Performance Debugging Toolbox for Binaries : Sensitivity Analysis and Dependence Profiling

Fabian Gruber

Université Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Fabian GRUBER

Thèse dirigée par **Fabrice RASTELLO**

préparée au sein du **Laboratoire d'Informatique de Grenoble** dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Débogage de performance pour code binaire: Analyse de sensitivité et profilage de dépendances

Performance Debugging Toolbox for Binaries: Sensitivity Analysis and Dependence Profiling

Thèse soutenue publiquement le **17 décembre 2019**, devant le jury composé de :

Monsieur FABRICE RASTELLO
DIRECTEUR DE RECHERCHE, INRIA CENTRE DE GRENOBLE RHÔNE-ALPES, Directeur de thèse
Monsieur BARTON MILLER
PROFESSEUR, UNIVERSITY OF WISCONSIN-MADISON – USA, Rapporteur
Monsieur DENIS BARTHOU
PROFESSEUR, BORDEAUX INP, Rapporteur
Monsieur FRANZ FRANCHETTI
PROFESSEUR, CARNEGIE MELLON UNIVERSITY – USA, Examinateur
Monsieur FREDERIC PETROT
PROFESSEUR, GRENOBLE INP, Président
Monsieur SVILEN KANEV
DOCTEUR-INGENIEUR, GOOGLE – USA, Examinateur

Résumé

Le débogage, tel qu'il est généralement défini, consiste à trouver et à supprimer les problèmes empêchant un logiciel de fonctionner correctement. Quand on parle de bogues et de débogage, on fait donc habituellement référence à des *bogues fonctionnels* et au *débogage fonctionnel*. Dans le contexte de cette thèse, cependant, nous parlerons des *bogues de performance* et de *débogage de performance*. Nous ne cherchons donc pas les problèmes engendrant de mauvais comportements du programme, mais les problèmes qui le rendent inefficace, trop lent, ou qui induisent une trop grande utilisation de ressources. À cette fin, nous avons développé des outils qui analysent et modélisent la performance pour aider les programmeurs à améliorer leur code de ce point de vue là. Nous proposons les deux techniques de débogage de performance suivantes: *analyse des goulets d'étranglement basée sur la sensibilité* et *Suggestions d'optimisation polyédrique basées sur les dépendances de données*.

Analyse des goulets d'étranglement basée sur la sensibilité Il peut être étonnamment difficile de répondre à une question apparemment anodine sur la performance d'un programme, comme par exemple celle de savoir s'il est limité par la mémoire ou par le CPU. En effet, le CPU et la mémoire ne sont pas deux ressources complètement indépendantes, mais sont composés de multiples sous-systèmes complexes et interdépendants. Ici, un blocage d'une ressource peut à la fois masquer ou aggraver des problèmes avec une autre ressource. Nous présentons une analyse des goulets d'étranglement basée sur la sensibilité qui utilise un modèle de performance de haut niveau implémenté dans Gus, un simulateur CPU rapide pour identifier les goulets d'étranglement de performance.

Notre modèle de performance a besoin d'une base de référence pour la performance attendue des différentes opérations sur un CPU, comme le pic IPC et comment différentes instructions se partagent les ressources du processeur. Malheureusement, cette information est rarement publiée par les fournisseurs de matériel, comme Intel ou AMD. Pour construire notre modèle de processeur, nous avons développé un système permettant de récupérer les informations requises en utilisant des micro-benchmarks générés automatiquement.

Suggestions d'optimisation polyédrique basées sur les dépendances de données Nous avons développé Mickey, un profileur dynamique de dépendances de données qui fournit un retour d'optimisation de haut niveau sur l'applicabilité et la rentabilité des optimisations manquées par le compilateur. Mickey exploite le modèle polyédrique, un puissant cadre d'optimisation pour trouver des séquences de transformations de boucle afin d'exposer la localité des données et d'implémenter à la fois le parallélisme gros-grain, c'est-à-dire au niveau thread, et grain-fin, c'est-à-dire au niveau vectoriel. Notre outil utilise une instrumentation binaire dynamique pour analyser des programmes écrits dans différents langages de programmation ou utilisant des bibliothèques tierces pour lesquelles aucun code source n'est disponible.

En interne Mickey utilise une représentation intermédiaire (RI) polyédrique qui code à la fois l'exécution dynamique des instructions d'un programme ainsi que ses dépendances de données. La RI ne capture pas seulement les dépendances de données à travers plusieurs boucles mais aussi à travers des appels de procédure, éventuellement récursifs. Nous avons développé un algorithme efficace de compression de traces qui construit cette RI polyédrique à partir de l'exécution d'un programme. L'folding algorithm trouve aussi des accès réguliers dans les accès mémoire pour prédire la possibilité et la rentabilité de la vectorisation. Il passe à l'échelle sur de gros programmes grâce à un mécanisme de sur-approximation sûr et sélectif pour les applications partiellement irrégulières.

Abstract

Debugging, as usually understood, revolves around finding and removing defects in software that prevent it from functioning correctly. That is, when one talks about bugs and debugging one usually means *functional bugs* and *functional debugging*. In the context of this thesis, however, we will talk about *performance bugs* and *performance debugging*. Meaning we want to find defects that do not cause a program to crash or behave wrongly, but that make it run inefficiently, too slow, or use too many resources. To that end, we have developed tools that analyse and model the performance to help programmers improve their code to get better performance. We propose the following two performance debugging techniques: *sensitivity based bottleneck analysis* and *data-dependence profiling driven optimization feedback*.

Sensitivity Based Performance Bottleneck Analysis Answering a seemingly trivial question about a program's performance, such as whether it is memory-bound or CPU-bound, can be surprisingly difficult. This is because the CPU and memory are not merely two completely independent resources, but are composed of multiple complex interdependent subsystems. Here a stall of one resource can both mask or aggravate problems with another resource. We present a *sensitivity based performance bottleneck analysis* that uses high-level performance model implemented in Gus, a fast CPU simulator to pinpoint performance bottlenecks.

Our performance model needs a baseline for the expected performance of different operations on a CPU, like the peak IPC and how different instructions compete for processor resources. Unfortunately, this information is seldom published by hardware vendors, such as Intel or AMD. To build our processor model, we have developed a system to reverse-engineer the required information using automatically generated micro-benchmarks.

Data-Dependence Driven Polyhedral Optimization Feedback We have developed Mickey, a dynamic data-dependence profiler that provides high-level optimization feedback on the applicability and profitability of optimizations missed by the compiler. Mickey leverages the polyhedral model, a powerful optimization framework for finding sequences of loop transformations to expose data locality and implement both coarse, i.e. thread, and fine-grain, i.e. vector-level, parallelism. Our tool uses dynamic binary instrumentation allowing it to analyze program written in different programming languages or using third-party libraries for which no source code is available.

Internally Mickey uses a polyhedral intermediate representation (IR) that encodes both the dynamic execution of a program's instructions as well as its data dependencies. The IR not only captures data dependencies across multiple loops but also across, possibly recursive, procedure calls. We have developed an efficient trace compression algorithm, called the folding algorithm, that constructs this polyhedral IR from a program's execution. The folding algorithm also finds strides in memory accesses to predict the possibility and profitability of vectorization. It can scale to real-life applications thanks to a safe, selective over-approximation mechanism for partially irregular data dependencies and iteration spaces.

Dedication

This thesis is dedicated to my future wife Kim Hyunyoung, my family, and my friends.
Thank you from the bottom of my heart.

Acknowledgements

The work presented in this thesis has been done in collaboration with my advisor Fabrice Rastello, Louis-Noël Pouchet from the Colorado State University, Christophe Guillon and others from STMicroelectronics, my colleagues Diogo Nunes Sampaio, Manuel Selva, Nicolas Tollenare, Nicolas Derumigny, and many more.
I sincerely thank them all for their advice, help, and hard work.
Without them my thesis would not have been possible.
I also sincerely thank Imma Presseguer for all the administrative work, moral support and advice.

Contents

Chapter 1

Introduction

Contents

It is now nearly fifteen years ago that Herb Sutter wrote his famous article “The Free Lunch Is Over” [201]. In this piece, he argued that the classical methods of boosting single-core performance, such as raising the CPU clock frequency and straight-line instruction throughput, are no longer feasible. He claims that the single-core performance of modern microprocessors has completely stagnated and that multi-core architectures and concurrent programming models are the only way to increase performance going forward. Only a few years later, Hill et al. [94] analysed the state of microprocessor designs of the time and gave some broad recommendations that, at least partially, contradict Sutter’s conclusions. One of their recommendations states that “Researchers should seek methods of increasing [single] core performance even at a high cost.”.

Figure 1.1 shows data describing the overall trends in performance parameters of microprocessors of the last 42 years. If we look at the data, it seems that both predictions have been, to some degree, proven correct. Clock frequency has completely stagnated, while the average core count of processors has exploded. And yet, the single-core performance of processors, as measured by the SpecINT [91, 92] benchmark suite, has not stopped rising. Albeit it is admittedly growing slower than it used to. Since the clock frequency has not been rising, we can deduce that the increase in performance has indeed been paid for “at a high”, by an increase in complexity of hardware and software.

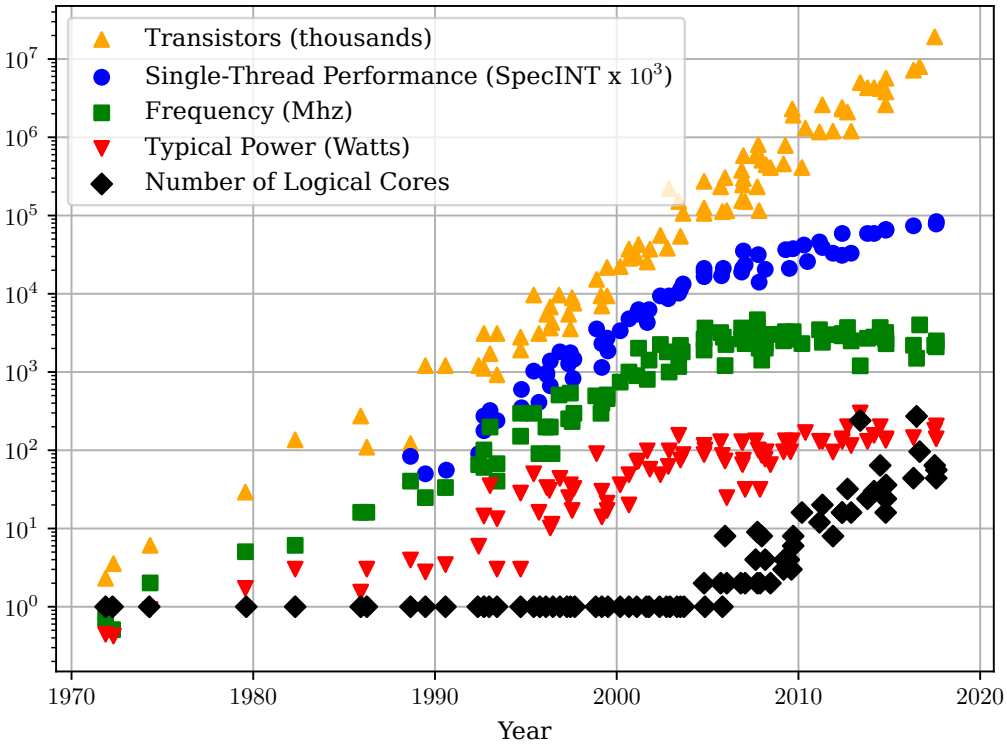


Figure 1.1 – 42 years of microprocessor trends [176].

In day to day programming, this increase of complexity is often hidden behind abstractions. A modern x86 processor, for example, in many ways still pretends to work much the same as a model from the 1980s. That is, to a programmer it, seems that instructions execute sequentially and that every load and store directly touches main memory. Under the hood, however, a modern CPU is a sophisticated parallel system with a deep hierarchy of memory

caches. However, all these abstractions very quickly break down when one wants to study and understand the performance of an application. Then it suddenly becomes crucially important to know which data dependency stopped two instructions from execution in parallel or which memory access caused a cache miss. To effectively analyse and debug the performance of a program, one has to look under the hood and remove the abstractions. It is the goal of this thesis to propose tools and techniques for helping programmers better understand the behaviour of their programs and guide them in the process of performance optimization.

1.1 Performance Debugging

Debugging, as usually understood, revolves around finding and removing defects in software that prevent it from functioning correctly. That is, when one talks about bugs and debugging one usually means *functional bugs* and *functional debugging*. In the context of this thesis, however, we will talk about *performance bugs* and *performance debugging*. Meaning we want to find defects that do not cause a program to crash or behave wrongly, but that make it run inefficiently, too slow, or use too many resources.

Even the best programmer has, at some point, written a buggy program. When they observe that something is amiss they will often use *functional debugging* tools like GDB [74] that help find errors in the behaviour of programs. That is, the classical problems for which programmers turn to functional debuggers are:

1. *Where* is the program misbehaving or crashing?
2. *Why* is the program misbehaving or crashing?

Usually, these tools do not directly try to answer the question *how* one can prevent a program from misbehaving or crashing since this would require changing the semantics of the program. While there are approaches that attempt this from the fields of software verification, like runtime enforcement [69], they require a specification of the intended semantics of a program and are not yet commonly used.

Performance debuggers or *performance analysis tools*, on the other hand, help finding problems in the *non-functional* aspects of programs. That is, the problems which make a programmer turn to these tools are:

1. *Where* is the program slow?
2. *Why* is the program slow?
3. *How* can one make the program run faster?

Historically, existing techniques have only measured and predicted the current performance characteristics of programs and so, similarly to functional debugging tools, only help answer the first two questions. In this thesis, we will develop two approaches that try to extrapolate the possible performance a system could achieve to help answer the last two questions, i.e., why programs are slow and how they can be made more efficient.

1.2 Background

1.2.1 Profiling

The standard approach for identifying performance bugs is profiling. The most straightforward profiling technique is to measure the time required to run a piece of code. Tools like GPROF [78] help automate this process by automatically inserting code to measure the runtime of individual functions via the compiler. Other tools like QPT [122] insert code to count the number of time every basic block in a program executes or how many times every edge in the control-flow-graph is taken. These approaches can help detect *hot* program regions, i.e., *where* a program is

spending most of its time. However, knowing only where the hot regions of a program are is all too often not enough to fix performance bugs.

To help pinpoint the causes of performance problems modern processors offer *hardware performance counters* [1], also called *hardware counters*, or *performance counters* [44, 151]. Hardware counters allow measuring many different performance-related events, such as the number of instructions executed or the number of cache hits and misses at different cache levels. At the most basic level, performance counters work the same as time measurements. To measure, for example, the number of L2 cache misses that occur in a given function it suffices to insert code to read the current value of the corresponding counter at the beginning and the end of the function and subtract the two values.

Many processors also support sampling modes which make it possible to use hardware counters without having to modify an application to explicitly read counters. Different vendors and processor generations implement a wide variety of different sampling models such as, for example, event-based sampling, time sampling, and instruction-based sampling. The details of how these different sampling techniques work, how they can be used, and how precise they differ quite a lot. However, the overall idea is that at regular, user-configured intervals the processor halts normal execution and stores the current values of the hardware performance counters, usually by writing some data to a preconfigured memory location or by raising a special interrupt. Sampling-based profilers then use statistical methods to extrapolate the behaviour of instructions for which they have no samples. Directly using hardware performance counters is quite complicated and so over time a plethora of libraries and tools, such as PAPI [65], LIKWID [172], HPCToolkit [4], Linux perf [144], Intel VTune [169], have been developed for this purpose.

Despite their utility hardware performance counters suffer from some severe limitations. While configuring the CPU to count performance events has no, or barely any, overhead, the reading of counters, either directly or through sampling, is not free. Directly reading counters uses at least some CPU resources such as registers and profiling requires either a costly CPU interrupt or some writes to memory. Consequently using hardware counters can cause an *observer effect* where the act of measuring the performance of an application changes its performance. This can lead to so-called performance *heisenbugs*, named after Heisenberg's uncertainty principle. It also makes fine-grain profiling using performance counters difficult.

Linux perf, a widely used sampling-based profiling tool, for example, by default limits the maximum sampling rate. This is to prevent programs from accidentally spending more time reading performance counters and writing to internal buffers than actually executing real code [88] On an Intel i5-4590 CPU running at 3.3 GHz the limit is 50k samples per second, which means one can at best obtain one sample every 66k instructions.

Another problem with performance counters is that one is limited to measuring a fixed set of metrics. With this approach, it is impossible to profile for an event for which the hardware offers no counter. Furthermore, even on the most recent processors, one can not enable all available performance event counters at the same time, but usually only a small handful, around four to ten.

An alternative to using hardware performance counters is to use a simulator such as GEM5 [23]. A cycle-accurate simulator makes it possible to produce a detailed trace of performance-related events without during execution without perturbing the behaviour of the original program. Since simulators are software, it is also possible to add new performance counters if necessary.

However, the power and flexibility of simulators also come at a cost. Developing a cycle-level simulator is extremely time consuming and requires a lot of expertise. A detailed simulation is also quite costly in terms of computation and runs considerably slower than native execution. For example, GEM5 at its highest level of precision requires, on average, one year to run a single

[1] The term "counter" can be misleading since modern processors do much more than just counting events.

SPEC2006 benchmark [180]. A common approach to speed up simulations is to only simulate parts of a program [188, 224] or to only simulate some aspect of the processor [36, 110, 152].

Simulators based on abstract high-level CPU core models, while not as accurate as a real cycle-level simulator, have been shown to provide reasonable accurate execution time predictions while running orders of magnitude faster than cycle-level simulation systems [68, 76, 202, 37].

At the end of the day, the data gathered using profiling, either using hardware counters or simulation, is still very low-level, recording information at the level of individual machine instructions. Tools like HPCToolkit [4], Intel VTune [169], and Linux perf [144] can map these results back to the source code of a program using debug information, making it possible to view aggregate results per loop or function. However, these tools still do not directly explain why a given piece of code has a performance problem, such as too many cache misses or stalled CPU cycles, or how to fix the problem. Finding and fixing the root causes of a performance problem often still require substantial expertise.

Feedback-Directed Optimization (FDO) takes one step towards the direction of using profiling information to improve performance [192, 11, 12]. FDO based systems collect runtime information using profiling and use it to guide optimization heuristics in compilers. Tools like, Intel Advisor [52] and AutoSCOPE [195], on the other hand, try to give human-readable optimization advice. They are, however, not able to automatically apply code transformations, or even necessarily prove that a transformation is valid. Instead, they propose optimizations such as parallelization or loop fusion, leaving the task of implementing this transformation to the programmer.

At this point, we have only given a brief overview of basic profiling techniques. Chapters 3 and 4 will present a number of more involved approaches using richer, but expensive to collect, performance events.

1.2.2 Instrumentation and dynamic binary analysis

Since profilers analyse events that occur during the execution of a program, they need some facility to trace these events. Sampling of hardware performance counters is an excellent low overhead method for collecting performance-related events, but it suffers from a number of limitations, as described in the previous section.

An alternative technique used by a large number of dynamic analysis tools, including those presented in this thesis, is instrumentation. Instrumentation, is a very versatile technique that allows capturing arbitrary program behaviour. Using instrumentation a profiler can implement any new performance counter it may require in software. Besides profiling, instrumentation is used in numerous other dynamic analyses such as dynamic control flow integrity [1] and memory corruption checking [185].

There are some new hardware-assisted program tracing technologies, such as Intel PT [51], on the horizon that may replace instrumentation for some profiling use cases. However, for the moment the available hardware is still somewhat unreliable and can lose events due to internal buffer overflows [75, 101].

Compiler based instrumentation, used by highly successful dynamic analysis tools such GPROF [78] and the LLVM Address Sanitizer [185], works directly in the compiler at the level of its intermediate representation (IR). Instrumenting programs during the compilation process has several advantages. In terms of productivity, the whole compiler infrastructure makes the task of implementing some instrumentation for profiling easy. Instrumentation can easily be performed at any level of the intermediate representation, allowing the profiling of high-level and low-level properties. It can also take advantage of all its knowledge of the program semantics and information provided by the diverse static analysis present in an optimizing compiler. In particular, any property that can be statically proven or is not data-dependent, does not have

to be profiled.

Binary instrumentation [196, 123, 134, 155, 134, 29, 145, 146], on the other hand, works directly at the level of machine code. Analysing and manipulating binaries is difficult and tedious, since, all too often, one has to re-discover the obvious. Because machine code contains much less structure than source code even relatively simple analyses needed for instrumentation, such as reconstructing the control-flow graph, become non-trivial or undecidable [145, 170]. However, working at the level of machine code has the important advantage of portability. That is, tools that directly handle machine code are not restricted to working with a given compiler, or programming language. It also makes it possible to compare the binaries produced by different compilers. If, for example, ICC turns out to generate the best performing code for an Intel platform, it is clearly interesting to be able to analyse this binary and not be limited to the output of GCC [80] or CLANG [164]. Binary instrumentation even makes it possible to analyse programs for which no source code is available or which integrate closed source third-party libraries.

Another approach that can be used to implement static binary instrumentation is decompilation. SecondWrite [8] and DisIRer [104], for example, are two tools that convert machine code to LLVM IR and GCC IR respectively. These compiler IRs are semantically much richer than raw machine code, and there are a variety of existing tools that can be used to analyse and transform them. The advantage of this approach is that it has the portability of a binary analysis tool combined with ease of use of compiler-based instrumentation. However, it is only applicable to programs that can be statically decompiled.

There are also dynamic binary analysis frameworks that decompile machine code to compiler IR at runtime [64]. However, while this makes it possible to apply the powerful analyses available in a static compiler to arbitrary binaries, it also suffers from some drawbacks. The main problem is that static compilers can afford to invest much more time in analysing and optimizing programs than dynamic compilers. Due to this, authors of static compiler IRs and analyses often prioritize expressiveness and maintainability over performance. A number of failed attempts to re-use static compiler back-ends in a JIT have shown that this can lead to a prohibitive overhead at runtime [49, 50].

Dynamic binary instrumentation and the applications we have used it for will be described in detail in Chapter 2.

1.2.3 The polyhedral model

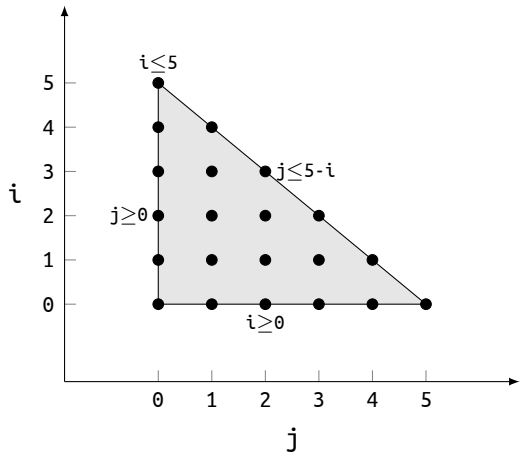
The most effective program optimizations for improving performance or energy consumption are typically based on the rescheduling of instructions to expose data locality, parallelism or both. The two main challenges for these kinds of optimizations are *what* transformations may be applied without breaking the program semantics and *where* in the program the optimizations should be applied to maximize the impact on the overall program performance. The *polyhedral model* [71], a mathematical model that allows reasoning about programs at the levels of loops, is one of the most powerful theoretical frameworks for finding and for applying such rescheduling optimizations.

The polyhedral model is usually applied in the context of static compilers [160, 211, 81, 162, 25, 218]. Here it is used to find sequences of loop transformations, including tiling, permutation, fusion, fission, and skewing, that aim to improve the temporal and spatial locality of data accesses and uncover both coarse (i.e., thread) and fine-grain (i.e., SIMD) parallelism. To determine the validity and profitability of these transformations, however, the polyhedral model needs precise information about data and control-flow dependencies. That is, it needs a precise model of the structure of loops and of the dependencies between statements.

The power of the polyhedral model comes, in part, from the fact that it does not try to model

```
for (i = 0; i <= 5; i++)
  for (j = 0; j <= 5 - i; j++)
    S: A[i][j] = B[i][j] + C[i] * D[j];
```

(a) An simple loop nest with one statement S.



(b) The 2D iteration space for statement S.

Figure 1.2 – An example program and its iteration space.

general programs. Instead, it focusses on regular loop nests operating on multi-dimensional arrays. Here, loop nests are represented by a finite Cartesian space, called the *iteration space* [223]. In this space, each level of the loop nest gives rise to one dimension. Every loop iteration then corresponds to a point in this iteration space whose coordinates are determined by the current values of the iteration variables of the enclosing loops. The polyhedral model requires that the bounds of every loop can be represented by a linear expression of the loop indices. Thanks to this restriction the iteration space can also be represented as an integer polyhedron, i.e., a set of points bounded by a set of linear constraints over the integers. Figure 1.2 shows a simple 2D loop nest and the corresponding iteration space. In this simple example, there is only one loop nest and only one statement and consequently also only one iteration space. In general, however, every statement in a program has its own iteration space. The polyhedral model also supports conditional statements, but conditionals are restricted to linear expressions in order to keep the iteration spaces representable. The expressions used to compute array indices, called *access functions*, are also restricted to linear functions.

Since every execution of a statement, i.e., every *dynamic instance*, is represented as a point in an iteration space the polyhedral model also uses linear relations to model data dependencies. Consider the following program:

```
for (i = 1; i < 5; i++)
  S: A[i] = A[i-1] + i;
```

Here every instance of the statement S depends on a previous instance of S. We note the instance of S executed in the ith iteration of the loop as $S[i]$. This dependence can then be expressed by the following relation (for simplicity, we omit the initialization of A and border conditions):

$$\{S[i] \to S[i'] : \ i' = i - 1 \wedge 2 < i < 5\}$$

Despite this apparent simplicity, the polyhedral model is still quite expressive, but this also comes with a cost. Finding a set of optimizing loop transformations in the polyhedral model requires solving one ore more integer programs. A big strength of the polyhedral model is that the complexity of these problems is independent of the number of points in the iteration space.

Rather, the complexity is exponential in the number of different loop dimensions, statements and dependencies. As a consequence, polyhedral compilers can, in practice, only handle programs consisting of no more than a few hundred instructions.

In a compiler, the static analyses required to build this representation can only be applied to programs written in a very restrictive style with no function calls, simple control-flow and no pointer indirection [48, 62]. While there have been attempts to extend the polyhedral model to work on more general programs [197, 18] these have only met with limited success. In practice, one quickly runs into problems with pointer aliasing and pointer indirection, which make it impossible to calculate access functions statically. This, in turn, makes it impossible to build the data dependence relation. The other big hurdle one often runs into programs with complex or data-dependent loop bounds and conditionals.

To overcome these limitations, compilers have started using *hybrid analysis*, which leverages information only available at runtime to allow for more aggressive optimization. Using code versioning combined with runtime checks for the validity of a transformation allows optimizing programs even when the optimization statically cannot be proven to be legal or is not even legal for every possible program execution [177, 178, 6, 62].

The Apollo [139] and PolyJIT [190] projects push the envelope even further bringing the polyhedral model to the world of JIT compilers. They are dynamic polyhedral optimizers that both find and apply polyhedral loop transformations at runtime. Apollo is even able to speculatively execute optimistically optimized code, rolling back the execution if and when any non-affine behaviour occurs.

1.3 Objectives

A large part of today's performance-sensitive code is written in sequential imperative languages such as C, C++ and Fortran. There have been many attempts to extend or replace these languages and programming models, but for the foreseeable future, they will still dominate the field. In this work, we present tools that work on these kinds of programs. They allow analysing and modelling the performance of existing systems to help guide programmers improve their code to get better performance.

In this thesis, we focus on analysing and modelling the performance of relatively small compute-intensive scientific kernels. We focus on the computational and memory behaviour of programs and do not investigate aspects like network traffic or hard disk usage. All tools and techniques we present work at the level of machine code on compiled and optimised binary programs.

We propose the following two performance debugging techniques: *sensitivity based bottleneck analysis* and *data-dependence profiling driven optimization feedback*.

1.3.1 Sensitivity Based Performance Bottleneck Analysis

Performance Bottleneck analysis Even answering a seemingly trivial question about a program's performance, such as whether it is memory-bound or CPU-bound can be surprisingly difficult. This is because the CPU and memory are not simply two completely independent resources that a program uses independently. Instead, they are composed of multiple complex interdependent subsystems. Here a stall of one resource can both mask or aggravate problems with another resource. It is even more complicated to quantify by "how much" a resource is a bottleneck, that is, by how much performance could be improved if that one bottleneck is removed.

We propose a *bottleneck analysis* that is based on *sensitivity analysis* or *differential profiling* [141, 47, 118, 132, 99]. A conceptually simple approach to finding bottlenecks that works as

follows: One runs the program under observation multiple times, each time varying the capacity of one or more resources. The bottleneck is then the resource whose change in capacity has affected the overall change in performance the most.

A real-world example of this approach uses dynamic voltage and frequency scaling (DVFS) [100]. Here, a program or parts of it are executed repeatedly, every time with a different CPU frequency. Note that the frequency at which the CPU operates does not affect the speed at which memory works. So, if a program finishes in roughly the same time while running at a lower frequency as when running at a high frequency it is clearly not CPU-bound. Reversely, if execution time is significantly affected by CPU frequency the program is CPU-bound. However, DVFS only works at a relatively high granularity and only allows detecting the presence or absence of a general CPU bottleneck. Unfortunately, real hardware does not directly provide many other "levers" that one can use to easily vary the capacity of resources. It is, for example, not possible to reduce the throughput of the vector unit to see by how much this affects performance.

Performance model To allow for a more fine-grained sensitivity analysis we have developed an abstract, resource-based, performance model. We have built Gus, a prototypical implementation of this model in a CPU emulator to make it possible to apply it to real programs. Gus can arbitrarily change the capacity of every resource independently making it a powerful tool for pinpointing performance bottlenecks.

Our performance model needs a baseline, a detailed specification of the expected performance of different operations on a CPU. That is, we need to know how many instructions of a given type the processor can execute per cycle, how many cycles instructions require to finish, and how instructions interact with each other. Unfortunately, hardware vendors like Intel or AMD are not very forthcoming with this kind of information and do not publish comprehensive datasheets. To build our processor model we have developed a system to reverse-engineer the required information using automatically generated micro-benchmarks.

1.3.2 Dependence Profiling Driven Polyhedral Optimization Feedback

The holy grail of performance debugging is to directly tell a user how they can change their code to improve performance. The most effective program transformations for improving performance or energy consumption are typically based on the rescheduling of instructions so as improve temporal and spatial locality. This can be used to uncover parallelism at both coarse, i.e., thread-level, and fine, i.e., vector-level, granularity. The two main challenges when using these kinds of optimizations are *what* transformations may be applied without breaking the program semantics and *where* in the program should the optimizations be applied to maximize the impact on the overall program performance.

The *polyhedral model* is a powerful framework for finding these and other kinds of optimizing transformations. The polyhedral model requires an exact description of both the control-flow of a program and the structure of its data dependencies. Historically, the polyhedral model has been designed to work on a restricted set of programs, called *affine programs*, for which it is easy to statically reconstruct exact dependence information [48]. While programs can be rewritten to fit these restrictions and help static analysis tasks, this requires significant effort and is not guaranteed to succeed. In full programs, and in particular those relying on external binaries visible only to the linker, often the data layout and calling context is inaccessible to static analyses.

Dynamic approaches [139, 190] push this boundary and handle programs which, even though they cannot be statically proven to fit the polyhedral model, do adhere to its restrictions at runtime. However, even tools that use dynamic information to build a polyhedral representation

still face a number of problems. The main difficulty being how to model applications that include some non-affine dependencies and memory accesses in otherwise affine code To address this difficulty existing dynamic polyhedral approaches either use overly pessimistic approximations and lose information [199] or do not scale [112, 171].

Data-dependence driven polyhedral optimization feedback We have developed MICKEY, a dynamic data-dependence profiler that builds a rich polyhedral IR from traces. MICKEY works on compiled binaries and provides feedback on the applicability and profitability of polyhedral loop transformations. It can scale to real-life applications thanks to a safe, selective over-approximation mechanism for partially irregular data dependencies and iteration spaces.

Polyhedral interprocedural IR MICKEY's polyhedral IR encodes both the dynamic execution of a program's instructions as well as its data dependencies. This IR captures the interprocedural loop nesting structure of a program in a form that is amenable to analysis by a polyhedral optimizer. It not only captures data dependencies across multiple loops but also, possibly recursive, procedure calls.

Linear time polyhedral trace compression To construct the polyhedral IR from a program's execution, we have developed the folding algorithm, an efficient trace compression algorithm. This technique was inspired by existing polyhedral trace compression techniques [112, 171], with the notable difference that it uses the geometry of iteration spaces which allows it to scale to large irregular programs. The folding algorithm also detects which instructions in a program are used to increment loop counters and finds strides in memory accesses by building scalar evolution expressions (SCEVs) [161, 213] for loads, stores and integer arithmetic instructions.

1.3.3 Contributions

To summarize, in this thesis we present the following contributions:

- A sensitivity based analysis tool to detect performance bottlenecks in programs (Chapter 3).
- A tool to reverse engineer performance characteristics of instructions on modern x86 CPUs (Section 3.5).
- A data-dependence based profiling tool that provides high-level optimization feedback (Chapter 4).
- An interprocedural polyhedral intermediate representation (Section 4.4).
- A linear time polyhedral trace compression algorithm to construct this representation (Section 4.5.5).

1.4 Structure of the Thesis

This thesis consists of four main chapters and is organized as follows:

- Chapter 2 gives a technical description of the dynamic binary instrumentation platform we developed for our work. It contains mostly technical details and can be skipped on a first read.
- Chapter 3 describes the bottleneck analysis algorithm and the CPU performance reverse-engineering tool.
- Chapter 4 presents our data dependence driven tool for giving optimization feedback.
- Finally, Chapter 5 concludes the thesis, giving a summary of the contributions and perspectives on possible future work.

Chapter 2

Dynamic Binary Instrumentation with QEMU

Contents

> Preliminary note: this chapter is very technical and describes the internals of some analyses on which techniques presented later in the thesis are based. This chapter does not necessarily present any novel approaches by itself, but gives the general technical context of the thesis. As a consequence, it is possible to skip it at first reading.

This chapter describes the internals of the QEMU CPU emulator and TCG plugin infrastructure (TPI), an infrastructure for building dynamic binary program analysis tools based on QEMU. It also gives a general overview over the field of dynamic binary translation and also show cases a number of analyses we have implemented using TPI. The chapter is intended to both show how TPI works in practice as well as to present some basic techniques which are used later throughput the thesis.

2.1 QEMU

QEMU [17, 166] is a processor emulator based on dynamic binary translation. That is, it allows running programs written for one CPU architecture on a different CPU architecture. QEMU only reproduces the functional aspects of a CPU, meaning it emulates the instruction set architecture (ISA). It does not, however, match the non-functional microarchitectural details of an architecture such as timing, cache effects, or speculative execution.

Vanilla QEMU only allows running programs on a different CPU architecture than the one they where written for. ONe can, for example, run a program compiled for a 64 bit x86 CPU on a 32 bit ARM computer, and vice versa. The machine code of the emulated program is translated and run as native machine code for the architecture QEMU is running on. This makes emulation relatively fast compared to interpreter based systems like GEM5 [23].

For the purposes of this work we use a modified version of QEMU [84] that includes TPI which makes it possible to alter programs as they execute. We chose to use QEMU for this since it can emulate a plethora of different CPU architectures and supports all major operating systems (OSs), including GNU/Linux, *BSD, Mac OS X and Windows. QEMU furthermore has a simple, more or less architecture independent, intermediate representation (tiny code generator (TCG) IR). This makes it easy to write analysis tools that work at the level of machine code but are still portable across different system. Details of how this works will be presented in Section 2.3, for now we will focus on presenting QEMU itself.

QEMU internally uses the following terminology, which we will reuse:

- *Host CPU/OS*: the CPU and OS on which QEMU is running.
- *Guest program/OS*: the program/OS you run inside QEMU.
- *Guest CPU* or *virtual CPU*: The state of a CPU core as emulated by QEMU.

QEMU can be run under two different modes: full system mode and user mode. Full system mode emulates a whole computer, including the CPU and peripherals such as network cards or hard disks. This means a whole operating system, including a kernel, is running on the simulated machine inside QEMU. This is similar to what sytems like VirtualBox [57] or Bochs [126] provide, but they can only run x86 programs on x86 hosts. User mode runs a single user space process and translates the machine code from the guest ISA to the host ISA. Under this mode QEMU runs a program compiled for one CPU architecture as a process in an operating system for another architecture. To make this possible QEMU translates system calls to the ABI of the host kernel. All the work done in this thesis has used QEMU in user mode, but it could easily be modified to support full system mode.

2.2 The Tiny Code Generator

The TCG is the component of QEMU responsible for translating guest machine instructions to host machine instructions. TCG is essentially a just-in-time compiler (JIT) compiler that takes real machine code as input instead of byte code.

Since TCG is a JIT it translates a program piece by piece at runtime. As a consequence the execution of code and its compilation are interleaved. That is, when a program tries to run a sequence of guest instructions the emulated guest CPU is halted and TCG translates the code to host instructions. Once TCG has finished compiling the emulation resumes by executing the new native code on the host. In the following we will refer to the moments when emulation is halted and TCG takes over as *translation time* and the moments when the code of the emulated program is running as *execution time*.

TCG is essentially a trace based JIT [150]. That is to say, it does not translate whole programs or functions at a time but only works on small linear sequences of guest instructions called *trace blocks*. These trace blocks are formed from the guest program using a very simple mechanism. Whenever the guest CPU executes an instruction that has not been translated yet a new trace block is started. This first instruction of the trace block is referred to as its *entry*. QEMU then parses the guest instructions following the entry until it hits a branch instruction [1]. The final trace block then contains the entry, the terminating branch and all instructions in between. Note that instructions in the guest program that are never executed are also never translated by TCG.

Since we are using QEMU in user mode kernel space instructions are not visible to TCG. When guest code performs a system call this is translated to a corresponding system call on the host system. Instructions that perform a system call, i.e. `sysenter` and `syscall` on x86, still are considered trace block terminators. They do not, however, change the program counter of the virtual CPU and the emulated execution simply falls through to the next guest instruction once the system call has finished.

Due to the way trace blocks are constructed they are very similar the basic blocks of the guest binary. That is

- a trace block has only one entry,
- a trace block has only one exit, [2]
- and consequently once a trace block has been entered all instructions in it are executed exactly once, in order.

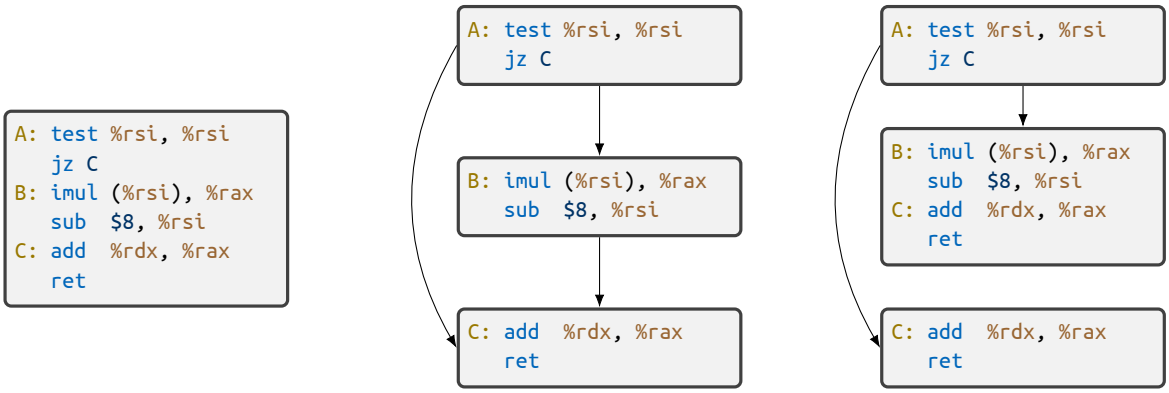
The important difference to the original guest basic blocks is that trace blocks can overlap, i.e. that a given guest instruction can be contained in multiple trace blocks. This happens for example if guest code branches to an instruction which is in the middle of another trace block that has already been translated. The only restriction being that an instruction can only be the entry for one trace block. An illustrative example showing how a piece of code is divided into basic blocks and trace blocks can be seen in Figure 2.1.

Internally, TCG does not directly translate the guest instructions in a trace block to host instructions. Instead it first translates traces to a more machine independent IR. Finally, the IR is then translated to machine instructions that can be executed directly on the host. The sequence instructions generated for a whole trace block is called a translation block (TB). This process is illustrated in Figure 2.2a.

TCG IR consists of simple RISC-like instructions working on registers and memory addresses. The IR distinguishes between registers of the simulated guest CPU and virtual registers in the

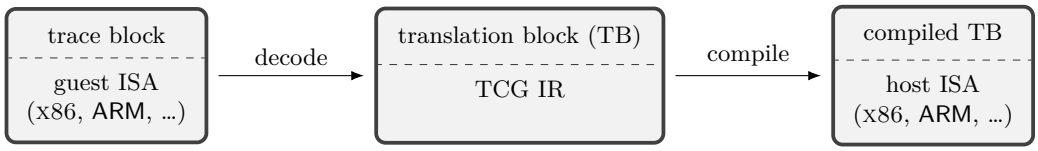
[1] QEMU also terminates trace blocks before it finds a branch if the trace block is too long or if the guest instructions would span two pages in memory

[2] like many other compilers QEMU is actually not very strict about this. Instructions that can fault, like memory accesses or division, are not considered block terminators.

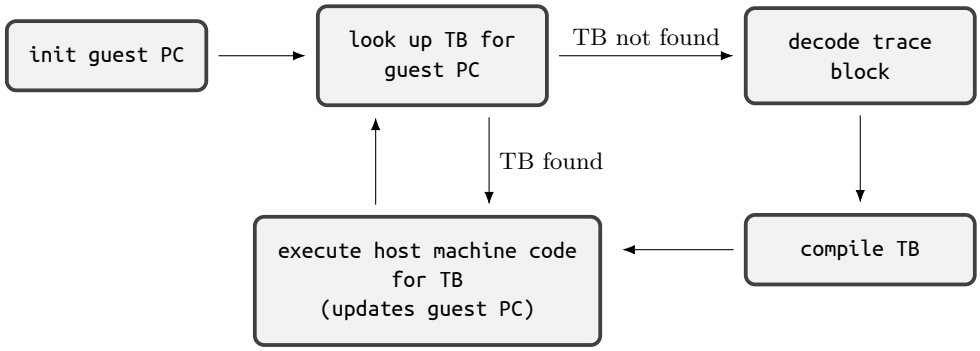


(a) original program (b) basic blocks (c) trace blocks

Figure 2.1 – A piece of x86 assembler in AT&T syntax and how it is divided into basic blocks and trace blocks. Note that `B:` and `C` are split into two basic blocks since `C` is the target of a branch. However, `B` and `C` form a single trace block since `B` does not end in a branch. There is a separate trace block containing only `C` which is created when the branch at the end of `A` is taken.



(a) Illustration of how TCG translates guest code to host code.



(b) Main emulator loop of QEMU.

Figure 2.2 – High-level view of CPU emulation with QEMU.

IR used to hold intermediate values. It also uses distinct instructions to access memory of the guest CPU and the host memory used for QEMU's internal data structures. There are likewise distinct opcodes for branches inside a TB and for branches that leave the current TB.

TCG IR is typed using a very simple type system. The only types are 32 and 64 bit integers and pointers. Very recently, 128 and 256 bit vector types where added too. But notably there are no types to represent floating point values.

During emulation the state of the guest CPU is represented as an object in the heap of the QEMU process. For example, an x86 CPU has 16 general purpose 64 bit registers and so the CPU object for x86 guests contains an array with 16 64 bit integers. An excerpt of the definition of the structure used to represent x86 guest CPUs can be seen in Figure 2.3. Since the registers of the guest CPU reside in host memory every use of a register in a guest program is translated to a host memory access reading a field of the guest CPU object. Likewise, a write to

```
struct CPUX86State {
    /* standard registers */
    unsigned long regs[16]; // register %rax at offset 0
                            // register %rdx at offset 0x10
                            // ...

    unsigned long eip;      // program counter at offset 0x80
    unsigned long eflags;
    ...
};
```

Figure 2.3 – C structure representing the state of an emulated x86 CPU in QEMU.

a guest register becomes a host memory write. An example of how TCG translates a simple x86 instruction that illustrates all this can be seen in Figure 2.4. To reduce the overhead introduced by these memory accesses TCG actually avoids inserting redundant loads and stores. Inside a TB it keeps the contents of guest CPU registers in host registers as much as possible and only flushes them back to host memory at the end of the block. This optimization is especially important for flag registers, like x86 condition codes, which are updated by most instructions but only read by a few.

TCGs IR is very simple, thus even translating a single guest machine instruction generates multiple IR instructions. Since real machine instructions can have quite complex behaviour the IR instructions for a single machine instruction can be a control flow graph (CFG) with loops and conditional branches. Fortunately, the CFG of the IR generated for a single guest instruction still is relatively simple though and follows some restrictions. That is, it only has one entry, and for any instruction, except for some branch instructions, it only has one exit. While TCG does perform some simple optimizations, such as simple dead code elimination and copy propagation passes, to clean up the IR before translating it, it is not an optimizing compiler.

Some complex instructions, such as those for handling system calls or exceptions, are not implemented directly as TCG IR instructions. Instead they are translated to calls to short functions implemented in C called helper functions

Since TCG IR does not have any floating-point types or instructions all guest machine instructions working with floating point numbers are translated to calls to helper functions written in C. Obviously these helper functions cannot directly accept floating-point values as arguments, since TCG IR has no way to represent them. Instead their arguments are passed to them as pointers to the host memory representing the guest CPU floating-point registers as arguments. This can be a source of significant performance overhead when running floating-point heavy programs in QEMU.

Quite recently vector types and instructions working on them have been added to TCGs IR. The current version of these instructions are relatively limited and only allow working with vectors of integer values. TCG instructions to accesses guest memory are also still limited to at most 64 bits. Consequently all guest vector loads and stores are split up into multiple load/store instructions in the IR. For the moment vector IR is not yet widely used throughout QEMU. In fact, only the ARM guest currently generates vector IR instructions for some vector guest instructions. Guest vector instructions of other platforms are still translated either to sequences of multiple scalar IR instructions or to calls of helper functions.

As shown in Figure 2.2b once a TB has finished executing in returns to the main emulator loop, which then looks up the next TB to run. To avoid the overhead of this lookup for every branch in the guest program, QEMU performs a dynamic optimization called *block chaining*.

When QEMU dynamically detects that a TB always branches to the same successor it patches a jump instruction into the host machine code for the TB. So, instead of exiting to the main emulator loop the first TB will directly branch to the code of the successor TB. This optimization allows QEMU to spend most of its time in generated machine code and to only exit to the main loop to translate new trace blocks or handle special events like interrupts

In user mode QEMU has spawns one OS thread per thread in the guest program where each OS thread has its own virtual CPU object. Therefore, the guest program can truly execute code concurrently. To ensure that the memory ordering rules of the guest ISA are respected by the host processor TCG inserts special memory barrier instructions wherever necessary [58]. Due to the way threads are emulated a guest CPU will have has at most as many cores running simultaneously as the host CPU. An accurate simulation of a highly parallel guest architecture on a less parallel host is therefore not possible.

Examples of how QEMU translates some simple x86 instructions to TCG IR and then back to x86 code are shown in Figures 2.4, 2.5 and 2.6.

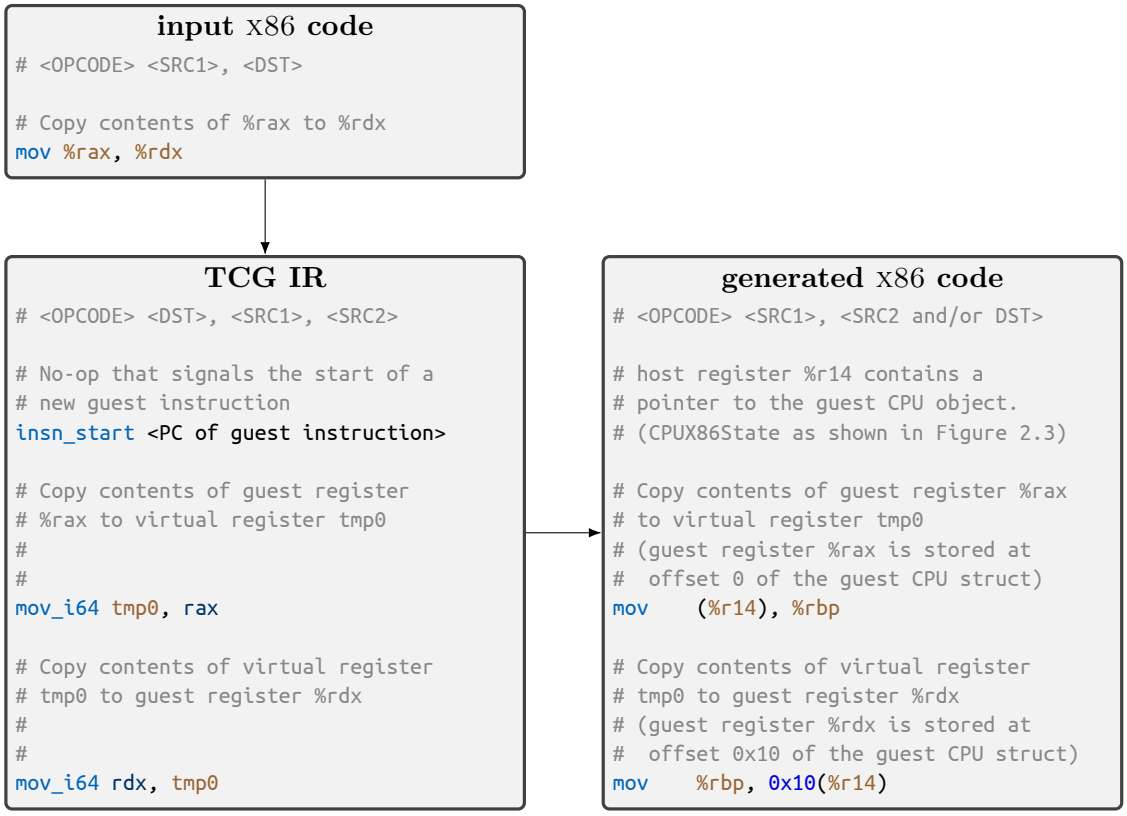


Figure 2.4 – TCG IR and x86 machine code generated to emulate an x86 `mov` instruction. Note that the operand order in x86 and TCG IR is reversed.

2.3 The TCG Plugin Infrastructure

As mentioned before the work presented in this thesis is not based on the main version of QEMU, but on a modified version [84, 85]. This modified QEMU has a mechanism for loading plugins, called the TCG plugin infrastructure (TPI), that can both observe and alter the translation process of TCG. To instrument a binary a plugin injects TCG IR instructions into TBs before they get translated back to host machine code. Since plugins operate at the IR level and not at the machine code level it is relatively easy to port plugins so they work with different guest

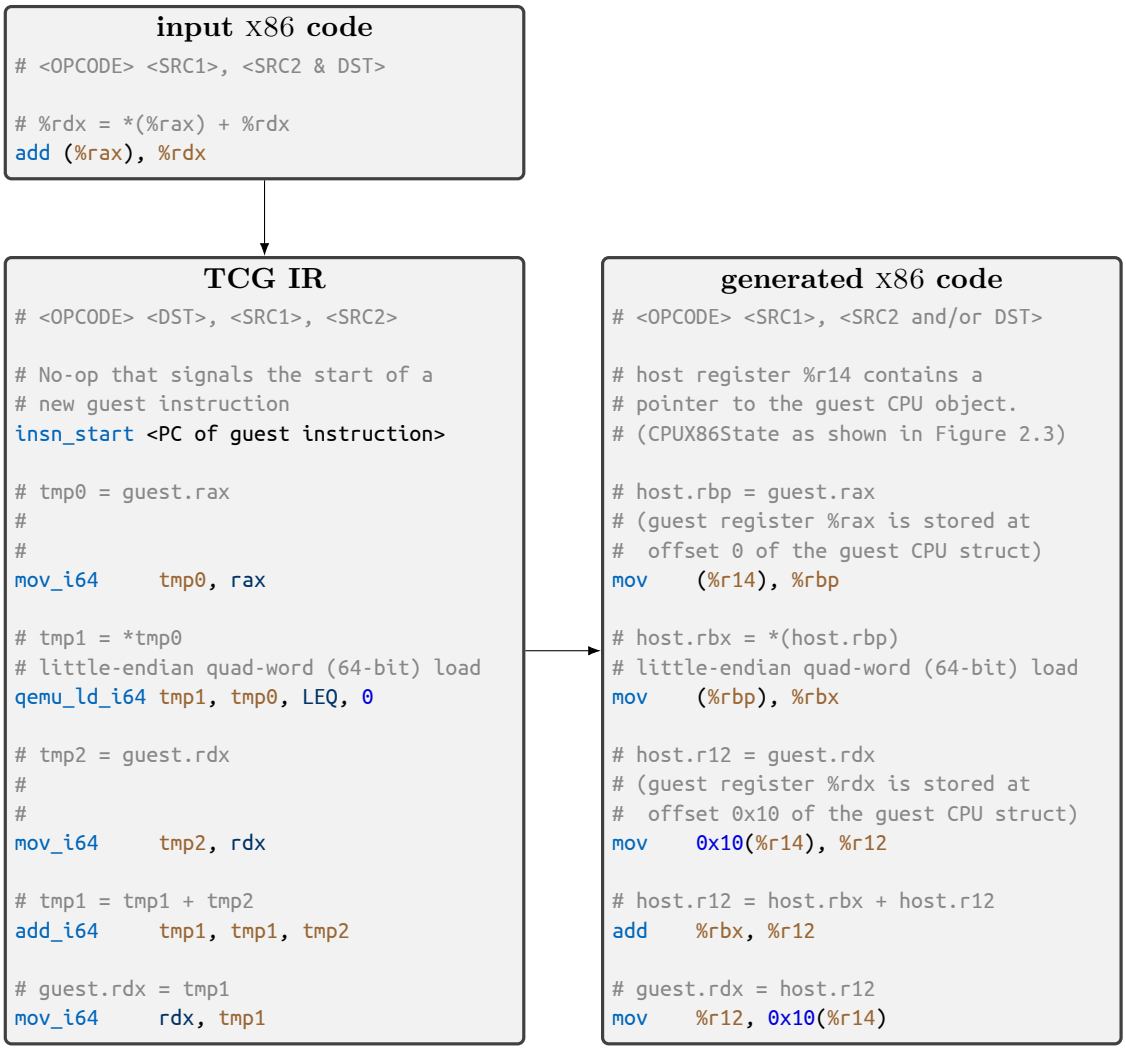


Figure 2.5 – TCG IR and x86 machine code generated to emulate an x86 `add` instruction (the IR to update the guest status flag register is omitted). Note that the operand order in x86 and TCG IR is reversed.

CPU architectures. Simple plugins whose behaviour does not depend on the exact semantics of guest instructions can work with multiple different guest ISA without any modification.

Even though a plugin can insert arbitrary code into the guest program the bulk of our instrumentation is not in TCG IR, but in C and C++. Instead of generating large amounts of IR we insert only small snippets that call the instrumentation functions. Even though this incurs some overhead we have chosen this approach to speed up the development process, since it is much easier to write in C/C++ than in the low level TCG IR.

Certain features of the design of QEMU and TCG incidentally work together to ease the development of plugins. TCG IR enforces a certain separation between host memory and guest memory, since it uses different opcodes to work with the one or the other. This means that the IR emitted can freely use separate virtual registers and does not have to take care not to accidentally clobber resources of the guest program. Guest programs also have no way of accessing the registers and memory of the plugin, meaning a faulty guest cannot cause faults in the emulator. Furthermore, temporary registers in TCG IR are distinct from guest machine registers, meaning one can freely insert code at any place without having to spill or restore values in and out of memory explicitly. TCGs register allocator automatically takes care of this.

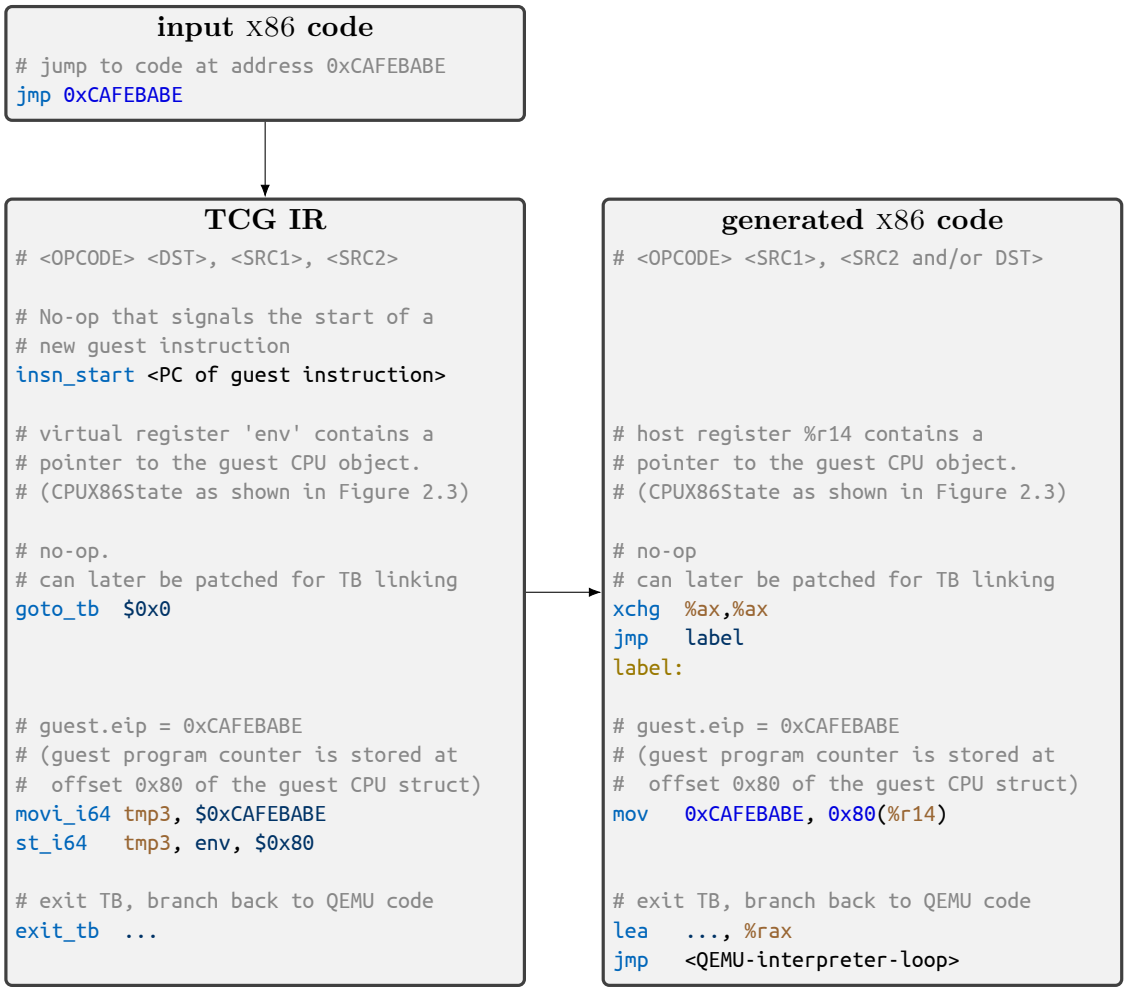


Figure 2.6 – TCG IR and x86 machine code generated to emulate a x86 `jmp` instruction. Note that the operand order in x86 and TCG IR is reversed.

A TPI plugin is simply a shared library which QEMU loads into memory on startup, before it even starts emulating the guest program. TPI offers a C API allowing plugins to register callback functions which are then called whenever certain events occur. Most of these events are related to the translation of guest instructions to TCG IR. There are, for example, events that signal that TCG has started or finished constructing a new TB, and also events that signal that TCG has started or finished decoding a guest machine instruction. Whenever such a callback is called the plugin can then choose to insert new IR instructions into the TB, or alter the existing ones. A short excerpt of the plugin API is shown in Figure 2.7

TCG decodes trace blocks, i.e., it generates all IR instructions in a TB, in one pass. Virtual registers are not in static single assignment (SSA) form [174], and instead are reused as much as possible to reduce memory consumption of TCG itself during translation. Consequently, a TPI plugin written using only callbacks must emit its instrumentation directly as TCG emits its IR or take care not to clobber virtual registers. As TPI only provides this event based API instrumentation plugins have to be written as a state machine or in continuation passing style.

Since TCG IR generates multiple IR instructions even for the simplest guest instruction writing plugins that maintain state across a TB can quickly become quite tedious. To make writing plugins easier we have developed a thin C++ API wrapper around the original TPI API that allows working with TCG IR in a more straightforward fashion. It allows plugins to work

```
struct TPIOpCode;         // a single TCG IR instruction
struct Translation_Block;

/// Object representing the state of a plugin.
/// Contains callbacks that will be called by TPI.
struct TCGPluginInterface {
  // Called whenever TCG is about to start decoding a new trace block.
  // IR for trace will be stored in 'tb'.
  void (*before_decode_first_instr)(const TCGPluginInterface *tpi,
                                    const TranslationBlock *tb);

  /// Called whenever TCG has finished decoding a trace block.
  void (*after_decode_last_instr)(const TCGPluginInterface *tpi,
                                  const TranslationBlock *tb);

  /// Called whenever TCG is about to decode the next instruction of
  /// the current trace block.
  void (*before_decode_instr)(const TCGPluginInterface *tpi, uint64_t pc);

  /// Called whenever TCG has inserted a new IR instruction into
  /// the current translation block.
  void (*after_gen_opc)(const TCGPluginInterface *tpi, const TPIOpCode *opcode);

  ...
};
```

Figure 2.7 – Core events of the TPI C API

with the IR for an entire TB at once and to perform multiple passes over the IR and to insert code at any point in the TB. We have also wrapped TCGs routines for allocating and freeing virtual registers so a plugin author does not have to worry about overwriting registers used by the guest code itself.

TCG IR has control flow but does not directly represent basic blocks, it just has `set_label` and `branch to label` instructions. Our IR layer also detects the basic blocks in a TB and makes it easy to iterate over successors and predecessors. However, the C++ TPI wrapper currently does not allow altering the control flow of a TB, i.e. it cannot change jump targets and also cannot insert or remove jumps.

2.4 Execution Tracing with QEMU

During this thesis we have developed different tools to perform high-level analyses on Executable and Linkable Format (ELF) [206] binaries. The input of these tools are traces of events describing different aspects of the execution of a program. This section describes three TPI plugins we have developed to trace a program's control-flow, data-dependencies and integer computations.

On a side note, we want to point out that execution traces of real world programs can become extremely large. So large, in fact, that realistically we cannot store entire traces in files. This can be illustrated with a little thought experiment. A modern CPU runs at a frequency of several GHz, meaning every second it can execute billions of instructions. When tracing a small program that only runs for one second if one retains even only one byte of data for every instruction that is executed the resulting raw trace will already contain more than one gigabyte of data. Consequently, all plugins presented here are designed as libraries that produce streams of events which can be treated directly in memory. If necessary, for example for debugging or

verification purposes, partial traces can still be stored to a file.

2.4.1 Control-flow tracing

The control-flow tracing plugin we have developed observes the branches executed in a program and has the goal of distinguishing local control-flow inside a function and calls and returns between functions. This information can in turn be used to:

- Recover the interprocedural call graph (CG) and intraprocedural control flow graph (CFG) of a program and
- track the *calling-context*, that is a compact representation of the call stack, during execution.

Since portability, across CPU architectures as well as across languages, was one of our main concerns we took great efforts to make our plugin independent of platform and language ABIs. It does not inspect the call stack and only uses basic assumptions about branch instructions of a given architecture. This approach is also robust against of software stack unwinders as used, for example, for exception handling in C++.

The main problem with control-flow tracing is that, as mentioned earlier, binary machine code is much less structured than source code. The plethora of high-level control flow found in programs, like loops, conditionals, function calls, returns and exception handling, are all reduced down to a small number of branch instructions. On x86, for instance, all normal control flow is implemented using only `call`, `ret`, `jmp` and `jcc`. Hence, the machine code conflates at least some different types of control flow by implementing them using the same instructions. Compilers can also sometimes use different machine instructions to implement the same control flow construct depending on they context are used in, e.g., replacing `call`s with a cheaper `jmp` where possible. This means our plugin has to be able to detect the intended purpose of a branch instruction from the instructions immediately surrounding it and branch target.

Since the plugin needs to be able to distinguish different forms of guest branch instructions it cannot work solely at the level of TCG IR. To make it easier to port the plugin between different architectures the main part of the code works does not work directly with guest opcodes. Instead we map the different guest branch instructions to a reasonably small set of abstract *terminator types* shown in Figure 2.10. Thanks to this only a small part of the plugin that is responsible for categorizing guest instructions is architecture dependent. The majority of the code and also all the instrumentation we generate does not need to be changed to support new platforms.

In practice, local branches inside functions are mostly regular and easy to handle. The main problems arise when trying to handle calls and returns between functions. For example, the "normal" way to translate a function call in C is using the `call` instruction. But this is by far not only way that function calls are actually implemented. The most commonly occurring alternative implementation, tail call optimization, which has already been mentioned, uses `jmp` instructions instead, which are normally used for control-flow inside a function. But there are also other, more exotic, ways to implement function calls. In highly hand optimized assembly code, as can be found in the GNU C standard library, there are cases where a function call occurs via fall-through, i.e., without any branch instructions. An example of this can be seen in Figure 2.8. Another reason why interprocedural control flow is more difficult to handle is that in modern code bases using virtual function dispatch and function pointers or lambda expressions indirect branches are much more common. In general it is not possible to statically detect the target of such indirect branches meaning we have no information about the target of the branch before actually executing the branch.

There are also cases where `call` instructions are used for other purposes than function calls. On pre 64 bit x86 CPUs there is no direct support for RIP relative addressing, which is necessary for writing position independent code. One idiom used to work around this is show in Figure 2.9.

```
0000000000149cd0 <__strcasecmp_ssse3>:
  149cd0:       f3 0f 1e fa             endbr64
  149cd4:       48 8b 05 e5 60 07 00    mov    0x760e5(%rip), %rax
  149cdb:       64 48 8b 10             mov    %fs:(%rax), %rdx
  149cdf:       0f 1f 44 00 00          nopl   0x0(%rax, %rax, 1)
  149ce4:       66 66 2e 0f 1f 84 00    data16 nopw %cs:0x0(%rax, %rax, 1)
  149ceb:       00 00 00 00
  149cef:       90                      nop                    <= fall-through!

0000000000149cf0 <__GI___strcasecmp_l_ssse3>:
  149cf0:       f3 0f 1e fa             endbr64
  149cf4:       48 8b 02                mov    (%rdx), %rax
  149cf7:       f7 80 78 02 00 00 01    testl  $0x1, 0x278(%rax)
  149cfe:       00 00 00
  149d01:       0f 85 d9 ba f5 ff       jne    a57e0 <__strcasecmp_l_nonascii>
  149d07:       89 f1                   mov    %esi, %ecx
  149d09:       89 f8                   mov    %edi, %eax
  149d0b:       48 83 e1 3f             and    $0x3f, %rcx
  149d0f:       48 83 e0 3f             and    $0x3f, %rax
  ...           ...                     ...
```

Figure 2.8 – Example of a function call by fall-through. As seen in `glibc 2.28.30 (Fedora 29)` in x86 (AT&T syntax). `__strcasecmp_ssse3` (an optimized version of `strcasecmp` using x86 `SSSE3` [54] instructions) does not end in a branch instruction. Instead execution will always fall-through to `__GI___strcasecmp_l_ssse3` (another optimized variant of `strcasecmp`).

```
      call here
here: pop %rax
      # %rax now contains the absolute address of `here'
```

Figure 2.9 – 32 bit x86 assembly code to obtain the value of the program counter (AT&T syntax).

It consists of a `call` instruction that jumps zero bytes away, that is, it only pushes the value of the program counter in the stack and falls through to the next instruction Even though this uses a call instruction it is not an actual function call and is never returned from.

On 32 bit ARM the program counter is a general purpose register that can be read and written by any instruction. Consequently any instruction can perform a branch by writing to the program counter register. The standard return instruction on 32 bit ARM, for example, is actually just "`pop pc`". I.e. to return one simply pops the return address directly from the stack into the program counter.

In this work we do not use any technique to reconstruct the boundaries of functions [9, 15]. Instead we rely on the ELF symbol table and DWARF Debug Information Format (DWARF) [205] debug information to detect the addresses and boundaries of functions. We also make the following assumptions about functions:

1. All code for a function is located in a single, contiguous ELF symbol; that is, they contain no holes.
2. Functions do not overlap; if they do we split them into multiple non-overlapping functions.

Since we want to not only support the regular code patterns emitted by most compilers, but also hand written assembly code we also assume that functions can have multiple entry points, i.e. a call to a function does not necessarily enter at its first instruction.

Unfortunately, the ELF symbol table and debug information do not necessarily contain

information for all functions in a binary. Static functions in C/C++, for example, which aren't part of a binary's interface are usually not included in the symbol table. As we currently have not implemented any algorithm for recovering functions from machine code we group all the code regions for which we have no information into one big "unknown" function. Since unmarked code is usually spread out over the binary and the shared libraries it loads we allow this special function to, unlike any other function, contain holes.

Given this model of functions is it is quite straightforward to distinguish the different types of branches. All information that is required for this is:

- The type of the instruction the trace that performed the branch, i.e., whether it was a `call` or `jump`, etc., and
- whether the source and the destination of the branch are in the same or in different functions.

The pseudo-code for calculating the branch type can be seen in Algorithm 2.1.

Finally, to detect when branches occur in the guest it suffices to simply track whenever execution enters a new TB. Since branch instructions always terminate their trace block we know that if a new TB is entered a branch has been executed. The event object, which can be seen in Figure 2.10, generated for every branch simply records the TB the branch originated from and the target of the branch. To implement this the control-flow tracing does two things whenever a new TB is translated:

- It allocates a descriptor object recording metadata for the TBs, and
- it inserts a call to a helper function at the beginning of the TB, passing the metadata object as an argument. This helper function will then emit a branch event every time the TB is executed.

Pseudo-code for this can be seen in Figure 2.11.

There is one particular case, though, where instrumenting only the entry of a TB does not suffice. When a function does not end in a branch but falls through to the next function after it, the current TB will typically not be terminated at this transition. Instead, the generated TB will contain instructions from both functions. In this case, we have to insert a call to a helper function between the two functions to emit a branch event whenever the fall-through occurs. Ordinary fall-throughs that do not leave their function do not require any special treatment and are handled the same as normal local branches.

Using this instrumentation we can capture nearly all forms of control flow. However, there are two types that cannot be recovered this way, namely: the interprocedural control-flow transfers that occur during stack unwinding, and the local fall-through branches from a `call` site to the instruction following it. The control-flow tracing plugin post-processes the raw stream of events generated by the instrumentation to detect these cases and inserts appropriate events. Pseudo-code for this can be seen in Algorithm 2.2.

During stack unwinding, which is usually used to implement exception handling for languages such as C++, one or more call frames can be discarded without using `return` instructions. Since we store the return destination of every call on the shadow call stack we can detect when the expected destination and the actual target do not match. In this case we pop off shadow stack frames, adding *unwind* edges to the CG as we go, until we find a matching return target. Note, that this only works if the call stack is used like a stack. Advanced uses of `setjmp/longjmp` or fully fledged continuations are not supported.

In most IRs used in compilers and analysis tools interprocedural and intraprocedural control-flow is strictly distinguished. For simplicity, function calls are furthermore not considered branch instructions from the point of view of the control-flow inside a procedure. Instead a function call is simply modelled as a single instruction with complex effects followed by a fall-through to the next instruction. QEMU, on the other hand, directly emulates the behaviour of the CPU and does not have any notion of local or non-local control-flow. Thus, the control-flow tracing

```
# types of different instructions that can terminate a TB
enum Terminator_Type:
  JUMP, CALL, RETURN, OTHER

# different types of branches we can detect
enum Branch_Type:
  # inside a function
  LOCAL_JUMP,
  # between functions
  CALL, TAIL_CALL, UNWIND, RETURN

struct Block:
  # unique ID for the function containing this block
  function: Function_Id
  # PC of first guest instruction in block
  start: PC
  # type of the last instruction in the block
  terminator_type: Terminator_Type

struct Branch_Event:
  src:    Block
  dst:    Block
  brtype: Branch_Type
```

Figure 2.10 – Event data structures used for control flow tracing.

instrumentation will only observe a call branch out of the current function and later a return branch from the called function to the instruction after the `call`. Without further treatment any consumer of the event stream that only looks at the local control-flow would wrongly believe that the function call never returned. Instead, the control-flow tracing plugin emits a synthetic fall-through event after every return as shown in Algorithm 2.2, line 28. We cannot simply insert this fall-through branch statically after every call instruction since some calls actually do not ever return.

We will now briefly describe two low-level analyses using the final stream of branch events produced by the control-flow tracing plugin to compute higher level information. Chapter 4 contains higher level applications that build on top of them.

2.4.1.1 CFG and CG Reconstruction

For the analyses proposed in thesis, as described in Chapter 4, we need as precise information about control-flow as possible. However, statically reconstructing the exact CFG and CG from machine code from is considered, in general, undecidable due to, among other things, data-dependent branches and instruction punning. Instead of an approximate static analysis we chose to use a dynamic analysis, based on our tracing infrastructure described above, which can perfectly capture the control-flow of a given execution of a program. Since the stream of events already distinguishes local branches and non-local control flow and contains both the source and destination of every branch it is trivial to reconstruct the CFG and CG from it. The trade-off being that the information is partial, only covering the parts of the program that actually where executed, and depend on the input data used.

A consequence of this is that our analysis does not, strictly speaking, recover the *basic blocks* of a program. Recall, that a basic block has only one entry, meaning that blocks have

```
# types defined in Figure 2.10

def detect_branch_type(src: Block, dst: Block) -> Branch_Type:
  if src.terminator_type == CALL:
    return CALL
  elif src.terminator_type == JUMP:
    if src.function == dst.function:
      # NOTE: we do not distinguish self-recursive
      #       tail calls and local branches.
      return LOCAL_JUMP
    else:
      return TAIL_CALL
  elif src.terminator_type == RETURN:
    return RETURN
  else:
    # system call or fall-through

    if src.function != dst.function:
      # fall-through to another function.
      return TAIL_CALL
    else:
      # fall-through end of block
      return LOCAL_JUMP
```

Algorithm 2.1 – Distinguish calls, tail calls and local jumps.

to be split at branch targets. In machine code, though, we cannot recover the destination of branch instructions and any instruction is a valid branch target. We however believe that treating every instruction as a separate block would be overly conservative. As a final output our analysis still constructs single entry-single exit blocks of instructions, but only considers branch targets actually observed. We call these *execution blocks* since they correspond to the basic blocks observed during execution. Execution blocks are constructed from trace blocks by splitting the trace blocks seen during execution such that every instruction is contained in at most one block.

To improve coverage of the guest program the control-flow reconstruction plugin does take into account the targets of direct conditional branches that are never taken. It only considers direct branches though, since their targets can be computed statically even if they are not taken.

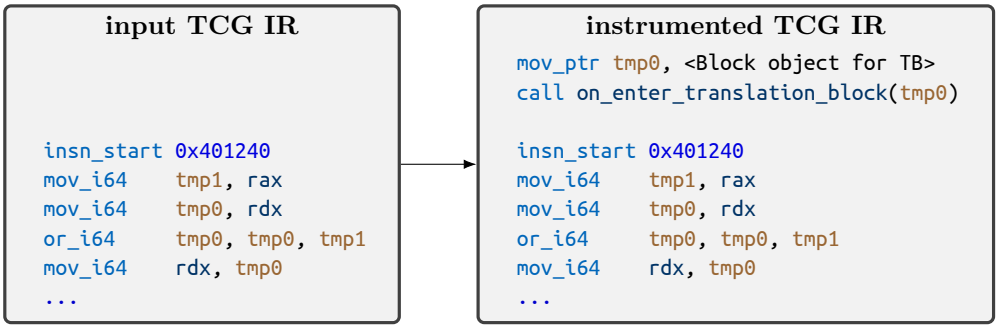
2.4.1.2 Calling-context tracking

The behaviour of any piece of code can vary wildly with the inputs and machine state, such as cache or branch predicator state, with which it is executed. Without a notion of context a profiler can not differentiate different executions of the same piece of code, possibly producing wildly misleading results. Ammons et al. [7] have proposed the idea of *context sensitive* profiling where information is not attached to a single static location in a program but to a path through the program. They also introduced the calling context tree (CCT) data structure, which encodes paths through the CG and makes it possible to track the dynamic *calling context* for each function call. Examples of a CG and CCT are shown in Figure 2.12

We have implemented a QEMU plugin which uses our control-flow trace to compute the calling contexts in which every instruction in a program executes. Note that, while the work of Ammons et al. goes beyond tracking calling contexts, this section is only meant to illustrate an

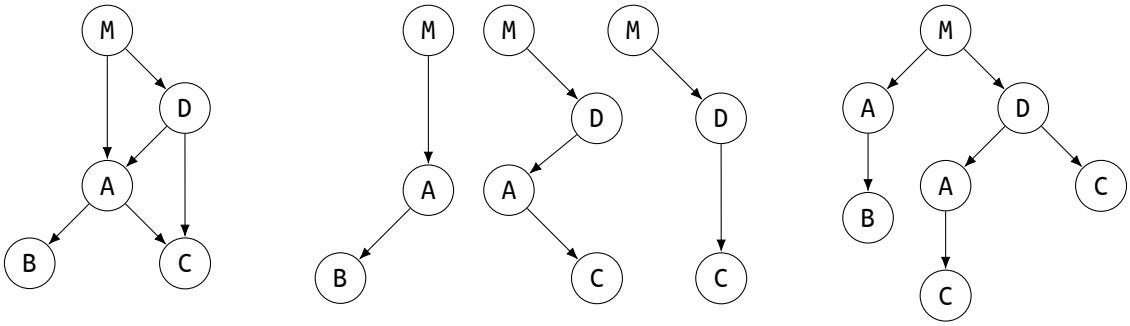
```
# types defined in Figure 2.10

# global variable
previous_block = None

# callback that is invoked whenever a block is executed
def on_enter_translation_block(block: Block):
  if previous_block != None:
    brtype = detect_branch_type(previous_block, block) # Algorithm 2.1
    emit Branch_Event(previous_block, block, brtype)
  previous_block = block
```

(a) Pseudo-code for the helper function that raises the branch events.



(b) Example of generated instrumentation in TCG IR.

Figure 2.11 – Code inserted to trace raw branch events.



(a) call graph (b) Some example call paths (c) calling context tree

Figure 2.12 – Example call graph and associated calling context tree.

application of control-flow tracing and focuses on our implementation of CCTs in QEMU. Section 4.4 will present an extended algorithm that also takes local control-flow inside of functions into account.

Our CCT implementation is, in essence, a trie [59, 27] encoding the set of call stacks that where live during a program's execution. Every node in the CCT represents one call frame and stores the address of the instruction that performed the call and the address where the call should return. The node data structure also contains an identifier of the function containing the call instruction for easy reference. The children of a CCT node in turn correspond to all the functions that have been called while the call frame from that node was live. By furthermore storing a parent pointer in every node of the CCT we can compactly represent a whole call stack

```
# Shadow call stack mirroring the real call stack of the guest.
call_stack: Stack<Call_Frame>

def on_branch(evt: Branch_Event):
  if evt.type == CALL:
    # Push shadow call frame on the call stack
    call_stack.push({evt.src, ...});
    # Forward call event
    emit evt
  elif evt.brtype == RETURN:
    caller = call_stack.pop()

    if caller.return_target != evt.dst.start:
      # Return did not go where it was supposed to due to
      # an exception or longjmp.
      # Unwind stack.

      while True:
        caller = call_stack.pop()
        if caller.return_target == evt.dst.start:
          break
        emit Unwind_Branch(caller)

    # emit the actual return event itself
    assert caller.return_target == evt.dst.start
    emit Return_Branch(caller, evt)

    # fall-through from TB terminated by call.
    if caller.function == evt.dst.function:
      emit Local_Branch(caller, evt.dst)
    else:
      emit Tail_Call(caller, evt.dst)
  else:
    ...
```

Algorithm 2.2 – Code to handle function returns and exception unwinding.

with a single node reference.

Note that this version the CCT has no bound on the depth of the tree. With recursion the depth of the CCT will in fact grow proportionally with the depth of the recursion. It also does not contain any information about local control-flow inside of functions. The original version, developed by Ammons et al., does in fact bound the depth of the tree and also encodes local flow information. However, it only achieves this by ignoring cycles in the CG and CFG. We have developed an extension of the CCT that captures both recursive call paths as well as loops inside of functions which will be described in Section 4.4.

In the following we will use references to CCT nodes plus the address of an instruction as instruction identifier. Including a CCT node in the identifier allows us to distinguish different dynamic executions of an instruction.

2.4.2 Data Dependence Tracing

The goal of data dependence tracing is to discover pairwise data dependencies between instructions. For our purposes we define that a data-dependence occurs when an instructions reads data produced by another instruction. Note that we also distinguish different dynamic executions of the same instruction. That is if an instruction is executed in a loop we will emit new data dependence events every iteration. While there are powerful static techniques to calculate data dependencies, like the polyhedral model, they only produce accurate results for a highly restricted class of programs. In order to be able to produce precise results for general programs we chose, as we did for control-flow tracing, to use a dynamic approach.

To detect dynamic data dependencies in programs we use shadow memory [45, 154, 229]. A shadow memory is a data structure that records a piece of information for each storage location used in a program. For dependency tracking the information is usually an identifier for the last instruction that modified that location. Now, whenever an instruction reads data from a storage location, i.e. performs a read from memory or a register, we can find the producer of that data in the shadow memory.

Our implementation of a shadow memory works at byte granularity, meaning that a 4-byte write will update four shadow memory cells and a 4-byte read will look at four shadow memory cells. We actually group entries of the shadow memory into shadow pages, similar to how normal memory is split into pages. These pages are stored in a search tree allowing for quick retrieval. Grouping cells together allows us to exploit the locality of memory accesses in two ways:

1. Even though a n-byte memory access touches n shadow cells all these cells are usually located on the same page. This means that, even though we work at a byte granularity, we only have to perform on lookup once per memory access. At most two, if the access spans two pages. A shadow page itself is just a large array and the actual reads or updates of shadow cells can be performed directly.
2. We have implemented a small and fast shadow page cache [229] which speeds up repeated accesses to the same page.

Our library is not only able to trace data dependencies via memory, but also via registers using a shadow register file. Here the mapping from guest CPU registers to shadow registers is much more straightforward, since the registers used by a machine instruction can be determined statically and also because the number of registers in a CPU is usually quite small. The shadow register file is implemented as an array with one entry for every guest CPU register. For simplicity we do not consider register aliasing. That is, a write or read of a register is always assumed to access the whole register.

An attentive reader might have noticed that we only track read-after-write (RAW), or *true*, dependencies. The plugin could easily be extended to also track write-after-write (WAW), or *output*, dependencies. For the most we chose not to do so because our current applications do not require it and because it would only generate unused events. Tracking write-after-read (WAR) or read-after-read (RAR) dependencies directly would be more complicated and costly since it requires updating the shadow memory at reads. If necessary WAR and RAR dependencies can be recalculated as a post-pass from the RAW and WAW dependence information.

To reduce the size of traces and ease later analysis our shadow memory implements a simple form of dynamic copy propagation. This mechanism applies whenever the instrumenting data-copying instructions which only move data between storage locations without modifying it. The instrumentation inserted to update the shadow memory for these instructions works slightly different. If a dynamic execution of a copy instruction only reads data produced by a single other instruction the identifier recorded in the shadow memory cell for the destination of the copy is not that of the copy instruction but that of the actual producer. This propagation mechanism is designed to, among other things, filter out spill code inserted by compilers to store and load the

contents of registers to and from the stack. The idea being that instructions that don't compute anything and only move data around are usually not interesting for dependence analysis. Since the exact definition of what constitutes a pure data copy is a bit vague and can vary along with different use cases. Our shadow memory can thus be configured to also consider instructions that perform integer truncation, sign-extension or conversion between integers and floating point numbers as copy operations.

For x86 guests we also detect classical dependence breaker idioms as documented by Intel and extensively used by compilers. These idioms are instructions used as hints by compilers to break false dependence chains via registers. Take for example the instruction `xor %rax, %rax`, which performs a bitwise xor on the contents of the register `%rax` and stores the result back into `%rax`. The result of this instruction does not depend on the contents of `%rax`, it will always set `%rax` to 0, and it does not create a data dependence on whoever last wrote to `%rax`.

Our current implementation of shadow memory does not support concurrent accesses by multiple threads from the guest program. Consequently, our dependence tracking plugin can only be used for serial programs.

2.4.3 Value tracing

The simplest form of tracing we have implemented is value tracing. The goal of value tracing is to capture any value architecturally computed by the guest program. That is any value written to a guest CPU register or to guest memory. We also consider memory addresses that a guest program accesses as values, even if the addresses are calculated inside the instruction and not stored anywhere. Same as the control-flow and data dependence tracing we use this as part of the dynamic analysis presented in Chapter 4. As shown in that chapter we use these traced values and addresses to detect loop iterators and regular memory access patterns.

The TPI value tracing plugin we have written generates one event for every guest instruction executed. For our purposes we actually do not consider every instruction, but only those that produce integral values and/or perform a memory access. Most branch, compare, floating point or vector instructions are not considered here. The event contains an identifier for the instruction along with space for one register sized value and one memory address. For instructions that do not generate a value, such as store instructions, we mark the value field as unused. Similarly, for instructions that do not access memory we mark the address field as unused. For simplicity an event can store at most one value and address each. In the rare case that an instructions produces more than one value or accesses more than one address we do not capture all these values but instead mark the corresponding field in the event as unused.

2.5 Evaluation of the Overhead of Plugins

In this section we show the overhead incurred by running programs in QEMU with some of our plugins. We also compare this to the slowdown of running the same programs in other binary translators.

To evaluate the overhead of our plugins we have used the latest revision (3.1) of the Rodinia benchmark suite [42, 43]. As our tool-chain does not support multi-threaded applications yet, each benchmark is manually modified to turn into a single thread. It is then compiled using GCC 8.1 using flags `-g -O2 -msse3`. We use `-sse3` since QEMU does not yet have full support any more recent vector instructions. The binary translators we used are QEMU version 3.1, Valgrind version 3.15.0 [155], and DynamoRIO version 7.1.0 [29]. All time measurements where performed on a single core of a Haswell i5-4590 CPU running at 3.3GHz.

We first evaluate the overhead of using QEMU-plugins as an instrumentation framework. For that purpose we compare the execution time in different contexts:

	(time)	(slowdown)							
	native	QEMU	Valgrind	Dynamorio	Dinero IV	Cachegrind	drcachesim	bftrace	shadow memory
backprop	$0.04s \pm 0.0s$	$7.5x$	$8.8x$	$1.1x$	$455x$	$28x$	$373x$	$24x$	$63x$
bfs	$1.1s \pm 0.0s$	$5.8x$	$7x$	$0.5x$	$921x$	$50x$	$859x$	$40x$	$173x$
b+tree	$1.1s \pm 0.0s$	$5x$	$6.8x$	$0.1x$	$850x$	$40x$	$654x$	$38x$	$101x$
cfd	$2.1s \pm 0.0s$	$28x$	$5x$	$0.2x$	$475x$	$34x$	$419x$	$44x$	$132x$
heartwall	$16.96s \pm 0.1s$	$31x$	$11x$	$0.01x$	$608x$	$29x$	$464x$	$43x$	$90x$
hotspot	$1.1s \pm 0.0s$	$5x$	$5x$	$0.3x$	$573x$	$52x$	$537x$	$33x$	$167x$
hotspot3D	$4.5s \pm 0.0s$	$24x$	$8.5x$	$0.1x$	$725x$	$48x$	$607x$	$39x$	$164x$
kmeans	$3.9s \pm 0.0s$	$38x$	$12x$	$0.1x$	$807x$	$39x$	$611x$	$61x$	$181x$
lavaMD	$4.1s \pm 0.0s$	$38x$	$21x$	$3.7x$	$627x$	$44x$	$521x$	$54x$	$130x$
leukocyte	$11.8s \pm 0.0s$	$56x$	$20x$	$1.9x$	$819x$	$48x$	$587x$	$75x$	$174x$
lud	$0.1s \pm 0.0s$	$22x$	$11x$	$0.6x$	$804x$	$56x$	$732x$	$49x$	$170x$
myocyte	$0.002s \pm 0.0s$	$24x$	$120x$	$22x$	$306x$	$157x$	$478x$	$82x$	$149x$
nn	$0.01s \pm 0.0s$	$8.9x$	$26x$	$3.7x$	$518x$	$57x$	$489x$	$41x$	$121x$
nw	$0.1s \pm 0.0s$	$2x$	$6.1x$	$0.7x$	$314x$	$19x$	$268x$	$9.8x$	$45x$
particlefilter	$0.3s \pm 0.0s$	$32x$	$10x$	$0.2x$	$746x$	$39x$	$611x$	$62x$	$127x$
pathfinder	$0.8s \pm 0.0s$	$6.6x$	$6.4x$	$0.7x$	$758x$	$41x$	$686x$	$39x$	$132x$
srad_v1	$0.5s \pm 0.0s$	$47x$	$6.8x$	$0.2x$	$416x$	$23x$	$308x$	$53x$	$100x$
srad_v2	$0.3s \pm 0.0s$	$40x$	$10x$	$1.2x$	$401x$	$26x$	$307x$	$48x$	$88x$
streamcluster	$38.1s \pm 0.3s$	$42x$	$7.4x$	$0.01x$	$473x$	$23x$	$357x$	$50x$	$102x$

Table 2.1 – Overhead of profiling on the Rodinia benchmark suite: slowdown compared to native execution (overhead = $\frac{\text{emulator time} - \text{native time}}{\text{native time}}$). Times are averaged over ten runs.

1. We take the native execution time as a baseline.
2. We evaluate raw execution time of different binary translators, i.e., without any instrumentation.
3. We evaluate the overhead of doing basic memory profiling of those frameworks (Dinero IV [66] for cache simulation under QEMU, `cachegrind` [152] for Valgrind, `drcachesim` for DynamoRIO).
4. We finally evaluate the overhead of CFG reconstruction plugin and a simple shadow-memory plugin that traces data dependencies between machine instructions.

Table 2.1 reports the slowdown compared to the native execution time on backprop (benchmark used for our case study) and Rodinia (average values). We observe the following: Without any plugins Valgrind and QEMU run at roughly the same speed, while DynamoRio is much faster; Still as one could expect, the main source of overhead is the inserted instrumentation itself. Even a highly optimised cache simulator like Cachegrind incurs a close to 4x slowdown compared to just running Valgrind by itself. The control-flow tracing plugin has a relatively low overhead since it only inserts instrumentation at the beginning of every trace block. The shadow memory plugin, on the other hand, has a substantially higher overhead.

As we can see from this table QEMU is clearly not the fastest binary translation system. However, as soon as we perform any significant instrumentation the overhead introduced of the instrumentation clearly dominates the overhead of the binary translator.

2.6 Related Work

This section discusses some previous work related to binary instrumentation and also regarding the applications thereof presented in this chapter.

QEMU based tools

Two projects closely related to our work are TEMU[193], part of the BitBlaze tool suite, and its successor DECAF [89]. Both of these tools implement a dynamic binary instrumentation infrastructure based on QEMU. TEMU and DECAF can only run using QEMU's full system mode which emulates and observe a whole operating system. Like TPI, both offer a plugin API which can be used to implement dynamic analyses. Their APIs, however, do not give plugins direct access to TCG IR. Instead they offer a fixed set of hooks in guest programs for which a plugin can register callbacks. TEMU and DECAF then take care of inserting these hook functions into the emulated binaries. There are hooks to, among other things, trace the entry and exit of functions or TBs, the execution of individual instructions, and also for tracking memory accesses.

Neither TEMU nor DECAF offer support for tracing data dependencies, they instead implement a taint tracing mechanism. Taint tracing is a dynamic analysis that is commonly used in security analysis. For this analysis the user labels certain sources of inputs, such as the keyboard, network interface, or hard disk, as *tainted*. The taint tracing mechanism then keeps track of the dynamic data-flow inside the guest system to detect where the tainted data is used. Both TEMU and DECAF implement this using a shadow memory working at bit-level granularity.

PANDA [64] is a dynamic binary analysis tool used for reverse engineering software. As with the other tools PANDA also allows writing individual analyses in the form of plugins. Like DECAF and TEMU PANDA offers a fixed set of hooks into a guest program's execution that plugins can use. PANDA can also optionally replace QEMU's standard compiler, TCG, with the JIT compiler of the LLVM compiler infrastructure [124, 165]. In this mode PANDA translates TCG IR to the semantically richer LLVM IR. Plugins can then inspect and arbitrarily modify guest code in LLVM IR form before it is translated to host machine code. Using a more high level IR makes writing many plugins easier and allows plugin authors to reuse the extensive catalogue of static analyses available with LLVM. In this mode PANDA also uses LLVM to compile the IR to host machine code. While LLVMs JIT compiler produces much better machine code than TCG it is also considerable slower. Overall PANDAs author observe a slowdown of around 10x compared to normal QEMU when running in this mode.

We would like to point out one last pragmatic difference between TPI and the other tools presented above. In their implementation all these tools have extensively modified the source code of QEMU itself. Since QEMU is a living project its code base changes over time as features are added [3] Consequently, it takes considerable effort to maintain the forked versions of QEMU and all project this author is aware of have fallen several releases behind the main QEMU version. TPIs implementation, on the other hand, tries to be as non-invasive as possible and only modifies the guest ISA independent parts of TCG. Since the API of TCG itself and its IR are relatively stable porting the plugin infrastructure to a new release of QEMU takes less than a day of work.

Dynamic binary instrumentation tools

There are, of course, many other dynamic binary translation and instrumentation frameworks beside QEMU. The most widely used ones, Valgrind [155], Intel Pin [134], and DynamoRIO [29], work similar to QEMU+TPI in user mode. Like QEMU they use trace based binary translation of programs. They do, however, offer neither an equivalent of QEMU full system mode nor are they able to cross translate between different ISAs and application binary interfaces (ABIs). I.e. the host CPU must be the same as the guest CPU and the guest and host kernel must also

[3] Over the last five years it has received, on average 573 commits per month to its Git repository, excluding merge commits.

use the same ABI. They also support less different CPU architectures than QEMU. Pin, for example, since it is an official Intel product only works on x86 programs.

Valgrind is a framework for building dynamic binary analyses that is used extensively to debug memory management and threading problems. Like QEMU it translates machine code to a platform independent IR before emitting the final machine code. Valgrinds IR is richer than TCGs allowing instructions to take whole expressions as arguments instead of only registers and constants. It also has extensive support for hooking or overwriting functions of the guest program by plugins. Contrary to TPI Valgrind does not support running multiple plugins at the same time.

Pin and DynamoRIO are binary instrumentation systems that provide a much thinner layer of abstraction from the underlying CPU architecture than Valgrind or TPI. Their IR, for example, does abstract away details of the underlying guest ISA but only offer a one to one representation of instructions. They also do not offer any form of virtual registers. This means that plugins need to manager machine registers used by their instrumentation themselves and also need to make sure to not clobber guest registers. The upside of this much more direct approach to binary translation is that both Pin and DynamoRIO introduce a much lower overhead than QEMU or Valgrind. This is because they 1 do not need to transform the guest code into a very different representation, and 2 the final machine code they emit is usually much closer to the original guest code.

Dyninst [145] is a set of C++ libraries for both static and dynamic binary analysis developed as part of the Paradyn project [146]. This framework allows instrumentation to be specified via DynC, a small C-like language, using an embedded C++ DSL or directly as snippets of machine instructions. Dyninst uses static parsing of the binary machine code to recover the CFG of a program. The parser can use symbol information, if available, but can also detect function boundaries by itself. It can also resolve some types of indirect jumps such as jump tables and uses an interprocedural analysis to detect functions that never return. It furthermore has an API to walk the runtime stack of running programs used in sampling tools such as HPCToolkit [4]. Finally, Dyninst can also perform some static data-flow analyses to detect which instructions affect or are affected by the value in a register or memory location.

Static binary instrumentation tools [123, 196, 173, 67] are less widely used today. As mentioned earlier in this chapter there are some inherently undecidable problems in static binary analysis. Consequently that a static instrumentation tool can not work on arbitrary programs. In practice though they work well, as long as they are presented with the mostly regular and structured machine code emitted by a compiler [8].

Static binary re-optimizers like BOLT [157] and the work of Koju et al. do not perform full binary instrumentation, but only try to transform a program to improve its performance. Since they only seek to improve performance they can just ignore code patterns that they cannot analyse. These tools also assume that the input binary has been produced with a well-known compiler. They use this to reconstruct high-level information by matching well-known code patterns and by making certain assumptions about the structure of the program.

There also exist static re-optimizers that work at the level of textual assembly language [60, 103]. The advantage of working at this level instead of the machine code level is that it is more structured. Jump labels, for example, are still available in assembly making it trivial to recover the local CFG of functions. Functions boundaries are also, for the most part, clearly demarcated. Lastly, assembly also clearly seperates code and static data sections. It even provides a modicum of type information, such as whether a object is a string or an array of integers.

Dynamic CFG reconstruction

Dynamic control-flow reconstruction as presented here is less commonly used than static approaches.

PinPlay [159], a Pin based framework for recording and replaying program executions contains a module to dynamically reconstruct the CFG of a program. Like with TPI, CFGs reconstructed by PinPlay are partial and depend on the actual control flow seen during execution. Since Pin is, like QEMU, a trace based system their definition of a *dynamic control flow graph* (DCFG)[227] used in PinPlay is essentially the same as the one used in this thesis. PinPlays DCFG also includes the virtual fall-through edges between a call instruction and the instruction following the call. Unlike our TPI based plugin PinPlay also supports multi-threaded applications.

The Forced Execution Engine (FXE) [225] is another QEMU based tool that reconstructs the CFG of an application from execution traces. However, FXE borrows ideas from concolic testing frameworks [34, 184] and merges multiple execution traces to build a more complete CFG. During execution FXE snapshots the state of the guest application at every conditional branch that has not yet been fully explored. It then later rewinds the program state and forces the conditional branches to follow the unexplored control paths. Unlike a full concolic execution engine FXE does not use an expensive constraint solver to generate program states that cause execution to flow to unexplored program regions. Instead, it simply directly modifies the guest CPU state object to control the target a conditional branch instruction will jump to. Since FXE modified the guest programs state to execute all possible targets of conditional branches it often runs into illegal program states and can exercise control-flow paths that are impossible in the original program. Finally, FXE can also explore all targets for some indirect branches. It performs some simple pattern matching to find jump tables in the guest program and then uses the same snapshot and replay mechanism to execute all possible indirect branch targets.

Data dependence tracing

There is a rich existing body of work on data dependence profiling techniques[153, 73, 70, 115, 114, 127, 215, 147, 46]. Until now we have only explored the first part of dependence profiling, the actual collection of dependence pairs. We will hold off on comparing our work with these tools until we have presented actual analysis part of it in Chapter 4. For now we will only discuss the details of the shadow memory implementations from these tools.

Shadow memories are a heavy-weight monitoring technique that come with a large performance overhead. This is simply due to the fact that for every memory access in a program the instrumentation performs add at least one, usually multiple, additional memory accesses. The most expensive part of a shadow memory requiring the most work is usually the mapping mechanism which translates guest memory pages to shadow memory pages. To alleviate the overhead of mapping implementations of a shadow memory make aggressive use of caching. Umbra [229] has both a per thread cache for the last translation performed as well as a separate one entry cache for every memory access instruction in the guest program.

Another approach to reduce the overhead is to avoid unnecessary accesses to the shadow memory. ParWiz [113] uses static analysis to coalesce adjacent memory accesses turning multiple smaller acesses to the shadow memory into a single big one. It is even able to hoist the instrumentation out of some static control loops potentially greatly reducing the number instrumentation instructions executed at runtime. Li et al. [128] also use static analyis to skip repeatedly executed memory accesses that lead to identical data dependencies.

Another big problem with shadow memories is their large space overhead. The SD^3 [115] algorithm used in the data-dependence profiler Prospector [114] reduces memory usage by de-

tecting stride patterns in memory accesses and using them to compress the information stored in the shadow memory. Another technique, which trades off space-usage for precision, uses signatures to detect dependencies instead of a shadow memory [129, 215]. Signatures are a data structure that encodes a possibly unbounded set of entries in bounded space. While signatures can be used to answer the same types of queries as a shadow memory, all while using less space, they are an approximate data structure. Consequently, one is no longer guaranteed to detect all data dependencies.

A shortcoming of many shadow memory based systems [73, 70, 155] as well as our own is that they do not support concurrent accesses from multiple threads. Even existing work on shadow memories supporting parallel access have severe limitations. Umbra [229], for example, only handles concurrent access to its internal mapping data structures which map from guest addresses to shadow pages. It does not provide synchronization for the actual reads and writes of shadow memory cells. SD^3 [115], on the other hand, only reports dependencies inside threads and not between threads. Valgrind [155], simply implements its own scheduler and interleaves the execution of all threads of the guest program on a single host thread.

Note, that shadow memories are not only for data dependence tracing, but are also a core component of memory error debugging tools [155, 185]. Shadow memories used in this context often have a smaller performance impact since they often need to only track one or two bits per byte in the guest process.

Hardware assisted tracing

Some CPU architectures allow tracing the execution of programs with little or no modification necessary. Instead of inserting instrumentation to collect a trace, this uses specialized features of the hardware itself. Support for this goes back as far as the 1950's [175]. The IBM 370, for example, introduced the Program event recording (PER) facility [56] which can capture a stream of memory write events, instruction fetch events and branch events.

More recently, Intel has released a feature called Processor Trace (PT) [51, 168]. When PT is activated, the CPU writes a stream of packets containing events about a process's execution into a buffer in user space memory. The current generation of CPUs supporting this technology emit events for every branch and so can be used for control-flow tracing. They can also emit packets that allow very precise execution time measurements.

The most recent generations of Intel CPUs have a specialized instruction, `ptwrite` that can inject packets with arbitrary user defined word-sized data into the PT event stream. This instruction could be leveraged by binary instrumentation tools to generate arbitrary traces with a lower overhead than using conventional means.

Under optimal conditions hardware assisted tracing can capture a record of a program's run with virtually no overhead in execution time. In practice this is not possible, simply due to the fact that the CPU can generate events faster than currently existing storage technologies can handle. Recent work in control-flow integrity enforcement [75, 101], for example, have found that Intel PT drops packets when tracing larger programs, such as those from the SPEC CPU2006 benchmark suite [92]. Furthermore, Intel PT traces are, by necessity, highly compressed and actually using them to reconstruct control-flow graphs requires a non-negligible amount of post-processing and re-parsing of the original binary. This means, that while PT is a very interesting technology it is not yet useable for the applications we target since they require very precise information.

2.7 Conclusion and Perspectives

This chapter presented the internals of QEMU as well of a plugin infrastructure allowing QEMU to be used for dynamic binary instrumentation. We have also presented a number of plugins we have developed on this basis to trace different aspects of a program's execution, such as control-flow and data dependencies.

While TPI itself is already quite mature, there are still a large number of improvements to the system and the plugins we implemented on top of it that we are planning to make.

CPU architectures often have internal states and flags that change they way they execute instructions. Examples of this are privilege levels on ARM and memory segmentation on x86. Many of these flags rarely or never change during the execution of a program. Consequently, to achieve high performance, the code emitted by TCG does not actually constantly check for them. Instead it emits specialized versions of TBs for different CPU states and switches between them if the state does change. We are currently evaluating if the we could extend this mechanism to allow plugins to generate different specialized versions of instrumentation. This could, for example, be used to speed up context sensitive profiling. Instead of having to maintain the context tree as a separate data structure and having to look up nodes dynamically it would be baked directly into the instrumentation code.

Another use case for code specialization in TCG would be to implement sampled execution of instrumentation with a low overhead [13]. To achieve this one simply creates one context in which no instrumentation is inserted and one where it is. This essentially would allow turning the instrumentation on and off simply by changing the context flag. The only overhead here is that TBs that one wants to instrument need to be compiled twice.

The instrumentation plugins we have implemented currently do not currently work with self-modifying code. This means our plugins will not work properly with, for example, most programs that internally use a JIT compiler. The problem here is not QEMU, which supports programs that create and alter machine code at runtime just fine. Instead the problem lies with our notion of instruction and identifier and calling contexts. For the moment we use the address of instructions in the binary for both. Consequently, if the memory containing a function that has already been executed is overwritten with another function our instruction identifiers will become invalid. We do, however, support the dynamic patching of the procedure linkage table (PLT) by the dynamic linker which happens when a program calls a function in a shared library. This works simply because a PLT indirection is only patched once the first time the associated function is called. Unfortunately, the first call will have a different calling context than later calls since it this first call is actually performed by code in the dynamic linker. In order to properly support self modifying program we will have to add hooks to QEMU's internal mechanism that detects self modifying code. We then plan to extend TPI's API to expose these events to plugins so they can update their internal data structures.

As mentioned in the previous section, shadow memories are difficult to use when one wants to monitor concurrent programs. The problem is simply that synchronization at the granularity of individual memory accesses has too high an overhead, even when done using atomic instructions. Another problem is that both synchronization and concurrent handling of can change the order in which shadow memory accesses are processed. Hence, the ordering of shadow memory accesses is no longer the same as the one used for the actual memory accesses of the underlying, or emulated guest, hardware. As a result, the state of the shadow memory and the program memory can be out of sync. To guarantee that both are updated correctly potentially requires reimplementing the CPU's memory consistency protocols. Fortunately, QEMU [58] already has support for emulating different guest CPU memory ordering models. It achieves this by injecting special memory barrier instructions in its IR. We are currently investigating if this mechanism could be used to help implement a genuinely concurrent shadow memory.

Chapter 3

Sensitivity Based Performance Bottleneck Analysis

Contents

3.1 Introduction

A basic question that regularly comes up when optimizing software is what are the bottlenecks of a system that limit performance. Examples of resources that can form bottlenecks are the memory bandwidth or the number of floating-point or load/store instructions a CPU can process per cycle. Once a programmer knows the bottlenecks, they can then rewrite their program to make more careful use of that resource or offload work to other resources. Knowing what the bottlenecks are can also be used to estimate whether a program still has any potential for improvement or if it already reached peak performance [221].

Programs usually use different resources in complex interdependent ways. For example, stalls caused by saturating one resource can mask or aggravate problems with other resources. Due to this, bottlenecks are not necessarily associated with a saturated resource. And so identifying them in a program is not a trivial task. Commonly used profiling tools such as PAPI [65], LIKWID [172], HPCToolkit [4], Linux perf [144], and Intel VTune [169], while good at highlighting hot regions of a program and detecting the current utilization of resources are unable to find real bottlenecks.

Another problem that is even harder to solve is detecting by "how much" a resource is a bottleneck. In other words, how much would the overall performance of a program improve if a given bottleneck was removed?

Sensitivity analysis [141, 47, 118, 132, 99], also called *differential profiling*, is a profiling technique used to both find bottlenecks and their severity. Sensitivity analysis works by executing a program multiple times, each time varying the usage or capacity of one or more resources. Bottlenecks can then be identified by observing by how much the change for each resource impacts the overall performance, that is how sensitive performance is to a given resource.

Varying the capacity of a resource, for example by changing the CPU frequency via DVFS [100], is preferable over changing the resource usage since it can be done without altering the profiled program. Unfortunately, today's hardware does not offer many ways to easily vary the capacity of resources. Thus, existing approaches commonly resort to altering the program under observation. Either by changing the data set [141], the number of threads or CPU cores it executes on [132, 141, 47] or by selectively removing instructions from a program [118]. The downside of this is that changing the number of threads or what instructions are executed has a hard to predict the impact on resource usage, which makes fine-grain sensitivity analysis based on these approaches difficult. Another problem is that these changes are also likely to alter a program's semantics, which in turn may alter its performance characteristics.

To avoid these issues, it would be preferable to directly vary resource capacities. The SAAKE system by Hong et al. [99] has shown that the limitations of real hardware can be overcome by driving sensitivity analysis with a simulator. SAAKE uses input independent abstract simulation to predict execution times of graphics processing unit (GPU) programs, which works well for GPUs with their simple in-order execution model. The performance of high-end CPU architectures, however, is much more dynamic and input dependent due to their aggressive use of speculative out-of-order execution and more complex caches.

In this chapter, we present Gus, a prototype sensitivity-based dynamic analysis tool to find bottlenecks that brings the ideas of SAAKE to the world of CPUs. Gus combines QEMU, a fast functional CPU emulator, with an abstract, resource-centric CPU performance model in the spirit of [68, 76, 37]. It is not designed to compete with cycle-accurate simulators, which can produce highly accurate simulations at the cost of very long simulation times. Instead, Gus aims to quickly deliver predictions that are "good enough" for sensitivity analysis.

Gus is a plugin based on the TPI infrastructure described in Chapter 2. By implementing our performance simulator on top of a functional simulator, we can process real program traces and accurately simulate input dependent program behaviour.

Gus also uses a CPU model for its simulations which is designed to be easy to implement and port to new CPU architectures. Its CPU model abstracts a uniform representation for the different processor resources, such as the floating-point or load/store units, and the different cache levels. This representation is low-level enough to capture real-world bottlenecks but also simple enough so it can be automatically generated from the performance characteristics of the CPU. To build a performance model of a processor, we require the latency and throughput of different types of instructions as well as how they share CPU resources.

One commonly way used to represent the sharing of functional units on a CPU is a *port mapping*. A port mapping is a description of which instruction type can execute on which functional unit. For architectures that internally decompose ISA level instructions into multiple smaller micro-operations (μops) a port mapping also needs to capture the μop decomposition for every instruction.

Unfortunately, many CPU vendors do not publish comprehensive port mappings or data sheets with performance metrics. Intel, for example, only publishes the latency and throughput for "commonly used instructions" in its Software Developer's Manual [54]. AMD used to publish an appendix for their manual detailing latencies, throughputs, and port mappings for all their x86 architectures. However, they no longer publicly release this appendix for CPUs since the Zen architecture. ARM is the only major company that still publishes this information. Unfortunately, CPU vendors that have their own ARM-based processor designs like Samsung, Qualcomm, and Apple do not.

Many of the above CPU vendors do, however, indirectly make this information available by contributing to the back-ends of open source compilers. More specifically, the same information required by Gus is also used by instruction scheduling algorithms in compiler back-ends. Unfortunately, the models of mainstream compilers are not always accurate or up-to-date. For example, the LLVM scheduling model for Intel Sandy Bridge processors, a CPU architecture released in 2011, still required some tweaks and bug fixes as recently as 2019 [212]. Skylake processors, on the other hand, which were released in 2015, were treated like Haswell CPUs for the purposes of instruction scheduling until 2017 [106] even though the Skylake architecture has much-improved handling for vector instructions.

To be able to build a more detailed and accurate CPU model for Gus we have started to build Pipedream, a tool that can reverse-engineer the latency, throughput, and port mapping of instructions. Pipedream achieves this by automatically generating micro-benchmarks and measuring their behaviour using hardware performance counters. To select the port mapping that best fits a given set of observations, we have developed an extension to the maximum flow problem that can be efficiently evaluated with a linear program.

Pipedream does not necessarily produce a model that perfectly matches the real hardware. It does not need to know the semantics of μops or what bit patterns encode which μop [119]. We only need a model that matches the performance behaviour of instruction and μops to allow Gus to predict execution times with reasonable accuracy.

While still a prototype, Pipedream is already able to build a model for a significant subset of the x86 instruction set with little user intervention.

To summarize, the contributions presented in this chapter are:

1. Gus – A sensitivity based performance debugging tool that can be used to find performance bottlenecks and predict how much overall performance can be improved by removing these bottlenecks.
2. Pipedream – A tool for reverse-engineering low-level performance characteristics of instructions on modern CPUs.

3.2 Modern CPU Microarchitecture

Before describing the details of our CPU model used to find performance bottlenecks, we give a high-level overview of the actual microarchitecture of modern processors. We only discuss topics relevant for either Gus or Pipedream and this section is by no means an in-depth exploration of the subject. For a deeper insight, we refer the reader to seminal introductory works [90, 186]. The focus of this section is on implementation techniques as used in general-purpose CPU architectures such as x86, ARM, PowerPC, or MIPS.

From the point of view of a programmer, the behaviour of a CPU is defined by its ISA. Since both Intel CPUs and AMD CPUs implement the same ISA one can run programs compiled for that ISA on either. Internally, however, the CPUs of the two companies, and even different generations of CPUs from the same company work quite differently. The actual design and implementation, the *microarchitecture*, of a processor can differ significantly from the ISA, as long as this difference is not observable from the outside. For example, most common ISAs give the illusion that a single core always executes programs sequentially, one instruction at a time. In reality, however, a modern CPU is a highly complicated parallel system where many instructions execute at once. This is called *instruction level parallelism* and corresponds to the dynamic parallelization of programs at the granularity of individual instructions.

The core of a CPU responsible for executing instructions can be conceptually separated into two main components:

- The *front-end*, which is responsible for fetching instructions from memory and decoding them.
- The *back-end* or *execution unit*, which is in charge of actually executing instructions.

High-performance instruction decoding for a complex instruction set such as x86 is in itself a complex multi-stage process. To feed the back-end of the CPU with enough instructions for parallel execution, the front-end usually decodes multiple instructions per cycle. The decoded instructions are then placed in the *instruction queue* which forms the interface between the front-end and the back-end.

3.2.1 Concepts of instruction level parallelism

The most important techniques for implementing instruction level parallelism are the following: *instruction pipelining*, *super-scalar execution*, *out-of-order execution*, and *speculative execution*.

Instruction pipelining: One of the oldest microarchitectural techniques for parallelising the execution of instructions is *instruction pipelining*. The idea of pipelining is to not execute every instruction as a whole before moving on to the next one but to overlap the execution of multiple instructions. That is, the execution of an instruction is split instruction up into multiple small steps, for example: *fetch input*, *compute result*, and *store result*. When one instruction is in the *compute result* phase the processor can already execute the *fetch input* step of next instruction, before the last step of the previous one has finished. In reality, instruction pipelines are much deeper and sometimes consist of multiple tens of steps.

Super-scalar execution: Super-scalar execution is an orthogonal approach to instruction pipelining. With pipelining the CPU splits every instruction into smaller parts and executes the parts of different instructions in parallel. A super-scalar processor, however, splits the back-end into multiple parts, containing not one but multiple execution units. For example, a processor with two separate functional units for scalar arithmetic and vector arithmetic can execute multiple instructions at the same time by multiplexing the linear instruction stream onto these two units. Super-scalar designs usually even contain multiple copies of highly used

functional units to execute multiple instances of common instructions in parallel. The individual execution units of a super-scalar processor can of course also be pipelined.

Out-of-order execution: By themselves, and even together, pipelining and super-scalar execution can not exploit all parallelism available in many programs because they are still limited by the order of instructions. Groups of instructions that use the same execution unit may, for example, block super-scalar execution, leaving the other execution units idle. Out-of-Order (OoO) processors break these constraints by dynamically rescheduling the instructions of a program at runtime. Most OoO architectures used today use a variation of the Tomasulo algorithm [207]. The idea of the Tomasulo algorithm is, essentially, to turn a sequential program into a data-flow program, which is then executed in parallel, only constrained by its data dependencies. Under this scheme, instructions are still decoded in program order, but they execute in parallel. In all typical implementations instructions also complete in order. That is, the side effects of instructions, such as writing to registers or memory, still appear to happen in the original order of the program. This last detail is essential to uphold the convenient illusion of sequential execution that ISAs provide to programmers.

Speculative execution: The last stumbling block for parallel execution is the frequency of branch instructions, i.e., the length of basic blocks in programs. Most basic blocks in a program contain less than 15 instructions [102]. Since the processor cannot know whether conditional branches will be taken or where indirect branches will jump, it cannot execute instructions after the branch before the branch itself finishes. That is, branch instructions essentially form barriers for parallel execution. The idea of speculative execution is simply to let the processor guess the destination of branches and speculatively execute instructions. If the prediction turns out to be wrong, the processor simply discards the results of any speculatively executed instructions and continues with the real branch target. Since even simple branch predictors correctly predict over 90% of all branches speculative execution generally has little to no overhead and effectively increases the "window" of parallel instructions.

3.2.2 Implementation of instruction level parallelism

The following gives a short overview of some of the data structures and algorithms used to implement super-scalar out-of-order execution.

Reservation stations and re-order buffer: To support out-of-order execution with in-order completion the processor needs buffers to store the instructions that are currently in-flight as well as their arguments and results. The exact number and of usage of these buffers varies between microarchitectures, but in this work we will focus on designs used in the current generation of processors from Intel, AMD, and ARM. Here, the so-called *reservation stations*, sometimes also called *schedulers*, are in charge of fetching and storing the operands of instructions. The *re-order buffer (ROB)*, or *retirement queue*, is another buffer that holds all instructions that are currently executing along with space to store their results.

Register renaming: As mentioned earlier, the biggest limiting factor for OoO execution are data dependencies. However, for various reasons, mostly to increase code density, processors ISAs only expose a small number of registers to programs. Consequently, registers are constantly reused for different independent computations which creates a large number of anti-dependencies in the program that hinder parallel execution. To alleviate this problem processors actually have a much larger *physical register file* with many more registers than the *logical register file* visible

at the ISA level. During execution the processor then performs *register renaming* where logical registers are mapped to physical registers to remove false dependencies. That is, instructions in machine code still use the logical registers provided by the ISA, but internally the CPU has a much larger pool of registers. This mapping happens on the fly when instructions are issued from the instruction queue to the reservation stations.

Execution ports A super-scalar processor usually has a large number of different functional units which implement the many different types of instructions. As mentioned before, the most commonly used units are also present multiple times to allow for more parallelism. To simplify runtime instruction scheduling, multiple functional units in the back-end are grouped together behind one *execution port*. That is, the scheduler issues instructions to execution ports which then multiplex them to the right functional unit depending on their type. Since most functional units are usually fully pipelined an execution port can optimally accept one instruction per cycle.

Given the above data structures, the life cycle of an instruction can be split up into multiple distinct steps. While the exact number and names of steps that are distinguished vary between processors and textbooks, here we will distinguish the following phases:

1. *Issue* – This step reads instructions from the instruction queue and is responsible for allocating a reservation station and a slot in the ROB. If no reservation slot or ROB entry is available the instruction and all instructions behind it are stalled.
2. *Execute* – In this step instructions first wait for all their operands to become available and are then dispatched to a functional unit for execution.
3. *Write* – During the Write step instructions store their results to the ROB and to any reservation station waiting for it.
4. *Retire* – In this last step, the result of instructions is copied from the ROB to registers or memory. Once an instruction has finished its ROB slot is freed.

Note that the reservation stations only hold data for an instruction while it waits for its operands and while it is executing. A slot in the re-order buffer, on the other hand, is still needed after the instruction has finished executing and is only liberated at the very end of retirement.

All modern ISAs contain some complex instructions that perform multiple tasks. An example of this would be an instruction that both access memory and performs some computation. To simplify the logic of the back-end the instruction decoder splits such complex instructions into multiple simpler micro-operations (μops). Our example instruction would then be decomposed into distinct load/store and compute μops. This means that the four stages above, the ROB and the reservation stations actually work at the level of μops and not entire instructions. This *μop decomposition* is especially important for complex instruction sets like x86. However, even simpler ISAs such as ARM use it for some instructions.

Since instructions are decomposed into μops, every instruction requires multiple slots in the ROB. Furthermore, the ROB can only retire a limited number of μops per cycle. To reduce pressure on the ROB some CPUs do not completely decompose instructions into μops in the front-end. Instead, as detailed below, they use a technique called *μop fusion* where some of the μops of one instruction can stay *fused* together and share one single ROB entry. Nevertheless, the two fused μops are still executed separately. They each need their own slot in the reservation stations and can be dispatched to different execution ports, but they do retire together. If an instruction has too many register operands its μops can not always be fused and will use multiple slots in the ROB. The Intel documentation calls this *μop unlamination*. It usually occurs for memory accesses with that use complex addressing modes.

Most CPUs that use μops store the last few μops that have been decoded in a μop cache similar to the instruction cache. This avoids repeatedly decoding the same instructions again

and again inside small loops. In this case, the end of the front-end is not the instruction queue but a separate *μop queue* that is fed from both the instruction queue and the μop cache. This μop queue then feeds into the reservation stations and ROB.

Finally, Intel and AMD processor architectures also implement a technique called *macro-op fusion*, or *branch fusion*, to further reduce pressure on the ROB and execution ports. Though the details vary for different architectures, a *macro-op* usually corresponds to a ISA level instruction. This technique is driven by the observation that there are some patterns of instructions that appear together very often and usually have some data dependence. The idea is then to fuse these instructions into a single μop at the decoding stage. Note that this is different from μop fusion since where the fused μops are later split apart again. With macro-op fusion, only one μop is placed in the μop queue which only ever requires one slot in the various queues and buffers of the back-end. The most common example of instructions that get fused are compare instructions and conditional branch instructions which are used in most loops on x86.

Figure 3.1 shows a simplified view of the Intel Skylake CPU architecture which show-cases all the data structures described above.

3.3 Related Work

This section describes previous work related to performance modelling, profiling, and bottleneck detection. The end of the section there also presents existing work in the field of reverse engineering CPU performance models and port mappings.

Analytical models and symbolic execution

Analytical and symbol execution based performance models are purely static approaches using program analysis and abstract mathematical models to estimate the performance of programs or processors.

Probably one of the most well known and performance modelling and visualization method is the Roofline model [221], popularized by Williams et al. in 2009, though variations of the Roofline model had been used long before that [121, 35, 96, 38] In its basic form, the Roofline model is a purely throughput based model consisting of a curve built from the peak GFLOPS and memory bandwidth achievable by a computer system. By comparing measured operational intensity of a program with this curve one can both judge how far a program is from its peak performance as well whether it is compute-bound or memory-bound. Over time the Roofline model has been extended and adapted for a wide range of alternative metrics [105, 133] and to better account for the behaviour of different computer architectures such as FPGAs [189].

The execution cache model (ECM) [210] is another analytical model for predicting the execution time and find bottlenecks in bandwidth limited kernels. Unlike the Roofline model, the ECM does not only consider throughputs, i.e. memory bandwidth and instruction throughput. That is, the ECM also takes data transfer times between different levels of the cache hierarchy into account. All computations, on the other hand, are assumed to overlap and, like the Roofline model, the ECM ignores data dependencies. The ECM also does not take cache hits or misses into account. It simply assumes all data is streamed from memory through the cache hierarchy. Like the Roofline model, the output of ECM model can be used to determine if a program is memory or compute-bound.

The Intel Architecture Code Analyzer (IACA) [95] is a closed-source static binary analysis tool that estimates the instructions per cycle (IPC) of small innermost loop kernels for Intel CPUs. IACA analyses straightline sequences of x86 machine code. It models them as the body of an infinite loop executing in a steady state. As output, it produces a detailed report on the expected IPC, execution port usage, and highlights possible bottlenecks in the CPU

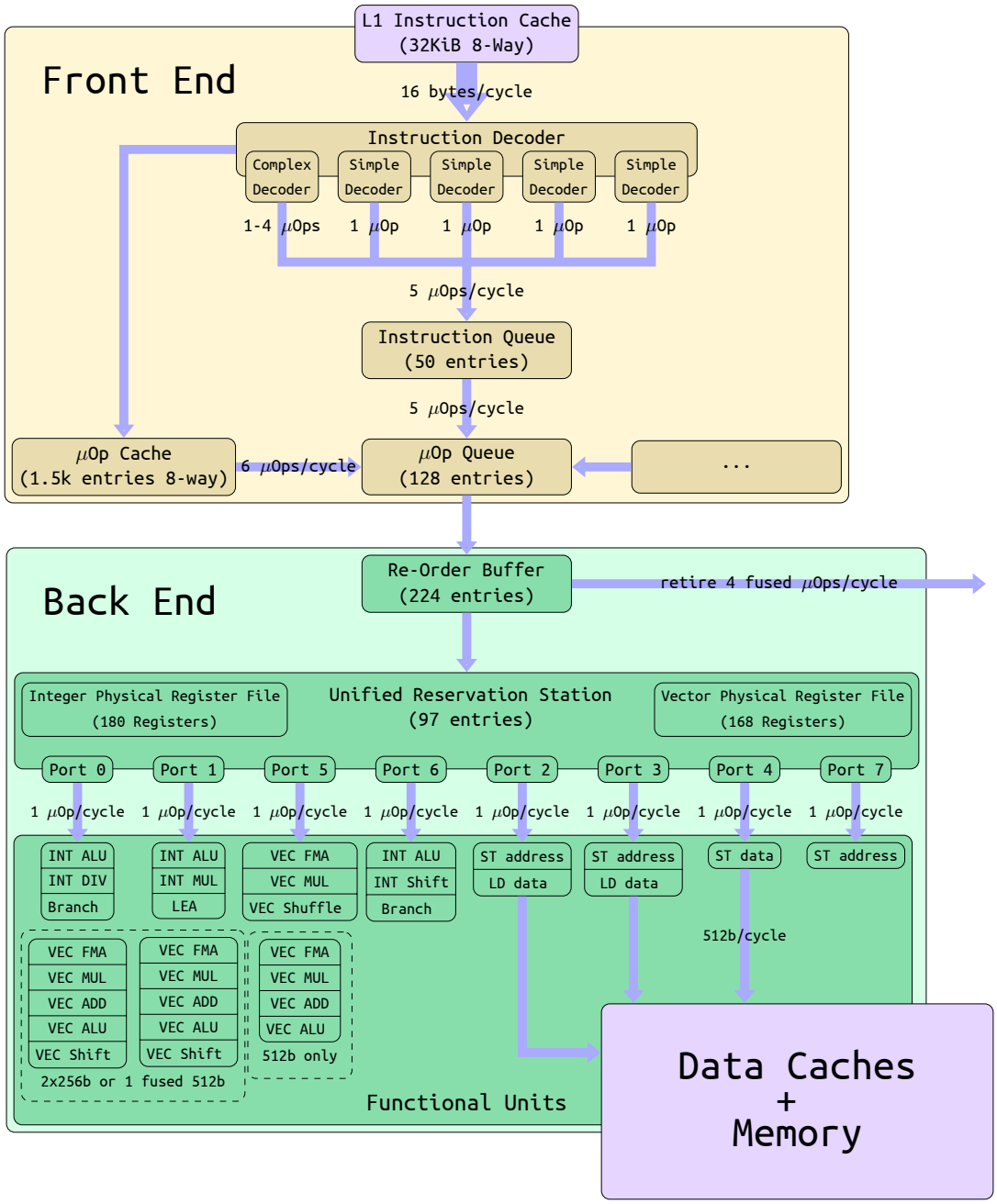


Figure 3.1 – The Intel Skylake CPU architecture. This image has been constructed from information from the Intel SDM [54] and marketing material [5]. Only known throughputs are marked.

pipeline. Earlier versions of IACA also produced information related to instruction latency, but this was discontinued in version 2.2. IACA tracks and models data dependencies via registers but assumes that no memory accesses alias or depend on one another. Contrary to the Roofline model or the ECM IACA focusses solely on the computational throughput of programs and does not take memory or caches into account. It simply assumes all memory accesses to hit in L1. IACA is only updated infrequently, the last supported architecture being Skylake, and has been completely discontinued in April of 2019.

The Open-Source Architecture Code Analyzer (OSACA) [125] and the LLVM Machine Code Analyzer (llvm-mca) [194] are open source versions of IACA. OSACA is developed by the University of Erlangen and llvm-mca [194] was contributed to the LLVM project [124, 165] by Sony. Like IACA, OSACA works at the level of binaries, while llvm-mca parses textual assembler. Also, like IACA both tools work by performing a symbolic execution of an innermost loop kernel. OSACA assumes the loop executes infinitely while llvm-mca executes a bounded, user-defined, number of iterations. OSACA's CPU model is the most abstract of all three tools, it only considers instruction throughput, latency and port usage. llvm-mca, on the other hand, has a more detailed simulation that models details such as instruction dispatch and retirement bandwidth. Neither OSACA nor llvm-mca model instruction decoding, branch prediction, or caches. llvm-mca does, however, model a load/store unit, including out-of-order execution of memory accesses with an x86 like memory consistency model. The machine model of llvm-mca is taken from Low-level Virtual Machine (LLVM) while OSACA's is manually reverse-engineered as described further in Section 3.3.0.2

Kerncraft [86] is a static performance analysis tool that uses both the Roofline and ECM model. Kerncraft works at the level of source code and analyses program written in polyhedral a subset of C. That is, all loops must be perfectly nested and have affine loop bounds. Kernels also can not contain function calls, if statements, pointer arithmetic, or irregular data accesses. The tool can predict scaling potential and help find bottlenecks by constructing Roofline and ECM model for kernels. To statically compute instruction throughput of programs Kerncraft compiles them and passes them to IACA. The cache behaviour of programs is estimated using a simple custom cache simulator. Kerncraft can also perform a symbolic data structure analysis to help determine optimal tile sizes for tiling.

The SAAKE system [99] by Hong et al. is one of the main inspirations for Gus. It uses a fast symbolic execution engine that estimates the runtime of GPU programs to drive sensitivity analysis for finding bottlenecks. SAAKE's input independent abstract simulation works well for the simpler microarchitectures of GPUs since they do not use out-of-order execution or speculation and handle branching control flow using predicated execution. Nevertheless, the authors admit that this approach does not scale to programs with significant divergence or complicated cache usage patterns. Since SAAKE does not actually simulate the execution of instructions, there are several things it can not compute that have to be provided externally. Whether a memory access hits the cache is modelled via fixed per cache level probabilities that are determined via a profiled execution of the kernel. The user also has to provide the number of loop iterations to simulate for each loop via annotations.

Machine learning has also been applied to the space of analytical performance modelling. Joseph et al. have used Radial Basis Function networks to train a non-linear analytical performance model [109]. They applied their model to quickly explore the performance of different CPU architectures. In this approach, one has to train a new network for every new benchmark program used to evaluate the architecture. The input for this training are the parameters of the CPU model, such as the size of the instruction queue and the sizes of caches, along with the execution time of the benchmark program on that architecture. Execution times are obtained using a cycle-accurate simulator. Once trained the model can then be used to predict the execution time of the benchmark programs on new architectures.

Ithemal is a tool that uses hierarchical LSTM to predict the number of cycles required to execute a small linear sequence of code when executed in an infinite loop. For training and evaluation Ithemal converts the machine instructions in a basic block into a sequence of abstract tokens consisting of the opcode as well as source and destination operands, clearly marked as such. The system achieves good prediction rates and requires no expert knowledge to build a model for a new architecture, though until now it has only been evaluated on x86 CPUs.

Profiling tools

This category of performance debugging tools encompasses profilers and other dynamic approaches that analyse performance by reasoning on actual executions of a program, whether in real hardware or in simulators. We also cover performance analysis tools that combine results from static analysis with information dynamically collected at runtime.

HPCToolkit[4], a profiling and performance visualization tool for large scale applications widely used in HPC computing. It does not use any instrumentation and relies solely on the sampling of performance counters and call stacks. The sampled call stacks are used to attribute measurement taken via performance counters to their proper calling context. To map results back from machine code to source code HPCToolkit uses static binary analysis. Its binary analysis module reconstructs function boundaries and loops and identifies inlined code [203]. It can even detect some compiler optimizations, such as loop unrolling, taking them into account for its binary to source code mappings. Overall, HPCToolkit's profiling mode excels at the task of locating performance problems and resource saturation in large applications but does not help with finding root causes of performance problems or the severity of bottlenecks. Besides its basic profiling tools, HPCToolkit also contains a module that uses sensitivity analysis to estimate how well programs scale for parallel execution [132]. It does this by varying the number of cores or compute nodes a program is allowed to run on.

The Tuning and Analysis Utilities (TAU) Performance System [187, 130, 135] is another profiling tool kit for large scale parallel applications written in C/C++, Fortran, or Java. It gathers performance information using instrumentation, which it can do both at the source level and at the binary level. The instrumentation it inserts can collect execution times and hardware performance counters at the granularity of loops and functions. It also provides an API and plugins for popular IDEs that allows programmers to manually instrument programs themselves or designate regions of interest for instrumentation. TAU includes sophisticated visualization tools that illustrate both the aggregate performance of instrumented regions as well as how their performance changed the course over a program's execution. Like HPCToolkit, TAU excels at pinpointing hot program regions and which resources are saturated but does not go beyond this.

Raw execution times of tasks and resource utilization are often not enough to judge by how much a parallel system can still be sped up and where to optimize first. Miller et al. [226] have developed an approach for evaluating the concurrency and finding bottlenecks in parallel programs based on critical path analysis. This analysis is based on the observation that the longest sequential chain of operations, i.e., its critical path, is usually the determining factor of the overall execution time of parallel and distributed programs. The execution time of such systems can thus be approximated by calculating the length of the critical path, that is by summing over the latency of every task on it. The IPS performance measurement system [149, 148] uses critical path analysis to help users diagnose performance problems in parallel and distributed programs. It uses operating system probes and program instrumentation to collect a program from which it computes the critical path of a program. Besides just computing the current critical path of a program IPS is also able to show how the critical path changes if some tasks from it were optimized or removed. This helps guide programmers to where optimization will actually improve overall program performance. Critical path analysis has also been used to find potential for parallelization in sequential programs [120, 73].

DECAN [118] is a dynamic performance analysis tool based on the MAQAO [61, 136] binary analysis and instrumentation framework. It is intended for detailed performance analysis of small kernels and not whole applications. DECAN finds bottlenecks by sensitivity analysis based on binary rewriting. That is, DECAN removes or modifies instructions in a kernel and checks which how much each transformation affects overall performance. It can, for example, remove all floating-point instructions from a loop to detect compute boundedness, or rewrite all memory

accesses to target a single memory address that is held in L1 to detect problems with caches or memory bandwidth. DECAN measures the performance of its modified kernels via hardware counters. The low overhead of this approach allows it to quickly explore a large set of variants for its sensitivity analysis. The downside of DECAN's approach is that its transformations are of course not semantic preserving and can easily introduce crashes or floating-point exceptions. Any transformation that causes such a crash then has to be excluded from the performance analysis. Changing the semantics of a program like this might, of course, also change its performance behaviour in other subtle ways, making it hard to verify or falsify the results produced by the tool.

Code Quality Analyzer (CQA) [39] is another MAQAO based machine code level performance analysis tool. Like DECAN, it is intended to be used to analyse the performance of small kernels and loops. CQA contains a number of static and dynamic analyses that finds low-level performance problems in compute-bound programs. It can, for example, detect problems with vectorization, unaligned memory accesses or register pressure. The output of CQA is a human-readable report that proposes different transformations and their potential benefit. To predict the execution time of the original and transformed program CQA uses a high-level CPU model and a symbolic execution engine similar to that in llvm-mca or OSACA. CQA's symbolic execution engine has no information when branches in a program are taken or not. Instead, it simply computes a separate result for every possible control path. The performance model of CQA does not take cache or memory effects into account and assumes that a kernel is compute-bound.

AutoSCOPE [31, 195] is one of the few performance debugging tools that not only tries to find hotspots and bottlenecks in a program but directly tackles the third problematic of performance debugging: "How can one make the program run faster?" (see Section 1.1). When analysing a program AutoSCOPE finds potential performance problems using HPCToolkit. At the same time, it also measures several performance metrics, such as cache miss rates at different cache levels, branch misprediction rates and IPC. For every potential performance hotspot detected AutoSCOPE then combines these measurements with some static information, such as loop nesting depth, and pattern matches them against a library of well-known performance pathologies. It then uses a ranking algorithm to sort the matching performance pathologies and reports possible optimizations that are known to usually be applicable for fixing them. For example, if AutoSCOPE detects a 2D loop nest that has a large number of TLB and L2 cache misses it will recommend the user to perform a loop interchange or perform tiling. AutoSCOPE does not actually perform any program analysis to verify if the recommended optimizations are legal or profitable or give precise instructions on how to actually perform them. It simply tries to recommend a type of transformation that is generally used for a category of performance problems.

MIAMI [137] is a set of performance analysis tools which tackles similar issues as Gus. That is, MIAMI tries to find both computational and memory performance bottlenecks and by how application performance can still be improved. Like Gus, MIAMI works at the level of x86 machine code, but it functions quite differently, making much heavier use of static analysis. To analyse the computational performance of loops and functions MIAMI computes a static critical-path driven modulo instruction schedule. This schedule is both used to predict execution times and to estimate the load for individual execution ports. MIAMI can estimate how much a program's performance can benefit from different optimizations by changing parameters of the scheduler. To, for example, detect potential for speedup from increasing instruction-level parallelism, MIAMI's scheduler relaxes data dependencies. While MIAMI does use profiling to collect execution frequencies of loops and blocks to help the scheduler, it still implements a generic scheduling algorithm which is not guaranteed to correspond to the actual hardware scheduler. Since it computes static schedules for functions, it is also unable to capture dynamic

interprocedural behaviour. To handle memory bottlenecks MIAMI uses reuse distance profiling and a dynamic analysis to recognise strided access patterns. The reuse distance profiler of MIAMI tracks the dynamic loop and calling context for every data use. Using this tree it can find interprocedural reuse patterns. It also allows MIAMI to detect potential for optimizations like loop interchange or tiling. Finally, memory access stride recognition is used to find memory accesses that are problematic for the hardware prefetcher [138].

3.3.0.1 CPU simulation

Cycle-level simulators are commonly used in the hardware development to explore the performance of new architectures [23, 30, 228]. While they produce very, potentially perfectly, accurate results, they take a large amount of expert knowledge to build and maintain and can be very slow. The very widely used simulator GEM5, for example, when at its highest level of accuracy executes only around 100k instructions per second [180]. At this speed, it takes, on average, around one year to run one benchmark from the SPEC CPU2006 benchmark suite. For larger applications, sampling is commonly used to speed up simulation [188, 224, 180]. With sampling, a simulator does not simulate the entire execution of a program, but only small slices of it. The program parts that are not simulated are either executed natively or handled by a faster, purely functional emulator. This allows even a slow simulator to be able to handle larger applications. For the purposes of implementing GUS, we have decided that simulation at this level of precision is not suitable.

ZSim [179] is an instruction driven CPU simulator. Instead of driving the simulation with a clock and modelling what each component does every cycle, ZSim simulates the behaviour of programs one instruction at a time. Like GUS, ZSim uses dynamic binary instrumentation to insert callbacks into programs to drive its simulation. The authors of ZSim claim that by using this more abstract simulation technique, it achieves orders of magnitude faster simulation times than cycle-level approaches. ZSim still implements a very detailed heavyweight CPU model similar in complexity to that of the above mentioned cycle-level simulators. It encompasses all aspects of a real CPU, from instruction decoding, to the execution pipeline, branch prediction, and caches. ZSim supports simulating large parallel systems with many cores. To this end, it also uses an accurate model of memory coherence on X86 including load-store reordering.

Even more high-level CPU core models, that focus only on simulating some key aspects of a processor have been shown to provide reasonable accurate execution time predictions while running orders of magnitude faster than cycle-level simulation systems [68, 76, 202, 37]. Interval simulation [68, 76] and instruction-window centric simulation [37] only model some components deemed relevant for performance in detail. Other CPU resources are assumed to function at their peak capacity, or close to it, and their behaviour is estimated using rough approximations.

Interval simulation is based around the assumption that modern processors most of the time smoothly run at their peak performance except when hindered to do so by infrequent miss events, like cache misses or branch mispredictions. The core data structure in an interval simulator is a sliding window of currently executing instructions, which simulates the ROB of an out-of-order CPU. Inside the intervals of uninterrupted execution, which are assumed to make up the majority of a program's runtime, the simulator does not track the flow of individual instructions through the CPU pipeline. It merely increases the simulated CPU clock and discards instructions from the instruction window at a steady rate. When a miss event occurs, the simulator uses a more detailed model to determine the performance penalty incurred by the miss. At this time, the instruction window is scanned for data dependencies to detect which miss events are overlapped and which instruction can be drained from the window. A strength of interval simulation, which allows it to achieve excellent performance, is that it only ever processes every dynamically executed instruction in a program once, in program order.

Instruction window centric simulation is an extension of interval simulation that model sacrifices higher prediction accuracy for slightly slower simulation time by using a more precise CPU model. In addition to an instruction window to simulate the ROB this type of simulator also models execution ports in a CPU and takes instruction and memory latencies into account by tracking the critical path along the data dependencies of all instructions in the instruction window. The Sniper multi-core CPU simulator [36] implements both an interval as well as an instruction window centric core model.

Cabezas et al. have combined a high-level CPU based simulator and the Roofline model to build dynamic bottleneck analysis [33]. They estimate execution times are by computing the critical path of the dynamic data-dependence graph of a program. The system not only considers instruction latencies but also includes a cache simulator to take the variable latencies of memory accesses into account. The simulator of Cabezas et al. does not directly work at the level of machine code, but instead executes programs in LLVM IR. To detect bottlenecks and judge their impact on performance the system uses a Roofline model.

3.3.0.2 CPU performance reverse-engineering

Probably the most renowned sources for details on the performance and microarchitecture of x86 processors is the personal web page of Agner Fog [72]. Fog has reverse-engineered the latencies, throughputs and port mappings of instructions on all Intel, AMD, VIA CPUs released since 1996 using handwritten microbenchmarks. Even engineers from Intel itself are known to have used his results [208]. His benchmarks and kernel drivers[1] are freely available. However, the reverse-engineering process is not automated, and the benchmark results have to be analysed manually to generate port mappings. Fog uses hardware counters to measure the activity of each execution port of a machine individually. If these counters are not available Fog's initial information about the port usage for instructions is obtained from other sources, such as official manuals. He then creates benchmarks that mix instructions with known and unknown port usage to infer port mappings. Among all the reverse engineering approaches presented here, Agner Fog's is the only one that reports μop micro-fusion.

The Intel Optimization Reference Manual [53], which is part of the Intel Software Developer's Manual (SDM) [54], contains diagrams giving a rough overview over the port mappings for some of their CPU architectures as well as tables with latencies and throughputs for some commonly used instructions.

EXEgesis [41] is a project from Google that parses the official human-readable Intel SDM [54] to produce a machine-readable ISA description as well as a software instruction encoder and decoder. Since Intel's manuals are not intended to be machine-readable and contain several inconsistencies, this is a very involved process that needs to be updated for every new version of the SDM. EXEgesis can automatically generate microbenchmarks from its ISA description to measure the throughput, latency and port usage of instructions. The tool then uses a mixed-integer linear program running in a custom Simplex solver to find the port mappings of instructions from these measurements. The benchmark generator and measurement parts of EXEgesis have been ported to the LLVM project where they are used to help validate its instruction scheduling models. To find port mappings EXEgesis requires hardware counters for each execution port since their solver needs to know the relative load of every port in a benchmark.

uops.info [3] provides a formalized and automated tool to produce the same results as Agner Fog. Like EXEgesis uops.info generates microbenchmarks to measure the throughput, latency and port usage of instructions. They use this information to detect so-called *blocking*

[1] Agner has used performance hardware counters in his work since before they where commonly exposed by operating systems.

instructions. That is, simple instructions that decompose into only one μop. The μop decomposition and port mapping for these instructions can be found directly using per port hardware counters. To find mappings for more complicated multi-μop instructions they then run multiple microbenchmarks that each mix one complex instruction with several blocking instructions that "blocks", i.e. fully saturates, a sub-set of the execution ports. UOPS.INFO then counts the number of μops executed on the blocked set of ports and detects if all μops of the complex instruction are accounted for or if they have been pushed to another port. By systematically running one such benchmark for every possible combination of ports UOPS.INFO is able to infer the μop decomposition of complex instructions. UOPS.INFO does not detect what μops compose a machine but requires a manually curated list of possible μops as input. Note that, unlike PIPEDREAM, UOPS.INFO does not use throughputs, i.e., instructions per cycle or μops per cycle, to build its model, but the absolute number of executed μops. Consequently, UOPS.INFO assumes that all ports of a machine can be saturated by simple blocking instructions, which is not the case on all X86 architectures. As explained later, UOPS.INFO's uses hardcoded information to handle special cases. For Intel architectures that have per execution port counters UOPS.INFO provides port mappings. They also recently started providing latency and throughput information for AMD Zen+ processors. However, no port mapping since this CPU lacks execution port counters. UOPS.INFO can also parse the output of Intel IACA to produce port mappings and estimate instruction throughput without running any benchmarks. UOPS.INFO was developed concurrently with our work and published only recently in April of 2019.

The OSACA [125] performance modelling tool, already mentioned above, essentially uses a CPU model solely based on the latency, throughput and port usage of instructions. Like GUS, OSACA uses microbenchmarks to measure the performance characteristics of instructions and build a performance model. Unlike the approaches described above, OSACA only measures latencies and throughputs and does not require per execution port performance counters. The construction of their performance model, however, is not automated. In [125] the authors sketch out a manual procedure to construct port mappings from the latency and throughput measurements for kernels that mix different instructions with known and unknown port mapping. Their procedure is similar to that used by GUS, but it requires the user to already roughly know what classes of instructions each execution port can execute. It can then find the port mapping for new instructions based on the known mapping of other instructions. The OSACA distribution already includes a number of port mappings for Intel and AMD processors constructed using this process.

Like OSACA, CQA [39] also uses instruction latencies, throughputs and port mappings for its internal performance model. Likewise, CQA also uses microbenchmarks to measure this information. It does, however, not rely on handwritten benchmarks, but generates them from a machine description. CQA only constructs a port mapping for simple instructions like the blocking instructions used by UOPS.INFO. For complex instructions it falls back to the mappings reported by Agner Fog.

3.4 GUS

This section describes GUS, our high-level CPU simulator used to predict execution times of programs and drive our sensitivity analysis for finding performance bottlenecks. We start with a high-level overview of the CPU performance model used in GUS. Section 3.4.2 gives an illustrative example where we run the simulator on a small program. Section 3.4.3 finally shows the pseudo-code of the simulator's main algorithm.

3.4.1 A resource-centric CPU model

Here we present the abstract performance model used by GUS to compute the execution time of programs used to drive its sensitivity analysis. Our current implementation focuses on modelling modern general purpose out-of-order CPUs composed of:

- A ROB.
- Execution ports that group functional units.
- A multi-level cache hierarchy.

In such architectures, instructions are first decoded and decomposed into μops that are then issued to the ROB. Once data dependencies have been resolved, after register renaming, μops are dispatched to the execution ports and get retired when their results are committed to registers or memory.

We focus on modelling the following three sources of bottlenecks for these kinds of architectures:

- The size of the ROB is not sufficient to hide the lack of instruction level parallelism. That is, it is filled with issued but not yet retired μops.
- Long latency instructions, including loads with a cache miss, causing an overly long critical path.
- One or more functional units are saturated.

GUS is an instruction driven simulator, similar to ZSim [179] or Sniper [36]. It is split into two parts:

1. The front-end, which is built around QEMU, a purely functional CPU emulator. It uses dynamic binary instrumentation to generate the stream of events that drive the simulation. One event is generated per instruction executed in a program. This event records the registers and memory addresses read and written by that instruction as well as the list of CPU resources it uses.
2. The abstract performance model, that consumes these events and tracks the state of the simulated CPU. The performance model computes the time at which every instruction is issued, at which it starts executing, and when it is retired. The overall execution time of a program is then the time the last instruction finishes. The abstract performance model consists of the following components:
   - A set of abstract throughput limited resources that model, amongst other things, the execution ports and the bandwidth between different levels of the cache.
   - A finite-sized instruction window which models the ROB.
   - A shadow memory and shadow register file used to track data dependencies.
   - A cache simulator to detect cache misses.

   The time at which an instruction can be issued, is dictated by the instruction window. The time it can start executing, $\texttt{t}_{\texttt{start}}$, is determined by the resources it uses and by its data dependencies. The time it retires, $\texttt{t}_{\texttt{end}}$, is simply $\texttt{t}_{\texttt{start}} + \text{instruction.latency}$.

Abstract resources:

The main element of GUS's CPU model are "abstract" throughput limited resources. Every instruction uses one or more of these resources as it executes. There is no global clock in the simulator that tracks the current time. Instead, for each resource, it only tracks the time when that resource will be available to accept another request. When all the resources an instruction requires are available, it can execute. Otherwise the simulated CPU stalls. Every resource is characterized solely by its throughput, i.e., the rate at which it can process instructions. Examples of things modelled using such abstract resources are the execution ports, the bandwidth between the L1 and L2 cache, or the retirement rate of the ROB.

We write the timestamp at which a resource R is available to accept an instruction as $\texttt{R.t}_{\texttt{avail}}$. Every time a resource is used its $\texttt{t}_{\texttt{avail}}$ is incremented by its *inverse throughput*. We also refer to the inverse throughput of a resource as its *gap*, since it determines the minimum amount of time that must elapse between two uses of the resource.

To model the throughput and sharing of CPU resources the standard formalism is a port mapping, a tripartite graph, which describes how instructions decompose into μops and which functional units μops can execute on. Gus instead uses a simpler two-level representation, called a *resource mapping*, where instructions are directly associated with a list of abstract resources. To account for μop decomposition, a resource can appear in this list multiple times. The details of how we create Gus's two-level representation from a three-level port mapping are explained later in Section 3.5.3.

Instruction window:

The instruction window is a fixed size buffer that models the ROB of the CPU. It that tracks all instructions that have been issued, but that have not been retired yet. The instruction window does not store instructions, but only timestamps that indicate when each instruction will retire. If the window is full no instruction can be issued until another one retires. We use $\texttt{t}_{\texttt{min}}$ to denote the earliest time a slot in the window will become free. $\texttt{t}_{\texttt{min}}$ is thus the earliest time the current instruction can be issued to the window. The instruction window effectively puts a limit on the amount of instruction-level parallelism the simulator allows.

The only two operations we perform on the instruction window are querying for the smallest element and replacing the smallest element. Both these operations can be efficiently implemented with a min-heap data structure.

Shadow memory, shadow register file, and cache simulator:

The shadow memory and shadow register are used to detect data dependencies between instructions. For each memory cell or register they store the time at which the data in this location will be available. It is updated through two mechanisms: First, when an instruction writes to a location, the shadow cell for that location will be set to the time when the instruction is retired. Second, when a cache miss occurs this counts as a "use" of all levels of the memory hierarchy up to the one where the miss occurred. The location in the shadow memory corresponding to the accessed memory location is then updated to the maximum of $\texttt{t}_{\texttt{end}}$ of the instruction that caused the miss of the $\texttt{t}_{\texttt{avail}}$ of the involved resources plus their gap. Gus uses the Dinero IV cache simulator to detect at which level of the memory hierarchy every memory access hits or misses. Gus assumes that register renaming works perfectly and only considers flow, or read-after-write, dependencies.

3.4.2 An illustrative example

In this section we unroll how the simulator processes a small example program. In this example we will use a simple CPU model which consists of:

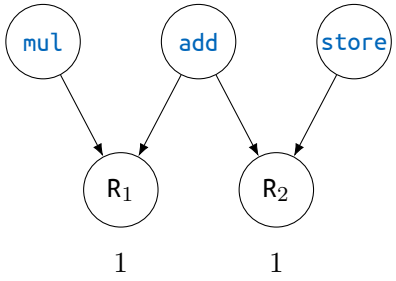
- Three different instructions: `add`, `mul`, `store`.
- Three resources: $\texttt{R}_1$, $\texttt{R}_2$ and $\texttt{R}_{12}$.
- An instruction window, with two slots. There are no μops in this example.
- three registers: `%r1`, `%r2`, and `%r3`.
- A memory hierarchy with two level of cache, modelled as two resources L1↦L2 and L2↦M. L1↦L2 denotes the bandwidth between the L1 and L2 cache. L2↦M denotes the bandwidth between the L2 cache and main memory. The bandwidth between the CPU and L1 is assumed to be infinite

Resource	Throughput	Gap
R1	1	1
R2	1	1
R12	2	½
L1↦L2	½	2
L2↦M	⅓	3

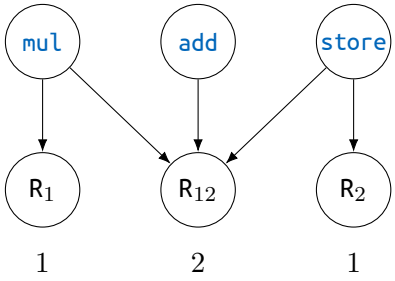
(a) Throughputs and gaps of resources.

Instruction	Latency
add	1
mul	3
store	1

(b) Instruction latencies



(c) Port mapping. `mul` and `store` can only execute on resources R1 and R2, respectively. `add` can use *either* resource R1 *or* R2.



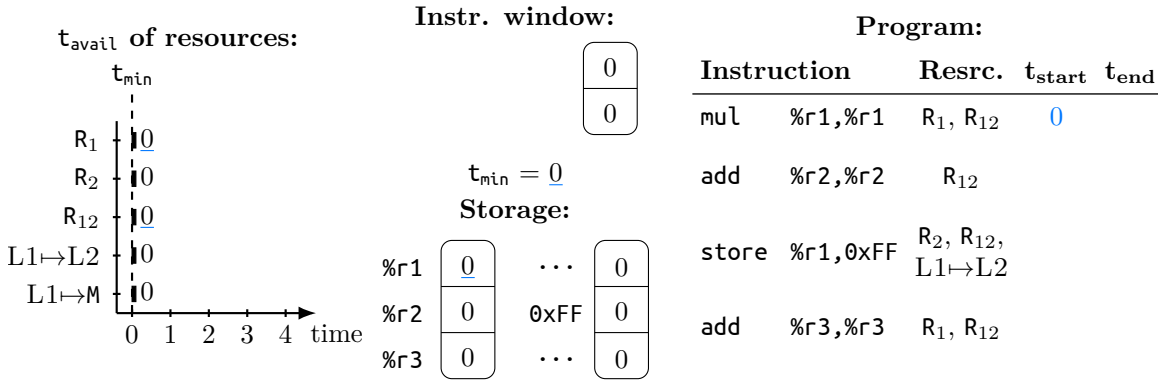
(d) Corresponding resource mapping. The fact that `add` can execute either on resource R1 or R2 is modelled by the combined resource R12. `mul` uses both R1 *and* R12

Figure 3.2 – Port mapping, resource mapping, and throughputs and gaps for individual resources of simple example CPU. In both the port and resource mapping resources are annotated with their throughput. `store` instructions also use L1↦L2 or L2↦M if a cache miss occurs.

The port mapping for this simple toy architecture and the resource mappings constructed from it are shown in Figures 3.2a and 3.2b. Figure 3.2c lists the throughputs of all resources. Figure 3.2d lists the latencies of instructions. One important detail here is the fact that the μop produced by `add` can execute *either* on resource R1 *or* R2. This is modelled in the corresponding resource mapping by creating a new *combined* resource R12. The μop for `mul`, which originally only used resource R1 in the port mapping now uses R1 *and* R12. The same holds for `store`, which goes from using only R2 to using R2 *and* R12. This combined resources R12 models the resource sharing between `add`, `mul`, and `store`. The details of how exactly we construct resource mappings are explained later in Section 3.5.3.

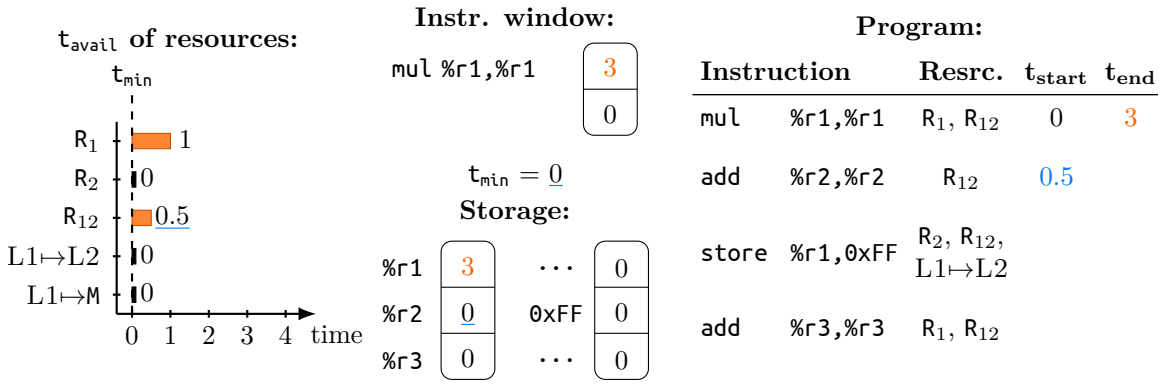
The example program consists of four machine instructions, and we visualize the state of the simulator after every instruction has finished executing. In every image, the t_{start} for the current instruction is shown highlighted in blue. All values used to compute t_{start} are underlined in blue. The t_{end} of every instruction and all state of the simulator that has changed while executing the instruction is highlighted in orange.

Initial state:
The image below shows the initial state of the simulator. Initially, t_{avail} of all resources, all slots in the instruction window, and all locations in the shadow memory and shadow register file are set to zero. At this point, we can already calculate the value for t_{start} of the first instruction, `mul %r1,%r1`. Any instruction can execute once a slot in the instruction window, all resources it uses, and all its data dependencies are available. `mul %r1,%r1` uses R1, R12 and `%r1`, so $t_{start} = \max(R1.t_{avail}, R12.t_{avail}, \texttt{\%r1}.t_{avail}) = 0$.



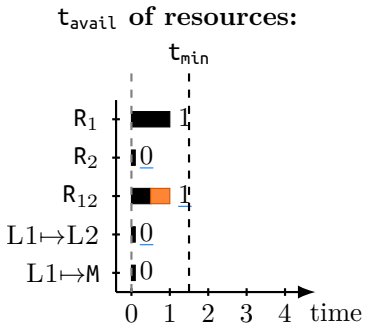
After instruction 1:
The next image shows the state of the simulator after the first instruction has finished executing. t_{avail} of all resources used by the instruction, here R_1 and R_{12}, has been incremented by the resources gap. For any instruction i t_{end} is defined as $t_{end} = t_{start} + i.\text{latency}$. Since here a `mul` has a latency of three, we calculate $t_{end} = 3$. This value for t_{end} is recorded in the instruction window and in the shadow register for `%r1`.

After executing the first instruction, we can calculate t_{start} for the second instruction, `add %r2,%r2`. This instructions uses R_{12} and reads from `%r2`, so $t_{start} = 0.5$.



After instruction 2:
The next step shows the state after the second instruction, `add %r2,%r2`. As before we update t_{avail} for all used resources and the shadow registers that are written to. This instruction fills the last available slot in the instruction window, so the t_{min} for the next instruction is updated.

The third instruction, `store %r1,0xFF`, has a data dependency on the first instruction via register `%r1`. Consequently it cannot begin executing before `mul %r1,%r1` has finished, even though both the instruction window and the resources it uses would allow it to execute earlier.



Instr. window:

`mul %r1,%r1`	3
`add %r2,%r2`	1.5

$t_{min} = 1.5$

Storage:

`%r1`	3	···	0
`%r2`	1.5	`0xFF`	0
`%r3`	0	···	0

Program:

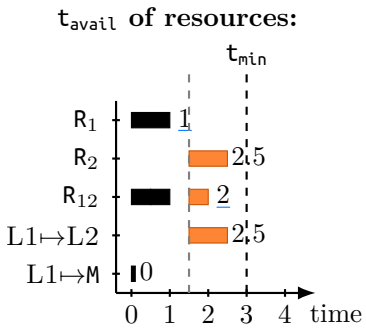
Instruction		Resrc.	t_{start}	t_{end}
`mul`	`%r1,%r1`	R_1, R_{12}	0	3
`add`	`%r2,%r2`	R_{12}	0.5	1.5
`store`	`%r1,0xFF`	R_2, R_{12}, L1↦L2	3	
`add`	`%r3,%r3`	R_1, R_{12}		

After instruction 3:

The third instruction, `store %r1,0xFF`, has to wait for a slot in the instruction window to be available before it can be issued. Consequently, the instruction will not request any resources before it gets that slot at timestamp 1.5. We model this bubble in the CPU pipeline by setting t_{avail} of all used resources to $t_{avail} = \max(t_{min}, t_{avail})$. No other instruction could have used the resources during this bubble since there was no free slot in the instruction window. We then, as always, increment t_{avail} of every used resource by its gap.

The third instruction is a memory-access that hits in L2, i.e., it causes a miss in L1. Thus, besides R_2 and R_{12} it also uses the resource L1↦L2. We also have to update the shadow memory cell for address `0xFF` to (L1↦L2).t_{avail}. The shadow memory cell for `0xFF` is then later overwritten with the t_{end} of the `store`.

The fourth instruction, `mul %r3,%r3`, again has to wait for a slot in the instruction window, even though there are available resources and no data dependencies. It can, at least, execute in parallel with the store.



Instr. window:

`mul %r1,%r1`	3
`store %r1,0xFF`	4

$t_{min} = 3$

Storage:

`%r1`	3	···	0
`%r2`	1.5	`0xFF`	4
`%r3`	0	···	0

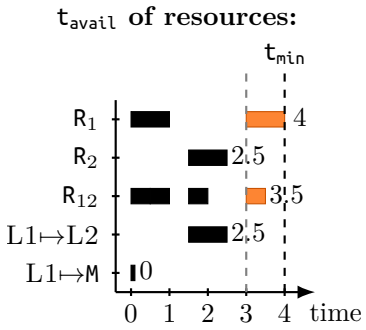
Program:

Instruction		Resrc.	t_{start}	t_{end}
`mul`	`%r1,%r1`	R_1, R_{12}	0	3
`add`	`%r2,%r2`	R_{12}	0.5	1.5
`store`	`%r1,0xFF`	R_2, R_{12}, L1↦L2	3	4
`add`	`%r3,%r3`	R_1, R_{12}	3	

After instruction 4:

The fourth, and last, instruction, `mul %r3,%r3`, waits until timestamp 3 to get a free slot in the instruction window. Same as with the `store` this causes a bubble in the CPU pipeline where some resources are unused.

This last instruction starts executing concurrently with the `store` instruction and also finishes at the same time. We can see that, in this simple example, the size of the instruction window is not big enough to mask the latency of the `mul` and cache miss of the `store`.



Instr. window:

`add %r3,%r3`	4
`store %r1,0xFF`	4

$t_{min} = 4$

Storage:

`%r1`	3	···	0
`%r2`	1.5	`0xFF`	4
`%r3`	4	···	0

Program:

Instruction		Resrc.	t_{start}	t_{end}
`mul`	`%r1,%r1`	R_1, R_{12}	0	3
`add`	`%r2,%r2`	R_{12}	0.5	1.5
`store`	`%r1,0xFF`	R_2, R_{12}, L1↦L2	3	4
`add`	`%r3,%r3`	R_1, R_{12}	3	4

3.4.3 The simulator algorithm

Algorithm 3.2 shows the main algorithm used by Gus to simulate CPUs. One important factor that helps keep the algorithm simple and efficient is that it does not directly model a scheduler. In real super-scalar processors the scheduler multiplexes instructions and μops onto the functional units or execution ports. Accurately modelling this would require maintaining ready queues and reservation stations. As illustrated in Figures 3.2a and 3.2b Gus instead models this using optimistically combined resources. In a port mapping we map μops to a list of ports; whenever the μop executes it uses *one* of these ports. Contrarily, in a resource mapping we map instructions to a list of abstract resources; whenever the instruction executes it uses *all* of these resources. Any resource can repeatedly appear in the list of an instruction, which allows modelling instructions that decompose into multiple μops of the same type. Algorithm 3.1 shows how we convert port mappings to resource mappings.

We use the Pipedream tool described in Section 3.5 to find port mappings for CPU architectures. Pipedream also determines the baseline throughput of all resources as described in Section 3.5.3. The throughput of a combined resource is equal to the throughputs of all sub-resources it is composed of.

As stated earlier, Gus finds performance bottlenecks using sensitivity analysis. That is, we run the simulator multiple times, each time increasing the throughput of one resource. We can vary the throughput of "atomic" resource built from only one port as well as the throughput of combined resources. We try multiple different throughput values for every resource. To limit the size of the search space that needs to be explored, we only ever change the throughput of one resource. Gus can also detect if instruction latency is a bottleneck by varying the latency of instructions. For now, we only use one global factor that affects the latency of all instructions. This factor is multiplied with the instruction latency on line 19 of the simulator algorithm. At the end, Gus reports how sensitive the program's performance is to each resource or latency. That is, by how much an increase of every resources throughput or instruction latency reduces the total program runtime.

Gus intentionally uses a very simple performance model to achieve good simulation times. Consequently, the simulator algorithm has to make a number of simplifying assumptions about programs and the underlying hardware:

- Like the ECM [210], we assume that the program has regular access memory patterns that can be perfectly prefetched. It is assumed that all memory accesses can be overlapped and so we do not take the latency of caches into account.
- The simulator has no model of memory consistency and allows memory accesses to be arbitrarily reordered.
- We currently assume that the bandwidth between the CPU and L1 is infinite and do not model it as a resource. The only penalty one incurs on an L1 hit is the latency of the

```
1  struct Muop:
2    # The muop uses one of these ports when it executes.
3    ports: Set[Port]
4
5  struct Instruction:
6    # Non-empty list of muops the instruction decomposes into
7    muops:     List[Muop]
8    # List of abstract resources used by the instruction.
9    # The instruction uses all these resources when it executes. Initially empty
10   resources: List[Resource]
11
12 def build_resource_model(port_mapping):
13   # Abstract resources that model the execution ports of a CPU.
14   # A set combining multiple ports is a combined resource.
15   resources: Set[Set[Port]] = {}
16
17   # Collect all sets of ports that used by muops in the port mapping.
18   for instruction in port_mapping:
19     for muop in instruction.muops:
20       resources ∪= {muop.ports}
21
22   # Create any missing port sets needed to model resource sharing.
23   while no fixed point is reached:
24     for port_set_1 in port_sets:
25       for port_set_2 in port_sets:
26         if port_set_1 ∩ port_set_2 != ∅:
27           resources ∪= {port_set_1 ∪ port_set_2}
28
29   # Assign resources to instructions
30   for instruction in port_mapping:
31     instructions.resources = []
32     for muop in instruction.muops:
33     for port_set in resources:
34       if muop.ports ⊆ port_set:
35         instructions.resources += port_set
```

Algorithm 3.1 – Algorithm to build a resource mapping from a port mapping. This only covers resources representing execution ports.

memory access instruction (line 6).

- Memory accesses do not wait for the address operand for the purposes of calculating cache misses. Instead, we assume memory requests happen as soon as the instruction has a slot in the instruction window. This models, to some degree, the ability of modern CPUs to speculate on the address of memory accesses (line 7).
- We do not model the write-result part of an instruction's execution (line 33).
- The simulator does not model load/store queues or load-store forwarding.
- The simulator does not model execution pipeline hazards or operand forwarding.
- We do not direcly model branch prediction or speculative execution. All branches are assumed to be correctly predicted and branches do not cause stalls in the CPU pipeline.

```
for instruction in trace:
  # [1]. Start memory requests, i.e., access caches.
  for loc in instruction.reads | instruction.writes:
    # Lowest level at which access hits.
    level = access_location(loc)
    if level in [L2, L3, MEM]:
      level.t_avail = max(level.t_avail, t_min) + level.gap
      shadow[loc] = max(shadow[loc], level.t_avail)
  # [2]. Compute time instruction can start executing.
  #      Start when a slot in the instruction window is available,
  t_start = t_min
  # [2.1]. Compute time all inputs are available.
  for loc in instruction.reads:
    t_start = max(shadow[loc], t_start)
  # [2.2]. Compute time all resources are available.
  for resource in instruction.resources:
    t_start = max(resource.t_avail, t_start)
  # [3]. Compute time execution of instruction finishes.
  t_end = t_start + instruction.latency
  # [4]. Update resource usage.
  for resource in instruction.resources:
    resource.t_avail = max(resource.t_avail, t_min) + resource.gap
  # [5]. Update shadow memory.
  for loc in instruction.writes:
    if loc is a register:
      shadow[loc] = t_end # ignore WAW, assume register renaming works perfectly
    else:
      shadow[loc] = max(shadow[loc], t_end)
  # [6]. Update tmin & window.
  window.push(t_end) # overwrites earliest available slot
  t_min = window.earliest_available_slot()
  # 7. Record t_end for instruction.
  instruction.t_end = t_end
```

Algorithm 3.2 – The Bottleneck Simulator Algorithm

3.5 Pipedream

This section describes how Pipedream builds the resource CPU model for Gus. This is a two-step process. First, Pipedream reverse engineers the following performance characteristics:

- Instruction latency – The number of cycles an instruction requires to execute.
- Peak μop retirement rate – How many fused μops the CPU can retire per cycle.
- Micro-fusion – The number of fused μops an instruction decomposes into.
- μop decomposition and μop port usage – The list of unfused μops every instruction decomposes into and the list of execution ports every one of these μops can execute on.

These real CPU resources are then mapped to and combined into abstract resources.

The first step of the reverse engineering process consists of generating a number of microbenchmarks. Pipedream then runs these benchmark, measuring their performance using hardware counters. The latency, throughput, and micro-fusion of different instructions can then be read directly from these measurements. The process of finding port mappings, i.e. μop decompositions and μop port usage, however, is more involved. For this purpose, we have de-

fined a variation of the maximum flow problem which we call the instruction flow problem. We have developed a linear program (LP) formulation of the instruction flow problem which can be used to calculate the peak IPC and micro-operations per cycle (MPC) a benchmark kernel can theoretically achieve with a given port mapping. The actual port mapping of the underlying hardware is then determined by finding the mapping for which the throughput predicted by instruction flow best matches the actual measured IPC and MPC.

The counters required by PIPEDREAM for its measurements the are:

- An accurate cycle counter, to calculate the IPC.
- A counter for the total number of *retired* μops, to calculate the *fused* MPC.
- A counter for the total number of μops *executed* on any execution port, to calculate the *unfused* MPC.
- A counter for each execution port that counts the number of unfused μops executed on that port.

All of these counters are available on Intel CPUs since Sandy Bridge. For now, we have built a model of the Skylake architecture.

The CPU model used in instruction flow is even simpler than that of GUS. It only considers instructions, μops, and the throughput of execution ports. Any other components, such as the ROB, the instruction scheduler, or latencies, are not directly taken into account. The benchmarks we apply the model to are all constructed in a way so that these resources never form a bottleneck.

In the following we will use $[i_1 i_2 \dots i_n]$ to denote a microkernel consisting of n instructions i_1 through i_n. By construction our kernels never have any data dependencies, so the order of instructions in a kernel is not relevant. That is, $[i_1 i_2] = [i_2 i_1]$. The number of occurences of an instruction in a kernel, however, is relevant. Thus, $[i_1 i_1 i_2] \neq [i_1 i_2]$. We the plus symbol to denote the concatenation of kernels, i.e. $[i_1] + [i_2] = [i_1 i_2]$. We also use the shorthand A^N to denote N repetitions of instruction A. So, for example, $[i_1^2 i_2^3] = [i_1 i_1 i_2 i_2 i_2]$.

Throughout the rest of this chapter we use a variation of Agner Fog's notation for execution ports, μops, and μop decomposition [72]. Execution ports are noted as $p_0 \dots p_n$. The identifier for a μop is simply the concatenated sequence of ports it can execute on. That is, a μop that can execute only on port p_4 will be written as p_4. A μop that can execute on either p_1 or p_3 is denoted as p_{13}. For simplicity, we will assume that there are at most ten execution ports so that they can be labelled with a single digit. A μop decomposition is simply the list of identifiers for every μop. If a μop occurs multiple times in a decomposition, it is prefixed with the number of occurrences. An instruction that decomposes into three μops, two of type p_{015} and one of type p_6, is written as $2p_{015}\ p_6$.

3.5.1 The instruction flow problem

Before describing the algorithm for finding port mappings, we introduce the instruction flow problem. The instruction flow problem is a variation of the maximum flow problem which allows for constraints that force the total flow that passes through two nodes to be equal. We use the instruction flow to model the steady state of a CPU with a given port mapping when it is executing a small microkernel in an infinite loop. The "flow" through such an instruction flow problem models the rates at which instructions and μops execute on the execution ports of the CPU. That is, a solution for an instance of the instruction flow problem directly encodes the IPC, MPC and execution port usage a kernel can achieve. Solving instruction flow instances is the core mechanism used in PIPEDREAM to evaluate and find port mappings.

PIPEDREAM uses the same notion of μop as GUS. That is, a μop is simply a list of execution ports, and we do not consider the latency of μops nor data dependencies between them.

We define an instance of the instruction flow is a multipartite directed acyclic graph (DAG)

$G = (V, E)$, where $V = (\{k_{IPC}\} \uplus I \uplus M \uplus P \uplus \{k_{MPC}\})$ is a set of vertices and $E \subseteq V \times V$ is a set of edges. V is partitioned into several disjoint subsets:

- $\{k_{IPC}\}$ – This singleton set contains the source vertex k_{IPC} through which flow enters the system. Conceptually, it is this vertex from which instructions flow into the CPU to execute, and so the flow entering at this point is the total IPC for the kernel.
- I – The set of vertices I represents the different instructions in the kernel. There is only one instruction vertex per *type* of instruction. I.e., if a kernel contains two `add` instructions there will be only one $i_{\texttt{add}} \in I$ vertex to model this.
- M – The set of vertices M represents the different μops that instructions decompose to. Some instructions can decompose into more than one μop of a given type, but for simplicity, we do not model this directly. Instead, if an instruction i decomposes into n instances of μop m there will be n distinct vertices $m_1, \ldots, m_n$.
- P – The set of vertices P represents the different execution ports of the modelled CPU.
- $\{k_{MPC}\}$ – This singleton set contains the sink vertex k_{MPC} through which flow leaves the system. Conceptually, it is this vertex at which μops execute and leave the CPU, so the flow leaving through this vertex is the total MPC for the kernel.

No two vertices inside one of the partitions in V can be connected by an edge, and there are only edges between partitions. Only edges between the following pairs of partitions are allowed:

- From $\{k_{IPC}\}$ to I – These edges indicate which instructions the kernel contains. Every instruction is connected to k_{IPC}.
- From I to M – These edges give the decomposition of instructions into μops.
- From M to P – These edges describe which execution port a μop can execute on.
- From P to $\{k_{MPC}\}$ – These are the edges through flow leaves the system. Every port is Every port is connected to k_{MPC}.

We usually omit k_{IPC} and k_{MPC} when drawing instruction flow graphs.

μop and port vertices follow the standard rules for the conservation of flow in a flow network. That is, the sum of all flow entering such a vertex via its incoming edges equals the flow leaving it via its outgoing edges. This does, however, not hold for instruction vertices. When an instruction decomposes into multiple μops it puts more pressure on the execution ports than an instruction that only produces one μop, i.e. the MPC of the kernel will be larger than its IPC. Consequently, when an instruction decomposes into more than one μop the flow leaving the vertex representing it in the instruction flow graph is larger than the incoming flow. To be precise, the total outgoing flow of an instruction vertex is equal to the incoming flow times the number of μops it decomposes into. On the other hand, when a μop is connected to more than one port, this means that the μop can execute on either of these ports, but it only goes to one of them. So at this level conservation of flow is upheld and the MPC is distributed along the different outgoing edges.

Besides the graph G, an instruction flow instance of also encompasses a set of constants that further describe the kernels and certain characteristics of the CPU architecture being modelled.

$o_i \in \mathbb{N}_0$ – The number of times an instruction $i \in I$ occurs in the kernel. If $o_i = 0$ then $(k, i) \notin E$. Likewise, if $o_i \neq 0$ then $(k, i) \in E$.

$\lambda_{\max} \in \mathbb{R}^+$ – The maximum fused MPC attainable on the CPU architecture. This is simply the maximum fused MPC observed during any benchmark run. This models the retirement bottleneck of the ROB. We need to take this into account since Intel CPUs can theoretically execute more μops than they can retire.

$\#\lambda_i \in \mathbb{N}$ – The number of fused μops an instruction $i \in I$ decomposes to.

$\#\mu_i \in \mathbb{N}$ – The number of unfused μops an instruction $i \in I$ decomposes to. This corresponds to the number of outgoing edges of the the i vertex.

A solution of an instance of the instruction flow assigns a weight to every vertex and edge in G. These weights correspond to IPC or MPC that passes through that part of the graph. We

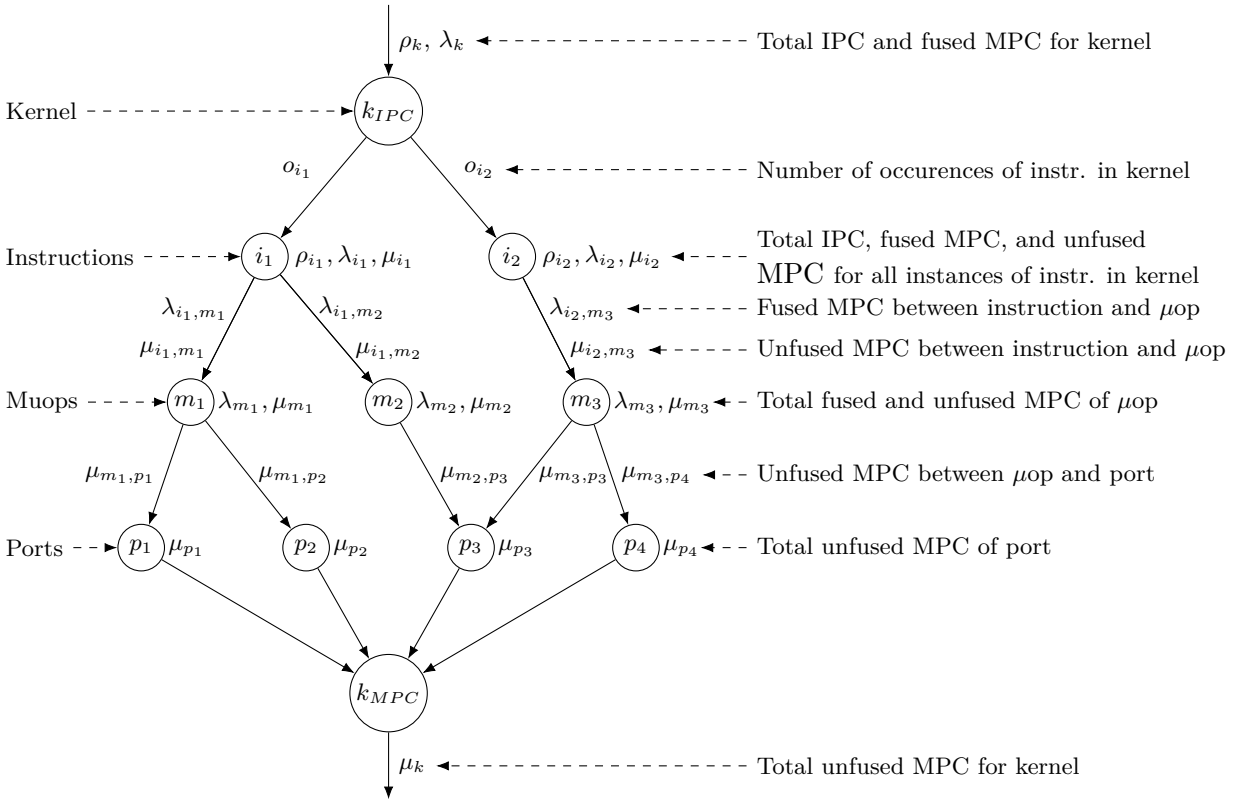


Figure 3.3 – Example instance of the instruction flow problem. An instance of an instruction produces one μop for every outgoing edge, i.e., i_1 decomposes into one m_1 *and* one m_2. One instance of a μop only executes on one port, i.e., m_1 can execute either on p_1 *or* p_2.

use the following notation for the weights of the different types of vertices and edges:

- ρ_k – The total IPC of kernel.
- λ_k – The total fused MPC of kernel.
- ρ_i – The IPC for all instances of instruction i in the kernel. The same instruction can have different IPCs in different kernels due to differing execution port usage.
- λ_i – The total fused MPC of all μops generated for all instances of instruction i in the kernel.
- μ_i – The total unfused MPC of all μops generated for all instances of instruction i in the kernel.
- $\lambda_{i,m}$ – The fused MPC of μop m generated from all instance of instruction i in the kernel.
- $\mu_{i,m}$ – The unfused MPC of μop m generated from all instance of instruction i in the kernel.
- λ_m – The total fused MPC of all μops generated for all instructions in the kernel.
- μ_m – The total unfused MPC of all μops generated for all instructions in the kernel.
- $\mu_{m,p}$ – The unfused MPC of μop m executed on port p in the kernel.
- μ_p – The total unfused MPC of all μops executed on port p in the kernel.
- μ_k – The total unfused MPC of kernel.

An example instance of the instruction flow problem indicating which weight is on which node or edge can be seen in Figure 3.3

A valid solution for an instance of the instruction flow problem must maximize the total flow given the following constraints:

C_1 – $\rho_k = \sum_{i \in I} \rho_i$

$$C_2 - \quad \lambda_k = \sum_{i \in I} \lambda_i$$
$$C_3 - \quad \mu_k = \sum_{p \in P} \mu_p$$

(total flow in and out of system)

$$C_4 - \forall i \in I. \quad \mu_i = \sum_{m \in M} \mu_{i,m}$$
$$C_5 - \forall m \in M. \quad \mu_m = \sum_{i \in I} \mu_{i,m}$$
$$C_6 - \forall m \in M. \quad \mu_m = \sum_{p \in P} \mu_{m,p}$$
$$C_7 - \forall p \in P. \quad \mu_p = \sum_{m \in M} \mu_{m,p}$$

(conservation of flow)

$$C_8 - \forall i \in I. \quad (k_{IPC}, i) \notin E \implies \rho_i = 0$$
$$C_9 - \forall i \in I, m \in M. \quad (i,m) \notin E \implies \mu_{i,m} = 0$$
$$C_{10} - \forall m \in M, p \in P. \quad (m,p) \notin E \implies \mu_{m,p} = 0$$

(flow only propagates along edges)

$$C_{11} - \forall i_1, i_2 \in I. \quad (o_{i_2} > 0 \ \wedge \ o_{i_2} > 0) \implies o_{i_2} * \rho_{i_1} = o_{i_1} * \rho_{i_2}$$

(IPC of kernel is split evenly among instructions)

$$C_{12} - \forall i \in I, m \in M. \quad (i,m) \in E \implies \rho_i = \mu_{i,m}$$

(unfused MPC is split evenly among μops for an instruction)

$$C_{13} - \forall i \in I. \quad \rho_i = \frac{1}{\#\mu_i} * \sum_{m \in M} \mu_{i,m}$$ (translate IPC to unfused MPC)

$$C_{14} - \forall i \in I. \quad \lambda_i = \frac{\#\lambda_i}{\#\mu_i} * \mu_i$$ (translate fused MPC to unfused MPC)

$$C_{15} - \forall p \in P. \quad \mu_p \leq 1$$ (a port executes at most one μop per cycle)

$$C_{16} - \quad \lambda_k \leq \lambda_{max}$$ (μop retirement limit)

$$C_{17} - \forall i \in I. \quad \mu_i \leq \mu_{[i]}$$ (bottlenecks other than retirement or ports)

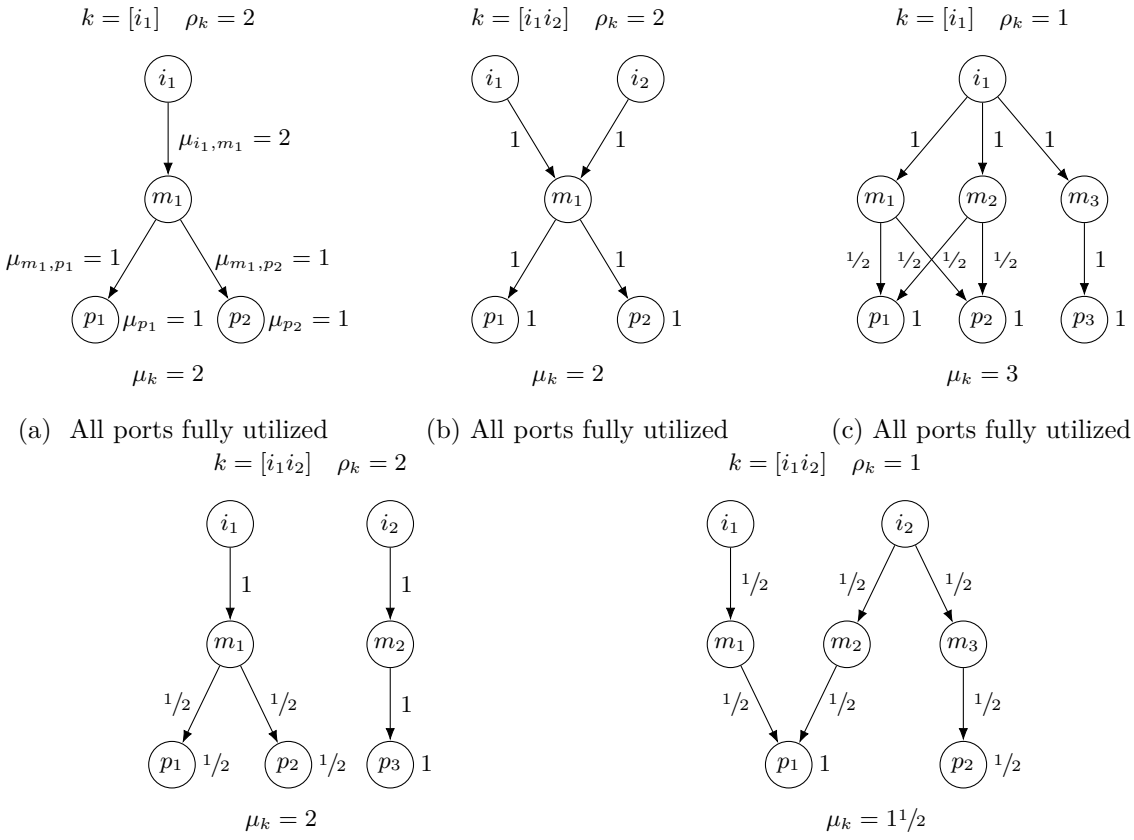
Figure 3.4 shows some instruction flow instances, each annotated with its solution according to these constraints.

Constraints $C_1 - C_{10}$ encode the basic flow properties, like "the sum of flow entering a vertex equals the sum of flow leaving it" and "flow can only propagate along edges". Remember that instruction vertices do not follow the standard rules for the conservation of flow. Constraint C_{13} models how flow entering these nodes is multiplied before leaving due to μop decomposition. By themselves, these constraints can easily be rewritten to a standard flow network for which a maximal solution corresponds exactly to the maximum flow solution. Such a solution would, however, allow nonsensical behaviour, such as two different instructions in a kernel steadily executing at different rates.

To forbid these solutions we add constraints C_{11} and C_{12}. C_{11} forces the IPC to be spread evenly among all instructions in a kernel, proportionally to the number of times they occur. Likewise, C_{12} forces all μops generated by an instruction to have the same MPC.

Constraints C_{15} and C_{16} limit the overall flow in the system by putting a bound on the rate at which μops can execute and retire. C_{15} encodes the maximum dispatch rate of execution ports of one unfused μop per cycle and C_{16} encodes the maximum rate at which fused μops can be retired. Note that C_{16} assumes all functional units of a CPU to be perfectly pipelined.

Finally, C_{17} limits the total MPC all μops for an instruction i can achieve by the total MPC of the kernel $[i]$ containing only that instruction. For most instructions this will not impose any additional throughput limitation not already encoded in C_{15} and C_{16}. It does, however, to some degree, allow handling instructions that are bottlenecked by CPU resources not currently present in the model. An example of this is the add-with-carry (`adc`) or subtract-with-borrow (`sbb`) instructions on x86 which have an implicit data dependency via the flags register limiting their IPC. As stated earlier, the instruction flow problem generally assumes that there are no



(a) All ports fully utilized

(b) All ports fully utilized

(c) All ports fully utilized

(d) Throughput of kernel is limited by p_3 and so p_1 and p_2 are not fully utilized.

(e) Throughput of kernel is limited by p_1 and so p_2 is not fully utilized.

Figure 3.4 – Example solutions to some simple instruction flow problems. For simplicity every instruction only appears once in every kernel and no μop fusion occurs. The names of weights are only shown in a, in all other figures we only give the values of weights.

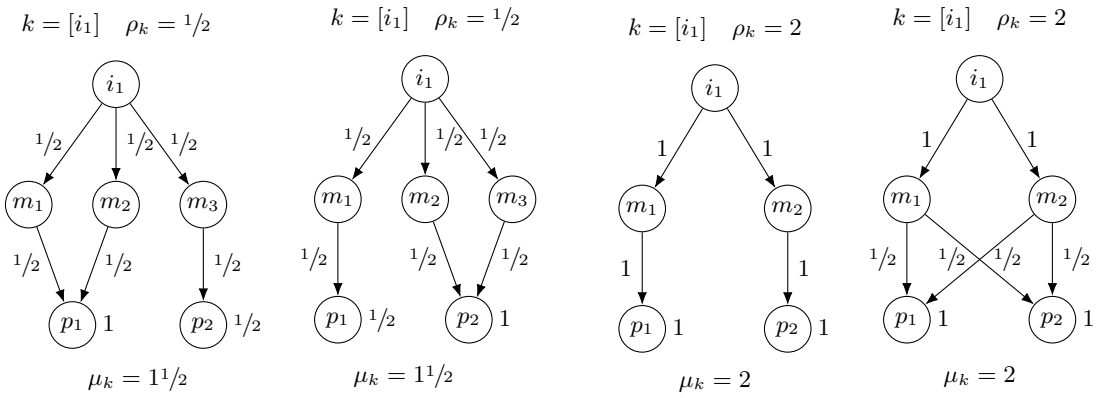
data dependencies. However, by adding C_{17} the IPC predicted for kernels containing these types of instructions is still accurate enough to find exact port mappings. Examples of this will be given in the experiments in Section 3.6.2.

Since in an instance of instruction flow G, o_i, $\#\mu_i$, $\#\lambda_i$, $\lambda_{\max}$, and $\mu_{[i]}$ are fixed all the above constraints are linear. They can thus be directly encoded as a linear optimization problem that computes the IPC and MPC of every kernel on the given port mapping. The objective function for this optimization problem is simply: maximizeμ_k That is, we maximize the weights representing the utilization of the execution ports in every kernel. Due to constraints C_4 to C_7 this "pressure" automatically propagates through the whole graph which determines the rest of the weights.

3.5.2 Finding port mappings

Now that we have introduced the instruction flow problem, we can show how it is used by PIPEDREAM to find a port mapping for a CPU architecture. The port mappings we produce correspond the I, M, and P vertices from instruction flow and the edges between them. That is, for every instruction we find which μops it decomposes into and for every μop we find the ports it can execute on.

The I level, that is, the instruction set of a CPU is usually well documented by vendors. For x86 we have used the ISA description from Intel XED. The P level, the execution ports,



(a) Two port mappings with identical IPC and MPC, but differing port utilization

(b) Two port mappings with identical IPC and MPC, and port utilization

Figure 3.5 – Examples of port mappings that produce the same IPC and MPC

are also relatively well documented. Most vendors at least disclose the number of execution ports an architecture has. However, which different μops a machine uses, what μops each instruction decomposes into, or even just the number of μops into which instruction decompose is undocumented.

It is the goal of PIPEDREAM to find this missing middle level of μops and the edges that connect all of the levels. The model produced by PIPEDREAM, however, does not necessarily perfectly reflect the real mapping and semantics of μops on the underlying hardware. We only need a model that accurately reflects the performance and resource sharing of instructions for use in GUS.

The first step in the reverse engineering process of PIPEDREAM is to determine for every instruction i the number of fused and unfused μops it decomposes into and the execution ports that it uses. In other words, we measure $\#\lambda_i$, $\#\mu_i$ and μ_p. This can be done directly using hardware performance counters and a single benchmark kernel $[i]$. To differentiate the empirically measured values for μ_p from those calculated by instruction flow we note them as $\mu_{\text{EMP},p}$. For instructions that decompose into a single μop, i.e., $\#\mu_i = 1$, this is already enough information to find its μop decomposition. Since such an instruction has only one μop all ports p where $\mu_{\text{EMP},p} \neq 0$ must be used by that μop.

For instructions that decompose into more than one μop the situation is not as simple though, since the sets of ports used by different μops can overlap. For these types of instructions we solve the instruction flow LP for each possible μop decomposition. We then compare this against the empirically measured total μop throughput, $\mu_{\text{EMP},k}$. The decomposition of an instruction is then assumed to be the one whose predicted throughput best matches the measured throughput, that is the mapping that minimizes the *simulation error*: $\frac{|\mu_k - \mu_{\text{EMP},k}|}{\mu_{\text{EMP},k}}$[2]. There are, however, two problems with this naïve algorithm.

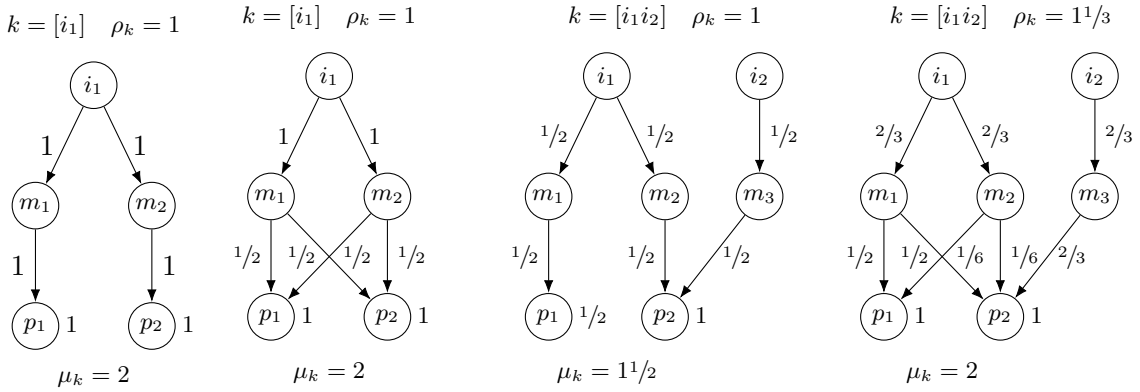
Problem 1: There are usually a large number of different port mappings that produce the same IPC, MPC and port usage. Figure 3.5 shows examples of such problematic cases.

Problem 2: Even though we do not have to enumerate all possible instruction flow instances for every instruction we still have to enumerate all possible μop decompositions. That is, the

[2]We could equally well define the simulation error using the predicted and measured IPC, ρ_k and $\rho_{\text{EMP},k}$, since the IPC and MPC are directly related.

k_{IPC}, I, M, and P nodes and the edges between k_{IPC}, I, and M are fixed, but we need to enumerate all possible combinations of edges between the M and P nodes.

Solution for problem 1



(a) The two port mappings from Figure 3.5b with identical IPC and MPC, and port utilization

(b) By adding i_2 to the kernel we can distinguish the two port mappings

Figure 3.6 – Example of how additional instructions can be used to distinguish port mappings

If we look at the two-port mappings shown in Figure 3.5a, we see that while they produce the same IPC and MPC the utilization of individual ports differs. We can extend the instruction flow LP to be able to distinguish these kinds of mappings by adding the following constraints:

C_{18} $-\forall p \in P.$ $\quad \mu_{\text{EMP},p} - \text{ERR} \leq \mu_p \leq \mu_{\text{EMP},p} + \text{ERR}$

(valid range for port utilization)

C_{19} $-\forall p \in P.$ $\quad 0 \leq \text{ERR}_{\mu,p} \leq \text{MAXERR}$

(upper bound for ERR, MAXERR is a constant)

Constraint C_{18} simply enforces that the usage of individual ports in an instruction flow solution does not differ too much from the actual measured usage. Here we do not fix the predicted μ_p to exact values, but only force them to be within an error margin ERR of the measured values. We do this since, in practice, performance counters are not perfectly accurate, and there is some noise in the measurements. The LP solver often cannot find any valid solution for any port mapping if we fix exact values for the port usage. For the LP we now use two optimizations objectives: First, minimizeERR, and second, as before, maximizeμ_k. That is, we first find the minimum value for ERR for which there is a solution, and we then find a solution that maximizes the throughput. Any solution for the second objective may not have a higher ERR than that found by the first objective. This type of lexicographic optimization problem with multiple objectives is directly supported by GUROBI, the LP solver we use. Finally, constraint C_{19} sets an upper bound for the maximum value of ERR. In practice, we set MAXERR to a small enough value so that the LP solver still finds a solution, for most instructions this is 0.1.

In practice, setting bounds on the port usage helps to speed up the LP solver. Usually, there are many mappings for which the solver can very quickly determine that there is no solution that can reproduce the observed throughputs. In our experience, the LP solver can filter out such infeasible mappings in under ten milliseconds while calculating a feasible solution can take several 100 milliseconds.

There are, however, port mappings that still can not be distinguished with this method. An example of this is shown in Figure 3.5b. We use the same general approach as Agner Fog [72] and UOPS.INFO [3] to find port mappings for these instructions. That is, to find the decomposition

for an instruction i, we generate multiple kernels that mix i with other instructions with a known decomposition and measure how they interact. An example of how this works can be seen in Figure 3.6. Algorithm 3.3 shows how PIPEDREAM uses this technique to find the μop decomposition is multi-μop instructions. Using single-μop instructions, whose decomposition we can easily find, we generate benchmarks that individually block all ports used by i.

Ideally, the single-μop instruction in every kernel would use exactly one port, however not every architecture has such an instruction for every port. On x86 processors, for example, there are two identical ports to handle the data transfer for memory reads. Any load μop can execute on either one of these two ports. Consequently, we generate one benchmark kernel for every μop that can be generated by a single-μop instruction. The number of different such μops is relatively small since we do not consider the semantics of μops, but only the ports they use. On x86, for example, integer addition, subtraction, any bitwise logical operations, and compare instructions all produce the same μop, i.e. use the same ports, even though they perform a variety of different tasks. The final μop decomposition for an instruction is then the one that minimizes the average simulation error $\frac{1}{k}\sum\frac{|\mu_k-\mu_{\text{EMP},k}|}{\mu_{\text{EMP},k}}$ for all k kernels. If there are multiple decompositions with the same simulation error we choose the one with the least number of edges between M and P. Section 3.6.2 will further explain this reasoning.

Solution for problem 2

A naive implementation of our algorithm for finding μop decompositions has to enumerate all possible decompositions for every different instruction, which is equivalent to enumerating all bipartite graphs. For an instruction with m μops on a CPU with p ports there are approximately $\sum_{k=1}^{p}\binom{p}{k}(-1)^{p-k}(2^k-1)^m$ such graphs [87]. Let us illustrate how many different decompositions there are in practice with a small example. On any Intel Haswell, Broadwell, or Skylake x86 CPU has eight execution ports, and the median number of μops an instruction decomposes into is two. Given these points, there are 183700 different possible μop decompositions for an average instruction. These are, of course, far too many graphs to test in a reasonable time since

```
for i in MULTI_MUOP_INSTRUCTIONS():
  measurements = MEASURE_KERNEL_PERFORMANCE([i])

  for muop in MUOPS_GENERATED_BY_SINGLE_MUOP_INSTRUCTION():
    if PORTS_USED(i) != PORTS_USED(muop):
      continue
    simple_inst  = INSTRUCTION_GENERATING_MUOP(muop)
    measurements += MEASURE_KERNEL_PERFORMANCE([simple_inst])
    measurements += MEASURE_KERNEL_PERFORMANCE([i simple_inst])

  best_muop_decomposition = None
  smallest_error          = ∞

  for muop_decomposition in POSSIBLE_MUOP_DECOMPOSITIONS():
    simulation_error = SOLVE_INSTFLOW_LP(muop_decomposition, measurements)

    if simulation_error < smallest_error:
      best_muop_decomposition = muop_decomposition
      smallest_error          = simulation_error
```

Algorithm 3.3 – Algorithm to find μop decomposition with per execution port hardware counters

we have to solve an LP problem for every such graph. To reduce the number of different μop decompositions we have to evaluate for an instruction i we use two techniques:

1. We only consider decompositions using the same ports as i. As stated above, all recent Intel CPU architectures have eight execution ports. However, the average instruction only uses four of these ports. Therefore, for an average instruction, we only need to consider around 80 different μop decompositions. Nevertheless, the number of possible decompositions still grows very rapidly, and, in practice, this approach can only handle instructions with up to three μops using up to four ports.
2. We can optionally also assume that all μops of an architecture can be produced by a single-μop instruction [3]. This way we no longer have to enumerate all possible bipartite graphs, but only all words over the alphabet of μops. For an instruction with m μops on a CPU with N different μops there are then m^N possible decompositions. On Skylake, for example, there are roughly 15 different μops. At first glance, this does not seem like an improvement over the previous technique, since we now have to explore $2^{15} \approx 32$thousand different decompositions for the average instruction. However, we do not always have to consider all 15 μops for every instruction. We only need to try μops that use a subset of the overall set of ports used by an instruction.
   In practice, this very significantly reduces the number of possible decompositions for an instruction. On Intel architectures the common case of an instruction that decomposes into a computation μop and a memory load μop there is only ever one possible mapping since the ports responsible for load/store and computations are distinct. Even instructions that decompose into four or five μops usually have less than one hundred possible decompositions.
   However, this assumption that all μops of an architecture can be generated using only single-μop instructions actually does not hold on x86. The one exception to this are memory store operations, which always decompose into at least two μops. However, both of these μops use a dedicated execution port, port 4 and port 7 respectively, which means we can easily detect instructions which use "impossible" μops PIPEDREAM does not know about yet. In these cases, PIPEDREAM falls back to the first technique to find these new μops. Once it has added the new μop to its list, other store instructions can be handled much faster.

3.5.3 Converting port mappings to resource models

Once PIPEDREAM has constructed a port mapping, we can convert it to a resource mapping as used by GUS as described in Section 3.4.3. In this model, we create the following abstract resources:

- Execution ports – We create one resource per execution port of the CPU. Since a port can accept one instruction per cycle, the throughput of each of these resources is one. To model μops that can execute on multiple execution ports, we also create virtual resources representing the combinations ports. By default, the throughput of combined resources is equal to the sum of the throughputs of all sub-resources from it is composed.
- μop retirement – We create one global resource to represent fused μop retirement limit whose throughput is equal to the measured $\lambda_{\max}$. Every instruction uses this resource proportionally to the number of fused μops it produces.
- Instruction throttles – Some instructions have a throughput that is limited by other CPU resources than functional units or the ROB. This behaviour is detected by PIPEDREAM by observing that the peak throughput of an instruction is significantly lower than its port mapping allows. For every such instruction i we create an abstract resource with the same throughput as the kernel $[i]$.

```
 1. spill registers to stack
 2. flush CPU pipeline
 3. start PAPI performance counters
 4. spill PAPI handles to stack
 5. flush CPU pipeline
 6. initialize values in registers (optional)
 7. benchmark-loop
 8. flush CPU pipeline
 9. restore PAPI handles from stack
10. read PAPI performance counters
11. flush CPU pipeline
12. restore registers
```

Figure 3.7 – General design of benchmarks generated by PIPEDREAM

3.5.4 Benchmark design

To take the required measurements for building GUS's CPU model PIPEDREAM uses automatically generated microbenchmarks written X86 assembler. All benchmarks are generated from a ISA description extracted from Intel XED [55]. Every benchmark is one procedure written in textual assembler, which is then assembled using GNU GAS. Multiple benchmarks are packaged into one shared library and then loaded into the PIPEDREAM process for execution.

PIPEDREAM benchmarks run in user-space and do not require root privileges on a machine. As a consequence we can not directly read hardware performance counters with the `rdpmc` instructions since it is by default only accessible from ring 0, i.e. kernel space. Instead, we have to use a system call to read counters. For convenience and portability, we use PAPI [65] for this. Using a system call to read performance counters has some wide-reaching consequences we have to account for in our benchmark design.

In general, the microbenchmark kernels we are interested in are tiny, consisting only of a handful of instructions, usually between one and five. However, the overhead of reading performance counters is much too high to such tiny kernels with any reasonable degree of accuracy. As a consequence, we have to repeatedly execute a kernel many times. Nevertheless, simply unrolling the kernel thousands of times may introduce skew the results of the measurements by putting stress on the instruction decoder or instruction cache. To avoid this skew, we instead wrap the kernel in a simple, statically counted loop. Since the instructions used to implement the loop also use some CPU resources which distorts the measurements we do unroll the kernel several times. The unrolling factor is chosen so that the loop still fits in the μop cache. Since modern CPUs have large μop caches with upwards of a thousand entries, the benchmark loop can safely be unrolled several dozen of times.

Figure 3.7 shows the general design of all benchmarks generated by PIPEDREAM. The CPU pipeline flushes in steps 2. and 5., implemented using a serializing instruction, prevent the CPU from reordering instructions from before the kernel to execute after performance counters have been started. Likewise, the flushes in steps 8. and 11. prevent instructions from inside the kernel to be executed after the counters have been read. Inside the benchmark kernel, we reserve one general-purpose register for the loop counter and, if the benchmark contains load/store instructions, another register as a base register for memory accesses. All other registers are available to the benchmark kernel. Register allocation in benchmarks is done after unrolling the kernel to prevent any unwanted false dependencies. We also allocate a large contiguous block

```
.kernel.loop.head:
  # vaddps     SRC1,  SRC2,  DST
  vaddps       %ymm1, %ymm1, %ymm0
  vaddps       %ymm1, %ymm1, %ymm2
  vaddps       %ymm1, %ymm1, %ymm3
  vaddps       %ymm1, %ymm1, %ymm4
  ...
  # Decrement loop counter & iterate
  # On Intel & AMD processors this is macro-fused into a single μop
  sub          $1, %r15
  jne          .kernel.loop.head
```

Figure 3.8 – Example benchmark to measure instruction throughput

```
.kernel.loop.head:
  # vaddps     SRC1,  SRC2,  DST
  vaddps       %ymm0, %ymm0, %ymm0
  vaddps       %ymm0, %ymm0, %ymm0
  vaddps       %ymm0, %ymm0, %ymm0
  vaddps       %ymm0, %ymm0, %ymm0
  ...
  # Decrement loop counter & iterate
  # On Intel & AMD processors this is macro-fused into a single μop
  sub          $1, %r15
  jne          .kernel.loop.head
```

Figure 3.9 – Example benchmark to measure instruction latency.

of memory called the *memory arena*. All load and store instructions are assigned a constant address inside this arena.

PIPEDREAM can also initialize any registers to a fixed value before the kernel executes, as shown in step 6.. Usually, all registers are initialized to zero, but some instructions required registers to be initialized to some special value. Integer division, for example, raises an exception if the divisor is zero, so we appropriately initialize registers to avoid this.

PIPEDREAM can not yet generate benchmarks for instructions that use a `rep` prefix, a `lock` prefix, or AVX512 instructions. In total, it can generate benchmarks for just over 2800 different instructions.

As cycle counter we used `CPU_CLK_UNHALTED`. To count the total number of μops executed in a benchmark we used the counters `UOPS_RETIRED:ALL`, for unfused μops, and `UOPS_RETIRED-:RETIRE_SLOTS` for fused μops. To count the number of μops executed on each execution port we used `UOPS_EXECUTED_PORT:PORT_0`/1/.../7. Finally, we used the average value over all runs for each counter.

Measuring instruction throughput and port usage

To measure the throughput of an instruction, i.e., how many instances of that instruction a processor can execute per cycle, we again generate a benchmark kernel containing only copies of that instruction. We then allocate the registers in the benchmark to create as many independent dependency chains as possible. To the same end, all load instructions are set to read from the

```
.kernel.loop.head:
  # cmp         SRC1,  SRC2
  # adcx        SRC1,  SRC2/DST
  cmp           %rax, %rax
  adcx          %rax, %rax
  cmp           %rax, %rax
  adcx          %rax, %rax
  ...
  # Decrement loop counter & iterate
  # On Intel & AMD processors this is macro-fused into a single μop
  sub           $1, %r15
  jne           .kernel.loop.head
```

Figure 3.10 – Example benchmark to measure instruction latency using an instruction with known latency. `cmp` reads two general purpose registers and writes to several bits of the flags register, by itself it cannot create a sequential dependency chain. `adcx` reads two general purpose registers and the carry bit of the flags register and writes one general purpose register as well as the carry bit. Together these instructions create a dependency chain via the carry bit.

same location in the memory arena, while store instructions all get their own memory location to write to. An example of such a throughput benchmark can be seen in Figure 3.8 Note that many x86 instructions have a fixed operand they both read from and write to; usually the `%rax` register or the flags register. For these instructions, we can at most create one dependency chain, so they always execute sequentially, with a throughput of at most one. Some other instructions, like division or square root, have a very long latency or are not fully pipelined and also have a throughput of less than one.

The measure the IPC of a benchmark we divide the number of instructions executed, which we know statically, by the number of cycles it takes to execute the kernel. For the fused and unfused MPC we divide the values of the corresponding counters for the number of executed μop by the number of cycles. To account for the loop around the benchmark kernel, we subtract the number of instructions or μops for such an empty loop from the total counts for each real benchmark. The instructions for the loop are assumed to execute in parallel with all other instructions, so we do not adjust the number of cycles.

The benchmarks used to measure usage of execution ports are structured the same as those for throughput measurements. To determine the usage of each execution port, we measure the number of unfused μops executed on it and divide that by the number of cycles as above.

Measuring instruction latency

For most instructions measuring latency, i.e., how many cycles an instruction requires to execute, is no more complicated than measuring the throughput. The only difference being that here we allocate registers and memory locations in a way to create exactly one dependency chain which forces the kernel to execute completely sequentially. An example of such a latency benchmark can be seen in Figure 3.9. The latency of an instruction is then defined as the number of instructions executed by the benchmark divided by the number of cycles it takes to execute it. We subtract the number of instructions and μops for the benchmark loop as is done for throughput benchmarks.

This approach is not sufficient for measuring the latency of all instructions since not all instructions can create a dependency chain with themselves. Instructions that read from memory and write to a register, like some move instructions, for instance, can not be benchmarked directly

with this approach. To measure the latency of these more complicated instructions, we add a second instruction with an already known latency to the benchmark kernel so in order to create a sequential dependency chain. The latency of the first instruction is then determined as the latency of the whole kernel minus the latency of the known instruction. For a small number of instruction types, less than 70, we even require two instructions with a known latency to create a dependency chain. Figure 3.10 shows a benchmark kernel containing two instructions to measure latency. Finally, there are some exotic instructions for controlling processor state that do not directly write to any registers or memory. Though Pipedream can create throughput benchmarks for these instructions, it current has no way to reliably benchmark their latency.

Measuring micro-fusion

The benchmarks used to measure μop micro-fusion are simpler than those for throughput and latency since they do not depend on timings. Here, we do not need to worry about decoding, and instruction or μop cache misses or data dependencies. So a benchmark can simply be a long repeated sequence of the same instruction for which we measure the number of fused and unfused μops executed.

3.6 Experiments

Both Gus and Pipedream are still early prototypes under active development. This section highlights preliminary results of their usefulness along several case studies.

3.6.1 Gus

3.6.1.1 Case study I: Matrix multiplication

The first case study shows how we optimize a matrix multiplication kernel based on feedback from Gus. Here we present a single-threaded version of the basic matrix-matrix product $C := AB$. All versions of the code are compiled with GCC 9.1 with the flags `-O2`. [3] All measurements are taken on a Skylake 7940X running at 2.3 GHz using PAPI [65]. All reported measurements are the median over ten executions.

We use square 192×192 matrices of 32 bit floating-point numbers. So each matrix uses 144 kB of memory. Skylake CPUs have a 32 kB L1 cache and a 1 MB L2 cache. Hence we cannot fit one whole matrix in L1, but all three matrices can fit in L2.

The resource mappings used for the simulator have been produced with Pipedream. For instructions that Pipedream cannot benchmark yet, such as conditional branches, we use the mappings of Agner Fog [72]. The data-transfer bandwidth between individual cache levels and memory have been determined using simple microbenchmarks.

For the sensitivity analysis we first run one simulation of a kernel with the default throughput for all resources and default instruction latency. We then run one 15 simulations per resource, each time incrementing the throughput of that resource by 1%. We configure Gus to only simulate the performance of the computational kernel. The rest of the benchmark program, that is the code to initialize data structures and print results, are only simulated functionally.

Step I: Naive matrix multiplication

We start from the basic naive implementation of a matrix multiplication, seen below. `N` is set to 192.

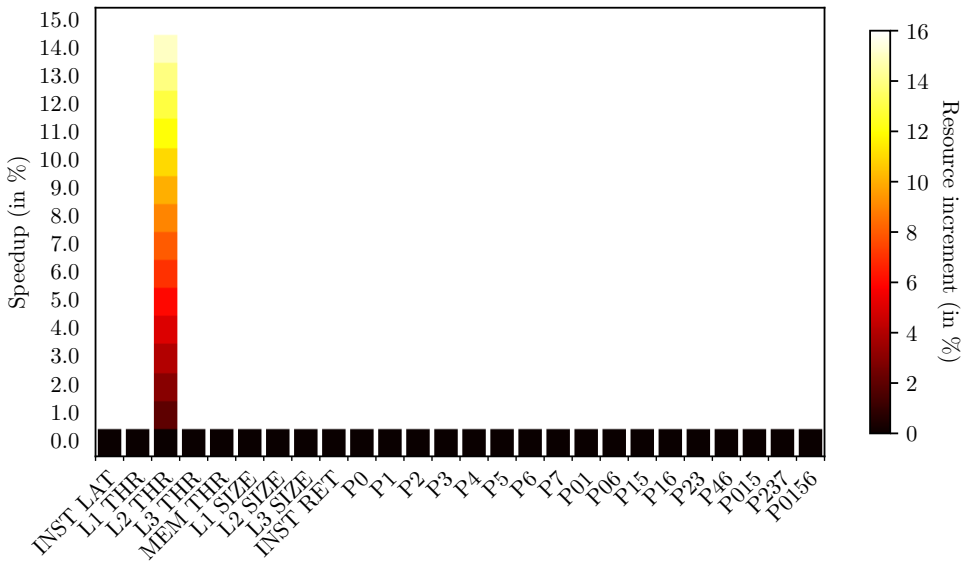
[3] All benchmarks here are limited to 128 bit SSE instructions, since QEMU does not support newer vector instructions yet.

```
for (int i = 0; i < N; i++) {
  for (int j = 0; j < N; j++) {
    for (int k = 0; k < N; k++) {
      C[i * N + j] += A[i * N + k] * B[k * N + j];
    }}}
```

The naive matrix multiplication kernel has a runtime of approximately 25 million cycles on the test machine. It takes 3.5 cycles to perform one accumulation into `C`, that is, to execute one iteration of the innermost loop. This corresponds to 0.5 32 bit floating-point operations per cycle (FLOPs/cycle).

There are likely two performance bottlenecks in this code. The first is the fact that the matrix `B` is traversed with a stride of `N` along the innermost `k` loop. That is, while it is stored in row-major order, the algorithm traverses it in column-major order. This is likely to cause a large number of cache misses depending on the value of `N`. The second possible problem is that the innermost loop forms a reduction in `C[i * N + j]`. This reduction prevents any instruction-level parallelism and forces the CPU to execute all iterations of the innermost sequentially.

Even for such a small example, it is not necessarily trivial to analytically detect which of the two problems forms the bigger bottleneck. Instead of manually analysing the kernel, we use Gus's sensitivity analysis. It finds that the performance is limited most by the bandwidth between the L1 and L2 cache. This is due to a large number of misses in the L1 cache where each miss causes data movement between L1 and L2. A visual representation of the results of the sensitivity analysis produced by our tool is shown below. We use a form of heat-map for the visualization. Each bar in it represents one abstract resource. The height of each bar and its colour indicate the speedup predicted by Gus if the throughput of that resource is increased. Take, for example, the resource L2 THR, which represents the throughput between L1 and L2. The graph shows that an increment of its throughput by 1% decreases the overall runtime of the kernel by 1%. If we increment the resource's throughput by 2%, the runtime decreases by approximately 2%. The bar fades to white around the $13-14\%$ mark, which indicates that a 15% increase in throughput produces a $13-14\%$ speedup. All other resources immediately face to white at the 0% mark. That is, according to Gus, the runtime of the kernel is not affected by increasing the throughput of any other resource. Hence, L2 THR is the bottleneck of this version of the kernel.



To improve the performance of the matrix multiplication kernel we need to apply a transformation that reduces its usage of the bandwidth between L1 and L2. In general, we consider three different optimizations for such cases:

1. Loop interchange – Changing the order of loops in a loop nest can improve cache usage by changing the strides of memory accesses.
2. Loop tiling – Tiling splits the iteration space of a loop nest into smaller tiles to ensure that data stays in cache until it is reused. Loop tiling is more difficult to apply since it introduces new loops into the kernel, which increases the number of control instructions, such as branches and comparisons. Have more loops with a smaller trip count may also negatively affect branch prediction. It is also non-trivial to find optimal tile sizes to ensure that data stays resident in the cache.
3. Data layout and alignment changes – Changing the alignment or layout of data can reduce conflict cache misses. In this case, we could pad the matrices with additional unused elements to ensure that the ways of the cache are fully utilized. However, changing the alignment and layout of data is a non-local change that requires code outside the kernel for allocating and inspecting the matrices to be modified too.

Here, we chose to perform a loop interchange on the kernel since it is the simplest of the three transformations.

Step II: Loop interchange

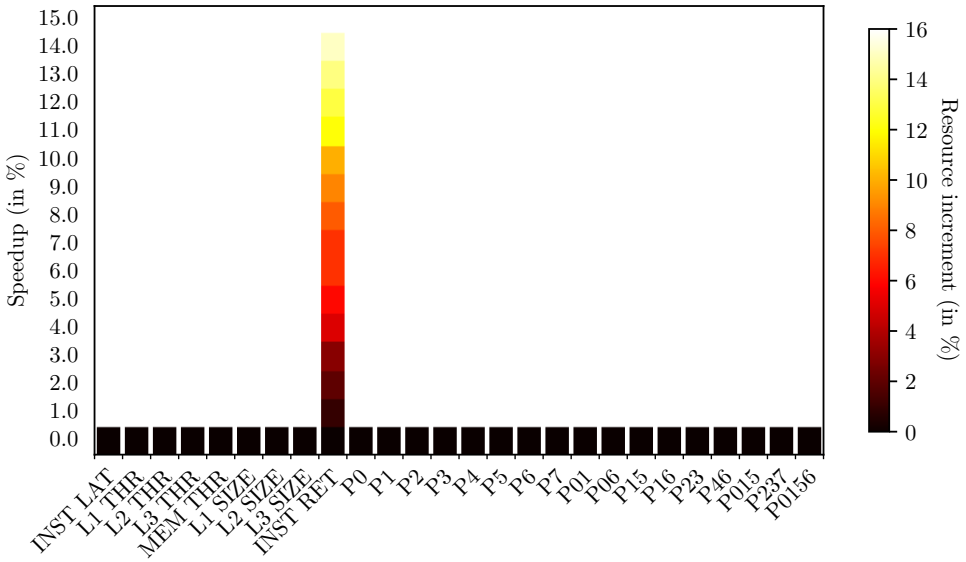
We perform a loop interchange that changes the order of loops from `ijk` to `kij`.

```
for (int k = 0; k < N; k++) {
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      C[i * N + j] += A[i * N + k] * B[k * N + j];
    }}}
```

The loop interchange improves the runtime of the kernel from 25 million to 15 million cycles. It now takes 2.1 cycles to execute one iteration of the innermost loop, and the FLOPs/cycle rise from 0.5 to 0.9. More detailed statistics are given in Table 3.1. There are two reasons why the interchange increases the performance of the kernel:

- Changing the order of loops also changes the strides of the memory accesses to the different matrices. It notably causes the kernel to traverse the matrix `B` in row-major instead of column-major order.
- The innermost loop of the kernel now no longer forms a reduction in `C[i * N + j]`, which exposes a large amount of parallelism since all iterations of the innermost can now execute in parallel.

Our sensitivity analysis now reports that the bottleneck of the modified kernel is now no longer the bandwidth between L1 and L2 but the rate at which the ROB can retire μops. In the visualization below, this is represented by the resource INST RET, which is used by every instruction. μop retirement is now a bottleneck due to the large amount of instruction level parallelism exposed by the interchange.



To further improve the performance of the matrix multiply kernel, we need to reduce the number of instructions that the CPU needs to retire per entry of the matrices. For kernels that are bounded by the ROB we consider three different optimizations:

1. Vectorization – Most vector instructions use the same number of slots in the ROB as their scalar counterparts. By replacing scalar instructions with vector instructions we can thus effectively increase the number of computations performed per μop.
2. Loop unrolling – Unrolling loops reduces the number of control instructions and also often exposes other redundant computations that can be removed. It is also often required to enable vectorization.
3. Redundancy elimination – This general involves manually removing redundant computations, such as factoring out terms in strided accesses, not already detected by the compiler.

In the current version of the code the compiler has already used the sophisticated addressing modes available on x86 to fold address computation, loads, stores, and arithmetic operations into as few instructions as possible. There is no redundancy left remove. The kernel is, however, a perfect candidate for unrolling and consecutive vectorization.

	Simulator			**Real execution**		
Kernel	**Cycles**	L1 **misses**	L2 **misses**	**Cycles**	L1 **misses**	L2 **misses**
Naive	25,688,364	7,224,804	6914	25,390,280	7,281,762	6076
Loop interchange	12,451,163	481,539	6914	15,165,666	474,428	5682

Table 3.1 – Effects of a loop interchange on the matrix multiplication kernel

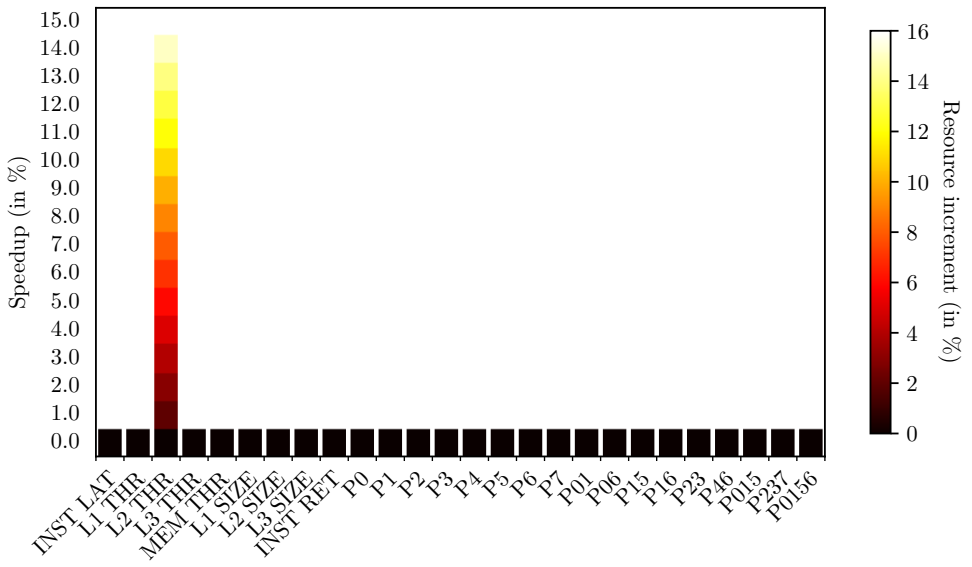
Step III: Loop unrolling and vectorization

To reduce the μop retirement bottleneck in the interchanged kernel unroll the innermost `j` loop and the vectorize the resulting code. We use 128-bit SSE vectors for this, which process four 32 bit floating-point numbers at once. Consequently, we reduce the number of instructions that need to be retired per iteration of the innermost loop by approximately four. Another optimization would to use a single fused multiply-add instruction (`fma`) instead of a seperate multiply and add. However, QEMU currently does not support `fma` instructions on x86.

```
for (int k = 0; k < N; k++) {
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j += 4) {
      // set all elements of vector a to A[ i * N + k]
      __m128 a = _mm_set_ps1(A[ i * N + k]);
      // vector load from C
      __m128 c = _mm_load_ps(&C[i * N + j]);
      // vector load from B
      __m128 b = _mm_load_ps(&B[k * N + j]);
      // c = (a * b) + c
      c = _mm_add_ps(_mm_mul_ps(a, b), c);
      // vector store to C
      _mm_store_ps(&C[i * N + j], c);
    }}}
```

This reduces the runtime of the kernel from 15 million cycles to 3.6 million cycles. It now takes 0.5 cycles instead of 2.1 to execute one accumulation into `C`. The FLOPs/cycle rise from 0.9 to 3.9.

The sensitivity analysis for this version of the kernel finds that the μop-retirement bottleneck has been removed. Instead, the bandwidth between L1 and L2 is again the main limiting factor for performance.



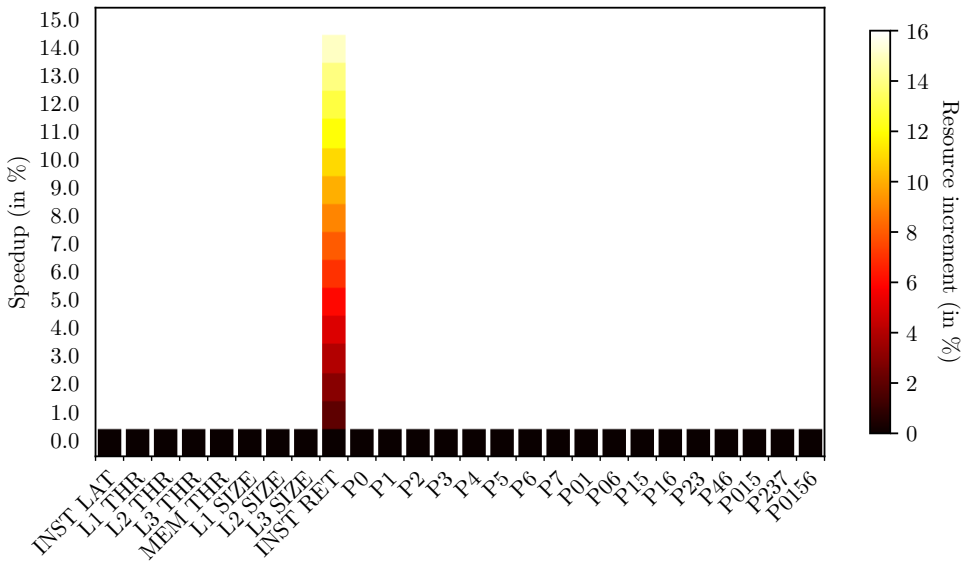
We consider the same three optimizations as before for reducing the number of cache misses and reducing the of the bandwidth between L1 and L2: loop interchange, loop tiling, and changes to the data alignment or layout. We have already performed a loop interchange, so this is no longer an option. As before, we prefer to not require users to change their code and opt-out of changing the data alignment or layout. The remaining option is loop tiling.

Step IV: Loop tiling

To improve reuse of data in caches, we tile the `i` and `j` loops with a tile size of 32. The tiled version no longer iterates over entire rows and columns of `C` directly. Instead, it operates on 32×32 sub-matrices that easily fit in the L1 cache. This optimization improves the runtime of the kernel from 3.6 million cycles to 3.2 million cycles. One accumulation into `C` now takes 0.45 cycles to execute, and the FLOPs/cycle rise from 3.9 to 4.3.

```
for (int I = 0; I < N; I += 32) {
  for (int J = 0; J < N; J += 32) {
    for (int k = 0; k < N; k++) {
      for (int i = I; i < I + 32; i++) {
        for (int j = J; j < J + 32; j += 4) {
          __m128 a = _mm_set_ps1(A[i * N + k]);
          __m128 c = _mm_load_ps(&C[i * N + j]);
          __m128 b = _mm_load_ps(&B[k * N + j]);
          c = _mm_add_ps(_mm_mul_ps(a, b), c);
          _mm_store_ps(&C[i * N + j], c);
        }}}}}
```

The sensitivity analysis now shows that tiling has effectively reduced the number of cache misses, and that the bottleneck of the kernel has moved back to the μop retirement rate.

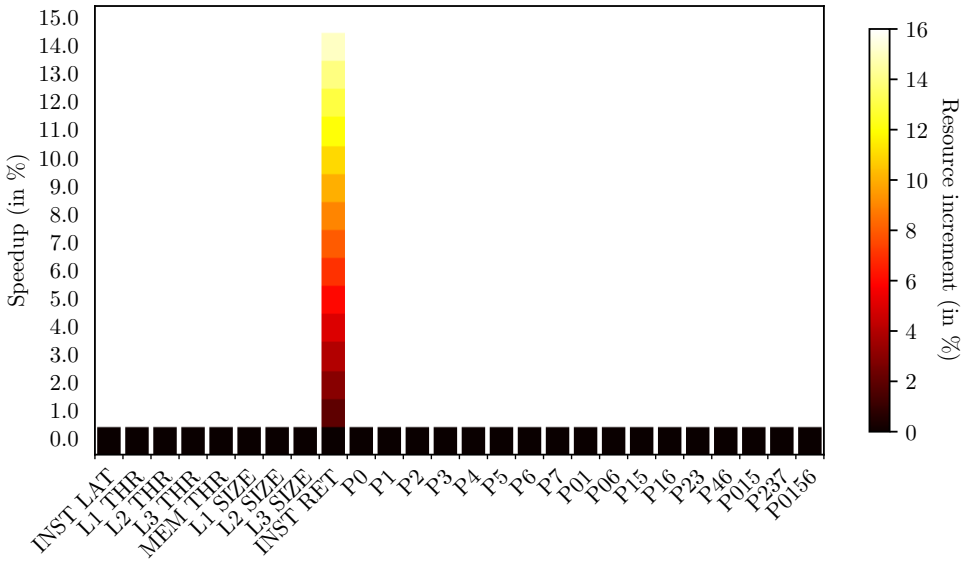


Same as at the end of Step II, we consider the following optimizations to remove the μop retirement bottleneck: vectorization, loop unrolling, and redundancy elimination. We have already vectorized the code and the compiler is able to eliminate any redundancy in the body of the inner loop. Consequently, the only remaining option is to unroll the inner loop even more.

Step V: Loop unrolling

To reduce the number of control instructions executed by the benchmark we unroll the inner `j` loop eight times. This last transformation reduces the runtime of the kernel from 3.2 million cycles to 2.1 million cycles. This version now takes 0.3 cycles to perform one accumulation into `C`, it achieves 6.6 FLOPs/cycle. The final version is more than ten times faster than the original, which required 25 million cycles to complete.

Note that even this last version is still bound by the retirement rate of μop instructions.



Conclusion

Figure 3.11 plots the floating-point operations per cycle predicted by Gus against those measured using hardware performance counters. The peak FLOPs/cycle achievable with SSE instructions is 8.

The mean absolute error of the execution times predicted by Gus across all five versions is 8%. The error in the runtime predicted by Gus is largest in the version after the loop interchange. There, Gus's predicted runtime is 17% faster than the actually measured runtime. As far as we can tell the error in the version after the loop interchange is due to our optimistic model of the ROB and macro- and micro-fusion, and μop unlamination. Pipedream is not yet able to detect μop unlamination, and so Gus does not model it. For the moment, we optimistically assume that both macro- and micro-fusion always occurs and that μops are never unlaminated.

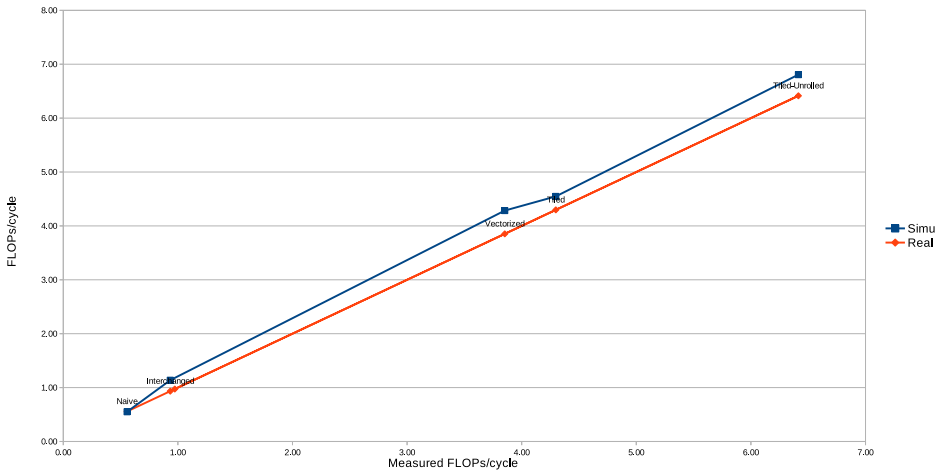


Figure 3.11 – **Case study I: Matrix Multiplication** Comparison of floating-point operations per cycle (FLOPs/cycle) as predicted by Gus and measured using hardware performance counters.

3.6.1.2 Case study II: Matrix block-copy

Our second case study illustrates how Gus detects a bottleneck in the execution port usage of a program caused by an optimization missed by the compiler. The code in question was taken

```
/// copy a 4x4 block from NxN matrix 'src' to 'dst'.
void copy_sse_4x4(float* restrict src, float* restrict dst,
                  unsigned i, unsigned j, unsigned N) {
  __m128 r0, r1, r2, r3;

  // vector load from src[i][j]
  r0 = _mm_load_ps(&src[(i + 0) * N + j]);
  r1 = _mm_load_ps(&src[(i + 1) * N + j]);
  r2 = _mm_load_ps(&src[(i + 2) * N + j]);
  r3 = _mm_load_ps(&src[(i + 3) * N + j]);

  // vector store to dst[j][i]
  _mm_store_ps(&dst[(j + 0) * N + i], r0);
  _mm_store_ps(&dst[(j + 1) * N + i], r1);
  _mm_store_ps(&dst[(j + 2) * N + i], r2);
  _mm_store_ps(&dst[(j + 3) * N + i], r3);
}

/// benchmark kernel used in this case study
unsigned N = ...; // matrix size
float *src = ...; // src NxN matrix
float *dst = ...; // dst matrix

for (unsigned repetitions = 0; repetitions < 64; repetitions++) {
  // copy matrix src to dst in 4x4 blocks
  for (unsigned i = 0; i < N; i += 4) {
    for (unsigned j = 0; j < N; j += 4) {
      copy_sse_4x4(mat, matT, i, j, N);
    }
  }
}
```

Figure 3.12 – Kernel for copying a 4×4 block from one matrix to another

from a library for high-performance matrix transpositions developed in our team. Fast transpositions of small, L1 resident matrices are an important building block for high-performance tiled matrix multiplication of large matrices. The library contains a variety of optimized kernels for different x86 architectures. The kernel used in this case study, `copy_sse_4x4`, shown in Figure 3.12, copies a 4×4 block of 32 bit floating-point numbers from one matrix to another. It targets copies of matrices that fit in L1. Here we use the version written with SSE intrinsics since QEMU does not support more recent vector instructions. `copy_sse_4x4` is used in the performance regression test suite of the library to establish a performance baseline for the actual block-wise transposition kernels. By comparing the throughput of a transposition against that of a raw block copy, the library detects if the transposition is bounded by data movement or vector shuffle instructions.

In this case study we call `copy_sse_4x4` in a loop to copy a 64×64 matrix block by block. We also repeat the entire matrix copy 64 times. Otherwise, the benchmark is too small, and we cannot reliably measure its runtime using hardware performance counters. Gus does not need performance counters and also works with smaller kernels. We only use hardware counters to calculate how accurate the runtime predictions of Gus are. As a compiler we use GCC 9.1 with the flags `-O3`. As in the first case study all time measurements have been taken on a Skylake

7940X running at 2.3 GHz.

The baseline of our benchmark takes 89k cycles to execute, which corresponds to 11.7 bytes copied per cycle. To test whether the loop control instructions in the kernel cause a significant overhead, we unroll the inner `j` loop of the benchmark four times. Surprisingly, this version is slower than the baseline. It takes 127k cycles to execute, at a rate of 8.2 bytes per cycle. This results indicates the existence of a performance bug in our benchmark.

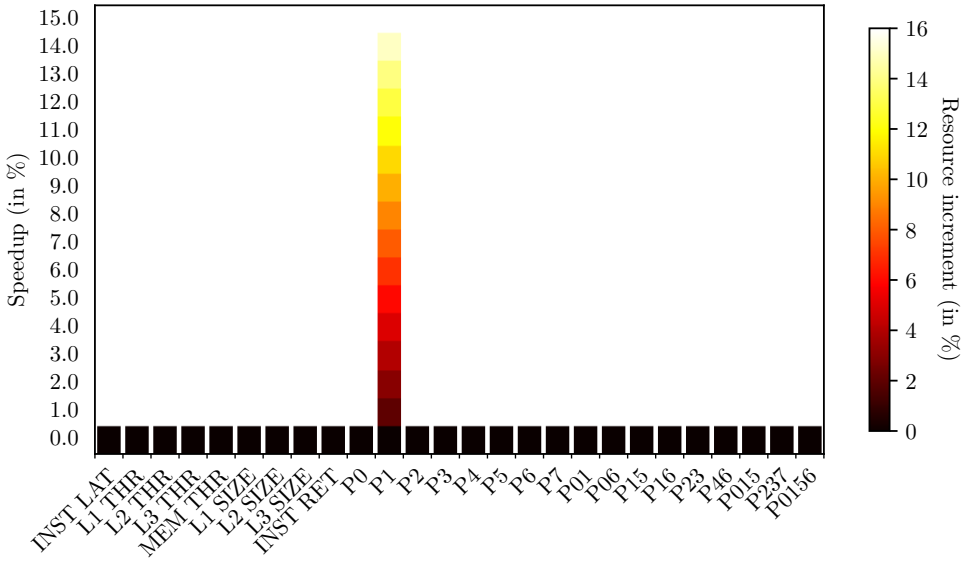


Figure 3.13 – Results of the sensitivity analysis for the benchmark from Figure 3.12. The bottleneck is execution port p_1, which handles general scalar and vector arithmetic and logic instructions.

We use Gus to identify this performance bug. The results of Gus's sensitivity analysis for the original version of the benchmark, shown in Figure 3.13, indicate that p_1 is the biggest bottleneck here. None of the other execution ports forms a relevant bottleneck. p_1 handles general scalar and vector arithmetic and logic instructions. It does neither execute load/store nor branch instructions. At the source level, the benchmark consists entirely of SSE load/store intrinsics and simple loops, so it is surprising that p_1 alone is a bottleneck.

The output Gus provides statistics, including the execution port usage, for all machine instructions in the analysed program. We investigate the assembly code of the benchmark to find the source of the performance bug. The only transformation that the compiler has performed is to inline `copy_sse_4x4`. Figure 3.14 shows the assembler for the body of the innermost loop of the benchmark. Besides the expected `movaps` load/store instructions the loop also contains a number of `lea` instruction; one `lea` per `movaps`. `lea`, *load effective address*, instructions are used to calculate memory addresses without performing a memory access. The `lea` instructions used here calculate 32 bit addresses and can only execute only on port p_1. All other instructions execute on other ports. The `lea` instructions are clearly the cause of the bottleneck.

If we look back at the source code of the benchmark in Figure 3.12 we see that it uses `unsigned` integers. On a 64-bit x86 platform an `unsigned` integer has 32 bits. Together the `lea` and `movaps` instructions calculate the accessed addresses `src[(i + 0) * N + j]`, `src[(i + 1) * N + j]`, and so on. All these addresses computations have a common term `src[i * N + j]`, which could be factored out and computed once. That is, `src[(i + 1) * N + j]` could be rewritten to `src[(i * N + j) + N]`. The author of the code expected the compiler to eliminate the redundant re-computation of this common term. However, the compiler was unable to do this. Most likely since it needed to preserve any possible integer over- or underflow that might occur during the address calculation.

Source	Assembler	Resources
	`copy_sse_4x4:`	
	`leal (%r12,%rax), %r10d`	P1
	`leal (%rax,%rbx), %r13d`	P1
`r0 = _mm_load_ps(&mat[(i+0)*N+j]);`	`movaps (%rdi,%r10,4), %xmm2`	P23
	`leal 0(%rbp,%rax), %r10d`	P1
`r1 = _mm_load_ps(&mat[(i+1)*N+j]);`	`movaps (%rdi,%r13,4), %xmm3`	P23
`r2 = _mm_load_ps(&mat[(i+2)*N+j]);`	`movaps (%rdi,%r10,4), %xmm1`	P23
	`leal (%r11,%rax), %r10d`	P1
	`addl $4, %eax`	P0156
`r3 = _mm_load_ps(&mat[(i+3)*N+j]);`	`movaps (%rdi,%r10,4), %xmm0`	P23
	`movl %ecx, %r10d`	
`_mm_store_ps(&matT[(j+0)*N+i], r0);`	`movaps %xmm3, (%rsi,%r10,4)`	P23 P4
	`leal (%rdx,%rcx), %r10d`	P1
	`addl %r9d, %ecx`	P0156
`_mm_store_ps(&matT[(j+1)*N+i], r1);`	`movaps %xmm2, (%rsi,%r10,4)`	P23 P4
	`movl %r8d, %r10d`	
`_mm_store_ps(&matT[(j+2)*N+i], r2);`	`movaps %xmm1, (%rsi,%r10,4)`	P23 P4
	`leal (%rdx,%r8), %r10d`	P1
	`addl %r9d, %r8d`	P0156
`_mm_store_ps(&matT[(j+3)*N+i], r3);`	`movaps %xmm0, (%rsi,%r10,4)`	P23 P4
	`cmpl %eax, %edx`	
	`ja copy_sse_4x4`	P6

Figure 3.14 – Assembler generated for the code in Figure 3.12. The code uses AT&T assembler syntax. Note the explicit address calculation using `lea` and `add`. The code was generated by GCC version 9.1 with flags `-O3`. Clang (8.0.1) produces comparable code. Register to register copies, `movl`, are assumed to be move eliminated so they use no ports. The `cmpl` and `ja` at the end of the loop are assumed to be macro-fused.

Unlike the compiler, we know that no over- or underflow can occur during these address computations. To "communicate" this to the compiler, we change all `unsigned` integers to signed `int`. `int` is also 32-bits wide here, but, according to the C standard, the compiler is allowed to assume that no arithmetic involving signed integers under- or overflows. Figure 3.16 shows the assembly code for this modified benchmark. As we can see, there are no more `lea` instructions in the inner loop body. We could also have manually hoisted the redundant address calculation. In both cases, GCC generates the same assembler code. The sensitivity analysis for this optimized version of the benchmark shows that p_4, the store-data port, is now the biggest bottleneck. This is precisely the behaviour we expect from a memory-copy kernel. The results of the sensitivity are shown in Figure 3.15. The optimized version of the benchmark takes 81k to execute, at a rate of 12.9 bytes per cycle. This constitutes a speedup of 11% over the baseline version.

3.6.2 Pipedream

To illustrate Pipedream's functionality we compare and discuss the port mappings produced by Pipedream, uops.info, and Agner Fog for several instructions. Since uops.info is currently not publicly available we use the port mappings, latencies and throughputs published online [2]. Abel et al. do not only publish their final results but also the logs of the experiments they ran to obtain them. Agner Fog's port mappings are taken from his personal website [72].

All measurements where performed on an Intel® i7-6600U Skylake CPU. All benchmark kernels where unrolled so the kernel contains 200 instructions and the kernel loop was set to iterate

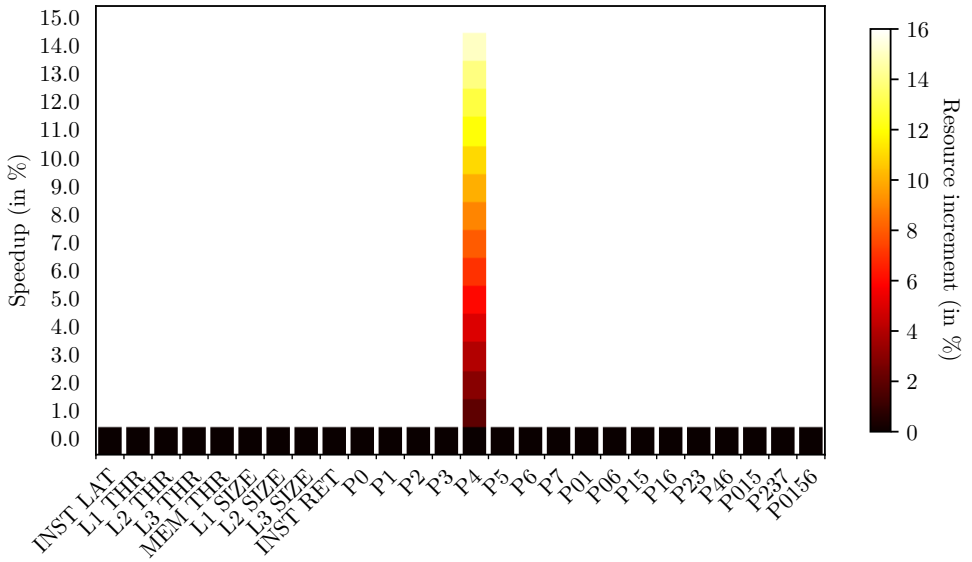


Figure 3.15 – Results of sensitivity analysis for optimized version of kernel from Figure 3.12 using `signed int` instead of `unsigned int`. The bottleneck is execution port p_4, the store-data port

Source	Assembler	Resources
	`copy_sse_4x4:`	
`r0 = _mm_load_ps(&mat[(i+0)*N+j]);`	`movaps (%rax,%r9,4), %xmm2`	P23
`r1 = _mm_load_ps(&mat[(i+1)*N+j]);`	`movaps (%rax,%r8,4), %xmm1`	P23
`r2 = _mm_load_ps(&mat[(i+2)*N+j]);`	`movaps (%rax,%rdi,4), %xmm0`	P23
`r3 = _mm_load_ps(&mat[(i+3)*N+j]);`	`movaps (%rax), %xmm3`	P23
	`addq $16, %rax`	P0156
`_mm_store_ps(&matT[(j+0)*N+i], r0);`	`movaps %xmm3, (%rcx)`	P23 P4
`_mm_store_ps(&matT[(j+1)*N+i], r1);`	`movaps %xmm2, (%rcx,%rdx,4)`	P23 P4
`_mm_store_ps(&matT[(j+2)*N+i], r2);`	`movaps %xmm1, (%rcx,%rdx,8)`	P23 P4
`_mm_store_ps(&matT[(j+3)*N+i], r3);`	`movaps %xmm0, (%rcx,%r11)`	P23 P4
	`addq %rsi, %rcx`	P0156
	`cmpq %rax, %r10`	
	`jne copy_sse_4x4`	P6

Figure 3.16 – Optimized version of kernel from Figure 3.12 using `signed int` instead of `unsigned int`. This version does not have any redundant address calculation. The code was generated by GCC version 9.1 with flags `-O3`. Clang (8.0.1) produces comparable code

ten thousand times. That is, every benchmark kernel executes 2 million instructions. Every benchmark was furthermore executed 750 times, where the first 250 measurements are considered warm-up iterations to force the CPU into its highest frequency. Only the measurements of the remaining 500 executions where used. To filter out noisy benchmark runs, such as those where Pipedream suffered a context-switch, we furthermore discard the 5% slowest runs.

The instruction latencies and throughputs Pipedream finds are the same as found by uops.info and Agner Fog. So the following case studies will focus on the construction of port mappings.

The x86 instruction set is highly complex and irregular. One big source of complexity is that in x86 the encoding of an isntruction's opcode and of its operands is largely orthogonal. This means that most instructions can be used with a large number of different operand combinations.

So, most instructions exist in a number of variants that differ in the number of operands, in the bit-width of operands, and if operands are stored in registers or memory. There are more than thirty variants of the scalar integer addition instruction `add` alone. Most of the time these different variants have the same performance and μop decomposition, but this is not always the case. 32 bit and 64 bit addition of register values, for example, both produce the same μop and have the same throughput and latency. But addition where one operand is in memory has a different decomposition and performance. Another example is integer multiplication where variants with different bit-widths have a different μop decomposition, even if all operands are in registers.

The current version of PIPEDREAM cannot benchmark all X86 instructions. First off, it only supports 64 bit mode unprivileged, i.e., user space instruction. Instructions with a `rep` or `lock` prefix, instructions that use the stack, or AVX512 instructions are not supported. Furthermore, only unconditional branches with an immediate operand are supported. We have only tested the port mapping algorithm for instructions that decompose into up to five unfused μops. It also currently cannot handle instructions with a very low peak IPC like integer division. This still covers a large subset of the X86 instruction set that includes all instructions encountered during the GUS case studies.

In the following we will list instructions using the same pseudo Intel assembler syntax used by Agner Fog. Here the first operand is always the destination. Operands in a general purpose register will be written as `r`, memory operands as `m`, and `xmm` vector registers operands as `x`. Operands are suffixed with their bit-width if it is relevant. So for example, `add r, r` denotes a general register to register addition, `add r, m` adds a value loaded from memory to a register, and `imul r32, r32` denotes a 32 bit multiplication. Some instructions operate on a fixed register, in these cases we directly write the name of the register. `shr r, cl`, for example, denotes a shift where the first operand is in a general purpose register and the second operand is the `cl` register.

3.6.2.1 Case study I: Arithmetic and logic instructions

This case study shows how PIPEDREAM detects decompositions for arithmetic and logic instructions that only operate on registers. We only present a small illustrative subset of all arithmetic and logic instructions that includes scalar and vector integer and floating point arithmetic, bitwise operations, shifts, comparisons, and conditional moves. This case study only considers instructions with register or immediate operands, load/store instructions are presented later.

On Skylake[4] arithmetic and logical instructions use up to four ports, 0, 1, 5, and 6. Most of these instructions produce only one μop, only a small number of more complex instructions decompose into multiple μops. Table 3.3 shows the decompositions we found for a number of different instructions. For all these instructions, and many more, PIPEDREAM, Fog Agner, and UOPS.INFO find exactly the same results.

For now we have only explored instruction that decompose into up to six μops, since this covers all instructions commonly emitted by compilers. Table 3.2 gives some examples for the number of possible μop decompositions PIPEDREAM needs to explore. Up until 4 μops we can still easily explore all possible mappings, for 5 or 6 μops we switch to only trying combinations of μops that can be emitted by single μop instructions.

Example: `haddpd` To illustrate how PIPEDREAM finds these port mappings we will now discuss the process for one instruction in more detail. For this we choose `haddpd x, x`, "packed double precision floating-point horizontal add", an SSE instruction working on vectors of 2 elements [54,

[4] and Haswell and Broadwell

	Ports 0, 1		Ports 0, 1, 5		Ports 0, 1, 5, 6	
#μops	**all decompos.**	**single-μop inst. only**	**all decompos.**	**single-μop inst. only**	**all decompos.**	**single-μop inst. only**
1	1	1	1	1	1	1
2	4	4	13	13	40	13
3	8	8	57	57	400	90
4	13	13	168	168	2306	396
5	19	19	402	402	9902	1324
6	26	26	843	843	35228	3699

Table 3.2 – Number of possible μop decompositions PIPEDREAM explores for instructions using only ports 0, 1, 5, 6. **all decompos.** lists the number of decompositions when we consider all possible port mappings. **single-μop inst. only** lists the number of decompositions when we only consider μops generated by single-μop instructions.

Instruction	**μops**	**Instruction**	**μops**	**Instruction**	**μops**
`add r,r`	p_{0156}	`neg r,r`	p_{0156}	`addps x,x`	p_{01}
`sub r,r`	p_{0156}	`adc r,r`	p_{06}	`subps x,x`	p_{01}
`imul r,r`	p_1	`sbb r,r`	p_{06}	`mulps x,x`	p_{01}
`imul r16,r16,i16`	p_1 p_{0156}	`bextr r,r,r`	p_{06} p_{15}	`divps x,x`	p_0
`imul r32,r32,i32`	p_1	`blsi r,r`	p_{15}	`haddps x,x`	p_{01} $2p_5$
`imul r64,r64,i32`	p_1	`bsf r,r`	p_1	`hsubps x,x`	p_{01} $2p_5$
`lea r16,i,r16`	p_{15} p_{0156}	`bswap r32,r32`	p_{15}	`paddb x,x`	p_{015}
`lea r64,i,r64`	p_{15}	`bswap r64,r64`	p_{06} p_{15}	`andps x,x`	p_{015}
`lea r16,i,r16,r16`	p_1 p_{0156}	`shr r,i`	p_{06}	`andnps x,x`	p_{015}
`lea r64,i,r64,r64`	p_1	`shl r,i`	p_{06}	`orps x,x`	p_{015}
`and r,r`	p_{0156}	`sal r,i`	p_{06}	`cmpps x,x`	p_{01}
`andn r,r,r`	p_{15}	`sar r,i`	p_{06}	`comiss x,x`	p_0
`or r,r`	p_{0156}	`rol r,i`	p_{06}	`minps x,x`	p_{06}
`xor r,r`	p_{0156}	`ror r,i`	p_{06}	`maxps x,x`	p_{06}
`cmp r,r`	p_{0156}	`cmovb r,r`	p_{06}	`cvtps2dq x,x`	p_{01}
`test r,r`	p_{0156}	`cmovnb r,r`	p_{06}	`cvtps2pd x,x`	p_{01} p_5
`mov r,r`	p_{0156}	`aesenc r,r`	p_0	`movaps x,x`	p_{015}
`movsx r64,r16`	p_{0156}	`aesdec r,r`	p_0	`movups x,x`	p_{015}
`movzx r64,r16`	p_{0156}	`vcvtph2ps x,x`	p_5 p_{01}	`unpcklps x,x`	p_5

Table 3.3 – μop decomposition for select arithmetic and logical instructions. PIPEDREAM, Agner Fog, and UOPS.INFO all find the same decomposition for these instruction. For vector operations we only list the "packed single" (ps) variant. The other variants behave exactly the same. No AVX2 instructions are listed, they generally have the same decomposition as the corresponding AVX or SSE instructions.

Vol. 2A 3-441]. `haddpd` "horizontally" adds the lower and upper element in each of its two input vectors and stores the resulting two elements in its first operand.

As a first step PIPEDREAM benchmarks the kernel [`haddpd x,x`] containing only the `haddpd` instruction. The rounded results of this measurement are the following:

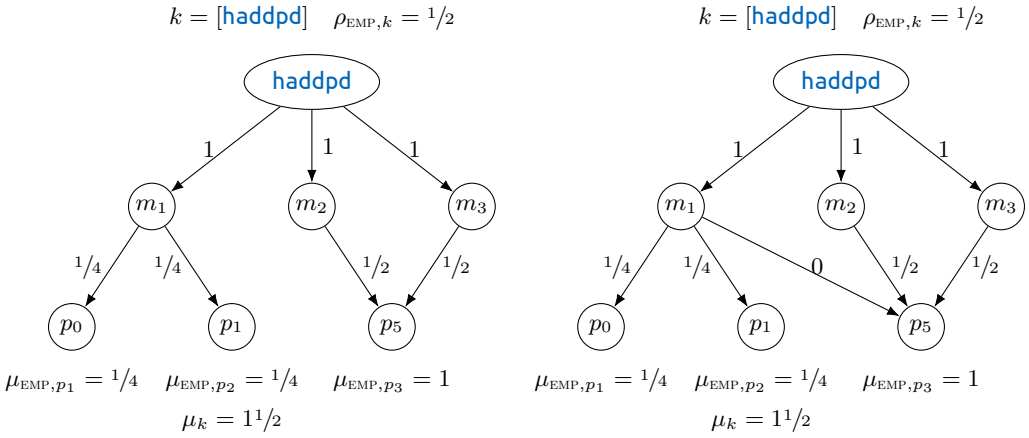
Kernel	**Measurements**					
	ρ_k	$\#\lambda_i$	$\#\mu_i$	μ_{p_0}	μ_{p_1}	μ_{p_5}
[`haddpd x,x`]	½	3	3	¼	¼	1

From this we can deduce that `haddpd` decomposes into three μops and uses three ports: 0, 1, and 5. The instruction produces three fused and three unfused μops, which means that no micro-fusion occurs. Given this port usage and number of μops there are 57 possible port mappings for `haddpd`. Out of those, 23 mappings have the same minimal simulation error for this benchmark with ERR = 0.05. The remaining 34 instruction flow instances only have solutions with very a large error and can be excluded.

To distinguish the 23 remaining mappings PIPEDREAM now runs several benchmarks that combine `haddpd` with the following single-μop instructions: `aesenc`, `bsf`, `unpcklps`. For illustrative purposes we also list the performance characteristics of the single-μop instructions in isolation. PIPEDREAM actually uses more single-μop instruction for different port combinations, but here this does not add any information.

Kernel	Measurements					
	ρ_k	$\#\lambda_i$	$\#\mu_i$	μ_{p0}	μ_{p1}	μ_{p5}
[`aesenc x,x`]	1	1	1	1	0	0
[`bsf r,r`]	1	1	1	0	1	0
[`unpcklps x,x`]	1	1	1	0	0	1
[`aesenc x,x haddpd x,x`]	1	-	-	2/3	1/3	1
[`bsf r,r haddpd x,x`]	1	-	-	1/3	2/3	1
[`unpcklps x,x haddpd x,x`]	2/3	-	-	1/6	1/6	1

PIPEDREAM then calculates the simulation error for each mapping for this additional kernels. Only two mappings remain that produce the same minimal simulation error: p_{01} $2p_5$ and p_{01} $2p_5$.



To break the tie between these two mappings PIPEDREAM chooses the mapping with the least number of edges between M and P: p_{01} $2p_5$. Port 5 is documented to implement vector shuffle instructions and ports 0 and 1 are used by vector addition. This indicates that `haddpd` is internally implemented using two shuffles and one addition.

Finally, PIPEDREAM creates the following two-level resource model for GUS from `haddpd` (using the same notation as for μop decompositions): R_{01} $2\mathsf{R}_5$ $3\mathsf{R}_{015}$ $\mathsf{R}_{\texttt{retire}}$. That is, every time a `haddpd` instruction executes, it uses resource R_{01} once, R_5 twice, R_{015} three times, and $\mathsf{R}_{\texttt{retire}}$ once. Here, $\mathsf{R}_{\texttt{retire}}$ is the global resource modelling the retirement bottleneck of the CPU.

Problem with port 6: There is one group of instructions for which PIPEDREAM currently finds a different μop decomposition than Agner Fog or UOPS.INFO. These are multi μop instructions that emit multiple μops p_{06} that all can execute on either ports 0 and 6. This includes some variants of bit shift and rol instructions as well as some conditional move instructions. Take for example the `shr r, cl` instruction, which according to Agner Fog and UOPS.INFO decomposes into $2p_{06}$. For such an instruction there are 4 possible mappings, all shown in Figure 3.17. By running a benchmark with an instruction that saturates port 0 we can exclude the two mappings in Figure 3.17a. The is the reasoning already presented in Figure 3.6 in Section 3.5.2.

However, there is no single-μop instruction we can use to saturate port 6 by itself. Most single-μop instruction that can execute on port 6 can also execute on at least one other port. The only instructions that can execute only on port 6 are unconditional branches. The throughput of branch instructions is limited by other components than execution ports and they can only saturate port 6 to ~40%. This means that we cannot construct a benchmark that reliably produces measurements that distinguish the two mappings in Figure 3.17b. $2p_{06}$ may be a closer match for the actual port layout of the underlying hardware, but the performance model constructed for GUS from either of those mappings describe the observed performances equally well.

It is unclear how the lack of an instruction that blocks port 6 affects UOPS.INFO. In the published experimental logs UOPS.INFO skirts the issue by not considering p_6 to be a valid μop for multi μop instructions. Unlike port 0, 1, or 5 no measurements blocking only port 6 are taken. This behaviour seems to be slightly incoherent since the algorithm used by UOPS.INFO needs to consider all subsets of ports.

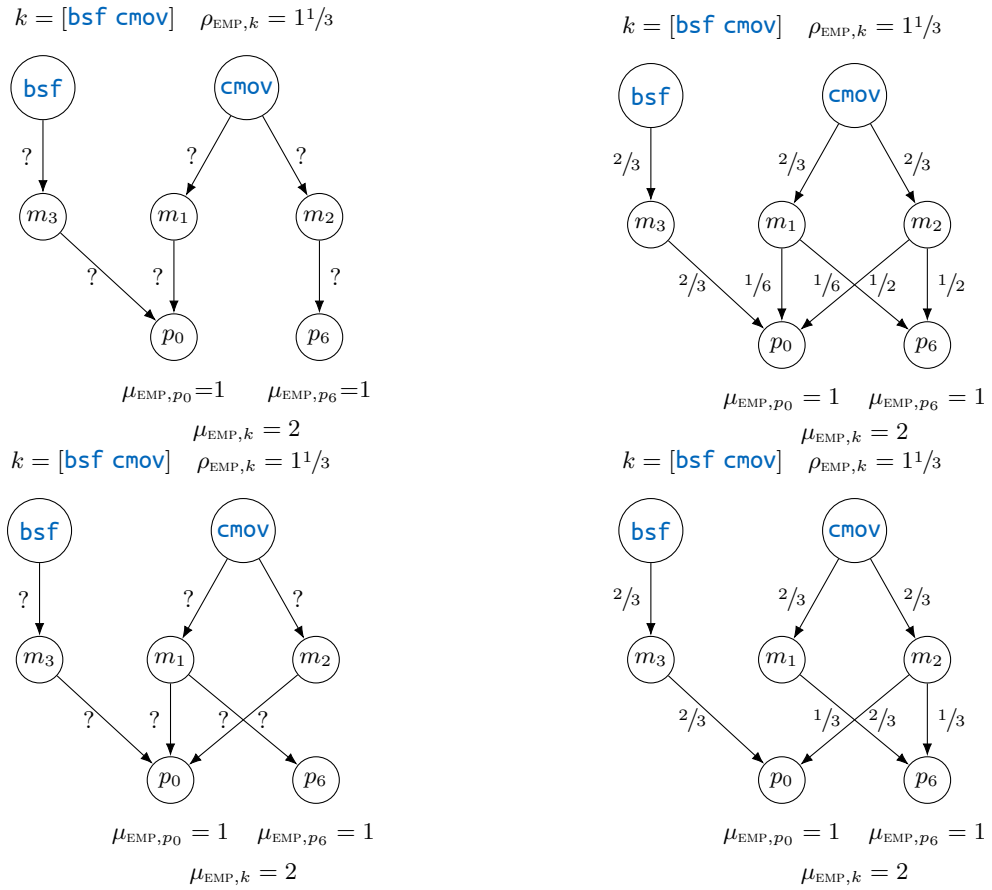
3.6.2.2 Case study II: Store

The simplest store instruction on a current X86 CPU is a `mov` that reads a 64 bit value from general purpose register and stores it to memory. When measuring port usage we can distinguish two variants of `mov` that use different addressing modes. In other ways, the way that memory addresses are encoded is different. The first, variant, which we call simple `mov`, specifies addresses using a base register and a constant immediate displacement. The second variant, complex `mov`, in addition to a base and displacement also uses a second index register to calculate memory addresses. Here we only consider the versions of store using 64 bit addresses and a 32 bit displacement. While there are several other variants that use different register and immediate bit-widths this does not seem to influence the performance or μop decomposition.

An Intel Skylake, as shown in Figure 3.1 has eight execution ports, numbered from zero to seven. Store instructions use ports 2, 3, 4 and 7. Only memory loads also use ports 2 and 3. No other instruction type uses ports 4 or 7. Any store instruction produces at least two μops, but there are single μop loads.

As mentioned earlier store instructions are the only instructions on current Intel X86 architectures that decompose into μops that cannot be produced by any single-μop instruction. This can be detected without the actual mapping by observing that any store instruction decomposes into at least two μops and that stores are the only instructions that use ports 4 and 7. Since the port usage of a store cannot be explained by a sequence of simple μops also produced by single-μop instructions PIPEDREAM falls back to evaluating all possible port mappings to find the μop decomposition.

The following table lists the IPC, MPC, and port usage for `mov` as well as the μop decompo-



(a) These two mappings can be excluded by saturating p_0 with another instruction since there is no valid solution of the corresponding instruction flow instance

(b) Two mappings that have the same average simulation error. PIPEDREAM chooses the mapping with the least number of edges (at the bottom)

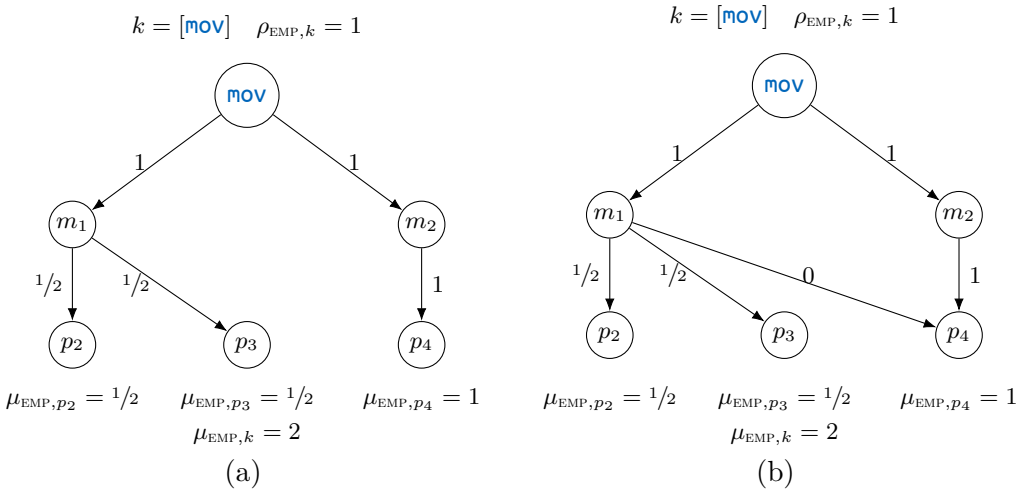
Figure 3.17 – Illustration why the μop decompositions p_6 p_{06} and $2p_{06}$ produce the same performance in our model.

sitions detected by PIPEDREAM, Agner Fog, and UOPS.INFO:

Instruction	Measurements							Decomposition		
	ρ_k	$\#\lambda_i$	$\#\mu_i$	μ_{p_2}	μ_{p_3}	μ_{p_4}	μ_{p_7}	PIPEDREAM	UOPS.INFO	Agner Fog
simple `mov m, r`	1	1	2	1/3	1/3	1	1/3	p_4 p_{237}	p_4 p_{237}	p_4 p_{237}
complex `mov m, r`	1	1	2	1/2	1/2	1	0	p_4 p_{23}	N/A	p_4 p_{237}

For the simple `mov` variant there are 40 different possible decompositions, while for complex `mov` there are 13. In both cases PIPEDREAM finds two best matching mappings that produce the

same simulation error. Below are those for the complex `mov` variant:



These two mappings could only be distinguished with an instruction that exercises extra pressure on p_4, but not on p_2 or p_3. Unfortunately, there is no such instruction on x86. Note that there can never be any flow over the edge between m_1 and p_4 in mapping (b) since p_4 is already saturated by m_2. PIPEDREAM chooses mapping (a) since it has less edges between μops and ports.

Agner Fog's results claim the behaviour of `mov` to be indentical for "all addressing modes" [72, page 238]. This seems to be simply an oversight and implies no testing was done for complex `mov`. UOPS.INFO does not list any results for the complex `mov` variant, only for the simple one. More generally, it does not have any results for any memory access instruction using an index register. As mentioned earlier UOPS.INFO is not able to find μops directly, only μop decompositions. The decomposition for `mov` is hardcoded and used as a building block for finding the decomposition of more complex store instructions.

3.6.2.3 Case study III: Folded load/store and arithmetic

As mentioned earlier, nearly every arithmetic or logical instruction has at least one version where the source operand is loaded from memory or the results is written to memory instead of to registers. These combined arithmetic and load/store instructions all decompose into multiple μops. One or more μops for the actual arithmetic or logical operation, one μop for a load, and two μops for a store.

Pure arithmetic/logical instructions and pure load/store operations are both restricted to using only half the number of execution ports on Skylake. An operation that combines a calculation with a memory access can potentially use all eight execution ports. So the number of different possible μop decompositions PIPEDREAM has to explore is potentially much larger. Consequently, we only explore μop combinations that can be expressed as a combination of μops produced by single-μop instructions. In practice, this works well and we never have to explore more than a few hundred different decompositions for any instruction.

3.7 Conclusion and Future Work

In this chapter, we have introduced GUS, a profiling based performance debugging tool for finding performance bottlenecks using sensitivity analysis, and PIPEDREAM, an automated tool for building CPU performance models. For now, the two tools are still early prototypes and under active development. There are still many cases for which we need to adjust the performance

model of Gus to get more accurate execution time predictions. The two tools are also not fully integrated yet. The port and resource mappings produced by Pipedream, for example, still need to be verified and ported to Gus. We are still working on these issues, and despite their early stage of development, both tools already give promising results.

Gus

Gus guides its users during the process of program optimization by identifying the resource or set of resources that form a performance bottleneck through sensitivity analysis. Using this method Gus points out which optimizations may effectively reduce execution times and also helps predict how much overall performance can still be improved.

The tool combines a fast purely functional CPU emulator with a high-level performance model to predict the execution time of programs. By using a simulator to predict execution times, we can finely control the throughput of individual resources, which we use in turn to drive a fine-grain sensitivity analysis.

For now, we have more or less directly modelled the execution ports of the CPU in Gus. Sensitivity analysis is only performed by varying the throughput of resources and the latency of all instructions at once. One could, however, also perform a more high-level sensitivity analysis, where instead of varying resource capacities we vary the resource usage of some instructions or type of instruction. By, for example, reducing the cache misses caused by the instructions for a given source-level statement one could detect if that statement forms a bottleneck or not. Another example would be to make loads and stores through the stack pointer "free", which would allow detecting if spill code inserted by the compiler is a problem or not.

For the moment Gus assumes, like the ECM performance model, that all accesses to main memory or cache levels above L1 can be overlapped. In other words, we do not take the latency of caches into account. This assumption that all memory accesses can be overlapped only holds for relatively simple programs, like the scientific kernels we have analysed heretofore. To expand the range of programs that Gus can accurately model we are planning to modify the simulator algorithm to also take the latencies of data transfers between cache levels and memory into account.

The Dinero IV cache simulator used in Gus only supports relatively simple cache replacement policies like least-recently-used (LRU). It has no implementation of algorithms commonly used in modern CPUs like pseudo-LRU, LIRS [107], or other adaptive strategies that blend together multiple policies [142]. Adding these policies to Dinero IV or switching to another cache simulator will allow for a more accurate simulation of cache behaviour and consequently a more precise detection of cache bottlenecks.

Another limitation of Gus is the simplistic model of the throughput and interaction between different levels of the cache hierarchy [97]. To address this issue we would like to develop, similarly to the resource mapping for execution ports, a resource mapping for caches.

Another limitation of Gus is that it has no notion of branch prediction or, more importantly, branch misprediction. All branches are optimistically assumed to be predicted correctly and do not cause bubbles in the CPU pipeline. We intend to add a simple branch predictor to Gus to allow it to more accurately model programs with many irregular data-dependent branches. However, we do not plan to model speculative execution. Gus will only attribute a constant penalty with mispredicted branches. Another option would be to add a pessimistic mode to Gus where all branches are assumed to be mispredicted. Together, the optimistic and pessimistic mode give a lower and upper bound on the execution time one can expect. This idea of running both an optimistic and pessimistic simulation could also be applied to other mechansims of the CPU, such as memory prefetching. For the moment Gus is always optimistic.

Pipedream

Pipedream is a prototype tool for measuring performance characteristics of CPUs and reverse-engineering performance models. We have used Pipedream to build a port mapping for the Intel Skylake CPU architecture. The port mapping is comparable in quality to that produced by other authors such as Agner Fog [72] and Abel et al. [3]. Pipedream can automatically find the port mappings for some instructions for which existing approaches fall back to manual analysis or rely on hardcoded assumptions. Pipedream is still a work in progress and for now it can only handle a part of the x86 instruction set.

The first priority for the development of Pipedream is to extend our coverage of x86 instructions that we can benchmark and build port mappings for. We then intend to test our tool on other CPU architectures than Skylake.

We have started working on an alternate version of Pipedream that can find port mappings for CPU architectures without performance counters for the number of μops executed on each individual execution port. This variant of Pipedream could, for example, be used to model processors from AMD. A major stumbling block when constructing for this is the sheer number of possible port mappings that have to be explored. To make Pipedream scale better to complex architectures we will need to implement some heuristics to more intelligently prune the search space. We have spent a considerable amount of time exploring an alternative mixed integer linear program (MILP) encoding of the instruction flow problem where the solver does not only calculate the throughput for a given mapping, but actually finds a mapping that best explains the observed measurements. The advantage of this is that MILP solvers like gurobi already implement very sophisticated heuristics for exploring large search spaces. However, selecting a port mapping and computing its peak throughput naturally leads to two opposing optimization goals. gurobi does support MILP programs with hierarchical objective functions where the solver optimizes for one objective at a time, but this does not allow encoding real multi-objective problems. We where unable to encode this modifed instruction flow problem with a single objective and are currently planning to use other solvers that support multiple opposing objective functions.

We are also working on a version of Pipedream that can even build a resource model for a CPU only with timing measurements. That is, without a counter for the number of μops retired or executed. Here, we do not attempt to reconstruct any information about the μop decomposition of instructions and only try to build an abstract two-level port mapping. Such a two-level mapping does not capture the real layout of the underlying hardware, but it can still capture the performance behaviour accurately enough for building a resource mapping for Gus.

Lastly, we intend to further investigate under which circumstances macro-fusion, micro-fusion, and μop unlamination occurs, and more importantly what stops it from occuring.

Chapter 4

Data-Dependence Driven Optimization Feedback

Contents

4.1 Introduction

The most effective program transformations for improving performance or energy consumption are typically based on the rescheduling of instructions so as improve temporal and spatial locality and uncover both coarse, i.e., thread-level, and fine-grain, i.e., vector-level, parallelism. The two main challenges for these kinds of optimizations are *what* transformations may be applied without breaking the program semantics and *where* in the program should the optimizations be applied to maximize the impact on the overall program performance. For example, detection of parallelism and spatial locality along existing dimensions requires only relatively simple, localized information. Showing the absence of dependencies or finding a small constant dependence distance at the innermost loop level is sufficient to prove that parallelization or vectorization is valid. If, on the other hand, there are more complicated dependencies direct parallelization is not possible. Depending on the structure of the dependencies loop transformations, such as skewing or interchange, can still be used to expose some degree of parallelism. This can then be exploited using other loop transformations such as tiling. The polyhedral model [71] is a powerful tool for finding and applying such rescheduling optimizations [25]. But to judge if a sequence of loop transformations is valid and profitable compilers using the polyhedral model [160, 81, 218, 25, 162, 22] require precise information about data and control-flow dependencies.

Historically, the polyhedral model been designed to work on restricted programming languages for which it is easy to reconstruct exact dependence information statically [48]. A key issue faced when analysing general-purpose languages are the ambiguities introduced by the language itself: for example, the use of pointers typically restricts the ability to precisely characterize the data being accessed due to aliasing. This, in turn, forces the analysis that reconstructs dependencies to use conservative approximations, which unnecessarily restrict the set of legal transformations. In practice polyhedral approaches can only be applied to programs written in a very restrictive style with no function calls, only very simple conditional statements and no indirections [48, 63, 62, 40].

When a region of the source program does fit the syntactic and semantic restrictions of the polyhedral model, it has been shown to be able to successfully find multi-dimensional loop nest transformations that lead to significant performance improvements [26, 160, 211, 81]. While programs can be rewritten to fit these restrictions and help static analyses this requires significant effort and is not guaranteed to succeed. In full programs, and in particular those relying on external binaries visible only to the linker, often the data layout and calling context is inaccessible to static analyses.

Dynamic analysis frameworks address this limitation by reasoning on a particular execution of the program [70, 20]. The feedback provided by existing frameworks mainly informs about the absence of dependencies along some loop in the original program, highlighting opportunities for parallelism [113, 219, 214, 209] or SIMD vectorization [98]. For tools that want to build a polyhedral representation with dynamic information, there are several problems. The main difficulty here is how to model applications including some non-affine dependencies and memory accesses in otherwise affine code. Existing dynamic polyhedral approaches either use overly pessimistic approximations and lose information [199] or do not scale [112, 171].

In this chapter, we present MICKEY, a dynamic analysis tool-chain based on the polyhedral model that addresses these challenges. This tool-chain works on compiled binaries and provides feedback on the applicability and profitability of polyhedral loop transformations. It then uses debugging data to map the suggestions back to the source code to help users implement the transformations. The tool-chain consists of three parts:

- the *front-end*, which instruments binaries to collect execution traces when they execute;
- the *folding algorithm*, which consumes these traces in a streaming fashion to build a compact polyhedral program representation;

- the *back-end*, which uses this representation to find interesting loop transformations.

MICKEY can accurately detect polyhedral data dependencies in programs and scales to real-life applications, which often include some non-affine dependencies in otherwise affine code.

The contributions presented in this chapter are:

1. The development of a compact inter-procedural intermediate representation that captures information useful for polyhedral optimizers such as properties of data dependencies, memory accesses, and induction variables in a uniform manner.
2. the folding algorithm, which builds this representation from program execution traces and scales to real-life applications thanks to a safe polyhedral over-approximation mechanism.
3. MICKEY, a tool that provides optimization feedback using binary instrumentation. MICKEY is the first framework that provides feedback on structured transformation potential for arbitrary statically complicated binaries.

For that, we propose a safe fine-grain polyhedral over-approximation mechanism for such dependencies. That is, our analysis emits a compact program representation allowing a classic polyhedral optimizer to find a wide range of possible transformations. Our analysis also detects the presence of fixed-stride memory accesses and induction variables. Fixed-stride memory accesses are useful for exposing potential loop transformations that improve spatial locality, like vectorization. Detecting induction variables allows removing unnecessary dependencies.

4.2 Illustrative Scenario

This section introduces the problem tackled by this work using a concrete example. It highlights why profiling and dynamic program analyses are efficient tools for performance analysis. It furthermore shows the limitations of existing dynamic techniques and how we overcome them. For this, we use `backprop`, a benchmark from the Rodinia benchmark suite [43]. `backprop` is a supervised learning method used to train artificial neural networks. We focus on the compute kernel `bpnn_layerforward` shown in Figure 4.1. This kernel is also used as a running example throughout the rest of this chapter.

4.2.1 Example problem: `backprop`

The main source of inefficiency in `backprop` is the 2D access to `conn` on line 13. The problem here is that `conn` is laid out in row-major order, but the accesses are in column-major order. This leads to unnecessary cache misses. A loop interchange, which switches the order of `j` and the `k` loop, solves this problem and furthermore unlocks vectorization opportunities. Identifying the profitability of these transformations requires detecting the strided access along the outer `j` loop.

At first glance, all loop bounds and memory access functions seem to be affine functions of loop invariant parameters, `n2` and `n1`, and enclosing loop iterators, `j` and `k`. This kernel should thus be a perfect target for static polyhedral optimizers such as Polly [81], built on LLVM, or Graphite [160], part of GCC. Nevertheless, this is not the case because the `conn` object is not a two-dimensional array, but an array of pointers, each allocated by a separate call to `malloc` as illustrated in Figure 4.2.

Because of that it is impossible to know statically if the pointers `conn[k]`, `l1`, and `l2` alias, that is, whether they refer to the same object at runtime. Due to this, a static compiler has to conservatively assume there is a dependence between the write on line 15 and the reads on line 13. This dependence prevents any transformation of the outer loop. Furthermore, even if the programmer provides non-aliasing information for the three pointers GCC and LLVM, and by extension Polly and Graphite, were still unable to handle this kernel. This is because of the way `conn` is allocated. Since `conn` is an array of pointers, two load instructions are required for

```
1  void bpnn_layerforward(float *l1, float *l2, float **conn, int n1, int n2) {
2    float sum;
3    int j, k;
4
5    /*** Set up thresholding unit ***/
6    l1[0] = 1.0;
7
8    /*** For each unit in second layer ***/
9    for (j = 1; j <= n2; j++) {
10     /*** Compute weighted sum of its inputs ***/
11     sum = 0.0;
12     for (k = 0; k <= n1; k++) {
13       sum += conn[k][j] * l1[k];
14     }
15     l2[j] = squash(sum);
16   }
17 }
18
19 float squash(float x) {
20   return 1.0 / (1.0 + exp(-x));
21 }
```

Figure 4.1 – A compute intensive kernel in `backprop`

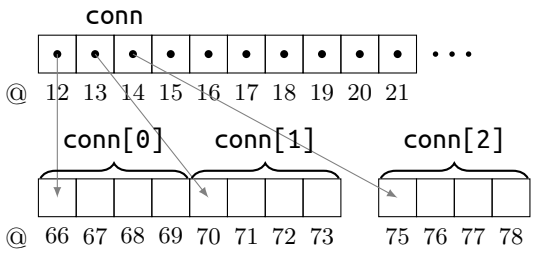


Figure 4.2 – Memory layout for the `conn` array with `n2` = 3. For simplicity pointers and numbers fit in one byte.

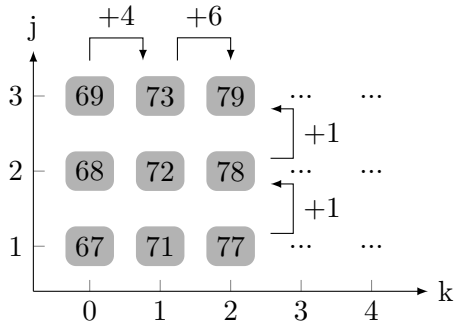


Figure 4.3 – Addresses used to access `conn[k][j]`.

the access `conn[k][j]`, as shown in Figure 4.12. The first one loads `conn[k]` into a temporary variable `tmp` and the second one loads the value of `tmp[j]` into another temporary used in the multiplication. In the IR of LLVM and GCC the fact that these two loads form a single two-dimensional access is lost. Instead, for both compilers `connk][j]` is split into two accesses, where the base pointer of the second access, `tmp`, depends on the dynamic contents of `conn`. Neither Polly nor Graphite are able to model such an indirect access. As any polyhedral optimizer, these compilers require an accurate polyhedral representation for every single memory access and are thus not able to build their polyhedral representation for the `backprop` kernel. Consequently, they are unable to perform any optimizations. Finally, in a static context, the function call on line 15 has to be inlined since it might hide other memory accesses or annotated as free of side effects. In fact, `squash` calls `exp` which writes to the global variable `errno` in case of a numerical error.

Intel's ICC compiler is able to model the 2D-access `conn[k][j]` as a single access, but its loop optimizer is quite limited and cannot perform an interchange since the loop nest in `backprop` is not perfectly nested. Like GCC and LLVM, ICC also is unable to statically resolve the aliasing

between `conn[k]`, `l1`, and `l2` and requires the user to provide no-alias annotations to provide any analysis results at all.

Dynamic analyses can side-step all these problems since they can actually observe the dynamic values of all pointers. They are thus able to detect the absence of pointer aliasing or dependence. However, in the case of `backprop`, reconstructing the striding information required for vectorization from the stream of memory addresses being accessed in a dynamic analysis is still not trivial. This is, again, because the `conn` object is not a two-dimensional array, but an array of pointers allocated by `malloc`. Since `malloc` gives no guarantees on the placement of allocations, the accesses along the innermost `k` dimension do not have a constant stride, but are irregular as shown in Figure 4.3.

Existing dynamic polyhedral approaches [112, 171] try to build a completely affine model and are not able to handle even only partially irregular applications like `backprop`. These algorithms do not take iterator values into account and directly work on a linear stream of memory addresses. The irregularity along the inner `k` loop either stops them from detecting that there even is an outer `j` or causes them to exhibit a prohibitively high time and space complexity. There are approaches that take iterator values into account which allows them to tolerate some irregular accesses using approximation [139, 199]. However, this approximation mechanism is relatively conservative and if two cells of the `conn` are too far apart in memory it will completely give up on trying to model the loop nest. As a consequence none of the existing dynamic approaches scale beyond very small input data sets.

Note that, when using a modern memory allocator for the `backprop` benchmark from Rodinia, `conn[0]`, `conn[1]`, …, `conn[n1]` often do happen to be laid out contiguously in memory. This is true despite the C standard giving no guarantees on the placement of allocations created with `malloc`. However, this behaviour is quite fragile and relies on the fact `backprop` is a simple benchmark program that does not interleave calls to `malloc` and `free`. This allows existing dynamic polyhedral approaches to model and optimize this synthetic benchmark. However, more realistic applications using irregular data structures that could profit from polyhedral optimization, such as sparse matrix algorithms, inherently use non-contiguous data structures and exhibit irregular access patterns [198, Section 5.2].

4.2.2 Solution: MICKEY

Despite the lack of information about aliasing and the presence of a non-affine memory access, the above computation kernel presents an interesting opportunity for optimization. Our dynamic analysis detects:

- the stride-1 access for `conn[k][j]` along the outer dimension j;
- the absence of dependence along dimension j;

From this information, our back-end was able to suggest loop interchange, vectorization, plus tiling which in our case lead to a speedup of $\times 5.3$.

The vectorization opportunity is revealed by looking at the scalar evolution [161, 213] of the addresses being accessed, that is, how they change as a function of the values of the iterators `k` and `j`. In the case of our example, the addresses used for loading `conn[k]` as shown in Figure 4.2, can be described with the function $0j + 1k + 12$, where 12 is the base address of `conn`. This is because `&conn[k]` does not depend on `j`, and `k` is incremented by one on each iteration. Note that due to the gap in the layout of `conn`, the addresses used to access `conn[k][j]` cannot be described by an affine function. This is shown in Figure 4.3. Our analysis is robust against this irregularity along dimension `k` and is able to produce the incomplete function $1j + \top k + 66$. Here, 66 is the base address of the nested array `conn[0]`, and $\top$ represents the fact that accesses are not affine along dimension `k`. However, the obtained function does indicate that the memory address increases by one every iteration of dimension `j`. We refer to this as a *stride-1* access.

The folding algorithm discovers not only the structure of memory accesses but also the structure of data dependencies in general. In our running example, it detects that there is no dependence between the reads on line 13 and the write on line 15. It is worth mentioning that while the algorithm is exactly the same for both, the structure of memory accesses and dependencies are detected separately. The folding algorithm can thus handle cases where accesses are non-affine, and dependencies are affine, and vice versa. Here the irregularity of the former does not hinder the folding algorithm from finding the structure of the latter.

The stride-1 access along dimension `j` allows deducing that SIMD vectorization might be profitable. Since `j` is not the innermost loop, it is necessary to perform a loop interchange before vectorizing. That this loop interchange is valid is clear from the absence of dependencies between the two loops observed by our tool for the profiled execution. Our analysis, like any dynamic approach reasoning on an execution, cannot guarantee that this holds in general, but it can still provide useful feedback. Note that the interchange will require an array expansion of the `sum` variable along with a new 1-dimensional loop iterating over `j` to fill the `l2` array.

4.3 Overview of MICKEY

MICKEY is a dynamic binary analysis tool implemented as a set of plugins for the TPI QEMU-plugin instrumentation interface presented in Section 2.3. Plugins primarily work at the level of the generic QEMU IR, making them CPU architecture agnostic. The polyhedral analysis backend of MICKEY, described in Section 4.6, is based on the PoCC compiler [162] that implements the PluTo scheduler [26]. An overview of the general structure of MICKEY is shown in Figure 4.4.

The first objective of MICKEY is to construct a *useful and analysable* representation from a carefully generated execution trace of the program Building such a representation constitutes our key set of contributions, and covers the first three stages below. The fourth and last stage then uses the polyhedral model to find optimizing loop transformations. Note that the goal here is not to automatically perform transformations, but instead to assist the user in figuring out where optimizations should be implemented, and more importantly which ones.

Interprocedural control structure

This first stage of MICKEY, described in Section 4.4.1, dynamically discovers both the intra-procedural CFGs and the interprocedural CG from an optimized binary. Using this information we construct the *interprocedural loop context tree* which combines the notions of loop-nesting-trees and their equivalent for call graphs, the *recursive-component-set*, both of which are used in the next stage. The first stage is implemented using an extended version of the basic CFG/CG reconstruction plugin described in Section 2.4.1.1.

Dynamic Dependence Graph

The second stage of MICKEY generates the actual Dynamic Dependence Graph (DDG) [153, 147, 46], a trace representing both the dynamic execution of a program's instructions as well as its data dependencies. It uses the interprocedural loop-call-context-tree to construct dynamic interprocedural iteration vectors (dynamic IIVs), a new inter-procedural representation of the program execution that unifies two abstractions: the intra-procedural schedule tree [111, 216] and the inter-procedural calling-context-tree [7]. The DDG, loop-call context tree (LCCT), and dynamic IIV are described in Section 4.4.3.

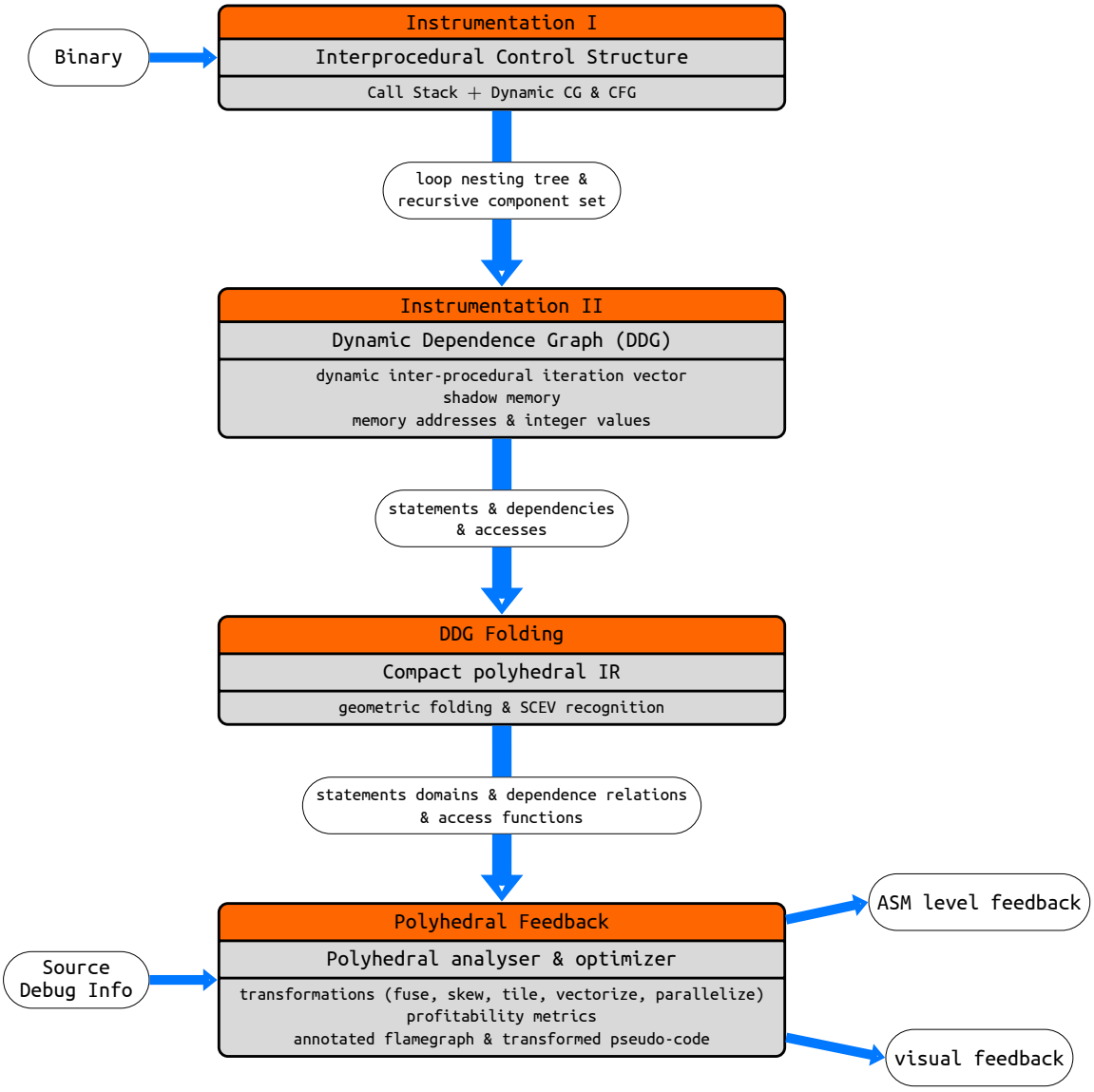


Figure 4.4 – Overview over the MICKEY framework.

Compact polyhedral IR

The third stage compacts, or folds, the DDG into a polyhedral program representation. Essentially, it merges sequences of individual operations in the trace are into sets using the geometry of the multi-dimensional space spanned by the dynamic IIVs. Intuitively, when a loop executes it generates one execution of a statement for each loop iteration and therefore one element in the trace for each execution. This stage amounts to folding those points back into a loop-equivalent representation, which itself can be the input of a polyhedral compiler. We also handle the general case, where folding occurs not only across multiple loops but also across, possibly recursive, procedure calls. This process is presented in Section 4.5.

This stage also detects which instructions in a program are used to increment loop counters and finds strides in memory accesses. The folding algorithm does this by building scalar evolution expressions (SCEVs) [161, 213] for loads, stores and integer arithmetic instructions. This technique borrows ideas from trace compression algorithms [112, 171], with the notable

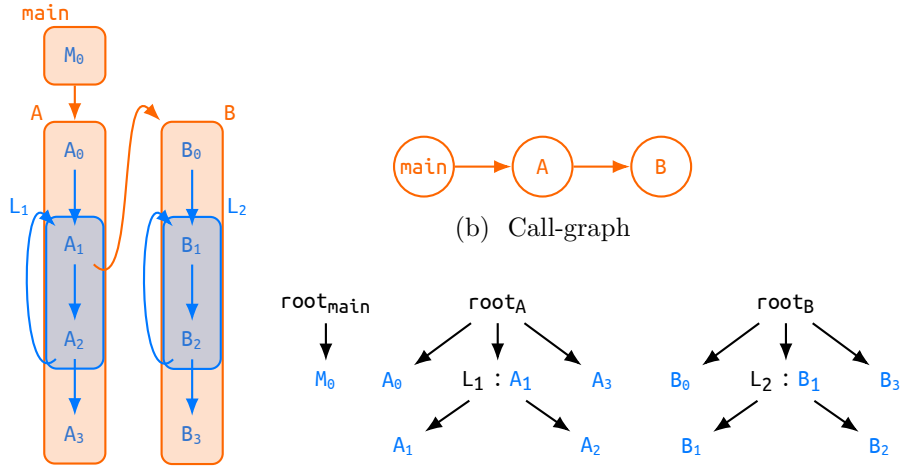
difference that it uses the geometry of iteration spaces which allows it to scale to large, partially irregular programs.

Polyhedral feedback

The last stage of MICKEY consists of the polyhedral analysis back-end. This back-end analyses the polyhedral DDG and suggests sequences of program transformations needed to expose multi-level parallelism and improved data locality, along with numerous statistics on the original and potentially transformed program including parallelization and vectorization potential. Section 4.6 describes this last step as well as how MICKEY provides human interpretable feedback to the user.

4.4 Interprocedural Program Representation

As most of the execution time of a program is usually spent in loops, MICKEY has been designed to find loop-based transformations. In practice, however, as illustrated in Figure 4.5, interesting loop nests are often spread across multiple functions over call chains, obfuscating them to traditional static analysis. MICKEY is designed to be able to represent both classical loops as well as function calls and returns in a unified way with polyhedra [222, 71].



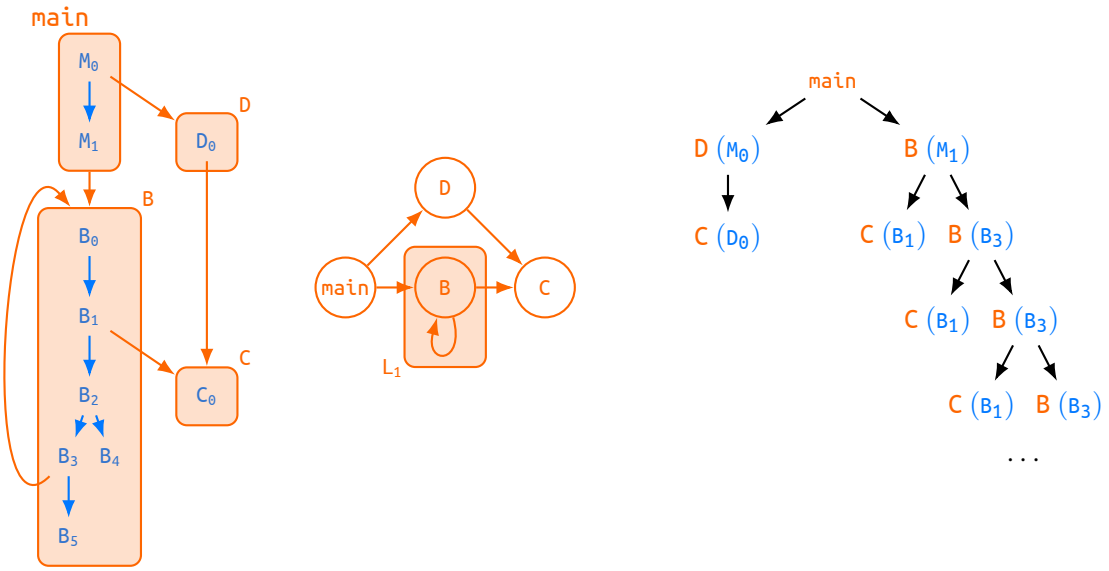
(a) Example program (b) Call-graph (c) Loop-nesting-forest

Figure 4.5 – Example of a loop nest that spans multiple functions. CFG nodes and edges in blue, CG nodes and edges in orange

MICKEY has been designed to handle several scenarios.

1. Nested loops containing function calls with side effects: here calls can be viewed as atomic instructions, but profiling the storage locations they access or modify and compute the dependencies between each call and the rest of the loop body is required.
2. Loops containing function calls that themselves contain loops, as shown in Figure 4.5: here the objective is to view the whole interprocedural region as a multi-dimensional loop nest.
3. Recursive functions, as shown in Figure 4.6: here the primary objective is to avoid the depth of the generated representation to grow proportionally with the depth of the call stack; A secondary objective is to detect any underlying regular loop structure amenable to polyhedral compilation (after recursion-to-iteration conversion).

This section explains how to go from the execution of a program to a representation of its interprocedural control structure.



(a) Example program (b) Call-graph and recursive-component-set (c) Calling-context-tree with call-site information. Call sites for every call in parenthesis

Figure 4.6 – Example of a loop in the call graph of a recursive program. CFG nodes and edges in blue, CG nodes and edges in orange

The first step of MICKEY is to extract the inter- and intra-procedural static control structure from a program in compiled binary form. This static control structure of the program will then be used in a later stage of MICKEY to transform the stream of raw events (`jump`/`call`/`return`) gathered during execution into loop events (`entry`/`iterate`/`exit`).

MICKEY performs this through instrumentation of `jump`, `call`, `return` and a few other instructions to dynamically trace any transfer of control during execution. It thus "dynamically" extracts the CFG of each function and then proceeds to statically build the *loop-nesting-tree* (see Section 4.4.1). MICKEY also dynamically extracts the CG of the whole program and builds the counterpart of loop-nesting-trees for call-graphs, the *recursive-component-set* (see Section 4.4.2).

4.4.1 Control-flow-graph and loop-nesting-tree

The notion of block used here is not exactly the same as a basic block, but that of an *execution block*, as detected by the control-flow tracing plugin described in Section 2.4.1.1. Since a translation block (TB) in QEMU can span multiple execution blocks, we also insert instrumentation inside TBs to raise virtual fall-through events where appropriate.

The algorithms presented in this section are an extension of the simple QEMU plugin described in Section 2.4.1.2.

For the profiled CFG, the loop detection algorithm used by MICKEY are computed in a single pass by the algorithm described in [220]. As formalized by Ramalingam in [167], a loop-nesting-forest can be recursively defined as follows:

1. Each stronlgy connected component (SCC) of the CFG containing at least one cycle constitutes the *region* of an outermost loop;
2. for each loop, one of its *entry* nodes is designated the *header* node of the loop;
3. all edges within the loop that target its header node are called *back-edges*;
4. "removing" all back-edges from the CFG allows one to recursively define the next level of sub-loops.

There exists an almost linear time algorithm for this.

Input:
- `event`: Branch event.
- `live_loops`: Stack of currently live loops. From outermost to innermost.

Emitted events:
- `E`($\mathbf{L}, \mathbf{H}$): entry into loop L; H is the header.
- `I`($\mathbf{L}, \mathbf{H}$): iteration of loop L; H is the header.
- `X`($\mathbf{L}, \mathbf{B}$): exit of loop L, jumping to basic-block B
- `N`($\mathbf{B}$): local jump to basic-block B.

```
1  if event.type is local-jump:
2    B = event.dst
3    while L = live_loops.peek() and L.isCFG and B not in L:
4      L.visiting = False
5      live_loops.pop()
6      emit X(L,B)
7    if B is loop-header:
8      H = B
9      L = H.loop
10     if L.visiting == False:
11       L.visiting = True
12       live_loops.push(L)
13       emit E(L,H)
14     else:
15       emit I(L,H)
16   emit N(B)
17 else:
18   ... # Algorithm 4.2
```

Algorithm 4.1 – CFG-loop events generated from a `jump` event.

An example CFG and its corresponding loop-nesting-tree is given in Figures 4.7a and 4.7b. Here, the CFG can be partitioned into one SCC (giving rise to loop $\mathtt{L}_1$ with B as its header) plus two separated basic-blocks A and E. The region of $\mathtt{L}_1$, once its back-edge (D, B) has been removed, contains one sub-SCC formed by the loop $\mathtt{L}_2$ and the basic-block B. Among the two entry nodes, C and D of $\mathtt{L}_2$ only one, C, is selected to be its header. As depicted in Algorithm 4.1, those notions (region, header, back-edge) are important as they allow associating loop events (E: entry, I: iterate, and X: exit) with the stream of raw control events (`jump`, `call`, `return`) that are produced by our instrumented execution. Here, generation of loop events is driven by the control event "`jump`" that is emitted by our instrumentation. Whenever we detect the start of a new iteration of a given loop (line 13), all live sub-loops are considered exited, that is, we emit an "`exit`" event (line 6). This is especially important for recursive loops as further explained in Algorithm 4.2.

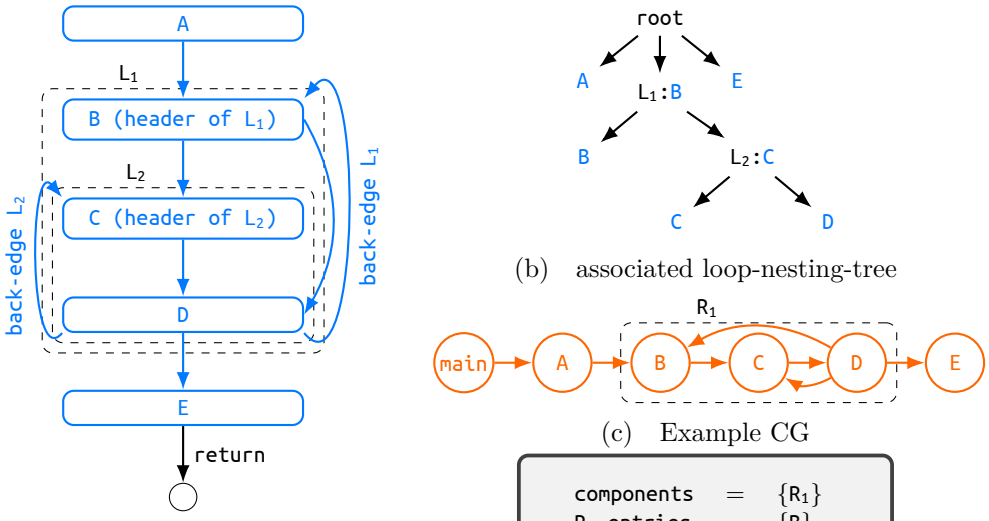
4.4.2 Call-graph and recursive-component-set

To uniquely identify each dynamic execution of an instruction, also called a *dynamic instruction*, MICKEY uses interprocedural iteration vectors. This is described in more detail in Section 4.4.3.

Note, that the modelling of programs containing recursive function calls with calling-context-paths is memory inefficient since the length of the paths is proportional to the depth of the recursion. While recursive function calls are found to be very uncommon in performance critical code [1], we do need to handle them to ensure robustness. MICKEY handles recursive programs

[1] the Rodinia benchmark suite, for example, does not use any recursion

in an elegant way. The main data structure for treating recursive control flow is the *recursive-component-set* which is for the call-graph what the loop-nesting-tree is for the control-flow-graph. An example of a recursive-component-set is shown in Figure 4.7.



(a) Example CFG

(b) associated loop-nesting-tree

(c) Example CG

(d) associated recursive-component-set

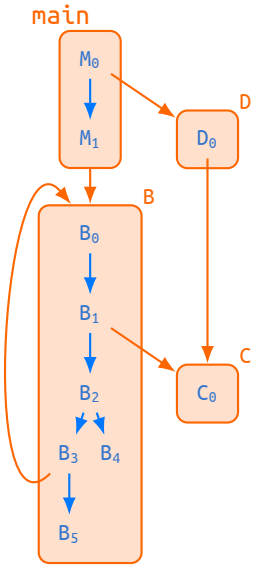
Figure 4.7 – Example CFG/CG and associated loop-nesting-tree/recursive-components-set

Before providing more details, we go through the illustrative example 1 show in Figure 4.8. Here, the edge from $\mathtt{B_3}$ to $\mathtt{B_0}$ (in orange) is not a CFG-edge but a recursive call to B from call site $\mathtt{B_3}$. The recursive-component-set computed from the CG contains one SCC with a cycle consisting of a single function B. This example raises several questions: 1. *Should* $\mathtt{C_0}$ *be part of the associated recursive loop?* It actually depends on the context: It should, when called from $\mathtt{B_1}$, while it should not when called from $\mathtt{D_0}$. 2. *What about* $\mathtt{B_5}$*?* One should observe that $\mathtt{B_5}$ will be executed as many times as the number of recursive calls to B: In other words, $\mathtt{B_5}$ should be part of a loop.

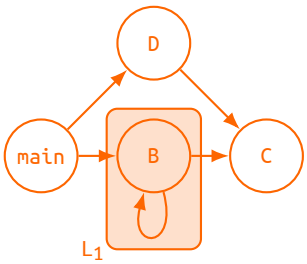
To conclude, while CG-cycles clearly represent potential dynamic loop structures, a CG-edge does not have the same semantic as a CFG-edge. In particular, *a* `call` *will never exit a recursive loop.* For MICKEY, the recursive loop "$\mathtt{L_1}$" of Ex. 2 is a dynamic notion defined as follows: 1. Entering $\mathtt{L_1}$ is characterized by a call to B (step 1). 2. $\mathtt{L_1}$ is exited when this (first) call gets unstacked, that is, when the number of returns reaches the number of calls (step 22). 3. Everything executed in between is part of the loop and iteration, and the corresponding increment of induction variables takes place whenever a call/return to/from B occurs (steps 10,15,20,and 21). As one can see, once the recursive-component-set has been computed from the CG, the only relevant information dynamically used to devise recursive-loop events corresponds to the *header* functions, here B.

As already stated, the recursive-component-set is for the CG what the loop-nesting-tree is for the CFG. In a similar way it can be recursively defined as follows:

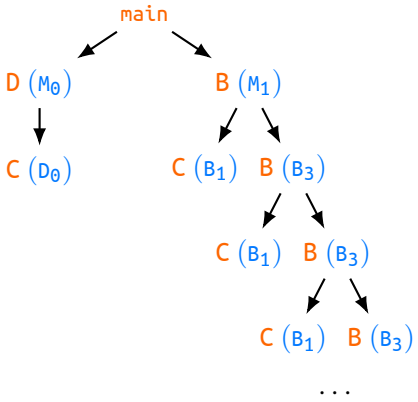
1. Find all the top-level SCCs with at least one cycle in the CG. Each gives rise to a distinct recursive-*component*.
2. For each component, determine all its *entry* nodes.
3. Repeat the following phases until no more cycles exist:
   (a) For a given SCC, choose an entry node and add it to the *headers*-set of the recursive-



(a) Example program



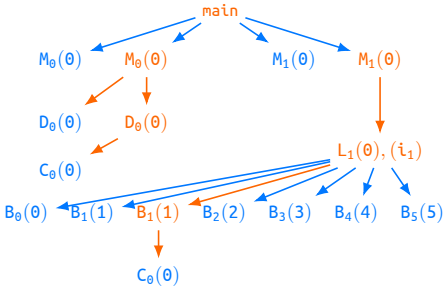
(b) Call-graph and recursive-component-set



(c) Calling-context-tree with call-site information. Call sites for every call in parenthesis

step	:	trace	call-stack	event	dynamic IIV
		M			$(\perp)$
1	:	M_0	M	$N(M_0)$	(M_0)
2	:	D_0	M/D	$C(D_0)$	(M_0/D_0)
3	:	C_0	M/D/C	$C(C_0)$	$(M_0/D_0/C_0)$
4	:	M_1	M	$R^2(M_1)$	(M_1)
5	:	B_0	M/B	$E_C(L_1, B_0)$	$(M_1/L_1, 0, B_0)$
6	:	B_1	M/B	$N(B_1)$	$(M_1/L_1, 0, B_1)$
7	:	C_0	M/B/C	$C(C_0)$	$(M_1/L_1, 0, B_1/C_0)$
8	:	B_2	M/B	$R(B_2)$	$(M_1/L_1, 0, B_2)$
9	:	B_3	M/B	$N(B_3)$	$(M_1/L_1, 0, B_3)$
10	:	B_0	M/B/B	$I_C(L_1, B_0)$	$(M_1/L_1, 1, B_0)$
11	:	B_1	M/B/B	$N(B_1)$	$(M_1/L_1, 1, B_1)$
12	:	C_0	M/B/B/C	$C(C_0)$	$(M_1/L_1, 1, B_1/C_0)$
13	:	B_2	M/B/B	$R(B_2)$	$(M_1/L_1, 1, B_2)$
14	:	B_3	M/B/B	$N(B_0)$	$(M_1/L_1, 1, B_3)$
15	:	B_0	M/B/B/B	$I_C(L_1, B_0)$	$(M_1/L_1, 2, B_0)$
16	:	B_1	M/B/B/B	$N(B_1)$	$(M_1/L_1, 2, B_1)$
17	:	C_0	M/B/B/B/C	$C(C_0)$	$(M_1/L_1, 2, B_1/C_0)$
18	:	B_2	M/B/B/B	$R(B_2)$	$(M_1/L_1, 2, B_2)$
19	:	B_4	M/B/B/B	$N(B_4)$	$(M_1/L_1, 2, B_4)$
20	:	B_5	M/B/B	$I_R(L_1, B_5)$	$(M_1/L_1, 3, B_5)$
21	:	B_5	M/B	$I_R(L_1, B_5)$	$(M_1/L_1, 4, B_5)$
22	:	$\perp$		$X_{R^2}(L_1, \perp)$	$(\perp)$

(d) Example trace, associated loop events, and dynamic interprocedural iteration vectors. R^2 stands for two consecutive returns



(e) Dynamic schedule tree

$\{M_0() : \}$ $\{M_0/D_0() : \}$
$\{M_0/D_0/C_0() : \}$ $\{M_1() : \}$
$\{M_1/L_1/B_0(i) : 0 \leq i \leq 2\}$
$\{M_1/L_1/B_1(i) : 0 \leq i \leq 2\}$
$\{M_1/L_1/B_1/C_0(i) : 0 \leq i \leq 2\}$
$\{M_1/L_1/B_2(i) : 0 \leq i \leq 2\}$
$\{M_1/L_1/B_3(i) : 0 \leq i \leq 1\}$
$\{M_1/L_1/B_4(i) : i = 2\}$
$\{M_1/L_1/B_5(i) : 3 \leq i \leq 4\}$

(f) Folded domains

Figure 4.8 – Example 1. Dynamic schedule tree. CFG nodes and edges in blue, CG nodes and edges in orange

component, i.e., top-level SCC, it belongs to.
(b) Remove all edges inside this SCC that point to this node.

The loop-nesting-forest construction algorithm can easily be adapted to build the recursive-component-set in almost linear time. The end result of the algorithm is a possibly empty set of recursive-components where each recursive-component has a non-empty set of headers and a non-empty set of entries. Algorithm 4.2 uses this data structure to associate loop events (entry, iterate, and exit) to control events (`call` and `return`). Here:

1. Entering a recursive loop is characterized by a call to a component's entry function (line 5).
2. A new iteration is started whenever a call/return to/from one of the components' header occurs (line 9).
3. An exit occurs when all the iterating calls to the headers have been unstacked, that is, when the number of returns equals the number of calls, and we are returning from the original function that entered the loop (line 25).

Tracking the state of the call stack is done with the following two data structures:

1. `L.stackcount` represents for a recursive-component L a counter of the number of calls-to a header minus the number of returns from it.
2. `L.entered_through` represents the function through which L was entered. If a recursive component is not currently live this is set to `undef`.

4.4.3 DDG: Dynamic Dependence Graph

The objective of the second stage of MICKEY ("Instrumentation II") is to profile the dynamic dependence graph of a given execution, that is, to build a graph that has one vertex for every dynamic instruction and one edge for every data dependence. Because we want to enable feedback with structured loop transformations, we need to map this graph to a "geometric" space that reflects the structural properties of the program. To this end, we tag each dynamic instruction with its iteration vector (IV). The IVs uniquely identify each dynamic instruction and naturally span a geometric space. A data dependency is then simply represented as the pair of the IVs of the producer and the consumer.

To handle interprocedural programs we also need a notion of calling context that is scalable in the presence of recursive calls. Our dynamic interprocedural IVs (dynamic IIVs) described in this section addresses those objectives by unifying two notions: 1. Kelly's mapping, which describes intraprocedural IVs and is used by the polyhedral framework [111]. 2. Calling-context-paths, used by profiling feedback tools. We first briefly recall those two notions.

Kelly's mapping

For a given function, Kelly's mapping can be explained using a form of *schedule tree* [216] as shown in Figure 4.9. Here a schedule tree is nothing else than a decorated loop-nesting-forest. The first decoration consists of associating a "static" index to each loop and basic-block: Recall the recursive characterization of loops given by Ramalingam in the previous section. For a given loop region (e.g. L_j in the schedule-tree of the fused version in Figure 4.9), its sub-regions (once back-edges have been removed – here statements S and T) form a directed-acyclic-graph (reduced DAG represented in dashed in Figure 4.9) that can be numbered using a topological order. This numbering is used to index the corresponding nodes at every level of the loop-nesting-tree (e.g. $S(0)$ and $T(1)$ for the fused schedule or $L_i(0)$ and $L_{i'}(1)$ for the fissioned one). The second decoration consists of associating a canonical induction variable, that is, an induction variable that starts at value 0 and increments by 1, to each loop-vertex. For example, the loop-vertex associated with L_j is labelled with L_j as well as the static index 0 followed by its canonical variables j, resulting in $L_j(0),(j)$. For any given statement, an IV with Kelly's

Input:

- `event`, `live_loops` (same as for Algorithm 4.1)

Emitted events:

- $\texttt{E}_\texttt{C}(\mathbf{L}, \mathbf{B})$: `call` to a function header of recursive-component L and entry into the corresponding loop. B is the current basic-block after the call.
- $\texttt{I}_\texttt{C}(\mathbf{L}, \mathbf{B})$ / $\texttt{I}_\texttt{R}(\mathbf{L}, \mathbf{B})$: `call`-to / `return`-from a function header of recursive-component L and iteration of the corresponding loop. B is the current basic-block after the call/return.
- $\texttt{X}_\texttt{R}(\mathbf{L}, \mathbf{B})$: `return` from a function header of recursive-component L and exit from that loop.

```
1  if event.type is call:
2    F = event.callee
3    B = event.dst
4    L = F.recursive_component
5    if F in L.entries and L.entered_through == undef:
6      L.was_entered_by = F
7      live_loops.push(L)
8      emit E_C(L,B)
9    elif F == L.header:
10     while L' = live_loops.peek() and L' in L:
11       L'.visiting = False
12       live_loops.pop()
13       emit X(L', B)
14     L.stackcount++
15     emit I_C(L, B)
16   else: emit C(F, B)
17 if event.type is return:
18   F = event.callee
19   B = event.dst
20   while L = live_loops.peek() and L.isCFG and L in F:
21     L.visiting = False
22     live_loops.pop()
23     emit X(L', B)
24   L = F.recursive_component
25   if F in L.entries and L.stackcount == 0 and L.entered_through == F:
26     L.entered_through = undef
27     emit X_R(L,B)
28   elif F is L.header:
29     L = F.recursive_component
30     L.stackcount--
31     emit I_R(L,B)
32 else:
33   ... # Algorithm 4.1
```

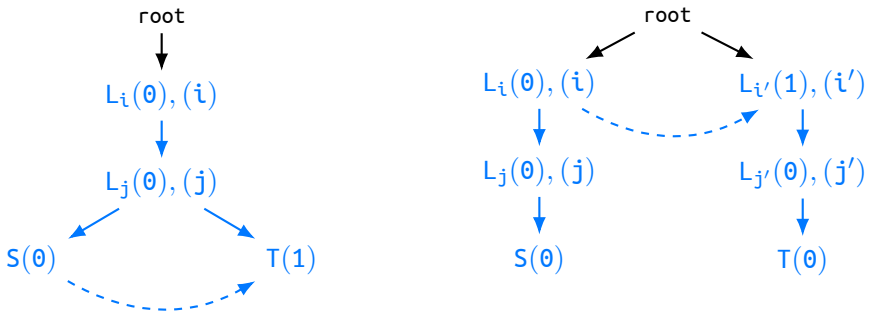
Algorithm 4.2 – Different recursive-loop events generated from a `call` or a `return` event.

mapping is nothing else than the vector given by the path from the root to its corresponding leaf, called its *iteration vector*. Figure 4.9 shows this mapping both in its textual form using region names and numerical form using indices. As one can see, an interesting property of the numerical form of this mapping is that the scheduling of the original code is given by the lexicographical order of the so obtained iteration vectors (unique per dynamic instance).

```
for (i = 0; i < N; i++)
  for (j = 0; j <= i; j++)
    { S; T; }
```

```
for (i1 = 0; i1 < N; i1++)
  for (j1 = 0; j1 <= i1; j1++)
    { S; }
for (i2 = 0; i2 < N; i2++)
  for (j2 = 0; j2 <= i2; j2++)
    { T; }
```

(a) Nested loop before and after fission



(b) Corresponding schedule trees. Dashed edges indicate ordering constraints in the program.

$$\begin{aligned} \mathtt{S} &\to [\mathtt{L_i}, \mathtt{i}, \mathtt{L_j}, \mathtt{j}, \mathtt{S}] \\ &\to [\mathtt{0}, \mathtt{i}, \mathtt{0}, \mathtt{j}, \mathtt{0}] \end{aligned} \qquad \begin{aligned} \mathtt{S} &\to [\mathtt{L_{i_1}}, \mathtt{i_1}, \mathtt{L_{j_1}}, \mathtt{j_1}, \mathtt{S}] \\ &\to [\mathtt{0}, \mathtt{i_1}, \mathtt{0}, \mathtt{j_2}, \mathtt{0}] \end{aligned}$$

$$\begin{aligned} \mathtt{T} &\to [\mathtt{L_i}, \mathtt{i}, \mathtt{L_j}, \mathtt{j}, \mathtt{T}] \\ &\to [\mathtt{0}, \mathtt{i}, \mathtt{0}, \mathtt{j}, \mathtt{1}] \end{aligned} \qquad \begin{aligned} \mathtt{T} &\to [\mathtt{L_{i_2}}, \mathtt{i_2}, \mathtt{L_{j_2}}, \mathtt{j_2}, \mathtt{T}] \\ &\to [\mathtt{1}, \mathtt{i_2}, \mathtt{0}, \mathtt{j_2}, \mathtt{0}] \end{aligned}$$

(c) Corresponding Kelly's mapping / iteration vector

Figure 4.9 – Schedule tree and Kelly's mapping

Calling-context-tree

Differently from the schedule tree, the calling-context-tree [7] (CCT) is only enumerative, i.e., it does not contain any loop indices and reflects a dynamic behaviour. In other words, it encodes the dynamic-call-tree in a *compact* way. The calling-context-tree of the example in Figure 4.8a can be seen in Figure 4.8c. This figure is slightly different from the *original* CCT, but corresponds to the current practice. To differentiate two calls to a common function from different basic-blocks, callees are labelled with their call site. In Figure 4.8c they are highlighted in blue and parenthesized. In this example, a calling-context-path, for example $\mathtt{M_1}/\mathtt{B_3} \ldots \mathtt{B_3}/\mathtt{B_1}/\mathtt{C}$, can be as long as the number of recursive calls to B. However, the calling context is fully encoded, making it possible to differentiate the different contexts within which basic-block $\mathtt{C_0}$ is executed. In the current example, one wants all the repeated calls from $\mathtt{B_1}$ to $\mathtt{C}$ to be folded into one element.

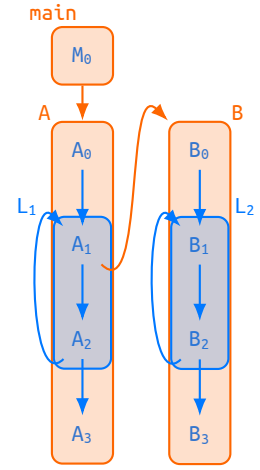
Dynamic interprocedural iteration vector (dynamic IIV)

The dynamic interprocedural iteration vector (dynamic IIV) is basically a combination of Kelly's mapping and the CCT. Similarly to Kelly's mapping, the dynamic IIV alternates between context-ids and canonical induction variables. Differently from Kelly's mapping, but more like the CCT, each context-id in a dynamic IIV is a, possibly empty, stack of calling contexts and the identifier for a basic block.

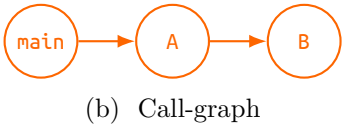
Examples of dynamic IIVs are shown in Figure 4.11: Function A contains a loop $\mathtt{L_1}$ that

	interprocedural	loops
schedule tree / IV	✗	✓
CCT / calling-context-path	✓	✗
dynamic schedule tree / dynamic IIV	✓	✓

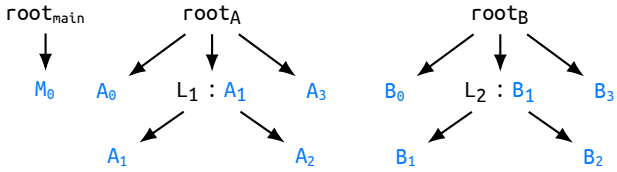
Figure 4.10 – *Dynamic* schedule tree ≡ schedule tree ∪ CCT



(a) Example program



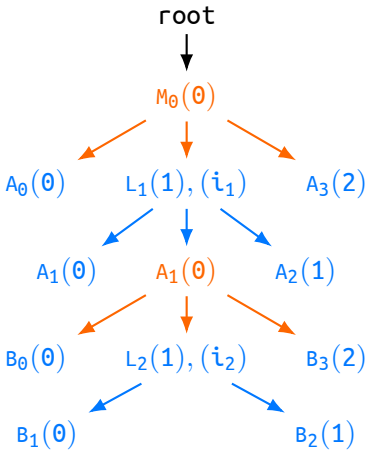
(b) Call-graph



(c) Loop-nesting-forest

step : trace	event	dynamic IIV
		(⊥)
1 : M_0	N(M_0)	(M_0)
2 : A_0	C(A_0)	(M_0/A_0)
3 : A_1	E(L_1,A_1)	($M_0/L_1, 0, A_1$)
4 : B_0	C(B_0)	($M_0/L_1, 0, A_1/B_0$)
5 : B_1	E(L_2,B_1)	($M_0/L_1, 0, A_1/L_2, 0, B_1$)
6 : B_2	N(B_2)	($M_0/L_1, 0, A_1/L_2, 0, B_2$)
7 : B_1	I(L_2,B_1)	($M_0/L_1, 0, A_1/L_2, 1, B_1$)
8 : B_2	N(B_2)	($M_0/L_1, 0, A_1/L_2, 1, B_2$)
9 : B_3	X(L_2,B_3)	($M_0/L_1, 0, A_1/B_3$)
10 : A_2	R(A_2)	($M_0/L_1, 0, A_2$)
11 : A_1	I(L_1,A_1)	($M_0/L_1, 1, A_1$)
...	...	...

(d) Example trace, associated loop events, and dynamic IIVs



(e) Dynamic schedule tree

Figure 4.11 – Example 2. Dynamic schedule tree. CFG nodes and edges in blue, CG nodes and edges in orange

contains a call to function B, itself containing a loop L_2. In this interprocedural example basic-block B_1 in loop L_2 belongs to a two-dimensional nested loop. This is reflected by our dynamic IIV (see steps 5,7 of Figure 4.11d) which will be ($M_0/L_1, i_1, A_1/L_2, i_2, B_1$) where i_1/i_2 are the canonical induction variables of loop L_1/L_2. Here, our context-ids are loop-ids, e.g., L_2, or statement-IDs, e.g., B_0. Each context-ID is decorated with a, possibly empty, call stack, e.g., A_1.

Similar to Kelly's mapping one can also construct a dynamic schedule tree from the dynamic IIVs of a program execution. Also, note that the schedule tree is for the dynamic IIVs what the calling-context-tree is for the set of calling context paths. The schedule tree for our example is shown in Figure 4.11e. The relationship between these three structures is summarized in

Figure 4.10. As described in more details later, MICKEY exposes the schedule tree to the programmer in the form of a flame-graph [79].

Figure 4.8a illustrates how the recursive-component-set is used to "fold" calling-context-paths in the presence of recursive calls. Here, $\texttt{M}_1$ is the caller site (step 4) before entering the recursive loop $\texttt{L}_1$ (step 5). Looking at $\texttt{C}_0$, its multiple instances are indexed by the corresponding recursive-loop induction variable $\texttt{i}_1$ (respectively 0,1, and 2 at steps 7, 12, and 17). Within the loop it gets executed when C is called from $\texttt{B}_1$. The associated dynamic IIV is $(\texttt{M}_1/\texttt{L}_1, \texttt{i}_1, \texttt{B}_1/\texttt{C}_0)$. As already mentioned, $\texttt{B}_5$ is also part of the loop: Indeed there are as many instances of $\texttt{B}_5$ as there are calls to B from $\texttt{B}_3$ (2 calls at steps 10,15). Observe that the value of $\texttt{i}_1$ does not reflect the size of the call stack: It does not go up and down. It keeps increasing. The main reason for doing so is related to our polyhedral back-end: 1. we want our indexing to be lexicographically increasing; 2. we want to match the semantics of the polyhedral model where iterators can be represented using simple canonical induction variables. To do so, any `return` except the last, which exits the recursive loop, associated with a `call` to B (steps 20,21) causes $\texttt{i}_1$ to be incremented by 1.

Input:
- `event`: Branch event (same as in Algorithm 4.1 and Algorithm 4.2)
- `diiv`: Current dynamic IIV.

Output:
- Updated dynamic IIV

```
1  if event == C(B) or E_C(L,B):
2    diiv.innermost.CTX.push(L)
3  if event == TC(B) or E_TC(L,B):
4    diiv.innermost.CTX.last = B
5  if event == E(L,B) or E_C(L,B):
6    diiv.innermost.CTX.last = L
7    diiv.push_dimension(0, B)
8  if event == X(L,B) or X_R(L,B):
9    diiv.pop_dimension()
10   diiv.innermost.CTX.last = B
11 if event == I(L,B) or I_C(L,B) or I_R(L,B):
12   diiv.innermost.IV++
13   diiv.innermost.CTX.last = B
14 if event == R(B):
15   diiv.innermost.CTX.pop()
16   diiv.innermost.CTX.last = B
```

Algorithm 4.3 – Updating of the dynamic IIV (`diiv`)

As previously described, loop events allow updating the dynamic IIV, as detailed in Algorithm 4.3. The algorithm uses the following notation:

- `diiv` is a dynamic IIV where the rightmost is the innermost loop dimension. Each dimension contains two fields, the induction variable `IV`, followed by the context variable `CTX`: For example, $\texttt{I}(\texttt{L}_1, \texttt{A}_1)$ applied to a dynamic IIV $(\texttt{M}_0/\texttt{L}_1, \texttt{0}, \texttt{A}_2)$ increments the IV of $\texttt{L}_1$ by one and changes the `CTX` to $\texttt{A}_1$. Finally, we get $\texttt{diiv} = (\texttt{M}_0/\texttt{L}_1, \texttt{1}, \texttt{A}_1)$.
- `CTX.last` represents the last element of the context variable: So, if $\texttt{CTX} = \texttt{M}_0/\texttt{D}_0/\texttt{C}_0$, then $\texttt{CTX.last} = \texttt{C}_0$.
- `CTX.push`(B) appends B to end of the rightmost context: For example, $\texttt{C}(\texttt{C}_0)$ applied to $\texttt{diiv} = (\texttt{M}_1/\texttt{L}_1, \texttt{0}, \texttt{B}_1)$ leads to $\texttt{diiv} = (\texttt{M}_1/\texttt{L}_1, \texttt{0}, \texttt{B}_1/\texttt{C}_0)$.
- `CTX.pop()` does the reverse: That is, $\texttt{R}(M_1)$ applied to $\texttt{diiv} = (\texttt{M}_0/\texttt{D}_0)$ pops $\texttt{D}_0$ and updates the last element to $\texttt{M}_1$ leading to a `CTX` of $(\texttt{M}_1)$.

- `push_dimension`$(n, \mathtt{B})$, where n is an integer, and `B` a block, adds those two fields to the dynamic IIV: For example, $\mathtt{E_C}(L_1, \mathtt{B_0})$ applied to $(\mathtt{M_1})$, appends $\mathtt{L_1}$ to `CTX` and adds a dimension with its innermost context set to $\mathtt{B_0}$. Which in turn results in $\mathtt{diiv} = (\mathtt{M_1}/\mathtt{L_1}, \mathtt{0}, \mathtt{B_0})$.
- `pop_dimension()` does the reverse: That is, $\mathtt{X}(L_2, B_3)$ applied to $\mathtt{diiv} = (\mathtt{M_0}/\mathtt{L_1}, \mathtt{0}, \mathtt{A_1}/\mathtt{L_2}, \mathtt{1}, \mathtt{B_2})$ results in $\mathtt{diiv} = (\mathtt{M_0}/\mathtt{L_1}, \mathtt{0}, \mathtt{A_1}/\mathtt{B_3})$.

4.5 The Folding Algorithm

This section gives an overview of the folding algorithm which lies at the heart of MICKEY and then presents its components in detail. Before describing the core algorithm itself we first describe its inputs and outputs in Section 4.5.1 and Section 4.5.2 and give an intuition of how it is used Section 4.5.3. Section 4.5.4 introduces the main components of the algorithm and how they interact. This is followed by a detailed description of the folding algorithm including an analysis of its complexity in Section 4.5.5.

4.5.1 Inputs

The inputs for the folding-based analysis are provided by our front-end. To handle any kind of loops in a uniform way, our front-end inserts *canonical* iterators in every loop. These iterators start at zero and advance by one every iteration. The front-end is implemented as a plugin for the dynamic binary translator QEMU [17, 84]. Even though the front-end analyses machine code, it works at the level of the generic QEMU IR, making it CPU architecture agnostic.

The inputs of the folding algorithm are streams of two types, one for instructions and one for dependencies. In the following sections, a *static instruction* is a machine instruction in the program binary. An *instruction instance* is one dynamic execution of a static instruction. A dependency is a pair consisting of an instruction instance that produced a value and another instance consuming it. We call those instances the *source* and the *destination* respectively. Also, our front-end only captures dataflow dependencies, that is, read-after-write dependencies for which there are no intermediate write to the same memory location or register.

Each input stream has a unique *identifier ID*. An instruction stream is identified by a static instruction. A stream of data dependencies is identified by a pair of static instructions. We note this as *Static instruction source* $\rightarrow$ *Static instruction destination*. The two types of streams have the same overall structure where each entry consists of two elements:

- an *iteration vector* (IV): a vector made up of the current values of all canonical loop iterators;
- a *label*: the definition of the label differs between the two types of streams and is described below.

For a given stream, all the IVs span a multi-dimensional space where each entry is a point. Thus, in the following we use the terms *entry* and *point* interchangeably. Also, note that IVs arrive in the input stream in lexicographical order.

Finally, it is worth mentioning that the front-end tracks the calling context in which instructions execute and generates different input streams for different calls to the same function.

Instructions An instruction stream for a static instruction *ID* contains all its instances. The label is a scalar value whose meaning depends on the type of the static instruction. If the instruction is an arithmetic instruction dealing with integers, the label is the integer value representing the result computed by the instruction. If the instruction is a memory access the label is the address read or written by the instance. As described in the next section, these labels are used to identify induction variables and fixed-stride memory accesses.

```
for (j = 1; j <= n2)
  sum = 0.0;
  for (k = 0; k <= n1)
    tmp1 = load(&conn + k)          I1 - Memory access
    tmp2 = load(tmp1 + j)           I2 - Memory access
    tmp3 = load(&l1 + k)            I3 - Memory access
    sum = sum + tmp2 * tmp3         I4 - Computation
    k = k + 1                       I5 - Computation
  j = j + 1                         I6 - Computation
```

Figure 4.12 – C-like binary version for the code of Figure 4.1

ID=I1		ID=I2		ID=I3		ID=I4		ID=I5		ID=I6	
IV (cj,ck)	Label	IV (cj,ck)	Label	IV (cj,ck)	Label	IV (cj,ck)	Label	IV (cj,ck)	Label	IV (cj)	Label
(0,0)	12	(0,0)	67	(0,0)	407	(0,0)	N/A	(0,0)	1		
(0,1)	13	(0,1)	71	(0,1)	408	(0,1)	N/A	(0,1)	2		
...	...	...	...	...	...	...	...	...	...		
(0,42)	54	(0,42)	243	(0,42)	449	(0,42)	N/A	(0,42)	43	(0)	2
(1,0)	12	(1,0)	68	(1,0)	407	(1,0)	N/A	(1,0)	1		
(1,1)	13	(1,1)	72	(1,1)	408	(1,1)	N/A	(1,1)	2		
...	...	...	...	...	...	...	...	...	...	...	...

Table 4.1 – Instruction input streams from example in Figure 4.12 with n1 = 42

To illustrate the contents of the input stream of instruction instances we again use the example of backprop from Figure 4.1. At the binary level, the considered loop-nest contains several instructions that are represented in an abstract C-like fashion in Figure 4.12. An excerpt of the six instruction streams for this example is shown in Table 4.1. The IV of each entry is the vector made up of the current values of all canonical loop iterators noted cj and ck in the table.

Dependencies A dependency stream for a pair of static instructions contains an entry for each pair of instances for these instructions that have a data dependence. The IV of an entry is the IV of the destination, whereas the label is the IV of the source. Table 4.2 shows three of the six dependency input streams for the example in Figure 4.12. In this example, all the dependencies except I4 → I4 are intra-iteration dependencies.

I1 → I2		I2 → I4		I4 → I4	
IV (cj,ck)	Label (cj',ck')	IV (cj,ck)	Label (cj',ck')	IV (cj,ck)	Label (cj',ck')
(0,0)	(0,0)	(0,0)	(0,0)		
(0,1)	(0,1)	(0,1)	(0,1)	(0,1)	(0,0)
...	...	...	...	...	...

Table 4.2 – Three of the six dependency input streams from example in Figure 4.12

4.5.2 Outputs

The folding algorithm processes each stream independently. For each stream, the final result of folding is a piecewise linear function mapping IVs to labels. We refer to this piecewise linear function as a *label function*. In the following we use the terms domain and *geometry* interchangeably. Each piece of the domain of a label function is described by a set of affine inequalities, hence it defines a *polyhedron*. More precisely, a label function can be written as:

$$f : \mathbb{N}^d \mapsto \mathbb{Z} \mid f(c_1, \cdots, c_d) = \begin{cases} k_{0,1} + k_{1,1}c_1 + \cdots + k_{d,1}c_d & \text{if} \quad (c_1, \cdots, c_d) \in \text{polyhedron}_1 \\ k_{0,2} + k_{1,2}c_1 + \cdots + k_{d,2}c_d & \text{if} \quad (c_1, \cdots, c_d) \in \text{polyhedron}_2 \\ \cdots \end{cases}$$

$$\text{where} \;\; k_{i,j} \in \mathbb{Z} \cup \{\top\}$$

Section 4.2.2 already showed two examples of label functions, that is, $0j{+}1k{+}12$ and $1j{+}\top k{+}66$.

The domain of a label function contains exactly the IVs of all entries of the input stream. Moreover, when the label function is applied to an IV of its domain it produces the label value associated with that point in the input stream. That is, a label function is a compact representation of an input stream since it can describe arbitrarily many points in one piece. It also directly exposes regularity in a form that polyhedral optimizers can exploit.

The coefficients of a label function may be either an integer or $\top$ [2] , as illustrated for the non-affine memory access of `backprop` shown in Figure 4.3. If a coefficient is $\top$, this indicates that the evolution of the label cannot compactly be expressed as an affine function along the corresponding dimension.

When a label function does not contain any $\top$ coefficient, it can be used to precisely reconstruct the input stream it was created from. As mentioned before, the domain of a label function contains all IVs from the input and no other points. We can thus reconstruct the stream simply by applying the label function to every point of its domain.

Instructions For an instruction stream, depending on the type of its corresponding static instruction, the label function either represents the integer values computed by the instruction or the addresses it accesses. These label functions are then used to identify induction variables and fixed-stride memory accesses. Table 4.3 illustrates the outputs for the input streams in Table 4.1 where `n2` $= 16$ and `n1` $= 42$. All instruction instances of a given input stream are now described by a single line. We notice from this table that four of the six instructions have an affine function where all the coefficients are known, that is, they are not $\top$. The affine function of instruction I4 is marked as N/A because it is computing floating point values. Instruction I2 has an affine function with the coefficient for dimension k being $\top$, as already discussed. In this case, the labels of the input stream cannot be reconstructed from the IVs. Nevertheless, the algorithm still outputs the single polyhedron describing the domain for this instruction and produces useful information for a polyhedral optimizer. It is also worth mentioning that, unlike in this example, the label function of each instruction can be made up of several pieces if the domain of the instruction cannot be represented as a single convex polyhedron. In this case, the domain would be represented as a union of polyhedra.

Dependencies The label function of a dependency is a piecewise linear function with multiple outputs. The label function maps IVs of the consumer instances of the dependence to IVs of the producer instances. That is, given an instruction instance the label function can be used to determine from which other instruction instance it consumed data. Table 4.4 illustrates the result of the folding-based analysis for the three dependency input streams in Table 4.2. All the

[2] as described later, the folding algorithm internally also uses special $\bot$ values for coefficients, but these do not appear in the output

ID	Polyhedron (cj,ck)	Label function f(cj,ck)
I1	$0 \leq cj \leq 15, 0 \leq ck \leq 42$	$0cj + 1ck + 12$
I2	$0 \leq cj \leq 15, 0 \leq ck \leq 42$	$1cj + \top ck + 67$
I3	$0 \leq cj \leq 15, 0 \leq ck \leq 42$	$0cj + 1ck + 407$
I4	$0 \leq cj \leq 15, 0 \leq ck \leq 42$	N/A
I5	$0 \leq cj \leq 15, 0 \leq ck \leq 42$	$0cj + 1ck + 1$
I6	$0 \leq cj \leq 15$	$1cj + 2$

Table 4.3 – Output of the folding algorithm for the instructions stream of Table 4.1 with n2 = 16 and n1 = 42

ID	Polyhedron (cj,ck)	Label function f(cj,ck)
I1 → I2	$0 \leq cj \leq 15, 0 \leq ck \leq 42$	$cj' = cj + 0ck,\ ck' = 0cj + ck$
I2 → I4	$0 \leq cj \leq 15, 0 \leq ck \leq 42$	$cj' = cj + 0ck,\ ck' = 0cj + ck$
I4 → I4	$0 \leq cj \leq 15, 1 \leq ck \leq 42$	$cj' = cj + 0ck,\ ck' = 0cj + ck - 1$

Table 4.4 – Output of the folding algorithm for the dependencies stream shown in Table 4.2

dependencies of a given input stream are now described by a single line. Each one of these lines states when the dependency between two instruction instances occurs. For example, the last line tells us that the instance (cj, ck) of I4 depends on the instance $(cj, ck - 1)$ of itself. As for the output regarding instruction streams, it is worth noting that in this example the domain of all the dependencies is described by a single polyhedron. Nevertheless, in more complex cases these domains can be represented by a union of polyhedra.

4.5.3 Using the output

The output of the folding algorithm is intended to be consumed by the back-end of our tool chain leveraging a classic polyhedral optimizer. Such an optimizer requires as input the list of instructions along with their domains and their dependencies. The back-end then searches which re-scheduling transformations can be applied to the instructions under the constraints imposed by the data dependencies.

Before providing dependencies to the back-end, the output stream of dependencies is pruned by removing all the dependencies involving a computation instruction identified as an *induction variable*. An induction variable is a computation instruction with a label function where all coefficients of all pieces are integers, that is, not $\top$. The initial loop iterators are an example of induction variable, that is, I5 and I6. Removing those instructions serves two purposes. First, induction variables always depend on their value from the previous iteration of the loop they are in. Consequently their dependencies constrain the execution to be completely sequential. Removing these instructions gives the back-end more freedom and may uncover parallelism or potential for other polyhedral transformations. The second reason for removing induction variables is simply that it reduces the number of instructions the polyhedral back-end has to deal with.

Then, still before providing the dependencies to the optimizer, we must process dependencies having $\top$ coefficients in their label function. Observe that the fact that some dependencies are not accurately captured by our folding algorithm is not a limitation of the approach, but a choice imposed by polyhedral back-ends, the complexity of which is combinatorial with the size of the polyhedral representation. To that end, we over-approximate those dependencies by imposing a lexicographical ordering over their IVs for the iterators having at least one $\top$

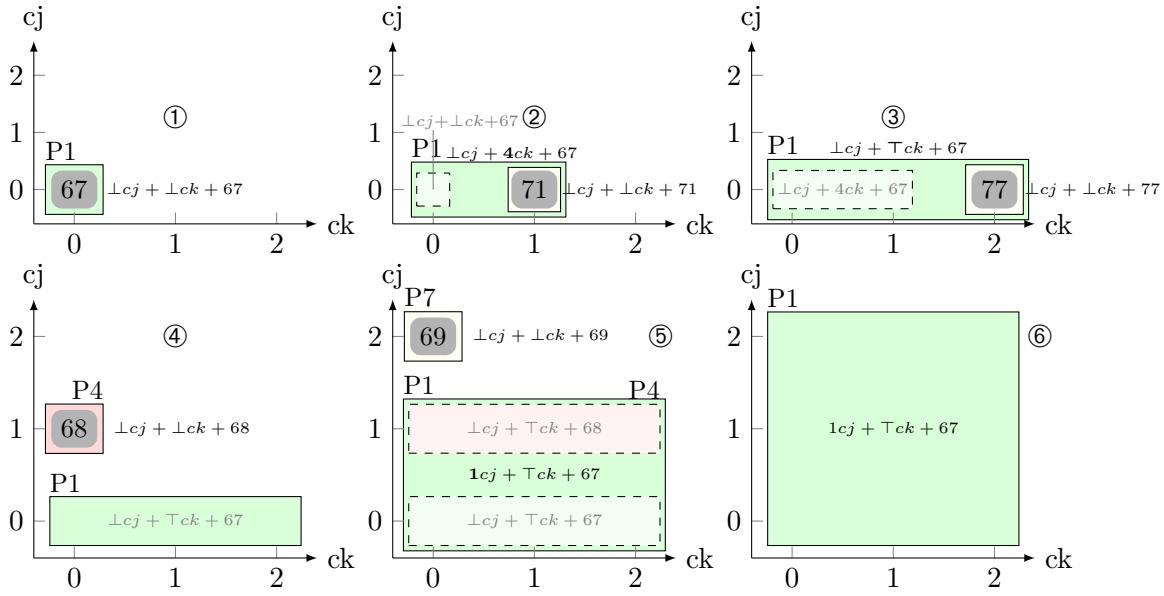


Figure 4.13 – Folding process for the input stream of I2 in Table 4.1 considering only three points in both dimensions.

coefficient. With this order, it is guaranteed that all instances of the producer come before any instances of the consumer that might possibly consume them. For instance, let us assume in our running example that the dependency I4 $\rightarrow$ I4 is not $cj' = cj + 0ck$, $ck' = 0cj + ck - 1$ but $cj' = cj + 0ck$, $ck' = 0cj + \top ck$: The over-approximated dependency given to the back-end would be $cj' = cj \wedge ck' \leq ck$.

Finally, the access functions for memory instructions are also given to the polyhedral optimizer so that it can identify opportunities for exposing vectorization and spatial locality. For this it needs information about stride which is given by a non-$\top$ coefficient in the label function of an instruction accessing memory.

4.5.4 Overview

As stated in the previous section, the folding algorithm processes the stream for each identifier separately. It is worth mentioning that exactly the same algorithm is used for both instruction and dependency streams. This algorithm receives points in a geometrical space as specified by the IVs. The main idea of the algorithm is to construct polyhedra from those points. For each polyhedron the algorithm also constructs an affine function describing the label of the points contained in the polyhedron. When receiving the first point, the algorithm creates a 0-dimensional polyhedron containing only that point. It then tries to grow this polyhedron with the next points, adding dimensions as necessary.

To give an intuition about how the folding algorithm works, let us consider the stream of I2 in Table 4.1.

4.5.4.1 Geometric folding

The folding process for I2 is illustrated in Figure 4.13. For now we will ignore the construction of the affine function. As shown, the process leads to the creation of many intermediary polyhedra which are merged as the algorithm executes. The polyhedron P1, a 3×3 square, is the final result of the algorithm. As shown in Figure 4.13 the main steps of the algorithm are as follows:

① create the 0-dimensional polyhedron P1 when the first point $(cj = 0, ck = 0)$ is received;

② when $(cj = 0, ck = 1)$ is received, P1 absorbs it to become a 1-dimensional polyhedron, that is, a line segment;
③ when $(cj = 0, ck = 2)$ is received, P1 absorbs it;
④ notice that the loop over ck is completed when point $(cj = 1, ck = 0)$ is received because the iterator of the surrounding loop cj increased. Then create the new 0-dimensional polyhedron P4;
- P4 absorbs $(cj = 1, ck = 1)$ to become a 1-dimensional polyhedron and then absorbs $(cj = 1, ck = 2)$ (not shown in Figure 4.13);

⑤ notice that the loop over ck is completed when point $(cj = 2, ck = 0)$ is received. P1 absorbs P4 along dimension cj. Then create the new 0-dimensional polyhedron P7;
- P7 absorbs $(cj = 2, ck = 1)$ to become a 1-dimensional polyhedron and then absorbs $(cj = 2, ck = 2)$ (not shown in Figure 4.13);

⑥ P1 absorbs P7 and becomes the final 3×3 square.

The geometric folding works exactly the same for dependencies as illustrated above for instructions. The only difference is the semantic of the reconstructed union of polyhedra. In the case of an instruction, this union defines when the instruction is executed. For a dependency it tells when the dependency occurs from the point of view of the destination.

4.5.4.2 Label folding

In the previous section we ignored the folding of the labels associated with each point in the input stream. Nevertheless, this label folding takes place at the same time as geometric folding. It is also performed in a streaming fashion. In the context of label folding, the symbol $\perp$ denotes a coefficient that has not yet been determined because the loop has not yet iterated along the dimension associated with that coefficient. As shown in Figure 4.13 the label folding proceeds as follows:

① create $f1(cj, ck) = \perp cj + \perp ck + 67$ when point $(cj = 0, ck = 0)$ with label 67 is received;
② update $f1$ to $\perp cj + \mathbf{4}ck + 67$ when P1 absorbs $(cj = 0, ck = 1)$ with label 71 is received because ck advanced by 1 and $71 - 67 = 4$;
③ check if $f1(cj, ck) = \perp cj + 4ck + 67$ is valid when P1 absorbs $(cj = 0, ck = 2)$ with label 77. It is not the case, so update $f1$ to $\perp cj + \top ck + 67$;
- repeat the steps above for P4 and get $f4(cj, ck) = \perp cj + \top ck + 68$ (not shown in Figure 4.13);

⑤ update $f1$ to $f1(cj, ck) = \mathbf{1}cj + \top ck + 67$ when P1 absorbs P4 because cj advanced by 1 and $68 - 67 = 1$;
⑥ check whether $f1(cj, ck) = 1cj + \top ck + 67$ is compatible with $f7(cj, ck) = \perp cj + \top ck + 69$, when P7 absorbs P1 to get the final 3×3 square. It is the case.

When the folding algorithm finishes all remaining $\perp$ coefficients can safely be set to zero. The intuition behind this is that at the end of the folding process a $\perp$ coefficient signals that the loop for this dimension only iterated once, that is, it never influenced the label value.

The algorithm that folds the labels of a dependency is the same as the one described above for the labels of an instruction. It is just applied *individually* for each scalar value in the label vector, that is, each component of the IV of the source of the dependency.

4.5.5 The algorithm

This section introduces the structure of the main algorithm itself and then explains its sub-components.

4.5.5.1 Main folding function

The main function is shown in Algorithm 4.4. As explained in Section 4.5, this main function, is applied to each input stream separately. To handle real-life applications, where input streams are huge, the algorithm works in a streaming fashion (line 10). It is not necessary to have the whole input available at once. The output is also emitted as a stream. The main principle of the algorithm, as depicted in the example in Figure 4.13, consists of maintaining a worklist of *intermediate* polyhedra per dimension. The intermediate polyhedra then grow by absorbing other polyhedra. Note that a d-dimensional polyhedron can only absorb $(d-1)$-dimensional polyhedra.

Elementary Polyhedra The absorption process, explained in Section 4.5.5.2, is restricted to only produce a sub-class of convex polyhedra that we call *elementary polyhedra*. Because the folding algorithm only produces elementary polyhedra, the term polyhedra implicitly refers to elementary polyhedra in the following. A d-dimensional elementary polyhedron is a convex polyhedron with 2^d vertices and a restricted shape. The shape restriction is motivated by complexity concerns for the absorption process as explained in Section 4.5.6.

We define elementary polyhedra in a D-dimensional space using the following recursive definition:

- an elementary 0-dimensional polyhedron is a polyhedron made of a single point;
- an elementary d-dimensional polyhedron is a convex polyhedron with 2^d extreme points such that:
  1. all its extreme points must have identical coordinates in dimensions higher than d. In other words, the polyhedron is flat on dimensions between $d+1$ and D;
  2. it has two $(d-1)$-faces flat on dimension d but with different coordinates for that dth dimension. The face with the lower coordinates in d is called the *lower face* and the one with the higher coordinates is called the *upper face* ;
  3. its lower and upper faces must themselves be $(d-1)$-elementary polyhedra;
  4. the edges connecting the lower and upper faces can be expressed as $k\vec{S}$. Where $k \in \mathbb{N}^*$ and $\vec{S}$, the *slope vector* of the edge, is a vector where all components are either -1, 0, or $+1$.

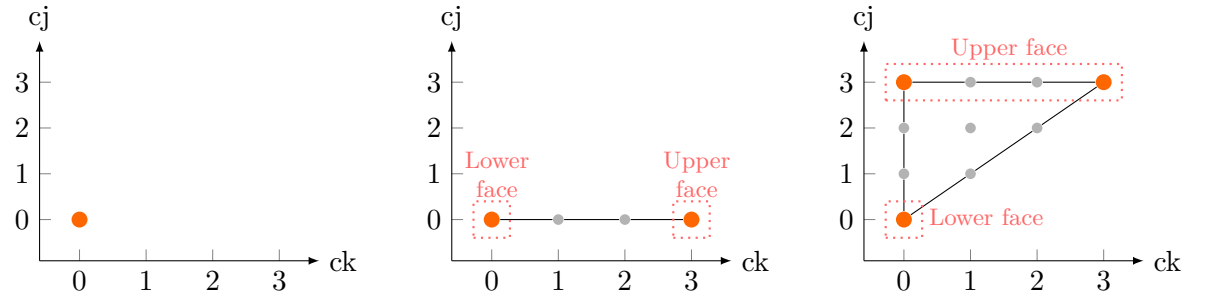
All faces of an elementary polyhedron beside the upper and lower face are called *side faces*.

More informally, an elementary 0-dimensional polyhedron is a polyhedron made of a single point. An elementary 1-dimensional polyhedron is an interval. An elementary 2-dimensional polyhedron is a trapezoid. An elementary 3-dimensional polyhedron is a trapezoidal prism. Every general polyhedron can be represented using unions of elementary polyhedra, meaning any iteration or dependence space can be described with them. The more regular a space is the fewer elementary polyhedra are necessary to represent it.

A polyhedron is *degenerate* on a given dimension if all its vertices have the same coordinate for that dimension, that is, it has zero width in that dimension. The elementary polyhedra produced by the folding algorithm may be degenerate on one or more dimensions.

Figure 4.14 shows examples of elementary polyhedra in a 2-dimensional space (D=2). The vertices of the polyhedra are shown as large dots. The other integer points included in the polyhedra are shown with small dots. Note that even though the lower face in Figure 4.14c is degenerate it is still represented using two vertices, but they have the same coordinates.

Producing only elementary polyhedra as described above allows controlling the worst case complexity of the absorption process described in Section 4.5.5.2. The choice of producing only such polyhedra is also motivated by the nature of the input streams that we want to process. The front-end we use to feed the folding algorithm always produces canonical IVs starting at zero and only ever advancing by one. Hence, elementary polyhedra are able to represent the



(a) 0-dimensional polyhedron (d=0) (b) 1-dimensional polyhedron (d=1) (c) 2-dimensional polyhedron (d=2) with degenerate lower face

Figure 4.14 – Examples of elementary polyhedra in a 2-dimensional space (D=2) with its vertices in orange

iteration space of most of the loops fitting the polyhedral model.

Data structures Folding works on spaces with a fixed number of dimensions D, that is, the dimensionality of the corresponding IVs. The state of the folding algorithm is stored using two dictionaries. The first one, `absorbers` (line 2), contains a list of intermediate polyhedra for each dimension. `absorbers[d]` only contains d-dimensional, potentially degenerate, polyhedra. The polyhedra in `absorbers[d]` are those that can still grow along dimension d by absorbing $(d-1)$-dimensional polyhedra. Those $(d-1)$-dimensional polyhedra are stored in `vertices_to_be_absorbed[d]` (line 5). The keys of the dictionary `vertices_to_be_absorbed[d]` are the lexicographically smallest vertices of the polyhedra to be absorbed. This point, which we name the *anchor*, is used to uniquely identify the absorbed polyhedron. `abso.upper_left` (line 26), the lexicographically smallest point of the upper face of `abso`, also called its *corner*, is the vertex from which the absorption is performed. For example, in Figure 4.13, in the absorption just before step 6, the anchor of P7 is $(cj = 2, ck = 0)$ and `upper_left` of P1 is $(cj =, ck = 0)$.

Steps of the algorithm When a point is received, the algorithm first processes the innermost dimension (numbered 1). Then, for each loop (but for the outermost one) that completes in the instrumented code, the algorithm processes its enclosing dimension. In other words, if no loop finishes the algorithm processes only dimension $d = 1$; if the innermost loop finishes it processes dimensions $d = 1$ and $d = 2$; if its enclosing loop finishes it processes dimensions $d = 1$, $d = 2$ and $d = 3$; etc. In line 17, `process_dims` represents that set of dimensions to be processed.

Before processing the different dimensions, the current point is added into `absorbers[0]` (line 14). This state is only transient, because as soon as the innermost dimension is processed, the point will be promoted into `vertices_to_be_absorbed[1]` (line 20). Then, for each dimension d of `process_dims` (processed from inner to outer), three steps are performed.

The first step (line 19 to 20) promotes all polyhedra in `absorbers[d-1]` into `vertices_to_be_absorbed[d]`. Because dimensions are processed in increasing order, that is, from innermost to outermost, when processing dimension d we are sure that `absorbers[d-1]` have already absorbed all the $(d-2)$-dimensional polyhedra it could. This promotion to d-dimensional degenerate polyhedra allows them to be absorbed in the next step by the d-dimensional polyhedra already in `absorbers[d]`.

In the second step (line 23 to 38), polyhedra from `absorbers[d]` try to absorb polyhedra in `vertices_to_be_absorbed[d]`. For absorption to be possible, the polyhedra should be geometrically compatible (line 30) and their label functions should match (line 29) as described in Section 4.5.5.1 and Section 4.5.5.2. If a polyhedron in `absorbers[d]` does not absorb any

```
1  # Per dimension list of absorber polyhedra.
2  absorbers: <int, poly_list_t>
3
4  # Per dimension dictionary mapping vertices to polyhedra to be absorbed
5  vertices_to_be_absorbed: <int, <point_t, poly_t>>
6
7  # While we have points
8  while True:
9    # End of stream?
10   point = wait_next_point()
11   if point == end_of_stream: break
12
13   # Put current point in absorbers[0]
14   absorbers[0].insert(new Polyhedron(point))
15
16   # for each dimension d such that d=1 or loop d-1 completed
17   for d in process_dims(point):
18     # Step 1: promote absorbers[d-1] -> vertices_to_be_absorbed[d]
19     for p in absorbers[d-1]:
20       p.move(absorbers[d-1], vertices_to_be_absorbed[d])
21
22     # Step 2: absorbers[d] try to absorb vertices_to_be_absorbed[d]
23     for abso in absorbers[d]:
24       absorbed = False
25       for v in abso.search_vectors:
26         corner     = abso.upper_left
27         to_be_abso = vertices_to_be_absorbed[d][corner + v]
28         if to_be_abso != None:
29           if has_compat_label(abso, to_be_abso, d) and
30              has_compat_geometry(abso, to_be_abso, d):
31             update_geometry(abso, to_be_abso, d)
32             update_label(abso.label_function, to.label_function)
33             absorbed = True
34             break
35
36       if not absorbed:
37         # abso will never absorb anyone along d, so promote it to the next dimension
38         abso.move(absorbers[d], vertices_to_be_absorbed[d+1])
39
40     # Step 3: promote all of remaining vertices_to_be_absorbed[d] -> absorbers[d]
41     for not_abs in vertices_to_be_absorbed[d].values:
42       not_abs.move(vertices_to_be_absorbed[d], absorbers[d])
43
44 # Stream finished, flush all pending polyhedra
45 flush_pending_polyhedra()
```

Algorithm 4.4 – The main folding algorithm

other polyhedron, then it will never grow again along dimension d. As a consequence, it is promoted into `vertices_to_be_absorbed[d+1]` (line 38). This promotion also transforms the d-dimensional polyhedron into a $(d+1)$-dimensional degenerate polyhedron.

The third and last step (line 41 to 42) promotes all the d-dimensional polyhedra in `vertices_to_be_absorbed[d]` that have not been absorbed. Since those polyhedra will never be absorbed again in dimension d, they are moved to the `absorbers[d]` list so that they will have a chance to themselves absorb other polyhedra next time dimension d is processed.

During the execution of the algorithm, a polyhedron is *retired*, that is it is emitted to the

output stream, when it is promoted to the dimension above the dimension of the space, that is, 3 for an instruction in a 2D loop nest. When the stream is finished, all remaining non-retired polyhedra are also retired. Retired polyhedra are written to the output stream and do not consume memory anymore. This is safe since we know that they will never grow anymore.

4.5.5.2 Absorption

As stated in the previous section, the second step of the folding algorithm grows polyhedra by letting them absorb each other. A d-dimensional polyhedron searches for candidates to absorb by checking if its corner touches the anchor of any other $(d-1)$-dimensional polyhedron (Algorithm 4.4, line 26). This search is performed by adding the *search vectors* `v` to the coordinates of the corner and performing a lookup in `vertices_to_be_absorbed[d]` to see if there is a polyhedron at this position (line 27). Once a candidate has been found, the algorithm must check that the absorption is possible (line 30), that is, leads to an elementary polyhedron (`has_compat_geometry`) with a correct label function (`has_compat_label`). Which search vectors are used for the lookup and how geometric compatibility is checked depends on whether the absorber is degenerate in d or not. If the absorber is degenerate we call this a *polyhedra merge*. An example of this is when P1 absorbs P4 in Figure 4.13. The second case, a *polyhedra extension*, occurs when the absorber is not degenerate, as seen for example when P1 absorbs P7. The `has_compat_geometry` function called once a candidate has been found is shown in Algorithm 4.5.

```
1  # Geometry compatibility check
2  def has_compat_geometry(abso, to_be_abso, d):
3    # Polyhedra merge case
4    if abso.is_degenerate_on(d):
5      for k in [0, 2^(d - 1)[ :
6        diff = to_be_abso.vertices[k] - abso.vertices[k]
7        if not diff.is_a_search_vector(d):
8          return False
9      for side_face in side_faces_of_merged_polyhedron(abso, to_be_abso, d):
10       if not all_points_lie_on_same_hyperplane(d, side_face):
11         return False
12
13   # Polyhedra extension case
14   else:
15     for k in [0, 2^(d - 1)[ :
16       v = abso.growing_directions[k]
17       if abso.vertices[k] + v != to_be_abso.vertices[k]:
18         return False
19   return True
```

Algorithm 4.5 – The `has_compat_geometry` function which ensures that polyhedra resulting from absorption is still elementary polyhedra

Polyhedra merge In this case, the d-dimensional absorber polyhedron is degenerate on dimension d. Hence, it has no edges yet along that dimension. As a consequence there are many possibilities where to look for the anchor of the to-be-absorbed polyhedron. Our algorithm uses the set of all possible 3^{d-1} search vectors written as $v = (0, \ldots, 0, 1, \delta_{d-1}, \ldots, \delta_1)$ where for $i < d$, $\delta_i \in \{-1, 0, 1\}$.

Once a candidate polyhedron has been found, the `has_compat_geometry` function call verifies that concatenating the vertices of the two polyhedra leads to a well formed polyhedron (Algorithm 4.5, line 4 to 11). First, the function checks (line 4 to 8) that all the corresponding vertices

of both polyhedra are connected through the search vectors used to find the anchor of the to-be-absorbed polyhedron described just above. Second, the function checks (line 9 to line 11) that all the side faces of the polyhedron resulting from the absorption are valid. As shown in Figure 4.15, even if the lower (a square) and the upper (a triangle) faces are valid elementary polyhedra, the result of the absorption may not be a valid polyhedron. For this we check if all the points of the side face lie on the same hyperplane (line 11). To begin with, we arbitrarily designate one point of the face as the origin of the plane. We then pick $d-1$ other points to calculate a normal vector n for the plane by calculating the nullspace of the space spanned by the vectors from the origin to the other points. And finally, we verify for all remaining points p that the dot product $(origin - p) \cdot n$ equals zero.

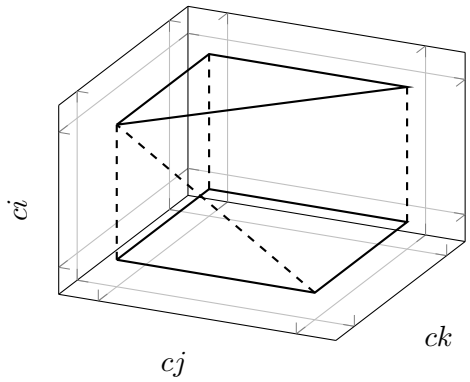


Figure 4.15 – Example of invalid polyhedron after absorption

If absorption is performed, the resulting polyhedron will no longer be degenerate in d. By construction, if well formed, the so obtained polyhedron is necessarily an elementary d-dimensional polyhedron. Its lower face will be the original absorbing polyhedron while its upper face will be the the absorbed polyhedron.

Polyhedra extension In this case the absorber is a non-degenerate d-dimensional polyhedron. Hence, the absorber already has edges along dimension d. When looking for candidates to absorb, there is only one search vector, the edge connecting the oldest vertex of the lower face to that of the upper face. To check if the absorption is legal, `has_compat_geometry` simply verifies (line 14 to line 18) whether the vertices of the two polyhedra can be connected using the existing edges of the absorber stored in the `growing_directions` list.

4.4.5.3 Compatibility and update of label functions

The absorption is performed only if both geometric and label compatibility are satisfied (line 30). This section describes how label functions are represented, created and combined together.

Label functions The data structure used for label functions is shown in Figure 4.16. `num_dimensions` is the number of loops enclosing the static instruction or the destination instruction associated with the input stream.

Creation Label functions are created when a new polyhedron is created from a single point (line 14). At this time, all the coefficients of the function are still unknown. Their types in the `coeff_types` array are set to $\perp$. The coordinates of the point used to create the new polyhedron are saved in the `initial_point` array. The first cell of this array is never used but still kept to make accesses more readable, that is, `initial_point[d]` contains the d^{th} coordinate.

```
enum Coefficient_Type:
  ⊥,     # coefficient undetermined
  INT,   # coefficient determined
  ⊤      # coefficient widened

struct Label_Function:
  num_dimensions: int
  init_point:     int[num_dimensions + 1]
  coeffs:         int[num_dimensions + 1]
  coeff_types:    Coefficient_Type[num_dimensions + 1]
```

Figure 4.16 – The data structure used to represent label functions

These coordinates are used when coefficients are updated. Note that once a coefficient has been updated from an unknown to a known value, it is never updated again except to be set to $\top$. The life cycle of a coefficient is then the following one:

$$\bot \rightarrow \mathbb{Z} \rightarrow \top$$

At creation time, `coeff[0]` is given the value of the label associated with the initial point. As long as there are some $\bot$ coefficients, `coeff[0]` contains the remaining amount contributed by unknown coefficients. We refer to `coeff[0]` as the *remaining value* in the following. This remaining value is updated whenever a coefficient is updated. When all coefficients are known, the remaining value represents the constant coefficient of the affine function.

The two polyhedra involved in a compatibility check along dimension d may be degenerate on one or more dimensions, including the d^{th} one. As a consequence, the check may be faced with affine functions where some coefficients are $\bot$. In the following, we note the label function of the absorbing polyhedron as `f_abso`, and that of the polyhedron to be absorbed as `f_to_be_abso`. We notice that the polyhedron to be absorbed is always degenerate on dimension d, as stated in Section 4.5.5.1. Hence, `f_to_be_abso.coeff_types[d]` $= \bot$.

All dimensions below are known For illustrative purposes we first cover the simplified case where all dimensions below d are known for the two label functions. The compatibility check for this simple case is shown in Algorithm 4.6. First the function `has_compat_label` verifies that all coefficients for dimensions from 1 to $d-1$ are the same. If this is not the case the two label functions are incompatible (line 6).

Otherwise, the check may be faced with two cases corresponding to the two different absorption cases described in Section 4.5.5.2. In the polyhedra merge case, where the absorber polyhedron is degenerate on dimension d, that is, `f_abso.coeff_types[d]` $= \bot$, the check always succeeds and the `has_compat_label` function returns `True` (line 10). Indeed, by setting the proper coefficient for dimension d and by updating the remaining value, it is always possible to make the two functions compatible as shown by the `update_label` function in Algorithm 4.6. The new coefficient is equal to the difference of remaining values (line 22). Note that, in general we would also have to divide the new coefficient by the progress made along dimension d. However, because absorption guarantees that the two polyhedra whose label functions are being merged touch each other the progress is always equal to 1. Finally, the remaining value is decreased by the effective contribution of the new coefficient taking into account the d^{th} coordinate of the initial point (line 28).

In the polyhedra extension case, the absorber polyhedron is not degenerate on dimension d. Its affine function already has a value computed for the coefficient on dimension d. Then

```
1  # Compatibility check when all dimensions below d are known
2  def has_compat_label(f_abso, f_to_be_abso, d):
3    # Verifies coefficients below d are the same
4    for q in range [1, d-1]:
5      if f_abso.coeffs[q] != f_to_be_abso.coeffs[q]:
6        return False
7
8    # Polyhedra merge case
9    if f_abso.coeffs[d] == ⊥:
10     return True
11
12   # Polyhedra extension case
13   else:
14     new_coeff_contrib = f_abso.coeffs[d] * f_to_be_abso.coeffs[d]
15     new_remain = f_to_be_abso.coeffs[0] - new_coeff_contrib
16     return new_remain == f_abso.coeffs[0]
17
18 # Update coefficient for dimension d and remaining value of f_abso.
19 # No need to update f_to_be_abso because it will be thrown after absorption
20 def update_label(f_abso, f_to_be_abso, d):
21   # Update of coefficient
22   new_coeff = f_to_be_abso.coeffs[0] - f_abso.coeffs[0]
23   f_abso.coeffs[d] = new_coeff
24   f_abso.coeff_types[d] = INT
25
26   # Update of remaining value
27   new_coeff_contrib = new_coeff * f_abso.init_point[d]
28   f_abso.coeffs[0] = f_abso.coeffs[0] - new_coeff_contrib
```

Algorithm 4.6 – Simplified version of the compatibility check and update of coefficient for the case when all dimensions below d are known. See Algorithm 4.7 and Algorithm 4.8 for the general case of `has_compat_label` and `update_label`.

`f_{abs}.coeff_types[d]` $\neq \perp$ and nothing needs to be updated. The compatibility check must only ensure that this coefficient is compatible with `f_to_be_abso`. This is done by first computing the contribution of the known coefficient of `f_abso` into `f_to_be_abso` using the initial point of `f_to_be_abso` (line 14). Then, the check subtracts this contribution from the remaining value of `f_to_be_abso` to compute its new remaining value. For the check to return `true`, this new remaining value must be equal to the remaining value of `f_abso` (line 16).

General case In the general case the two polyhedra may be degenerate for some dimensions below d. This happens if a dimension below d only iterates once. The compatibility check described above must take this into account.

Function `has_compat_label` in Algorithm 4.7 shows the general compatibility check between two polyhedra. The check is performed on all the matching pairs of label functions of the two polyhedra. Remember that there are several such label functions in the case of dependencies, one for each dimension of the source instruction.

The check works by comparing the coefficients of both functions for all the dimensions from 1 to d. If both coefficients for a dimension are known they must be the same or the check fails (line 15). If one is known and not the other (line 20 and line 24), then the function increments the total contribution coming from the other function for the function having the unknown coefficient. At the end of the loop (line 33), the check ensures that the coefficient for dimension d in `f_abso` is compatible with `f_to_be_abso`. This check relies on the total contribution variables

```
1  # General compatibility check
2  def has_compat_label(abso, to_be_abso, d):
3    # Loop over all pairs of label functions
4    for f_abso, f_to_be_abso in abso.label_functions, to_be_abso.label_functions:
5      # Contributions from other functions
6      abso_diff       = 0
7      to_be_abso_diff = 0
8
9      # Loop over all coefficients
10     for q in [1, d]:
11       abso_t       = f_abso.coeff_types[q]
12       to_be_abso_t = f_to_be_abso.coeff_types[q]
13
14       # Both coefficients already determined, must be the same
15       if abso_t != ⊥ and to_be_abso_t != ⊥:
16         if f_abso.coeffs[q] != f_to_be_abso.coeffs[q]:
17           return False
18
19       # One coefficient is not known, the other is
20       if abso_t == ⊥ and to_be_abso_t != ⊥:
21         abso_diff = abso_diff - f_abso.init_point[q] * f_to_be_abso.coeffs[q]
22
23       # One coefficient is known, the other is not
24       if abso_t != ⊥ and to_be_abso_t == ⊥:
25         to_be_abso_diff = to_be_abso_diff - f_abso.coeffs[q] * f_to_be_abso.init_point[q]
26
27       # Both coefficients are unknown, can be ignored
28       if abso_t == ⊥ and to_be_abso_t == ⊥:
29         continue
30
31   if f_abso.coeff_types[d] == ⊥:
32     return True
33   if f_abso.coeffs[0] + abso_diff == f_to_be_abso.coeffs[0] + to_be_abso_diff:
34     return True
35   return False
```

Algorithm 4.7 – General compatibility check for label functions

incremented during the loop to ensure that the two functions still produce the same value after merging.

In case they are compatible, the new coefficients, that is, the one on dimension d and potentially others, and the new remaining value for the function of the absorber are computed by the same principles as in the simplified case from Algorithm 4.6. The code for the general case of the `updated_label` function that determines coefficients is shown in Algorithm 4.8.

Label widening As shown by the `backprop` example, the folding algorithm must be capable of identifying labels that are affine on some dimensions and not on others. To that end, the algorithm has a mechanism called *label widening* enabling it to skip the matching of labels on a per dimension basis. If the compatibility check between two coefficients fails, then instead of returning `False` (line 17 in Algorithm 4.7), the coefficient is set to $\top$ and `True` is returned instead. The absorption can still happen, even if the labels of the two polyhedra are not fully compatible. The label function of the resulting polyhedron is no longer a fully accurate representation of the input stream. Nevertheless, this mechanism allows the folding algorithm to handle real life applications without a perfect affine behaviour. The name label widening stems from the fact

```
# Update coefficient for all dimensions <= d and remaining value of f_abso.
# No need to update f_to_be_abso because it will be thrown after absorption
def update_label(abso, to_be_abso, d):
  # Loop over all pairs of label functions
  for f_abso, f_to_be_abso in abso.label_functions, to_be_abso.label_functions:
    abso_diff       = 0
    to_be_abso_diff = 0

    for q in range(1, q + 1):
      abso_t       = f_abso.coeff_types[q]
      to_be_abso_t = f_to_be_abso.coeff_types[q]

      # One coefficient is not known, the other is
      if abso_t == ⊥ and to_be_abso_t != ⊥:
        abso_diff = abso_diff - f_abso.init_point[q] * f_to_be_abso.coeffs[q]
        set_coefficient(f_abso, f_to_be_abso.coeffs[q], q)

      # One coefficient is known, the other is not
      if abso_t != ⊥ and to_be_abso_t == ⊥:
        to_be_abso_diff = to_be_abso_diff - f_abso.coeffs[q] * f_to_be_abso.init_point[q]

      # Both coefficients are unknown, can be ignored
      if abso_t == ⊥ and to_be_abso_t == ⊥:
        continue

  if f_abso.coeff_types[d] == ⊥:
    new_coeff = to_be_abso_diff - f_to_be_abso.coeffs[0] - f_abso.coeffs[0]
    set_coefficient(f_abso, new_coeff, q)

def set_coefficient(fn, coeff_val: int, q: Dimension):
  delta = -coeff_val * self.init_point[q]
  fn.coeff_types[q] = INT
  fn.coeffs[q]      = coeff_val
  fn.coeffs[0]      = fn.coeffs[0] + delta
```

Algorithm 4.8 – General case for determining coefficients of label functions

that in the case of dependencies it widens the label functions from equalities to inequalities, as shown in Section 4.5.3.

The integration of this feature into Algorithm 4.7 is straightforward. A $\top$ coefficient is compatible with any other coefficient, and when performing absorption, any such coefficient in one of the two label functions leads to a $\top$ coefficient in the updated function.

The label widening mechanism is crucial for the label functions of instructions because $\top$ is a clear indicator that a memory access is not affine along a dimension. For dependencies it simply reduces the size of the output given to the back-end by reducing the number of produced pieces.

4.5.5.4 Geometric give up

Even with the label widening mechanism described above, some applications may lead to the creation of a huge number of polyhedra. This happens when the geometry of instructions and dependencies are not affine. It occurs for statements surrounded by *if* conditionals in the program. In the worst case, the folding algorithm creates one polyhedron for each dynamic instruction and for each dynamic dependency.

To mitigate this issue, the folding algorithm has another global option called *geometric give-up*. This options allows defining an upper limit on the number of intermediate polyhedra. Remember that an intermediate polyhedron is a polyhedron in one of the worklists that can still grow by absorbing other polyhedra. Before creating a new polyhedron (line 14), the algorithm checks if the number of intermediate polyhedra exceeds the threshold. If so, the associated input stream is marked as *give up*. Once a stream has been marked as give up all intermediate polyhedra for that stream are discarded. The discarded polyhedra are then replaced by hyper-rectangle that starts at the origin and extends to the maximum coordinate seen in the IVs of any point contained in the discarded polyhedra. In other words, the geometry of the input stream is over-approximated by a single large polyhedron. From then on every time a new point is received for the given up stream, the folding algorithm previously described is skipped. Instead, only the maximum coordinates of the hyper-rectangle are updated as necessary for every point. Lastly, all coefficients for all outputs of the label function for this input stream are set to $\top$, that is, a geometric give up implies giving up on all dimensions of the label function.

Similar to the widening of label functions once geometric give up has occurred it is no longer possible to reproduce the original input stream. However, the over-approximated geometry is guaranteed to contain all points seen in the input.

4.5.6 Complexity analysis

Let us first recall the main idea of our folding algorithm. The folding process starts with polyhedra of dimensionality zero, one for each point. Then, absorption is performed dimension by dimension from innermost to outermost. As the process advances, the dimensionality of the polyhedra involved grows. It turns out that the complexity of an absorption also grows with its dimensionality. But as the dimensionality increases, the number of absorptions, that is the number of intermediate polyhedra, also decreases. The more regularity, the more the number of intermediate polyhedra decreases. In other words, as formalised below, but for fully irregular programs for which neither label widening nor geometric give-up have been enabled, one should expect an overall complexity linear in the number of input points.

More formally, in a D-dimensional space, we denote note the total number of input points as N and the overall number of intermediate polyhedra seen in the `absorbers[d]` list when iterating over it as $N_d, \forall 1 \leq d \leq D$ (Algorithm 4.4, line 23). For a given $d \leq D$ we have:

- the number of total iterations of the `for` loop over `absorbers[d]` (line 23) is N_d;
- for each absorber, there are at most 3^{d-1} look-ups to find a polyhedron to absorb (line 25);
- testing if absorption is possible with regards to the label criterion (`has_compat_label` line 30) has cost $O(d)$;
- testing if absorption is possible with regard to the geometry criterion (`has_compat_geometry` line 30) has cost of $O\left(d \times 2^{d-1} + (2 \times (d-1)) \times \left(d^3 + d \times (2^{d-1} - d)\right)\right) = O\left(d^2 \times 2^{d-1}\right)$. Here, the first $d \times 2^{d-1}$ corresponds to checking the search vectors or growing directions. The factor $(2 \times (d-1))$ comes from the loop over the side faces of the merged polyhedron (line 9). The term d^3 corresponds to calculating the normal vector of the hyperplane of the sideface. And the final $d \times (2^{d-1} - d)$ corresponds to checking if the remaining points of the side lie on the same hyperplane.

This leads to an overall complexity of:

$$O\left(\sum_{d=1}^{D} N_d \times 3^{d-1} \times d \times 2^{d-1}\right) = O\left(\sum_{d=1}^{D} N_d \times 6^{d-1} \times d\right)$$

To illustrate the notations, let us assume a perfectly nested loop of depth D and size $N = n_D \times \cdots \times n_2 \times n_1$. Let also consider the scenarios where either the loop nest is fully regular

or geometrically regular only and label widening is enabled. In these two cases, the folding algorithm leads to a single polyhedron. We have, $N = n_1 \times n_2 \times \cdots \times n_D$, $N_1 = N$, $N_2 = N/n_1$, $N_3 = N/(n_1 \times n_2)$, ..., $N_D = n_d$. The overall complexity is then:

$$O\left(N + \sum_{d=2}^{D} \frac{N \times 6^{d-1} \times d}{\prod_{j=1}^{d-1} n_j}\right) = O\left(N + N \times \sum_{d=2}^{D} d \times \prod_{j=1}^{d-1} \frac{6}{n_j}\right)$$

$$= O\left(N + N \times \sum_{d=2}^{D} \prod_{j=1}^{d-1} \frac{j+1}{j} \times \frac{6}{n_j}\right)$$

Observe that in practice we will almost always have $\forall 1 \leq j < D$, $n_j \geq \frac{j+1}{j} \times 6$, which leads to a complexity of $O(N)$.

Obviously, N_d being always bounded by N, we have a worst case complexity of $O\left(N \times D \times 6^D\right)$. This worst case scenario will occur for fully irregular input streams where every absorption fails even with label widening. That is, where absorption failures are caused by geometric incompatibility. Geometric give-up allows the algorithm to handle these input streams with a linear complexity.

4.6 Polyhedral Feedback

An essential motivation for folding DDGs into polyhedral structures is to enable the use of advanced polyhedral compilation systems, which are capable of finding a schedule that maximizes parallelism and finds tiling opportunities [26].

4.6.1 Polyhedral compilation of folded-DDGs

Typically, a polyhedral compiler is applied to small numerical kernels made of a handful of statements [77, 26, 143, 163]. Polyhedral schedulers suffer scalability challenge for larger programs [143] since their complexity typically grows exponentially with the number of statements in the input program. Our DDG folding and over-approximation techniques allow going from programs with thousands of statements, vastly exceeding the typical program scale these schedulers can handle, to only a few hundreds.

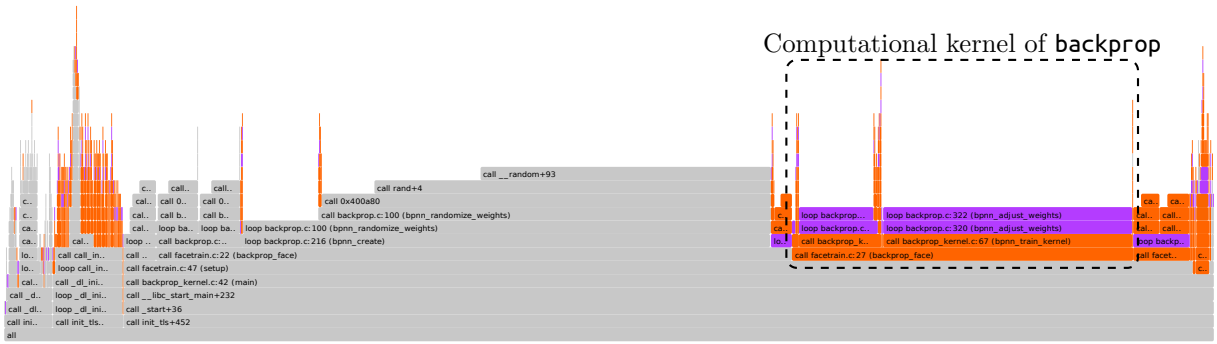
Numerous customizations for scalability of the polyhedral compiler have been implemented, ranging from constraining the space of legal schedules to accelerate the scheduling process to approximation of the code generation process to quickly output a decorated simplified AST describing the program structure after transformation. For example, the presence of large integer constants causes combinatorial blow up in the ILP solver used to solve the scheduling problem [158]. We implemented a parameterization of iteration domains, to replace those constants by a parameter, an unknown, but constant integer value. That is, a domain $\{[i] : 0 <= i < 1024\}$ is replaced by $[n] \rightarrow \{[i] : 0 <= i < n \wedge n \geq 1024\}$ prior to scheduling. We control the number of parameters introduced by reusing the same parameter for a range of values. That is, if the value $x \in [1024 - s, 1024 + s]$, for some arbitrary number $s \in \mathbb{Z}$, usually set to $s = 20$, then we replace x by $n + (x - 1024)$.

The reader may refer to the available implementation in PoCC [162] for further details about the simplifications implemented, computation of profitability metrics is implemented in the PolyFeat module.

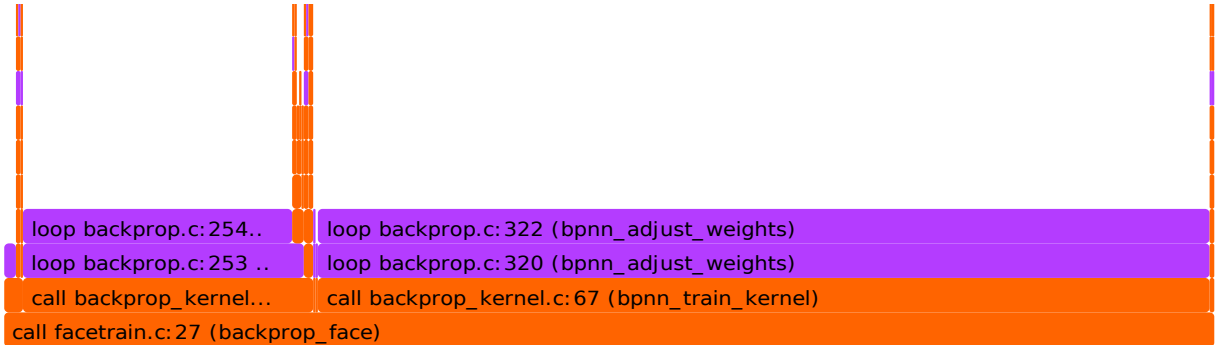
4.6.2 User feedback

The final output of MICKEY is a report that details which regions of a program can benefit from polyhedral optimization. Here we display the feedback produced for the `backprop` benchmark. The region of interest corresponds the function `facetrain`, which, transitively, contains two calls each of the computational kernels `bpnn_layerforward` and `bpnn_adjust_weights`. Both kernels function contain a doubly-nested loop. In this case, MICKEY will suggest that fusion and tiling is possible, even across the two calls. I.e., as if the function had been inlined.

The actual textual output produced by MICKEY is currently relatively undeveloped. It is mostly based on annotating the disassembled machine code of the input program and thus still quite hard to use for non-experts.



(a) Annotated flame-graph for entire `backprop` program. Also includes the parts of libc called by `backprop`. Irregular parts of the program are greyed out. The parts that are potentially interesting for polyhedral optimization are coloured. The width of nodes represents the number of instructions executed in it.



(b) Zoomed in flame-graph for the main computational kernel in `backprop`. Function calls in orange, loops in purple

Figure 4.17 – Annotated flame-graph for `backprop`.

4.6.2.1 Statistics on the region of interest

The main user feedback produced by MICKEY is the dynamic schedule-tree described in Section 4.4.3 along with an AST that shows the structure of the input program after the application of the suggested transformations. This AST embeds various metrics on instruction count, loop properties such parallelism and tilability, and the list of statements in every loop. This lets the user estimate the effort required to manually rewrite the code to apply the suggested transformations.

To better visualize the various program regions, this information can be rendered as a flame-graph[79]. This flame-graph is a hierarchic representation of the structure of a program. That is, a tree where each node represents one region of code, either a loop or a function. The children of a node are the functions called from that code region or the loops contained in it. In this visualization the width of a node is proportional to its estimated computational hotness, determined by the number of instructions it executed. Colours and gradients are used to encode the estimated potential for optimization detected by MICKEY. Non-interesting regions, for example, can be greyed out. These flame-graphs are SVG files, which can be clicked and zoomed to obtain detailed information on each region. An example flame-graph, created from `backprop`, can be seen in Figure 4.17.

We also display explicit statistics for each region, breaking down by instruction type, as shown in Figure 4.18. The statistics are computed automatically from the folded DDG by reasoning on the number of points in each iteration domain/dependence, and the associated assembly instructions.

Statistics on the instruction mix are useful in particular to estimate the operational intensity as the ratio of load/store operations per total non-memory operations. Besides MICKEY furthermore reports how many instructions each different loop level contains to highlight the potential impact of transformations across deep loop nests. We also compute the values for these metrics assuming the proposed transformation had been implemented. This is intended to assist the user in determining the potential benefit of a transformation.

```
Number of iteration domains: 69
Number of dependence polyhedra: 76
Totals: 6816353 operations, 6816434 operation-to-operation dependences
Total operations: 6816353
Total FLOPs: 3670321 (53.85%)
Total load/store: 3145965 (46.15%)
Total INT: 50 (0.00%)
Total others: 17 (0.00%)
Maximal loop depth: 2
  -> depth=0:  0 instructions (0.00%) 0 operations (0.00%)
  -> depth=1: 40 instructions (57.97%) 340 operations (0.00%)
  -> depth=2: 29 instructions (42.03%) 6816013 operations (100.00%)
```

Figure 4.18 – Statistics on the fat region of interest

4.6.2.2 Transformations feedback: simplified AST after transformation

A key feedback provided to the user is a simplified AST that displays the code structure after applying the suggested transformation. Figure 4.19 shows an excerpt of this AST.

This AST obeys the following principles:

1. All statistics on the number of operations and percentages shown are cumulative, i.e., they represent the total number of operations dominated by a node. Similarly, the statement list is also cumulative, i.e., it captures all statements under a node, wherever they are located in the sub-tree.
2. Loops in the output program are always named `i′`, `j′`, `k′`, `l`, l', ...
3. A loop can have the following attributes: `sequential`, or `parallel`. The `tilable` qualifier can appear for both `parallel` and `sequential` loops. A `tilable` loop is necessarily permutable with any other `tilable` loop in the same loop nest. A `sequential/tilable` loop necessarily

```
...
|= AST=(0,0,i')
|-- Loop i' => parallel/tilable loop, 2097203 operations (30.77%)
    +-- alignment: 3 stride-0 (524322) 2 stride-1 (1048576) 0 stride-N (0)
|-- contains: S64 S67 S62 S65 S57 S54 S55
|-- Stmts full names:
                + 4011800000000101_4011bf(c0,c1); // LOAD
                + 4011800000000101_4011cb(c0,c1); // LOAD
                + 4011800000000101_4011ac(c0,c1); // LOAD
                + 4011800000000101_4011c3(c0,c1); // FLOAT
                + 4011800000000001_4011cb(c0,c1); // LOAD
                + 4011800000000001_4011bf(c0,c1); // LOAD
                + 4011800000000001_4011c3(c0,c1); // FLOAT

     |= AST=(0,0,i',j')
     |-- Loop j' => parallel/tilable loop, 2097203 operations (30.77%)
         +-- alignment: 1 stride-0 (524288) 0 stride-1 (0) 4 stride-N (1048610)
     |-- contains: S64 S67 S62 S65 S57 S54 S55
     |-- Stmts full names:
                     + 4011800000000101_4011bf(c0,c1); // LOAD
                     + 4011800000000101_4011cb(c0,c1); // LOAD
                     + 4011800000000101_4011ac(c0,c1); // LOAD
                     + 4011800000000101_4011c3(c0,c1); // FLOAT
                     + 4011800000000001_4011cb(c0,c1); // LOAD
                     + 4011800000000001_4011bf(c0,c1); // LOAD
                     + 4011800000000001_4011c3(c0,c1); // FLOAT

|= AST=(0,6,i')
|-- Loop i' => parallel/tilable loop, 1048576 operations (15.38%)
    +-- alignment: 2 stride-0 (1048576) 0 stride-1 (0) 0 stride-N (0)
|-- contains: S3 S4
...
```

Figure 4.19 – Excerpt of simplified AST produced for `backprop`

only has forward dependencies. A `sequential`, but not `tilable`, loop may or may not have only forward dependencies, i.e., be permutable with other loops. A `parallel` is always sinkable to any lower loop level, including the last one, independent of any tilability.

4. The alignment information for memory accesses is populated only if the input DDG has information about access functions. If the label function describing the memory accesses has only $\top$ coefficients then 0 will be printed everywhere.
5. The alignment information displayed for a loop shows how often different memory access strides occur in that loop if it where transformed to be an inner-most loop. This gives the user an indication of the profitability of loop interchanges. For example for the purpose of vectorization as has been shown in the `backprop` example. Note, however that it is not always possible to sink a loop to be inner-most. This is only possible for loops marked as parallel.
6. The statements in the "`Stmt full names`" list appear in their prefix order in the AST. That is, the order in which they will syntactically occur in the transformed code.
7. The Kelly's mapping style iteration vector `|= AST=(0,i',j')` displayed for each node visualizes its position in the AST. The first component represents the root of the schedule

tree and is always 0. This is then followed by an arbitrary sequence of constant numbers and loop iterators. Nodes are displayed in the order of a prefix traversal of the schedule tree, i.e., by increasing lexicographic order.

4.6.2.3 Transformations feedback: transformed pseudo-code

Technically, Mickey does not only determine the existence of a loop transformation sequence to implement parallelism and locality: it also generates the code structure that is the result of this transformation. This representation of the transformed program can be pretty-printed as pseudo-code. The C-like pseudo code is intended to help users with implementing more complicated loop transformations.

Figure 4.20 shows an example of this pseudo-code for a part of `backprop`, where a tiling of size 32x32 was found and implemented. Each statement in this pseudo-code corresponds to one machine instruction of the original program. We also attach annotations to loops and statements to show about the properties of each loop. For example if a loop is parallel or not. The user can inspect this code to understand which statements should be grouped together under the same loop nest to implement fusion.

```
...
parallel for (jTile = 0; jTile <= floor(32767/32); jTile++) {
  for (i = 0; i <= 15; i++) {
    parallel for (j = (32 * jTile); j <= min(32767, ((32 * jTile) + 31)); j++) {
      4012280000000101_401249(i, j); /* @LOAD@ */;
      4012280000000101_40125e(i, j); /* @FLOAT@ */;
      4012280000000101_401265(i, j); /* @FLOAT@ */;
      4012280000000101_401275(i, j); /* @FLOAT@ */;
      4012280000000101_40127d(i, j); /* @FLOAT@ */;
      4012280000000101_401289(i, j); /* @STORE@ */;
      4012280000000101_401281(i, j); /* @FLOAT@ */;
      4012280000000101_401285(i, j); /* @STORE@ */;
    }
  }
}
...
```

Figure 4.20 – Excerpt of transformed pseudo-code produced for `backprop`. Each statement consists of a loop-call context tree node ID, `4012280000000101`, and an instruction PC. For example `401249`.

Statement names, e.g., `4012280000000101_401249`, are composed of the ID of a context tree ID, `4012280000000101`, and an instruction PC, `401249`. We also emit separate context includes details about all ASM instructions, such as the original source file and line for this instructions maps to. Note that displaying the ASM instruction and not only its type is possible, but not shown here for simplicity.

The user can inspect this pseudo-code at a high-level, only looking at the loop structure and the attributes of the loops. Or they can “zoom in” to understand which statements should be grouped together, how complex the loop bounds may need to be, what range they take, and so on.

4.6.2.4 Transformations feedback: statistics

Finally, Mickey also produces several summary statistics for every program region. This helps the user to quickly assess the potential for some key performance-impacting optimizations, such as parallelization and vectorization. An example of this is shown in Figure 4.21. All information shown here is cumulative. For example, SIMD-ready operations counts the total number of operations that are dominated by an inner-most loop which is parallel. Note that the number of fat fused components is the total number of distinct outer-most loops that dominate more than 5% of the total operations, plus the fraction of total operations that they cumulatively dominate.

```
...
[Stats] Total number of OpenMP-ready operations: 6816209 (100.00% total ops)
[Stats] Total number of OpenMP-ready operations in non-inner-most loops: 6816013
↪  (100.00% total ops)
[Stats] Total number of OpenMP-ready operations in outer loops only:  6816209
↪  (100.00% total ops)
[Stats] Total number of SIMD-ready operations: 6816209 (100.00% total ops)
[Stats] Total number of stride 0/1 load/stores in SIMD-ready loops: 1572930 (50.00%
↪  total load/store ops)
[Stats] Maximal number of possible stride 0/1 load/stores after some loop
↪  permutation: 3145845 (100.00% total load/store ops)
[Stats] Total number of (max. 2D) tilable operations: 6816353 (100.00% total ops)
[Stats] Total number fat (>= 5% ops) fused components: 4 (99.99% total ops)
...
```

Figure 4.21 – Excerpt of summary statistics produced for `backprop`

4.7 Experimental Results

In this section we demonstrate what kind of user feedback Mickey provides on a number of case studies and the Rodinia benchmark suite [42, 43], and how this feedback can be used in practice. For this we have run three sets of experiments.

1. We performed a number of case studies where we ran Mickey on a benchmark, implemented the optimizations it suggested and measured the obtained speedup (Section 4.7.1).
2. We ran Polly, a static polyhehdral compiler, over the Rodinia benchmarks to see what transformations it performs (Section 4.7.2).
3. We applied Mickey to the Rodinia benchmarks to see what transformations it suggests and how well the folding algorithm compresses execution traces (Section 4.7.3 and Section 4.7.4).

For our experiments we used the OpenMP version of the most recent version Rodinia, 3.1. It consists of 20 benchmarks written in C and C++ benchmarks that are explicitly parallelized with OpenMP. We did not use the benchmark `mummergpu` since it is partially written in CUDA and our tool only supports CPU code.

As Mickey's front-end currently does multi-threaded programs all benchmarks where run in a single thread. Most Rodinia benchmarks did not require modifying the source code to execute in one thread, it suffices to set the environment variable `OMP_NUM_THREADS` to one when running them. However, some benchmarks contain calls to `omp_set_num_threads`. In these cases we changed the code or command line arguments, as necessary, so that `omp_set_num_threads` is

only ever called with an argument of 1.

All Rodinia benchmarks and case studies where compiled using GCC 8.1.1. The version of the front-end used for these experiments uses QEMU 3.1. Since this version of QEMU cannot handle AVX instructions we used the compiler flags `-g -O2 -msse3`. For speedup measurements programs where compiled using the Intel ICC and IFORT compilers (version 18.0.3, flags `-Ofast -march=native -mtune=native`). The measurements where taken on a machine with a Xeon Ivy Bridge CPU with two 6 core CPUs (24 hyperthreads total), each running at 2.1 GHz (GFlop/s are averaged over 50 runs).

All benchmarks where compiled for and run on X86 CPUs. Since many X86 instructions both read or write memory and perform computations at the same time the instructions streams that form the input of the folding algorithm are actually more complicated than the ones presented in Section 4.5.1 and Table 4.1. In reality the label of an instruction can have multiple values to account both for the addresses accessed and the values produced. The label functions for instructions thus potentially have multiple outputs as well, just like those for dependencies.

4.7.1 Case studies

This section aims to present case studies that illustrate how the feedback provided by MICKEY can concretely be used to speed up programs.

4.7.1.1 Case study I

This case study illustrates a simple feedback MICKEY can provide to users. It shows how MICKEY pinpoints the absence of dependencies along some existing orthogonal outer loop dimensions, which enables coarse-grain parallelism via OpenMP parallel pragmas.

We selected `alexnet` for this study, a deep convolutional neural network used for image classification. We exploit the ability of MICKEY to discover loop structures, to report typical runtime values to evaluate the profitability of the transformation, and to handle interprocedural analysis in a robust way.

The version of `alexnet` we used [230] has five sequential convolutional layers each implemented by a call to a function called `convolution`. Each call to `convolution` uses a different input size, weights and convolutional kernel size. Furthermore, three of these five layers are split in two, implemented by calling `convolution` twice, each time with only half of the input data. As reported in Table 4.5, MICKEY detects that the three outermost loops inside of `convolution` are parallel. It also detects that the two calls to `convolution` for the split layers can be performed in parallel.

We obtained the reported speedup by adding a single `omp pragma parallel for` on the outermost loop in `convolution`. To exploit the nested parallelism of the second layer we use a double nested parallel region. Applying these transformations increases the performance from 1 GFlop/s to 10.9 GFlop/s for the first layer and from 3.8 GFlop/s to 5.6 GFlop/s for the second. Note that convolution is a heavily bandwidth-bound problem and performance already tops out when using 12 of the 24 hyperthreads.

4.7.1.2 Case study II

This case study again illustrates a relatively simple feedback from MICKEY. Here MICKEY finds that dependencies live within the first quartant [14]. This allows exposing data locality and fine-grain parallelism through a loop interchange.

For this study, we selected `backprop`, a supervised learning method used to train artificial neural networks extracted from the Rodinia benchmark suite [43]. In addition to what was outlined in the previous case study, MICKEY exploits its ability to associate an iteration vector with

Code region	%ops	omp parallel loops	speedup
conv1 @ alexnet.cc:70	15%	layers.cc:{227,229,230}	7.6 x
$conv2_1$ @ alexnet.cc:119 $conv2_2$ @ alexnet.cc:120	29%	alexnet.cc:{119} layers.cc:{220,222,224}	1.5 x

Table 4.5 – Results of the alexnet case study. Reported line numbers are from debug information. **%ops** gives percentage of total number of instructions in program executed in a region. Suggested parallel loops are represented using a set of line numbers. I.e. layers.cc:{227,229,230} is a 3D loop nest.

each memory access, and to match pointer values with scalar evolution. Since the scalar evolution expressions for a memory access describe the addresses it reads/writes, they can be used to detect strided accesses along any dimension, which in turn can be used to detect vectorization potential.

The flame-graph provided by MICKEY is reported in Figure 4.17 where the regions of interest are: 1. the first call (of two) to bpnn_layerfoward with a constant value of n2 = 16 ; 2. the last call (of two) of bpnn_adjust_weights with a constant value of ndelta = 16. Both functions contain a 2D-nested-loop, called $L_{\texttt{layer}}$ and $L_{\texttt{adjust}}$ respectively. For both loop nests MICKEY suggests an interchange followed by vectorization. For example, see $L_{\texttt{layer}}$ on Table 4.6 for which the two loop dimensions are reported to be on lines 249 (outer) and 253 (inner) of file backprop.c: The loop nest is fully permutable, that is, interchange is possible. Also, only the outermost loop is parallel, and there are more stride 0/1 accesses along the outermost dimension (100%) than along the innermost (50%).

To enable the suggested transformations, one needs to specialize the two interesting function calls and array expand the scalar sum. The other calls to bpnn_layerfoward and bpnn_adjust_weights are left unchanged, since they (a) take up much less of the overall program runtime (b) have different arguments and do not profit from the transformation. Applying the transformation improved, in our case, bpnn_layerfoward from 0.5 GFlop/s to 2.8 GFlop/s and bpnn_adjust_weights from 0.3 GFlop/s to 5.1 GFlop/s.

Code region	%ops	loop	parallel	permutable	%stride 0/1	speedup
backprop_kernel.c:52	14%	backprop.c:249	yes	yes	100%	5.3 x
($L_{\texttt{layer}}$)		backprop.c:253	no	yes	50%	
backprop_kernel.c:57	46%	backprop.c:321	yes	yes	100%	7.8 x
($L_{\texttt{adjust}}$)		backprop.c:322	yes	yes	50%	

Table 4.6 – Results of the backprop case study. Reported line numbers are from debug information. The two loop nests, backprop.c:{249,253} and backprop.c:{321,322}, both profit from a loop interchange since the outer loop has more stride 0/1 accesses.

4.7.1.3 Case study III

This case study illustrates an example of advanced feedback. By providing exact dependence vector "directions" MICKEY shows that data locality and coarse-grain parallelism can be exposed through loop skewing and tiling.

For this study, we selected GemsFDTD, a finite difference time domain method from the SPEC CPU2006 benchmark suite [92] written in Fortran90. Analysing Fortran code is not a problem for MICKEY as it works at the binary level. However, the compiler we used (gfortran-8.1.1) messes up the debug information, making it necessary for the user to shift the line numbers for the provided code references by hand. This case study fully exploits MICKEY's ability to

model (and compress in a polyhedral form) the data dependencies, instead of simply checking their existence/absence. This knowledge about the structure of dependencies allows MICKEY to check for tiling opportunities. First, MICKEY detects that four functions from the benchmark execute a large amount of the program's total number of dynamic instructions. These functions are `updateH_homo`, `updateE_homo`, `UPML_updateH`, and `UPML_updateE`; Inside the first two of those functions are the five hottest loop nests, so we focus on them. As reported in Figure 4.7, MICKEY annotates all five loops as fully parallel and tilable. So, to obtain a speedup we tile each loop along all dimensions with a tile size of 32 and mark the outermost loop parallel with an `OMP PARALLEL DO` directive. Recall that tiled code can always be also coarse-grain parallelized using wavefront parallelism, as exploited by the Pluto polyhedral scheduler [26]. Tiling and parallelising the loops increased performance in `updateE_homo` from 1.3 GFlop/s to 2.7 GFlop/s and `updateH_homo` from 1.3 GFlop/s to 3.7 GFlop/s

Code region	op	tilable loop nests	speedup
`update.F90:106`	20%	`update.F90:106,107,121`	2.6 x
`update.F90:240`	18%	`update.F90:240,241,244`	1.9 x

Table 4.7 – Results of the `GemsFDTD` case study. Reported line numbers are from debug information (shifted to account for problem in `gfortran`).

4.7.2 Static polyhedral compilers and Rodinia

Benchmark	Reasons	Benchmark	Reasons	Benchmark	Reasons
`backprop`	A	`kmeans`	IFA	`particlefilter`	CF
`bfs`	BF	`lavaMD`	BF	`pathfinder`	BP
`b+tree`	BF	`leukocyte`	ICBFAP	`srad_v1`	IF
`cfd`	F	`lud`	BF	`srad_v2`	IF
`heartwall`	ICBF	`myocyte`	CBA	`streamcluster`	ICBFAP
`hotspot`	B	`nn`	IF		
`hotspot3D`	BF	`nw`	IF		

F. non-affine memory access function
A. unhandled pointer aliasing
P. base pointer not loop invariant
I. unhandled call to an external function
C. complex CFG
B. non-affine loop bound/conditional

Table 4.8 – Reasons why Polly fails to analyse Rodinia benchmarks

To assess how well static approaches fare with the Rodinia benchmark suite we ran the static LLVM-based [124] polyhedral compiler Polly [81] over the entire suite. This is similar to the studies done in [40, 191]. We chose Polly over other, more sophisticated polyhedral optimizers like PoCC [162], PPCG [218] or Pluto [25] since it can parse the source code of the Rodinia benchmarks as is. PoCC, PPCG and Pluto only support a small subset of C, no C++ at all, and require the user to mark code regions that should be analysed with pragmas. As a consequence neither of them can work on Rodinia without extensive rewriting of the benchmark sources. Polly, on the other hand, works at the level of LLVM IR and not at the source code level. LLVM can easily parse the any of the Rodinia benchmarks without any problem. However, since LLVM IR is quite low-level compared to C/C++ source code Polly has to reconstruct a lot of information in order to build a polyhedral representation of a program.

For this experiment we used Polly version 7.0.1 and the flags -O3 -ffast-math -polly-process-unprofitable. Kernels that span multiple functions were inlined to allow Polly to see the same code region as Mickey. Calls to external functions from the libc or the OpenMP runtime were not inlined. Where such calls are present this usually results in Polly being unable to analyse the kernel, though since we use -ffast-math it can handle calls to simple functions such as `exp` or `sqrt`. Programs where compiled without -fopenmp so that the compiler does not outline loop nests into separate functions.

Despite all of this Polly was unable to build a polyhedral model of the whole region of interest for any of the benchmarks in Rodinia. While it was able to model some smaller subregions, 1D or 2D loop nests, in most benchmarks, in nearly all cases its own profitability metric decided not to optimize them. Two notable exceptions are the `heartwall` and `lud` benchmarks. In `heartwall` Polly was able to model a sequence of nine 2D loop nests that accounts for roughly two thirds of the code in the body of the kernel. For `lud` it managed to model the whole inner 3D loop nest of the kernel, but not the outermost loop. Finally, in all benchmarks the inability to model the outermost loops blocks Polly from exploiting the thread level parallelism inherent in the Rodinia benchmark suite.

A summary of the reason why Polly failed is shown in Table 4.8. The most common reason for which Polly is unable to analyse a benchmark are calls to external functions. It should be noted that the majority of these calls are calls to the OpenMP runtime or logging code and could be removed without substantially altering the semantics of the benchmarks. Only a few benchmarks, such as `heartwall` an `nn`, interleave calls to external functions that read the benchmarks input with the actual computation kernel. The second most common problem are non-affine loop bounds or conditionals. While there are benchmarks like `streamcluster` that contain loops with dynamic data-dependent loop bounds many of these cases could be rewritten to fit the polyhedral model. Polly can generate runtime alias checks to solve the aliasing problems in different benchmarks, this is however not turned on by default since this increases code size and has a possible runtime overhead.

In conclusion, most benchmark kernels in Rodinia can be modified to be amenable to polyhedral analysis. However, this would require a non-negligible amount of upfront effort by the user without any guarantee that the existing static polyhedral optimizers could find any optimizing transformations.

4.7.3 Optimization feedback on Rodinia

The goal of this section is to demonstrate that Mickey can be systematically applied on a full benchmark suite, and find potential for optimization. Since the output of Mickey for each benchmark is extensive here we only illustrate the application of Mickey on Rodinia using aggregate metrics. These metrics are not meant to be used as is by the end user. Instead, the user is expected to work on one benchmark at a time, and navigate the feedback we provide.

Summary statistics

Table 4.9 presents summary statistics about the Rodinia 3.1 (CPU only) benchmarking suite, that we computed/aggregated by processing the feedback from Mickey each benchmark. The benchmarks where run on a reduced data set that exercises the same control paths as the full data set. This was done since the polyhedral optimizer back-end currently exhausts memory for larger DDGs.

Column %Aff reports the percentage of dynamic operations that are part of a fully affine region without over-approximation. The low proportion of affine code reported for `heartwall`, `hotspot`, and `lud` is the consequence of not supporting lattices at folding time: These programs

Bench.	#V	#E	%Aff	Region	%ops	%Mops	%FPops	interproc.	skew	%||ops	%simdops	%reuse	%Preuse	ld-src	ld-bin	TileD	%Tilops	C	Comp.	fusion
backprop	10 M	10 M	85%	facetrain.c:25	67%	30%	76%	Y	N	100%	100%	50%	100%	2D	2D	2D	100%	6	4	S
bfs	2 M	1 M	21%	bfs.cpp:137	55%	66%	18%	N	N	100%	100%	1%	1%	3D	3D	2D	100%	1	1	M
b+tree	23 M	24 M	49%	main.c:2345	26%	99%	0%	N	N	100%	100%	44%	44%	3D	3D	3D	100%	15	4	S
cfd	251 M	372 M	98%	*3d_cpu.cpp:480	98%	42%	99%	Y	N	100%	61%	18%	42%	5D	4D	3D	100%	1	3	S
heartwall	4 G	8 G	1%	main.c:536	99%	38%	56%	Y	N	100%	100%	0%	0%	7D	6D	5D	100%	1	3	S
hotspot	11 M	16 M	0%	*_openmp.cpp:318	81%	35%	89%	Y	Y	100%	100%	3%	3%	4D	4D	2D	100%	1	1	S
hotspot3D	210 M	256 M	99%	3D.c:261	49%	28%	81%	N	N	100%	99%	11%	11%	4D	4D	3D	100%	1	1	M
kmeans	513 M	647 M	97%	*_clustering.c:160	97%	56%	98%	Y	N	100%	100%	46%	53%	4D	4D	4D	100%	1	3	S
lavaMD	879 M	1 G	0%	kernel_cpu.c:123	99%	69%	92%	N	N	100%	100%	0%	0%	4D	4D	3D	100%	1	2	S
leukocyte	4 G	9 G	39%	detect_main.c:51	37%	64%	64%	Y	N	100%	100%	63%	63%	4D	4D	3D	100%	11	5	S
lud	42 M	71 M	4%	lud.c:121	99%	44%	70%	Y	N	99%	98%	0%	1%	5D	5D	3D	99%	3	3	S
myocyte	1 M	866 K	89%	main.c:283	99%	80%	80%	Y	N	100%	99%	47%	47%	4D	3D	1D	99%	1	3	S
nn	1 M	2 M	1%	nn_openmp.c:119	31%	42%	71%	Y	N	100%	0%	0%	0%	1D	1D	1D	100%	1	1	M
nw	80 M	93 M	99%	needle.cpp:308	79%	73%	27%	Y	Y	100%	100%	77%	77%	4D	4D	2D	100%	2	2	S
particlefilter	628 M	678 M	27%	*_seq.c:593	99%	5%	14%	N	N	99%	100%	55%	55%	3D	3D	2D	100%	22	2	S
pathfinder	62 M	48 M	67%	pathfinder.cpp:99	31%	83%	16%	N	Y	100%	0%	0%	40%	2D	2D	2D	100%	1	1	M
srad_v1	1 G	2 G	99%	main.c:241	99%	31%	93%	Y	N	99%	100%	18%	18%	3D	3D	2D	100%	1	1	S
srad_v2	600 M	864 M	98%	srad.cpp:114	96%	31%	92%	Y	N	100%	100%	14%	14%	3D	3D	2D	100%	1	1	S
streamcluster	779 M	1 G	97%	*_omp.cpp:1269	99%	6%	13%	Y	-	-	-	-	-	6D	6D	-	-	52	-	-

Table 4.9 – Summary statistics computed from MICKEY's feedback on the Rodinia benchmark suite. Columns are explained in Section 4.7.3

contain hand linearised nested loops whose bounds use modulo expressions and so are not recognized as fully affine. Note that, even when parts of a benchmark are not affine, we can still find affine over-approximations for those regions, and potentially find transformations for the program as a whole.

Based on the statistics provided by MICKEY, the biggest region for which the optimizer suggests a transformation has been selected by hand. The line number of the outermost loop or call to kernel function is given in column Region. We considered a region to be interprocedural (column *interproc.*) if inlining was required to perform the transformation or if it contained a call to libc or the OpenMP runtime. Column %ops reports the percentage of dynamic operations of the program executed inside the region, while %Mops and %FPops reports the percentage of memory (respectively floating point) operations of the region itself. Note that the sum of %Mops and %FPops can be greater than 100% since on X86 a single instruction can both load and store to/from memory and perform an operation.

The next group of metrics shows what can be achieved via semantics-preserving structured transformations. skew displays whether skewing is used in the proposed transformation, we tend to avoid skewing unless it really provides improvements in parallelism and tilability. %||ops gives the percentage of dynamic operations that can be parallelized using OpenMP parallel pragmas. If a non-inner loop dimension is detected as parallel, then all its operations are considered to be parallelisable. As a loop has at least two iterations, at least two parallel blocks can be exposed when a loop is reported parallel. Similarly, %simdops reports the percentage of operations that occur in parallel innermost loops.

The %reuse/%Preuse metrics report space locality that is available in the program: %reuse is the percentage of load/stores that are stride-0 or stride-1 in the existing innermost loops in the program, while %Preuse reports the maximal percentage of load/store operations that can be made stride-0 or stride-1 via a sequence of loop permutations.

We report the maximal loop depth of the region in the source code (*ld-src*) and in the binary code (*ld-bin*). This shows whether the compiler performed any transformation that modifies the loop depth (e.g., full loop unrolling for `cfd`). Next the maximal tiling depth (*TileD*) is reported, along with %Tilops, the percentage of operations that can be tiled.

As soon as a region can be tiled, coarse-grain (wavefront) parallelism is possible, and data reuse could be improved. MICKEY does not currently provide feedback on temporal locality potential, but as illustrated in the `backprop` case study %reuse allows to evaluate spatial locality improvements through tiling/interchange.

Finally, the metrics C/Comp./fusion outline the complexity of the loop fusion/distribution structure that originates from the structured transformation proposed, and is an indication of the difficulty to manually implement a transformation. Any outermost loop with more than 5% of the total region operation counts as one "component". For example, if the region is made of two consecutive loop nests executing each half of the operations, then 2 components will be reported. C reports the number of components in the binary code; Comp. the number of components after applying the proposed structured transformation, using the fusion heuristic reported in fusion (M for maximal loop fusion, and S for *smartfuse*, a somewhat balanced fusion/distribution strategy).

Note that `streamcluster`, the least affine of all benchmarks, exhausted memory in the polyhedral back-end and therefore no result is displayed. Benchmark `mummergpu` is not included in the results since it contains CUDA code and the front-end can only instrument code run on the CPU.

4.7.4 Trace compression results

In this section we show how well the folding algorithm is able to compress the raw DDG generated from the Rodinia benchmarks.

Table 4.10 gives statistics on the size and precision of the output of four versions of the folding algorithm:

- **F** is the basic algorithm as described in Section 4.5, with label widening for instructions and without for dependencies.
- $\mathbf{F_W}$ is the algorithm with label widening for both instructions and dependencies.
- $\mathbf{F_{GG}}$ is the same as **F** but with geometric give up.
- $\mathbf{F_{GG,W}}$ is the same as $\mathbf{F_W}$ but with geometric give up.

The threshold for the geometric give up was set to allow $4d + 1$ intermediate polyhedra in each d dimensional space. That is, enough for the affine function constructed to be made up of up to four d dimensional pieces.

For each algorithm we report the following statistics:

- **#P** is the number of polyhedra in the output stream.
- For dependencies, **%A** is the number of dependence instances that where in an affine piece of the label function. A piece of the label function is considered affine if it has no $\top$ coefficient. This column is omitted for algorithm **F** since by construction it always contains 100%.
- Similarly for instructions, **%A** is the number of instruction instances that where in an affine piece of the label function. A piece of the label function of a static instruction is considered affine if it either:
  - does not perform a memory access, or
  - has no $\top$ coefficient in its memory access function.
- **#ML** is the maximum number of intermediate polyhedra live at any moment of the execution, indicating the memory usage of the algorithm.

The **Input Size** columns show the total number of entries in all dependency and instruction input streams. Since this experiment does not require running Mickey's polyhedral back-end we where able to run all benchmarks beside `streamcluster` with their largest data-set. `streamcluster` was not run with its full input set since it triggers the worst case complexity of the folding algorithm in $\mathbf{F_W}$ mode. That is, with label widening but without geometric give up there are large amount of intermediary polyhedra live at the same time. With all other configurations `streamcluster` finishes in reasonable time even with the full input set.

Finally, as in the last section the numbers reported in Table 4.10 correspond to applying the folding-based analysis on the hot region of each benchmark, we have filtered out the phases where the benchmarks read their input or write their output. This hot region often involves numerous function calls.

Since the polyhedral optimization performed in the back-end is an exponential problem it is crucial that the output of the folding-based analysis is of tractable size. Table 4.10 clearly shows that $\mathbf{F_{GG}}$ and $\mathbf{F_{GG,W}}$ produce drastically smaller outputs than the other two versions. As indicated by the **%A** column, $\mathbf{F_{GG,W}}$ is roughly as precise as $\mathbf{F_{GG}}$, but produces an even smaller output. In fact only the output of $\mathbf{F_{GG,W}}$ is small enough for the back-end to handle.

4.8 Related Work

This section describes previous work related to dynamic data-flow analysis and trace compression.

Benchmark	Dependencies											
	Input Size	F		F_W			F_{GG}			$F_{GG,W}$		
		#P	#ML	F	%A	#ML	F	%A	#ML	#P	%A	#ML
backprop	19 M	164	390	164	100%	390	164	100%	390	164	100%	390
bfs	5 M	810 K	958 K	781 K	94%	943 K	79	34%	774	75	34%	774
b+tree	156 M	212 K	706 K	201 K	99%	575 K	115	97%	3 K	115	97%	3 K
cfd	2 G	1 M	4 M	953 K	98%	2 M	924	23%	9 K	919	23%	9 K
heartwall	211 G	14 K	10 K	13 K	90%	8 K	1 K	10%	5 K	1 K	10%	5 K
hotspot	79 M	22 K	44 K	22 K	94%	44 K	797	0%	6 K	797	0%	6 K
hotspot3D	15 G	168	1 K	162	91%	1 K	168	100%	1 K	162	91%	1 K
kmeans	3 G	167 K	73 K	30 K	98%	12 K	248	98%	528	239	98%	513
lavaMD	18 G	31 K	2 K	30 K	94%	2 K	31 K	100%	2 K	30 K	94%	2 K
leukocyte	5 G	517 K	169 K	514 K	97%	120 K	233	99%	70 K	232	97%	68 K
lud	718 M	6 K	1 K	4 K	98%	1 K	4 K	98%	1 K	4 K	98%	1 K
myocyte	5 M	14 K	19 K	14 K	100%	19 K	14 K	100%	19 K	14 K	100%	19 K
nn	782 K	124	265	124	100%	238	124	100%	264	124	100%	238
nw	217 M	301	1 K	296	99%	1 K	301	100%	1 K	296	99%	1 K
particlefilter	3 G	4 K	93 K	3 K	99%	2 K	590	8%	2 K	581	8%	2 K
pathfinder	74 M	35	139	35	100%	135	35	100%	139	35	100%	135
srad_v1	3 G	254	855	246	94%	828	254	100%	855	246	94%	828
srad_v2	1 G	281	816	273	97%	796	281	100%	816	273	97%	796
streamcluster	2 G	1 M	1 M	1 M	85%	1 M	8 K	85%	13 K	6 K	85%	13 K

Benchmark	Instructions						
	Input Size	F_W			$F_{GG,W}$		
		#P	%A	#ML	#P	%A	#ML
backprop	15 M	140	99%	304	140	99%	304
bfs	5 M	481 K	84%	491 K	38	54%	367
b+tree	99 M	122 K	79%	279 K	160	79%	1 K
cfd	1 G	566 K	97%	1 M	574	23%	5 K
heartwall	115 G	1 K	69%	3 K	1 K	9%	3 K
hotspot	47 M	13 K	70%	27 K	521	0%	3 K
hotspot3D	11 G	84	85%	782	84	85%	782
kmeans	1 G	29 K	95%	9 K	76	95%	288
lavaMD	9 G	15 K	71%	1 K	15 K	71%	1 K
leukocyte	2 G	355 K	84%	72 K	128	84%	40 K
lud	414 M	2 K	97%	843	2 K	97%	843
myocyte	3 M	9 K	99%	9 K	9 K	99%	9 K
nn	855 K	160	100%	189	160	100%	189
nw	111 M	155	100%	555	155	100%	555
particlefilter	1 G	1 K	99%	1 K	474	11%	1 K
pathfinder	42 M	24	61%	116	24	61%	116
srad_v1	2 G	179	93%	531	179	93%	531
srad_v2	721 M	204	93%	493	204	93%	493
streamcluster	1 G	611 K	71%	618 K	3 K	71%	6 K

Table 4.10 – Evaluation of the DDG compression of the folding algorithm on the Rodinia benchmark suite. Columns are explained in Section 4.7.4

Dynamic data-flow analysis

Dynamic data dependency analysis is a technique typically used to provide feedback to the programmer, e.g., about the existence or absence of dependencies along loops. The detection of parallelism along canonical directions, for loop transformations such as vectorization, has been particularly investigated [127, 32, 219, 113, 10, 114, 70, 214, 209, 73, 52, 149, 148], as it requires only relatively localized information. Another use case for dependence analysis the evaluation of effective reuse [137, 131, 21, 19] with the objective of pinpointing data-locality problems.

The SD^3 trace compression algorithm, already mentioned in Section 2.6, reduces the space required to store the dependence graph using memory access strides along innermost loops. It dynamically detects strides in the stream of memory accesses and also directly detects data-dependencies in strided format, without necessarily having to produce a raw DDG first. PSnAP [156], a memory access trace-compression system, uses a similar approach but achieves more aggressive compression ratios by allowing lossy compression. PSnAP does not track dependencies but only produces compressed memory access streams for use in cache simulators and similar performance prediction tools. It does not attempt to perfectly capture the stream addresses that a program addresses, but only a synthetic stream that exhibits similar patterns and regularities as the original. To this end, PSnAP groups accesses by the region of memory they access. The output of PSnAP is then a stream of grouped accesses, where for each group it tracks the number of accesses and a histogram that track the frequency at which different access strides.

Existing trace compression algorithms [112, 171] can be used to extract a polyhedral representation from an instrumented program execution. However, although they excel in rebuilding a polyhedral representation for a purely affine execution, they suffer inherent limitations for even partially non-affine traces. They share the idea of using pattern matching with affine functions with our folding algorithm but do not exploit geometric information as we do. The nested loop recognition algorithm of Ketterlin et al. [112] detects outer loops by maintaining and repeatedly examining a finite window of memory accesses. Another algorithm by Rodriguez et al. [171] instead introduces a new loop into its representation every time it cannot handle an access. For perfectly regular programs, Ketterlin's approach only requires a small window and Rodriguez's will only create as many loops as there are in the original program. In that simple case using a geometric approach does not make much difference, and both algorithms are very efficient. That is, for regular programs, with D the dimension of the iteration space and n the number of points, the complexity of both non-geometric approaches is $O(2^D n)$, same as our approach. However, in the context of profiling large non-fully affine programs, none of these two existing approaches can be used. The complexity of Ketterlin's algorithm grows quadratically with a parameter k that bounds the size of the window. Unfortunately, this forces a trade-off between speed and quality of the output when choosing the size of this window. If k is smaller than the number of irregularities along the innermost dimension, it is not able to capture the regularity, and thus compress, along outer dimensions. On the other hand, The complexity of Rodriguez's approach increases exponentially with the number of irregularities. So in practice, it has to give up even for nearly affine traces.

Streaming convex hull algorithms [28, 93] could also be applied to build a compact geometric representation of an instrumented execution. However, our approach can precisely represent non-convex polyhedra via a union of convex ones, while convex hull can only approximate this case with a single polyhedron.

Similarly to us, existing runtime polyhedral optimizers [190, 139] use runtime information to create a polyhedral representation of a program. PolyJIT [190] focuses on handling programs that do not fit the polyhedral model statically because of memory accesses, loop bounds and conditionals that are described by quadratic functions involving parameters. Apollo [108, 139]

handles this case, and many others, preventing static polyhedral optimizers from operating. Compared to our analysis, PolyJIT focuses on identifying 100% affine programs which may be rare in practice for many reasons such as the memory allocation concerns pointed out for `backprop`. Contrary to our analysis front-end, which tracks both separately, Apollo only traces memory accesses and then recomputes the dependencies from them. To handle quasi-affine access pattern, Apollo proposes a *tube* approximation mechanisms [199]. However, the tube mechanism is relatively limited and as soon as the stride distance of accesses is greater than a given constant threshold it gives up. In practice, in programs similar to the illustrative example of `backprop`, which use non-contiguous multidimensional arrays, Apollo will, as opposed to our analysis, neither manage to over-approximate the non-constant stride along inner-most dimensions nor detect the stride of 1 along the outer dimensions.

Static compilers increasingly use hybrid analyses approaches to handle programs that they can not completely model. These compilers use code versioning combined with runtime alias checks [6, 62] or dynamic checks for the validity of transformation [177, 178] to speculatively optimize programs even when the optimizations are not guaranteed to always be legal. These approaches very effectively broaden the range of programs that can be optimized, but it is still very difficult to statically decide which optimizations are profitable. The analysis presented here in this chapter could be applied to this problem of finding profitable transformations as a form of FDO [192].

Calling context trees and interprocedural program representations

As described in Section 2.4.1.2, using calling context trees to disambiguate instructions in different calling contexts is an idea from Ammons, Ball and Larus [7]. In order to bound the depth of the tree in the presence of recursion Ammons et al. do not take cyclic paths through the CG into account. Instead the CG is transformed into an acyclic graph by pruning some edges. At runtime, the calls corresponding to these edges are of course still executed, but the algorithm that constructs the CCT does not take them into account. The same approach is used to ignore cycles in the CFGs of functions that contain loops.

Loop-call context trees [181] simply encode the calling context of intra-procedural loops and suffer from the same size problems in recursive programs. However, they do not track loop iterations at runtime, only loop entries and exits. Consequently, this approach can not track which events happened during which iteration of a loop, only that it happened while the loop was active.

[200] presents an algorithm to compactly encode recursive calling contexts implemented in a profiling tool using dynamic binary instrumentation. Here a path through the CG is encoded using linear sequences of small integer IDs. By aggressively reusing IDs for different functions in different contexts one can effectively keep the number of bits required to represent IDs small. Recursive calling contexts are then represented using run-length encoding. That is, the algorithm first finds acyclic sub-part of recursive CGs and then applies run-length encoding to represent repeated occurrences of these acyclic sub-CGs compactly. This works very well for simple recursive programs but breaks down for more complex cases. To handle problematic cases the paper proposes to virtually unroll these recursive loops. Here the recursive program is not actually unrolled, but the profiling tool changes its instrumentation on the fly to simulate unrolling. For now, the unrolling factor has to be chosen ahead of time to fit a given application.

Kobeissi et al. [116] have developed an extension for the LLVM-based [124] polyhedral compiler Polly [81] which can apply polyhedral transformations to recursive loop nests [116]. Their system first statically detects recursive functions in LLVM IR. It then generates an instrumented version of the program to dynamically trace the execution of basic blocks inside of recursive functions. This stream of basic block IDs is then fed to the polyhedral trace compression of Ketterlin

et al. [112], already mentioned above, to discover if the recursive program's dynamic behaviour can be modelled using polyhedral loop nests. In the third and final step, the tool then generates an optimized version of the program where recursive functions have been transformed into loops and optimized with Polly. However, the tool of Kobeissi et al. [116] is still a proof of concept. It neither statically nor dynamically verifies whether the polyhedral behaviour observed during the profiling step is input data-dependent or not. Consequently, the optimizer may apply loop transformations that are only legal for some inputs.

4.9 Conclusion and Perspectives

In this chapter we introduced MICKEY, a profiling-based polyhedral optimization feedback tool. MICKEY tackles the problem of dynamic data dependence profiling and polyhedral optimization from binary programs, while addressing the associated scalability challenges of handling traces with billions of operations, as we demonstrated. Our tool scales to real-life applications by safely over-approximating dependencies that do not fit the polyhedral model while still recovering precise information for those that do. It can also handle programs with recursive function calls.

Numerous technical contributions were required to enable *polyhedral transformation feedback on binary programs*, a significant step in complexity compared to prior approaches typically limited to discovering parallelization potential. We implemented numerous feedback reporting schemes for the user in MICKEY: flame-graphs, statistics on striding and vectorization potential per loop, proposed potential structured transformation, and a simplified annotated pseudo-code to help users to implement suggested transformations. However, the current output produced by our tool is still quite low-level and very much tied to the structure of the input binary, not necessarily to that of the programs source code. As a consequence, a user of MICKEY has to have quite some expertise both in reading machine code and in polyhedral transformations to effectively leverage the optimization feedback.

The process of mapping our results back from the machine code back to the source code is in itself a research topic, and currently we do not provide much more than what `objdump` does. Our ongoing efforts in this direction leverage techniques to undo compiler optimizations [4] and polyhedral program equivalence techniques [217, 16, 182]. We have also contacted a Human-Computer-Interaction researcher to help us better represent the information collected by MICKEY to make it easier to use and understand.

We are also currently working on several extensions to the folding algorithm that will allow it to handle more programs. The first one consists of adding new dimensions not present in the program to our representation. In other words, an instruction contained in, for example, a 2-dimensional loop nest in the program could be represented by a 3-dimensional polyhedron. This mechanism, already at work in trace compression algorithms [112, 171] will allow our analysis to handle tiled stencil computations and programs where 2-dimensional arrays are traversed by linearized 1-dimensional loops.

Another extension of the folding algorithm we have started to implement is a mechanism to reduce the loss of information that occurs during the widening of label functions. Instead of simply marking widened coefficients as $\top$ we could store an interval of values a widened coefficient can take. This would allow us to put a lower and upper bound on the label values that a widened label function can produce even when it has been widened. Simply put, a label function is no longer a single linear function, but a pair of linear functions. The first gives a lower bound on label values, and the other gives an upper bound. If no widening is necessary the two functions coincide, which means we can still perfectly reproduce the input stream the label function was created from.

Chapter 5

Conclusion and Future Work

Contents

For decades hardware and software have increased in complexity to provide ever more performance. To keep compatibility with older software and to make the life of programmers simple this complexity is often hidden behind layers of abstraction. When analysing performance one often has to "look under the hood" and peel away these layers of abstractions to properly understand the behaviour of our programs. This thesis presented a number of performance debugging tools that help with the daunting task of analysing the operation of programs at the machine code level. These tools guide programmers in the process of performance analysis and optimization by pointing out potential performance bottlenecks and program transformations that may fix them.

5.1 Summary

This section summarizes the main contributions of the thesis and highlights some difficulties we encountered.

5.1.1 Dynamic Binary Instrumentation with QEMU

Chapter 2 presented QEMU, the TCG plugin infrastructure (TPI), and several dynamic binary analyses we have developed on top of them. We have implemented a number of plugins that, for example, trace data-dependencies, or dynamically reconstruct the CFG and CG of programs. All these plugins were written to be as language and compiler independent as possible and to be easily portable between CPU architectures. This makes it possible to analyse programs written in different programming languages with on single tool. They can also be used to compare the output produced by different compilers for a single program.

Both QEMU and TPI are mature and actively maintained projects. QEMU's more or less machine-independent TCG IR makes it possible to write binary analyses that can be easily ported to different target architectures. In industry, TPI is mainly used for low-level performance measurements and to explore the performance of new hardware platforms. TPI directly exposes the internals of QEMU and provides only a simple event-based API that is relatively cumbersome to use for implementing higher-level analyses. During the course of this thesis we have developed a number of wrappers for both TPI and QEMU itself. These wrappers make it possible for plugins to inspect and modify TCG IR in a direct style at a higher granularity entire translation blocks.

One pain point of implementing performance debugging tools with QEMU is that its support for vector instructions on x86 is severely outdated. As of 2019 it only supports 128-bit SSE vector instructions up to the SSSE3 instruction set. It does not even support SSE4 instructions, which were released nearly 15 years ago. For x86 QEMU only covers the instructions emitted by mainstream compilers with default settings. In other words, GCC or Clang without any `-march` flags. There recently has been a Google Summer of Code project to add support for newer vector instructions to QEMU [24]. However, this project has not been finished and it not clear when and if it will be merged into upstream QEMU. For other CPU architectures, such as ARM, QEMU is much more up to date and sometimes even implements extensions that are not available in hardware yet. Our short term objective is thus to port all our plugins to ARM.

5.1.2 Sensitivity-Based Performance Bottleneck Analysis

Chapter 3 has presented Gus, a prototype profiling tool that uses sensitivity analysis to find performance bottlenecks. The sensitivity analysis is driven by a CPU simulator that predicts execution times using a high-level performance model. By using a simulator, we can exert precise and fine-grained control over the throughput of different CPU resources, such as the functional

units or the memory bandwidth. Gus's CPU model, based on the notion of abstract, throughput limited resources, is intentionally very high-level to allow for fast simulation times.

Gus is still under active development, and we are still making alterations to its performance model to allow it to model the performance of a broader range of programs more precisely. As a first step, we will extend the model to consider not only the throughput but also the latency of data transfers between cache levels and memory. We are, among other things, also working on adding more realistic cache replacement policies and a simple form of branch prediction to the simulator. Another limitation of Gus is the simplistic model of the throughput and interaction between different levels of the cache hierarchy [97]. To address this issue we would like to develop, similarly to the resource mapping for execution ports, a resource mapping for caches. An essential factor to consider here is that we do not want the model to become overly complex, as this may significantly slow down the simulator.

A big advantage of using a simulator to drive the sensitivity analysis is that it allows to decouple functional execution of a program from its performance model. We see great potential in implementing a more high-level sensitivity analysis that can detect if a specific source-level statement forms a bottleneck by varying the resource usage of its corresponding instructions.

Chapter 3 also presented Pipedream, a tool for measuring specific performance characteristics of CPUs, which we use to build Gus's performance model. Pipedream finds measured performance characteristics such as the throughput and latency of instructions by running a large set of automatically generated microbenchmarks. The tool can also find port mappings by analysing performance measurements of specially crafted microkernels using a LP solver. We have used it to produce a port mapping for the Intel Skylake CPU architecture that is comparable in quality to that produced by other authors such as Agner Fog [72] and Abel et al. [3]. Pipedream is able to find the port mappings for some instructions for which existing approaches fall back to manual analysis.

For the moment Pipedream can only benchmark and model a subset of the entire x86 instruction set. We are still working on extending our benchmark generator to cover more instructions and refining the instruction flow LP solver to be more robust to measurement noise. There are also several other machine parameters, such as the size of the ROB, that we have, for the moment, taken from the vendor's documentation that we intend to reverse-engineer ourselves. We are also developing a modified version of Pipedream that can build port mappings even for a machine without sophisticated hardware performance counters, using only time measurements of microbenchmarks.

5.1.3 Data-Dependence Driven Optimization Feedback

Chapter 4 presented Mickey, a performance debugging tool that provides feedback on the applicability and profitability of polyhedral optimizations. Mickey analyses compiled and already optimized programs to find optimizations missed by the compiler. Internally, Mickey uses a polyhedral, interprocedural IR to capture the iterations domains of instructions and the data-flow between them. This IR is constructed dynamically from a program's execution using the folding algorithm, an efficient trace algorithm developed by us. Mickey scales to real-life applications thanks to a safe, selective over-approximation mechanism for partially irregular data dependencies and iteration spaces. We have applied Mickey to an entire suite of benchmarks and found a large number of different transformations, ranging from vectorization, parallelization, and loop interchange, to loop tiling.

We have developed numerous technical contributions in order to provide polyhedral transformation feedback for compiled programs. Mickey has to bridge the gap between the high-level mathematical framework of the polyhedral model and the low-level details and complexities of real machine code. One the one hand, Mickey traces and models the data-flow between

deeply nested loops that can potentially span multiple functions and even recursive functions. On the other hand, it has be able to handle the highly irregular control-flow of manually optimized assembler and be aware of machine code idioms such as dependency breaking instructions. While the current folding algorithm gives reasonably tight approximations for the benchmarks we have applied it to, it seems some dependence patterns would profit from an gradual widening mechanism.

The design of MICKEY's IR and the folding algorithm have been guided by the polyhedral nature of the analysis back-end. We could very well explore profiling for other forms of regularity, such as tree structured parallelism.

5.2 Perspectives

A challenge faced by performance debugging is how to exclude "false" performance bugs. That is, just because a program spends a lot of time in a region does not mean that its performance can be improved. To fix a performance bug one needs a valid transformation that improves the usage of a bottleneck resource. MICKEY is able to tell if such a transformation exists, while GUS is able to tell if a given resource is a bottleneck. However, for the moment GUS and MICKEY are unconnected. For the above reasons we believe that both tools could greatly profit from each other. As an example, when GUS finds that a program is bandwidth bound, MICKEY could tell user if a loop interchange or tiling are possible. Reversely, when MICKEY finds that tiling is possible, GUS can judge if it is actually profitable.

One fundamental aspect of performance debugging is the user experience of programmers. We believe that our tools made a big step towards providing programmers with more useful feedback than just aggregated execution times or cache miss statistics. One of the great advantages of the performance debugging by simulation, which we have not yet developed, is that it allows linking the low-level behaviour of the architecture with higher-level program semantics. However, both GUS and MICKEY more or less target expert developers that are at least familiar with machine code and program optimization. Even for such an expert, the way we currently present the feedback produced by our tools is not always straightforward to use. We have contacted people working in the field of human-computer interaction and are currently exploring different methods to visualize our optimization feedback to make it easier to use. In general, improving the usability of performance debugging tools is vital for making them more widely used.

An essential factor for the usability of our tools is the quality of the mapping from machine instructions back to the source code. For the moment, we solely rely on DWARF debug information for this purpose. One of the biggest stumbling blocks here is that GUS and MICKEY work on optimized binaries. Unfortunately, mainstream compilers, such as GCC and CLANG, are notorious for producing hard-to-use debug information at higher optimization levels. This problem is especially noticeable for programs where the compiler aggressively reschedules instructions. Other performance debugging tools [203, 140] circumvent this problem using binary analyses that "undo" optimizations of the compiler which makes the mapping from binary to source code clearer. Another idea, that would like to explore is to leverage techniques that show program equivalence for this mapping problem. For the moment, it is not clear to us whether it would be better to improve the quality of the debug information emitted by the compiler directly or to solve this problem using more sophisticated binary analysis techniques.

5.3 List of Publications

Part of the work presented in this thesis has been published in the following articles:

- Fabian Gruber, Manuel Selva, Diogo Sampaio, Christophe Guillon, Antoine Moynault, Louis-Noël Pouchet, and Fabrice Rastello "Data-flow/Dependence Profiling for Structured Transformations". Article published in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (PPoPP '19) [82].
- Manuel Selva, Fabian Gruber, Diogo Sampaio, Christophe Guillon, Antoine Moynault, Louis-Noël Pouchet, and Fabrice Rastello "Building a Polyhedral Representation from an Instrumented Execution: Making Dynamic Analyses of non-Affine Programs Scalable". Will be published in *ACM Transactions on Architecture and Code Optimization* (TACO '19) [183].
- Fabian Gruber, Manuel Selva, Diogo Sampaio, Christophe Guillon, Louis-Noël Pouchet, and Fabrice Rastello "*Building of a Polyhedral Representation from an Instrumented Execution: Making Dynamic Analyses of non-Affine Programs Scalable*". Research report published at Inria HAL [83]

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. "Control-flow Integrity Principles, Implementations, and Applications". In: *ACM Trans. Inf. Syst. Secur.* 13.1 (Nov. 2009). ISSN: 1094-9224. 5

[2] Andreas Abel and Jan Reineke. *uops.info - Latency, Throughput, and Port Usage Information.* URL: `https://uops.info/` (visited on 03/07/2019). 78

[3] Andreas Abel and Jan Reineke. "Uops.Info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS '19. 2019. 47, 63, 65, 78, 87, 141

[4] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. "HPCTOOLKIT: tools for performance analysis of optimized parallel programs". In: *Concurrency and Computation: Practice and Experience* 22.6 (2010). 4, 5, 31, 36, 44, 138

[5] Kumar Akhilesh and Trivedi Malay. *Intel® Xeon Scalable Process Architecture Deep Dive.* presentation from the Intel® Xeon launch event. June 2017. 42

[6] Péricles Alves, Fabian Gruber, Johannes Doerfert, Alexandros Lamprineas, Tobias Grosser, Fabrice Rastello, and Fernando Magno Quintão Pereira. "Runtime Pointer Disambiguation". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications.* OOPSLA 2015. 2015. 8, 137

[7] Glenn Ammons, Thomas Ball, and James R. Larus. "Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling". In: *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation.* PLDI '97. 1997. 24, 26, 94, 103, 137

[8] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. "A Compiler-level Intermediate Representation Based Binary Analysis and Rewriting System". In: *Proceedings of the 8th ACM European Conference on Computer Systems.* EuroSys '13. 2013. 6, 31

[9] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. "An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries". In: *25th USENIX Security Symposium (USENIX Security 16).* 2016. 21

[10] Ran Ao, Guangming Tan, and Mingyu Chen. "ParaInsight: An Assistant for Quantitatively Analyzing Multi-granularity Parallel Region". In: *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on.* IEEE. 2013. 136

[11] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. "Adaptive Optimization in the JalapeÑO JVM". In: *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications.* OOPSLA '00. 2000. 5

[12] Matthew Arnold, Michael Hind, and Barbara G. Ryder. "Online Feedback-directed Optimization of Java". In: *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '02. 2002. 5

[13] Matthew Arnold and Barbara G. Ryder. "A Framework for Reducing the Cost of Instrumented Code". In: *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. PLDI '01. 2001. 34

[14] Utpal K. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, 1988. ISBN: 0898382890. 128

[15] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. "BYTEWEIGHT: Learning to Recognize Functions in Binary Code". In: *23rd USENIX Security Symposium (USENIX Security 14)*. 2014. 21

[16] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P. Sadayappan. "PolyCheck: Dynamic Verification of Iteration Space Transformations on Affine Programs". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. 2016. 138

[17] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. 2005. 12, 106

[18] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. "The Polyhedral Model Is More Widely Applicable Than You Think". In: *Compiler Construction*. Ed. by Rajiv Gupta. 2010. 8

[19] Erik Berg and Erik Hagersten. "Fast data-locality profiling of native execution". In: *ACM SIGMETRICS Performance Evaluation Review*. Vol. 33. 1. ACM. 2005. 136

[20] Kristof Beyls and Erik H. D'Hollander. "Discovery of Locality-improving Refactorings by Reuse Path Analysis". In: *Proceedings of the Second International Conference on High Performance Computing and Communications*. HPCC'06. 2006. 90

[21] Kristof Beyls and Erik D'Hollander. "Discovery of Locality-Improving Refactorings by Reuse Path Analysis". In: *High Performance Computing and Communications* (2006). 136

[22] Wlodzimierz Bielecki and Marek Palkowski. "Tiling arbitrarily nested loops by means of the transitive closure of dependence graphs". In: *AMCS : International Journal of Applied Mathematics and Computer Science* 26.4 (2016). 90

[23] Nathan Binkert et al. "The Gem5 Simulator". In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011). ISSN: 0163-5964. 4, 12, 46

[24] Jan Bobek and Richard Henderson. URL: `https://wiki.qemu.org/Google_Summer_of_Code_2019#AVX` (visited on 03/10/2019). 140

[25] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. 2008. 6, 90, 130

[26] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. 2008. 90, 94, 122, 130

[27] Peter Brass. *Advanced data structures*. Vol. 193. Cambridge University Press Cambridge, 2008. 25

[28] G. S. Brodal and R. Jacob. "Dynamic planar convex hull". In: *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.* Nov. 2002. 136

[29] Derek L. Bruening. "Efficient, Transparent, and Comprehensive Runtime Code Manipulation". AAI0807735. PhD thesis. Cambridge, MA, USA, 2004. 6, 28, 30

[30] Doug Burger and Todd M. Austin. "The SimpleScalar Tool Set, Version 2.0". In: *SIGARCH Comput. Archit. News* 25.3 (June 1997). ISSN: 0163-5964. 46

[31] Martin Burtscher, Byoung-Do Kim, Jeff Diamond, John McCalpin, Lars Koesterke, and James Browne. "PerfExpert: An Easy-to-Use Performance Diagnosis Tool for HPC Applications". In: *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. Nov. 2010. 45

[32] Khansa Butt, Abdul Qadeer, Ghulam Mustafa, and Abdul Waheed. "Runtime analysis of application binaries for function level parallelism potential using QEMU". In: *Open Source Systems and Technologies (ICOSST), 2012 International Conference on*. IEEE. 2012. 136

[33] Victoria Caparrós Cabezas and Markus Püschel. "Extending the roofline model: Bottleneck analysis with microarchitectural constraints". In: *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. Oct. 2014. 47

[34] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. "EXE: Automatically Generating Inputs of Death". In: *ACM Trans. Inf. Syst. Secur.* 12.2 (Dec. 2008). ISSN: 1094-9224. 32

[35] David Callahan, John Cocke, and Ken Kennedy. "Estimating interlock and improving balance for pipelined architectures". In: *Journal of Parallel and Distributed Computing* 5.4 (1988). ISSN: 0743-7315. 41

[36] Trevor E. Carlson, Wim Heirman, Lieven Eeckhout, and Ibrahim Hur. *The Sniper Multi-Core Simulator*. URL: `http://snipersim.org/w/The_Sniper_Multi-Core_Simulator` (visited on). 5, 47, 49

[37] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. "An Evaluation of High-Level Mechanistic Core Models". In: *ACM Trans. Archit. Code Optim.* 11.3 (Aug. 2014). ISSN: 1544-3566. 5, 36, 46

[38] Steve Carr and Ken Kennedy. "Improving the Ratio of Memory Operations to Floating-point Operations in Loops". In: *ACM Trans. Program. Lang. Syst.* 16.6 (Nov. 1994). ISSN: 0164-0925. 41

[39] Andres S. Charif-Rubial, Emmanuel Oseret, José Noudohouenou, William Jalby, and Ghislain Lartigue. "CQA: A code quality analyzer tool at binary level". In: *2014 21st International Conference on High Performance Computing (HiPC)*. Dec. 2014. 45, 48

[40] Prasanth Chatarasi, Jun Shirako, and Vivek Sarkar. "Polyhedral transformations of explicitly parallel programs". In: *5th International Workshop on Polyhedral Compilation Techniques (IMPACT)*. 2015. 90, 130

[41] Chatelet Chatelet, Clement Courbet, Ondrej Sykora, and Nicolas Paglieri. *Google EXEgesis*. URL: `https://llvm.org/docs/CommandGuide/llvm-exegesis.html` (visited on 03/07/2019). 47

[42] 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: *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee. 2009. 28, 127

[43] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron. "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads". In: *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*. IISWC '10. 2010. 28, 91, 127, 128

[44] Dehao Chen, Neil Vachharajani, Robert Hundt, Xinliang Li, Stephane Eranian, Wenguang Chen, and Weimin Zheng. "Taming Hardware Event Samples for Precise and Versatile Feedback Directed Optimizations". In: *IEEE Transactions on Computers* 62.2 (Feb. 2013). ISSN: 0018-9340. 4

[45] Tong Chen, Jin Lin, Xiaoru Dai, Wei-Chung Hsu, and Pen-Chung Yew. "Data Dependence Profiling for Speculative Optimizations". In: *Compiler Construction*. Ed. by Evelyn Duesterwald. 2004. 27

[46] Jong-Deok Choi, Barton P. Miller, and Robert H. B. Netzer. "Techniques for Debugging Parallel Programs with Flowback Analysis". In: *ACM Trans. Program. Lang. Syst.* 13.4 (Oct. 1991). ISSN: 0164-0925. 32, 94

[47] Cristian Coarfa, John Mellor-Crummey, Nathan Froyd, and Yuri Dotsenko. "Scalability Analysis of SPMD Codes Using Expectations". In: *Proceedings of the 21st Annual International Conference on Supercomputing*. ICS '07. 2007. 8, 36

[48] Jean-François Collard, Denis Barthou, and Paul Feautrier. "Fuzzy Array Dataflow Analysis". In: *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP '95. 1995. 8, 9, 90

[49] Apple Corporation. *JavaScriptCore FTL JIT*. URL: `https://trac.webkit.org/wiki/FTLJIT` (visited on 16/08/2019). 6

[50] Google Corporation. *Unladden swallow Python VM*. URL: `https://code.google.com/archive/p/unladen-swallow/` (visited on 16/08/2019). 6

[51] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Chapter 35: Intel Processor Trace. 2019. 5, 33

[52] Intel Corporation. *Intel® Advisor*. 2017. URL: `https://software.intel.com/en-us/advisor` (visited on 02/08/2019). 5, 136

[53] Intel® Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2011. 47

[54] Intel® Corporation. *Intel® 64 and IA-32 architectures Software Developer's Manual*. 2011. 21, 37, 42, 47, 80

[55] Intel® Corporation. *Intel® X86 Encoder Decoder*. URL: `https://intelxed.github.io/` (visited on 22/08/2019). 66

[56] International Business Machines Corporation. *Program event recording (PER) facility*. URL: `https://www.ibm.com/support/knowledgecenter/en/SSB23S_1.1.0.13/gtpp1/gtpp1mst67.html` (visited on 08/07/2019). 33

[57] Oracle Corporation. *Oracle VirtualBox VM*. URL: `https://www.virtualbox.org` (visited on 20/07/2019). 12

[58] Emilio G. Cota, Paolo Bonzini, Alex Bennée, and Luca P. Carloni. "Cross-ISA Machine Emulation for Multicores". In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. CGO '17. 2017. 16, 34

[59] Rene De La Briandais. "File Searching Using Variable Length Keys". In: *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*. IRE-AIEE-ACM '59 (Western). 1959. 25

[60] Benoît Dupont de Dinechin, François de Ferrière, Christophe Guillon, and Artour Stoutchinin. "Code generator optimizations for the ST120 DSP-MCU core". In: *Proceedings of the 2000 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES 2000, San Jose, California, USA, November 7-18, 2000*. 2000. 31

[61] Lamia Djoudi, Denis Barthou, Patrick Carribault, Christophe Lemuet, Jean-Thomas Acquaviva, William Jalby, et al. "Maqao: Modular assembler quality analyzer and optimizer for itanium 2". In: *The 4th Workshop on EPIC architectures and compiler technology, San Jose*. Vol. 200. 5. 2005. 44

[62] Johannes Doerfert, Tobias Grosser, and Sebastian Hack. "Optimistic Loop Optimization". In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. CGO '17. 2017. 8, 90, 137

[63] Johannes Doerfert, Clemens Hammacher, Kevin Streit, and Sebastian Hack. "Spolly: speculative optimizations in the polyhedral model". In: *IMPACT 2013* (2013). 90

[64] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. "Repeatable Reverse Engineering with PANDA". In: *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*. PPREW-5. 2015. 6, 30

[65] Jack Dongarra, Kevin London, Shirley Moore, Philip Mucci, Daniel Terpstra, Haihang You, and Min Zhou. "Experiences and lessons learned with a portable interface to hardware performance counters". In: *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. IEEE. 2003. 4, 36, 66, 69

[66] Jan Edler and Mark D. Hill. *Dinero IV trace-driven uniprocessor cache simulator*. 1998. 29

[67] Andrew Edwards, Hoi Vo, and Amitabh Srivastava. *Vulcan binary transformation in a distributed environment*. Microsoft Research Technical Report, MSR-TR-2001-50. 2001. 31

[68] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. "A Mechanistic Performance Model for Superscalar Out-of-order Processors". In: *ACM Trans. Comput. Syst.* 27.2 (May 2009). ISSN: 0734-2071. 5, 36, 46

[69] Yliès Falcone, Leonardo Mariani, Antoine Rollet, and Saikat Saha. "Runtime Failure Prevention and Reaction". In: *Lectures on Runtime Verification: Introductory and Advanced Topics*. Ed. by Ezio Bartocci and Yliès Falcone. Cham: Springer International Publishing, 2018, pp. 103–134. ISBN: 978-3-319-75632-5. DOI: 10.1007/978-3-319-75632-5_4. URL: https://doi.org/10.1007/978-3-319-75632-5_4. 3

[70] Karl-Filip Faxén, Konstantin Popov, Sverker Jansson, and Lars Albertsson. "Embla - Data Dependence Profiling for Parallel Programming". In: *2008 International Conference on Complex, Intelligent and Software Intensive Systems*. Mar. 2008. 32, 33, 90, 136

[71] Paul Feautrier and Christian Lengauer. "Polyhedron model". In: *Encyclopedia of Parallel Computing*. Springer, 2011, pp. 1581–1592. 6, 90, 96

[72] Agner Fog. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. Aug. 2019. URL: http://www.agner.org/optimize/ (visited on 02/08/2019). 47, 57, 63, 69, 78, 85, 87, 141

[73] Saturnino Garcia, Donghwan Jeon, Christopher M. Louie, and Michael Bedford Taylor. "Kremlin: Rethinking and Rebooting Gprof for the Multicore Age". In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. 2011. 32, 33, 44, 136

[74] *GDB: The GNU Project Debugger*. URL: https://www.gnu.org/software/gdb (visited on 05/07/2019). 3

[75] Xinyang Ge, Weidong Cui, and Trent Jaeger. "GRIFFIN: Guarding Control Flows Using Intel Processor Trace". In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '17. 2017. 5, 33

[76] Davvy Genbrugge, Stijn Eyerman, and Lieven Eeckhout. "Interval simulation: Raising the level of abstraction in architectural simulation". In: *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. Jan. 2010. 5, 36, 46

[77] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. "Semi-automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies". In: *Int. J. Parallel Program.* 34.3 (June 2006). ISSN: 0885-7458. 122

[78] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. "Gprof: A call graph execution profiler". In: *ACM Sigplan Notices*. Vol. 17. 6. ACM. 1982. 3, 5

[79] Brendan Gregg. *Blazing Performance with Flame Graphs*. URL: https://www.usenix.org/conference/lisa13/technical-sessions/plenary/gregg. Presentation at UNSENIX LISA 2013. 105, 124

[80] Arthur Griffith. *GCC: the complete reference*. McGraw-Hill, Inc., 2002. 6

[81] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. "Polly - Performing Polyhedral Optimizations on a Low-level Intermediate Representation". In: *Parallel Processing Letters* 22.04 (2012). 6, 90, 91, 130, 137

[82] Fabian Gruber, Manuel Selva, Diogo Sampaio, Christophe Guillon, Antoine Moynault, Louis-Noël Pouchet, and Fabrice Rastello. "Data-flow/Dependence Profiling for Structured Transformations". In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. PPoPP '19. 2019. 143

[83] Fabian Gruber, Manuel Selva, Diogo Sampaio, Christophe Guillon, Louis-Noël Pouchet, and Fabrice Rastello. *Building of a Polyhedral Representation from an Instrumented Execution: Making Dynamic Analyses of non-Affine Programs Scalable*. Research Report RR-9244. Jan. 2019. URL: https://hal.inria.fr/hal-01967828. 143

[84] Christophe Guillon. "Program Instrumentation with QEMU". In: *Proceedings of the International QEMU User's Forum*. QUF '11. 2011. 12, 16, 106

[85] Christophe Guillon, Christophe Lyon, Cèdric Vincent, Francois De-Ferriere, Antoine Moynault, Fabian Gruber, and Bouvier Pierrick. *QEMU Plugins Infrastructure on Github*. 2019. URL: http://github.com/atos-tools/qemu (visited on 26/06/2019). 16

[86] Julian Hammer, Jan Eitzinger, Georg Hager, and Gerhard Wellein. "Kerncraft: A Tool for Analytic Performance Modeling of Loop Kernels". In: *Tools for High Performance Computing 2016*. Ed. by Christoph Niethammer, José Gracia, Tobias Hilbrich, Andreas Knüpfer, Michael M. Resch, and Wolfgang E. Nagel. 2017. 43

[87] Phil Hanlon. "The enumeration of bipartite graphs". In: *Discrete Mathematics* 28.1 (1979). 64

[88] Dave Hansen. *Linx perf patch to introduces a maximum hardware performance counter sampling frequency*. May 2013. URL: `https://lkml.org/lkml/2013/5/29/640` (visited on 14/08/2019). 4

[89] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. "Make It Work, Make It Right, Make It Fast: Building a Platform-neutral Whole-system Dynamic Binary Analysis Platform". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. 2014. 30

[90] John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011. 38

[91] John L. Henning. "SPEC CPU2000: Measuring CPU performance in the new millennium". In: *Computer* 33.7 (2000). 2

[92] John L. Henning. "SPEC CPU2006 Benchmark Descriptions". In: *SIGARCH Comput. Archit. News* 34.4 (Sept. 2006). ISSN: 0163-5964. 2, 33, 129

[93] John Hershberger and Subhash Suri. "Convex hulls and related problems in data streams". In: *Proc. of the ACM/DIMACS Workshop on Management and Processing of Data Streams*. 2003. 136

[94] Mark D. Hill and Michael R. Marty. "Amdahl's Law in the Multicore Era". In: *Computer* 41.7 (July 2008). ISSN: 0018-9162. 2

[95] Israel Hirsh and Gideon S. *Intel® Architecture Code Analyzer*. URL: `https://software.intel.com/en-us/articles/intel-architecture-code-analyzer` (visited on 03/07/2019). 41

[96] Roger W. Hockney and Ian J. Curington. "f12: A parameter to characterize memory and communication bottlenecks". In: *Parallel Computing* 10.3 (1989). ISSN: 0167-8191. 41

[97] Johannes Hofmann, Jan Eitzinger, and Dietmar Fey. "Execution-Cache-Memory Performance Model: Introduction and Validation". In: *CoRR* abs/1509.03118 (2015). arXiv: `1509.03118`. 86, 141

[98] Justin Holewinski, Ragavendar Ramamurthi, Mahesh Ravishankar, Naznin Fauzia, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. "Dynamic trace-based analysis of vectorization potential of applications". In: *ACM SIGPLAN Notices* 47.6 (2012). 90

[99] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, Prashant Singh Rawat, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P. Sadayappan. "GPU Code Optimization Using Abstract Kernel Emulation and Sensitivity Analysis". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. 2018. 8, 36, 43

[100] Chung-hsing Hsu and Wu-chun Feng. "A Power-Aware Run-Time System for High-Performance Computing". In: *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. Nov. 2005. 9, 36

[101] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. "Enforcing Unique Code Target Property for Control-Flow Integrity". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. 2018. 5, 33

[102] Jian Huang and David J. Lilja. "Extending value reuse to basic blocks with compiler support". In: *IEEE Transactions on Computers* 49.4 (Apr. 2000). ISSN: 0018-9340. 39

[103] Robert Hundt, Easwaran Raman, Martin Thuresson, and Neil Vachharajani. "MAO – An Extensible Micro-architectural Optimizer". In: *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '11. 2011. 31

[104] Yuan-Shin Hwang, Tzong-Yen Lin, and Rong-Guey Chang. "DisIRer: Converting a Retargetable Compiler into a Multiplatform Binary Translator". In: *ACM Trans. Archit. Code Optim.* 7.4 (Dec. 2010). ISSN: 1544-3566. 6

[105] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. "Cache-aware Roofline model: Upgrading the loft". In: *IEEE Computer Architecture Letters* 13.1 (Jan. 2014). ISSN: 1556-6056. 41

[106] *Introduction of a separate scheduling model for Intel Skylake processors in LLVM*. 19 Sept. 2017. URL: `https://llvm.org/svn/llvm-project/llvm/trunk/?p=313613` (visited on 10/07/2019). 37

[107] Song Jiang and Xiaodong Zhang. "LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance". In: *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '02. 2002. 86

[108] Alexandra Jimborean, Luis Mastrangelo, Vincent Loechner, and Philippe Clauss. "VMAD: An Advanced Dynamic Program Analysis and Instrumentation Framework". In: *Compiler Construction: 21st International Conference, CC 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 – April 1, 2012. Proceedings*. Ed. by Michael O'Boyle. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 220–239. ISBN: 978-3-642-28652-0. DOI: `10.1007/978-3-642-28652-0_12`. URL: `https://doi.org/10.1007/978-3-642-28652-0_12`. 136

[109] P. J. Joseph, Kapil Vaswani, and Matthew J. Thazhuthaveetil. "A Predictive Performance Model for Superscalar Processors". In: *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 39. 2006. 43

[110] E. K. Ardestani and J. Renau. "ESESC: A Fast Multicore Simulator Using Time-Based Sampling". In: *International Symposium on High Performance Computer Architecture*. HPCA'19. 2013. 5

[111] W. Kelly and W. Pugh. "A unifying framework for iteration reordering transformations". In: *Proceedings 1st International Conference on Algorithms and Architectures for Parallel Processing*. Vol. 1. Apr. 1995. 94, 101

[112] Alain Ketterlin and Philippe Clauss. "Prediction and Trace Compression of Data Access Addresses Through Nested Loop Recognition". In: *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '08. 2008. 10, 90, 93, 95, 136, 138

[113] Alain Ketterlin and Philippe Clauss. "Profiling Data-Dependence to Assist Parallelization: Framework, Scope, and Optimization". In: *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-45. Dec. 2012. 32, 90, 136

[114] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. "Prospector: A dynamic data-dependence profiler to help parallel programming". In: *HotPar'10: Proceedings of the USENIX workshop on Hot Topics in parallelism*. 2010. 32, 136

[115] Minjang Kim, Nagesh B. Lakshminarayana, Hyesoon Kim, and Chi-Keung Luk. "SD3: An Efficient Dynamic Data-Dependence Profiling Mechanism". In: *IEEE Transactions on Computers* 62.12 (Dec. 2013). ISSN: 0018-9340. 32, 33

[116] Salwa Kobeissi and Philippe Clauss. "The Polyhedral Model Beyond Loops Recursion Optimization and Parallelization Through Polyhedral Modeling". In: *IMPACT 2019 - 9th International Workshop on Polyhedral Compilation Techniques, In conjunction with HiPEAC 2019*. Jan. 2019. 137, 138

[117] Toshihiko Koju, Reid Copeland, Motohiro Kawahito, and Moriyoshi Ohara. "Re-constructing High-level Information for Language-specific Binary Re-optimization". In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. CGO '16. 2016. 31

[118] Souad Koliaï, Zakaria Bendifallah, Mathieu Tribalat, Cédric Valensi, Jean-Thomas Acquaviva, and William Jalby. "Quantifying Performance Bottleneck Cost Through Differential Analysis". In: *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ICS '13. 2013. 8, 36, 44

[119] Philipp Koppe, Benjamin Kollenda, Marc Fyrbiak, Christian Kison, Robert Gawlik, Christof Paar, and Thorsten Holz. "Reverse engineering x86 processor microcode". In: *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 2017. 37

[120] M. Kumar. "Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications". In: *IEEE Trans. Comput.* 37.9 (Sept. 1988). ISSN: 0018-9340. 44

[121] H. T. Kung. "Memory Requirements for Balanced Computer Architectures". In: *Proceedings of the 13th Annual International Symposium on Computer Architecture*. ISCA '86. 1986. 41

[122] James R. Larus and Thomas Ball. "Rewriting Executable Files to Measure Program Behavior". In: *Softw. Pract. Exper.* 24.2 (Feb. 1994). ISSN: 0038-0644. 3

[123] James R. Larus and Eric Schnarr. "EEL: Machine-independent Executable Editing". In: *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*. PLDI '95. 1995. 6, 31

[124] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO '04. 2004. 30, 43, 130, 137

[125] Jan Laukemann, Julian Hammert, 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)*. Nov. 2018. 43, 48

[126] Kevin P. Lawton. "Bochs: A Portable PC Emulator for Unix/X". In: *Linux J.* 1996.29es (Sept. 1996). ISSN: 1075-3583. 12

[127] Zhen Li, Rohit Atre, Zia Ul-Huda, Ali Jannesari, and Felix Wolf. "DiscoPoP: A profiling tool to identify parallelization opportunities". In: *Tools for High Performance Computing 2014*. Springer, 2015, pp. 37–54. 32, 136

[128] Zhen Li, Michael Beaumont, Ali Jannesari, and Felix Wolf. "Fast data-dependence profiling by skipping repeatedly executed memory operations". In: *International Conference on Algorithms and Architectures for Parallel Processing*. Springer. 2015. 32

[129] Zhen Li, Ali Jannesari, and Felix Wolf. "An efficient data-dependence profiler for sequential and parallel programs". In: *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE. 2015. 33

[130] Kathleen A. Lindlan, Janice Cuny, Allen D. Malony, Sameer Shende, Bernd Mohr, Reid Rivenburgh, and Craig Rasmussen. "A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates". In: *SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. Nov. 2000. 44

[131] Xu Liu and John Mellor-Crummey. "Pinpointing data locality problems using data-centric analysis". In: *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*. IEEE. 2011. 136

[132] Xu Liu and Bo Wu. "ScaAnalyzer: A Tool to Identify Memory Scalability Bottlenecks in Parallel Programs". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '15. 2015. 8, 36, 44

[133] O. G. Lorenzo, T. F. Pena, J. C. Cabaleiro, J. C. Pichel, and F. F. Rivera. "3DyRM: a dynamic roofline model including memory latency information". In: *The Journal of Supercomputing* 70.2 (Nov. 2014). ISSN: 1573-0484. 41

[134] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: building customized program analysis tools with dynamic instrumentation". In: *Acm sigplan notices*. Vol. 40. 6. ACM. 2005. 6, 30

[135] Allen D. Malony, Sameer S. Shende, Nick Trebon, Jaideep Ray, R. Armstrong, Craig Rasmussen, and M. Sottile. "Performance technology for parallel and distributed component software". In: *Concurrency and Computation: Practice and Experience* 17.2-4 (2005). eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.931. 44

[136] *MAQAO (Modular Assembly Quality Analyzer and Optimizer) project*. (Visited on 10/07/2019). 44

[137] G. Marin, J. Dongarra, and D. Terpstra. "MIAMI: A framework for application performance diagnosis". In: *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Mar. 2014. 45, 136

[138] Gabriel Marin, Collin McCurdy, and Jeffrey S. Vetter. "Diagnosis and Optimization of Application Prefetching Performance". In: *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ICS '13. 2013. 46

[139] Juan Manuel Martinez Caamaño, Manuel Selva, Philippe Clauss, Artyom Baloian, and Willy Wolff. "Full runtime polyhedral optimizing loop transformations with the generation, instantiation, and scheduling of code-bones". In: *Concurrency and Computation: Practice and Experience* 29.15 (2017). e4192 cpe.4192. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4192. 8, 9, 93, 136

[140] Omayma Matoussi and Frédéric Pétrot. "A mapping approach between IR and binary CFGs dealing with aggressive compiler optimizations for performance estimation". In: *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2018. 142

[141] Paul E. McKenney. "Differential Profiling". In: *Proceedings of the 3rd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. MASCOTS '95. 1995. 8, 36

[142] Nimrod Megiddo and Dharmendra S. Modha. "ARC: A Self-Tuning, Low Overhead Replacement Cache." In: *FAST*. Vol. 3. 2003. 2003. 86

[143] Sanyam Mehta and Pen-Chung Yew. "Improving Compiler Scalability: Optimizing Large Programs at Small Price". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. 2015. 122

[144] Arnaldo Carvalho de Melo. "The new linux'perf'tools". In: *Slides from Linux Kongress*. Vol. 18. 2010. 4, 5, 36

[145] Xiaozhu Meng and Barton P. Miller. "Binary Code is Not Easy". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. 2016. 6, 31

[146] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. "The Paradyn parallel performance measurement tool". In: *Computer* 28.11 (Nov. 1995). ISSN: 0018-9162. 6, 31

[147] Barton P. Miller and Jong-Deok Choi. "A Mechanism for Efficient Debugging of Parallel Programs". In: *SIGPLAN Not.* 23.7 (June 1988). ISSN: 0362-1340. 32, 94

[148] Barton P. Miller, Morgan Clark, Jeff Hollingsworth, Steven Kierstead, Alvin Sek-See. Lim, and Timothy Torzewski. "IPS-2: the second generation of a parallel program measurement system". In: *IEEE Transactions on Parallel and Distributed Systems* 1.2 (Apr. 1990). ISSN: 1045-9219. 44, 136

[149] Barton P. Miller and Cui-Qing Yang. "IPS: An Interactive and Automatic Performance Measurement Tool for Parallel and Distributed Programs." In: vol. 7. Jan. 1987. 44, 136

[150] James G. Mitchell. *The design and construction of flexible and efficient interactive programming systems.* Tech. rep. CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1970. 13

[151] Tipp Moseley, Neil Vachharajani, and William Jalby. "Hardware Performance Monitoring for the Rest of Us: A Position and Survey". In: *Network and Parallel Computing*. Ed. by Erik Altman and Weisong Shi. 2011. 4

[152] Nicholas Nethercote. *Dynamic binary analysis and instrumentation*. Tech. rep. University of Cambridge, Computer Laboratory, 2004. 5, 29

[153] Nicholas Nethercote and Alan Mycroft. "Redux: A dynamic dataflow tracer". In: *Electronic Notes in Theoretical Computer Science* 89.2 (2003). 32, 94

[154] Nicholas Nethercote and Julian Seward. "How to Shadow Every Byte of Memory Used by a Program". In: *Proceedings of the 3rd International Conference on Virtual Execution Environments*. VEE '07. 2007. 27

[155] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *SIGPLAN Not.* 42.6 (June 2007). ISSN: 0362-1340. 6, 28, 30, 33

[156] Catherine Mills Olschanowsky, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. "PSnAP: Accurate Synthetic Address Streams through Memory Profiles". In: *Languages and Compilers for Parallel Computing*. Ed. by Guang R. Gao, Lori L. Pollock, John Cavazos, and Xiaoming Li. 2010. 136

[157] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. "BOLT: A Practical Binary Optimizer for Data Centers and Beyond". In: *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*. CGO 2019. 2019. 31

[158] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998. 122

[159] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. "PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs". In: *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '10. 2010. 32

[160] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. "GRAPHITE: Polyhedral analyses and optimizations for GCC". In: *Proceedings of the 2006 GCC Developers Summit*. 2006. 6, 90, 91

[161] Sebastian Pop, Albert Cohen, and Georges-André Silber. "Induction Variable Analysis with Delayed Abstractions". In: *Proceedings of the First International Conference on High Performance Embedded Architectures and Compilers*. HiPEAC'05. 2005. 10, 93, 95

[162] Louis-Noël Pouchet. *the PoCC polyhedral compiler collection*. URL: `http://pocc.sourceforge.net` (visited on 18/06/2019). 6, 90, 94, 122, 130

[163] Benoît Pradelle, Benoît Meister, Muthu Baskaran, Athanasios Konstantinidis, Thomas Henretty, and Richard Lethin. "Scalable hierarchical polyhedral compilation". In: *Parallel Processing (ICPP), 2016 45th International Conference on*. IEEE. 2016. 122

[164] The LLVM Project. *Clang: a C language family frontend for LLVM*. URL: `https://clang.llvm.org` (visited on 10/07/2019). 6

[165] The LLVM Project. *The LLVM Compiler Infrastructure*. URL: `https://www.llvm.org` (visited on 10/07/2019). 30, 43

[166] QEMU. *The QEMU Project*. 2019. URL: `https://www.qemu.org` (visited on 15/03/2019). 12

[167] Ganesan Ramalingam. "On Loops, Dominators, and Dominance Frontiers". In: *ACM Trans. Program. Lang. Syst.* 24.5 (Sept. 2002). ISSN: 0164-0925. 97

[168] James Reinders. *Processor Tracing*. URL: `https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing` (visited on 08/07/2019). 33

[169] James Reinders. "Vtune performance analyzer essentials: Measurement and tuning techniques for software developers". In: Engineer to Engineer 1.2 (2005). 4, 5, 36

[170] Thomas Reps, Junghee Lim, Aditya Thakur, Gogul Balakrishnan, and Akash Lal. "There's Plenty of Room at the Bottom: Analyzing and Verifying Machine Code". In: *Computer Aided Verification*. Ed. by Tayssir Touili, Byron Cook, and Paul Jackson. 2010. 6

[171] Gabriel Rodríguez, José M. Andión, Mahmut T. Kandemir, and Juan Touriño. "Trace-based Affine Reconstruction of Codes". In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. CGO '16. 2016. 10, 90, 93, 95, 136, 138

[172] Thomas Röhl, Jan Eitzinger, Georg Hager, and Gerhard Wellein. "LIKWID Monitoring Stack: A flexible framework enabling job specific performance monitoring for the masses". In: *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. Sept. 2017. 4, 36

[173] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. "Instrumentation and optimization of Win32/Intel executables using Etch". In: *Proceedings of the USENIX Windows NT Workshop*. Vol. 1997. 1997. 31

[174] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. "Global Value Numbers and Redundant Computations". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. 1988. 18

[175] Saul Rosen. *Lectures on the Measurement and Evaluation of the Performance of Computing Systems*. Vol. 23. page 47. SIAM, 1976. 33

[176] Karl Rupp et al. *Microprocessor Trend Data*. Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten. New plot and data collected for 2010-2015 by K. Rupp. URL: `https://github.com/karlrupp/microprocessor-trend-data` (visited on 05/07/2019). 2

[177] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. "Hybrid Analysis: Static & Dynamic Memory Reference Analysis". In: *International Journal of Parallel Programming* 31.4 (Aug. 2003). ISSN: 1573-7640. 8, 137

[178] Diogo N. Sampaio, Louis-Noël Pouchet, and Fabrice Rastello. "Simplification and Runtime Resolution of Data Dependence Constraints for Loop Transformations". In: *Proceedings of the International Conference on Supercomputing*. ICS '17. 2017. 8, 137

[179] Daniel Sanchez and Christos Kozyrakis. "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ISCA '13. 2013. 46, 49

[180] A. Sandberg, N. Nikoleris, T. E. Carlson, E. Hagersten, S. Kaxiras, and D. Black-Schaffer. "Full Speed Ahead: Detailed Architectural Simulation at Near-Native Speed". In: *2015 IEEE International Symposium on Workload Characterization*. Oct. 2015. 5, 46

[181] Yukinori Sato, Yasushi Inoguchi, and Tadao Nakamura. "On-the-fly Detection of Precise Loop Nests Across Procedures on a Dynamic Binary Translation System". In: *Proceedings of the 8th ACM International Conference on Computing Frontiers*. CF '11. 2011. 137

[182] Markus Schordan, Pei-Hung Lin, Dan Quinlan, and Louis-Noël Pouchet. "Verification of polyhedral optimizations with constant loop bounds in finite state space computations". In: *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer. 2014. 138

[183] Manuel Selva, Fabian Gruber, Diogo Sampaio, Christophe Guillon, Antoine Moynault, Louis-Noël Pouchet, and Fabrice Rastello. "Building a Polyhedral Representation from an Instrumented Execution: Making Dynamic Analyses of non-Affine Programs Scalable". In: *ACM Transactions on Architecture and Code Optimization*. TACO '19 22.2 (2019). (to be published in 2019). 143

[184] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: A Concolic Unit Testing Engine for C". In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-13. 2005. 32

[185] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. "AddressSanitizer: A Fast Address Sanity Checker." In: *USENIX Annual Technical Conference*. 2012. 5, 33

[186] John Paul Shen and Mikko H. Lipasti. *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013. 38

[187] Sameer S. Shende and Allen D. Malony. "The Tau Parallel Performance System". In: *The International Journal of High Performance Computing Applications* 20.2 (2006). eprint: https://doi.org/10.1177/1094342006064482. 44

[188] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. "Automatically Characterizing Large Scale Program Behavior". In: *SIGOPS Oper. Syst. Rev.* 36.5 (Oct. 2002). ISSN: 0163-5980. 5, 46

[189] Bruno da Silva, An Braeken, Erik H. D'Hollander, and Abdellah Touhafi. "Performance Modeling for FPGAs: Extending the Roofline Model with High-level Synthesis Tools". In: *Int. J. Reconfig. Comput.* 2013 (Jan. 2013). ISSN: 1687-7195. 41

[190] Andreas Simbürger, Sven Apel, Armin Größlinger, and Christian Lengauer. "PolyJIT: Polyhedral Optimization Just in Time". In: *International Journal of Parallel Programming* (Aug. 2018). ISSN: 1573-7640. 8, 9, 136

[191] Andreas Simbürger and Armin Größlinger. "On the Variety of Static Control Parts in Real-World Programs: from Affine via Multi-dimensional to Polynomial and Just-in-Time". In: *Proc. of the 4th Inter. Workshop on Polyhedral Compilation Techniques, Vienna, Austria*. 2014. 130

[192] Michael D. Smith. "Overcoming the Challenges to Feedback-directed Optimization (Keynote Talk)". In: *SIGPLAN Not.* 35.7 (Jan. 2000). ISSN: 0362-1340. 5, 137

[193] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. "BitBlaze: A New Approach to Computer Security via Binary Analysis". In: *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper*. Dec. 2008. 30

[194] Sony Corporation and LLVM Project. *LLVM Machine Code Analyzer*. URL: https://llvm.org/docs/CommandGuide/llvm-mca.html (visited on 05/07/2019). 43

[195] Olalekan A Sopeju, Martin Burtscher, Ashay Rane, and James Browne. "AutoSCOPE: Automatic Suggestions for Code Optimizations Using PerfExpert". In: *Proceedings of the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications* (July 2011). 5, 45

[196] Amitabh Srivastava and Alan Eustace. "ATOM: A System for Building Customized Program Analysis Tools". In: *SIGPLAN Not.* 39.4 (Apr. 2004). ISSN: 0362-1340. 6, 31

[197] Michelle Mills Strout, Geri Georg, and Catherine Olschanowsky. "Set and Relation Manipulation for the Sparse Polyhedral Framework". In: *Languages and Compilers for Parallel Computing*. Ed. by Hironori Kasahara and Keiji Kimura. 2013. 8

[198] Aravind Sukumaran-Rajam. "Beyond the Realm of the Polyhedral Model: Combining Speculative Program Parallelization with Polyhedral Compilation". Theses. Université de Strasbourg, Nov. 2015. URL: https://hal.inria.fr/tel-01251748. 93

[199] Aravind Sukumaran-Rajam and Philippe Clauss. "The Polyhedral Model of Nonlinear Loops". In: *ACM Trans. Archit. Code Optim.* 12.4 (Dec. 2015). ISSN: 1544-3566. 10, 90, 93, 137

[200] William N. Sumner, Yunhui Zheng, Dasarath Weeratunge, and Xiangyu Zhang. "Precise Calling Context Encoding". In: *IEEE Transactions on Software Engineering* 38.5 (Sept. 2012). ISSN: 0098-5589. 137

[201] Herb Sutter. "The free lunch is over: A fundamental turn toward concurrency in software". In: *Dr. Dobb's journal* 30.3 (2005). 2

[202] Tarek M. Taha and Scott Wills. "An Instruction Throughput Model of Superscalar Processors". In: *IEEE Transactions on Computers* 57.3 (Mar. 2008). ISSN: 0018-9340. 5, 46

[203] Nathan R. Tallent, John M. Mellor-Crummey, and Michael W. Fagan. "Binary Analysis for Measurement and Attribution of Program Performance". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. 2009. 44, 142

[204] O. Tange. "GNU Parallel - The Command-Line Power Tool". In: *The USENIX Magazine* 36.1 (Feb. 2011).

[205] The DWARF Standards Committee. *The DWARF Debugging Information Format Standard Version 5*. Feb. 2017. URL: `http://dwarfstd.org/Dwarf5Std.php`. 21

[206] The TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2*. May 1995. URL: `https://refspecs.linuxbase.org/elf/elf.pdf`. 19

[207] Robert Marco Tomasulo. "An Efficient Algorithm for Exploiting Multiple Arithmetic Units". In: *IBM Journal of Research and Development* 11.1 (Jan. 1967). ISSN: 0018-8646. 39

[208] Craig Topper. *Update to the LLVM scheduling model for Intel Sandy Bridge, Haswell, Broadwell, and Skylake processors*. Mar. 2018. URL: `https://github.com/llvm/llvm-project/commit/cdfcf8ecda8065fda495d73ed16277668b3b56dc` (visited on 22/08/2019). 47

[209] Georgios Tournavitis and Björn Franke. "Semi-automatic Extraction and Exploitation of Hierarchical Pipeline Parallelism Using Profiling Information". In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT '10. 2010. 90, 136

[210] Jan Treibig and Georg Hager. "Introducing a performance model for bandwidth-limited loop kernels". In: *International Conference on Parallel Processing and Applied Mathematics*. Springer. 2009. 41, 54

[211] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta. "GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation". In: *GCC Research Opportunities Workshop (GROW'10)*. Jan. 2010. 6, 90

[212] *Update to the LLVM scheduling model for Intel Sandy Bridge processors*. June 2019. URL: `https://github.com/llvm/llvm-project/commit/8a32ca381d1ecbb04b456a109bcb4f4ab846380b` (visited on 10/08/2019). 37

[213] Robert A. Van Engelen. "Efficient symbolic analysis for optimizing compilers". In: *International Conference on Compiler Construction*. Springer. 2001. 10, 93, 95

[214] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. "The Paralax Infrastructure: Automatic Parallelization with a Helping Hand". In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT '10. 2010. 90, 136

[215] Rajeshwar Vanka and James Tuck. "Efficient and Accurate Data Dependence Profiling Using Software Signatures". In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. CGO '12. Signature (kind of like a bloom filter) based approximate detection. Multi-threaded. 2012. 32, 33

[216] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. "Schedule trees". In: *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques. Vienna, Austria*. 2014. 94, 101

[217] Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. "Equivalence checking of static affine programs using widening to handle recurrences". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 34.3 (2012). 138

[218] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. "Polyhedral parallel code generation for CUDA". In: *ACM Trans. Archit. Code Optim.* 9.4 (Jan. 2013). ISSN: 1544-3566. 6, 90, 130

[219] Zheng Wang, Georgios Tournavitis, Björn Franke, and Michael F.P. O'boyle. "Integrating profile-driven parallelism detection and machine-learning-based mapping". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 11.1 (2014). 90, 136

[220] Tao Wei, Jian Mao, Wei Zou, and Yu Chen. "A New Algorithm for Identifying Loops in Decompilation". In: *Static Analysis*. Ed. by Hanne Riis Nielson and Gilberto Filé. 2007. 97

[221] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Commun. ACM* 52.4 (Apr. 2009). ISSN: 0001-0782. 36, 41

[222] Michael E. Wolf and Monica S. Lam. "A Data Locality Optimizing Algorithm". In: *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. PLDI '91. 1991. 96

[223] M. Wolfe. "More Iteration Space Tiling". In: *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*. Supercomputing '89. 1989. 7

[224] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling". In: *Proceedings of the 30th Annual International Symposium on Computer Architecture*. ISCA '03. 2003. 5, 46

[225] Liang Xu, Fangqi Sun, and Zhendong Su. *Constructing precise control flow graphs from binaries*. University of California, Davis, Technical Report. 2009. 32

[226] Cui-Qing Yang and Barton P. Miller. "Critical path analysis for the execution of parallel and distributed programs". In: *Proceedings. The 8th International Conference on Distributed Computing Systems*. June 1988. 44

[227] Charles Yount, Harish Patil, Mohammad S. Islam, and Aditya Srikanth. "Graph-matching-based simulation-region selection for multiple binaries". In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Mar. 2015. 32

[228] Matt T. Yourst. "PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator". In: *2007 IEEE International Symposium on Performance Analysis of Systems Software*. Apr. 2007. 46

[229] Qin Zhao, Derek Bruening, and Saman Amarasinghe. "Umbra: Efficient and Scalable Memory Shadowing". In: *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '10. 2010. 27, 32, 33

[230] Wei Zuo, Louis-Noël Pouchet, Andrey Ayupov, Taemin Kim, Chung-Wei Lin, Shinichi Shiraishi, and Deming Chen. "Accurate High-level Modeling and Automated Hardware/Software Co-design for Effective SoC Design Space Exploration". In: *Proceedings of the 54th Annual Design Automation Conference 2017*. DAC '17. 2017. 128

Glossary

μop micro-operation 37, 40, 41, 47–51, 54, 56–62, 64–69, 71–75, 80–85, 87, 144, 146, 147

ABI application binary interface 30, 31

CCT calling context tree 24–26, 103, 104, 137, 144
CFG control flow graph 15, 20, 23, 26, 29, 31, 32, 94, 97–100, 137, 140
CG call graph 20, 22–26, 94, 97, 99, 100, 137, 140, 144
CPU central processing unit iii, v, 2, 4, 5, 8–10, 12–17, 19, 20, 27, 28, 30–34, 36–38, 40, 41, 43, 45–51, 53–58, 60, 61, 64–66, 69, 70, 72, 78, 79, 82, 83, 85–87, 140, 141, 144
CQA Code Quality Analyzer 45, 48

DAG directed acyclic graph 57
DBT dynamic binary translation 12
DDG Dynamic Dependence Graph 94–96, 122, 124, 125, 131, 134, 136
DVFS dynamic voltage and frequency scaling 9, 36
DWARF DWARF Debug Information Format 21, 142
dynamic IIV dynamic interprocedural iteration vector 94, 95, 100, 101, 103–106, 147

ECM execution cache model 41–43, 54, 86
ELF Executable and Linkable Format 19, 21

FDO Feedback-Directed Optimization 5, 137

GPU graphics processing unit 36, 43

helper function A short function written in C used to implement complex instructions in QEMU 15

IACA Intel Architecture Code Analyzer 41–43, 48
IPC instructions per cycle iii, v, 41, 45, 57–63, 68, 80, 83, 144
IR intermediate representation v, 5, 6, 10, 12–20, 22, 30, 31, 34, 47, 92, 94, 130, 137, 140–142
ISA instruction set architecture 12, 16, 17, 30, 31, 37–41, 47, 61, 66
IV iteration vector 101, 104–113, 121

JIT just-in-time compiler 13, 30, 34

LCCT loop-call context tree 94
LLVM Low-level Virtual Machine 43, 47, 91, 92, 167
llvm-mca LLVM Machine Code Analyzer 43, 45
LP linear program 57, 62, 63, 65, 141
LRU least-recently-used 86

MILP mixed integer linear program 87
MPC micro-operations per cycle 57–63, 68, 83, 144

OS operating system 12, 16

OSACA Open-Source Architecture Code Analyzer 43, 45, 48

PLT procedure linkage table 34

RAR read-after-read 27
RAW read-after-write 27
RI représentation intermédiaire iii
ROB re-order buffer 39–41, 46, 47, 49, 50, 57, 58, 65, 71, 72, 75, 141

SCC stronlgy connected component 97–99, 101
SCEV scalar evolution expression 10, 95
SSA static single assignment 18

TAU Tuning and Analysis Utilities 44
TB translation block 13–16, 18, 19, 22, 30, 34, 97, 140
TCG tiny code generator 12–20, 30, 31, 34, 140, 168
TPI TCG plugin infrastructure 12, 16, 18, 19, 28, 30–32, 34, 36, 94, 140

WAR write-after-read 27
WAW write-after-write 27