmarks is that the predictors under scrutiny failed to correctly detect these dependencies.

In Section 4.6.1, we presented in Figure 4.3 and Table 4.3 general statistics on the tools on the full set of benchmarks. We now remove the 1112 benchmarks flagged as latency bound through memory-carried dependencies by `Gus` from the dataset, and present in Figure 4.4 a comparative box plot for the tools under scrutiny. We also present in Table 4.5 the same statistics on this pruned dataset. While the results for `llvm-mca`, `uiCA` and `IACA` globally improved significantly, the most noticeable improvements are the reduced spread of the results and the Kendall's τ correlation coefficient's increase.

From this, we argue that detecting memory-carried dependencies is a weak point in current state-of-the-art static analyzers, and that their results could be significantly more accurate if improvements are made in this direction.

Conclusion and future works

In this chapter, we have presented a fully-tooled approach that enables:

— the generation of a wide variety of microbenchmarks, reflecting both the expertise contained in an initial benchmark suite, and the diversity of code transformations allowing to stress different aspects of a performance model — or even a measurement environment, *eg.* `BHive`; and

— the comparability of various measurements and analyses applied to each of these microbenchmarks.

Thanks to this tooling, we were able to show the limits and strengths of various performance models in relation to the expertise contained in the Polybench suite. We discuss throughput results in Section 4.6.1 and bottleneck prediction in Section 4.6.3.

We were also able to demonstrate the difficulties of reasoning at the level of a basic block isolated from its context. We specifically study those difficulties in the case of `BHive` in Section 4.6.2. Indeed, the actual values — both from registers and memory — involved in a basic block's computation are constitutive not only of its functional properties (*ie.* the result of the calculation), but also of some of its non-functional properties (*eg.* latency, throughput).

We were also able to show in Section 4.6.4 that state-of-the-art static analyzers struggle to account for memory-carried dependencies; a weakness significantly impacting their overall results on our benchmarks. We believe that detecting and accounting for these dependencies is an important topic — which we will tackle in the following chapter.

Moreover, we present this work in the form of a modular software package, each component of which exposes numerous adjustable parameters. These components can also be replaced by others fulfilling the same abstract function: another initial benchmark suite in place of Polybench, other loop nest optimizers in place of PLUTO and PoCC, other code analyzers, and so on. This software modularity reflects the fact that our contribution is about interfacing and communication between distinct issues.

Furthermore, we believe that the contributions we made in the course of this work may eventually be used to face different, yet neighbouring issues. These perspectives can also be seen as future works:

Program optimization. The whole program processing we have designed can be used not only to evaluate the performance model underlying a static analyzer, but also to guide program optimization itself. In such a perspective, we would generate different versions of the same program using the transformations discussed in Section 4.3 and colored blue in Figure 4.1. These different versions would then feed the execution and measurement environment outlined in Section 4.4 and colored orange in Figure 4.1. Indeed, thanks to our previous work, we know that the results of these comparable analyses and measurements would make it possible to identify which version is the most efficient, and even to reconstruct information indicating why (which bottlenecks, etc.).

However, this approach would require that these different versions of the same program are functionally equivalent, *ie.* that they compute the same result from the same inputs; yet we saw in Section 4.4 that, as it stands, the transformations we apply are not concerned with preserving the semantics of the input codes. To recover this semantic preservation property, abandoning the kernelification pass we have presented suffices; this however would require to control L1-residence otherwise.

Dataset building. Our microbenchmarks generation phase outputs a large, diverse and representative dataset of microkernels. In addition to our harness, we believe that such a dataset could be used to improve existing data-dependant solutions.

Inductive methods, for instance in `AnICA`, strive to preserve the properties of a basic block through successive abstractions of the instructions it contains, so as to draw the most general conclusions possible from a particular experiment. Currently, `AnICA` starts off from randomly generated basic blocks. This approach guarantees a certain variety, and avoids over-specialization, which would prevent it from finding interesting cases too far from an initial dataset. However, it may well lead to the sample under consideration being systematically outside the relevant area of the search space — *ie.* having no relation to real-life programs or those in the user's field.

On the other hand, machine learning methods based on neural networks, for instance in `Ithemal`, seek to correlate the result of a function with the characteristics of its input — in this case to correlate a throughput prediction with the instructions making up a basic block — by backpropagating the gradient of a cost function. In the case of `Ithemal`, it is trained on benchmarks originating from a data suite. As opposed to random generation, this approach offers representative samples, but comes with a risk of lack of variety and over-specialization.

Comparatively, our microbenchmark generation method is natively meant to produce a representative, varied and large dataset. We believe that enriching the dataset of the above-mentioned methods with our benchmarks might extend their results and reach.

Chapter 5

Static extraction of memory-carried dependencies

In the previous chapter, our major finding was that, in the current state of the art, code analyzers deal poorly with memory-carried dependencies. We found this flaw to be responsible, in our dataset, for a roughly 1.5× increase in MAPE, and up to 2.6× on the third quartile of error.

The large impact of dependencies on the final runtime of a kernel is, in reality, not very surprising. In chapters 2 and 3, we did not consider latency; hence, the only impact of an instruction was its throughput, each instruction being issued as soon as possible. Dependencies, however, force the processor to wait for some instructions' results before issuing some others; the *latency* of an instruction becomes a critical factor.

On Skylake, for instance, the instruction `add %rax, %rbx` has a latency of one full cycle. Thus, the kernel

```
1 add %rax, %rbx
2 add %rbx, %rcx
```

executes, in steady state, in half a cycle without accounting for the dependency; yet these two instructions in isolation would take 1 1/4 cycles when accounting for the dependency. Some instructions still are more extreme; for instance, the `vfmadd*pd %ymm0, %ymm1, %ymm2` family of instructions have a latency of four full cycles, while without dependencies, two can be issued every cycle.

In the previous chapter, we also presented `Gus`, a dynamic code analyzer based on `QEMU`, which we found to be very effective to detect memory-carried dependencies and the slowdown they incur on the whole program. However, this solution results in a runtime increase of about two orders of magnitude, which may not be acceptable in many use cases.

In this chapter, we instead present `staticdeps`, a fully static analyzer able to detect memory-carried dependencies in many cases. We evaluate it by providing `uiCA` with its analysis of dependencies, bringing it on-par with `Gus` on the full, non-pruned dataset of the previous chapter.

5.1 Types of dependencies

A dependency, in the most general sense, can be seen as an interaction between two instructions stemming from shared data. This definition is willingly broad as, depending on the circumstances, the CPU implementation, …, some categories of dependencies must be taken into account, while some may be ignored.

Read-write categories. The first distinction that can be made between dependencies, and the one that is most often made, is whether the data through which the dependency is created is read or written. They can be broken down into four categories:

— read-after-write (RaW);
— write-after-write (WaW);
— write-after-read (WaR);
— read-after-read (RaR).

For instance, in the kernel presented in the introduction of this chapter, the first instruction (`add %rax, %rbx`) reads its first operand, the register `%rax`, and both reads and writes its second operand `%rbx`. The second `add` has the same behaviour. Thus, as `%rbx` is written at line 1, and read at line 2, there is a read-after-write dependency between the two.

Most of the time, *dependency* is actually used to mean *read-after-write dependency*, sometimes called "flow dependency". However, depending on the actual hardware implementation of the architecture, other kinds of dependencies might induce a latency. While a read-after-read dependency will not induce a latency in the vast majority of architectures, a write-after-read could prevent instructions to be re-ordered in a way that the writing instruction commits its result before the reading instruction uses the previously stored value. In most modern CPUs, the processor actually has more physical registers than what is exposed to the user through the ISA; a renaming phrase will allocate those registers to avoid the effects of WaR and WaW dependencies as much as possible.

For the present chapter, *we only consider read-after-write dependencies*; however, all the techniques we present are applicable to other types of dependencies if the considered architecture requires to take them into account.

Dependency medium. In the example above, we only introduced dependencies induced through registers, or *register-carried*. There are, however, other channels.

As we saw in the introduction to this chapter, as well as in the previous chapter, dependencies can also be *memory-carried*, in more or less straightforward ways, such as in the examples from Listing 2, where the last line always depends on the first.

```
1 add %rax, (%rbx)
2 add (%rbx), %rcx
```

```
1 add %rax, (%rbx)
2 add $8, %rbx
3 add -8(%rbx), %rcx
```

```
1 add %rax, (%rbx)
2 lea 16(%rbx), %r10
3 add -16(%r10), %rcx
```

Listing 2 – Examples of memory-carried dependencies.

Some dependencies are also *flag-carried*. These are very akin to register-carried dependency, but are not directly visible in the instruction. For instance, a subtract operation may set flags indicating whether the result is zero, and a subsequent jump may use this flag to chose whether the branch is taken or not.

Depending on the architecture, other channels may still exist.

In this chapter, we focus on register-carried and memory-carried dependencies, with a large emphasis on memory-carried dependencies.

Presence of loops. The previous examples were all pieces of *straight-line code* in which a dependency arose. However, many dependencies are actually *loop-carried*, such as those in Listing 3. In Listing 3a, line 2 depends on the previous iteration's line 2 as `%r10` is read, then written back. In Listing 3b, line 3 depends on line 2 of the same iteration; but line 2 alsp depends on line 3 two iterations ago by reading `-16(%rbx, %r10)`.

```
loop:
    add (%rax), %r10
    add $8, %rax
    jmp loop
```

(a) Compute the sum of array `A`'s terms in `%r10`. `%rax` points to `A`.

```
loop:
    mov -16(%rbx, %r10), (%rbx, %r10)
    add (%rax, %r10), (%rbx, %r10)
    add $8, %r10
    jmp loop
```

(b) Compute `B[i] = A[i] + B[i-2]`. `%rax` points to `A`, `%rbx` points to `B`.

Listing 3 – Examples of loop-carried dependencies.

5.2 A baseline: dynamic dependencies detection with `valgrind`

As we already saw, a dynamic analyzer, such as `Gus`, has direct access to the actual data dependencies occurring throughout an execution. While such analyzers are often too slow to use in practice, they can be used as a baseline to evaluate static alternatives.

As it is a complex tool performing a wide range of analyses, `Gus` is, however, unnecessarily complex to simply serve as a baseline. For the same reason, it is also impractically slower than a simple dynamic analysis. For this reason, we implement `depsim`, a dynamic analysis tool based on top of `valgrind`, whose goal is to report dependencies encountered at runtime.

5.2.1 Valgrind

Most low-level developers and computer scientists know `valgrind` [NS03] as a memory analysis tool, reporting bad memory accesses, memory leaks and the like. However, this is only a small part of `valgrind` — the `memcheck` tool. The whole program is actually a binary instrumentation framework, upon which the famous `memcheck` is built.

`valgrind` supports a wide variety of platforms, including x86-64 and ARM. However, at the time of the writing, it supports AVX2, but does not yet support AVX-512 on x86-64 [Val23] — there is, however, ongoing work in that direction [Vol22]. While its documentation is seemingly sparse and not frequently maintained, its code is well-commented and readable, with interfaces cleanly exposed; thus, once one gets a good enough comprehension of how the project's code is structured, this lack of documentation is not problematic. `valgrind` is also a free and open-source software, distributed under the GNU GPLv2 license.

In order to instrument a binary file, `valgrind` first lifts the original assembly code to an intermediate representation. The instrumentation tool is then able to modify this intermediate representation before passing it back to Valgrind, which will re-compile it to a native binary before running it.

While this intermediate representation, called `VEX`, is convenient to instrument a binary, it may further be used as a way to obtain *semantics* for some assembly code, independently of the Valgrind framework.

5.2.2 Depsim

The tool we wrote to extract runtime-gathered dependencies, `depsim`, is able to extract dependencies through both registers, memory and temporary variables — in its intermediate representation, Valgrind keeps some values assigned to temporary variables in static single-assignment (SSA) form. It however supports a flag to detect only memory-carried dependencies, as this will be useful to evaluate our static algorithm later.

As a dynamic tool, the distinction between straight-line code and loop-carried dependencies is irrelevant, as the analysis follows the actual program flow.

In order to track dependencies, each basic block of the program is instrumented. Dependencies are stored as a hash table and represented as a pair of source and destination program counter; they are mapped to a number of encountered occurrences.

Dependencies through temporaries are, by construction, resident to a single basic block — they are thus statically detected at instrumentation time. At runtime, the occurrence count of those dependencies is updated whenever the basic block is executed.

For both register- and memory-carried dependencies, each write is instrumented by adding a runtime write to a *shadow* register file or memory, noting that the written register or memory address was last written at the current program counter. Each read, in turn, is instrumented by adding a fetch to this shadow register file or memory, retrieving the last program counter at which this location was written to; the dependency count between this program counter and the current program counter is then incremented.

In practice, the shadow register file is simply implemented as an array holding, for each register id, the last program counter that wrote at this location. The shadow memory is instead implemented as a hash table.

At the end of the run, all the dependencies retrieved are reported. Care is taken to translate back the runtime program counters to addresses in the original ELF files, using the running process' memory map.

Dependencies are mostly relevant if their source and destination are close enough to be computationally meaningful. To this end, we also introduce in `depsim` a notion of *dependency lifetime*. As we do not have access without a heavy runtime slowdown to elapsed cycles in `valgrind`, we define a *timestamp* as the number of instructions executed since beginning of the program's execution; we increment this count at each branch instruction to avoid excessive instrumentation slowdown.

We further annotate every write to the shadow memory with the timestamp at which it occurred. Whenever a dependency should be added, we first check that the dependency has not expired — that is, that it is not older than a given threshold. This threshold is tunable for each run — and may be set to infinity to keep every dependency.

5.3 Static dependencies detection

Depending on their type, some dependencies are significantly harder to statically detect than others.

Register-carried dependencies, when in straight-line code, can be detected by keeping track of which instruction last wrote each register in a *shadow register file*. This is most often supported by code analyzers — for instance, `llvm-mca` and `uiCA` support it.

Loop-carried dependencies can, to some extent, be detected the same way. As the basic block is always assumed to be the body of an infinite loop, a straight-line analysis can be performed on a duplicated kernel. This strategy is *eg.* adopted by `Osaca` [Lau+19, §II.D]. When dealing only with register accesses, this strategy is always sufficient: as each iteration always executes the same basic block, it is not possible for an instruction to depend on another instruction two iterations earlier or more.

Memory-carried dependencies, however, are significantly harder to tackle. While basic heuristics can handle some simple cases, in the general case two main difficulties arise:

(i) pointers may *alias*, *ie.* point to the same address or array; for instance, if `%rax` points to an array, it may be that `%rbx` points to `%rax` + 8, making the detection of such a dependency difficult;

(ii) arbitrary arithmetic operations may be performed on pointers, possibly through diverting paths: *eg.* it might be necessary to detect that `%rax` $+ 16 << 2$ is identical to `%rax` $+ 128/2$;

this requires semantics for assembly instructions and tracking formal expressions across register values — and possibly even memory.

Tracking memory-carried dependencies is, to the best of our knowledge, not done in code analyzers, as our results in chapter 4 suggests.

While the strategy previously used for register-carried dependencies is sufficient to detect loop-carried dependencies from one occurrence to the next one, it is not sufficient at all times when the dependencies tracked are memory-carried. For instance, in the second example from Listing 3, an instruction depends on another two iterations ago.

Dependencies can reach arbitrarily old iterations of a loop: in this example, `-8192(%rbx, %r10)` may be used to reach 1 024 iterations back. However, while far-reaching dependencies may *exist*, they are not necessarily *relevant* from a performance analysis point of view. Indeed, if an instruction i_2 depends on a result previously produced by an instruction i_1, this dependency is only relevant if it is possible that i_1 is not yet completed when i_2 is considered for issuing — else, the result is already produced, and i_2 needs never wait to execute.

The reorder buffer (ROB) of a CPU can be modelled as a sliding window of fixed size over μOPs. In particular, if a μOP μ_1 is not yet retired, the ROB may not contain μOPs more than the ROB's size ahead of μ_1. This is in particular also true for instructions, as the vast majority of instructions decode to at least one μOP [1]. We formalize this in subsection 5.3.1 below.

A possible solution to detect loop-carried dependencies in a kernel $\mathcal{K}$ is thus to unroll it until it contains about $|\text{ROB}| + |\mathcal{K}|$. This ensures that every instruction in the last kernel can find dependencies reaching up to $|\text{ROB}|$ back.

On Intel CPUs, the reorder buffer size contained 224 μOPs on Skylake (2015), or 512 μOPs on Golden Cove (2021) [Wik21]. These sizes are small enough to reasonably use this solution without excessive slowdown.

5.3.1 Far-reaching dependencies do not impact performance

Definition (*Distance between instructions*)

Let $(I_p)_{0 \leq p < n}$ be a trace of executed instructions. For $p < p'$, distance $(I_p, I_{p'})$ is the overall number of decoded μOPs for the subtrace $(I_r)_{p < r < p'}$.

Theorem (*Long distance dependencies*)

Given a kernel $\mathcal{K}$, there exists $R \in \mathbb{N}$, only dependent of microarchitectural parameters, such that the presence or absence of a dependency between two instructions that are separated by at least $R - 1$ other μOPs has no impact on the performance of this kernel.

More formally, let $\mathcal{K}$ be a kernel of n instructions. Let $(I_p)_{p \in \mathbb{N}}$ be the trace of $\mathcal{K}$'s instructions executed in a loop. For any $p, q \in \mathbb{N}$ such that distance $(I_p, I_q) \geq R - 1$, $\overline{\mathcal{K}}$ is invariant in the presence or absence of a dependency between the pairs of instructions $(I_{p+kn}, I_{q+kn})_{k \in \mathbb{N}}$.

To prove this assertion, we require a few postulates to model the functioning of a CPU and, in particular, how μOPs transit in (decoded) and out (retired) the reorder buffer.

1. Some `mov` instructions from register to register may, for instance, only have an impact on the renamer; no μOPs are dispatched to the backend.

Postulate (*Reorder buffer as a circular buffer*)

The reorder buffer is a circular buffer of size $R \in \mathbb{N}^+$. It contains only decoded μOPs. Let us denote i_d the μOP at position d in the reorder buffer. Assume i_d just got decoded.

As the buffer is a circular FIFO, we have that for every q and q' in $[0, R)$:

$$(q - d - 1)\%R < (q' - d - 1)\%R \iff i_q \text{ was decoded before } i_{q'}$$

If a μOP has not been retired yet (issued and executed), it cannot be replaced in the ROB by any freshly decoded instruction. In other words, every non-retired decoded μOP — also called *in-flight* — remains in the reorder buffer. This is possible thanks to the notion of *full reorder buffer*:

Postulate (*Full reorder buffer*)

Let us denote by i_d the μOP that just got decoded. The reorder buffer is said to be *full* if for $q = (d + 1)\%R$, μOP i_q is not retired yet.

If the reorder buffer is full, then instruction decoding is stalled.

Let $(I_p)_{0 \leq p < n}$ be a trace of executed instructions. Each of these instructions are iteratively decoded, issued, and retired. We will also denote by $(i_q)_{0 \leq q < m}$ the trace of decoded μOPs. To prove the theorem above, we need to state that any two in-flight μOPs are distant of at most R μOPs.

For any instruction I_p, we denote as Q_p the range of indices such that $(i_q)_{q \in Q_p}$ are the μOPs obtained from the decoding of I_p.

Note that in practice, it is possible that we do not have $\bigcup_p Q_p = [0, n)$, as *eg.* branch mispredictions may introduce unwanted μOPs in the pipeline. However, as the worst case for the lemma below occurs when no such “spurious” μOPs are present, we may safely ignore such occurrences.

Lemma (*Distance of in-flight μOPs*)

For any pair of instructions $(I_p, I_{p'})$, and two corresponding μOPs, $(i_q, i_{q'})$ such that $q \in Q_p, q' \in Q_{p'}$,

$$\text{inflight}(i_q) \wedge \text{inflight}(i_{q'}) \implies \text{distance}\left(I_p, I_{p'}\right) < R - 1$$

Proof. The sets (Q_p) are disjoint pairwise, and for any pair of instructions $(I_p, I_{p'})$, and any two corresponding μOPs, $(i_q, i_{q'})$ such that $q \in Q_p$, $q' \in Q_{p'}$, $p < p' \implies q < q'$.

Thus, $\text{distance}\left(I_p, I_{p'}\right) < |q' - q|$.

Observe that at any time, the content of the ROB can be seen as a window of length at most R over $(i_q)_{0 \leq q < m}$. Consequently, if both i_q and $i_{q'}$ are in-flight then $|q' - q| < R$, and thus $\text{distance}\left(I_p, I_{p'}\right) < R - 1$. □

Postulate (*Issue delay*)

Reasons why the issue of a μOP i is delayed can be:

1. i is not yet in the reorder buffer
2. i depends on μOP i' which is not retired yet
3. ports on which i can be mapped are all occupied

Proof of Long distance dependencies theorem. The theorem above is now a direct consequence of the previous observations. Let us consider two μOPs, i and i', respectively introduced by instructions I_p and I_q. Assume a delayed issue for μOP i where the unique cause is a dependence from μOP i', that is:

1. i is already in the reorder buffer
2. i depends on μOP i' which is not retired yet
3. at least one port on which i can be mapped is available

Since i' is not retired yet and i' is "before" i, i' is still in the reorder buffer, *ie.* both i and i' are in the reorder buffer.

By the previous lemma, we have distance $(I_p, I_q) < R - 1$.

By contrapositive, for any two instructions I_a, I_b such that distance $(I_a, I_b) \geq R - 1$, no μOP of I_b may have its execution delayed by a dependency between I_a and I_b. □

Remark

What we stated earlier is a direct consequence of this theorem: to detect meaningful dependencies over a kernel $\mathcal{K}$, it suffices to analyze the kernel unrolled enough times to obtain a sequence of $R + |\mathcal{K}|$ instructions, as this yields a sequence where every instruction from the last occurrence of $\mathcal{K}$ is preceded by at least $R - 1$ instructions.

5.4 Staticdeps

The static analyzer we present, `staticdeps`, only aims to tackle the difficulty (ii) mentioned in section 5.3: tracking dependencies across arbitrarily complex pointer arithmetic.

To do so, `staticdeps` works at the basic-block level, unrolled enough times to fill the reorder buffer as detailed above; this way, arbitrarily long-reaching relevant loop-carried dependencies can be detected.

This problem could be solved using symbolic calculus algorithms. However, those algorithms are not straightforward to implement, and the equality test between two arbitrary expressions can be costly.

5.4.1 The `staticdeps` heuristic

Instead, we use a heuristic based on random values. We consider the set $\mathcal{R} = \{0, 1, \ldots, 2^{64} - 1\}$ of values representable by a 64-bits unsigned integer; we extend this set to $\bar{\mathcal{R}} = \mathcal{R} \cup \{\bot\}$, where $\bot$ denotes an invalid value. We then proceed as previously for register-carried dependencies, applying the following principles.

— Whenever an unknown value is read, either from a register or from memory, generate a fresh value from $\mathcal{R}$, uniformly sampled at random. This value is saved to a shadow register file or memory, and will be used again the next time this same data is accessed.

```
function STATICDEPS(basic_block)
    deps ← ∅
    insn_count, cur_iter, cur_instruction ← 0
    shadow_memory, shadow_registers, last_wrote_at ← ∅_map
    function FRESH
        return random uint64_t value
    function READ_MEMORY(address)
        Assert address ≠ ⊥
        if address ∉ shadow_memory then
            shadow_memory[address] ← FRESH
        return shadow_memory[address]
    function READ_REGISTER(register)
        ...                                                   ▷ Likewise
    function EXPR_VALUE(expr)
        if expr == Register(reg) then
            return READ_REGISTER(reg)
        else if expr == Memory(addr_expr) then
            addr ← EXPR_VALUE(expr)
            if addr ∈ last_wrote_at then
                deps ← deps ∪ (last_wrote_at[addr] → (cur_iter, cur_instruction))
            return READ_MEMORY(addr)
        else if expr == IntegerArithmeticOp(operator, op1, ..., opN) then
            if EXPR_VALUE(op_i) == ⊥ for any i then
                return ⊥
            return semantics(operator)(                       ▷ Provided by Valgrind's Vex
              EXPR_VALUE(op1), ..., EXPR_VALUE(opN))
        else return ⊥
    function ITER_STATEMENT(statement)
        lhs, rhs ← statement
        if lhs == Register(reg) then
            shadow_register[reg] ← EXPR_VALUE(rhs)
        else if lhs == Memory(addr_expr) then
            addr ← EXPR_VALUE(addr_expr)
            last_wrote_at[addr] ← (cur_iter, cur_instruction)
            shadow_memory[addr] ← EXPR_VALUE(rhs)
        else if ... then                                      ▷ Etc.
    while insn_count <= |ROB| + |basic_block| do
        cur_instruction ← 0
        for statement ∈ basic_block do
            ITER_STATEMENT(statement)
            if statement is last statement of an instruction then
                                       ▷ An instruction can be composed of multiple statements
                cur_instruction ← cur_instruction + 1
        cur_iter ← cur_iter + 1
    return deps
```

Algorithm 2 – The `staticdeps` algorithm

— Whenever an integer arithmetic operation is encountered, compute the result of the operation and save the result to the shadow register file or memory.
— Whenever another kind of operation, or an operation that is unsupported, is encountered, save the destination operand as $\perp$; this operation is assumed to not be valid pointer arithmetic. Operations on $\perp$ always yield $\perp$ as a result.
— Whenever writing to a memory location, compute the written address using the above principles, and proceed as with a dynamic analysis, keeping track of the instruction that last wrote to a memory address.
— Whenever reading from a memory location, compute the read address using the above principles, and generate a dependency from the current instruction to the instruction that last wrote to this address (if known).

5.4.2 Practical implementation

We implement `staticdeps` in Python, using `pyelftools` and the `capstone` disassembler — which we already introduced in section 2.4 — to extract and disassemble the targeted basic block. The semantics needed to compute encountered operations are obtained by lifting the kernel's assembly to `valgrind`'s `VEX` intermediary representation.

The implementation of the heuristic detailed above provides us with a raw list of dependencies across iterations of the considered basic block. We then "re-roll" the unrolled kernel by transcribing each dependency to a triplet (`source_insn`, `dest_insn`, Δk), where the first two elements are the source and destination instruction of the dependency *in the original, non-unrolled kernel*, and Δk is the number of iterations of the kernel between the source and destination instruction of the dependency.

We detail our `staticdeps` algorithm in psudocode in Algorithm 2.

Finally, we filter out spurious dependencies: each dependency found should occur for each kernel iteration i at which $i + \Delta k$ is within bounds. If the dependency is found for less than 80 % of those iterations, the dependency is declared spurious and is dropped.

5.4.3 Limitations

In chapter 4, we argued that one of the shortcomings that most crippled state-of-the-art tools was that analyses were conducted out-of-context, considering only the basic block at hand. This analysis is also true for `staticdeps`, as it is still focused on a single basic block in isolation; in particular, any aliasing that stems from outside of the analyzed basic block is not visible to `staticdeps`.

Work towards a broader analysis range, *eg.* at the scale of a function, or at least initializing values with gathered assertions — maybe based on abstract interpretation techniques — could be beneficial to the quality of dependencies detections.

Given how `staticdeps`'s heuristic is based on randomness, it may yield false positives: two registers could theoretically be assigned the same value sampled at random, making them aliasing addresses. This is, however, very improbable, as values are sampled from a set of cardinality 2^{64}. If necessary, the error can be reduced by amplification: running multiple times the algorithm on different randomness seeds reduces the error exponentially.

Conversely, `staticdeps` should not present false negatives due to randomness. Dependencies may go undetected, *eg.* because of out-of-scope aliasing or unsupported operations. However, no dependency that falls into the scope of `depsim`'s analysis should be missed because of random initialisations.

5.5 Evaluation

We evaluate the relevance of `staticdeps` results in two ways: first, we compare the detected dependencies to those extracted at runtime by `depsim`, to evaluate the proportion of dependencies actually detected. Then, we evaluate the relevance of our static analysis from a performance debugging point of view, by enriching `uiCA`'s model with `staticdeps` and assessing, using `CesASMe`, the benefits brought to the model.

We finally evaluate our claim that using a static model instead of a dynamic analysis, such as `Gus`, makes `staticdeps` yield a result in a reasonable amount of time.

5.5.1 Comparison to `depsim` results

The `staticdeps`'s model contribution largely resides in its ability to track memory-carried dependencies, including loop-carried ones. We thus focus on evaluating this aspect, and restrict both `depsim` and `staticdeps` to memory-carried dependencies.

We use the binaries produced by `CesASMe` as a dataset, as we already assessed its relevance and contains enough benchmarks to be statistically meaningful. We also already have tooling and basic-block segmentation available for those benchmarks, making the analysis more convenient.

Recompiling `CesASMe`'s dataset

In practice, benchmarks from `CesASMe` are roughly of the following form:

```
1 for(int measure=0; measure < NUM_MEASURES; ++measure) {
2     measure_start();
3     for(int repeat=0; repeat < NUM_REPEATS; ++repeat) {
4         for(int i=0; i < BENCHMARK_SIZE; ++i) {
5             /* Some kernel, independent of measure, repeat */
6         }
7     }
8     measure_stop();
9 }
```

While this is sensible for conducting throughput measures, it also introduces unwanted dependencies. If, for instance, the kernel consists in $A[i] = C \times A[i] + B[i]$, implemented by

```
1 loop:
2     vmulsd (%rax,%rdi), %xmm0, %xmm1
3     vaddsd (%rbx,%rdi), %xmm1, %xmm1
4     vmovsd %xmm1, (%rax,%rdi)
5     add $8, %rdi
6     cmp %rdi, %r10
7     jne loop
```

a read-after-write dependency from line 4 to line 2 is reported by `depsim` —- although there is no such dependency inherent to the kernel.

However, each iteration of the `measure` (outer) loop and each iteration of the `repeat` (inner) loop will read again each `A[i]` (*ie.* `(%rax,%rdi)` in the assembly) value from the previous inner loop, and write it back. This creates a dependency to the previous iteration of the inner loop, which should in practice be meaningless if `BENCHMARK_SIZE` is large enough. Such dependencies, however, pollute the evaluation results: as `depsim` does not report a dependency's distance, they are considered meaningful; and as they cannot be detected by `staticdeps` — which is unaware of the outer and inner loop —, they introduce unfairness in the evaluation. The actual loss of precision introduced by not discovering such dependencies is instead assessed later by enriching `uiCA` with `staticdeps`.

To avoid detecting these dependencies with `depsim`, we **recompile `CesASMe`'s benchmarks** from the C source code of each benchmark with `NUM_MEASURES = NUM_REPEATS = 1`. We use these

cov_p (%)	cov_u (%)	cov_w (%)
96.0	94.4	98.3

Table 5.1 – Periodic, unweighted and weighted coverage of `staticdeps` on `CesASMe`'s binaries recompiled without repetitions, with a lifetime of 512.

recompiled benchmarks only in the current section. While we do not re-run code transformations from the original Polybenchs, we do recompile the benchmarks from C source. Thus, the results from this section *are not comparable* with results from other sections, as the compiler may have used different optimisations, instructions, etc.

Dependency coverage

For each binary generated by `CesASMe`, we use its cached basic block splitting and occurrence count. Among each binary, we discard any basic block with fewer than 10 % of the occurrence count of the most-hit basic block; this avoids considering basic blocks which were not originally inside loops, and for which loop-carried dependencies would make no sense — and could possibly create false positives.

For each of the considered binaries, we run our dynamic analysis, `depsim`, and record its results. We use a lifetime of 512 instructions for this analysis, as this is roughly the size of recent Intel reorder buffers [Wik21]; as discussed in section 5.3, dependencies spanning farther than the size of the ROB are not microarchitecturally relevant. Dependencies whose source and destination program counters are not in the same basic block are discarded, as `staticdeps` cannot detect them by construction.

For each of the considered basic blocks, we run our static analysis, `staticdeps`. We discard the Δk parameter — how many loop iterations the dependency spans —, as our dynamic analysis does not report an equivalent parameter, but only a pair of program counters.

Dynamic dependencies from `depsim` are converted to *periodic dependencies* in the sense of `staticdeps` as described in subsection 5.4.2: only dependencies occurring on at least 80% of the block's iterations are kept — else, dependencies are considered measurement artifacts. The *periodic coverage* of `staticdeps` dependencies for this basic block *wrt.* `depsim` is the proportion of dependencies found by `staticdeps` among the periodic dependencies extracted from `depsim`:

$$\text{cov}_p = \frac{|\text{found}|}{|\text{found}| + |\text{missed}|}$$

We also keep the raw dependencies from `depsim` — that is, without converting them to periodic dependencies. From these, we consider two metrics: the unweighted dependencies coverage,

$$\text{cov}_u = \frac{|\text{found}|}{|\text{found}| + |\text{missed}|}$$

identical to cov_p but based on unfiltered dependencies, as well as the weighted dependencies coverage,

$$\text{cov}_w = \frac{\sum_{d \in \text{found}} \rho_d}{\sum_{d \in \text{found} \,\cup\, \text{missed}} \rho_d}$$

where ρ_d is the number of occurrences of the dependency d, dynamically detected by `depsim`. Note that such a metric is not meaningful for periodic dependencies as, by construction, each dependency occurs as many times as the loop iterates.

These metrics are presented for the 3 500 binaries of `CesASMe` in Table 5.1. The obtained coverage is consistent between the three metrics used (cov_p, cov_u, cov_w) and the reported coverage is very close to 100 %, giving us good confidence on the accuracy of `staticdeps`.

cov_u (%)	cov_w (%)
95.0	93.7

Table 5.2 – Unweighted and weighted coverage of `staticdeps` on `CesASMe`'s binaries recompiled without repetitions, with an infinite lifetime, as a proxy for points-to analysis.

"Points-to" aliasing analysis

The same methodology can be re-used as a proxy for estimating the rate of aliasing independent pointers in our dataset. Indeed, a major approximation made by `staticdeps` is to assume that any new encountered pointer — function parameters, value read from memory, … — does *not* alias with previously encountered values. This is implemented by the use of a fresh random value for each value yet unknown.

Determining which pointers may point to which other pointers — and, by extension, may point to the same memory region — is called a *points-to analysis* [EGH94]. In the context of `staticdeps`, it characterizes the pointers for which taking a fresh value was *not* representative of the reality.

If we detect, through dynamic analysis, that a value derived from a pointer `a` shares a value with one derived from a pointer `b` – say, `a + k == b + l` —, we can deduce that `a` *points-to* `b`. This is true even if `a + k` at the very beginning of the execution is equal to `b + l` at the very end of the execution: although the pointers will not alias (that is, share the same value at the same moment), they still point to the same memory region and should not be treated as independent.

Our dynamic analyzer, `depsim`, does not have this granularity, as it only reports dependencies between two program counters. A dependency from a PC p to a PC q however implies that a value written to memory at q was read from memory at p, and thus that one of the pointers used at p aliases with one of the pointers used at q.

We thus conduct the same analysis as before, but with an infinite lifetime to account for far-ranging dependencies. We then use cov_u and cov_w as a proxy to measure whether assuming the pointers independent was reasonable: a bad coverage would be a clear indication of non-independent pointers treated as independent. A good coverage is not, formally, an indication of the absence of non-independent pointers: the detected static dependencies may come of other pointers at the same PC. We however believe it reasonable to consider it a good proxy for this metric, as a single assembly line often reads a single value, and usually at most two. We do not use the cov_p metric here, as we want to keep every detected dependency to detect possible aliasing.

The results of this analysis are presented in Table 5.2. The very high coverage rate gives us good confidence that our hypothesis of independent pointers is reasonable, at least within the scope of Polybench, which we believe representative of scientific computation — one of the prominent use-cases of tools such as code analyzers.

5.5.2 Enriching `uiCA`'s model

To estimate the real gain in performance debugging scenarios, we integrate `staticdeps` into `uiCA`.

There is, however, a discrepancy between the two tools: while `staticdeps` works at the assembly instruction level, `uiCA` works at the μOP level. In real hardware, dependencies indeed occur between μOPs; however, we are not aware of the existence of a μOP-level semantic description of the x86-64 ISA (which, by essence, would be declined for each specific processor, as the ISA itself is not concerned with μOPs). This level of detail was thus unsuitable for the `staticdeps` analysis.

Dataset	Bencher	Datapoints	MAPE	Median	Q1	Q3	K_τ
Full	uiCA	3500	29.59 %	18.26 %	7.11 %	52.99 %	0.58
	+ staticdeps	3500	19.15 %	14.44 %	5.86 %	23.96 %	0.81
Pruned	uiCA	2388	18.42 %	11.96 %	5.42 %	23.32 %	0.80
	+ staticdeps	2388	18.77 %	12.18 %	5.31 %	23.55 %	0.80

Table 5.3 – Evaluation through `CesASMe` of the integration of `staticdeps` to `uiCA`

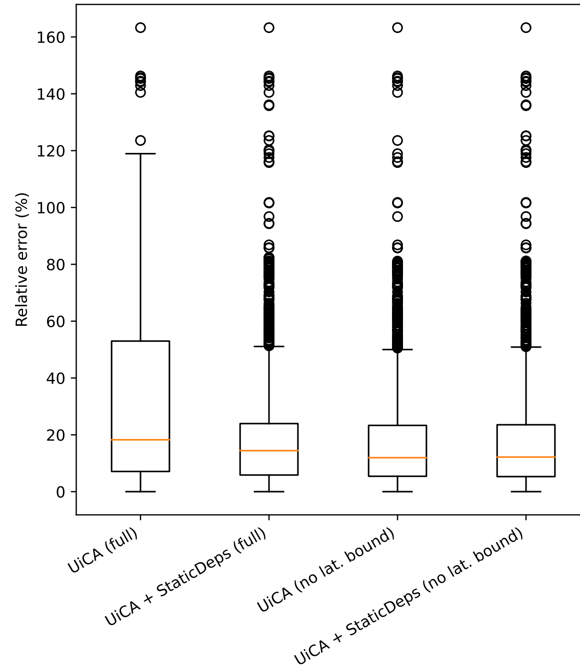


Figure 5.1 – Statistical distribution of relative errors of `uiCA`, with and without `staticdeps` hints, with and without pruning latency bound through memory-carried dependencies rows

We bridge this gap in a conservative way: whenever two instructions i_1, i_2 are found to be dependant, we add a dependency between each couple $\mu_1 \in i_1, \mu_2 \in i_2$. This approximation is thus largely pessimistic, and should predict execution times biased towards a slower computation kernel. A finer model, or a finer (conservative) filtering of which μOPs must be considered dependent — *eg.* a memory dependency can only come from a memory-related μOP — may enhance the accuracy of our integration.

We then evaluate our gains by running `CesASMe`'s harness as we did in chapter 4, running both `uiCA` and `uiCA` + `staticdeps`, on two datasets: first, the full set of 3 500 binaries from the previous chapter; then, the set of binaries pruned to exclude benchmarks heavily relying on memory-carried dependencies introduced in subsection 4.6.4. If `staticdeps` is beneficial to `uiCA`, we expect `uiCA` + `staticdeps` to yield significantly better results than `uiCA` alone on the first dataset. On the second dataset, however, `staticdeps` should provide no significant contribution, as the dataset was pruned to not exhibit significant memory-carried latency-boundness. We present these results in Table 5.3, as well as the corresponding box-plots in Figure 5.1.

We deduce two things from this experiment.

First, the full dataset `uiCA` + `staticdeps` row is extremely close, on every metric, to the pruned, `uiCA`-only row. On this basis, we argue that `staticdeps`' addition to `uiCA` is very conclusive: the hints provided by `staticdeps` are sufficient to make `uiCA`'s results as good on the full dataset as they were before on a dataset pruned of precisely the kind of dependencies

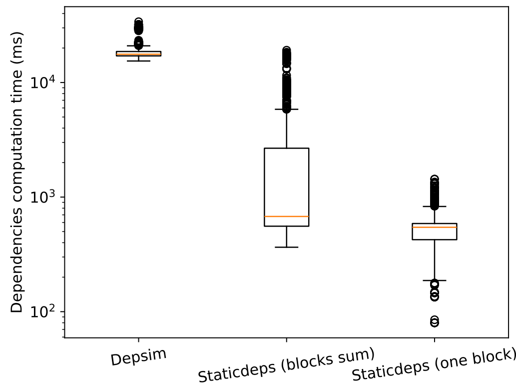


Figure 5.2 – Statistical distribution of `staticdeps` and `depsim` run times on `CesASMe`'s kernels — log y scale

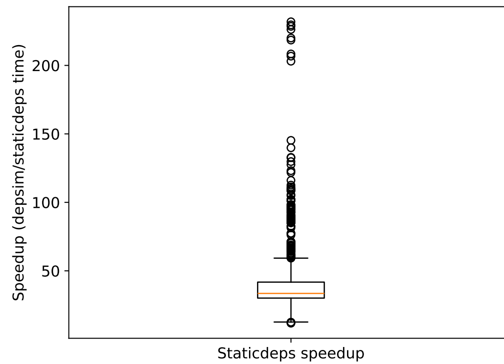


Figure 5.3 – Statistical distribution of `staticdeps`' speedup over `depsim` on `CesASMe`'s kernels

Sequence	Average	Median	Q1	Q3
Seq. (i) — `depsim`	18083 ms	17645 ms	17080 ms	18650 ms
Seq. (iii) — `staticdeps` (sum)	2307 ms	677 ms	557 ms	2700 ms
Seq. (ii) — `staticdeps` (single)	529 ms	545 ms	425 ms	588 ms
Seq. (iv) — speedup	×36.1	×33.5	×30.1	×41.7

Table 5.4 – Statistical distribution of `staticdeps` and `depsim` run times and speedup on `CesASMe`'s kernels

we aim to detect. Thus, at least on workloads similar to Polybench, `staticdeps` is able to resolve the issue of memory-carried dependencies for `uiCA`'s throughput analysis.

Furthermore, `uiCA` and `uiCA` + `staticdeps`' results on the pruned dataset are extremely close. From this, we argue that `staticdeps` does not introduce false positives when no dependency should be found; its addition to `uiCA` does not negatively impact its accuracy whenever it is not relevant.

5.5.3 Analysis speed

The main advantage of a static analysis of dependencies over a dynamic one is its execution time — we should expect from `staticdeps` an analysis time far lower than `depsim`'s.

To assess this, we evaluate on the same `CesASMe` kernels four data sequences:

(i) the execution time of `depsim` on each of `CesASMe`'s kernels;

(ii) the execution time of `staticdeps` on each of the basic blocks of each of `CesASMe`'s kernels;

(iii) for each of those kernels, the sum of the execution times of `staticdeps` on the kernel's constituting basic blocks;

(iv) for each basic block of each of `CesASMe`'s kernels, `staticdeps`' speedup *wrt.* `depsim`, that is, `depsim`'s execution time divided by `staticdeps`'.

As `staticdeps` is likely to be used at the scale of a basic block, we argue that the sequence (ii) is more relevant than sequence (iii); however, the latter might be seen as more fair, as one run of `depsim` yields dependencies of all of the kernel's constituting basic blocks.

We plot the statistical distribution of these series in Figure 5.2 and Figure 5.3, and give numerical data for some statistical indicators in Table 5.4. We note that `staticdeps` is 30

to 40 times faster than depsim. Furthermore, staticdeps is written in Python, more as a proof-of-concept than as production-ready software; meanwhile, depsim is written in C on top of valgrind, an efficient, production-ready software. We expect that with optimization efforts, and a rewrite in a compiled language, the speedup would reach two to three orders of magnitude.

Conclusion

In this chapter, we studied data dependencies within assembly kernels; and more specifically, data dependencies occurring through memory accesses, which we call *memory-carried dependencies*. CesASMe's analysis showed in chapter 4 that this kind of dependency was responsible for a significant portion of state-of-the-art analyzers' prediction errors.

We introduce staticdeps, a heuristic approach based on random values as representatives of abstract values. This approach is able to find data dependencies, including memory-carried ones, loop-carried or not, leveraging semantics of the assembly code provided by valgrind's VEX. It is, however, still unable to find aliasing addresses whose source of aliasing is outside of the studied block's scope — and, as such, suffers from the *lack of context* pointed out in the previous chapter.

Our evaluation of staticdeps against a dynamic analysis baseline, depsim, shows that it finds between 95 % and 98 % of the existing dependencies, depending on the metric used, giving us good confidence in the reliability of staticdeps. We further enrich uiCA with staticdeps, and find that it performs on the full CesASMe's dataset as well as uiCA alone on the pruned dataset of CesASMe, removing memory-carried bottlenecks. From this, we conclude that staticdeps is very successful at finding the data dependencies through memory that actually matter from a performance analysis perspective. We also find that, despite being written in pure Python, staticdeps is at least 30× faster than its C dynamic counterpart, depsim; as such, we expect a compiled and optimized implementation of staticdeps to be two to three orders of magnitude faster than depsim.

Chapter 6

Wrapping it all up

In chapter 2, we introduced `Palmed`, a framework to build a backend model. Following up in chapter 3, we introduced a frontend model for the ARM-based Cortex A72 processor. Then, in chapter 5, we further introduced a dependency detection model. Put together, these three parts cover the major bottlenecks that a code analyzer must take into account.

Both the two first models — frontend and backend — already natively output a cycles per iteration metric; we reduce our dependencies model to a cycles per iteration metric by computing the *critical path*, described below.

To conclude this manuscript, we take a minimalist first approach at combining those three models into a predictor, that we call `A72 combined`, by taking the maximal prediction among the three models.

This method is clearly less precise than *eg.* `uiCA` or `llvm-mca`'s methods, which simulate iterations of the kernel while accounting for each model. It however allows us to quickly and easily evaluate an *upper bound* of the quality of our models: a more refined tool using our models should obtain results at least as good as this method — but we could expect it to perform significantly better.

6.1 Critical path model

To account for dependencies-induced bottlenecks, we compute the *critical path* along the data dependencies graph of the microkernel; that is, the longest path in this graph weighted with source instructions' latencies. The length of this path sets a lower bound to the execution time, as each source instruction must be issued and yield a result before the destination instruction can be issued. This approach is also taken by `Osaca` [Lau+19].

In our case, we use instructions' latencies inferred by `Palmed` and its backend `Pipedream` on the A72.

So far, however, this method would fail to account for out-of-orderness: the latency of an instruction is hidden by other computations, independent of the former one's result. This instruction-level parallelism is limited by the reorder buffer's size.

We thus unroll the kernel as many times as fits in the reorder buffer — accounting for each instruction's μOP count, as we have a frontend model readily available —, and compute the critical path on this unrolled version. Finally, the metric in cycles per iteration is obtained by dividing this critical path's length by the number of times we unrolled the kernel.

6.2 Evaluation

We evaluate `A72 combined` with `CesASMe` on the Raspberry Pi's Cortex A72, using the same set of benchmarks as in chapter 4 recompiled for AArch64. As most of the code analyzers

Bencher	**Datapoints**	**Failures**		**MAPE**	**Median**	**Q1**	**Q3**	K_τ
		(Count)	(%)	(%)	(%)	(%)	(%)	
A72 combined	1767	9	(0.51 %)	19.26 %	12.98 %	5.57 %	25.38 %	0.75
llvm-mca	1775	1	(0.06 %)	32.60 %	25.17 %	8.84 %	59.16 %	0.69
Osaca (backend)	1773	3	(0.17 %)	49.33 %	50.19 %	33.53 %	64.94 %	0.67
Osaca (crit. path)	1773	3	(0.17 %)	84.02 %	70.39 %	40.37 %	91.47 %	0.24

Table 6.1 – Evaluation through `CesASMe` of the `A72 combined` model

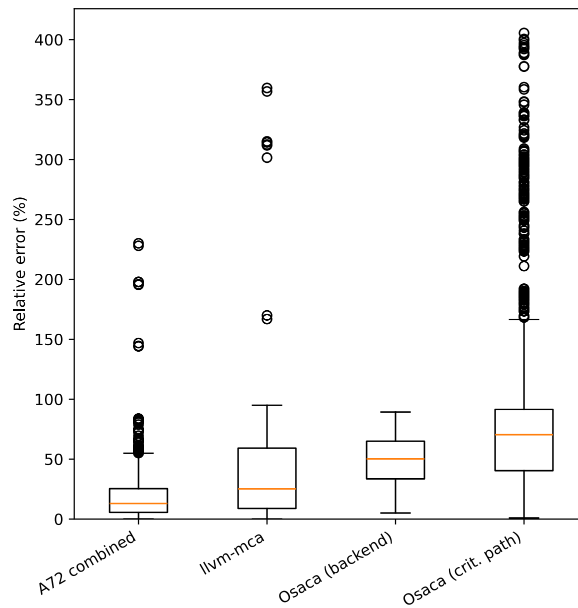


Figure 6.1 – Evaluation through `CesASMe` of the `A72 combined` model

we studied are unable to run on the A72, we are only able to compare `A72 combined` to the baseline `perf` measure, `llvm-mca` and `Osaca`. We use `llvm-mca` at version 18.1.8 and `Osaca` at version 0.5.0. We present the results in Table 6.1 and in Figure 6.1.

Our `A72 combined` model significantly outperforms `llvm-mca`, with a median error approximately half lower than `llvm-mca`'s and a third quartile level with its median. We expect that an iterative model, such as `llvm-mca` or `uiCA`, based on our models' data, would yet significantly outperform `A72 combined`.

6.3 Towards a modular approach?

These models, however — frontend, backend and dependencies —, are only very loosely dependent upon each other. The critical path model, for instance, requires the number of μOPs in one instruction, while the frontend model is purely standalone. Should a standardized format or API for these models emerge, swapping *eg.* our backend model for `uops.info` and running our tool on Intel CPUs would be trivial. Yet better, one could build a "meta-model" relying on these model components handling a logic way more performant than our simple `max`-based model, on which anyone could hot-plug *eg.* a custom frontend model.

The usual approach of the domain to try a new idea, instead, is to create a full analyzer implementing this idea, such as what we did with `Palmed` for backend models, or such as `uiCA`'s implementation, focusing on frontend analysis.

In hindsight, we advocate for the emergence of such a modular code analyzer. It would maybe not be as convenient or well-integrated as "production-ready" code analyzers, such as `llvm-mca` — which is officially packaged for Debian. It could, however, greatly simplify the academic process of trying a new idea on any of the three main models, by decorrelating them. It would also ease the comparative evaluation of those ideas, while eliminating many of the discrepancies between experimental setups that make an actual comparison difficult — the reason that prompted us to make `CesASMe` in chapter 4. Indeed, with such a modular tool, it would be easy to run the same experiment, in the same conditions, while only changing *eg.* the frontend model but keeping a well-tried backend model.

Conclusion

During this manuscript, we explored the main bottlenecks that arise while analyzing the low-level performance of a microkernel:

— frontend bottlenecks — the processor's frontend is unable to saturate the backend with instructions (chapter 2);
— backend bottlenecks — the backend is saturated with instructions from the frontend and is unable to process them fast enough (chapter 3);
— dependencies bottlenecks — data dependencies between instructions prevent the backend from being saturated; the latter is stalled awaiting previous results (chapter 5).

We also conducted in chapter 4 a systematic comparative study of a variety of state-of-the-art code analyzers.

State-of-the-art code analyzers such as `llvm-mca` or `uiCA` already boast a good accuracy. Both of these tools — and most of the others also — are however based on models obtained by various degrees of manual investigation, and cannot be adapted without further manual effort to future or uncharted microprocessors.

The field of microarchitectural models for code analysis emerged with fundamentally manual methods, such as Agner Fog's tables. Such tables, however, may now be produced in a more automated way using `uops.info` — at least for certain microarchitectures; `PMEvo` pushes further in this direction by automatically computing a frontend model from benchmarks — but still has trouble scaling to a full instruction set. In its own way, `Ithemal`, a machine-learning based approach, could also be considered automated — yet, it still requires a large training set for the intended processor, which must be at least partially crafted manually. This trend towards model automation seems only natural as new microarchitectures keep appearing, while new ISAs such as ARM reach the supercomputer area.

We investigated this direction by exploring the three major bottlenecks mentioned earlier in the perspective of providing fully-automated, benchmarks-based models for each of them. Optimally, these models should be generated by simply executing a program on a machine running on top of the targeted microarchitecture.

— We contributed to `Palmed`, a framework able to extract a port-mapping of a processor, serving as a backend model.
— We manually extracted a frontend model for the Cortex A72 processor. We believe that the foundation of our methodology works on most processors. To this end, we provide a parametric model that may serve as a scaffold for future works willing to build an automatic frontend model. Some parameters of this model must however still be investigated, and their relative importance evaluated.
— We provided with `staticdeps` a method to to extract data dependencies between instructions. It is able to detect *loop-carried* dependencies (dependencies that span across multiple loop iterations), as well as *memory-carried* dependencies (dependencies based on reading at a memory address written by another instruction). While the former is widely implemented, the latter is, to the best of our knowledge, an original contribution. We bundled this method in a processor-independent tool, based on semantics of the ISA

provided by `valgrind`, which supports a variety of ISAs.

We evaluated independently these three models, each of them providing satisfactory results: `Palmed` is competitive with the state of the art, with the advantage of being automatic; our frontend model significantly improves a backend model's accuracy and our dependencies model significantly improves `uiCA`'s results, while being consistent with a dynamic dependencies analysis.

Finally, in the pre-conclusive chapter Wrapping it all up, we loosely combine our three pieces of code analyzer model into a very simple full model, returning the maximal prediction from the three major bottleneck analyzers. While this simplistic model would certainly benefit from a more integrated approach, our results already significantly surpass `llvm-mca`, one of the very few state-of-the-art tools on ARM processors.

We also identified multiple weaknesses in the current state of the art from our comparative experiments with `CesASMe`.

First, none of the state-of-the-art tools have a good support for dependencies across memory. Such dependencies were present in about a third of `CesASMe`'s benchmark set. While we built this benchmark set aiming for representative data, there is no clear evidence that these dependencies are so strongly present in the codes analyzed in real usecases. We however believe that such cases regularly occur, and we also saw that the performance of code analyzers drops sharply in their presence.

We also found the bottleneck prediction offered by some code analyzers still very uncertain. In our experiments, the tools disagreed more often than not on the presence or absence of a bottleneck, with no outstanding tool; we are thus unable to conclude on the relative performance of tools on this aspect. On the other hand, sensitivity analysis, as implemented *eg.* by `Gus`, seems a theoretically sound way to evaluate the presence or absence of a bottleneck in a microkernel; it is, however, prohibitively slow for many usecases. In this respect, a study of code analyzers' predictions against results from sensitivity analysis would certainly bring more conclusive results.

Finally, we observed on `BHive`'s results the effects of a *lack of context* for an analysis. `BHive` measures a real execution, on real hardware, of a kernel; as such, it yields excellent accuracy in many cases, with a median error of about 8%. Yet, it still lacks in accuracy in many other cases, with its third quartile (23%) above `uiCA` or `IACA`'s median result (about 18%), and far-reaching outliers bringing its mean error on-par with `uiCA`'s. Indeed, what precedes a loop nest and the real values present in registers impact the performance of the loop nest. The effects can be of fairly high level, such as pointer aliasing, leading to false positives or negatives in dependency detections. They can also be of a microarchitectural level, such as the observable performance loss of memory accesses — even with cache hits — when memory reads cross a cache line boundary.

This lack of context incurs a significant loss of accuracy for static analyzers, as we saw in subsection 4.6.2 that the same instruction, depending on its registers' values, can be twice as slow even without aliasing, or 19 times slower upon aliasing. With `CesASMe`, we sketch the embryo of a solution, with a simple and fast pass of dynamic analysis through instrumentation, gathering data for a subsequent pass of static analysis. Such a method might help recreating the context needed for an accurate analysis.

Source code and data availability

The software written during my PhD is available under free software licenses — as publicly funded works should be. The raw data resulting from experiments is typically available within the relevant project's repository.

Below is a list of the most important source code repositories. A longer list, including dependencies and less important projects, can be found here [1].

— `Palmed`: here [2]
— A72 frontend: here [3]
— `CesASMe`: here [4]
— `staticdeps`: here [5]
— `A72 combined`: here [6]
— This manuscript: here [7]

1. https://gitlab.inria.fr/tbastian/phd-repo-links
2. https://gitlab.inria.fr/nderumig/palmed
3. https://gitlab.inria.fr/tbastian/a72_frontend
4. https://gitlab.inria.fr/CORSE/genbenchs
5. https://gitlab.inria.fr/tbastian/staticdeps
6. https://gitlab.inria.fr/CORSE/a72_combined
7. https://git.tobast.fr/tobast/phd-thesis

Bibliography

[15] *Cortex-A72 Software Optimization Guide*. ARM. Mar. 2015.

[23a] *AMD64 Architecture Programmer's Manual, volume 2*. AMD. June 2023.

[23b] *Intel® 64 and IA-32 Architectures Optimization Reference Manual Volume 1*. Intel Corporation. Sept. 2023.

[23c] *Intel® 64 and IA-32 Architectures Software Developer's Manual, volume 1*. Intel Corporation. June 2023.

[23d] *Software Optimization Guide for the AMD Zen4 Microarchitecture*. Publication number 57647. Advanced Micro Devices (AMD). Jan. 2023.

[AR19] Andreas Abel and Jan Reineke. « uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures ». In: *ASPLOS*. ASPLOS '19. Providence, RI, USA: ACM, 2019, pp. 673–686. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304062. URL: http://doi.acm.org/10.1145/3297858.3304062.

[AR22] Andreas Abel and Jan Reineke. « UiCA: Accurate Throughput Prediction of Basic Blocks on Recent Intel Microarchitectures ». In: *Proceedings of the 36th ACM International Conference on Supercomputing*. ICS '22. Virtual Event: Association for Computing Machinery, 2022. ISBN: 9781450392815. DOI: 10.1145/3524059.3532396. URL: https://doi.org/10.1145/3524059.3532396.

[ARM] ARM. *Cortex A-72*. https://developer.arm.com/Processors/Cortex-A72.

[Bal+13] Daniel Balouek et al. « Adding Virtualization Capabilities to the Grid'5000 Testbed ». In: *Cloud Computing and Services Science*. Ed. by Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan. Vol. 367. Communications in Computer and Information Science. Springer International Publishing, 2013, pp. 3–20. ISBN: 978-3-319-04518-4. DOI: 10.1007/978-3-319-04519-1_1.

[Bar20] Barcelona Supercomputing Center. *Technical information on the MareNostrum 4 supercomputer's ARM cluster*. https://www.bsc.es/innovation-and-services/technical-information-cte-arm. 2020.

[Bas23] Théophile Bastian. *Rowmajor vs. colmajor experiments*. https://gitlab.inria.fr/tbastian/rowmajor-measure. Oct. 2023.

[Bia18] Andrea Di Biagio. *[RFC] llvm-mca: a static performance analysis tool*. https://lists.llvm.org/pipermail/llvm-dev/2018-March/121490.html. Request for comments on the llvm-dev mailing-list. Mar. 2018.

[BLK18] James Bucek, Klaus-Dieter Lange, and Jóakim von Kistowski. « SPEC CPU2017: Next-Generation Compute Benchmark ». In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE 2018*. Ed. by Katinka Wolter, William J. Knottenbelt, André van Hoorn, and Manoj Nambiar. Berlin, Germany: ACM, Apr. 2018, pp. 41–42. DOI: 10.1145/3185768.3185771. URL: https://doi.org/10.1145/3185768.3185771.

[Bon20] Uday Bondhugula. *High Performance Code Generation in MLIR: An Early Case Study with GEMM*. 2020. arXiv: 2003.00532 [cs.PF].

[BRS07] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. *PLuTo: A Practical and Fully Automatic Polyhedral Parallelizer and Locality Optimizer*. Tech. rep. OSU-CISRC-10/07-TR70. The Ohio State University, Oct. 2007.

[Che+09] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. « Rodinia: A benchmark suite for heterogeneous computing ». In: *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 2009, pp. 44–54. DOI: 10.1109/IISWC.2009.5306797.

[Che+19] Yishen Chen, Ajay Brahmakshatriya, Charith Mendis, Alex Renda, Eric Atkinson, Ondřej Sýkora, Saman Amarasinghe, and Michael Carbin. « BHive: A Benchmark Suite and Measurement Framework for Validating x86-64 Basic Block Performance Models ». In: *2019 IEEE International Symposium on Workload Characterization (IISWC)*. 2019, pp. 167–177. DOI: 10.1109/IISWC47752.2019.9042166.

[Com+95] TIS Committee et al. *Tool interface standard (TIS) executable and linking format (ELF) specification version 1.2*. 1995.

[Con23] Contribution of Working Groups I, II and III to the Sixth Assessment Report of the Intergovernmental Panel on Climate Change [Core Writing Team, H. Lee and J. Romero (eds.)] *IPCC, 2023: Climate Change 2023: Synthesis Report*. Ed. by IPCC, Geneva, Switzerland. doi: 10.59327/IPCC/AR6-9789291691647. 2023.

[Del19] Dell. *Estimated product carbon footprint for PowerEdge C6420*. Tech. rep. https://i.dell.com/sites/csdocuments/CorpComm_Docs/en/carbon-footprint-poweredge-c6420.pdf. Dell, 2019.

[Der+22] Nicolas Derumigny, Théophile Bastian, Fabian Gruber, Guillaume Iooss, Christophe Guillon, Louis-Noël Pouchet, and Fabrice Rastello. « PALMED: Throughput Characterization for Superscalar Architectures ». In: *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2022, pp. 106–117. DOI: 10.1109/CGO53902.2022.9741289.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. « Context-sensitive interprocedural points-to analysis in the presence of function pointers ». In: *SIGPLAN Not.* 29.6 (June 1994), pp. 242–256. ISSN: 0362-1340. DOI: 10.1145/773473.178264. URL: https://doi.org/10.1145/773473.178264.

[Ele] Electricity Maps data. https://app.electricitymaps.com/.

[ESE06] S. Eyerman, J.E. Smith, and L. Eeckhout. « Characterizing the branch misprediction penalty ». In: *2006 IEEE International Symposium on Performance Analysis of Systems and Software*. 2006, pp. 48–58. DOI: 10.1109/ISPASS.2006.1620789.

[Fog16] Agner Fog. *Discussion on blogpost*. https://www.agner.org/optimize/blog/read.php?i=581. 2016.

[Fog20] Agner Fog. *Instruction tables: Lists of instruction latencies, through-puts and micro-operation breakdowns for Intel, AMD and VIA CPUs*. 2020. URL: http://www.agner.org/optimize/instruction_tables.pdf.

[Fuj23] Fujitsu Limited. *Supercomputer Fugaku retains first place worldwide in HPCG and Graph500 rankings*. https://www.fujitsu.com/global/about/resources/news/press-releases/2022/1115-01.html. May 2023.

[Goo] Google. *EXEgesis*. https://github.com/google/EXEgesis.

[Gru19] Fabian Gruber. « Performance Debugging Toolbox for Binaries: Sensitivity Analysis and Dependence Profiling ». 2019GREAM071. PhD thesis. Université Grenoble Alpes, 2019. URL: http://www.theses.fr/2019GREAM071/document.

[Gup+22] Udit Gupta, Mariam Elgamal, Gage Hills, Gu-Yeon Wei, Hsien-Hsin S. Lee, David Brooks, and Carole-Jean Wu. « ACT: Designing Sustainable Computer Systems with an Architectural Carbon Modeling Tool ». In: *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ISCA '22. New York, New York: Association for Computing Machinery, 2022, pp. 784–799. ISBN: 9781450386104. DOI: 10.1145/3470496.3527408. URL: https://doi.org/10.1145/3470496.3527408.

[Gur] LLC Gurobi Optimization. *Gurobi Optimizer*. https://www.gurobi.com.

[Haa23] Rene Haas. *Together, we are building the future of computing, on Arm*. https://www.arm.com/company/news/2023/09/building-the-future-of-computing-on-arm. ARM, 2023.

[Inta] Intel. *VTune profiler*. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.

[Intb] Intel Corporation. *Intel Architecture Code Analyzer (IACA)*. https://software.intel.com/en-us/articles/intel-architecture-code-analyzer/.

[Int03] Intel. *oneAPI Math Kernel Library (oneMKL)*. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html. 2003.

[Jev66] William Stanley Jevons. *The coal question; an inquiry concerning the progress of the nation and the probable exhaustion of our coal-mines*. Macmillan, 1866.

[Kav07] Nikolaos Kavvadias. *Hardware looping unit*. 2007.

[Ken38] Maurice G Kendall. « A new measure of rank correlation ». In: *Biometrika* 30.1/2 (1938), pp. 81–93.

[Lau+18] Jan Laukemann, Julian Hammer, Johannes Hofmann, Georg Hager, and Gerhard Wellein. « Automated Instruction Stream Throughput Prediction for Intel and AMD Microarchitectures ». In: *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 2018, pp. 121–131. DOI: 10.1109/PMBS.2018.8641578.

[Lau+19] Jan Laukemann, Julian Hammer, Georg Hager, and Gerhard Wellein. « Automatic Throughput and Critical Path Analysis of x86 and ARM Assembly Kernels ». In: *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 2019, pp. 1–6. DOI: 10.1109/PMBS49563.2019.00006.

[Lin] Linux Kernel. *perf: Linux profiling with performance counters*. http://perf.wiki.kernel.org/index.php/Main_Page.

[MAC18] Charith Mendis, Saman P. Amarasinghe, and Michael Carbin. « Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks ». In: *CoRR* abs/1808.07412 (2018). arXiv: 1808.07412. URL: http://arxiv.org/abs/1808.07412.

[Man+23] Filippo Mantovani, Pablo Vizcaino, Fabio Banchelli, Marta Garcia-Gasulla, Roger Ferrer, Georgios Ieronymakis, Nikolaos Dimou, Vassilis Papaefstathiou, and Jesus Labarta. « Software Development Vehicles to enable extended and early co-design: a RISC-V and HPC case of study ». In: *International Conference on High Performance Computing*. Springer. 2023, pp. 526–537.

[Man23] Filippo Mantovani. Private communication during the ACACES summer school. July 2023.

[Mat21] Satoshi Matsuoka. « Fugaku and A64FX: the First Exascale Supercomputer and its Innovative Arm CPU ». In: *2021 Symposium on VLSI Circuits*. 2021, pp. 1–3. DOI: 10.23919/VLSICircuits52068.2021.9492415.

[Muc+99] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. « PAPI: A portable interface to hardware performance counters ». In: *Proceedings of the department of defense HPCMP users group conference*. Vol. 710. 1999.

[NS03] Nicholas Nethercote and Julian Seward. « Valgrind: A Program Supervision Framework ». In: *Electr. Notes Theor. Comput. Sci.* 89.2 (2003), pp. 44–66.

[Pou09] Louis-Noël Pouchet. *PoCC, the Polyhedral Compiler Collection*. https://www.cs.colostate.edu/~pouchet/software/pocc/. 2009.

[PY16] Louis-Noël Pouchet and Tomofumi Yuki. *PolyBench/C: The polyhedral benchmark suite, version 4.2*. http://polybench.sf.net. 2016.

[QC] Nguyen Anh Quynh and the Capstone collaborators. *Capstone engine*. https://www.capstone-engine.org/.

[QEM] QEMU. *QEMU: the FAST! processor emulator*. https://www.qemu.org.

[Ren+21] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M. Tullsen, and Ashish Venkat. « I See Dead µops: Leaking Secrets via Intel/AMD Micro-Op Caches ». In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 2021, pp. 361–374. DOI: 10.1109/ISCA52012.2021.00036.

[RH20] Fabian Ritter and Sebastian 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*. Ed. by Alastair F. Donaldson and Emina Torlak. London, UK: ACM, June 2020, pp. 608–622. DOI: 10.1145/3385412.3385995. URL: https://doi.org/10.1145/3385412.3385995.

[RH22] Fabian Ritter and Sebastian Hack. « AnICA: Analyzing Inconsistencies in Microarchitectural Code Analyzers ». In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (Oct. 2022). DOI: 10.1145/3563288. URL: https://doi.org/10.1145/3563288.

[Rou87] Peter J. Rousseeuw. « Silhouettes: A graphical aid to the interpretation and validation of cluster analysis ». In: *Journal of Computational and Applied Mathematics* 20 (1987), pp. 53–65. ISSN: 0377-0427. DOI: https://doi.org/10.1016/0377-0427(87)90125-7. URL: https://www.sciencedirect.com/science/article/pii/0377042787901257.

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

[SRO04] Ravi P Singh, Charles P Roth, and Gregory A Overkamp. *Hardware loops*. US Patent 6,748,523. June 2004.

[TJ01] D. Talla and L.K. John. « Cost-effective hardware acceleration of multimedia applications ». In: *Proceedings 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors. ICCD 2001*. 2001, pp. 415–424. DOI: 10.1109/ICCD.2001.955060.

[Val23] Valgrind developpers. *Valgrind — supported platforms*. https://valgrind.org/info/platforms.html. Sept. 2023.

[Vis+21] A V Vishnekov, E M Ivanova, N A Stepanov, and N D Shaimov. « A Simulation Model for Macro- and Micro-Fusion Algorithms in the CPU Core ». In: *Journal of Physics: Conference Series* 1740.1 (Jan. 2021), p. 012053. DOI: 10.1088/1742-6596/1740/1/012053. URL: https://dx.doi.org/10.1088/1742-6596/1740/1/012053.

[Vol22] Tanya Volnina. « Enable AVX-512 instructions in Valgrind ». In: FOSDEM, 2022. URL: https://fosdem.org/2022/schedule/event/valgrind_avx512/.

[Wan+13] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. « AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on X86 CPUs ». In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13. Denver, Colorado: Association for Computing Machinery, 2013. ISBN: 9781450323789. DOI: 10.1145/2503210.2503219. URL: https://doi.org/10.1145/2503210.2503219.

[War63] Joe H. Ward. « Hierarchical Grouping to Optimize an Objective Function ». In: *Journal of the American Statistical Association* 58.301 (1963), pp. 236–244. ISSN: 01621459. URL: http://www.jstor.org/stable/2282967 (visited on 09/15/2023).

[Wat+11] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. « The risc-v instruction set manual, volume i: Base user-level isa ». In: *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62* 116 (2011), pp. 1–32.

[Wik21] WikiChip. *Intel Details Golden Cove: Next-Generation Big Core For Client and Server SoCs*. https://fuse.wikichip.org/news/6111/intel-details-golden-cove-next-generation-big-core-for-client-and-server-socs/. Aug. 2021.

[Xia] Zhang Xianyi. *OpenBLAS: an optimized BLAS library*. https://www.qemu.org.

[YM16] Richard York and Julius Alexander McGee. « Understanding the Jevons paradox ». In: *Environmental Sociology* 2.1 (2016), pp. 77–87. DOI: 10.1080/23251042.2015.1106060.