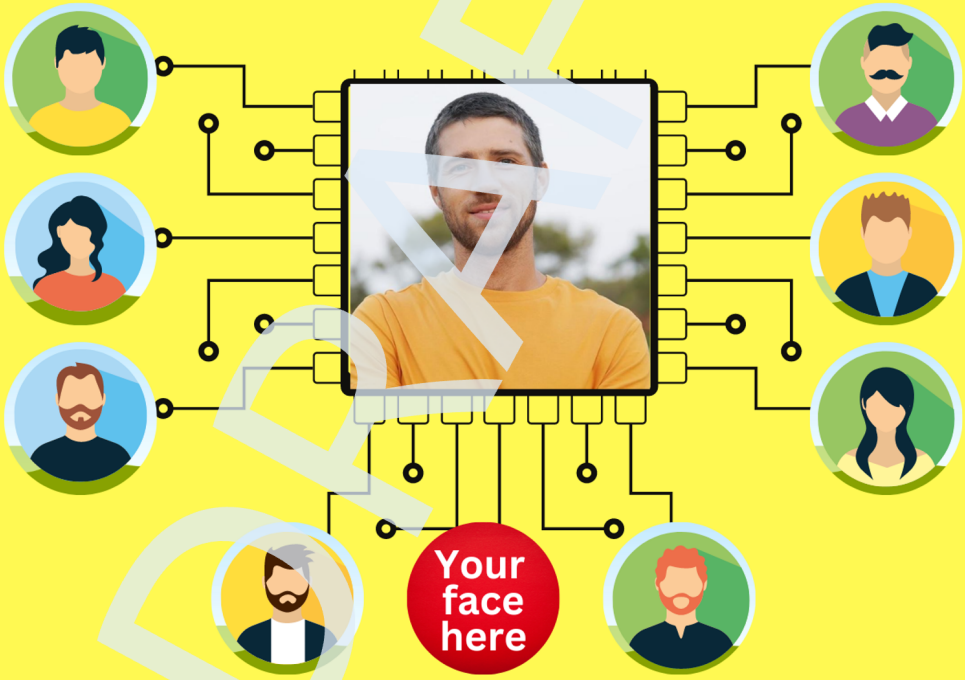
Performance Analysis and Tuning on Modern CPUs

Software Developers guide for discovering and implementing HW-specific optimizations

</>

Curated by industry experts



Second edition

Notices

Responsibility. Knowledge and best practice in the field of engineering and software development are constantly changing. Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods, they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the author nor contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operations of any methods, products, instructions, or ideas contained in the material herein.

Trademarks. Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. Intel, Intel Core, Intel Xeon, Intel Pentium, Intel Vtune, and Intel Advisor are trademarks of Intel Corporation in the U.S. and/or other countries. AMD is a trademark of Advanced Micro Devices Corporation in the U.S. and/or other countries. ARM is a trademark of Arm Limited (or its subsidiaries) in the U.S. and/or elsewhere. Readers, however, should contact the appropriate companies for complete information regarding trademarks and registration.

Affiliation. At the time of writing, the book's primary author (Denis Bakhvalov) is an employee of Intel Corporation. All information presented in the book is not an official position of the aforementioned company, but rather is an individual knowledge and opinions of the author. The primary author did not receive any financial sponsorship from Intel Corporation for writing this book.

Advertisement. This book does not advertise any software, hardware, or any other product.

Copyright

Preface

About The Author

Denis Bakhvalov is a senior developer at Intel, where he works on C++ compiler projects that aim at generating optimal code for a variety of different architectures. Performance engineering and compilers were always among the primary interests for him. Denis has started his career as a software developer in 2008 and has since worked in multiple areas, including developing desktop applications, embedded, performance analysis, and compiler development. In 2016 Denis started his easyperf.net blog, where he writes about performance analysis and tuning, C/C++ compilers, and CPU microarchitecture. Denis is a big proponent of an active lifestyle, which he practices in his free time. You can find him playing soccer, tennis, running, and playing chess. Besides that, Denis is a father of 2 beautiful daughters.

Contacts:

- Email: dendibakh@gmail.com
- Twitter: @dendibakh
- LinkedIn: @dendibakh

From The Author

I started this book with a simple goal: educate software developers to better understand their applications' performance on modern hardware. I know how confusing this topic might be for a beginner or even for an experienced developer. This confusion mostly happens to developers that don't have prior occasions of working on performance-related tasks. And that's fine since every expert was once a beginner.

I remember the days when I was starting with performance analysis. I was staring at unfamiliar metrics trying to match the data that didn't match. And I was baffled. It took me years until it finally "clicked", and all pieces of the puzzle came together. At the time, the only good sources of information were software developer manuals, which are not what mainstream developers like to read. So I decided to write this book, which will hopefully make it easier for developers to learn performance analysis concepts.

Developers who consider themselves beginners in performance analysis can start from the beginning of the book and read sequentially, chapter by chapter. Chapters 2-4 give developers a minimal set of knowledge required by later chapters. Readers already familiar with these concepts may choose to skip those. Additionally, this book can be used as a reference or a checklist for optimizing SW applications. Developers can use chapters 7-11 as a source of ideas for tuning their code.

Target Audience

This book will be primarily useful for software developers who work with performance-critical applications and do low-level optimizations. To name just a few areas: High-Performance Computing (HPC), Game Development, data-center applications (like Facebook, Google, etc.), High-Frequency Trading. But the scope of the book is not limited to the mentioned industries. This book will be useful for any developer who wants to understand the performance of their application better and know how it can be diagnosed and improved. The author hopes that the material presented in this book will help readers develop new skills that can be applied in their daily work.

Readers are expected to have a minimal background in C/C++ programming languages to understand the book's examples. The ability to read basic x86 assembly is desired but is not a strict requirement. The author also expects familiarity with basic concepts of computer architecture and operating systems like central processor, memory, process, thread, virtual and physical memory, context switch, etc. If any of the mentioned terms are new to you, I suggest studying this material first.

Acknowledgments

Huge thanks to Mark E. Dawson, Jr. for his help writing several sections of this book: "Optimizing For DTLB" (Section 8.4), "Optimizing for ITLB" (Section 11.8), "Cache Warming" (Section 12.3.2), System Tuning (Section 12.5),

Section 13.1 about performance scaling and overhead of multithreaded applications, Section 13.5 about using COZ profiler, Section 13.6 about eBPF, "Detecting Coherence Issues" (Section 13.7). Mark is a recognized expert in the High-Frequency Trading industry. Mark was kind enough to share his expertise and feedback at different stages of this book's writing.

Next, I want to thank Sridhar Lakshmanamurthy, who authored the major part of Chapter 3 about CPU microarchitecture. Sridhar has spent decades working at Intel, and he is a veteran of the semiconductor industry.

Big thanks to Nadav Rotem, the original author of the vectorization framework in the LLVM compiler, who helped me write the Section 9.4 about vectorization.

Clément Grégoire authored a Section 9.4.2.5 about ISPC compiler. Clément has an extensive background in the game development industry. His comments and feedback helped address in the book some of the challenges in the game development industry.

This book wouldn't have come out of the draft without its reviewers: Dick Sites, Wojciech Muła, Thomas Dullien, Matt Fleming, Daniel Lemire, Ahmad Yasin, Michele Adduci, Clément Grégoire, Arun S. Kumar, Surya Narayanan, Alex Blewitt, Nadav Rotem, Alexander Yermolovich, Suchakrapani Datt Sharma, Renat Idrisov, Sean Heelan, Jumana Mundichipparakkal, Todd Lipcon, Rajiv Chauhan, Shay Morag, and others.

Also, I would like to thank the whole performance community for countless blog articles and papers. I was able to learn a lot from reading blogs by Travis Downs, Daniel Lemire, Andi Kleen, Agner Fog, Bruce Dawson, Brendan Gregg, and many others. I stand on the shoulders of giants, and the success of this book should not be attributed only to myself. This book is my way to thank and give back to the whole community.

Last but not least, thanks to my family, who were patient enough to tolerate me missing weekend trips and evening walks. Without their support, I wouldn't have finished this book.

Table Of Contents

1 Introduction

They say, "performance is king". It was true a decade ago, and it certainly is now. According to [domo.com, 2017], in 2017, the world has been creating 2.5 quintillions[1] bytes of data every day, and as predicted in [statista.com, 2018], this number is growing 25% per year. In our increasingly data-centric world, the growth of information exchange fuels the need for both faster software (SW) and faster hardware (HW). Fair to say, the data growth puts demand not only on computing power but also on storage and network systems.

In the PC era,[2] developers usually were programming directly on top of the operating system, with possibly a few libraries in between. As the world moved to the cloud era, the SW stack got deeper and more complex. The top layer of the stack on which most developers are working has moved further away from the HW. Those additional layers abstract away the actual HW, which allows using new types of accelerators for emerging workloads. However, the negative side of such evolution is that developers of modern applications have less affinity to the actual HW on which their SW is running.

Software programmers have had an "easy ride" for decades, thanks to Moore's law. It used to be the case that some SW vendors preferred to wait for a new generation of HW to speed up their application and did not spend human resources on making improvements in their code. By looking at Figure 1, we can see that single-threaded performance growth is slowing down. Single-threaded performance is a performance of a single HW thread inside a CPU core when measured in isolation.

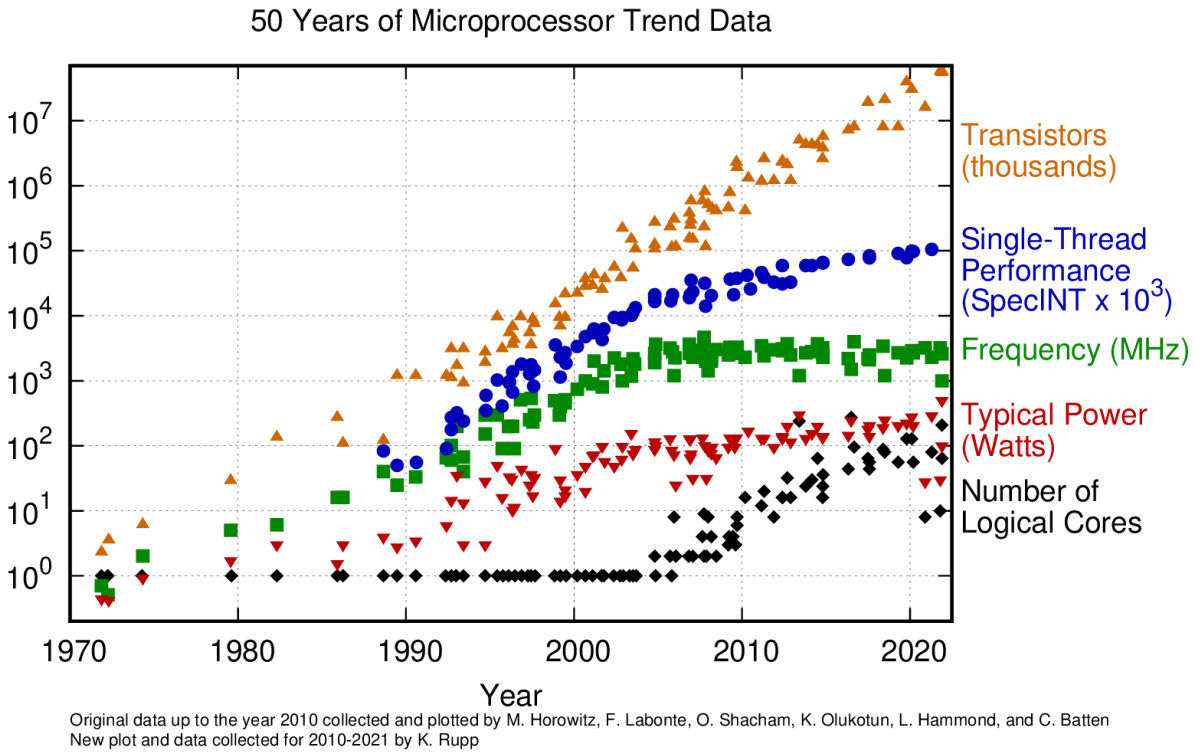


Figure 1: 50 Years of Microprocessor Trend Data. © *Image by K. Rupp via karlrupp.net*

When it's no longer the case that each HW generation provides a significant performance boost [Leiserson et al., 2020], we must start paying more attention to how fast our code runs. When seeking ways to improve performance, developers should not rely on HW. Instead, they should start optimizing the code of their applications.

> "Software today is massively inefficient; it's become prime time again for software programmers to get really good at optimization." - Marc Andreessen, the US entrepreneur and investor (a16z Podcast, 2020)

[1] Quintillion is a thousand raised to the power of six (10^{18}).
[2] From the late 1990s to the late 2000s where personal computers where dominating the market of computing devices.

Personal Experience: While working at Intel, I hear the same story from time to time: when Intel clients experience slowness in their application, they immediately and unconsciously start blaming Intel for having slow CPUs. But when Intel sends one of our performance ninjas to work with them and help them improve their application, it is not unusual that they help speed it up by a factor of 2x, sometimes even 10x.

Reaching high-level performance is challenging and usually requires substantial efforts, but hopefully, this book will give you the tools to help you achieve it.

1.1 Why Do We Still Need Performance Tuning?

Modern CPUs are getting more and more cores each year. As of the end of 2019, you can buy a high-end server processor which will have more than 100 logical cores. This is very impressive, but that doesn't mean we don't have to care about performance anymore. Very often, application performance might not get better with more CPU cores. The performance of a typical general-purpose multithread application doesn't always scale linearly with the number of CPU cores we assign to the task. Understanding why that happens and possible ways to fix it is critical for the future growth of a product. Not being able to do proper performance analysis and tuning leaves lots of performance and money on the table and can kill the product.

[TODO]: include discussion on "Clean code, horrible performance"?

According to [Leiserson et al., 2020], at least in the near term, a large portion of performance gains for most applications will originate from the SW stack. Sadly, applications do not get optimal performance by default. The paper also provides an excellent example that illustrates the potential for performance improvements that could be done on a source code level. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices are summarized in Table 1. The end result of applying multiple optimizations is a program that runs over 60,000 times faster. The reason for providing this example is not to pick on Python or Java (which are great languages), but rather to break beliefs that software has "good enough" performance by default.

Table 1: Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices running on a dual-socket Intel Xeon E5-2666 v3 system with a total of 60 GB of memory. From [Leiserson et al., 2020].

Version	Implementation	Absolute speedup	Relative speedup
1	Python	1	—
2	Java	11	10.8
3	C	47	4.4
4	Parallel loops	366	7.8
5	Parallel divide and conquer	6,727	18.4
6	plus vectorization	23,224	3.5
7	plus AVX intrinsics	62,806	2.7

Here are some of the most important factors that prevent systems from achieving optimal performance by default:

1. **CPU limitations**: it's so tempting to ask: "*Why doesn't HW solve all our problems?*" Modern CPUs execute instructions at incredible speed and are getting better with every generation. But still, they cannot do much if instructions that are used to perform the job are not optimal or even redundant. Processors cannot magically transform suboptimal code into something that performs better. For example, if we implement a sorting routine using BubbleSort algorithm, a CPU will not make any attempts to recognize it and use the better alternatives, for example, QuickSort. It will blindly execute whatever it was told to do.
2. **Compiler limitations**: *"But isn't it what compilers are supposed to do? Why don't compilers solve all our problems?"* Indeed, compilers are amazingly smart nowadays, but can still generate suboptimal code. Compilers are great at eliminating redundant work, but when it comes to making more complex decisions like function inlining, loop unrolling, etc. they may not generate the best possible code. For example, there is no binary "yes" or "no" answer to the question of whether a compiler should always inline a function into the place where it's called. It usually depends on many factors which a compiler should take into account.

Often, compilers rely on complex cost models and heuristics, which may not work for every possible scenario. Additionally, compilers cannot perform optimizations unless they are certain it is safe to do so, and it does not affect the correctness of the resulting machine code. It may be very difficult for compiler developers to ensure that a particular optimization will generate correct code under all possible circumstances, so they often have to be conservative and refrain from doing some optimizations. Finally, compilers generally do not transform data structures used by the program, which are also crucial in terms of performance.

3. **Algorithmic complexity analysis limitations**: developers are frequently overly obsessed with complexity analysis of the algorithms, which leads them to choose the popular algorithm with the optimal algorithmic complexity, even though it may not be the most efficient for a given problem. Considering two sorting algorithms, InsertionSort and QuickSort, the latter clearly wins in terms of Big O notation for the average case: InsertionSort is $O(N^2)$ while QuickSort is only O(N log N). Yet for relatively small sizes of `N` (up to 50 elements), InsertionSort outperforms QuickSort. Complexity analysis cannot account for all the branch prediction and caching effects of various algorithms, so people just encapsulate them in an implicit constant `C`, which sometimes can make drastic impact on performance. Blindly trusting Big O notation without testing on the target workload could lead developers down an incorrect path. So, the best-known algorithm for a certain problem is not necessarily the most performant in practice for every possible input.

Limitations described above leave the room for tuning the performance of our SW to reach its full potential. Broadly speaking, the SW stack includes many layers, e.g., firmware, BIOS, OS, libraries, and the source code of an application. But since most of the lower SW layers are not under our direct control, a major focus will be made on the source code. Another important piece of SW that we will touch on a lot is a compiler. It's possible to obtain attractive speedups by making the compiler generate the desired machine code through various hints. You will find many such examples throughout the book.

> **Personal Experience:** To successfully implement the needed improvements in your application, you don't have to be a compiler expert. Based on my experience, at least 90% of all transformations can be done at a source code level without the need to dig down into compiler sources. Although, understanding how the compiler works and how you can make it do what you want is always advantageous in performance-related work.

Also, nowadays, it's essential to enable applications to scale up by distributing them across many cores since single-threaded performance tends to reach a plateau. Such enabling calls for efficient communication between the threads of application, eliminating unnecessary consumption of resources and other issues typical for multi-threaded programs.

It is important to mention that performance gains will not only come from tuning SW. According to [Leiserson et al., 2020], two other major sources of potential speedups in the future are algorithms (especially for new problem domains like machine learning) and streamlined hardware design. Algorithms obviously play a big role in the performance of an application, but we will not cover this topic in this book. We will not be discussing the topic of new hardware designs either since, most of the time, SW developers have to deal with existing HW. However, understanding modern CPU design is important for optimizing applications.

> "During the post-Moore era, it will become ever more important to make code run fast and, in particular, to tailor it to the hardware on which it runs." [Leiserson et al., 2020]

The methodologies in this book focus on squeezing out the last bit of performance from your application. Such transformations can be attributed along rows 6 and 7 in Table 1. The types of improvements that will be discussed are usually not big and often do not exceed 10%. However, do not underestimate the importance of a 10% speedup. It is especially relevant for large distributed applications running in cloud configurations. According to [Hennessy, 2018], in the year 2018, Google spends roughly the same amount of money on actual computing servers that run the cloud as it spends on power and cooling infrastructure. Energy efficiency is a very important problem, which can be improved by optimizing SW.

> "At such scale, understanding performance characteristics becomes critical – even small improvements in performance or utilization can translate into immense cost savings." [Kanev et al., 2015]

1.2 Who Needs Performance Tuning?

Performance engineering does not need to be justified much in industries like High-Performance Computing (HPC), Cloud Services, High-Frequency Trading (HFT), Game Development, and other performance-critical areas. For instance, Google reported that a 2% slower search caused 2% fewer searches per user.[3] For Yahoo! 400 milliseconds faster page load caused 5-9% more traffic.[4] In the game of big numbers, small improvements can make a significant impact. Such examples prove that the slower the service works, the fewer people will use it.

There is a famous quote: "Premature optimization is the root of all evil". But the opposite is often true as well. Postponed performance engineering work may be too late and cause as much evil as premature optimization. For developers working with performance-critical projects, it is crucial to know how underlying HW works. In such industries, it is a fail-from-the-start when a program is being developed without HW focus. Performance characteristics of a software must be a first-class-citizen along with correctness and security starting from day 1. ClickHouse DB is an example of a successful software product that was built around a small but very efficient kernel.

Interestingly, performance engineering is not only needed in the aforementioned areas. Nowadays, it is also required in the field of general-purpose applications and services. Many tools that we use every day simply would not exist if they failed to meet their performance requirements. For example, Visual C++ IntelliSense[5] features that are integrated into Microsoft Visual Studio IDE have very tight performance constraints. For IntelliSense autocomplete feature to work, they have to parse the entire source codebase in the order of milliseconds.[6] Nobody will use source code editor if it takes it several seconds to suggest autocomplete options. Such a feature has to be very responsive and provide valid continuations as the user types new code. The success of similar applications can only be achieved by designing SW with performance in mind and thoughtful performance engineering.

Sometimes fast tools find use in the areas they were not initially designed for. For example, nowadays, game engines like Unreal[7] and Unity[8] are used in architecture, 3d visualization, film making, and other areas. Because game engines are so performant, they are a natural choice for applications that require 2d and 3d rendering, physics engine, collision detection, sound, animation, etc.

> "Fast tools don't just allow users to accomplish tasks faster; they allow users to accomplish entirely new types of tasks, in entirely new ways." - Nelson Elhage wrote in article[9]on his blog (2020).

I hope it goes without saying that people hate using slow software. Performance characteristics of an application can be a single factor for your customer to switch to a competitor's product. By putting emphasis on performance, you can give your product a competitive advantage.

Performance engineering is important and rewarding work, but it may be very time-consuming. In fact, performance optimization is a never-ending game. There will always be something to optimize. Inevitably, the developer will reach the point of diminishing returns at which further improvement comes at a very high engineering cost and likely will not be worth the efforts. Performance assessment of your application against HW theoretical limits helps understanding the potential headroom for optimizations. Knowing when to stop optimizing is a critical aspect of performance work. Some organizations achieve it by integrating this information into the code review process: source code lines are annotated with the corresponding "cost" metric. Using that data, developers can decide whether improving the performance of a particular piece of code is worth it.

Before starting performance tuning, make sure you have a strong reason to do so. Optimization just for optimization's sake is useless if it doesn't add value to your product. Mindful performance engineering starts with clearly defined performance goals, stating what you are trying to achieve and why you are doing it. Also, you should pick the metrics that you will use to measure whether you reach the goal or not. You can read more on the topic of setting performance goals in [Gregg, 2013] and [Akinshin, 2019].

Nevertheless, it is always great to practice and master the skill of performance analysis and tuning. If you picked up the book for that reason, you are more than welcome to keep on reading.

[3] Slides by Marissa Mayer - https://assets.en.oreilly.com/1/event/29/Keynote Presentation 2.pdf
[4] Slides by Stoyan Stefanov - https://www.slideshare.net/stoyan/dont-make-me-wait-or-building-highperformance-web-applications
[5] Visual C++ IntelliSense - https://docs.microsoft.com/en-us/visualstudio/ide/visual-cpp-intellisense
[6] In fact, it's not possible to parse the entire codebase in the order of milliseconds. Instead, IntelliSense only reconstructs the portions of AST that has been changed. Watch more details on how the Microsoft team achieves this in the video: https://channel9.msdn.com/Blogs/Seth-Juarez/Anders-Hejlsberg-on-Modern-Compiler-Construction.
[7] Unreal Engine - https://www.unrealengine.com.
[8] Unity Engine - https://unity.com/
[9] Reflections on software performance by N. Elhage - https://blog.nelhage.com/post/reflections-on-performance/

1.3 What Is Performance Analysis?

Ever found yourself debating with a coworker about the performance of a certain piece of code? Then you probably know how hard it is to predict which code is going to work the best. With so many moving parts inside modern processors, even a small tweak to the code can trigger significant performance change. That's why the first advice in this book is: *Always Measure*. Many people rely on intuition when they try to optimize their application. And usually, it ends up with random fixes here and there without making any real impact on the performance of the application.

Inexperienced developers often make changes in the source code and hope it will improve the performance of the program. One such example is replacing `i++` with `++i` all over the code base, assuming that the previous value of `i` is not used. In the general case, this change will make no difference to the generated code because every decent optimizing compiler will recognize that the previous value of `i` is not used and will eliminate redundant copies anyway.

Many micro-optimization tricks that circulate around the world were valid in the past, but current compilers have already learned them. Additionally, some people tend to overuse legacy bit-twiddling tricks. One of such examples is using XOR-based swap idiom,[10] while in reality, simple `std::swap` produces faster code. Such accidental changes likely won't improve the performance of the application. Finding the right place to fix should be a result of careful performance analysis, not intuition and guesses.

There are many performance analysis methodologies that may or may not lead you to a discovery. The CPU-specific approaches to performance analysis presented in this book have one thing in common: they are based on collecting certain information about how the program executes. Any change that ends up being made in the source code of the program is driven by analyzing and interpreting collected data.

Locating a performance bottleneck is only half of the engineer's job. The second half is to fix it properly. Sometimes changing one line in the program source code can yield a drastic performance boost. Performance analysis and tuning are all about how to find and fix this line. Missing such opportunities can be a big waste.

1.4 What Is Discussed in this Book?

This book is written to help developers better understand the performance of their application, learn to find inefficiencies, and eliminate them. *Why my hand-written compression algorithm performs two times slower than the conventional one? Why did my change in the function cause performance to drop by half? Customers are complaining about the slowness of my application, where should I start? Have I optimized the program to its full potential? What performance analysis tools are available on my platform? What are the techniques to reduce the number of cache misses and branch mispredictions?* Hopefully, by the end of this book, you will have the answers to those questions.

Here is the outline of what this book contains:

- Chapter 2 discusses how to conduct fair performance experiments and analyze their results. It introduces the best practices for performance testing and comparing results.
- Chapters 3 provides basics of CPU microarchitecture and Chapter 4 covers terminology and metrics used in performance analysis; we recommend you not to skip these chapters even if you think you know this already.
- Chapter 5 explores several of the most popular approaches for doing performance analysis. It explains how profiling techniques work, what runtime data can be collected, and how it can be done.
- Chapter 6 examines features provided by modern CPUs to support and enhance performance analysis. It shows how they work and what problems they can solve.
- Chapter 7 gives an overview of the most popular tools available on major platforms, including Linux, Windows and MacOS, running on x86- and ARM-based processors.
- Chapters 8-11 contain recipes for typical performance problems. These chapters are organized according to the Top-down Microarchitecture Analysis methodology, which is one of the most important concepts of the book. Don't worry if some terms are not yet clear to you, we will cover everything step by step. Chapter 8 (Memory Bound) is about optimizing memory accesses, cache friendly code, memory profiling, huge pages, and a few other techniques. Chapter 9 (Core Bound) is about optimizing computations and explores function inlining, loop optimizations, and vectorization. Chapter 10 (Bad Specualtion) is about branchless programming that is used to avoid frequently mispredicted branches. Chapter 11 (FrontEnd Bound) is about machine code layout optimizations, such as basic block placement, function splitting, profile-guided optimizations and others.

[10] XOR-based swap idiom - https://en.wikipedia.org/wiki/XOR_swap_algorithm

- Chapter 13 contains optimization topics not specifically related to any of the categories covered in the previous four chapters, but are still important enough to find their place in this book. There you will find low-latency techniques, tips on tuning your system for the best performance, faster alternatives to standard library funcitons, and others.
- Chapter 14 discusses techniques for analyzing multithreaded applications. It outlines some of the most important challenges of optimizing the performance of multithreaded applications and the tools that can be used to analyze them. The topic itself is quite big, so the chapter only focuses on HW-specific issues, like "False Sharing".
- Chapter 15 talks about the current and future trends in the world of SW and HW performance. We discuss advances in traditional design of computer systems as well as some innovative ideas.

Examples provided in this book are primarily based on open-source software: Linux as the operating system, the LLVM-based Clang compiler for C and C++ languages, and Linux `perf` as the profiling tool. The reason for such a choice is not only the popularity of the mentioned technologies but also the fact that their source code is open, which allows us to better understand the underlying mechanism of how they work. This is especially useful for learning the concepts presented in this book. We will also sometimes showcase proprietary tools that are "big players" in their areas, for example, Intel® VTune™ Profiler.

1.5 What Is not Discussed in this Book?

System performance depends on different components: CPU, OS, memory, I/O devices, etc. Applications could benefit from tuning various components of the system. In general, engineers should analyze the performance of the whole system. However, the biggest factor in systems performance is its heart, the CPU. This is why this book primarily focuses on performance analysis from a CPU perspective, occasionally touching on OS and memory subsystems.

The scope of the book does not go beyond a single CPU socket, so we will not discuss optimization techniques for distributed, NUMA, and heterogeneous systems. Offloading computations to accelerators (GPU, FPGA, etc.) using solutions like OpenCL and openMP is not discussed in this book.

This book centers around the Intel x86-64 CPU architecture and does not provide specific tuning recipes for AMD, ARM, or RISC-V chips. Nonetheless, many of the principles discussed in this book apply well to those processors. Also, Linux is the OS of choice for this book, but again, for most of the examples we provide, it doesn't matter since the same techniques benefit applications that run on Windows and macOS operating systems.

All the code snippets in this book are written in C, C++, or x86 assembly languages, but to a large degree, ideas from this book can be applied to other languages that are compiled to native code like Rust, Go, and even Fortran. Since this book targets user-mode applications that run close to the hardware, we will not discuss managed environments, e.g., Java.

Finally, the author assumes that readers have full control over the software that they develop, including the choice of libraries and compiler they use. Hence, this book is not about tuning purchased commercial packages, e.g., tuning SQL database queries.

1.6 Exercises

As a supplemental material for this book, we developed a collection of free lab assignments that are available at https://github.com/dendibakh/perf-ninja. Performance Ninja is an online course where you can practice low-level performance analysis and tuning. We offer lab assignments from that repository throughout the book. For example, when you see `perf-ninja::warmup`, this corresponds to the lab assignment that is located in `labs/misc/warmup` folder of the aforementioned repository.

You can solve those assignments on your local machine or submit your code to Github for automated benchmarking. If you choose the latter, follow the instructions on the "Get Started" page of the repo. If you're stuck on a problem, you can check the videos associated with the lab. Those videos explain a possible solution to a problem.

Chapter Summary

- The single-threaded performance of CPUs is not increasing as rapidly as it used to a few decades ago. That is why performance tuning is becoming more important than it has been for the last 40 years. The computing

industry is changing now much more heavily than at any time since the 90s.

- According to [Leiserson et al., 2020], SW tuning will be one of the key drivers for performance gains in the near future. The importance of performance tuning should not be underestimated. For large distributed applications, every small performance improvement results in immense cost savings.
- Software doesn't have optimal performance by default. Certain limitations exist that prevent applications from reaching their full performance potential. Both HW and SW environments have such limitations. CPUs cannot magically speed up slow algorithms. Compilers are far from generating optimal code for every program. Due to HW specifics, the best-known algorithm for a certain problem is not always the most performant. All this leaves room for tuning the performance of our applications.
- For some types of applications, performance is not just a feature. It enables users to solve new kinds of problems in a new way.
- SW optimizations should be backed by strong business needs. Developers should set quantifiable goals and metrics which must be used to measure progress.
- Predicting the performance of a certain piece of code is nearly impossible since there are so many factors that affect the performance of modern platforms. When implementing SW optimizations, developers should not rely on intuition but use careful performance analysis instead.

Part1. Performance Analysis on a Modern CPU

2 Measuring Performance

The first step on the path to understand an application's performance is knowing how to measure it. People attribute performance as being one of the features of an application.[11] But unlike other features, performance is not a boolean property: an application can be very slow, blazingly fast or anywhere in between. This is why it's impossible to answer "yes" or "no" to the question of whether an application has performance.

Performance problems are usually harder to track down and reproduce than most functional issues. Although, sometimes we have to deal with non-deterministic and hard to reproduce performance bugs as well. Often, every run of a program is functionally the same but somewhat different from performance standpoint. For example, when unpacking a zip-file, we get the same result over and over again, which means this operation is reproducible. However, it's impossible to reproduce exactly the same CPU cycle-by-cycle performance profile of this operation.

Anyone ever concerned with performance evaluations likely knows how hard it is sometimes to conduct fair performance measurements and draw accurate conclusions from it. Performance measurements can be very unexpected and counterintuitive. Changing a seemingly unrelated part of the source code can surprise us with a significant impact on program performance. This phenomenon is called measurement bias. Because of the presence of error in measurements, performance analysis requires statistical methods to process them. This topic deserves a whole book just by itself. There are many corner cases and a huge amount of research done in this field. We will not go all the way down this rabbit hole. Instead, we will just focus on high-level ideas and directions to follow.

Conducting fair performance experiments is an essential step towards getting accurate and meaningful results. Designing performance tests and configuring the environment are both important components in the process of evaluating performance. This chapter will give a brief introduction to why modern systems yield noisy performance measurements and what you can do about it. We will touch on the importance of measuring performance in real production deployments.

Not a single long-living product exists without ever having performance regressions. This is especially important for large projects with lots of contributors where changes are coming at a very fast pace. This chapter devotes a few pages discussing the automated process of tracking performance changes in Continuous Integration and Continuous Delivery (CI/CD) systems. We also present general guidance on how to properly collect and analyze performance measurements when developers implement changes in their source codebase.

The end of the chapter describes SW and HW timers that can be used by developers in time-based measurements and common pitfalls when designing and writing a good microbenchmark.

2.1 Noise in Modern Systems

There are many features in HW and SW that are intended to increase performance. But not all of them have deterministic behavior. Let's consider Dynamic Frequency Scaling[12] (DFS): this is a feature that allows a CPU to increase its frequency for a short time interval, making it run significantly faster. However, the CPU can't stay in "overclocked" mode for a long time, so later, it decreases its frequency back to the base value. DFS usually depends a lot on a core temperature, which makes it hard to predict the impact on our experiments.

If we start two runs of the benchmark, one right after another on a "cold" processor,[13] the first run could possibly work for some time in "overclocked" mode and then decrease its frequency back to the base level. However, it's possible that the second run might not have this advantage and will operate at the base frequency without entering "turbo mode". Even though we run the exact same version of the program two times, the environment in which they run is not the same. Figure 2 shows a situation where dynamic frequency scaling can cause variance in measurements. As you can see, the first run is 1 second faster than the second one due to the fact that it was running on a higher

[11] Blog post by Nelson Elhage "Reflections on software performance": https://blog.nelhage.com/post/reflections-on-performance/.
[12] Dynamic Frequency Scaling - https://en.wikipedia.org/wiki/Dynamic_frequency_scaling.
[13] By cold processor, we mean the CPU that stayed in an idle mode for a while, allowing it to cool down its temperature.

frequency. Such a scenario can frequently happen when benchmarking on laptops since they have limited heat dissipation.

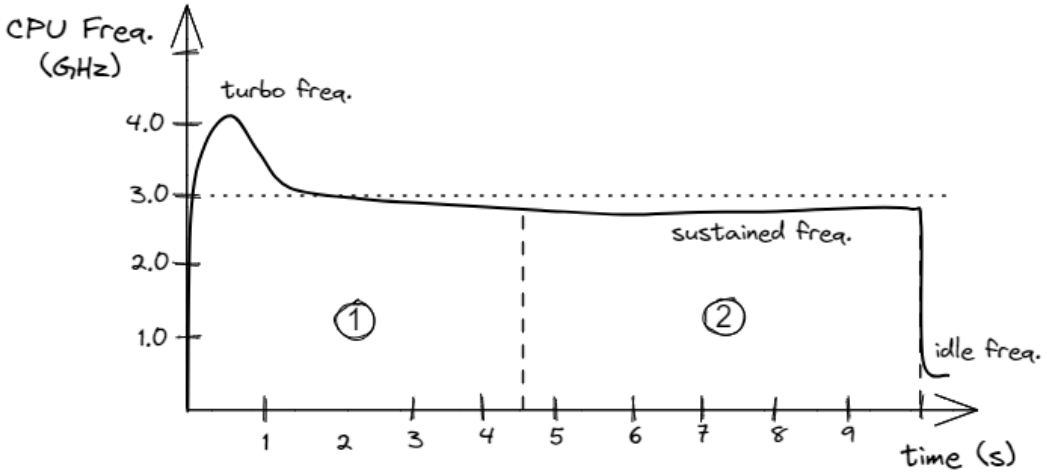


Figure 2: Variance in performance caused by frequency scaling: the first run is 1 second faster than the second.

Frequency Scaling is a HW feature, but variations in measurements might also come from SW features. Let's consider the example of a filesystem cache. If we benchmark an application that does lots of file manipulation, e.g. `git status` command, the filesystem can play a big role in performance. When the first iteration of the benchmark runs, the required entries in the filesystem cache could be missing. However, the filesystem cache will be warmed-up when running the same benchmark the second time, making it noticeably faster than the first run.

Unfortunately, measurement bias does not only come from environment configuration. [Mytkowicz et al., 2009] paper demonstrates that UNIX environment size (i.e., the total number of bytes required to store the environment variables) and link order (the order of object files that are given to the linker) can affect performance in unpredictable ways. Moreover, there are numerous other ways of affecting memory layout and potentially affecting performance measurements. One approach to enable statistically sound performance analysis of software on modern architectures was presented in [Curtsinger & Berger, 2013]. This work shows that it's possible to eliminate measurement bias that comes from memory layout by efficiently and repeatedly randomizing the placement of code, stack, and heap objects at runtime. Sadly, these ideas didn't go much further, and right now, this project is almost abandoned.

Remember that even running a task manager tool, like Linux top, can affect measurements since a CPU core will be activated and assigned to it. This might affect the frequency of the core that is running the actual benchmark.

Having consistent measurements requires running all iterations of the benchmark with the same conditions. However, it is not possible to replicate the exact same environment and eliminate bias completely: there could be different temperature conditions, power delivery spikes, neighbor processes running, etc. Chasing all potential sources of noise and variation in the system can be a never-ending story. Sometimes it cannot be achieved, for example, when benchmarking a large distributed cloud service.

So, eliminating non-determinism in a system is helpful for well-defined, stable performance tests, e.g., microbenchmarks. For instance, when you implement a code change and want to know the relative speedup ratio by benchmarking two different versions of the same program. This is a scenario where you can control most of the variables in the benchmark, including its input, environment configuration, etc. In this situation, eliminating non-determinism in a system helps to get a more consistent and accurate comparison. After finishing with local testing, remember to verify projected performance improvements are mirrored in real-world measurements. Readers can find some examples of features that can bring noise into performance measurements and how to disable them in Appendix A. Also, there are tools that can set up the environment to ensure benchmarking results with a low variance; one such tool is temci[14].

It is not recommended to eliminate system non-deterministic behavior when estimating real-world performance improvements. Engineers should try to replicate the target system configuration, which they are optimizing for.

[14] Temci - https://github.com/parttimenerd/temci.

Introducing any artificial tuning to the system under test will diverge results from what users of your service will see in practice. Also, any performance analysis work, including profiling (see Section 5.4), should be done on a system that is configured similar to what will be used in a real deployment.

Finally, it's important to keep in mind that even if a particular HW or SW feature has non-deterministic behavior, that doesn't mean it is considered harmful. It could give an inconsistent result, but it is designed to improve the overall performance of the system. Disabling such a feature might reduce the noise in microbenchmarks but make the whole suite run longer. This might be especially important for CI/CD performance testing when there are time limits for how long it should take to run the whole benchmark suite.

2.2 Measuring Performance in Production

When an application runs on shared infrastructure (typical in a public cloud), there usually will be other workloads from other customers running on the same servers. With technologies like virtualization and containers becoming more popular, public cloud providers try to fully utilize the capacity of their servers. Unfortunately, it creates additional obstacles for measuring performance in such an environment. Sharing resources with neighbor processes can influence performance measurements in unpredictable ways.

Analyzing production workloads by recreating them in a lab can be tricky. Sometimes it's not possible to synthesize exact behavior for "in-house" performance testing. This is why more and more often, cloud providers and hyperscalers choose to profile and monitor performance directly on production systems [Ren et al., 2010]. Measuring performance when there are "no other players" may not reflect real-world scenarios. It would be a waste of time to implement code optimizations that perform well in a lab environment but not in a production environment. Having said that, it doesn't eliminate the need for continuous "in-house" testing to catch performance problems early. Not all performance regressions can be caught in a lab, but engineers should design performance benchmarks representative of real-world scenarios.

It's becoming a trend for large service providers to implement telemetry systems that monitor performance on user devices. One such example is the Netflix Icarus[15] telemetry service, which runs on thousands of different devices spread all around the world. Such a telemetry system helps Netflix understand how real users perceive Netflix's app performance. It allows engineers to analyze data collected from many devices and to find issues that would be impossible to find otherwise. This kind of data allows making better-informed decisions on where to focus the optimization efforts.

One important caveat of monitoring production deployments is measurement overhead. Because any kind of monitoring affects the performance of a running service, it's recommended to use only lightweight profiling methods. According to [Ren et al., 2010]: "To conduct continuous profiling on datacenter machines serving real traffic, extremely low overhead is paramount". Usually, acceptable aggregated overhead is considered below 1%. Performance monitoring overhead can be reduced by limiting the set of profiled machines as well as using longer time intervals.

Measuring performance in such production environments means that we must accept its noisy nature and use statistical methods to analyze results. A good example of how large companies like LinkedIn use statistical methods to measure and compare quantile-based metrics (e.g., 90th percentile Page Load Times) in their A/B testing in the production environment can be found in [Liu et al., 2019].

2.3 Automated Detection of Performance Regressions

It is becoming a trend that SW vendors try to increase the frequency of deployments. Companies constantly seek ways to accelerate the rate of delivering their products to the market. Unfortunately, this doesn't automatically imply that SW products become better with each new release. In particular, software performance defects tend to leak into production software at an alarming rate [Jin et al., 2012]. A large number of changes in software impose a challenge to analyze all of those results and historical data to detect performance regressions.

Software performance regressions are defects that are erroneously introduced into software as it evolves from one version to the next. Catching performance bugs and improvements means detecting which commits change the performance of the software (as measured by performance tests) in the presence of the noise from the testing infrastructure. From database systems to search engines to compilers, performance regressions are commonly experienced by almost all large-scale software systems during their continuous evolution and deployment life cycle.

[15] Presented at CMG 2019, https://www.youtube.com/watch?v=4RG2DUK03_0.

It may be impossible to entirely avoid performance regressions during software development, but with proper testing and diagnostic tools, the likelihood for such defects silently leaking into production code can be reduced significantly.

The first option that comes to mind is: having humans to look at the graphs and compare results. It shouldn't be surprising that we want to move away from that option very quickly. People tend to lose focus quickly and can miss regressions, especially on a noisy chart, like the one shown in Figure 3. Humans will likely catch performance regression that happened around August 5th, but it's not obvious that humans will detect later regressions. In addition to being error-prone, having humans in the loop is also a time consuming and boring job that must be performed daily.

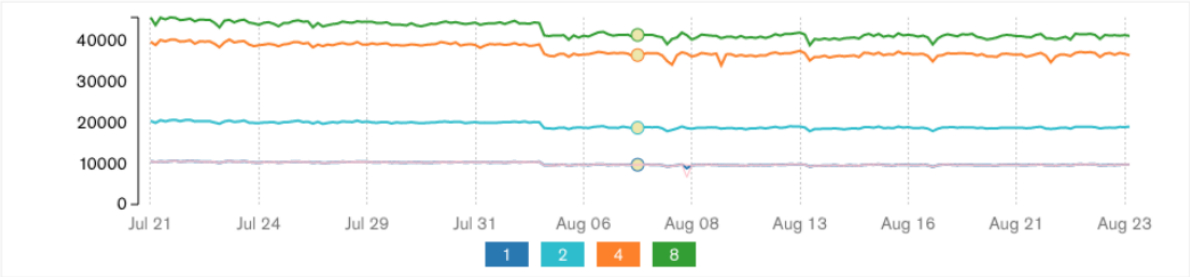


Figure 3: Performance trend graph for four tests with a small drop in performance on August 5th (the higher value, the better). © *Image from [Daly et al., 2020]*

The second option is to have a threshold, e.g. 2%: every code modification that has performance within the threshold is considered noise, and everything above the threshold is considered a regression. It is somewhat better than the first option but still has its own drawbacks. Fluctuations in performance tests are inevitable: sometimes, even a harmless code change[16] can trigger performance variation in a benchmark. Choosing the right value for the threshold is extremely hard and does not guarantee a low rate of false-positive as well as false-negative alarms. Setting the threshold too low might lead to analyzing a bunch of small regressions that were not caused by the change in source code but due to some random noise. Setting the threshold too high might lead to filtering out real performance regressions. Small changes can pile up slowly into a bigger regression, which can be left unnoticed. For instance, suppose you have a threshold of 2%. If you have two consecutive 1.5% regressions, they both will be filtered out. But throughout two days, performance regression will sum up to 3%, which is bigger than the threshold. By looking at Figure 3, we can make an observation that the threshold requires per test adjustment. The threshold that might work for the green (upper line) test will not necessarily work equally well for the purple (lower line) test since they have a different level of noise. An example of a CI system where each test requires setting explicit threshold values for alerting a regression is LUCI,[17] which is a part of the Chromium project.

A third option is using statistical analysis to identify performance regressions. A simple example of this is using Student's t-test[18]) to compare the arithmetic mean of 100 runs of program A to that of 100 runs of program B. However, parametric tests such as this assume normal (i.e., Gaussian) sample distributions, which is often not true with typically right-skewed, multimodal system performance runtime histograms. Therefore, misapplying statistical tools in such cases runs the risk of producing misleading results. Fortunately, more appropriate statistical tools exist for non-normal distributions called "non-parametric" tests, examples of which include Mann-Whitney, Anderson-Darling, and Kolmogorov–Smirnov (more about that in the next section). Python and R offer these as downloadable packages for those interested in rolling their own automated performance regression test frameworks, while a growing list of open-source projects like stats-pal[19] offer ready-made frameworks for plugging into existing CI/CD pipelines.

An even more sophisticated statistical approach to identify performance regressions was taken in [Daly et al., 2020]. MongoDB developers implemented change point analysis for identifying performance changes in the evolving code base of their database products. According to [Matteson & James, 2014], change point analysis is the process of detecting distributional changes within time-ordered observations. MongoDB developers utilized an "E-Divisive means" algorithm that works by hierarchically selecting distributional change points that divide the time series into

[16] The following article shows that changing the order of the functions or removing dead functions can cause variations in performance: https://easyperf.net/blog/2018/01/18/Code_alignment_issues
[17] LUCI - https://chromium.googlesource.com/chromium/src.git/+/master/docs/tour_of_luci_ui.md
[18] Student's t-test - https://en.wikipedia.org/wiki/Student's_t-test
[19] Stats-pal - https://github.com/JoeyHendricks/STATS-PAL

clusters. Their open-sourced CI system called Evergreen[20] incorporates this algorithm to display change points on the chart and opens Jira tickets. More details about this automated performance testing system can be found in [Ingo & Daly, 2020].

Another interesting approach is presented in [Alam et al., 2019]. The authors of this paper presented `AutoPerf`, which uses hardware performance counters (PMC, see Section 3.9.1) to diagnose performance regressions in a modified program. First, it learns the distribution of the performance of a modified function based on its PMC profile data collected from the original program. Then, it detects deviations of performance as anomalies based on the PMC profile data collected from the modified program. `AutoPerf` showed that this design could effectively diagnose some of the most complex software performance bugs, like those hidden in parallel programs.

Regardless of the underlying algorithm of detecting performance regressions, a typical CI system should automate the following actions:

1. Setup a system under test.
2. Run a benchmark suite.
3. Report the results.
4. Determine if performance has changed.
5. Alert on unexpected change in performance.
6. Visualize the results for a human to analyze.

CI system should support both automated and manual benchmarking, yield repeatable results, and open tickets for performance regressions that were found. It is very important to detect regressions promptly. First, because fewer changes were merged since a regression happened. This allows us to have a person responsible for regression to look into the problem before they move to another task. Also, it is a lot easier for a developer to approach the regression since all the details are still fresh in their head as opposed to several weeks after that.

Lastly, the CI system should alert, not just on software performance regressions, but on unexpected performance improvements, too. For example, someone may check-in a seemingly innocuous commit which, nonetheless, reduces latency by a whopping 10% in the Automated Performance Regression harness. Your initial instinct may be to celebrate this fortuitous performance boost and proceed on with your day. However, while this commit may have passed all functional tests in your CI pipeline, chances are that this unexpected latency improvement uncovered a gap in functional testing which only manifested itself in the performance regression results. This scenario occurs often enough that it warrants explicit mention: treat the Automated Performance Regression harness as part of a holistic software testing framework, not as a silo.

Authors of the book highly recommend setting up an automated statistical performance tracking system. Try using different algorithms and see which works best for your application. It will certainly take time, but it will be a solid investment in the future performance health of your project.

2.4 Manual Performance Testing

It is great when engineers can leverage existing performance testing infrastructure during development. In the previous section, we discussed that one of the nice-to-have features of the CI system is the possibility to submit performance evaluation jobs to it. If this is supported, then the system would return the results of testing a patch that the developer wants to commit to the codebase. It may not always be possible due to various reasons, like hardware unavailability, setup is too complicated for testing infrastructure, a need to collect additional metrics. In this section, we provide basic advice for local performance evaluations.

When making performance improvements in our code, we need a way to prove that we actually made it better. Also, when we commit a regular code change, we want to make sure performance did not regress. Typically, we do this by 1) measuring the baseline performance, 2) measuring the performance of the modified program, and 3) comparing them with each other. The goal in such a scenario is to compare the performance of two different versions of the same functional program. For example, we have a program that recursively calculates Fibonacci numbers, and we decided to rewrite it in an iterative fashion. Both are functionally correct and yield the same numbers. Now we need to compare the performance of two programs.

It is highly recommended to get not just a single measurement but to run the benchmark multiple times. So, we have N measurements for the baseline and N measurements for the modified version of the program. Now we need a

[20] Evergreen - https://github.com/evergreen-ci/evergreen

way to compare those two sets of measurements to decide which one is faster. This task is intractable by itself, and there are many ways to be fooled by the measurements and potentially derive wrong conclusions from them. If you ask any data scientist, they will tell you that you should not rely on a single metric (min/mean/median, etc.).

Consider two distributions of performance measurements collected for two versions of a program in Figure 4. This chart displays the probability we get a particular timing for a given version of a program. For example, there is a ~32% chance the version `A` will finish in ~102 seconds. It's tempting to say that `A` is faster than `B`. However, it is true only with some probability `P`. This is because there are some measurements of `B` that are faster than `A`. Even in the situation when all the measurements of `B` are slower than every measurement of `A` probability `P` is not equal to `100%`. This is because we can always produce one additional sample for `B`, which may be faster than some samples of `A`.

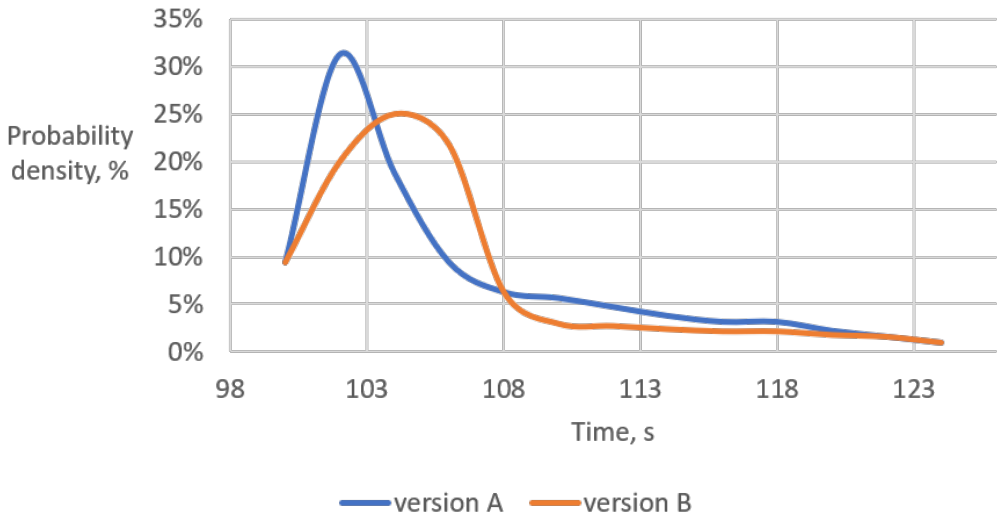


Figure 4: Comparing 2 performance measurement distributions.

An interesting advantage of using distribution plots is that it allows you to spot unwanted behavior of the benchmark.[21] If the distribution is bimodal, the benchmark likely experiences two different types of behavior. A common cause of bimodally distributed measurements is code that has both a fast and a slow path, such as accessing a cache (cache hit vs. cache miss) and acquiring a lock (contended lock vs. uncontended lock). To "fix" this, different functional patterns should be isolated and benchmarked separately.

Data scientists often present measurements by plotting the distributions and avoid calculating speedup ratios. This eliminates biased conclusions and allows readers to interpret the data themselves. One of the popular ways to plot distributions is by using box plots (see Figure 5), which allow comparisons of multiple distributions on the same chart.

While visualizing performance distributions may help you discover certain anomalies, developers shouldn't use them for calculating speedups. In general, it's hard to estimate the speedup by looking at performance measurement distributions. Also, as discussed in the previous section, it doesn't work for automated benchmarking systems. Usually, we want to get a scalar value that will represent a speedup ratio between performance distributions of 2 versions of a program, for example, "version `A` is faster than version `B` by `X%`".

The statistical relationship between the two distributions is identified using Hypothesis Testing methods. A comparison is deemed *statistically significant* if the relationship between the data-sets would reject the null hypothesis[22] according to a threshold probability (the significance level).

- If the distributions are Gaussian (normal distribution), then using a parametric hypothesis test (e.g., Student's T-test) to compare the distributions will suffice. Though it is worth to mention that Gaussian distributions are very rarely seen in performance data. So, be cautious using formulas from statistics textbooks assuming Gaussian distributions.

[21] Another way to check this is to run the normality test: https://en.wikipedia.org/wiki/Normality_test.
[22] Null hypothesis - https://en.wikipedia.org/wiki/Null_hypothesis.

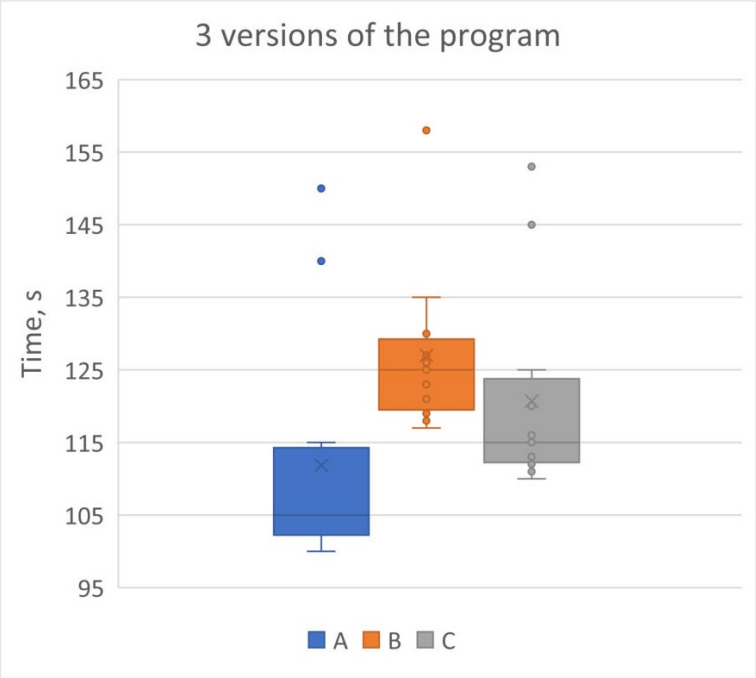


Figure 5: Box plots.

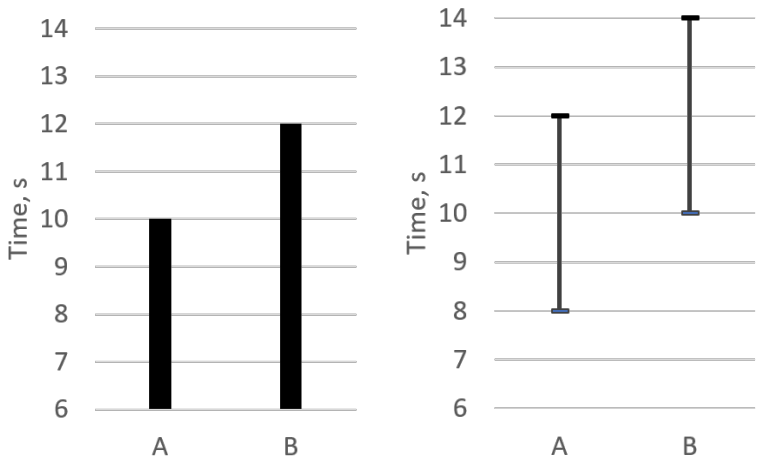
- If the distributions being compared are not Gaussian (e.g., heavily skewed or multimodal), then it's possible to use non-parametric tests (e.g., Mann-Whitney,[23] Kruskal Wallis,[24] etc.).

Hypothesis Testing methods are great for determining whether a speedup (or slowdown) is random or not. Therefore, it is best used in Automated Testing Frameworks to verify that the commit didn't introduce any performance regressions. A good reference specifically about statistics for performance engineering is a book by Dror G. Feitelson, "Workload Modeling for Computer Systems Performance Evaluation",[25] that has more information on modal distributions, skewness, and other related topics.

Once it has been determined that the difference is statistically significant via the hypothesis test, then the speedup can be calculated as a ratio between the means or geometric means, but there are caveats. On a small collection of samples, the mean and geometric mean can be affected by outliers. Unless distributions have low variance, do not consider averages alone. If the variance in the measurements is on the same order of magnitude as the mean, the average is not a representative metric. Figure 6 shows an example of 2 versions of the program. By looking only at averages (6a), it's tempting to say that version `A` is a 20% speedup over version `B`. However, taking into account the variance of the measurements (6b), we can see that it is not always the case. If we take the worse score for version `A` and the best score for version `B`, we can say that version `B` is a 20% speedup over version `A`. For normal distributions, a combination of mean, standard deviation, and standard error can be used to gauge a speedup between two versions of a program. Otherwise, for skewed or multimodal samples, one would have to use percentiles that are more appropriate for the benchmark, e.g., min, median, 90th, 95th, 99th, max, or some combination of these.

One of the most important factors in calculating accurate speedup ratios is collecting a rich collection of samples, i.e., run the benchmark a large number of times. This may sound obvious, but it is not always achievable. For example, some of the SPEC CPU 2017 benchmarks[26] run for more than 10 minutes on a modern machine. That means it would take 1 hour to produce just three samples: 30 minutes for each version of the program. Imagine that you have not just a single benchmark in your suite, but hundreds. It would become very expensive to collect statistically sufficient data even if you distribute the work across multiple machines.

[23] Mann-Whitney U test - https://en.wikipedia.org/wiki/Mann-Whitney_U_test.
[24] Kruskal-Wallis analysis of variance - https://en.wikipedia.org/wiki/Kruskal-Wallis_one-way_analysis_of_variance.
[25] Book "Workload Modeling for Computer Systems Performance Evaluation" - https://www.cs.huji.ac.il/~feit/wlmod/.
[26] SPEC CPU 2017 benchmarks - http://spec.org/cpu2017/Docs/overview.html#benchmarks



(a) Averages only (b) Full measurement intervals

Figure 6: Two histograms showing how averages could be misleading.

How do you know how many samples are required to reach statistically sufficient distribution? The answer to this question again depends on how much accuracy you want your comparison to have. The lower the variance between the samples in the distribution, the lower number of samples you need. Standard deviation is the metric that tells you how consistent the measurements in the distribution are. One can implement an adaptive strategy by dynamically limiting the number of benchmark iterations based on standard deviation, i.e., you collect samples until you get a standard deviation that lies in a certain range. This approach requires the number of measurements to be more than one. Otherwise, the algorithm will stop after the first sample because a single run of a benchmark has `std.dev.` equals to zero. Once you have a standard deviation lower than the threshold, you could stop collecting measurements. This strategy is explained in more detail in [Akinshin, 2019, Chapter 4].

Another important thing to watch out for is the presence of outliers. It is OK to discard some samples (for example, cold runs) as outliers by using confidence intervals, but do not deliberately discard unwanted samples from the measurement set. For some types of benchmarks, outliers can be one of the most important metrics. For example, when benchmarking SW that has real-time constraints, 99-percentile could be very interesting. There is a series of talks about measuring latency by Gil Tene on YouTube that covers this topic well.

2.5 Software and Hardware Timers

To benchmark execution time, engineers usually use two different timers, which all the modern platforms provide:

- **System-wide high-resolution timer**: this is a system timer that is typically implemented as a simple count of the number of ticks that have transpired since an arbitrary starting date, called the epoch[27]. This clock is monotonic; i.e., it always goes up. System time can be retrieved from the OS with a system call.[28] Accessing the system timer on Linux systems is possible via the `clock_gettime` system call. System timer has a nano-seconds resolution, is consistent between all the CPUs and is independent of CPU frequency. Even though the system timer can return timestamps with nano-seconds accuracy, it is not suitable for measuring short running events because it takes a long time to obtain the timestamp via the `clock_gettime` system call. But it is OK to measure events with a duration of more than a microsecond. The `de facto` standard for accessing system timer in C++ is using `std::chrono` as shown in Listing 1.
- **Time Stamp Counter (TSC)**: this is an HW timer which is implemented as an HW register. TSC is monotonic and has a constant rate, i.e., it doesn't account for frequency changes. Every CPU has its own TSC, which is simply the number of reference cycles (see Section 4.6) elapsed. It is suitable for measuring short events with a duration from nanoseconds and up to a minute. The value of TSC can be retrieved by using compiler built-in function `__rdtsc` as shown in Listing 2, which uses `RDTSC` assembly instruction under the

[27] Unix epoch starts at 1 January 1970 00:00:00 UT: https://en.wikipedia.org/wiki/Unix_epoch.
[28] Retrieving system time - https://en.wikipedia.org/wiki/System_time#Retrieving_system_time

Listing 1 Using C++ std::chrono to access system timer

```cpp
#include <cstdint>
#include <chrono>

// returns elapsed time in nanoseconds
uint64_t timeWithChrono() {
  using namespace std::chrono;
  auto start = steady_clock::now();
  // run something
  auto end = steady_clock::now();
  uint64_t delta = duration_cast<nanoseconds>
      (end - start).count();
  return delta;
}
```

hood. More low-level details on benchmarking the code using the `RDTSC` assembly instruction can be accessed in the white paper [Paoloni, 2010].

Listing 2 Using ___rdtsc compiler builtins to access TSC

```cpp
#include <x86intrin.h>
#include <cstdint>

// returns the number of elapsed reference clocks
uint64_t timeWithTSC() {
    uint64_t start = __rdtsc();
    // run something
    return __rdtsc() - start;
}
```

Choosing which timer to use is very simple and depends on how long the thing is that you want to measure. If you measure something over a very small time period, TSC will give you better accuracy. Conversely, it's pointless to use the TSC to measure a program that runs for hours. Unless you really need cycle accuracy, the system timer should be enough for a large proportion of cases. It's important to keep in mind that accessing system timer usually has higher latency than accessing TSC. Making a `clock_gettime` system call can be easily ten times slower than executing `RDTSC` instruction, which takes 20+ CPU cycles. This may become important for minimizing measurement overhead, especially in the production environment. A performance comparison of different APIs for accessing timers on various platforms is available on a wiki page[29] of the CppPerformanceBenchmarks repository.

2.6 Microbenchmarks

Microbenchmarks are small self-contained programs that people write to quickly test a hypothesis. Usually, microbenchmarks are used to choose the best implementation of a certain relatively small algorithm or functionality. Nearly all modern languages have benchmarking frameworks. In C++, one can use the Google benchmark[30] library, C# has BenchmarkDotNet[31] library, Julia has the BenchmarkTools[32] package, Java has JMH[33] (Java Microbenchmark Harness), etc.

When writing microbenchmarks, it's very important to ensure that the scenario you want to test is actually executed by your microbenchmark at runtime. Optimizing compilers can eliminate important code that could render the

[29] CppPerformanceBenchmarks wiki - https://gitlab.com/chriscox/CppPerformanceBenchmarks/-/wikis/ClockTimeAnalysis
[30] Google benchmark library - https://github.com/google/benchmark
[31] BenchmarkDotNet - https://github.com/dotnet/BenchmarkDotNet
[32] Julia BenchmarkTools - https://github.com/JuliaCI/BenchmarkTools.jl
[33] Java Microbenchmark Harness - http://openjdk.java.net/projects/code-tools/jmh/etc

experiment useless, or even worse, drive you to the wrong conclusion. In the example below, modern compilers are likely to eliminate the whole loop:

```
// foo DOES NOT benchmark string creation
void foo() {
  for (int i = 0; i < 1000; i++)
    std::string s("hi");
}
```

A simple way to test this is to check the performance profile of the benchmark and see if the intended code stands out as the hotspot. Sometimes abnormal timings can be spotted instantly, so use common sense while analyzing and comparing benchmark runs. One of the popular ways to keep the compiler from optimizing away important code is to use `DoNotOptimize`-like[34] helper functions, which do the necessary inline assembly magic under the hood:

```
// foo benchmarks string creation
void foo() {
  for (int i = 0; i < 1000; i++) {
    std::string s("hi");
    DoNotOptimize(s);
  }
}
```

If written well, microbenchmarks can be a good source of performance data. They are often used for comparing the performance of different implementations of a critical function. What defines a good benchmark is whether it tests performance in realistic conditions in which functionality will be used. If a benchmark uses synthetic input that is different from what will be given in practice, then the benchmark will likely mislead you and will drive you to the wrong conclusions. Besides that, when a benchmark runs on a system free from other demanding processes, it has all resources available to it, including DRAM and cache space. Such a benchmark will likely champion the faster version of the function even if it consumes more memory than the other version. However, the outcome can be the opposite if there are neighbor processes that consume a significant part of DRAM, which causes memory regions that belong to the benchmark process to be swapped to the disk.

For the same reason, be careful when concluding results obtained from unit-testing a function. Modern unit-testing frameworks, e.g. GoogleTest, provide the duration of each test. However, this information cannot substitute a carefully written benchmark that tests the function in practical conditions using realistic input (see more in [Fog, 2004, chapter 16.2]). It is not always possible to replicate the exact input and environment as it will be in practice, but it is something developers should take into account when writing a good benchmark.

Questions and Exercises

1. Is it safe to take an average time over a series of measurements?
2. Suppose you've identified a performance bug that you're now trying to fix in your development environment. How you would reduce the noise in the system to have more pronounced benchmarking results.
3. Is it OK to track overall performance of a program with function-level unit tests?
4. Does your organization has performance regression system in place? If yes, can it be improved? If no, think about the strategy of installing one. Take into consideration: what is changing and what isn't (source code, compiler, HW config, etc), how often a change occurs, what is the measurement variance, what is the running time of the benchmark and how many iterations you can run?

Chapter Summary

- Debugging performance issues is usually harder than debugging functional bugs due to measurement instability.
- You can never stop optimizing unless you set a particular goal. To know if you reached the desired goal, you need to come up with meaningful definitions and metrics for how you will measure that. Depending on what you care about, it could be throughput, latency, operations per second (roofline performance), etc.
- Modern systems have non-deterministic performance. Eliminating non-determinism in a system is helpful for well-defined, stable performance tests, e.g., microbenchmarks. Measuring performance in production deployment requires dealing with a noisy environment by using statistical methods for analyzing results.

[34] For JMH, this is known as the `Blackhole.consume()`.

- More and more often, vendors of large distributed SW choose to profile and monitor performance directly on production systems, which requires using only light-weight profiling techniques.
- It is very beneficial to employ an automated statistical performance tracking system for preventing performance regressions from leaking into production software. Such CI systems are supposed to run automated performance tests, visualize results, and alert on discovered perfomance anomalies.
- Visualizing performance distributions also may help discover performance anomalies. It is a safe way of presenting performance results to a wide audience.
- Statistical relationship between performance distributions is identified using Hypothesis Testing methods. Once it's determined that the difference is statistically significant, then the speedup can be calculated as a ratio between the means or geometric means.
- It's OK to discard cold runs to ensure that everything is running hot, but do not deliberately discard unwanted data. If you decide to discard some samples, do it uniformly for all distributions.
- To benchmark execution time, engineers can use two different timers, which all the modern platforms provide. The system-wide high-resolution timer is suitable for measuring events whose duration is more than a microsecond. For measuring short events with high accuracy, use Time Stamp Counter.
- Microbenchmarks are good for proving something quickly, but you should always verify your ideas on a real application in practical conditions. Make sure that you are benchmarking the meaningful code by checking performance profiles.

3 CPU Microarchitecture

This chapter provides a brief summary of the critical CPU microarchitecture features that have a direct impact on software performance. The goal of this chapter is not to cover all the details and trade-offs of CPU architectures, which are already covered extensively in the literature [Hennessy & Patterson, 2017]. In this chapter we will provide a quick recap of the CPU hardware features that are present in modern processors.

3.1 Instruction Set Architecture

The instruction set is the vocabulary used by software to communicate with the hardware. The instruction set architecture (ISA) defines the contract between the software and the hardware. Intel x86, ARM v8, RISC-V are examples of current-day ISA that are most widely deployed. All of these are 64-bit architectures, i.e., all address computation uses 64-bit. ISA developers and CPU architects typically ensure that software or firmware that conforms to the specification will execute on any processor built using the specification. Widely deployed ISA franchises also typically ensure backward compatibility such that code written for the GenX version of a processor will continue to execute on GenX+i.

Most modern architectures can be classified as general purpose register-based, load-store architectures where the operands are explicitly specified, and memory is accessed only using load and store instructions. In addition to providing the basic functions in the ISA such as load, store, control, scalar arithmetic operations using integers and floating-point, the widely deployed architectures continue to enhance their ISA to support new computing paradigms. These include enhanced vector processing instructions (e.g., Intel AVX2, AVX512, ARM SVE) and matrix/tensor instructions (Intel AMX). Software mapped to use these advanced instructions typically provide orders of magnitude improvement in performance.

Modern CPUs support 32b and 64b precision for arithmetic operations. With the fast-evolving field of deep learning, the industry has a renewed interest in alternate numeric formats for variables to drive significant performance improvements. Research has shown that deep learning models perform just as good, using fewer bits to represent the variables, saving on both compute and memory bandwidth. As a result, several CPU franchises have recently added support for lower precision data types such as 8bit integers (int8, e.g., Intel VNNI), 16b floating-point (fp16, bf16) in the ISA, in addition to the traditional 32-bit and 64-bit formats for arithmetic operations.

3.2 Pipelining

Pipelining is the foundational technique used to make CPUs fast wherein multiple instructions are overlapped during their execution. Pipelining in CPUs drew inspiration from the automotive assembly lines. The processing of instructions is divided into stages. The stages operate in parallel, working on different parts of different instructions. DLX is an example of a simple 5-stage pipeline defined by [Hennessy & Patterson, 2017] and consists of:

1. Instruction fetch (IF)
2. Instruction decode (ID)
3. Execute (EXE)
4. Memory access (MEM)
5. Write back (WB)

Figure 7 shows an ideal pipeline view of the 5-stage pipeline CPU. In cycle 1, instruction x enters the IF stage of the pipeline. In the next cycle, as instruction x moves to the ID stage, the next instruction in the program enters the IF stage, and so on. Once the pipeline is full, as in cycle 5 above, all pipeline stages of the CPU are busy working on different instructions. Without pipelining, instruction `x+1` couldn't start its execution until instruction `x` finishes its work.

Most modern CPUs are deeply pipelined, aka super pipelined. The throughput of a pipelined CPU is defined as the number of instructions that complete and exit the pipeline per unit of time. The latency for any given instruction is the total time through all the stages of the pipeline. Since all the stages of the pipeline are linked together, each stage must be ready to move to the next instruction in lockstep. The time required to move an instruction from one stage to the other defines the basic machine cycle or clock for the CPU. The value chosen for the clock for a given pipeline is defined by the slowest stage of the pipeline. CPU hardware designers strive to balance the amount

Instruction	Clock cycle								
	1	2	3	4	5	6	7	8	9
Instruction x	IF	ID	EXE	MEM	WB				
Instruction x+1		IF	ID	EXE	MEM	WB			
Instruction x+2			IF	ID	EXE	MEM	WB		
Instruction x+3				IF	ID	EXE	MEM	WB	
Instruction x+4					IF	ID	EXE	MEM	WB

Figure 7: Simple 5-stage pipeline diagram.

of work that can be done in a stage as this directly defines the frequency of operation of the CPU. Increasing the frequency improves performance and typically involves balancing and re-pipelining to eliminate bottlenecks caused by the slowest pipeline stages.

In an ideal pipeline that is perfectly balanced and doesn't incur any stalls, the time per instruction in the pipelined machine is given by

$$\text{Time per instruction on pipelined machine} = \frac{\text{Time per instruction on nonpipelined machine}}{\text{Number of pipe stages}}$$

In real implementations, pipelining introduces several constraints that limit the ideal model shown above. Pipeline hazards prevent the ideal pipeline behavior resulting in stalls. The three classes of hazards are structural hazards, data hazards, and control hazards. Luckily for the programmer, in modern CPUs, all classes of hazards are handled by the hardware.

- **Structural hazards**: are caused by resource conflicts. To a large extent, they could be eliminated by replicating the hardware resources, such as using multi-ported registers or memories. However, eliminating all such hazards could potentially become quite expensive in terms of silicon area and power.
- **Data hazards**: are caused by data dependencies in the program and are classified into three types:

  Read-after-write (RAW) hazard requires dependent read to execute after write. It occurs when an instruction x+1 reads a source before a previous instruction x writes to the source, resulting in the wrong value being read. CPUs implement data forwarding from a later stage of the pipeline to an earlier stage (called "*bypassing*") to mitigate the penalty associated with the RAW hazard. The idea is that results from instruction x can be forwarded to instruction x+1 before instruction x is fully completed. If we take a look at the example:

  ```
  R1 = R0 ADD 1
  R2 = R1 ADD 2
  ```

  There is a RAW dependency for register R1. If we take the value directly after addition `R0 ADD 1` is done (from the `EXE` pipeline stage), we don't need to wait until the `WB` stage finishes, and the value will be written to the register file. Bypassing helps to save a few cycles. The longer the pipeline, the more effective bypassing becomes.

  Write-after-read (WAR) hazard requires dependent write to execute after read. It occurs when an instruction x+1 writes a source before a previous instruction x reads the source, resulting in the wrong new value being read. WAR hazard is not a true dependency and is eliminated by a technique called register renaming[35]. It is a technique that abstracts logical registers from physical registers. CPUs support register renaming by keeping a large number of physical registers. Logical (architectural) registers, the ones that are defined by the ISA, are just aliases over a wider register file. With such decoupling of architectural state,[36] solving WAR hazards is simple; we just need to use a different physical register for the write operation. For example:

[35] Register renaming - https://en.wikipedia.org/wiki/Register_renaming.
[36] Architectural state - https://en.wikipedia.org/wiki/Architectural_state.

```
R1 = R0 ADD 1
R0 = R2 ADD 2
```

There is a WAR dependency for register R0. Since we have a large pool of physical registers, we can simply rename all the occurrences of `R0` register starting from the write operation and below. Once we eliminated WAR hazard by renaming register `R0`, we can safely execute the two operations in any order.

Write-after-write (WAW) hazard requires dependent write to execute after write. It occurs when instruction x+1 writes a source before instruction x writes to the source, resulting in the wrong order of writes. WAW hazards are also eliminated by register renaming, allowing both writes to execute in any order while preserving the correct final result.

- **Control hazards**: are caused due to changes in the program flow. They arise from pipelining branches and other instructions that change the program flow. The branch condition that determines the direction of the branch (taken vs. not-taken) is resolved in the execute pipeline stage. As a result, the fetch of the next instruction cannot be pipelined unless the control hazard is eliminated. Techniques such as dynamic branch prediction and speculative execution described in the next section are used to overcome control hazards.

3.3 Exploiting Instruction Level Parallelism (ILP)

Most instructions in a program lend themselves to be pipelined and executed in parallel, as they are independent. Modern CPUs implement a large menu of additional hardware features to exploit such instruction-level parallelism (ILP). Working in concert with advanced compiler techniques, these hardware features provide significant performance improvements.

3.3.1 OOO Execution

The pipeline example in Figure 7 shows all instructions moving through the different stages of the pipeline in-order, i.e., in the same order as they appear in the program. Most modern CPUs support out-of-order (OOO) execution, i.e., sequential instructions can enter the execution pipeline stage in any arbitrary order only limited by their dependencies. OOO execution CPUs must still give the same result as if all instructions were executed in the program order. An instruction is called *retired* when it is finally executed, and its results are correct and visible in the architectural state. To ensure correctness, CPUs must retire all instructions in the program order. OOO is primarily used to avoid underutilization of CPU resources due to stalls caused by dependencies, especially in superscalar engines described in the next section.

Dynamic scheduling of these instructions is enabled by sophisticated hardware structures such as scoreboards and techniques such as register renaming to reduce data hazards. In the 1960s, some work to support dynamic scheduling and out-of-order execution included the Tomasulo algorithm,[37] implemented in the IBM360, and Scoreboading,[38] which was implemented in the CDC6600. Those pioneering efforts have influenced all modern CPU architectures.The scoreboard hardware is used to schedule the in-order retirement and all machine state updates. It keeps track of data dependencies of every instruction and where in the pipe the data is available. Most implementations strive to balance the hardware cost with the potential return. Typically, the size of the scoreboard determines how far ahead the hardware can look for scheduling such independent instructions.

Figure 8 details the concept underlying out-of-order execution with an example. Assume instruction x+1 cannot execute in cycles 4 and 5 due to a conflict. An in-order CPU would stall all subsequent instructions from entering the EXE pipeline stage. In an OOO CPU, subsequent instructions that do not have any conflicts (e.g., instruction x+2) can enter and complete its execution. All instructions still retire in order, i.e., the instructions complete the WB stage in the program order.

3.3.2 Superscalar Engines and VLIW

Most modern CPUs are superscalar i.e., they can issue more than one instruction in a given cycle. Issue-width is the maximum number of instructions that can be issued during the same cycle. Typical issue-width of current generation CPUs ranges from 2 to 6. To ensure the right balance, such superscalar engines also have more than one

[37] Tomasulo algorithm - https://en.wikipedia.org/wiki/Tomasulo_algorithm.
[38] Scoreboarding - https://en.wikipedia.org/wiki/Scoreboarding.

Instruction	Clock cycle									
	1	2	3	4	5	6	7	8	9	10
Instruction x	IF	ID	EXE	MEM	WB					
Instruction x+1		IF	ID			EXE	MEM	WB		
Instruction x+2			IF	ID	EXE	MEM			WB	
Instruction x+3				IF	ID		EXE	MEM		WB

Figure 8: The concept of out-of-order execution.

execution unit and/or pipelined execution units. CPUs also combine superscalar capability with deep pipelines and out-of-order execution to extract the maximum ILP for a given piece of software.

Instruction	Clock cycle					
	1	2	3	4	5	6
Instruction x	IF	ID	EXE	MEM	WB	
Instruction x+1	IF	ID	EXE	MEM	WB	
Instruction x+2		IF	ID	EXE	MEM	WB
Instruction x+3		IF	ID	EXE	MEM	WB

Figure 9: The pipeline diagram for a simple 2-way superscalar CPU.

Figure 9 shows an example CPU that supports 2-wide issue width, i.e., in each cycle, two instructions are processed in each stage of the pipeline. Superscalar CPUs typically support multiple, independent execution units to keep the instructions in the pipeline flowing through without conflicts. In addition to pipelining, replicating execution units further increases the performance of a machine.

Architectures such as the Intel Itanium moved the burden of scheduling a superscalar, multi-execution unit machine from the hardware to the compiler using a technique known as VLIW - Very Long Instruction Word. The rationale is to simplify the hardware by requiring the compiler to choose the right mix of instructions to keep the machine fully utilized. Compilers can use techniques such as software pipelining, loop unrolling, etc. to look further ahead than can be reasonably supported by hardware structures to find the right ILP.

3.3.3 Speculative Execution

As noted in the previous section, control hazards can cause significant performance loss in a pipeline if instructions are stalled until the branch condition is resolved. One technique to avoid this performance loss is hardware branch prediction logic to predict the likely direction of branches and allow executing instructions from the predicted path (speculative execution).

Let's consider the short code example in Listing 3. For a processor to understand which function it should execute next, it should know whether the condition `a < b` is false or true. Without knowing that, the CPU waits until the result of the branch instruction will be determined, as shown in Figure 10a.

With speculative execution, the CPU takes a guess on an outcome of the branch and initiates processing instructions from the chosen path. Suppose a processor predicted that condition `a < b` will be evaluated as true. It proceeded without waiting for the branch outcome and speculatively called function `foo` (see Figure 10b, speculative work is marked with `*`). State changes to the machine cannot be committed until the condition is resolved to ensure that the architecture state of the machine is never impacted by speculatively executing instructions. In the example

Listing 3 Speculative execution

```
if (a < b)
  foo();
else
  bar();
```

Instruction	Clock cycle							
	1	2	3	4	5	6	7	8
BRANCH (a < b)	IF	ID	EXE	MEM	WB			
CALL foo				IF	ID	EXE	MEM	WB
// INSTR from foo					IF	ID	EXE	MEM

(a) No speculation

Instruction	Clock cycle						
	1	2	3	4	5	6	7
BRANCH (a < b)	IF	ID	EXE	MEM	WB		
CALL foo		IF*	ID*	EXE	MEM	WB	
// INSTR from foo			IF*	ID	EXE	MEM	WB

(b) Speculative execution

Figure 10: The concept of speculative execution.

above, the branch instruction compares two scalar values, which is fast. But in reality, a branch instruction can be dependent on a value loaded from memory, which can take hundreds of cycles. If the prediction turns out to be correct, it saves a lot of cycles. However, sometimes the prediction is incorrect, and the function `bar` should be called instead. In such a case, the results from the speculative execution must be squashed and thrown away. This is called the branch misprediction penalty, which we discuss in Section 4.8.

To track the progress of speculation, the CPU supports a structure called the reorder buffer (ROB). The ROB maintains the status of all instruction execution and retires instructions in-order. Results from speculative execution are written to the ROB and are committed to the architecture registers, in the same order as the program flow and only if the speculation is correct. CPUs can also combine speculative execution with out-of-order execution and use the ROB to track both speculation and out-of-order execution.

3.3.4 Branch Prediction

As we just have seen, correct predictions greatly improve execution as they allow a CPU to make forward progress without having results of previous instructions available. However, bad speculation often incurs costly performance penalties. Modern CPUs employ fairly sophisticated dynamic branch prediction mechanism, which provide very high accuracy and can adapt to dynamic changes in branch behavior. There are three types of branches which could be handled in a special way:

- **Unconditional jumps and direct calls**: they are the easiest to predict as they are always taken and go in the same direction every time.
- **Conditional branches**: they have two potential outcomes: taken or not taken. Taken branches can go forward or backward. Forward conditional branches are usually generated for `if-else` statements, which have a high chance of not being taken as frequently it represents an error checking code. Backward conditional jumps are frequently seen in loops and used to go to the next iteration of a loop; such branches are usually

taken.
- **Indirect calls and jumps**: they have many targets. An indirect jump or indirect call can be generated for a `switch` statement, a function pointer, or a `virtual` function. A return from a function deserves attention because it has many potential targets as well.

Most prediction algorithms are based on previous outcomes of the branch. The core of the branch prediction unit (BPU) is a branch target buffer (BTB), which caches the target addresses for every branch. Prediction algorithms consult the BTB to generate the next address to fetch every cycle. The CPU uses that new address to fetch the next block of instructions. If no branches are identified in the current fetch block, the next address to fetch will be the next sequential aligned fetch block (fall through).

Unconditional branches do not require prediction, we just need to lookup the target address in the BTB. Remember, the BPU needs to generate the next address to fetch every cycle to avoid the pipeline stalls. We could have extracted the address just from the instruction encoding itself, but then we have to wait until the decode stage is over, which will introduce a bubble in the pipeline and make things slower. So, the next fetch address has to be determined at the time when the branch is fetched.

For conditional branches, we first need to predict whether it will be taken or not. If not taken, then we fall through, no need to lookup the target. Otherwise, we lookup the target address in the BTB. Conditional branches usually account for the biggest portion of total branches and are the main source of misprediction penalties in production software. For indirect branches we need to select one of the possible targets, but the prediction algorithm can be very similar to conditional branches.

All prediction mechanisms try to exploit two important principles, which are similar to what we will discuss with caches later:

- **Temporal correlation**: the way a branch resolves may be a good predictor of the way it will resolve at the next execution. Also know as local correlation.
- **Spatial correlation**: several adjacent branches may resolve in a highly correlated manner (a preferred path of execution). Also know as global correlation.

The best accuracy is often achieved by leveraging local and global correlation together. So, not only we look at the outcome history of the current branch, but also we correlate it with outcomes of other branches.

Another common technique used is called hybrid prediction. The idea is that some branches have biased behavior. For example, if a conditional branch goes in one direction 99.9% of the time, there is no need to use complex predictor and pollute its data structures. A much more simpler machanisms can be used. Another example is a loop branch. If a branch has loop behavior, then it can be predicted using a dedicated loop predictor, which will remember the number of iterations the loop typically executes.

Today, state of the art prediction is dominated by TAGE-like [Seznec & Michaud, 2006] or perceptron-based [Jimenez & Lin, 2001] predictors. Championship[39] branch predictors make less than 3 mispredictions per 1000 instructions. Modern CPUs routinely reach >95% prediction rate on most workloads.

3.4 SIMD Multiprocessors

Another variant of multiprocessing that is widely used for many workloads is referred to as Single Instruction Multiple Data (SIMD). As the name indicates, in SIMD processors, a single instruction operates on many data elements in a single cycle using many independent functional units. Operations on vectors and matrices lend themselves well to SIMD architectures as every element of a vector or matrix can be processed using the same instruction. SIMD architecture allows more efficient processing of a large amount of data and works best for data-parallel applications that involve vector operations.

Figure 11 shows scalar and SIMD execution modes for the code in Listing 4. In a traditional SISD (Single Instruction, Single Data) mode, addition operation is separately applied to each element of arrays `a` and `b`. However, in SIMD mode, addition is applied to multiple elements at the same time. If we target a CPU architecture which has execution units capable of performing operations on 256-bit vectors, we can process four double-precision elements with a single instruction. This leads to issuing 4x less instructions and can potentially gain a 4x speedup over four scalar computations. But in practice, performance benefits are not so straightforward for various reasons.

[39] 5th Championship Branch Prediction competition - https://jilp.org/cbp2016.

Listing 4 SIMD execution

```
double *a, *b, *c;
for (int i = 0; i < N; ++i) {
  c[i] = a[i] + b[i];
}
```

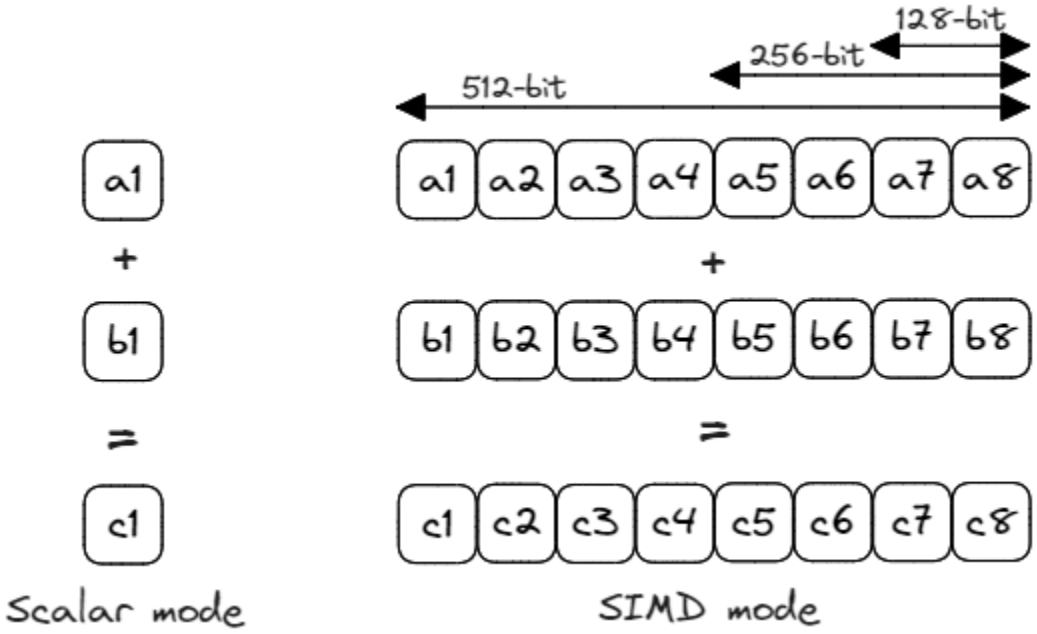


Figure 11: Example of scalar and SIMD operations.

For a regular SISD instructions, processors utilize general-purpose registers. Similarly, for SIMD instructions, CPUs have a set of SIMD registers to keep the data loaded from memory and store the intermediate results of computations. In our example, two regions of 256 bits of contiguous data corresponding to arrays `a` and `b` will be loaded from memory and stored in two separate vector registers. Next, the element-wise addition will be done and result will be stored in a new 256-bit vector register. Finally, the result will be written from the vector register to a 256-bit memory region that corresponds to the array `c`. Note, the data elements can be either integers or floating-point numbers.

Most of the popular CPU architectures feature vector instructions, including x86, PowerPC, Arm, and RISC-V. In 1996 Intel released MMX, a SIMD instruction set, that was designed for multimedia applications. Following MMX, Intel introduced new instruction sets with added capabilities and increased vector size: SSE, AVX, AVX2, AVX-512. Arm has optionally supported the 128-bit NEON instruction set in various versions of its architecture. In version 8 (aarch64), this support was made mandatory, and new instructions were added.

As the new instruction sets became available, work began to make them usable to software engineers. The software changes required to exploit SIMD instructions are known as *code vectorization*. At first, SIMD instructions were programmed in assembly. Later, special compiler intrinsics were introduced, which are small high-level source code functions that provide 1-to-1 mapping to SIMD instructions. Today all of the major compilers support autovectorization for the popular processors, i.e. they can generate SIMD instructions straight from the high-level code written in C/C++, Java, Rust and other languages.

To allow a code to run on systems that support different vector lengths, Arm introduced the SVE instruction set. Its defining characteristic is the concept of *scalable vectors*: their length is unknown at compile time. With SVE, there is no need to port software to every possible vector length. Users don't have to recompile the source code of their applications to leverage wider vectors when they become available in newer CPU generations. Another example of scalable vectors is the RISC-V V extension (RVV), which was ratified in late 2021. Some implementations

support quite wide (2048 bit) vectors, and up to eight can be grouped together to yield 16,384 bit vectors, which greatly reduces the number of instructions executed. At each loop iteration, user code typically does `ptr += number_of_lanes`, where `number_of_lanes` is not known at compile time. ARM SVE provides special instructions for such length-dependent operations while RVV allows to query/set the `number_of_lanes`.

Also, CPUs increasingly accelerate the matrix multiplications often used in machine learning. Intel's AMX extension, supported in Sapphire Rapids, multiplies 8-bit matrices of shape 16x64 and 64x16, accumulating into a 32-bit 16x16 matrix. By contrast, the unrelated but identically named AMX extension in Apple CPUs, as well as Arm's SME extension, compute outer products of a row and column, respectively stored in special 512-bit registers, or scalable vectors.

Initially SIMD was driven by multimedia applications and scientific computations but later found its use in many other domains. Over time, the set of operations supported in SIMD instruction sets has steadily increased. In addition to straightforward arithmetic as shown above, newer use cases of SIMD include:

- String processing: finding characters, validating UTF-8,[40] parsing JSON[41] and CSV[42];
- Hashing,[43] random generation,[44] cryptography(AES);
- Columnar databases (bit packing, filtering, joins);
- Sorting built-in types (VQSort,[45] QuickSelect);
- Machine Learning and Artificial Inteligence (speeding up PyTorch, Tensorflow).

3.5 Exploiting Thread Level Parallelism

Techniques described previously rely on the available parallelism in a program to speed up execution. In addition to that, CPUs support techniques to exploit parallelism across processes and/or threads executing on the CPU. Next, we will discuss three techniques to exploit Thread Level Parallelism (TLP): multicore systems, simultaneous multithreading and hybrid architectures. Such techniques allow to eke out the most efficiency from the available hardware resources and to improve the throughput of the system.

3.5.1 Multicore Systems

As processor architects began to reach the practical limitations of semiconductor design and fabrication, the GHz race slowed down and designers had to focus on other innovations to improve CPU performance. One of the key directions was the multicore design which attempted to increase core counts for each generation. The ideas is to replicate multiple processor cores on a single chip and let them serve different programs at the same time. As such, one of the cores can run the web browser, another core can render a video, yet another playing music, all at the same time. For a server machine, request from different clients can be served on separate cores, which greatly increases the throughput of such system.

The first consumer focused dual-core processor was Intel Core 2 Duo, released in 2005, that was followed by the AMD Athlon X2 architecture released later that same year. Multicore systems caused many software components to be redesigned and affected the way we write code. These days nearly all processors in consumer-facing devices are multicore CPUs. At the time of writing this book, high-end laptops have more than ten physical cores inside and server processors reaching almost 100 cores.

It may sound very impressive, but we cannot add cores infinetly. First of all, each core generates heat when it's working and safely dissipating that heat from the cores through the processor package remains a challenge. That means that when more cores are running, heat can quickly exceed the cooling capability. In such situation, multicore processors will reduce clock speeds. This is one of the reasons you can see server chips with a big number of cores having much lower frequencies than processors that go into laptops and desktops.

Cores in a multicore system are connected to each other and to the shared resources, such as last-level cache and memory controllers. Such communication channel is called *interconnect*, which frequently has either a ring or a mesh topology. Another challenge for CPU designers is to keep the machine balanced as the core counts gets

[40] UTF-8 validation - https://github.com/rusticstuff/simdutf8
[41] Parsing JSON - https://github.com/simdjson/simdjson.
[42] Parsing CSV - https://github.com/geofflangdale/simdcsv
[43] SIMD hashing - https://github.com/google/highwayhash
[44] Random generation - abseil library
[45] Sorting - VQSort

higher. When you replicate cores, some resources remain shared, for example, memory buses and last-level cache. This results in diminishing returns to performance as cores are added, unless you also address throughput of other shared resources, e.g. interconnect bandwidth, last-level cache size and bandwidth, memory bandwidth, etc. Shared resources frequently become the source of performance issues in a multicore system.

3.5.2 Simultaneous Multithreading

A more sophisticated approach to improve multithreaded performance is Simultaneous Multithreading (SMT). Very frequently people use the term *Hyperthreading* to describe the same thing. The goal of the technique it is to fully utilize the available width of the CPU pipeline. SMT allows multiple software threads to run simultaneously on the same physical core using shared resources. More precisely, instructions from multiple software threads execute concurrently in the same cycle. Those don't have to be threads from the same process, they can be completely different programs happened to be schedulled on the same physical core.

An example of execution on a non-SMT and an SMT2 processor is shown in Figure 12. In both cases the width of the processor pipeline is four, each slot representing an opportunity to issue a new instruction. A 100% machine utilization is when there are no unused slots, which never happens in real workloads. It's easy to see that for the non-SMT case, there are many unused slot, the available resources are not utilized well. It may happen for a variety of reasons, one of the typical reason is a cache miss. At cycle 3, thread 1 cannot make forward progress because it is waiting for data to arrive. SMT processors take this opportunity to schedule useful work from another thread. The goal here is to occupy unused slots by another thread to hide memory latency, improve hardware utilization and multithreaded performance.

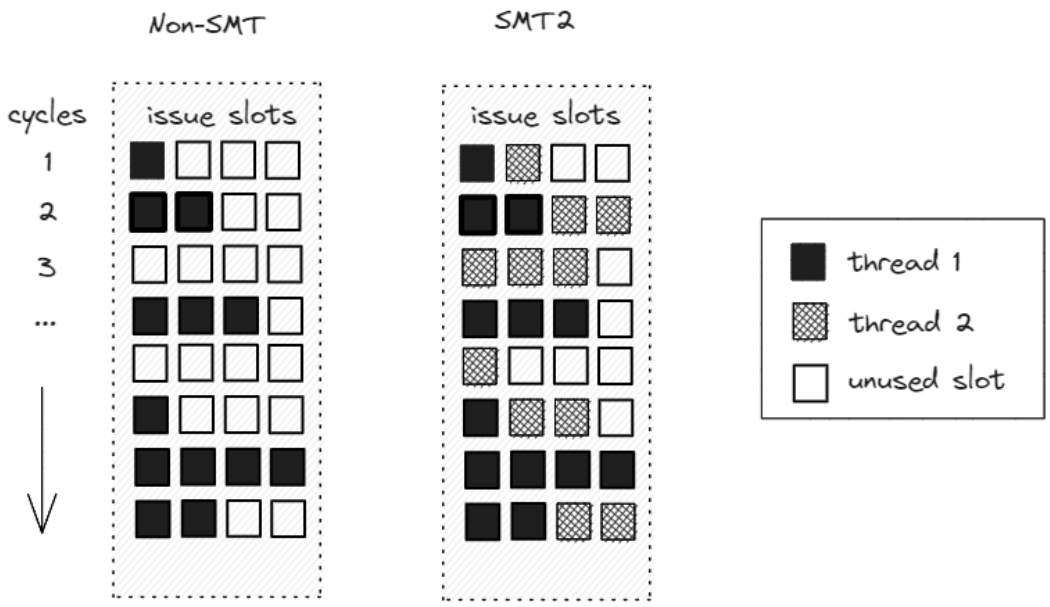


Figure 12: Execution on a 4-wide non-SMT and a 4-wide SMT2 processor.

With a SMT2 implementation, each physical core is represented with two logical cores, which are visible to the operating system as two independent processors available to take work. Consider a situation when we have 16 software threads ready to run, and only 8 physical cores. In a non-SMT system, only 8 threads will run at the same time, while with SMT2 we can execute all 16 threads simultaneously. In another hypothetical situation, if two programs run on a SMT-enabled core and each consistently utilize only two out of four available slots, then there is a high chance they will run as fast as if they would be running alone on that physical core.

Although two programs run on the same processor core, they are completely separated from each other. In an SMT-enabled processor, even though instructions are mixed, they have different context which helps preserve correctness of the execution. To support SMT, a CPU must replicate architectural state (program counter, registers) to maintain thread context. Other CPU resources can be shared. In a typical implementation, cache resources are dynamically shared amongst the hardware threads. Resources to track OOO and speculative execution can either be replicated or partitioned.

In an SMT2 core, both logical cores are truly running at the same time. In the CPU front-end, they fetch instructions in an alternating order (every cycle or a few cycles). In the back-end, each cycle a processor selects instructions for execution from all threads. Instruction execution is mixed as the processor dynamically schedules execution ports among both threads.

So, SMT is a very flexible setup, that allows to recover unused CPU issue slots. SMT provides equal single-thread performance, in addition to its benefits for multiple threads. Modern multi-threaded CPUs support either two-way (SMT2) or four-way (SMT4).

SMT has its own disadvantages as well. Since some resources are shared among the logical cores, they could eventually compete to use those resources. The most typical example of SMT penalty is related to utilization of L1 and L2 caches. Since they are shared between two logical cores, they could lack space in caches and force eviction of the data that will be used by another thread in the future.

SMT brings a considerable burden on software developers as it makes harder to predict and measure performance of an application that runs on an SMT core. Imagine you're running a critical code on an SMT core, and suddenly the OS puts another demanding job on a sibling logical core. Your code nearly maxes out the resources of the machine, and now you need to share it with someone else. This problem is especially pronounced in a cloud environment when you cannot predict whether your application would have noisy neighbors or not.

There is also a security concern with certain simultaneous multithreading implementations. Researchers showed that some earlier implementations had a vulnerability through which it is possible for one application to steal critical information (like cryptographic keys) from another application that runs on the sibling logical core of the same processor by monitoring its cache use. We will not dig deeper into this since it is not a book about HW security.

3.5.3 Hybrid Architectures

Computer architects also developed a hybrid CPU design, where two types of cores (or more) are put in the same processor. Typically, more powerful cores are coupled with relatively slower cores to address different goals. In such a system, big cores are used for latency-sensitive task and small cores are used for better battery-saving. But also, both types of cores can be utilized at the same time to improve multithreaded performance. All cores have access to the same memory, so workloads can migrate from big to small cores and back on the fly. The intention is to create a multicore processor that can adapt better to dynamic computing needs and use less power. For example, video games have parts of single-core burst performance as well as parts where they can scale to many cores.

The first mainstream hybrid architecture was ARM's big.LITTLE, which was introduced in October 2011. Other vendors followed this approach. Apple introduced its M1 chip in 2020 that has four high-performance "Firestorm" and four energy-efficient "Icestorm" cores. Intel introduced its Alderlake hybrid architecture in 2021 with eight P- and eight E-cores in the top configuration.

Hybrid architectures combine the best sides of both core types, but it comes with its own set of challenges. First of all, it requires cores to be fully ISA-compatible, i.e. they should be able to execute the same set of instructions. Otherwise, the schedulling becomes restricted. For example, if a big core features some fancy instructions that are not available on small cores, than you can only assign big cores to run workloads that use such instructions. That's why usualy vendors use the "greatest common denominator" approach when choosing the ISA for a hybrid processor.

Even with ISA-compatible cores, schedulling becomes challenging. Different types of workloads call for specific schedulling scheme, e.g. bursty execution vs. steady execution, low IPC vs. high IPC, low improtance vs. high importance, etc. It becomes non-trivial very quickly. Here are a few considerations for optimal scheduling:

- Leverage small cores to conserve power. Do not wake up big cores for the background work.
- Recognize candidates (low importance, low IPC) for offloading to smaller cores. Similarly, promote high importance, high IPC tasks to big cores.
- When assigning a new task, use an idle big core first. In case SMT, use big cores with both logical threads idle. After that, use idle small cores. After that, use sibling logical threads of big cores.

From a programmer's perspective, no code changes are needed to make use of hybrid systems. This approach became very popular in client-facing devices, especially in smartphones. We will take a look at Intel's Alderlake design later in the book.

3.6 Memory Hierarchy

To effectively utilize all the hardware resources provisioned in the CPU, the machine needs to be fed with the right data at the right time. Understanding the memory hierarchy is critically important to deliver on the performance capabilities of a CPU. Most programs exhibit the property of locality; they don't access all code or data uniformly. A CPU memory hierarchy is built on two fundamental properties:

- **Temporal locality**: when a given memory location is accessed, it is likely that the same location will be accessed again in the near future. Ideally, we want this information to be in the cache next time we need it.
- **Spatial locality**: when a given memory location is accessed, it is likely that nearby locations will be accessed in the near future. This refers to placing related data close to each other. When a program reads a single byte from memory, typically, a larger chunk of memory (cache line) is fetched because very often, the program will require that data soon.

This section provides a summary of the key attributes of memory hierarchy systems supported on modern CPUs.

3.6.1 Cache Hierarchy

A cache is the first level of the memory hierarchy for any request (for code or data) issued from the CPU pipeline. Ideally, the pipeline performs best with an infinite cache with the smallest access latency. In reality, the access time for any cache increases as a function of the size. Therefore, the cache is organized as a hierarchy of small, fast storage blocks closest to the execution units, backed up by larger, slower blocks. A particular level of the cache hierarchy can be used exclusively for code (instruction cache, i-cache) or for data (data cache, d-cache), or shared between code and data (unified cache). Furthermore, some levels of the hierarchy can be private to a particular CPU, while other levels can be shared among CPUs.

Caches are organized as blocks with a defined block size (**cache line**). The typical cache line size in modern CPUs is 64 bytes. Caches closest to the execution pipeline typically range in size from 8KiB to 32KiB. Caches further out in the hierarchy can be 64KiB to 16MiB in modern CPUs. The architecture for any level of a cache is defined by the following four attributes.

3.6.1.1 Placement of Data within the Cache. The address for a request is used to access the cache. In direct-mapped caches, a given block address can appear only in one location in the cache and is defined by a mapping function shown below.

$$\text{Number of Blocks in the Cache} = \frac{\text{Cache Size}}{\text{Cache Block Size}}$$

$$\text{Direct mapped location} = (\text{block address}) \bmod (\text{Number of Blocks in the Cache})$$

In a fully associative cache, a given block can be placed in any location in the cache.

An intermediate option between the direct mapping and fully associative mapping is a set-associative mapping. In such a cache, the blocks are organized as sets, typically each set containing 2, 4, 8 or 16 blocks. A given address is first mapped to a set. Within a set, the address can be placed anywhere, among the blocks in that set. A cache with m blocks per set is described as an m-way set-associative cache. The formulas for a set-associative cache are:

$$\text{Number of Sets in the Cache} = \frac{\text{Number of Blocks in the Cache}}{\text{Number of Blocks per Set (associativity)}}$$

$$\text{Set (m-way) associative location} = (\text{block address}) \bmod (\text{Number of Sets in the Cache})$$

3.6.1.2 Finding Data in the Cache. Every block in the m-way set-associative cache has an address tag associated with it. In addition, the tag also contains state bits such as valid bits to indicate whether the data is valid. Tags can also contain additional bits to indicate access information, sharing information, etc. that will be described in later sections.

Figure 13 shows how the address generated from the pipeline is used to check the caches. The lowest order address bits define the offset within a given block; the block offset bits (5 bits for 32-byte cache lines, 6 bits for 64-byte cache lines). The set is selected using the index bits based on the formulas described above. Once the set is selected, the tag bits are used to compare against all the tags in that set. If one of the tags matches the tag of the incoming request and the valid bit is set, a cache hit results. The data associated with that block entry (read out of the data

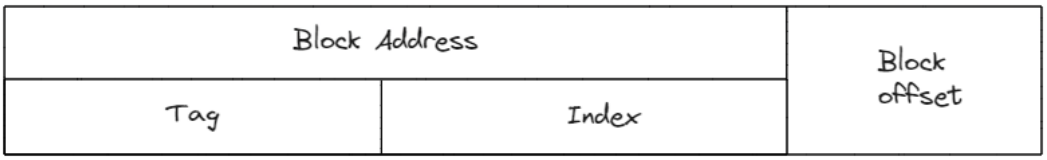


Figure 13: Address organization for cache lookup.

array of the cache in parallel to the tag lookup) is provided to the execution pipeline. A cache miss occurs in cases where the tag is not a match.

3.6.1.3 Managing Misses. When a cache miss occurs, the controller must select a block in the cache to be replaced to allocate the address that incurred the miss. For a direct-mapped cache, since the new address can be allocated only in a single location, the previous entry mapping to that location is deallocated, and the new entry is installed in its place. In a set-associative cache, since the new cache block can be placed in any of the blocks of the set, a replacement algorithm is required. The typical replacement algorithm used is the LRU (least recently used) policy, where the block that was least recently accessed is evicted to make room for the miss address. Another alternative is to randomly select one of the blocks as the victim block. Most CPUs define these capabilities in hardware, making it easier for executing software.

3.6.1.4 Managing Writes. Read accesses to caches are the most common case as programs typically read instructions, and data reads are larger than data writes. Handling writes in caches is harder, and CPU implementations use various techniques to handle this complexity. Software developers should pay special attention to the various write caching flows supported by the hardware to ensure the best performance of their code.

CPU designs use two basic mechanisms to handle writes that hit in the cache:

- In a write-through cache, hit data is written to both the block in the cache and to the next lower level of the hierarchy.
- In a write-back cache, hit data is only written to the cache. Subsequently, lower levels of the hierarchy contain stale data. The state of the modified line is tracked through a dirty bit in the tag. When a modified cache line is eventually evicted from the cache, a write-back operation forces the data to be written back to the next lower level.

Cache misses on write operations can be handled in two ways:

- In a *write-allocate or fetch on write miss* cache, the data for the missed location is loaded into the cache from the lower level of the hierarchy, and the write operation is subsequently handled like a write hit.
- If the cache uses a *no-write-allocate policy*, the cache miss transaction is sent directly to the lower levels of the hierarchy, and the block is not loaded into the cache.

Out of these options, most designs typically choose to implement a write-back cache with a write-allocate policy as both of these techniques try to convert subsequent write transactions into cache-hits, without additional traffic to the lower levels of the hierarchy. Write through caches typically use the no-write-allocate policy.

3.6.1.5 Other Cache Optimization Techniques. For a programmer, understanding the behavior of the cache hierarchy is critical to extract performance from any application. This is especially true when CPU clock frequencies increase while the memory technology speeds lag behind. From the perspective of the pipeline, the latency to access any request is given by the following formula that can be applied recursively to all the levels of the cache hierarchy up to the main memory:

$$\text{Average Access Latency} = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}$$

Hardware designers take on the challenge of reducing the hit time and miss penalty through many novel micro-architecture techniques. Fundamentally, cache misses stall the pipeline and hurt performance. The miss rate for any cache is highly dependent on the cache architecture (block size, associativity) and the software running on the machine. As a result, optimizing the miss rate becomes a hardware-software co-design effort. As described in the previous sections, CPUs provide optimal hardware organization for the caches. Additional techniques that can be implemented both in hardware and software to minimize cache miss rates are described below.

3.6.1.5.1 HW and SW Prefetching. One method to reduce a cache miss and the subsequent stall is to prefetch instructions as well as data into different levels of the cache hierarchy prior to when the pipeline demands. The assumption is the time to handle the miss penalty can be mostly hidden if the prefetch request is issued sufficiently ahead in the pipeline. Most CPUs support implicit hardware-based prefetching that is complemented by explicit software prefetching that programmers can control.

Hardware prefetchers observe the behavior of a running application and initiate prefetching on repetitive patterns of cache misses. Hardware prefetching can automatically adapt to the dynamic behavior of the application, such as varying data sets, and does not require support from an optimizing compiler or profiling support. Also, the hardware prefetching works without the overhead of additional address-generation and prefetch instructions. However, hardware prefetching is limited to learning and prefetching for a limited set of cache-miss patterns that are implemented in hardware.

Software memory prefetching complements the one done by the HW. Developers can specify which memory locations are needed ahead of time via dedicated HW instruction (see Section 8.2). Compilers can also automatically add prefetch instructions into the code to request data before it is required. Prefetch techniques need to balance between demand and prefetch requests to guard against prefetch traffic slowing down demand traffic.

3.6.2 Main Memory

Main memory is the next level of the hierarchy, downstream from the caches. Requests to load and store data are initiated by the Memory Controller Unit (MCU). In the past, this circuit was located inside the motherboard chipset, in the north bridge chip. But nowadays, most processors have this component embedded, so the CPU has a dedicated memory bus connecting it to the main memory.

Main memory uses DRAM (Dynamic Random Access Memory), technology that supports large capacities at reasonable cost points. When comparing DRAM modules, people usually look at memory density and memory speed, besides its price, of course. Memory density defines how much memory the module has, measured in GB. Obviously the more available memory the better as it is a precious resource used by the OS and applications.

Performance of main memory is described by latency and bandwidth. Memory latency is the time elapsed between the memory access request is issued and when the data is available to use by CPU. Memory bandwidth defines how many bytes can be fetch per some period of time, usually measured in gigabytes per second.

3.6.2.1 DDR DDR (Double Data Rate) DRAM technology is the predominant DRAM technology supported by most CPUs. Historically, DRAM bandwidths have improved every generation while the DRAM latencies have stayed the same or even increased. Table 2 shows the top data rate, peak bandwidth, and the corresponding reading latency for the last three generations of DDR technologies. The data rate is measured as a million transfers per sec (MT/s). The latencies shown in this table correspond to the latency in the DRAM device itself. Typically, the latencies as seen from the CPU pipeline (cache miss on a load to use) are higher (in the 50ns-150ns range) due to additional latencies and queuing delays incurred in the cache controllers, memory controllers, and on-die interconnects. See an example of measuring observed memory latency and bandiwdth in Section 4.10.

Table 2: Performance characteristics for the last three generations of DDR technologies.

DDR Generation	Year	Highest Data Rate(MT/s)	Peak Bandwidth (Gbytes/s)	In-device Read Latency(ns)
DDR3	2007	2133	12.8	10.3
DDR4	2014	3200	25.6	12.5
DDR5	2020	6400	51.2	14

It is worth to mention that DRAM chips require memory cells being periodically refreshed. Because the bit value is stored as the presence of an electric charge on a tiny capacitor, it can lose its charge as the time passes. To prevent this, there is a special circuitry that reads each cell and writes it back, effectively restoring the capacitor's charge. While a DRAM chip is in its refresh procedure, it is not serving memory access requests.

DRAM module is organized as sets of DRAM chips. Memory *rank* is a term that describes how many sets of DRAM chips exist on a module. For example, a single-rank (1R) memory module contains one set of DRAM chips. A

dual-rank (2R) memory module has two sets of DRAM chips, therefore doubling the capacity of a single-rank module. Likewise, there are quad-rank (4R) and octa-rank (8R) memory modules available for purchase.

Each rank consists of multiple DRAM chips. Memory *width* defines how wide the bus of each DRAM chip is. And since each rank is 64-bits wide (or 72-bits wide for ECC RAM), it also defines the number of DRAM chips present within the rank. Memory width can be one of three values: `x4`, `x8` or `x16`, which define how wide is the bus that goes to each chip. As an example, Figure 14 shows the organization of 2Rx16 dual-rank DRAM DDR4 module, total 2GB capacity. There are four chips in each rank, with a 16-bit wide bus. Combined, the four chips provide 64-bit output. The two ranks are selected one at a time through a rank select signal.

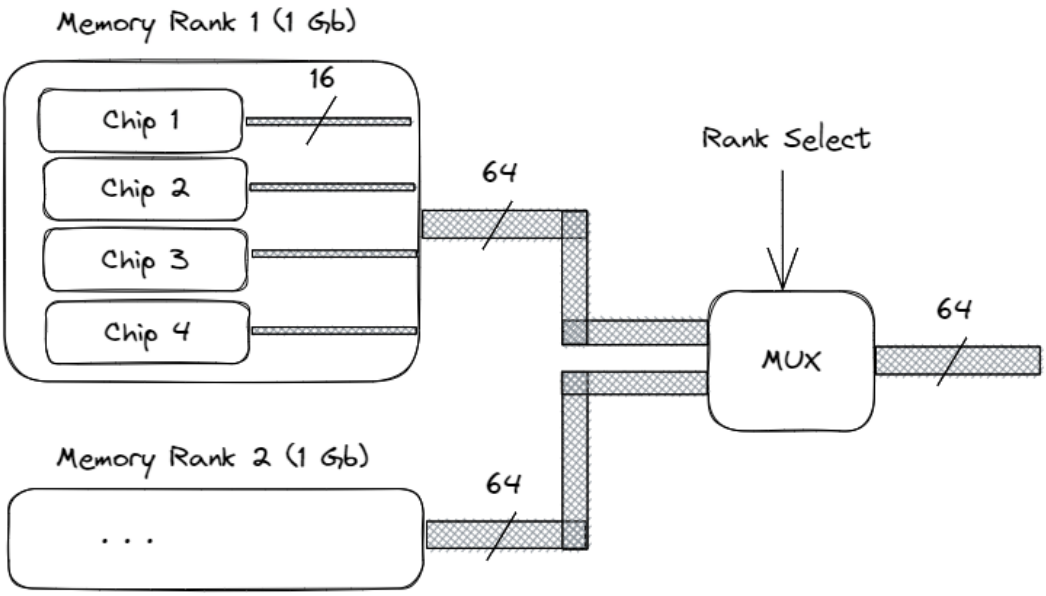


Figure 14: Organization of 2Rx16 dual-rank DRAM DDR4 module, total 2GB capacity.

There is no direct answer whether performance of single-rank or dual-rank is better as it depends on the type of application. Switching from one rank to another through rank select signal needs additional clock cycles, which may increase the access latency. On the other hand, if a rank is not accessed, it can go through its refresh cycles in parallel while other ranks are busy. As soon as the previous rank completes data transmission, the next rank can immediately start its transmission. Also, single-rank modules produce less heat and are less likely to fail.

Going further, we can install multiple DRAM modules in a system to not only increase memory capacity, but also memory bandwidth. Setups with multiple memory channels are used to scale up the communication speed between the memory controller and the DRAM.

A system with a single memory channel has a 64-bit wide data bus between the DRAM and memory controller. The multi-channel architectures increase the width of the memory bus, allowing DRAM modules to be accessed simultaneously. For example, the dual-channel architecture expands the width of the memory data bus from 64 bit to 128 bit, doubling the available bandwidth, see Figure 15. Notice, that each memory module, is still a 64-bit device, but we connect them differently. It is very typical nowadays for server machines to have four and eight memory channels.

Alternatively, you could also encounter setups with duplicated memory controllers. For example, a processor may have two integrated memory controllers, each of them capable of supporting several memory channels. The two controllers are independent and only view their own slice of the total physical memory address space.

We can do a quick calculations to determine the maximum memory bandwidth for a given memory technology, using a simple formula below:

$$\text{Max. Memory Bandwidth} = \text{Data Rate} \times \text{Bytes per cycle}$$

For example, for a single-channel DDR4 configuration, the data rate is `2400 MT/s` and 64 bits or 8 bytes can be transfered each memory cycle, thus the maximum bandiwdth equals to `2400 * 8 = 19.2 GB/s`. Dual-channel or

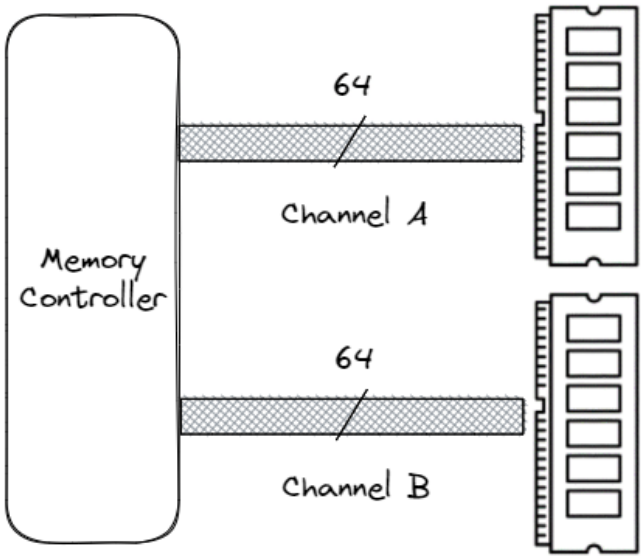


Figure 15: Organization of a dual-channel DRAM setup.

dual memory controller setups double the bandwidth to `38.4 GB/s`. Remember though, that those numbers are theoretical maximums, that assume that a data transfer will occur at each memory clock cycle, which in fact never happens in practice. So, when measuring actual memory speed, you will always see a value lower than the maximum theoretical transfer bandwidth.

To enable multi-channel configuration, you need to have a CPU and a motherboard that supports such architecture and install an even number of identical memory modules in the correct memory slots on the motherboard. The quickest way to check the setup on Windows is by running a hardware identification utility like `CPU-Z` or `HwInfo`, on Linux one can use `dmidecode` command. But also, you can run memory bandwidth benchmarks like Intel `mlc` or `Stream`.

To make use of multiple memory channels in a system, there is a technique called interleaving. It spreads adjacent addresses within a page across multiple memory devices. Example of a 2-way interleaving for sequential memory accesses is shown in Figure 16. As before, we have dual-channel memory configuration (channels A and B) with two independent memory controllers. Modern processors interleave per four cache lines (256 bytes), i.e. first four adjacent cache lines go to the channel A, and then the next set of four cache lines go to the channel B.

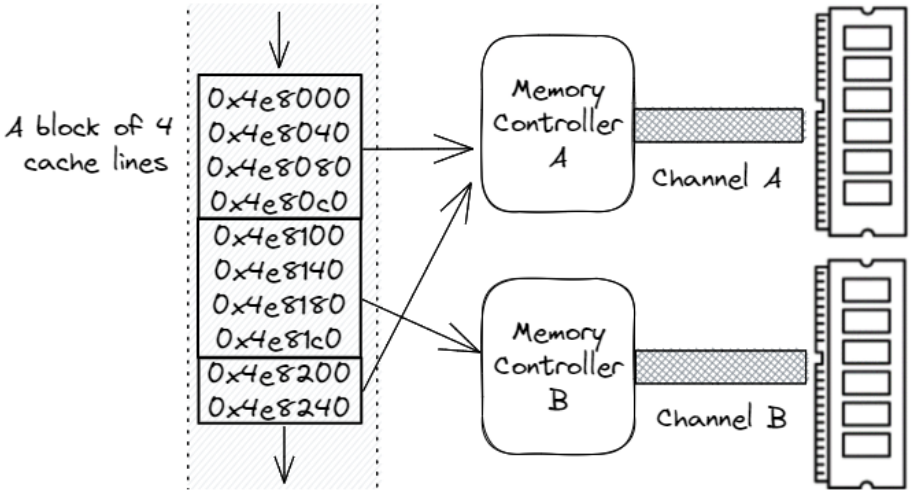


Figure 16: 2-way interleaving for sequential memory access.

Without interleaving, consecutive adjacent accesses would be sent to the same memory controller, not utilizing the second available controller. While using the technique increases hardware-level parallelism and allows to effectively utilize all available bandwidth from the memory devices in a setup. For most workloads, performance is maximized when all the channels are populated as it spreads a single memory region across as many DRAM modules as possible.

While increased memory bandwidth is generally good, it does not always translate into better system performance and is highly dependent on the application. On the other hand, it's important to watch out for available and utilized memory bandwidth, because once it becomes the primary bottleneck, the application stops scaling, i.e. adding more cores doesn't make it run faster.

3.6.2.2 GDDR and HBM Besides multi-channel DDR, there are other technologies that target workloads where higher memory bandwidth is required to achieve greater performance. Technologies such as GDDR (Graphics DDR) and HBM (High Bandwidth Memory) are the most notable ones. They find their use in high-end graphics, high-performance computing such as climate modeling, molecular dynamics, physics simulation, but also in autonomous driving, and of course, AI/ML. They are a natural fit there because such applications require moving large amounts of data very quickly.

GDDR was primary designed for graphics and nowadays it is used on virtually every high-performance graphics card. While GDDR shares some charasteristics with DDR, it is also quite different. While DRAM DDR is designed for lower latencies, GDDR is built for much higher bandwidth, because it is located in the same package as the processor chip itself. Similar to DDR, the GDDR interface transfers two 32 bit (64-bit total) wide data words per clock cycle. The latest GDDR6X standard can achieve up to 168 GB/s bandwidth, operating at a relatively low 656 MHz frequency.

HBM is a new type of CPU/GPU memory that vertically stacks memory chips, also called 3D stacking. Similar to GDDR, HBM drastically shortens the distance data needs to travel to reach a processor. The main difference from DDR and GDDR, is that HBM memory bus is very wide: 1024 bits per each HBM stacks. It allows HBM to achieve ultra-high bandwidth. The latest HBM3 standard supports up to 665 GB/s bandwidth per package. It also operates at a low frequency of 500 Mhz and has a memory density of up to 48 GB per package.

A system with HBM onboard will be a good choice if you're looking to get as much memory bandwidth as you can get. However, at the time of writing, this technology is quite expensive. As GDDR is predominantly used in graphics cards, HBM may be a good option to accelerate certain workloads that run on CPU. In fact, we start seeing first x86 general purpose server chips with integrated HBM.

3.7 Virtual Memory

Virtual memory is the mechanism to share the physical memory attached to a CPU with all the processes executing on the CPU. Virtual memory provides a protection mechanism, restricting access to the memory allocated to a given process from other processes. Virtual memory also provides relocation, the ability to load a program anywhere in physical memory without changing the addressing in the program.

In a CPU that supports virtual memory, programs use virtual addresses for their accesses. But while user code operates on virtual addresses, retrieving the data from memory requires physical address. Also, to effectively manage the scarce physical memory, it is divided into pages. Thus applications operate on a set of pages that an operating system has provided.

Address translation is required for accessing data as well as the code (instructions). The mechanism for a system with a page size of 4KB is shown on Figure 17. The virtual address is split into two parts. The virtual page number (52 most signicant bits) is used to index into the page table to produce a mapping between the virtual page number and the corresponding physical page. To offset within a 4KB page we need 12 bits, the rest 52 bits of a 64-bit pointer can be used for the address of page itself. Notice that the offset whitin a page (12 least signicant bits) does not require translation, and it is used "as-is" to access the physical memory location.

The page table can either be single-level or nested. Figure 18 shows one example of a 2-level page table. Notice, how the address gets split into more piecies. First thing to mention, is that 16 most significant bits are not used. This can seem like a waste of bits, but even with the remaining 48 bits we can address 256 TB of total memory (2^{48}). Some applications use those unused bits to keep metadata, also known as *pointer tagging*.

Nested page table is a radix tree that keeps physical page addresses along with some metadata. To find a translation for such a 2-level page table, we first use bits 32..47 as an index into the Level-1 page table also known as *page*

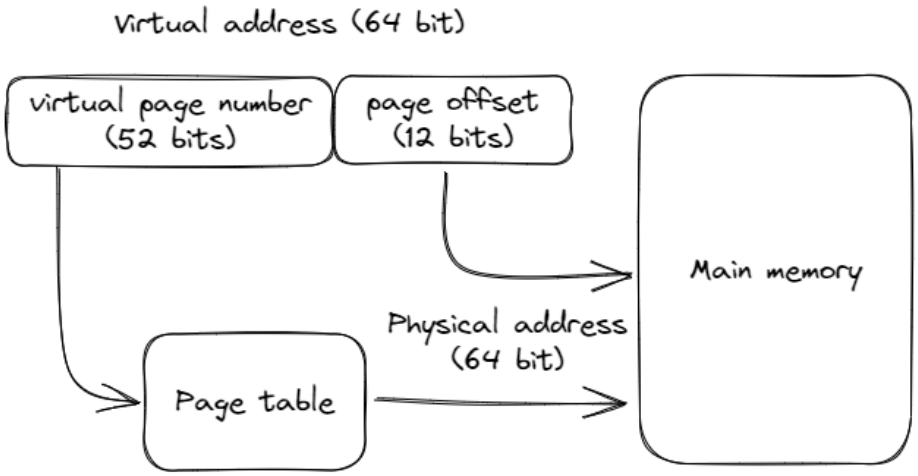


Figure 17: Virtual-to-physical address translation for 4KB pages.

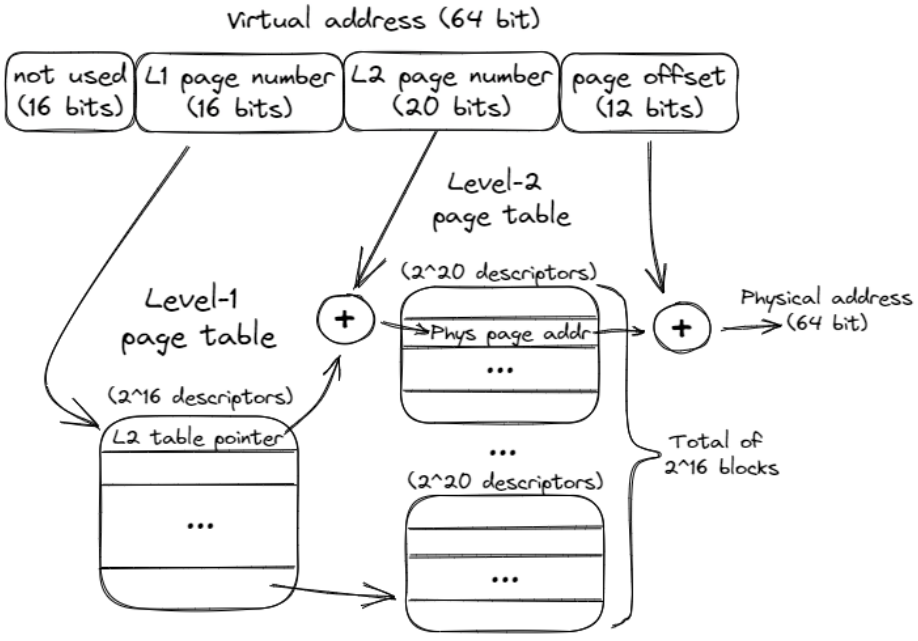


Figure 18: Exmaple of a 2-level page table.

table directory. Every descriptor in the directory points to one of the 2^{16} blocks of Level-2 tables. Once we find the appropriate L2 block, we use bits 12..31 to find the physical page address. Concatenating it with the page offset (bits 0..11) gives us the physical address, which can be used to retrieve the data from the DRAM.

The exact format of the page table is strictly dictated by the CPU for the reasons we will discuss a few paragraphs later. Thus the variations of page table organization are limited by what a CPU supports. Nowadays it is common to see 4- and 5-level page tables. Modern CPUs support 4-level page table with 48 bit pointers (256 TB of total memory) and 5-level page tables with 57 bit pointers (128 PB of total memory).

Breaking page table into multiple levels doesn't change the total addressable memory. However, a nested approach does not require to store the entire page table as a contiguous array and does not allocate blocks that have no descriptors. This saves memory space but adds overhead when traversing the page table.

Failure to provide a physical address mapping is called a *page fault*. It occurs if a requested page is invalid or is not currently in the main memory. The two most common reasons are: 1) OS committed to allocating a page but hasn't yet backed it with a physical page, 2) accessed page was swapped out to disk and is not currently stored in RAM.

3.7.1 Translation Lookaside Buffer (TLB)

A search in a hierarchical page table could be expensive, requiring traversing through the hierarchy potentially making several indirect accesses. Such traversal is usually called *page walk*. To reduce the address translation time, CPUs support a hardware structure called translation lookaside buffer (TLB) to cache the most recently used translations. Similar to regular caches, TLBs are often designed as a hierarchy of L1 ITLB (Instructions), L1 DTLB (Data), followed by a shared (instructions and data) L2 STLB. To lower the memory access latency, TLB and cache lookups happen in parallel, because data caches operate on virtual addresses and do not require prior address translation

TLB hierarchy keep translations for a relatively large memory space. Still, misses in TLB can be very costly. To speed up handling of TLB misses, CPUs have a mechanism called *HW page walker*. Such unit can perform a page walk directly in HW by issuing the required instructions to traverse the page table, all without interrupting the kernel. This is the reason why the format of the page table is dictated by the CPU, to which OS'es have to comply. High-end processors have several HW page walkers that can handle multiple TLB misses simultaneously. With all the acceleration offered by modern CPUs, TLB misses cause performance bottlenecks for many applications.

3.7.2 Huge Pages

Having a small page size allows to manage the available memory more efficiently and reduce fragmentation. The drawback though is that it requires to have more page table entries to cover the same memory region. Consider two page sizes: 4KB, which is a default on x86, and 2MB *huge page* size. For an application that operates on 10MB data, we need 2560 entries in first case, and just 5 entries if we would map the address space onto huge pages. Those are named *Huge Pages* on Linux, *Super Pages* on FreeBSD, and *Large Pages* on Windows, but they all mean the same thing. Through the rest of the book we will refer to it as Huge Pages.

Example of an address that points to the data within a huge page is shown in Figure 19. Just like with a default page size, the exact address format when using huge pages is dictated by the HW, but luckily we as programmers usually don't have to worry about it.

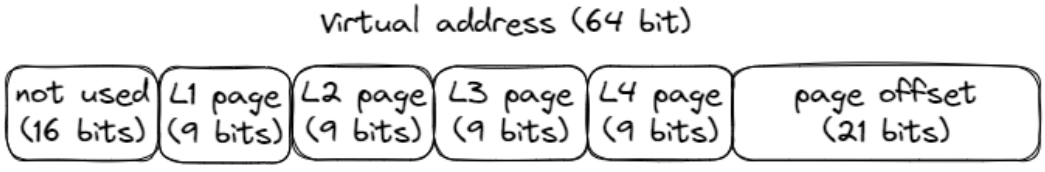


Figure 19: Virtual address that points within a 2MB page.

Using huge pages drastically reduces the pressure on the TLB hierarchy since fewer TLB entries are required. It greatly increases the chance of a TLB hit, you will see examples later in the book. The downsides of using huge pages are memory fragmentation and, in some cases, non-deterministic page allocation latency. It is harder for the operating system to manage large blocks of memory and to ensure effective utilization of available memory. To

satisfy a 2MB huge page allocation request at runtime, an OS needs to find a contiguous chunk of 2MB. If unable to find, it needs to reorganize the pages, resulting in longer allocation latency. We will discuss how to use huge pages to reduce the frequency of TLB misses in the second part of the book.

3.8 Modern CPU Design

To see how all the concepts we talked about in this chapter are used in practice, let's take a look at the implementation of Intel's 12th generation core, Goldencove, which became available in 2021. This core is used as P-core inside Alderlake and Sapphire Rapids platforms. Figure 20 shows the block diagram of the Goldencove core. Notice, that this section only describes a single core, not the entire processor. So, we will skip discussion about frequencies, core counts, L3 caches, core interconnects, memory latency and bandwidth, and other things.

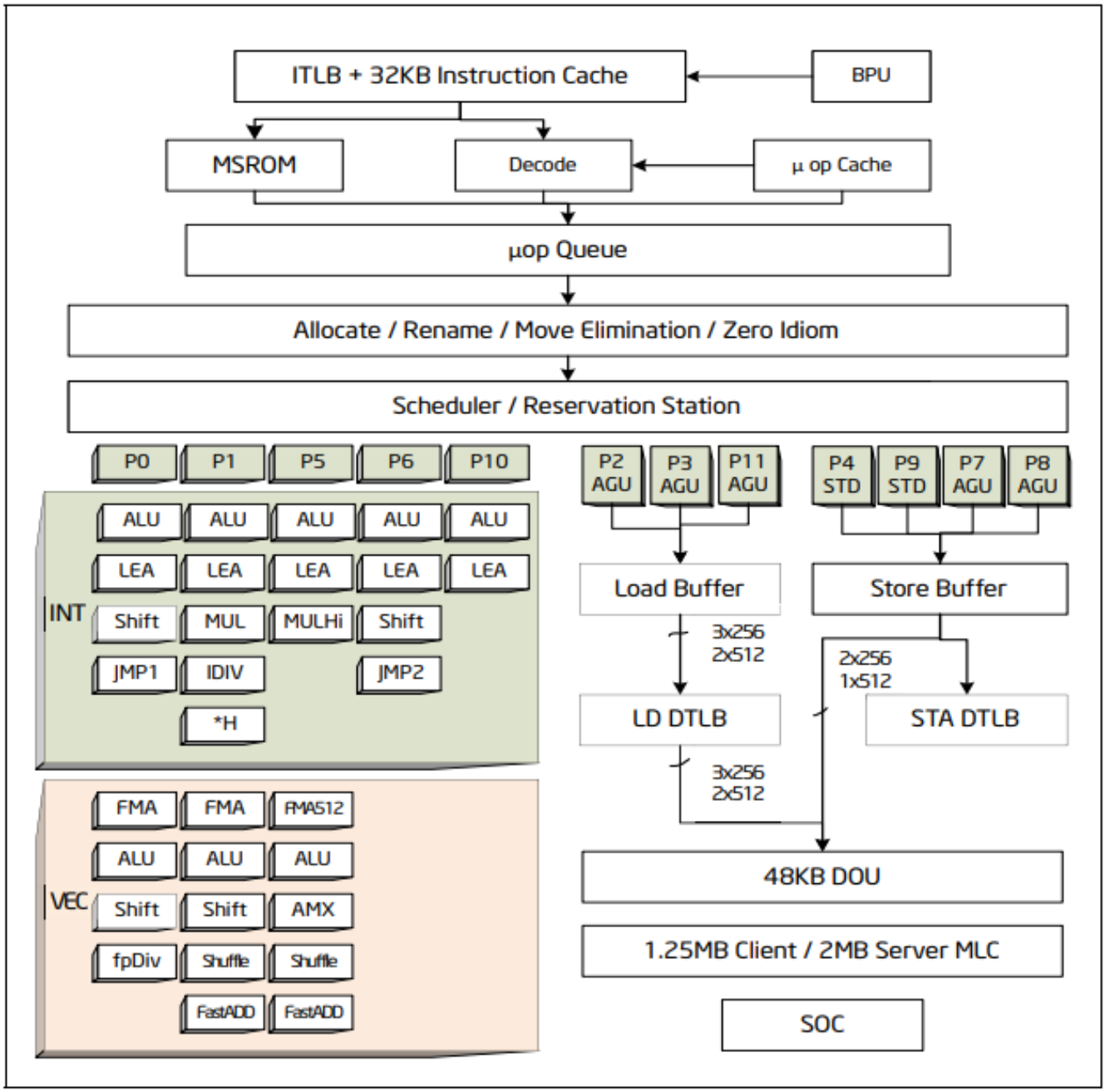


Figure 20: Block diagram of a CPU Core in the Intel GoldenCove Microarchitecture. © *Image from [Intel, 2023b].*

The core is split into an in-order front-end that fetches and decodes x86 instructions into u-ops and a 6-wide

superscalar, out-of-order backend. The Goldencove core supports 2-way SMT. It has a 32KB first-level instruction cache (L1 I-cache), and a 48KB first-level data cache (L1 D-cache). The L1 caches are backed up by a unified 1.25MB (2MB in server chips) second-level cache, the L2 cache. The L1 and L2 caches are private to each core. At the end of this section we also take a look at the TLB hierarchy.

3.8.1 CPU Front-End

The CPU Front-End consists of a number of data structures that serve the main goal to efficiently fetch and decode instructions from memory. Its main purpose is to feed prepared instructions to the CPU Back-End, which is responsible for the actual execution of instructions.

Technically, instruction fetch is the first stage to execute an instruction. But once a program reaches a steady state, branch predictor unit (BPU) steers the work of the CPU Front-End. That's the reason for the arrow that goes from the BPU to the instruction cache. The BPU predicts direction of all branch instructions and steers the next instruction fetch based on this prediction.

The heart of the BPU is a branch target buffer (BTB) with 12K entries, which keeps the information about branches and their targets which is used by the prediction algorithms. Every cycle, the BPU generates next address to fetch and passes it to the CPU Front-End.

The CPU Front-End fetches 32 bytes per cycle of x86 instructions from the L1 I-cache. This is shared among the two threads, so each thread gets 32 bytes every other cycle. These are complex, variable-length x86 instructions. First, the pre-decode determines and marks the boundaries of the variable instructions by inspecting the instruction. In x86, the instruction length can range from 1-byte to 15-bytes instructions. This stage also identifies branch instructions. The pre-decode stage moves up to 6 instructions (also referred to as Macro Instructions) to the instruction queue (not shown on the block diagram) that is split between the two threads. The instruction queue also supports a macro-op fusion unit that detects that two macroinstructions can be fused into a single micro operation (UOP). This optimization saves bandwidth in the rest of the pipeline.

Later, up to six pre-decoded instructions are sent from the instruction queue to the decoder unit every cycle. The two SMT threads alternate every cycle to access this interface. The 6-way decoder converts the complex macro-Ops into fixed-length UOPs. Decoded UOPs are queued into the Instruction Decode Queue (IDQ), labeled as "uop Queue" on the diagram.

A major performance-boosting feature of the front-end is the Decoded Stream Buffer (DSB) or the UOP Cache. The motivation is to cache the macro-ops to UOPs conversion in a separate structure that works in parallel with the L1 I-cache. When the BPU generates new address to fetch, the DSB is also checked to see if the UOPs translations are already available in the DSB. Frequently occurring macro-ops will hit in the DSB, and the pipeline will avoid repeating the expensive pre-decode and decode operations for the 32 bytes bundle. The DSB can provide eight UOPs per cycle and can hold up to 4K entries.

Some very complicated instructions may require more UOPs than decoders can handle. UOPs for such instruction are served from Microcode Sequencer (MSROM). Examples of such instructions include HW operation support for string manipulation, encryption, synchronization, and others. Also, MSROM keeps the microcode operations to handle exceptional situations like branch misprediction (which requires a pipeline flush), floating-point assist (e.g., when an instruction operates with a denormalized floating-point value), and others. MSROM can push up to 4 uops per cycle into the IDQ.

The Instruction Decode Queue (IDQ) provides the interface between the in-order front-end and the out-of-order backend. IDQ queues up the UOPs in order. The IDQ can hold 144 uops per logical processor in single thread mode, or 72 uops per thread when SMT is active. This is where the in-order CPU Front-End finishes and the out-of-order CPU Back-End starts.

3.8.2 CPU Back-End

The CPU Back-End employs an OOO engine that executes instructions and stores results. The heart of the CPU backend is the 512 entry ReOrder buffer (ROB). This unit is reffered as "Allocate / Rename" on the diagram. It serves a few purposes. First, it provides register renaming. There are only 16 general-purpose integer and 32 vector/SIMD architectural registers, however, the number of physical registers is much higher.[46] Physical registers

[46] Around 300 physical GPRs and a similar number of vector registers. The official number is not diclosed.

are located in a structure called physical register file (PRF). The mappings from architecture-visible registers to the physical registers are kept in the register alias table (RAT).

Second, ROB allocates execution resources. When an instruction enters the ROB, a new entry gets allocated and resources are assigned to it, mainly execution port and the output physical register. ROB can allocate up to 6 UOPs per cycle.

Third, ROB tracks the speculative execution. When the instruction finished its execution its status gets updated and it stays there until the previous instructions also finish. It' done that way because instructions are always retired in program order. Once the instruction retires, its ROB entry gets deallocated and results of the instruction become visible. The retiring stage is wider than the allocation: ROB can retire 8 instruction per cycle.

There are certain operations which processors handle in a specific manner, often called idioms, which require no or less costly execution. Processors recognize such cases and allow them to run faster then regular instructions. Here are some of such cases:

- **Zeroing**: to assign zero to a register, compilers often use `XOR / PXOR / XORPS / XORPD` instructions, e.g. `XOR RAX, RAX`, which are preferred by compilers instead of the equivalent `MOV RAX, 0x0` instruction as the XOR encoding uses fewer encoding bytes. Such zeroing idioms are not executed as any other regular instruction and are resolved in the CPU front-end, which saves execution resources. The instruction later retires as usual.
- **Move elimination**: similarly to the previous one, register-to-register mov operations, e.g. `MOV RAX, RBX`, are executed with zero cycle delay.
- **NOP instruction**: `NOP` is often used for padding or alignment purposes. It simply gets marked as completed without allocating it into the reservation station.
- **Other bypases**: CPU architects also optimized certain arithmetical operations. For example, multiplying any number by one will always gives the same number. The same goes for dividing any number by one. Multiplying any number by zero always gives the same number, etc. Some CPUs can recognize such cases in runtime and run them with shorter latency than regular multiplication or divide.

The "Scheduler / Reservation Station" (RS) is the structure that tracks the availability of all resources for a given UOP and dispatches the UOP to the assigned port once it is ready. When an instruction enters the RS, scheduler starts tracking its data dependencies. Once all the source operands become available, the RS tries to dispatch it to a free execution port. The RS has fewer entries than the ROB. It can dispatch up to 6 UOPs per cycle.

As shown in Figure 20, there are 12 execution ports:

- Ports 0, 1, 5, 6, and 10 provide all the integer, FP, and vector ALU. UOPs dispatched to those ports do not require memory operations.
- Ports 2, 3, and 11 are used for address generation (AGU) and for load operations.
- Ports 4 and 9 are used for store operations (STD).
- Ports 7 and 8 are used for address generation.

A dispatched arithmetical operation can go to either INT or VEC execution port. Integer and Vector/FP register stacks are located separately. Operations that move values from Int stack to FP and vice-versa (e.g. convert, extraxt, insert) incur additional penalty.

3.8.3 Load-Store Unit

The Goldencove core can execute up to three loads and up to two stores per cycle. Once a load or a store leaves the scheduler, the load-store (LS) unit is responsible for accessing the data and saving it in a register. The LS unit has a load queue (LDQ, labeled as "Load Buffer") and a store queue (STQ, labeled as "Store Buffer"), their sizes are not disclosed.[47] Both LDQ and STQ receive operations at dispatch from the scheduler.

When a new memory load request comes, the LS queries the L1 cache using a virtual address and looks up the physical address translation in the TLB. Those two operations are initiated simultaneously. The size of L1 D-cache is 48KB. If both operations result in a hit, the load delivers data to the integer unit or the floating-point unit and leaves the LDQ. Similarly, a store would write the data to the data cache and exit the STQ.

In case of a L1 miss, the hardware initiates a query of the (private) L2 cache tags. The L2 cache comes in two variants: 1.25MB for client and 2MB for server processors. While the L2 cache is being queried, a fill buffer (FB) is

[47] LDQ and STQ sizes are not disclosed, but people have measured 192 and 114 entries respectively.

allocated, which will keep the cache line once it arrives. The Goldencove core has 16 fill buffers. As a way to lower the latency, a speculative query is sent to the L3 cache in parallel with L2 cache lookup.

If two loads access the same cache line, they will hit the same FB. Such two loads will be "glued" together and only one memory request will be initiated. The LS unit dynamically reorders operations, supporting both loads bypassing older loads and loads bypassing older non-conflicting stores. Also, the LS unit supports store-to-load forwarding when there is an older store that contains all of the load's bytes, and the store's data has been produced and is available in the store queue.

In case the L2 miss is confirmed, the load continues to wait for the results of L3 cache, which incurs much higher latency. From that point, the request leaves the core and enters the "uncore", the term you may frequently see in profiling tools. The outstanding misses from the core are tracked in the Super Queue (SQ), which can track up to 48 uncore requests. In a scenario of L3 miss, the processor begins to set up a memory access. Further details are beyond the scope of this chapter.

When a store happens, in a general case, to modify a memory location, the processor needs to load the full cache line, change it, and then write it back to memory. If the address to write is not in the cache, it goes through a very similar mechanism as with loads to bring that data in. The store cannot be complete until the data is not written to the cache hierarchy.

Of course, there are a few optimizations done for store operations as well. First, if we're dealing with a store or multiple adjacent stores (aka *streaming stores*) that modify entire cache line, there is no need to read the data first as all of the bytes will be clobbered anyway. So, the processor will try to combine writes to fill entire cache lines. If this succeeds no memory read operation is needed at all.

Second, write combining allows multiple stores to be assembled and written further out in the cache hierarchy as a unit. So, if multiple stores modify the same cache line, only one memory write will be issued to the memory subsystem. Modern processors have a data structure called *store buffer* that tries to combine stores. A store instruction copies the data that will be written from a register into the store buffer. From there it may be written to the L1 cache or it may be combined with other stores to the same cache line. The store buffer capacity is limited, so it can hold requests for partial writing to a cache line only for some time. However, while the data sits in the store buffer waiting to be written, other load instructions can read the data straight from the store buffers (store-to-load forwarding).

Finally, if we happen to read the data before overwriting it, the cache line typically stays in the cache, displacing some other line. This behavior can be altered with the help of a *non-temporal* store, that will not keep the modified line in the cache. It is useful in situations when we know that we don't need the data once we have changed it. Non-temporal loads a stores help to utilize cache space more efficiently by not evicting other data that might be needed soon.

3.8.4 TLB Hierarchy

Recall from the previous discussion, translations from virtual to physical addresses are cached in TLB. Golden Cove's TLB hierarchy is presented in Figure 21. Similar to a regular data cache, it has two levels, where level 1 has separate instances for instructions (ITLB) and data (DTLB). L1 ITLB has 256 entries for regular 4K pages and covers the memory space of 256 * 4KB equals 1MB, while L1 DTLB has 96 entries that covers 384 KB.

The second level of the hierarchy (STLB) caches translations for both instructions and data. It is a larger storage that serves requests that miss in the L1 TLBs. L2 STLB can accomdate 2048 most recent data and instruction page address translations, which covers a total of 8MB of memory space. There are fewer entries available for 2M huge pages: L1 ITLB has 32 entries, L1 DTLB has 32 entries, and L2 STLB can only use 1024 entries that are also shared regular 4K pages.

In case a translation was not found in the TLB hierarchy, it has to be retrieved from the DRAM by "walking" the kernel page tables. There is a mechanism for speeding up such scenarios, called HW page walker. Recall that the page table is built as a radix tree of sub-tables, with each entry of the sub-table holding a pointer to the next level of the tree.

The key element to speed up the page walk procedure is a set of Paging-Structure Caches[48] that caches the hot entries in the page table structure. For the 4-level page table, we have the least significant twelve bits (11:0) for page

[48] AMD's equivalent is called Page Walk Caches.

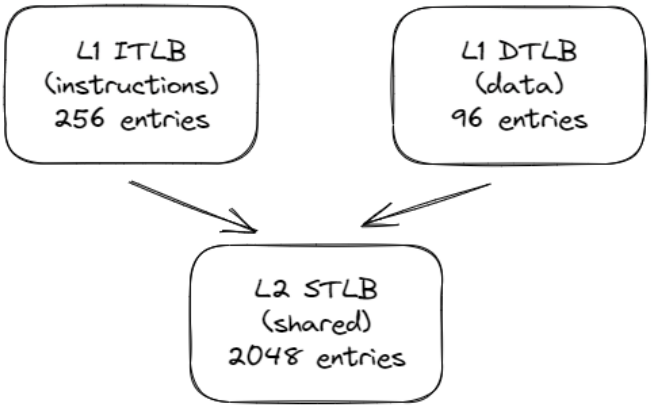


Figure 21: TLB hierarchy of Golden Cove.

offset (not translated), and bits 47:12 for the page number. While each entry in a TLB is an individual complete translation, Paging-Structure Caches cover only the upper 3 levels (bits 47:21). The idea is to reduce the number of loads required to execute in case of a TLB miss. For example, without such caches we would have to execute 4 loads, which would add latency to the instruction completion. But with the help of the Paging-Structure Caches, if we find a translation for the levels 1 and 2 of the address (bits 47:30), we only have to do the remaining 2 loads.

The Goldencove microarchitectures has four dedicated page walkers, which allows it to process 4 page walks simultaneously. In the event of a TLB miss, these HW units will issue the required loads into the memory subsystem and populate the TLB hierarchy with new entries. The page-table loads generated by the page walkers can hit in L1, L2, or L3 caches (details are not disclosed). Finally, page walkers can anticipate a future TLB miss and speculatively do a page walk to update TLB entries before a miss actually happens.

Goldencove specification doesn't disclose how resources are shared between two SMT threads. But in general, caches, TLBs and execution units are fully shared to improve the dynamic utilization of those resources. On the other hand, buffers for staging instructions between major pipe stages are either replicated or partitioned. These buffers include IDQ, ROB, RAT, RS, LDQ and STQ. PRF is also replicated.

3.9 Performance Monitoring Unit

Every modern CPU provides means to monitor performance, which are aggregated into the Performance Monitoring Unit (PMU). It incorporates features that help developers in analyzing the performance of their applications. An example of a PMU in a modern Intel CPU is provided in Figure 22. Most modern PMUs have a set of Performance Monitoring Counters (PMC) that can be used to collect various performance events that happen during the execution of a program. Later in Section 5.3, we will discuss how PMCs can be used for performance analysis. Also, the PMU has other features that enhance performance analysis, like LBR, PEBS, and PT, for which entire chapter 6 is devoted.

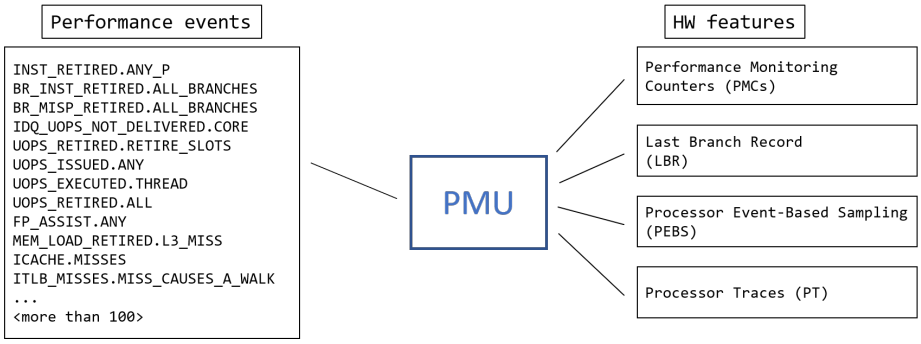


Figure 22: Performance Monitoring Unit of a modern Intel CPU.

As CPU design evolves with every new generation, so do their PMUs. It is possible to determine the version of the

PMU in your CPU using the `cpuid` command, as shown in Listing 5. A similar information can be extracted from the kernel message buffer by checking the output of `dmesg` command. Characteristics of each Intel PMU version, as well as changes to the previous version, can be found in [Intel, 2023b, Volume 3B, Chapter 20].

Listing 5 Querying your PMU

```
$ cpuid
...
Architecture Performance Monitoring Features (0xa/eax):
      version ID                               = 0x4 (4)
      number of counters per logical processor = 0x4 (4)
      bit width of counter                     = 0x30 (48)
...
Architecture Performance Monitoring Features (0xa/edx):
      number of fixed counters    = 0x3 (3)
      bit width of fixed counters = 0x30 (48)
...
```

3.9.1 Performance Monitoring Counters

If we imagine a simplified view of the processor, it may look something like what is shown in Figure 23. As we discussed earlier in this chapter, a modern CPU has caches, a branch predictor, an execution pipeline, and other units. When connected to multiple units, a PMC can collect interesting statistics from them. For example, it can count how many clock cycles have passed, how many instructions executed, how many cache misses or branch mispredictions happened during that time, and other performance events.

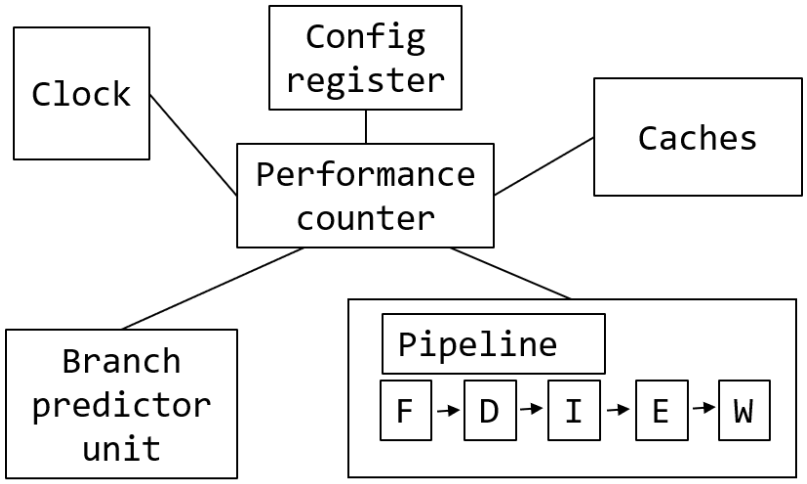


Figure 23: Simplified view of a CPU with a performance monitoring counter.

Typically, PMCs are 48-bit wide, which enables analysis tools to run for a long time without interrupting a program's execution.[49] Performance counter is a HW register implemented as a Model Specific Register (MSR). That means that the number of counters and their width can vary from model to model, and you can not rely on the same number of counters in your CPU. You should always query that first, using tools like `cpuid`, for example. PMCs are accessible via the `RDMSR` and `WRMSR` instructions, which can only be executed from kernel space. Luckily, you only have to care about this if you're a developer of a performance analysis tool, like Linux perf or Intel Vtune profiler. Those tools handle all the complexity of programming PMCs.

When engineers analyze their applications, it is common for them to collect the number of executed instructions and elapsed cycles. That is the reason why some PMUs have dedicated PMCs for collecting such events. Fixed counters

[49] When the value of PMCs overflows, the execution of a program must be interrupted. SW then should save the fact of an overflow. We will discuss it in more details later.

always measure the same thing inside the CPU core. With programmable counters, it's up to the user to choose what they want to measure.

For example, in the Intel Skylake architecture (PMU version 4, see Listing 5), each physical core has three fixed and eight programmable counters. The three fixed counters are set to count core clocks, reference clocks, and instructions retired (see Chapter 4 for more details on these metrics). AMD Zen4 and ARM Neoverse V1 cores support 6 programmable performance monitoring counters per processor core, no fixed counters.

It's not unusual for a PMU to provide more than one hundred events available for monitoring. Figure 22 shows just a small part of all the performance events available for monitoring on a modern Intel CPU. It's not hard to notice that the number of available PMCs is much smaller than the number of performance events. It's not possible to count all the events at the same time, but analysis tools solve this problem by multiplexing between groups of performance events during the execution of a program (see Section 5.3.3).

- For Intel CPUs, the complete list of performance events can be found in [Intel, 2023b, Volume 3B, Chapter 20] or at perfmon-events.intel.com.
- ADM doesn't publish a list of performance monitoring events for every AMD processor. Curious readers may find some information in the Linux perf source code[50]. Also, you can list performance events available for monitoring using AMD uProf command line tool. General information about AMD performance counters can be found in [AMD, 2023, 13.2 Performance Monitoring Counters].
- For ARM chips, performance events are not strictly defined. Vendors implement cores following an ARM architecture, but performance events vary widely, both in what they mean and what events are supported. For the ARM Neoverse V1 processor, that ARM designs themselves, the list of performance events can be found in [Arm, 2022b].

Questions and Exercises

1. Describe pipelining, out-of-order and speculative execution.
2. How register renaming helps to speed up execution?
3. Describe spatial and temporal locality.
4. What is the size of the cache line in majority of modern processors?
5. Name components that constitue the CPU frontend and backend?
6. What is the organization of the 4-level page table? What is a page fault?
7. What is the default page size in x86 and ARM architectures?
8. What role does TLB (Translation Lookaside Buffer) play?

Chapter Summary

- Instruction Set Architecture (ISA) is a fundamental contract between SW and HW. ISA is an abstract model of a computer that defines the set of available operations and data types, set of registers, memory addressing, and other things. You can implement a specific ISA in many different ways. For example, you can design a "small" core that prioritizes power efficiency or a "big" core that targets high performance.
- The details of the implementation are incapsulated in a term CPU "microarchitecture". It has been a topic that was researched by thousands of computer scientists for a long time. Through the years, many smart ideas were invented and implemented in mass-market CPUs. The most notable are pipelining, out-of-order execution, superscalar engines, speculative execution and SIMD processors. All these techniques help exploit Instruction-Level Parallelism (ILP) and improve singe-threaded performance.
- In parallel with single-threaded performance, HW designers began pushing multi-threaded performance. Vast majority or modern client-facing devices have a processor with multiple cores inside. Some cores double the number of observable CPU cores with the help of Simultaneous Multithreading (SMT). SMT allows multiple software threads to run simultaneously on the same physical core using shared resources. More recent technique in this direction is called "hybrid" processors that combines different types of cores in a single package to better support diversity of workloads.
- Memory hierarchy in modern computers includes several level of caches that reflect different tradeoffs in speed of access vs. size. L1 cache tends to be closest to a core, fast but small. L3/LLC cache is slower but also bigger. DDR is the predominant DRAM technology integrated in most platforms. DRAM modules vary in the

[50] Linux source code for AMD cores - https://github.com/torvalds/linux/blob/master/arch/x86/events/amd/core.c

number of ranks and memory width which may have a slight impact on system performance. Processors may have multiple memory channels to access more than one DRAM module simultaneously.

- Virtual memory is the mechanism for sharing the physical memory with all the processes running on the CPU. Programs use virtual addresses in their accesses, which get translated into physical addresses. The memory space is split into pages. The default page size on x86 is 4KB, on ARM is 16KB. Only the page address gets translated, the offset within the page is used as it is. The OS keeps the translation in the page table, which is implemented as a radix tree. There are HW features that improve the performance of address translation: mainly Translation Lookaside Buffer (TLB) and HW page walkers. Also, developers can utilize Huge Pages to mitigate the cost of address translation in some cases (discussed later in the book).
- We looked at the design of a recent Intel's GoldenCove microarchitecture. Logically, the core is split into Front-End and Back-End. Front-End consists of Branch Predictor Unit (BPU), L1-I cache, instruction fetch and decode logic, and IDQ, which feeds instructions to the CPU Back-End. The Back-End consists of OOO engine, execution units, load-store unit, L1-D cache, and a TLB hierarchy.
- Modern processors have some performance monitoring features which are encapsulated into Performance Monitoring Unit (PMU). The central place in this unit is a concept of Performance Monitoring Counters (PMC) that allow to observe specific events that happen while the program is running, for example, cache misses and branch mispredictions.

4 Terminology and Metrics in Performance Analysis

Like many engineering disciplines, Performance Analysis is quite heavy on using peculiar terms and metrics. For a beginner, it can be a very hard time looking into a profile generated by an analysis tool like Linux `perf` and Intel VTune Profiler. Those tools juggle with many complex terms and metrics, however, it is a "must-know" if you're set to do any serious performance engineering work.

Since we mentioned Linux `perf`, let us briefly introduce the tool as we have many examples of using it in this and later chapters. Linux `perf` is a performance profiler that you can use to find hotspots in a program, collect various low-level CPU performance events, analyze call stacks, and many other things. We will use Linux `perf` extensively throughout the book as it is one of the most popular performance analysis tools. Another reason why we prefer showcasing Linux `perf` is because it is open-sourced, which allows enthusiastic readers to explore the mechanics of what's going on inside a modern profiling tool. This is especially useful for learning concepts presented in this book because GUI-based tools, like Intel® VTune™ Profiler, tend to hide all the complexity. We will have a more detailed overview of Linux `perf` in chapter 7.

This chapter is a gentle introduction to the basic terminology and metrics used in performance analysis. We will first define the basic things like retired/executed instructions, IPC/CPI, UOPs, core/reference clocks, cache misses and branch mispredictions. Then we will see how to measure the memory latency and bandwidth of a system and introduce some more advanced metrics. In the end, we will benchmark four industry workloads and look at the collected metrics.

4.1 Retired vs. Executed Instruction

Modern processors typically execute more instructions than the program flow requires. This happens because some of them are executed speculatively, as discussed in Section 3.3.3. For usual instructions, the CPU commits results once they are available, and all preceding instructions are already retired. But for instructions executed speculatively, the CPU keeps their results without immediately committing their results. When the speculation turns out to be correct, the CPU unblocks such instructions and proceeds as normal. But when it comes out that the speculation happens to be wrong, the CPU throws away all the changes done by speculative instructions and does not retire them. So, an instruction processed by the CPU can be executed but not necessarily retired. Taking this into account, we can usually expect the number of executed instructions to be higher than the number of retired instructions.

There is an exception. Certain instructions are recognized as idioms and are resolved without actual execution. An example of it can be NOP, move elimination and zeroing, see Section 3.8.2. Such instructions do not require an execution unit but are still retired. So, theoretically, there could be a case when the number of retired instructions is higher than the number of executed instructions.

There is a fixed performance counter (PMC) in most modern processors that collects the number of retired instructions. There is no performance event to collect executed instructions, though there is a way to collect executed and retired *uops* as we shall see soon. The number of retired instructions can be easily obtained with Linux `perf` by running:

```
$ perf stat -e instructions ./a.exe
  2173414  instructions  #    0.80  insn per cycle
# or just simply do:
$ perf stat ./a.exe
```

4.2 CPU Utilization

CPU utilization is the percentage of time the CPU was busy during a time period. Technically, a CPU is considered utilized when it is not running the kernel `idle` thread.

$$CPU\ Utilization = \frac{CPU\_CLK\_UNHALTED.REF\_TSC}{TSC},$$

where `CPU_CLK_UNHALTED.REF_TSC` counts the number of reference cycles when the core is not in a halt state, `TSC` stands for timestamp counter (discussed in Section 2.5), which is always ticking.

If CPU utilization is low, it usually translates into a poor performance of an application since a portion of time was wasted by a CPU. However, high CPU utilization is not always an indication of good performance. It is merely a sign that the system is doing some work but does not exactly say what it is doing: the CPU might be highly utilized even though it is stalled waiting on memory accesses. In a multithreaded context, a thread can also spin while waiting for resources to proceed. Later in Section 13.2, we will discuss parallel efficiency metrics, and in particular take a look at "Effective CPU utilization" which filters spinning time.

Linux `perf` automatically calculates CPU utilization across all CPUs on the system:

```
$ perf stat -- a.exe
  0.634874  task-clock (msec) #    0.773 CPUs utilized
```

4.3 CPI and IPC

Those are two fundamental metrics that stand for:

- Cycles Per Instruction (CPI) - how many cycles it took to retire one instruction on average.

$$IPC = \frac{INST\_RETIRED.ANY}{CPU\_CLK\_UNHALTED.THREAD},$$

  where `INST_RETIRED.ANY` counts the number of retired instructions, `CPU_CLK_UNHALTED.THREAD` counts the number of core cycles while the thread is not in a halt state.

- Instructions Per Cycle (IPC) - how many instructions were retired per one cycle on average.

$$CPI = \frac{1}{IPC}$$

Using one or another is a matter of preference. The main author of the book prefers to use `IPC` as it easier to compare. With IPC, we want as many instructions per cycle as possible, so the higher IPC, the better. With `CPI`, it's the opposite: we want as fewer cycles per instruction as possible, so the lower CPI the better. The comparison that uses "the higher the better" metric is simpler since you don't have to do the mental inversion every time. In the rest of the book we will mostly use IPC, but again, there is nothing wrong with using CPI either.

Relation between IPC and CPU clock frequency is very interesting. In the broad sense, `performance = work / time`, where we can express work as the number of instructions and time as seconds. The number of seconds a program was running is `total cycles / frequency`:

$$Performance = \frac{instructions \times frequency}{cycles} = IPC \times frequency$$

As we can see, performance is proportional to IPC and frequency. If we increase any of the two metrics, performance of the program is set to grow.

From the perspective of benchmarking, IPC and frequency are two independent metrics. We've seen many engineers mistakenly mixing them up and thinking that if you increase the frequency, the IPC will also go up. But it's not, the IPC will stay the same. If you clock a processor at 1 GHz instead of 5Ghz, you will still have the same IPC. It is very confusing, especially since IPC has all to do with CPU clocks. Frequency only tells how fast a single clock is, whereas IPC doesn't account for the speed at which clocks change, it counts how much work is done every cycle. So, from the benchmarking perspective, IPC solely depends on the design of the processor regardless of the frequency. Out-of-order cores typically much higher IPC than in-order cores. When you increase the size of CPU caches or improve branch prediction, the IPC usually goes up.

Now, if you ask a HW architect, they will certainly tell you that there is dependency between IPC and frequency. From the CPU design perspective, you can deliberatly downclock the processor, which will make every cycle longer and allow to squeeze more work into each one of those. In the end, you will get higher IPC but lower frequency. HW vendors approach this performance equation in different ways. For example, Intel and AMD chips usually have very high frequency, with recent 13900KS processor crossed the mark of 6Ghz turbo frequency out of the box with no overclocking required. Apple M1/M2 chips on the other hand have lower frequency but compensate it with a higher

IPC. Lower frequency allows for lower power consumption. Higher IPC on the other hand usually requires more complicated design, more transistors and larger die size. We will not go into all the design tradeoffs here as it is a topic for a different book. We will talk about future advancements in IPC and frequency in the last chapter.

IPC is useful for evaluating both HW and SW efficiency. HW engineers use this metric to compare CPU generations and CPUs from different vendors. Since IPC is the measure of how good is the performance of the microarchitecture, engineers and media uses it to express the gain in performance of the newest CPU over the previous generation. Although to make a fair comparison, you need to run both systems on the same frequency.

IPC is also a useful metric for evaluating software. It gives you an intuition of how fast instructions in your application progress through the CPU pipeline. Later in this chapter you will see several production applications with varying IPC. Memory intensive applications are usually characterized with a low IPC (0-1), while computationally intensive workloads tend have high IPC (4-6).

Linux `perf` users can measure the IPC for their workload by running:

```
$ perf stat -e cycles,instructions -- a.exe
  2369632  cycles
  1725916  instructions  #    0,73  insn per cycle
# or as simple as:
$ perf stat ./a.exe
```

4.4 UOPs (micro-ops)

Microprocessors with the x86 architecture translate complex CISC-like instructions into simple RISC-like microoperations, abbreviated as µops or uops. We will use the "uop" notation as it is easier to write. A simple addition instruction such as `ADD rax, rbx` generates only one µop, while more complex instruction like `ADD rax, [mem]` may generate two: one for reading from `mem` memory location into a temporary (un-named) register, and one for adding it to the `rax` register. The instruction `ADD [mem], rax` generates three uops: one for reading from memory, one for adding, and one for writing the result back to memory.

The main advantage of splitting instructions into micro operations is that uops can be executed:

- **Out of order**: consider `PUSH rbx` instruction, that decrements the stack pointer by 8 bytes and then stores the source operand on the top of the stack. Suppose that `PUSH rbx` is "cracked" into two dependent micro operations after decode:

  ```
  SUB rsp, 8
  STORE [rsp], rbx
  ```

  Often, function prologue saves multiple registers using `PUSH` instructions. In our case, the next `PUSH` instruction can start executing after the `SUB` uop of the previous `PUSH` instruction finishes, and doesn't have to wait for the `STORE` uop, which can now go asynchronously.

- **In parallel**: consider `HADDPD xmm1, xmm2` instruction, that will sum up (reduce) two double precision floating point values in both `xmm1` and `xmm2` and store two results in `xmm1` as follows:

  ```
  xmm1[63:0] = xmm2[127:64] + xmm2[63:0]
  xmm1[127:64] = xmm1[127:64] + xmm1[63:0]
  ```

  One way to microcode this instruction would be to do the following: 1) reduce `xmm2` and store the result in `xmm_tmp1[63:0]`, 2) reduce `xmm1` and store the result in `xmm_tmp2[63:0]`, 3) merge `xmm_tmp1` and `xmm_tmp2` into `xmm1`. Three uops in total. Notice that steps 1) and 2) are independent and thus can be done in parallel.

Even though we were just talking about how instructions are split into smaller pieces, sometimes, uops can also be fused together. There are two types of fusion in modern CPUs:

- **Microfusion**: fuse uops from the same machine instruction. Microfusion can only be applied to two types of combinations: memory write operations and read-modify operations. For example:

  ```
  add    eax, [mem]
  ```

There are two uops in this instruction: 1) read the memory location `mem`, and 2) add it to `eax`. With microfusion, two uops are fused into one at the decoding step.

- **Macrofusion**: fuse uops from different machine instructions. The decoders can fuse arithmetic or logic instruction with a subsequent conditional jump instruction into a single compute-and-branch µop in certain cases. For example:

```
.loop:
  dec rdi
  jnz .loop
```

With macrofusion, wwo uops from `DEC` and `JNZ` instructions are fused into one.

Both Micro- and Macrofusion save bandwidth in all stages of the pipeline from decoding to retirement. The fused operations share a single entry in the reorder buffer (ROB). The capacity of the ROB is utilized better when a fused uop uses only one entry. Such fused ROB entry is later dispatched to two different execution ports but is retired again as a single unit. Readers can learn more about uop fusion in [Fog, 2012].

To collect the number of issued, executed, and retired uops for an application, you can use Linux `perf` as follows:

```
$ perf stat -e uops_issued.any,uops_executed.thread,uops_retired.slots -- ./a.exe
  2856278  uops_issued.any
  2720241  uops_executed.thread
  2557884  uops_retired.slots
```

The way instructions are split into micro operations may vary across CPU generations. Usually, the lower number of uops used for an instruction means that HW has a better support for it and is likely to have lower latency and higher throughput. For the latest Intel and AMD CPUs, the vast majority of instructions generate exactly one uop. Latency, throughput, port usage, and the number of uops for x86 instructions on recent microarchitectures can be found at the uops.info[51] website.

4.5 Pipeline Slot

Another important metric which some performance tools use is the concept of a *pipeline slot*. A pipeline slot represents hardware resources needed to process one uop. Figure 24 demonstrates the execution pipeline of a CPU that has 4 allocation slots every cycle. That means that the core can assign execution resources (renamed source and destination registers, execution port, ROB entries, etc.) to 4 new uops every cycle. Such a processor is usually called a *4-wide machine*. During six consecutive cycles on the diagram, only half of the available slots were utilized. From a microarchitecture perspective, the efficiency of executing such code is only 50%.

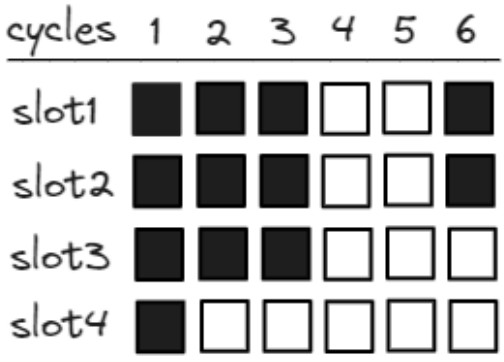


Figure 24: Pipeline diagram of a 4-wide CPU.

Intel's Skylake and AMD Zen3 cores have 4-wide allocation. Intel's SunnyCove microarchitecure was a 5-wide design. As of 2023, most recent Goldencove and Zen4 architectures both have 6-wide allocation. Apple M1 design is not officially disclosed but is measured to be 8-wide.[52]

[51] Instruction latency and Throughput - https://uops.info/table.html
[52] Apple Microarchitecture Research - https://dougallj.github.io/applecpu/firestorm.html

Pipeline slot is one of the core metrics in Top-down Microarchitecture Analysis (see Section 6.1). For example, Front-End Bound and Back-End Bound metrics are expressed as a percentage of unutilized Pipeline Slots due to various reasons.

4.6 Core vs. Reference Cycles

Most CPUs employ a clock signal to pace their sequential operations. The clock signal is produced by an external generator that provides a consistent number of pulses each second. The frequency of the clock pulses determines the rate at which a CPU executes instructions. Consequently, the faster the clock, the more instructions the CPU will execute each second.

$$Frequency = \frac{Clockticks}{Time}$$

The majority of modern CPUs, including Intel and AMD CPUs, don't have a fixed frequency at which they operate. Instead, they implement dynamic frequency scaling, which is called *Turbo Boost* in Intel's CPUs, and *Turbo Core* in AMD processors. It enables the CPU to increase and decrease its frequency dynamically. Decreasing the frequency reduces power consumption at the expense of performance, and increasing the frequency improves performance but sacrifices power savings.

The core clock cycles counter is counting clock cycles at the actual frequency that the CPU core is running at, rather than the external clock (reference cycles). Let's take a look at an experiment on Skylake i7-6000 processor running a single-threaded application, which has a base frequency of 3.4 GHz:

```
$ perf stat -e cycles,ref-cycles ./a.exe
  43340884632  cycles    # 3.97 GHz
  37028245322  ref-cycles # 3.39 GHz
      10,899462364 seconds time elapsed
```

Metric `ref-cycles` counts cycles as if there were no frequency scaling. The external clock on the setup has a frequency of 100 MHz, and if we scale it by the *clock multiplier*, we will get the base frequency of the processor. The clock multiplier for Skylake i7-6000 processor equals 34: it means that for every external pulse, the CPU executes 34 internal cycles when it's running on the base frequency.

Metric `cycles` counts real CPU cycles, i.e., taking into account frequency scaling. Using the formula above we can confirm that the average operating frequency was `43340884632 cycles / 10.899 sec = 3.97 Ghz`. When you compare performance of two versions of a small piece of code, measuring the time in clock cycles is better than in nanoseconds, because you avoid the problem of the clock frequency going up and down.

4.7 Cache Miss

As discussed in Section 3.6, any memory request missing in a particular level of cache must be serviced by higher-level caches or DRAM. This implies a significant increase in the latency of such memory access. The typical latency of memory subsystem components is shown in Table 3. There is also an interactive view[53] that visualizes the latency of different operations in modern systems. Performance greatly suffers, especially when a memory request misses in Last Level Cache (LLC) and goes all the way down to the main memory. Intel® Memory Latency Checker[54] (MLC) is a tool used to measure memory latencies and bandwidth and how they change with increasing load on the system. MLC is useful for establishing a baseline for the system under test and for performance analysis. We will use this tool when will talk about memory latency and bandwidth several pages later.

Table 3: Typical latency of a memory subsystem in x86-based platforms.

Memory Hierarchy Component	Latency (cycle/time)
L1 Cache	4 cycles (~1 ns)
L2 Cache	10-25 cycles (5-10 ns)
L3 Cache	~40 cycles (20 ns)
Main Memory	200+ cycles (100 ns)

[53] Interactive latency - https://colin-scott.github.io/personal_website/research/interactive_latency.html
[54] Memory Latency Checker - https://www.intel.com/software/mlc

A cache miss might happen both for instructions and data. According to Top-down Microarchitecture Analysis (see Section 6.1), an instruction cache (I-cache) miss is characterized as a Front-End stall, while a data cache (D-cache) miss is characterized as a Back-End stall. Instruction cache miss happens very early in the CPU pipeline during instruction fetch. Data cache miss happens much later during the instruction execution phase.

Linux `perf` users can collect the number of L1-cache misses by running:

```
$ perf stat -e mem_load_retired.fb_hit,mem_load_retired.l1_miss,
  mem_load_retired.l1_hit,mem_inst_retired.all_loads -- a.exe
   29580  mem_load_retired.fb_hit
   19036  mem_load_retired.l1_miss
  497204  mem_load_retired.l1_hit
  546230  mem_inst_retired.all_loads
```

Above is the breakdown of all loads for the L1 data cache and fill buffers. A load might either hit the already allocated fill buffer (`fb_hit`), or hit the L1 cache (`l1_hit`), or miss both (`l1_miss`), thus `all_loads = fb_hit + l1_hit + l1_miss`. We can see that only 3.5% of all loads miss in the L1 cache, thus the *L1 hit rate* is 96.5%.

We can further break down L1 data misses and analyze L2 cache behavior by running:

```
$ perf stat -e mem_load_retired.l1_miss,
  mem_load_retired.l2_hit,mem_load_retired.l2_miss -- a.exe
  19521  mem_load_retired.l1_miss
  12360  mem_load_retired.l2_hit
   7188  mem_load_retired.l2_miss
```

From this example, we can see that 37% of loads that missed in the L1 D-cache also missed in the L2 cache, thus the *L2 hit rate* is 63%. In a similar way, a breakdown for the L3 cache can be made.

4.8 Mispredicted Branch

Modern CPUs try to predict the outcome of a branch instruction (taken or not taken). For example, when the processor sees code like this:

```
dec eax
jz .zero
# eax is not 0
...
zero:
# eax is 0
```

In the above example, the `jz` instruction is a branch. Modern CPU architectures try to predict the outcome of every branch to increase performance. This is called "Speculative Execution" that we discussed in Section 3.3.3. The processor will speculate that, for example, the branch will not be taken and will execute the code that corresponds to the situation when `eax is not 0`. However, if the guess is wrong, this is called "branch misprediction", and the CPU is required to undo all the speculative work that it has done recently. This typically involves a penalty between 10 and 20 clock cycles.

Linux `perf` users can check the number of branch mispredictions by running:

```
$ perf stat -e branches,branch-misses -- a.exe
   358209  branches
    14026  branch-misses #    3,92% of all branches
# or simply do:
$ perf stat -- a.exe
```

4.9 Performance Metrics

In addition to the performance events that we discussed earlier in this chapter, performance engineers frequently use metrics, which are built on top of raw events. Table 4 shows a list of metrics for Intel's 12th-gen Goldencove architecture along with descriptions and formulas. The list is not exhaustive, but it shows the most important metrics. Complete list of metrics for Intel CPUs and their formulas can be found in TMA_metrics.xlsx[55]. The last section in this chapter shows how performance metrics can be used in practice.

Table 4: A list (not exhaustive) of secondary metrics along with descriptions and formulas for Intel Goldencove architecture.

Metric Name	Description	Formula
L1MPKI	L1 cache true misses per kilo instruction for retired demand loads.	1000 * MEM_LOAD_RETIRED.L1_MISS_PS / INST_RETIRED.ANY
L2MPKI	L2 cache true misses per kilo instruction for retired demand loads.	1000 * MEM_LOAD_RETIRED.L2_MISS_PS / INST_RETIRED.ANY
L3MPKI	L3 cache true misses per kilo instruction for retired demand loads.	1000 * MEM_LOAD_RETIRED.L3_MISS_PS / INST_RETIRED.ANY
Branch Mispr. Ratio	Ratio of all branches which mispredict	BR_MISP_RETIRED.ALL_BRANCHES / BR_INST_RETIRED.ALL_BRANCHES
Code STLB MPKI	STLB (2nd level TLB) code speculative misses per kilo instruction (misses of any page-size that complete the page walk)	1000 * ITLB_MISSES.WALK_COMPLETED / INST_RETIRED.ANY
Load STLB MPKI	STLB data load speculative misses per kilo instruction	1000 * DTLB_LOAD_MISSES.WALK_COMPLETED / INST_RETIRED.ANY
Store STLB MPKI	STLB data store speculative misses per kilo instruction	1000 * DTLB_STORE_MISSES.WALK_COMPLETED / INST_RETIRED.ANY
Load Miss Real Latency	Actual Average Latency for L1 data-cache miss demand load operations (in core cycles)	L1D_PEND_MISS.PENDING / MEM_LOAD_COMPLETED.L1_MISS_ANY
ILP	Instr.-Level-Parallelism per-core (average number of uops executed when there is execution)	UOPS_EXECUTED.THREAD / UOPS_EXECUTED.CORE_CYCLES_GE_1, divide by 2 if SMT is enabled
MLP	Memory-Level-Parallelism per-thread (average number of L1 miss demand load when there is at least one such miss.)	L1D_PEND_MISS.PENDING / L1D_PEND_MISS.PENDING_CYCLES
DRAM BW Use GB/sec	Average external Memory Bandwidth Use for reads and writes	(64 * (UNC_M_CAS_COUNT.RD + UNC_M_CAS_COUNT.WR) / 1GB) / Time
IpCall	Instructions per near call (lower number means higher occurrence rate)	INST_RETIRED.ANY / BR_INST_RETIRED.NEAR_CALL
Ip Branch	Instructions per Branch	INST_RETIRED.ANY / BR_INST_RETIRED.ALL_BRANCHES
IpLoad	Instructions per Load	INST_RETIRED.ANY / MEM_INST_RETIRED.ALL_LOADS_PS
IpStore	Instructions per Store	INST_RETIRED.ANY / MEM_INST_RETIRED.ALL_STORES_PS
IpMispredict	Number of Instructions per non-speculative Branch Misprediction	INST_RETIRED.ANY / BR_MISP_RETIRED.ALL_BRANCHES

[55] TMA metrics - https://github.com/intel/perfmon/blob/main/TMA_Metrics.xlsx.

Metric Name	Description	Formula
IpFLOP	Instructions per FP (Floating Point) operation	See TMA_metrics.xlsx
IpArith	Instructions per FP Arithmetic instruction	See TMA_metrics.xlsx
IpArith Scalar SP	Instructions per FP Arith. Scalar Single-Precision instruction	INST_RETIRED.ANY / FP_ARITH_INST_RETIRED.SCALAR_SINGLE
IpArith Scalar DP	Instructions per FP Arith. Scalar Double-Precision instruction	INST_RETIRED.ANY / FP_ARITH_INST_RETIRED.SCALAR_DOUBLE
Ip Arith AVX128	Instructions per FP Arithmetic AVX/SSE 128-bit instruction	INST_RETIRED.ANY / (FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE+ FP_ARITH_INST_RETIRED.128B_PACKED_SINGLE)
Ip Arith AVX256	Instructions per FP Arithmetic AVX* 256-bit instruction	INST_RETIRED.ANY / (FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE+ FP_ARITH_INST_RETIRED.256B_PACKED_SINGLE)
Ip SWPF	Instructions per SW prefetch instruction (of any type)	INST_RETIRED.ANY / SW_PREFETCH_ACCESS.T0:u0xF

A few notes on those metrics. First, ILP and MLP metrics do not represent theoretical maximums for an application, rather they measure ILP and MLP on a given machine. On an ideal machine with infinite resources numbers will be higher. Second, all metrics besides "DRAM BW Use" and "Load Miss Real Latency" are fractions; we can apply fairly straightforward reasoning to each of them to tell whether a specific metric is high or low. But to make sense of "DRAM BW Use" and "Load Miss Real Latency" metrics, we need to put it in a context. For the former, we would like to know if a program saturates the memory bandwidth or not. The latter gives you an idea for the average cost of a cache miss, which is useless by itself unless you know the latencies of the cache hierarchy. We will discuss how to find out cache latencies and peak memory bandwidth in the next section.

Formulas in the table give an intuition on how performance metrics are calculated, so that you can build similar metrics on another platform as long as underlying performance events are available there. Some tools can report performance metrics automatically. If not, you can always calculate those metrics manually since you know the formulas and corresponding performance events that must be collected.

4.10 Memory Latency and Bandwidth

Inefficient memory accesses are often a dominant performance bottleneck in modern environments. Thus, how quickly a processor can fetch data from the memory subsystem is a critical factor in determining application performance. There are two aspects of memory performance: 1) how fast a CPU can fetch a single byte from memory (latency), and 2) how many bytes it can fetch per second (bandwidth). Both are important in various scenarios, we will look at a few examples later. In this section, we will focus on measuring peak performance of the memory subsystem components.

One of the tools that can become helpful on x86 platforms is Intel Memory Latency Checker (MLC),[56] which is available for free on Windows and Linux. MLC can measure cache and memory latency and bandwidth using different access patterns and under load. On ARM-based systems there is no similar tool, however, users can download and build memory latency and bandwidth benchmarks from sources. Example of such projects are lmbench[57], bandwidth[58] and Stream[59].

We will only focus on a subset of metrics, namely idle read latency and read bandwidth. Let's start with the read latency. Idle means that while we do the measurements, the system is idle. This will give us the minimum time required to fetch data from memory system components, but when the system is loaded by other "memory-hungry" applications, this latency increases as there may be more queueing for resources at various points. MLC measures idle latency by doing dependent loads (aka pointer chasing). A measuring thread allocates a buffer and initializes it

[56] Intel MLC tool - https://www.intel.com/content/www/us/en/download/736633/intel-memory-latency-checker-intel-mlc.html
[57] lmbench - https://sourceforge.net/projects/lmbench
[58] Memory bandwidth benchmark by Zack Smith - https://zsmith.co/bandwidth.php
[59] Stream - https://github.com/jeffhammond/STREAM

such that each cache line (64-byte) is pointing to another line. By appropriately sizing the buffer, we can ensure that almost all the loads are hitting in certain level of cache or memory.

Our system under test is an Intel Alderlake box with Core i7-1260P CPU and 16GB DDR4 @ 2400 MT/s single channel memory. The processor has 4P (Performance) hyperthreaded and 8E (Efficient) cores. Every P-core has 48KB of L1d cache and 1.25MB of L2 cache. Every E-core has 32KB of L1d cache, four E-cores form a cluster that has access to a shared 2MB L2 cache. All cores in the system are backed by a 18MB L3-cache. If we use a 10MB buffer, we can be certain that repeated accesses to that buffer would miss in L2 but hit in L3. Here is the example `mlc` command:

```
$ ./mlc --idle_latency -c0 -L -b10m
Intel(R) Memory Latency Checker - v3.10
Command line parameters: --idle_latency -c0 -L -b10m

Using buffer size of 10.000MiB
*** Unable to modify prefetchers (try executing 'modprobe msr')
*** So, enabling random access for latency measurements
Each iteration took 31.1 base frequency clocks (  12.5  ns)
```

The option `--idle_latency` measures read latency without loading the system. MLC has the `--loaded_latency` option to measure latency when there is memory traffic generated by other threads. The option `-c0` pins the measurement thread to logical CPU 0, which is on a P-core. The option `-L` enables large pages to limit TLB effects in our measurements. The option `-b10m` tells MLC to use a 10MB buffer, which will fit in L3 cache on our system.

The chart on 25 shows read latencies of L1, L2, and L3 caches. There are four different regions on the chart. The first region on the left from 1KB to 48KB buffer size corresponds to L1d cache, which is private to each physical core. We can observe 0.9ns latency for E-core and a slightly higher 1.1ns for P-core. Also, we can use this chart to confirm the cache sizes. Notice how E-core latency starts climbing after a buffer size goes above 32KB but E-core latency stays constant up to 48KB. That confirms that L1d cache size in E-core is 32KB, and in P-core it is 48KB.

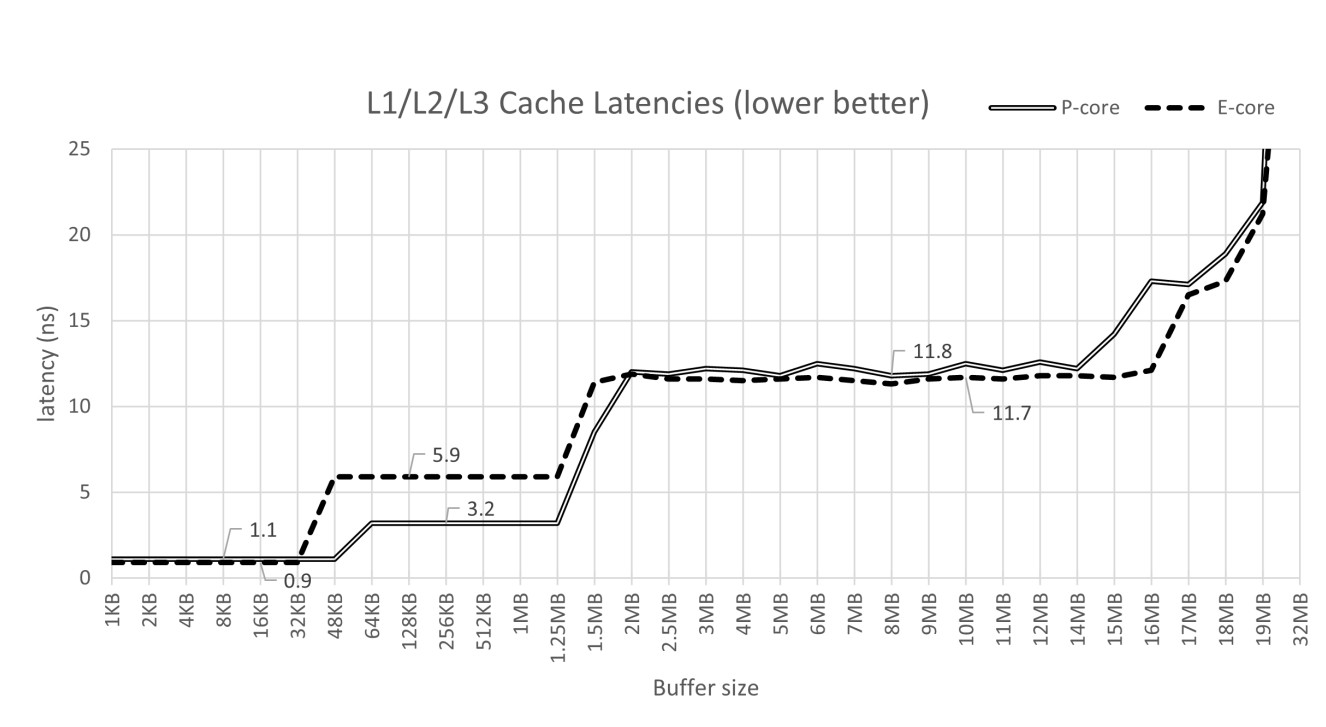


Figure 25: L1/L2/L3 cache read latencies on Intel Core i7-1260P, measured with the mlc tool, large pages enabled.

The second region shows the L2 cache latencies, which for E-core is almost twice higher than for P-core (5.9ns vs. 3.2ns). For P-core the latency increases after we cross 1.25MB buffer size, which is expected. But we expect E-core latency to stay the same until 2MB, which is not happening in our measurements.

The third region from 2MB up to 14MB corresponds to L3 cache latency, which is roughly 12ns for both types of cores. The total size of L3 cache that is shared between all cores in the system is 18MB. Interestingly, we start seeing some unexpected dynamics starting from 15MB, not 18MB. Most likely it has to do with some accesses miss in L3 and require going to the main memory.

The fourth region corresponds to memory latency, only the beginning of it is shown on the chart. After we cross the 18MB boundary, the latency climbes very steeply and starts to level off at 24MB for E-core and 64MB for P-core. With a much larger buffer size of 500MB, E-core access latency is 45ns and P-core is 90ns. This measures the memory latency since almost no loads hit in the L3 cache.

Using a similar technique we can measure bandwidth of various components of the memory hierarchy. For measuring bandwidth, MLC executes load requests which results are not used by any subsequent instructions. This allows MLC to generate maximum possible bandwidth. MLC spawns one software thread on each of the configured logical processors. The addresses that each thread accesses are independent and there is no sharing of data between threads. As with the latency experiments, the buffer size used by the threads determine whether MLC is measuring L1/L2/L3 cache b/w or memory b/w.

```
./mlc --max_bandwidth -k0-15 -Y -L -b10m
Measuring Maximum Memory Bandwidths for the system
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using all the threads from each core if Hyper-threading is enabled
Using traffic with the following read-write ratios
ALL Reads        :      33691.53
3:1 Reads-Writes :      30352.45
2:1 Reads-Writes :      29722.28
1:1 Reads-Writes :      28382.45
Stream-triad like:      30503.68
```

The new options here are `-k`, which specifies a list of CPU numbers used for measurements. The `-Y` option tells MLC to use AVX2 loads, i.e. 32 bytes at a time. MLC measures bandwidth with different read-write ratios, but in the diagram below we only show all-read bandwidth as it gives us an intuition about peak memory bandwidth. But other ratios can also be important. Combined latency and bandwidth numbers for our system under test as measured with Intel MLC are shown in 26.

Cores can draw much higher bandwidth from lower level caches like L1 and L2 than from shared L3 cache or main memory. Shared caches such as L3 and E-core L2, scale reasonably well to serve requests from multiple cores at the same time. For example, single E-core L2 bandwidth is 100GB/s. With two E-cores from the same cluster, I measured 140 GB/s, three E-cores - 165 GB/s, and all four E-cores can draw 175 GB/s from the shared L2. The same goes for L3 cache, which allows for 60 GB/s for a single P-core and only 25 GB/s for a single E-core. But when all the cores are used, L3 cache can sustain bandwidth of 300 GB/s.

Notice, that we measure latency in nanoseconds and bandwidth in GB/s, thus they also depend on the frequency at which cores are running. In various circumstances, the observed numbers may be different. For example, let's assume that when running solely on the system at full turbo frequency, P-core has L1 latency `X` and L1 bandwidth `Y`. When the system is fully loaded, we may observe these metrics drop to `1.25X` and `0.75Y` respectively. To mitigate the frequency effects, instead of nanoseconds, latencies and metrics can be represented using core cycles, normalized to some sample frequency, say 3Ghz.

Knowledge of the primary characteristics of a machine is fundamental to assessing how well a program utilizes available resources. We will continue this discussion in Section 5.5 about Roofline performance model. If you constantly analyze performance on a single platform, it is a good idea to memorize latencies and bandwidth of various components of the memory hierarchy or have them handy. It helps to establish the mental model for a system under test which will aid your further performance analysis as you will see next.

4.11 Case Study: Analyzing Performance Metrics of Four Benchmarks

Putting together everything we discussed so far in this chapter, we run four benchmarks from different domains and calculated their performance metrics. First of all, let's introduce the benchmarks.

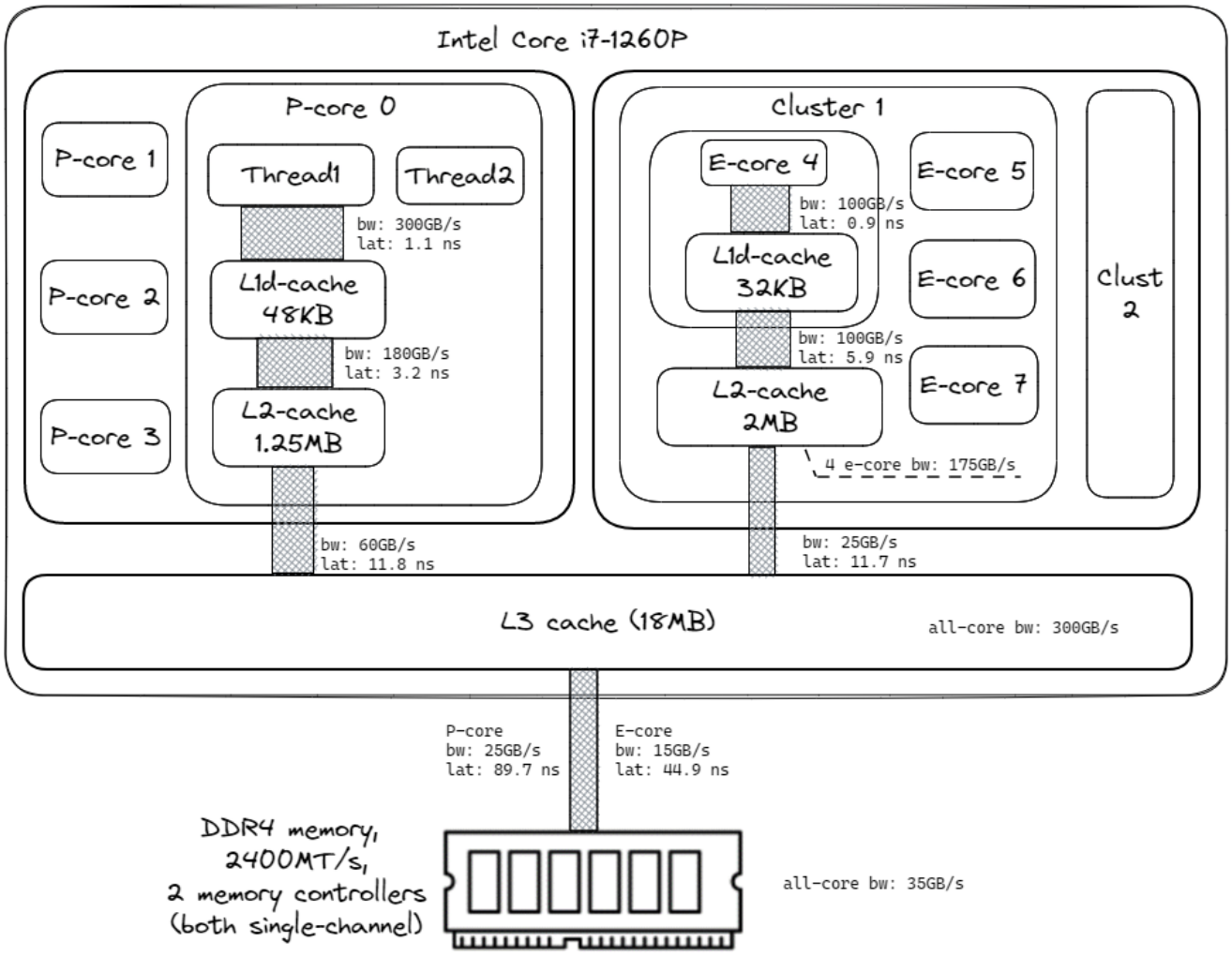


Figure 26: Block diagram of the memory hierarchy of Intel Core i7-1260P and external DDR4 memory.

1. Blender 3.4 - an open-source 3D creation and modeling software project. This test is of Blender's Cycles performance with BMW27 blend file. All HW threads are used. URL: https://download.blender.org/release. Command line: `./blender -b bmw27_cpu.blend -noaudio --enable-autoexec -o output.test -x 1 -F JPEG -f 1`.
2. Stockfish 15 - an advanced open-source chess engine. This test is a stockfish built-in benchmark. A single HW thread is used. URL: https://stockfishchess.org. Command line: `./stockfish bench 128 1 24 default depth`.
3. Clang 15 selfbuild - this test uses clang 15 to build clang 15 compiler from sources. All HW threads are used. URL: https://www.llvm.org. Command line: `ninja -j16 clang`.
4. CloverLeaf 2018 - a Lagrangian-Eulerian hydrodynamics benchmark. All HW threads are used. This test uses clover_bm.in input file (Problem 5). URL: http://uk-mac.github.io/CloverLeaf. Command line: `./clover_leaf`.

For the purpose of this exercise, we run all four benchmarks on the machine with the following characteristics:

- 12th Gen Alderlake Intel(R) Core(TM) i7-1260P CPU @ 2.10GHz (4.70GHz Turbo), 4P+8E cores, 18MB L3-cache
- 16 GB RAM, DDR4 @ 2400 MT/s
- 256GB NVMe PCIe M.2 SSD
- 64-bit Ubuntu 22.04.1 LTS (Jammy Jellyfish)

To collect performance metrics, we use `toplev.py` script that is a part of pmu-tools[60] written by Andi Kleen:

```
$ ~/workspace/pmu-tools/toplev.py -m --global --no-desc -v -- <app with args>
```

Table 5 provides a side-by-side comparison of performance metrics for our four benchmarks. There is a lot we can learn about the nature of those workloads just by looking at the metrics. Here are the hypothesis we can make about the benchmarks before collecting performance profiles and diving deeper into the code of those applications.

- **Blender**. The work is split fairly equally between P-cores and E-cores, with a decent IPC on both core types. The number of cache misses per kilo instructions is pretty low (see `L*MPKI`). Branch misprediction contributes as a bottleneck: the `Br. Misp. Ratio` metric is at `2%`; we get 1 misprediction every `610` instructions (see `IpMispredict` metric), which is not bad, but is not perfect either. TLB is not a bottleneck as we very rarely miss in STLB. We ignore `Load Miss Latency` metric since the number of cache misses is very low. The ILP is reasonably high. Goldencove is a 6-wide architecture; ILP of `3.67` means that the algorithm utilizes almost `2/3` of the core resources every cycle. Memory bandwidth demand is low, it's only 1.58 GB/s, far from the theoretical maximum for this machine. Looking at the `Ip*` metrics we can tell that Blender is a floating point algorithm (see `IpFLOP` metric), large portion of which is vectorized FP operations (see `IpArith AVX128`). But also, some portions of the algorithm are non-vectorized scalar FP single precision instructions (`IpArith Scal SP`). Also, notice that every 90th instruction is an explicit software memory prefetch (`IpSWPF`); we expect to see those hints in the Blender's source code. Conclusion: Blender's performance is bound by FP compute with occasional branch mispredictions.
- **Stockfish**. We ran it using only one HW thread, so there is zero work on E-cores, as expected. The number of L1 misses is relatively high, but then most of them are contained in L2 and L3 caches. Branch misprediction ratio is high; we pay the misprediction penalty every `215` instructions. We can estimate that we get one mispredict every `215 (instructions) / 1.80 (IPC) = 120` cycles, which is very frequently. Similar to the Blender reasoning, we can say that TLB and DRAM bandwidth is not an issue for Stockfish. Going further, we see that there is almost no FP operations in the workload. Conclusion: Stockfish is an integer compute workload, which is heavily affected by branch mispredictions.
- **Clang 15 selfbuild**. Compilation of C++ code is one of the tasks which has a very flat performance profile, i.e. there are no big hotspots. Usually you will see that the running time is attributed to many different functions. First thing we spot is that P-cores are doing 68% more work than E-cores and have 42% better IPC. But both P- and E-cores have low IPC. The L*MPKI metrics doesn't look troubling at a first glance, however, in combination with the load miss real latency (`LdMissLat`, in core clocks), we can see that the average cost of a cache miss is quite high (~77 cycles). Now, when we look at the `*STLB_MPKI` metrics, we notice substantial difference with any other benchmark we test. This is another aspect of Clang compiler (and other compilers as well), is that the size of the binary is relatively big: it's more than 100 MB. The code constantly jumps to

[60] pmu-tools - https://github.com/andikleen/pmu-tools.

distant places causing high pressure on the TLB subsystem. As you can see the problem exists both for ITLB (instructions) and DTLB (data). Let's proceed with our analysis. DRAM bandwidth use is higher than for the two previous benchmarks, but still not reaching even half of the maximum memory bandwidth on our platform (which is ~25 GB/s). Another concern for us is the very small number of instruction per call (`IpCall`), only ~41 instruction per function call. This is unfortunately the nature of the compilation codebase: it has thousands of small functions. Compiler has to be very aggressive with inlining all those functions and wrappers. Yet, we suspect that the performance overhead associated with making a function call remains an issue. Also, one can spot the high `ipBranch` and `IpMispredict` metric. For Clang compilation, every fifth instruction is a branch and one of every ~35 branches gets mispredicted. There are almost no FP or vector instructions, but this is not surprising. Conclusion: Clang has a large codebase, flat profile, many small functions, "branchy" code; performance is affected by data cache and TLB misses and branch mispredictions.

- **CloverLeaf**. As before we start with analyzing instructions and core cycles. The amount of work done by P- and E-cores is roughly the same, but it takes P-cores more time to do this work, resulting in a lower IPC of one logical thread on P-core compared to one physical E-core. We don't have a good explanation to that just yet. The `L*MPKI` metrics is high, especially the number of L3 misses per kilo instructions. The load miss latency (`LdMissLat`) is off charts, suggesting an extremely high price of the average cache miss. Next, we take a look at the `DRAM BW use` metric and see that memory bandwidth is fully saturated. That's the problem: all the cores in the system share the same memory bus, they compete for the access to the memory, which effectively stalls the execution. CPUs are undersupplied with the data that they demand. Going further, we can see that CloverLeaf does not suffer from mispredictions or function call overhead. The instruction mix is dominated by FP double-precision scalar operations with some parts of the code being vectorized. Conclusion: multi-threaded CloverLeaf is bound by memory bandwidth.

Table 5: Performance Metrics of Four Benchmarks.

Metric Name	Core Type	Blender	Stockfish	Clang15-selfbuild	CloverLeaf
Instructions	P-core	6.02E+12	6.59E+11	2.40E+13	1.06E+12
Core Cycles	P-core	4.31E+12	3.65E+11	3.78E+13	5.25E+12
IPC	P-core	1.40	1.80	0.64	0.20
CPI	P-core	0.72	0.55	1.57	4.96
Instructions	E-core	4.97E+12	0	1.43E+13	1.11E+12
Core Cycles	E-core	3.73E+12	0	3.19E+13	4.28E+12
IPC	E-core	1.33	0	0.45	0.26
CPI	E-core	0.75	0	2.23	3.85
L1MPKI	P-core	3.88	21.38	6.01	13.44
L2MPKI	P-core	0.15	1.67	1.09	3.58
L3MPKI	P-core	0.04	0.14	0.56	3.43
Br. Misp. Ratio	E-core	0.02	0.08	0.03	0.01
Code stlb MPKI	P-core	0	0.01	0.35	0.01
Ld stlb MPKI	P-core	0.08	0.04	0.51	0.03
St stlb MPKI	P-core	0	0.01	0.06	0.1
LdMissLat (Clk)	P-core	12.92	10.37	76.7	253.89
ILP	P-core	3.67	3.65	2.93	2.53
MLP	P-core	1.61	2.62	1.57	2.78
DRAM BW (GB/s)	All	1.58	1.42	10.67	24.57
IpCall	All	176.8	153.5	40.9	2,729
IpBranch	All	9.8	10.1	5.1	18.8
IpLoad	All	3.2	3.3	3.6	2.7
IpStore	All	7.2	7.7	5.9	22.0
IpMispredict	All	610.4	214.7	177.7	2,416
IpFLOP	All	1.1	1.82E+06	286,348	1.8
IpArith	All	4.5	7.96E+06	268,637	2.1
IpArith Scal SP	All	22.9	4.07E+09	280,583	2.60E+09
IpArith Scal DP	All	438.2	1.22E+07	4.65E+06	2.2
IpArith AVX128	All	6.9	0.0	1.09E+10	1.62E+09

Metric Name	Core Type	Blender	Stockfish	Clang15-selfbuild	CloverLeaf
IpArith AVX256	All	30.3	0.0	0.0	39.6
IpSWPF	All	90.2	2,565	105,933	172,348

As you can see from this study, there is a lot one can learn about behavior of a program just by looking at the metrics. It answers the "what?" question, but doesn't tell you the "why?". For that you will need to collect performance profile, which we will introduce in later chapters. In the second part of the book we discuss how to mitigate performance issues that we suspect in the four benchmarks that we analyzed.

Keep in mind that the summary of performance metrics in Table 5 only tells you about the *average* behavior of a program. For example, we might be looking at CloverLeaf's IPC of `0.2`, while in reality it may never run with such an IPC, instead it may have 2 phases of equal duration, one running with IPC of `0.1`, and the second with IPC of `0.3`. Performance tools tackle this by reporting statistical data for each metric along with the average value. Usually, having min, max, 95th percentile, and variation (stdev/avg) is enough to understand the distribution. Also, some tools allow plotting the data, so you can see how the value for a certain metric changed during the program running time. As an example, Figure 27 shows the dynamics of IPC, L*MPKI, DRAM BW and average frequency for the CloverLeaf benchmark. The `pmu-tools` package can automatically build those charts once you add `--xlsx` and `--xchart` options.

```
$ ~/workspace/pmu-tools/toplev.py -m --global --no-desc -v --xlsx workload.xlsx -xchart --
    ./clover_leaf
```

Even though the deviation from the values reported in the summary is not very big, we can see that the workload is not always stable. After looking at the IPC chart we can hypothesize that there are no various phases in the workload and the variation is caused by multiplexing between performance events (discussed in Section 5.3). Yet, this is only a hypothesis that needs to be confirmed or disproved. Possible ways to proceed would be to collect more data points by running collection with higher granularity (in our case it's 10 sec) and study the source code. Be careful when drawing conclusions just from looking at the numbers, always obtain a second source of data that confirm your hypothesis.

In summary, looking at performance metrics helps building the right mental model about what is and what is *not* happening in a program. Going further into analysis, this data will serve you well.

Questions and Exercises

1. What is the difference between CPU core clock and reference clock?
2. What is the difference between retired and executed instruction?
3. When you increase the frequency, does IPC goes up, down, or stays the same?
4. Take a look at the `DRAM BW Use` formula in Table 4. Why do you think there is a constant `64`?
5. Measure bandwidth and latency of the cache hierarchy and memory on the machine you use for development/benchmarking using MLC, stream or other tools.
6. Run the application that you're working with on a daily basis. Collect performance metrics. Does anything surprises you?

Capacity Planning Exercize: Imagine you are the owner of four applications we benchmarked in the case study. The management of your company has asked you to build a small computing farm for each of those applications with the primary goal to maximize performance (throughput). A spending budget you were given is tight but enough to buy 1 mid-level server system (Mac Studio, Supermicro/Dell/HPE server rack, etc.) or 1 high-end desktop (with overclocked CPU, liquid cooling, top GPU, fast DRAM) to run each workload, so 4 machines in total. Those could be all four different systems. Also, you can use the money to buy 3-4 low-end systems, the choise is yours. The management wants to keep it under $10'000 per application, but they are flexible (10-20%) if you can justify the expense. Assume that Stockfish remains single-threaded. Look at the performance characteristics for the four applications once again and write down which computer parts (CPU, memory, discrete GPU if needed) you would buy for each of those workloads. Which specification parameters you will prioritize? Where you'll go with the most expensive part and where you can save money? Try to describe it in as much details as possible, search the web for exact components and their prices. Account for all the components of the system: motherboard, disk drive, cooling

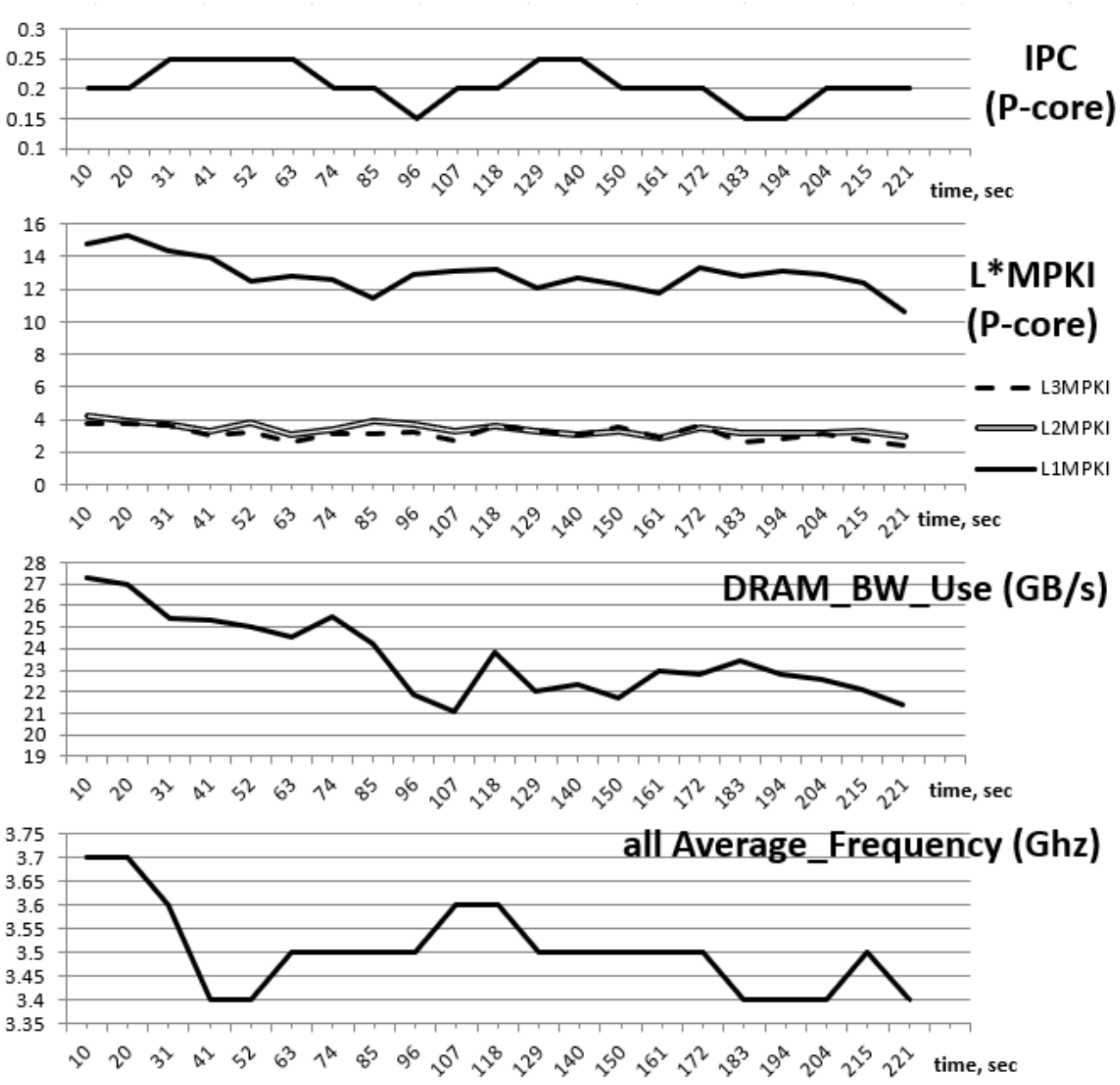


Figure 27: A set of metrics charts for the CloverLeaf benchmark.

solution, power delivery unit, case/tower, etc. What additional performance experiments you would run to guide your decision?

Chapter Summary

- In this chapter, we introduced the basic metrics in performance analysis such as retired/executed instructions, CPU utilization, IPC/CPI, UOPs, pipeline slots, core/reference clocks, cache misses and branch mispredictions. We showed how each of these metrics can be collected with Linux perf.
- For more advanced performance analysis, there are many derivative metrics that one can collect. For instance, MPKI (misses per kilo instructions), Ip* (instructions per function call, branch, load, etc), ILP, MLP and others. The case study in this chapter shows how we can get actionable insights from analyzing these metrics. Although, be carefull about drawing conclusions just by looking at the aggregate numbers. Don't fall in the trap of "excel performance engineering", i.e. only collect performance metrics and never look at the code. Always seek for a second source of data (e.g. performance profiles, discussed later) that will confirm your hypothesis.
- Memory bandwidth and latency are crucial factors in performance of many production SW packages nowadays, including AI, HPC, databases, and many general-purpose applications. Memory bandwidth depends on the DRAM speed (in MT/s) and the number of memory channels. Modern high-end server platforms have 8-12 memory channels and can reach up to 500 GB/s for the whole system and up to 50 GB/s in single-threaded mode. Memory latency nowadays doesn't change a lot, in fact it is getting slightly worse with new DDR4 and DDR5 generations. Majority of systems fall in the range of 70-110 ns per memory access.

5 Performance Analysis Approaches

When you're working on a high-level optimization, e.g. integrating a better algorithm in the application, it is usually easy to tell whether the performance improves or not since the benchmarking results are pronounced well. Big speedups, like 2x, 3x, etc. are relatively easy from performance analysis perspective. When you eliminate an extensive computation from a program, you expect to see a visible difference in the running time.

But also, there are situations when you see a small change in the execution time, say 5%, and you have no clue where it's coming from. Timing or throughput measurements alone do not provide any explanation on why performance goes up or down. In this case, we need more insights about how a program executes. That is the situation when we need to do performance analysis to understand the underlying nature of the slowdown or speedup that we observe.

Performance analysis is akin to detective's work. To solve a performance mystery, you gather all the data that you have and try to form a hypothesis. Once a hypothesis is made, you design an experiment that will either prove or disprove it. It can go back and forth a few times before you find a clue. And just like a good detective, you try to collect as many proofs as possible that will support your hypothesis. Once you have enough clues, you make a compelling explanation for the behavior you're observing.

When you just start working on a performance issue, you probably only have measurements, e.g. before and after the code change. Based on that measurements you conclude that the program became slower by `X` percent. If you know that the slowdown ocurred right after a certain commit, it may already give you enough information to fix the problem. But if you don't have good reference points, then the space of possible reasons for the slowdown is endless – you need to gather more data. One of the most popular approaches for collecting such data is to profile an application and look at the hotspots. This chapter introduces this and several other approaches for gathering data that were proven to be useful in performance engineering.

The next question comes: "What performance data is available and how to collect it?" Both HW and SW layers of the stack have facilities to track performance events and record them while our program is running. In this context, by HW, we mean the CPU, which executes the program, and by SW, we mean the OS, libraries, the application itself, and other tools armed for the analysis. Typically, the SW stack provides high-level metrics like time, number of context switches, and page-faults, while CPU monitors cache-misses, branch mispredictions, and other CPU-related events. Depending on the problem you are trying to solve, some metrics are more useful than others. So, it doesn't mean that HW metrics will always give us a more precise overview of the program execution. Some metrics, like the number of context-switches, for instance, cannot be provided by a CPU. Performance analysis tools, like Linux Perf, can consume data from both the OS and the CPU.

As you have probably guessed, there are hundreds of data sources that a performance engineer may use. Since this book is about CPU low-level performance, we will focus on collecting HW-level information. We will introduce some of the most popular performance analysis techniques: Code Instrumentation, Tracing, Characterization, Sampling, and the Roofline model. We also discuss static performance analysis techniques and compiler optimization reports which do not involve running the actual application.

5.1 Code Instrumentation

Probably the first approach for doing performance analysis ever invented is code *instrumentation*. It is a technique that inserts extra code into a program to collect specific runtime information. Listing 6 shows the simplest example of inserting a `printf` statement at the beginning of a function to count the number of times this function was called. After that you run the program and count the number of times you see "foo is called" in the output. I think every programmer in the world did it at some point of their carreer at least once.

Listing 6 Code Instrumentation

```
int foo(int x) {
+ printf("foo is called");
  // function body...
}
```

The plus sign at the beginning of a line means that this line was added and is not present in the original code. In general, instrumentation code is not meant to be pushed into the codebase, it's rather for collecting the needed data and then can be thrown away.

A slightly more interesting example of code instrumentation is presented in Listing 7. In this made-up code example, the function `findObject` searches the coordinates of an object with some properties `p` on a map. The function `findObj` returns the confidence level of locating the right object with the current coordinates `c`. If it is an exact match, we stop the search loop and return the coordinates. If the confidence is above the `threshold`, we choose to `zoomIn` to find more precise location of the object. Otherwise, we get the new coordinates within the `searchRadius` to try our search next time.

Instrumentation code consists of two classes: `histogram` and `incrementor`. The former keeps track of whatever variable values we are interested in and frequencies of their occurence and then prints the histogram *after* the program finishes. The latter is just a helper class for pushing values into the `histogram` object. It is simple enough and can be adjusted to your specific needs quickly. I have a slightly more advanced version of this code which I usually copy-paste into whatever project I'm working on and then delete.

Listing 7 Code Instrumentation

```
+ struct histogram {
+   std::map<uint32_t, std::map<uint32_t, uint64_t>> hist;
+   ~histogram() {
+     for (auto& tripCount : hist)
+       for (auto& zoomCount : tripCount.second)
+         std::cout << "[" << tripCount.first << "]["
+                   << zoomCount.first << "] :  "
+                   << zoomCount.second << "\n";
+   }
+ };
+ histogram h;

+ struct incrementor {
+   uint32_t tripCount = 0;
+   uint32_t zoomCount = 0;
+   ~incrementor() {
+      h.hist[tripCount][zoomCount]++;
+   }
+ };

Coords findObject(const ObjParams& p, Coords c, float searchRadius) {
+ incrementor inc;
  while (true) {
+   inc.tripCount++;
    float match = findObj(c, p);
    if (exactMatch(match))
      return c;
    if (match > threshold) {
      searchRadius = zoomIn(c, searchRadius);
+     inc.zoomCount++;
    }
    c = getNewCoords(searchRadius);
  }
  return c;
}
```

In this hypothetical scenario, we added instrumentation to know how frequently we `zoomIn` before we find an object. The variable `inc.tripCount` counts the number of iterations the loop runs before it exits, and the variable

`inc.zoomCount` counts how many times we reduce the search radius. We always expect `inc.zoomCount` to be less or equal `inc.tripCount`. Here is the output one may observe after running the instrumented program:

```
[7][6]:  2
[7][5]:  6
[7][4]:  20
[7][3]:  156
[7][2]:  967
[7][1]:  3685
[7][0]:  251004
[6][5]:  2
[6][4]:  7
[6][3]:  39
[6][2]:  300
[6][1]:  1235
[6][0]:  91731
[5][4]:  9
[5][3]:  32
[5][2]:  160
[5][1]:  764
[5][0]:  34142
[4][4]:  5
[4][3]:  31
[4][2]:  103
[4][1]:  195
[4][0]:  14575
...
```

The first number in the square bracket is the trip count of the loop, and the second is the number of `zoomIn`s we made within the same loop. The number after the column sign is the number of occurences of that particular combination of the numbers. For example, two times we observed 7 loop iterations and 6 `zoomIn`s, 251004 times the loop ran 7 iterations and no `zoomIn`s, and so on. You can then plot the data for better visualization, employ some other statistical methods, but the main point we can make is that `zoomIn`s are not frequent. There were a total of 10k `zoomIn` calls for the 400k times the `findObject` was called.

Later in the book you will see many examples how such information can be used for data-driven optimizations. In our case, we can assume that `findObject` often fails to find the object. It means that the next iteration of the loop will try to find the object using new coordinates but still within the same search radius. Knowing that, we could attempt a number of optimizations: 1) run multiple searches in parallel, synchronize if any of them succeeded, 2) precompute certain things for the current search region, thus eliminating repetitive work inside `findObj`, 3) software pipeline generating next coordinates (`getNewCoords`) and prefetch corresponding map locations from memory. Part 2 of the book looks deeper into some of this techniques.

Code instrumentation provides very detailed information when you need specific knowledge about the execution of the program. It allows us to track any information about every variable in the program. Using such a method often yields the best insight when optimizing big pieces of code because you can use a top-down approach (instrumenting the main function then drilling down to its callees) of locating performance issues. While code instrumentation is not very helpful in the case of small programs, it gives the most value and insight by letting developers observe the architecture and flow of an application. This technique is especially helpful for someone working with an unfamiliar codebase.

It's also worth mentioning that code instrumentation shines in complex systems with many different components that react differently based on inputs or over time. Sampling techniques (discussed in Section 5.4) squash that valuable information, not allowing us to detect abnormal behaviors. For example, in games, usually, there is a renderer thread, a physics thread, an animations thread, etc. Instrumenting such big modules helps to reasonably quickly understand what module is the source of issues. As sometimes, optimizing is not only a matter of optimizing code but also data. For example, rendering is too slow because of uncompressed mesh, or physics are too slow because of too many objects in a scene.

The instrumentation technique is heavily used in performance analysis of real-time scenarios, such as video games and embedded development. Some profilers mix up instrumentation with other techniques like tracing and sampling. We will look at one of such hybrid profilers called Tracy in Section 7.7.

While code instrumentation is powerful in many cases, it does not provide any information about how the code executes from the OS or CPU perspective. For example, it can't give you information about how often the process was scheduled in and out from the execution (known by the OS) or how many branch mispredictions occurred (known by the CPU). Instrumented code is a part of an application and has the same privileges as the application itself. It runs in userspace and doesn't have access to the kernel.

But more importantly, the downside of this technique is that every time something new needs to be instrumented, say another variable, recompilation is required. This can become a burden to an engineer and increase analysis time. Unfortunately, there are additional downsides. Since usually, you care about hot paths in the application, you're instrumenting the things that reside in the performance-critical part of the code. Injecting instrumentation code in a hot path might easily result in a 2x slowdown of the overall benchmark. Remember not to benchmark instrumented program, i.e., do not measure score and do analysis in the same run. Keep in mind that by instrumenting the code, you change the behavior of the program, so you might not see the same effects you saw earlier.

All of the above increases time between experiments and consumes more development time, which is why engineers don't manually instrument their code very often these days. However, automated code instrumentation is still widely used by compilers. Compilers are capable of automatically instrumenting the whole program and collect interesting statistics about the execution. The most widely known use cases for automated instrumentation are code coverage analysis and Profile Guided Optimizations (see Section 11.7).

When talking about instrumentation, it's important to mention binary instrumentation techniques. The idea behind binary instrumentation is similar but it is done on an already-built executable file and not on a source code level. There are two types of binary instrumentation: static (done ahead of time) and dynamic (instrumentation code inserted on-demand as a program executes). The main advantage of dynamic binary instrumentation is that it does not require program recompilation and relinking. Also, with dynamic instrumentation, one can limit the amount of instrumentation to only interesting code regions, not the whole program.

Binary instrumentation is very useful in performance analysis and debugging. One of the most popular tools for binary instrumentation is the Intel Pin[61] tool. Pin intercepts the execution of the program in the occurrence of an interesting event and generates new instrumented code starting at this point in the program. It allows collecting various runtime information, for example:

- instruction count and function call counts.
- intercepting function calls and execution of any instruction in an application.
- allows "record and replay" the program region by capturing the memory and HW registers state at the beginning of the region.

Like code instrumentation, binary instrumentation only allows instrumenting user-level code and can be very slow.

5.2 Tracing

Tracing is conceptually very similar to instrumentation yet slightly different. Code instrumentation assumes that the user can orchestrate the code of their application. On the other hand, tracing relies on the existing instrumentation of a program's external dependencies. For example, the `strace` tool enables us to trace system calls and can be thought of as the instrumentation of the Linux kernel. Intel Processor Traces (see Appendix D) enables you to log instructions executed by the program and can be thought of as the instrumentation of the CPU. Traces can be obtained from components that were appropriately instrumented in advance and are not subject to change. Tracing is often used as the black-box approach, where a user cannot modify the code of the application, yet they want insights on what the program is doing behind the scenes.

An example of tracing system calls with the Linux `strace` tool is provided in Listing 8, which shows the first several lines of output when running the `git status` command. By tracing system calls with `strace` it's possible to know the timestamp for each system call (the leftmost column), its exit status, and the duration of each system call (in the angle brackets).

[61] PIN - https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

Listing 8 Tracing system calls with strace.

```
$ strace -tt -T -- git status
17:46:16.798861 execve("/usr/bin/git", ["git", "status"], 0x7ffe705dcd78
                  /* 75 vars */) = 0 <0.000300>
17:46:16.799493 brk(NULL)               = 0x55f81d929000 <0.000062>
17:46:16.799692 access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT
                  (No such file or directory) <0.000063>
17:46:16.799863 access("/etc/ld.so.preload", R_OK) = -1 ENOENT
                  (No such file or directory) <0.000074>
17:46:16.800032 openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
                  <0.000072>
17:46:16.800255 fstat(3, {st_mode=S_IFREG|0644, st_size=144852, ...}) = 0
                  <0.000058>
17:46:16.800408 mmap(NULL, 144852, PROT_READ, MAP_PRIVATE, 3, 0)
                  = 0x7f6ea7e48000 <0.000066>
17:46:16.800619 close(3)                = 0 <0.000123>
...
```

The overhead of tracing very much depends on what exactly we try to trace. For example, if we trace a program that almost never makes system calls, the overhead of running it under `strace` will be close to zero. On the other hand, if we trace a program that heavily relies on system calls, the overhead could be very large, e.g. 100x.[62] Also, tracing can generate a massive amount of data since it doesn't skip any sample. To compensate for this, tracing tools provide filters that enable you to restrict data collection to a specific time slice or for a specific section of code.

Usually, tracing similar to instrumentation is used for exploring anomalies in the system. For example, you may want to determine what was going on in an application during a 10s period of unresponsiveness. As you will see later, sampling methods are not designed for this, but with tracing, you can see what lead to the program being unresponsive. For example, with Intel PT, you can reconstruct the control flow of the program and know exactly what instructions were executed.

Tracing is also very useful for debugging. Its underlying nature enables "record and replay" use cases based on recorded traces. One such tool is the Mozilla rr[63] debugger, which performs record and replay of processes, supports backwards single stepping and much more. Most of the tracing tools are capable of decorating events with timestamps, which allows us to have a correlation with external events that were happening during that time. That is, when we observe a glitch in a program, we can take a look at the traces of our application and correlate this glitch with what was happening in the whole system during that time.

5.3 Workload Characterization

Workload characterization is a process of describing a workload by means of quantitative parameters and functions. In simple words, it means counting an absolute number of certain performance events. The goal of characterization is to define the behavior of the workload and extract its most important features. On a high level, an application can belong to one or many of the following types: interactive, database, real-time, network-based, massively parallel, etc. Different workloads can be characterized using different metrics and parameters to address a particular application domain.

This is a book about low-level performance, remember? So, we will focus on extracting features related to the CPU and memory performance. The best example of a workload characterization from a CPU perspective is Top-down Microarchitecture Analysis (TMA) methodology, which we will closely look at in Section 6.1. TMA attempts to characterize an application by putting it into one of 4 buckets: CPU Front End, CPU Back End, Retiring, and Bad Speculation depending on what is causing the most performance issues. TMA uses Performance Monitoring Counters (PMCs) to collect the needed information and identify the inefficient use of CPU microarchitecture.

But even without a fully-fledged characterization methodology, collecting absolute number of certain performance

[62] An article about `strace` by B. Gregg - http://www.brendangregg.com/blog/2014-05-11/strace-wow-much-syscall.html
[63] Mozilla rr debugger - https://rr-project.org/.

events can be helpful. We hope that the case study in the previous chapter that examined performance metrics of four different benchmarks, proved that. PMCs are a very important instrument of low-level performance analysis. They can provide unique information about the execution of a program. PMCs are generally used in two modes: "Counting" and "Sampling". Counting mode is used for workload characterization, while sampling mode is used for finding hotspots, which we will discuss soon.

5.3.1 Counting Performance Events

The idea behind counting is very simple: we want to count the absolute number of certain performance events while our program is running. Figure 28 illustrates the process of counting performance events from the start to the end of a program.

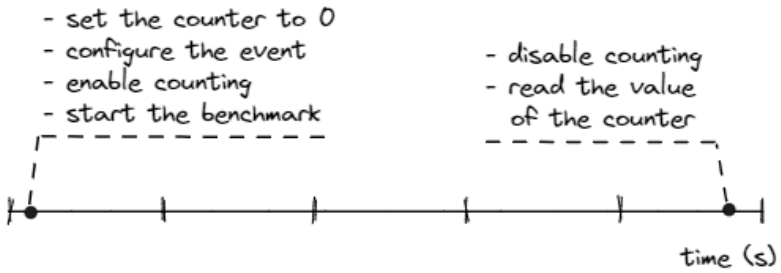


Figure 28: Counting performance events.

The steps outlined in Figure 28 roughly represent what a typical analysis tool will do to count performance events. This process is implemented in the `perf stat` tool, which can be used to count various HW events, like the number of instructions, cycles, cache-misses, etc. Below is an example of the output from `perf stat`:

```
$ perf stat -- ./a.exe
 10580290629  cycles          #    3,677 GHz
  8067576938  instructions    #    0,76  insn per cycle
  3005772086  branches        # 1044,472 M/sec
   239298395  branch-misses   #    7,96% of all branches
```

It is very informative to know this data. First of all, it enables us to quickly spot some anomalies like a high branch misprediction rate or low IPC. In addition, it might come in handy when you've made a code change and you want to verify that the change has improved performance. Looking at absolute numbers might help you justify or reject the code change. The main author uses `perf stat` as a simple benchmark wrapper. Since the overhead of counting events is minimal, almost all benchmarks can be automatically ran under `perf stat`. It serves as a first step in performance investigation. Sometimes anomalies can be spotted right away, which can save you some analysis time.

5.3.2 Manual Performance Counters Collection

Modern CPUs have hundreds of countable performance events. It's very hard to remember all of them and their meanings. Understanding when to use a particular PMC is even harder. That is why generally, we don't recommend manually collecting specific PMCs unless you really know what you are doing. Instead, we recommend using tools like Intel Vtune Profiler that automate this process. Nevertheless, there are situations when you are interested in collecting specific PMCs.

A complete list of performance events for all Intel CPU generations can be found in [Intel, 2023b, Volume 3B, Chapter 19]. PMCs description is also available at perfmon-events.intel.com. Every event is encoded with `Event` and `Umask` hexadecimal values. Sometimes performance events can also be encoded with additional parameters, like `Cmask`, `Inv` and others. An example of encoding two performance events for the Intel Skylake microarchitecture is shown in Table 6.

Table 6: Example of encoding Skylake performance events.

Event Num.	Umask Value	Event Mask Mnemonic	Description
C0H	00H	INST_RETIRED. ANY_P	Number of instructions at retirement.
C4H	00H	BR_INST_RETIRED. ALL_BRANCHES	Branch instructions that retired.

Linux `perf` provides mappings for commonly used performance counters. They can be accessed via pseudo names instead of specifying `Event` and `Umask` hexadecimal values. For example, `branches` is just a synonym for `BR_INST_RETIRED.ALL_BRANCHES` and will measure the same thing. List of available mapping names can be viewed with `perf list`:

```
$ perf list
  branches           [Hardware event]
  branch-misses      [Hardware event]
  bus-cycles         [Hardware event]
  cache-misses       [Hardware event]
  cycles             [Hardware event]
  instructions       [Hardware event]
  ref-cycles         [Hardware event]
```

However, Linux `perf` doesn't provide mappings for all performance counters for every CPU architecture. If the PMC you are looking for doesn't have a mapping, it can be collected with the following syntax:

```
$ perf stat -e cpu/event=0xc4,umask=0x0,name=BR_INST_RETIRED.ALL_BRANCHES/ -- ./a.exe
```

Performance counters are not available in every environment since accessing PMCs requires root access, which applications running in a virtualized environment typically do not have. For programs executing in a public cloud, running a PMU-based profiler directly in a guest container does not result in useful output if a virtual machine (VM) manager does not expose the PMU programming interfaces properly to a guest. Thus profilers based on CPU performance counters do not work well in a virtualized and cloud environment [Du et al., 2010]. Although the situation is improving. VmWare® was one of the first VM managers to enable[64] virtual CPU Performance Counters (vPMC). AWS EC2 cloud enabled[65] PMCs for dedicated hosts.

5.3.3 Multiplexing and Scaling Events

There are situations when we want to count many different events at the same time. But with only one counter, it's possible to count only one thing at a time. That's why PMUs have multiple counters in it (in recent Intel's Goldencove microarchitecture there are 12 programmable PMCs, 6 per HW thread). Even then, the number of fixed and programmable counter is not always sufficient. Top-down Microarchitecture Analysis (TMA) methodology requires collecting up to 100 different performance events in a single execution of a program. Modern CPUs don't have that many counters, and here is when multiplexing comes into play.

If there are more events than counters, the analysis tool uses time multiplexing to give each event a chance to access the monitoring hardware. Figure 29a shows an example of multiplexing between 8 performance events with only 4 PMCs available.

With multiplexing, an event is not measured all the time, but rather only during a portion of time. At the end of the run, a profiling tool needs to scale the raw count based on total time enabled:

$$final\ count = raw\ count \times (time\ running / time\ enabled)$$

Let's take Figure 29b as an example. Say, during profiling, we were able to measure an event from group 1 during three time intervals. Each measurement interval lasted 100ms (`time enabled`). The program running time was

[64] VMWare PMCs - https://www.vladan.fr/what-are-vmware-virtual-cpu-performance-monitoring-counters-vpmcs/
[65] Amazon EC2 PMCs - http://www.brendangregg.com/blog/2017-05-04/the-pmcs-of-ec2.html

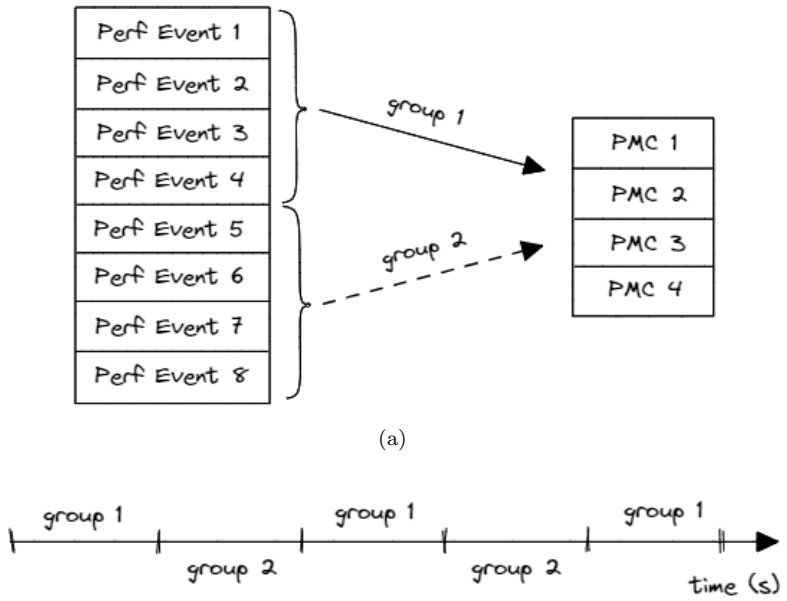


Figure 29: Multiplexing between 8 performance events with only 4 PMCs available.

500ms (`time running`). The total number of events for this counter was measured as 10'000 (`raw count`). So, the final count needs to be scaled as follows:

$$final\ count = 10'000 \times (500ms/(100ms \times 3)) = 16'666$$

This provides an estimate of what the count would have been had the event been measured during the entire run. It is very important to understand that this is still an estimate, not an actual count. Multiplexing and scaling can be used safely on steady workloads that execute the same code during long time intervals. However, if the program regularly jumps between different hotspots, i.e. has different phases, there will be blind spots that can introduce errors during scaling. To avoid scaling, one can try to reduce the number of events to be not bigger than the number of physical PMCs available. However, this will require running the benchmark multiple times to measure all the counters one is interested in.

5.3.4 Using Marker APIs

In certain scenarios, we might be interested in analyzing performance of a specific code region, not an entire application. This can be a situation when you're developing a new piece of code and want to focus just on that code. Naturally, you would like to track optimization progress and capture additional performance data that will help you along the way. Most performance analysis tools provide specific *marker APIs* that let you do that. Here are a few examples:

- Likwid has `LIKWID_MARKER_START / LIKWID_MARKER_STOP` macros.
- Intel VTune has `__itt_task_begin / __itt_task_end` functions.
- AMD uProf has `amdProfileResume / amdProfilePause` functions.

Such a hybrid approach combines benefits of instrumentation and performance events couting. Instead of measuring the whole program, marker APIs allow us to attribute performance statistics to code regions (loops, functions) or functional piecies (remote procedure calls (RPCs), input events, etc.). The quality of the data you get back can easily justify the effort. While chasing performance bug that happens only with a specific type of RPCs, you can enable monitoring just for that type of RPC.

Below we provide a very basic example of using libpfm4,[66] one of the popular Linux libraries for collecting performance monitoring events. It is built on top of the Linux `perf_events` subsystem, which lets you access performance

[66] libpfm4 - https://sourceforge.net/p/perfmon2/libpfm4/ci/master/tree/

event counters directly. The `perf_events` subsystem is rather low-level, so the `libfm4` package is useful here, as it adds both a discovery tool for identifying available events on your CPU, and a wrapper library around the raw `perf_event_open` system call. Listing 9 shows how one can use `libpfm4` to instrument the `render` function of the C-Ray[67] benchmark.

In this code example, we first initialize `libpfm` library and configure performance events and the format that we will use to read them. In the C-Ray benchmark, the `render` function is only called once. In your own code, be carefull about not doing `libpfm` initialization multiple times. Then, we choose the code region we want to analyze, in our case it is a loop with a `trace` function call inside. We surround this code region with two `read` system calls that will capture values of performance counters before and after the loop. Next, we save the deltas for later processing, in this case, we aggregated (code is not shown) it by calculating average, 90th percentile and maximum values. Running it on an Intel Alderlake-based machine, we've got the output shown below. Root priviledges are not required, but `/proc/sys/kernel/perf_event_paranoid` should be set to less than 1. When reading counters from inside a thread, the values are for that thread alone. It can optionally include kernel code that ran and was attributed to the thread.

```
$ ./c-ray-f -s 1024x768 -r 2 -i sphfract -o output.ppm
Per-pixel ray tracing stats:
                       avg         p90         max
------------------------------------------------
nanoseconds   |      4571 |      6139 |     25567
instructions  |     71927 |     96172 |    165608
cycles        |     20474 |     27837 |    118921
branches      |      5283 |      7061 |     12149
branch-misses |        18 |        35 |       146
```

Remember, that the instrumentation that we added measures the per-pixel ray tracing stats. Multiplying average numbers by the number of pixels (`1024x768`) should give us roughly the total stats for the program. A good sanity check in this case is to run `perf stat` and compare the overall C-Ray statistics for the performance events that we've collected.

The C-ray benchmark primarily stresses the floating-point performance of a CPU core, which generally should not cause high variance in the measuments, in other words, we expect all the measurements to be very close to each other. However, we see that it's not the case, as p90 values are 1.33x average numbers and max is sometimes 5x slower than the average case. The most likely explanation here is that for some pixels the algorithm hits a corner case, executes more instructions and subsequently runs longer. But it's always good to confirm the hypothesis by studying the source code or extending the instrumentation to capture more data for the "slow" pixels.

The additional instrumentation code showed in Listing 9 causes 17% overhead, which is OK for local experiments, but quite high to run in production. Most large distributed systems aim for less than 1% overhead, and for some up to 5% can be tolerable, but it's unlikely that users would be happy with 17% slowdown. Managing the overhead of your instrumentation is critical, especially if you choose to enable it in production environment.

Overhead is usefully calculated as occurence rate per unit of time or work (RPC, database query, loop iteration, etc.). If a system call on our system is roughly 1.6 microseconds of CPU time, and we do it twice for each pixel (iteration of the outer loop), the overhead is 3.2 microseconds of CPU time per pixel.

There are many strategies to bring the overhead down. As a general rule, your instrumentation should always have a fixed cost, e.g., a deterministic syscall, but not a list traversal or dynamic memory allocation, otherwise it will interfere with the measurements. The instumentation code has three logical parts: collecting the information, storing it, and reporting it. To lower the overhead of the first part (collection), we can decrease the sampling rate, e.g. sample each 10th RPC and skip the rest. For a long-running application, performance can be monitored with a relatively cheap random sampling - randomly select which events to observe. These methods sacrifice collection accuracy but still provide a good estimate of the overall performance characteristics while incurring a very low overhead.

For the second and third parts (storing and aggregating), the recommendation is to only collect, processes, and retain only much data as you need to understand the performance of the system. You can avoid storing every sample in memory by using "online" algorithms for calculating mean, variance, min, max and other metrics. This will

[67] C-Ray benchmark - https://openbenchmarking.org/test/pts/c-ray

drastically reduce the memory footprint of the instrumentation. For instance, variance and standard deviation can be calculated using Knuth's online-variance algorithm. A good implementation[68] uses less than 50 bytes of memory.

For long routines, you can collect counters at the beginning, end, and some parts in the middle. Over consequtive runs, you can binary search for the part of the routine that performs poorest and optimize it. Repeat this until all the poorly-performing spots are removed. If tail latency is of a primary concern, emitting log messages on a particularly slow run can provide useful insights.

In the Listing 9, we collected 4 events simultaneously, though the CPU has 6 programmable counters. You can open up additional groups with different sets of events enabled. The kernel will select different groups to run at a time. The `time_enabled` and `time_running` fields indicate the multiplexing. They are both durations in nanoseconds. The `time_enabled` field indicates how many nanoseconds the event group has been enabled. The `time_running` indicates how much of that enabled time the events were actually collecting. If you had two event groups enabled simultaneously that couldn't fit together on the HW counters, you might see them both converge to `time_running = 0.5 * time_enabled`. Scheduling in general is complicated so verify before depending on your exact scenario.

Capturing multiple events simultaneously allows to calculate various metrics that we discussed in Chapter 4. For example, capturing `INSTRUCTIONS_RETIRED` and `UNHALTED_CLOCK_CYCLES` enables us to measure IPC. We can observe the effects of frequency scaling by comparing CPU cycles (`UNHALTED_CORE_CYCLES`) vs the fixed-frequency reference clock (`UNHALTED_REFERENCE_CYCLES`). It is possible to detect when the thread wasn't running by requesting CPU cycles consumed (`UNHALTED_CORE_CYCLES`, only counts when the thread is running) and comparing against wall-clock time. Also, we can normalize the numbers to get the event rate per second/clock/instruction. For instance, measuring `MEM_LOAD_RETIRED.L3_MISS` and `INSTRUCTIONS_RETIRED` we can get the `L3MPKI` metric. As you can see, the setup is very flexible.

The important property of grouping events is that the counters will be available atomically under the same `read` system call. These atomic bundles are very useful. First, it allows us to correlate events within each group. Say we measure IPC for a region of code, and found that it is very low. In this case, we can pair two events (instructions and cycles) with a third one, say L3 cache misses, to check if it contributes to a low IPC that we're dealing with. If it doesn't, we continue factor analysis using other events. Second, event grouping helps to mitigate bias in case a workload has different phases. Since all the events within a group are measured at the same time, they always capture the same phase.

In some scenarios, instrumentation may become a part of a functionality or a feature. For example, a developer can implement an instrumentation logic that detects decrease in IPC (e.g. when there is a busy sibling HW thread running) or decreasing CPU frequency (e.g. system throttling due to heavy load). When such event occurs, application automatically defers low-priority work to compensate for the temporarily increased load.

5.4 Sampling

Sampling is the most frequently used approach for doing performance analysis. People usually associate it with finding hotspots in the program. To put it broadly, sampling helps to find places in the code that contribute to the biggest number of certain performance events. If we consider finding hotspots, the problem can be reformulated as which place in the code consumes the biggest amount of CPU cycles. People often use the term "Profiling" for what is technically called sampling. According to Wikipedia,[69] profiling is a much broader term and includes a wide variety of techniques to collect data, including interrupts, code instrumentation, and PMC.

It may come as a surprise, but the simplest sampling profiler one can imagine is a debugger. In fact, you can identify hotspots by a) run the program under the debugger, b) pause the program every 10 seconds, and c) record the place where it stopped. If you repeat b) and c) many times, you can build a histogram from those samples. The line of code where you stopped the most will be the hottest place in the program. Of course, this is not an efficient way to find hotspots, and we don't recommend doing this. It's just to illustrate the concept. Nevertheless, this is a simplified description of how real profiling tools work. Modern profilers are capable of collecting thousands of samples per second, which gives a pretty accurate estimate about the hottest places in a benchmark.

As in the example with a debugger, the execution of the analyzed program is interrupted every time a new sample is captured. At the time of interrupt, the profiler collects the snapshot of the program state, which constitutes one

[68] Accurately computing running variance - https://www.johndcook.com/blog/standard_deviation/
[69] Profiling(wikipedia) - https://en.wikipedia.org/wiki/Profiling_(computer_programming).

sample. Information collected for every sample may include an instruction address that was executed at the time of interrupt, register state, call stack (see Section 5.4.3), etc. Collected samples are stored in a dump file, which can be further used to display most time-consuming parts of the program, a call graph, etc.

5.4.1 User-Mode and Hardware Event-based Sampling

Sampling can be performed in 2 different modes, using user-mode or HW event-based sampling (EBS). User-mode sampling is a pure SW approach that embeds an agent library into the profiled application. The agent sets up the OS timer for each thread in the application. Upon timer expiration, the application receives the `SIGPROF` signal that is handled by the collector. EBS uses hardware PMCs to trigger interrupts. In particular, the counter overflow feature of the PMU is used, which we will discuss shortly.

User-mode sampling can only be used to identify hotspots, while EBS can be used for additional analysis types that involve PMCs, e.g., sampling on cache-misses, TMA (see Section 6.1), etc.

User-mode sampling incurs more runtime overhead than EBS. The average overhead of the user-mode sampling is about 5% when sampling with the interval of 10ms, while EBS has less than 1% overhead. Because of less overhead, you can use EBS with a higher sampling rate which will give more accurate data. However, user-mode sampling generates fewer data to analyze, and it takes less time to process it.

5.4.2 Finding Hotspots

In this section, we will discuss the mechanics of using PMCs with EBS. Figure 30 illustrates the counter overflow feature of the PMU, which is used to trigger performance monitoring interrupt (PMI), aka `SIGPROF`. At the start of a benchmark, we configure the event that we want to sample on. Identifying hotspots means knowing where the program spends most of the time. So sampling on cycles is very natural, and it is a default for many profiling tools. But it's not necessarily a strict rule; we can sample on any performance event we want. For example, if we would like to know the place where the program experiences the biggest number of L3-cache misses, we would sample on the corresponding event, i.e., `MEM_LOAD_RETIRED.L3_MISS`.

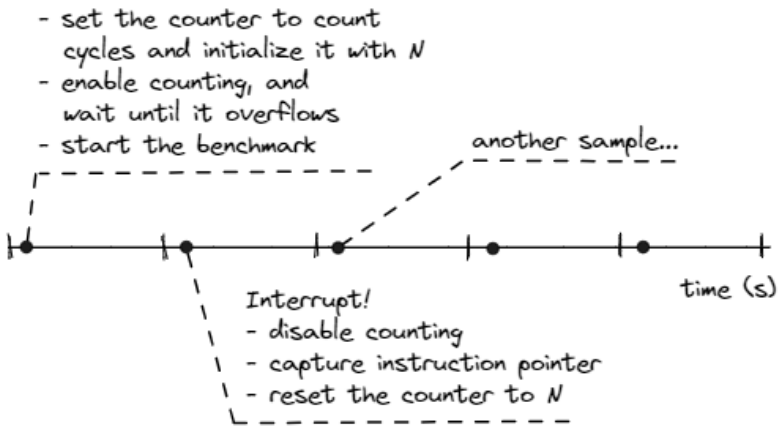


Figure 30: Using performance counter for sampling

After we initialized the register, we start counting and let the benchmark go. We configured PMC to count cycles, so it will be incremented every cycle. Eventually, it will overflow. At the time the register overflows, HW will raise a PMI. The profiling tool is configured to capture PMIs and has an Interrupt Service Routine (ISR) for handling them. We do multiple steps inside ISR: first of all, we disable counting; after that, we record the instruction which was executed by the CPU at the time the counter overflowed; then, we reset the counter to `N` and resume the benchmark.

Now, let us go back to the value `N`. Using this value, we can control how frequently we want to get a new interrupt. Say we want a finer granularity and have one sample every 1 million instructions. To achieve this, we can set the counter to `(unsigned) -1'000'000` so that it will overflow after every 1 million instructions. This value is also referred to as the "sample after" value.

We repeat the process many times to build a sufficient collection of samples. If we later aggregate those samples, we could build a histogram of the hottest places in our program, like the one shown on the output from Linux

`perf record/report` below. This gives us the breakdown of the overhead for functions of a program sorted in descending order (hotspots). An example of sampling the x264[70] benchmark from the Phoronix test suite[71] is shown below:

```
$ time -p perf record -F 1000 -- ./x264 -o /dev/null --slow --threads 1
    ../Bosphorus_1920x1080_120fps_420_8bit_YUV.y4m
[ perf record: Captured and wrote 1.625 MB perf.data (35035 samples) ]
real 36.20 sec
$ perf report -n --stdio
# Samples: 35K of event 'cpu_core/cycles/'
# Event count (approx.): 156756064947
# Overhead  Samples  Shared Object  Symbol
# ........  .......  .............  ......................................
  7.50%     2620     x264           [.] x264_8_me_search_ref
  7.38%     2577     x264           [.] refine_subpel.lto_priv.0
  6.51%     2281     x264           [.] x264_8_pixel_satd_8x8_internal_avx2
  6.29%     2212     x264           [.] get_ref_avx2.lto_priv.0
  5.07%     1787     x264           [.] x264_8_pixel_avg2_w16_sse2
  3.26%     1145     x264           [.] x264_8_mc_chroma_avx2
  2.88%     1013     x264           [.] x264_8_pixel_satd_16x8_internal_avx2
  2.87%     1006     x264           [.] x264_8_pixel_avg2_w8_mmx2
  2.58%      904     x264           [.] x264_8_pixel_satd_8x8_avx2
  2.51%      882     x264           [.] x264_8_pixel_sad_16x16_sse2
  ...
```

Linux perf collected `35'035` samples, which means that the process of interrupting the execution happened so many times. We also used `-F 1000` which sets the sampling rate at 1000 samples per second. This roughly matches the overall runtime of 36.2 seconds. Notice, Linux perf provided the approximate number of total cycles elapsed. If we divide it by the number of samples, we'll have `156756064947 cycles / 35035 samples = 4.5 million` cycles per sample. That means that Linux perf set the number `N` to roughly `4'500'000` to collect 1000 samples per second. The number `N` can be adjusted by the tool dynamically according to the actual CPU frequency.

And of course, the most valuable for us is the list of hotspots sorted by the number of samples attributed to each function. After we know what are the hottest functions, we may want to look one level deeper: what are the hot parts of code inside every function. To see the profiling data for functions that were inlined as well as assembly code generated for a particular source code region, we need to build the application with debug information (`-g` compiler flag).

There are two main uses cases for the debug information: debugging a functional issue (a bug) and performance analysis. For functional debugging we need as much information as possible, which is the default when you pass `-g` compiler flag. However, if a user doesn't need full debug experience, having line numbers is enough for performance profiling. You can reduce the amount of generated debug information to just line numbers of symbols as they appear in the source code by using the `-gline-tables-only` option.[72]

Linux `perf` doesn't have rich graphic support, so viewing hot parts of source code is not very convinient, but doable. Linux `perf` intermixes source code with the generated assembly, as shown below:

```
# snippet of annotating source code of 'x264_8_me_search_ref' function
$ perf annotate x264_8_me_search_ref --stdio
Percent | Source code & Disassembly of x264 for cycles:ppp
------------------------------------------------------------
  ...
         :                  bmx += square1[bcost&15][0];   <== source code
  1.43   : 4eb10d:  movsx   ecx,BYTE PTR [r8+rdx*2]        <== corresponding machine code
         :                  bmy += square1[bcost&15][1];
```

[70] x264 benchmark - https://openbenchmarking.org/test/pts/x264.
[71] Phoronix test suite - https://www.phoronix-test-suite.com/.
[72] In the past there were LLVM compiler bugs when compiling with debug info (`-g`). Code transformation passes incorrectly treated the presence of debugging intrinsics which caused different optimizations decisions. It did not affect functionality, only performance. Some of them were fixed, but it's hard to say if any of them are still there.

```
 0.36  : 4eb112:  movsx  r12d,BYTE PTR [r8+rdx*2+0x1]
       :                 bmx += square1[bcost&15][0];
 0.63  : 4eb118:  add    DWORD PTR [rsp+0x38],ecx
       :                 bmy += square1[bcost&15][1];
 ...
```

Most profilers with a Graphical User Interface (GUI), like the Intel VTune Profiler, can show source code and associated assembly side-by-side. Also, there are tools that can visualize the output of Linux `perf` raw data with a rich graphical interface similar to Intel Vtune and other tools. You'll see all that in more details in chapter 7.

5.4.3 Collecting Call Stacks

Often when sampling, we might encounter a situation when the hottest function in a program gets called from multiple functions. An example of such a scenario is shown in Figure 31. The output from the profiling tool might reveal that `foo` is one of the hottest functions in the program, but if it has multiple callers, we would like to know which one of them calls `foo` the most number of times. It is a typical situation for applications that have library functions like `memcpy` or `sqrt` appear in the hotspots. To understand why a particular function appeared as a hotspot, we need to know which path in the Control Flow Graph (CFG) of the program caused it.

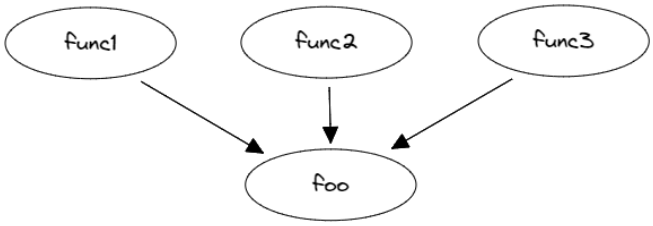


Figure 31: Control Flow Graph: hot function "foo" has multiple callers.

Analyzing the source code of all the callers of `foo` might be very time-consuming. We want to focus only on those callers that caused `foo` to appear as a hotspot. In other words, we want to figure out the hottest path in the CFG of a program. Profiling tools achieve this by capturing the call stack of the process along with other information at the time of collecting performance samples. Then, all collected stacks are grouped, allowing us to see the hottest path that led to a particular function.

Collecting call stacks in Linux `perf` is possible with three methods:

1. Frame pointers (`perf record --call-graph fp`). Requires binary being built with `--fnoomit-frame-pointer`. Historically, the frame pointer (`RBP` register) was used for debugging since it enables us to get the call stack without popping all the arguments from the stack (aka stack unwinding). The frame pointer can tell the return address immediately. However, it consumes one register just for this purpose, so it was expensive. It can also be used for profiling since it enables cheap stack unwinding.
2. DWARF debug info (`perf record --call-graph dwarf`). Requires binary being built with DWARF debug information `-g` (`-gline-tables-only`). Obtains call stacks through stack unwinding procedure.
3. Intel Last Branch Record (LBR) Hardware feature `perf record --call-graph lbr`. Obtains call stacks by parsing the LBR stack (a set of HW registers). Not as deep call graph as the first two methods. See more information about LBR in Section 6.2.

Below is the example of collecting call stacks in a program using LBR. By looking at the output, we know that 55% of the time `foo` was called from `func1`, 33% of the time from `func2` and 11% from `fun3`. We can clearly see the distribution of the overhead between callers of `foo` and can now focus our attention on the hottest edge in the CFG of the program, which is `func1 -> foo`, but we should probably also pay attention to the edge `func2 -> foo`.

```
$ perf record --call-graph lbr -- ./a.out
$ perf report -n --stdio --no-children
# Samples: 65K of event 'cycles:ppp'
# Event count (approx.): 61363317007
# Overhead       Samples  Command  Shared Object     Symbol
# ........  ............  .......  ................  ......................
```

```
  99.96%         65217  a.out    a.out             [.] foo
         |
          --99.96%--foo
                    |
                    |--55.52%--func1
                    |          main
                    |          __libc_start_main
                    |          _start
                    |
                    |--33.32%--func2
                    |          main
                    |          __libc_start_main
                    |          _start
                    |
                     --11.12%--func3
                               main
                               __libc_start_main
                               _start
```

When using Intel Vtune Profiler, one can collect call stacks data by checking the corresponding "Collect stacks" box while configuring analysis. When using the command-line interface, specify the `-knob enable-stack-collection=true` option.

It is very important to know an effective way to collect call stacks. Developers that are not familiar with the concept try to obtain this information by using a debugger. They do so by interrupting the execution of a program and analyze the call stack (e.g. `backtrace` command in `gdb` debugger). Don't do this, let a profiling tool to do the job, which is much faster and gives much more accurate data.

5.5 Roofline Performance Model

The Roofline performance model is a throughput-oriented performance model that is heavily used in the HPC world. It was developed at the University of California, Berkeley, in 2009. The "roofline" in this model expresses the fact that the performance of an application cannot exceed the machine's capabilities. Every function and every loop in a program is limited by either compute or memory capacity of a machine. This concept is represented in Figure 32. The performance of an application will always be limited by a certain "roofline" function.

Hardware has two main limitations: how fast it can make calculations (peak compute performance, FLOPS) and how fast it can move the data (peak memory bandwidth, GB/s). The maximum performance of an application is limited by the minimum between peak FLOPS (horizontal line) and the platform bandwidth multiplied by arithmetic intensity (diagonal line). The roofline chart in Figure 32 plots the performance of two applications `A` and `B` against hardware limitations. Application `A` has lower arithmetic intensity and its performance is bound by the memory bandwidth, while application `B` is more compute intensive and doesn't suffer as much from memory bottlenecks.Similar to this, `A` and `B` could represent two different functions within a program and have different performance characteristics. The Roofline performance model accounts for that and can display multiple functions and loops of an application on the same chart.

Arithmetic Intensity (AI) is a ratio between FLOPS and bytes and can be extracted for every loop in a program. Let's calculate the arithmetic intensity of code in Listing 10. In the innermost loop body, we have an addition and a multiplication; thus, we have 2 FLOP. Also, we have three read operations and one write operation; thus, we transfer `4 ops * 4 bytes = 16` bytes. Arithmetic intensity of that code is `2 / 16 = 0.125`. AI serves as the value on the X-axis of a given performance point.

Traditional ways to speed up an application's performance is to fully utilize the SIMD and multicore capabilities of a machine. Often times, we need to optimize for many aspects: vectorization, memory, threading. Roofline methodology can assist in assessing these characteristics of your application. On a roofline chart, we can plot theoretical maximums for scalar single-core, SIMD single-core, and SIMD multicore performance (see Figure 33). This will give us an understanding of the room for improving the performance of an application. If we found that our application is compute-bound (i.e. has high arithmetic intensity) and is below the peak scalar single-core performance, we should consider forcing vectorization (see Section 9.4) and distributing the work among multiple

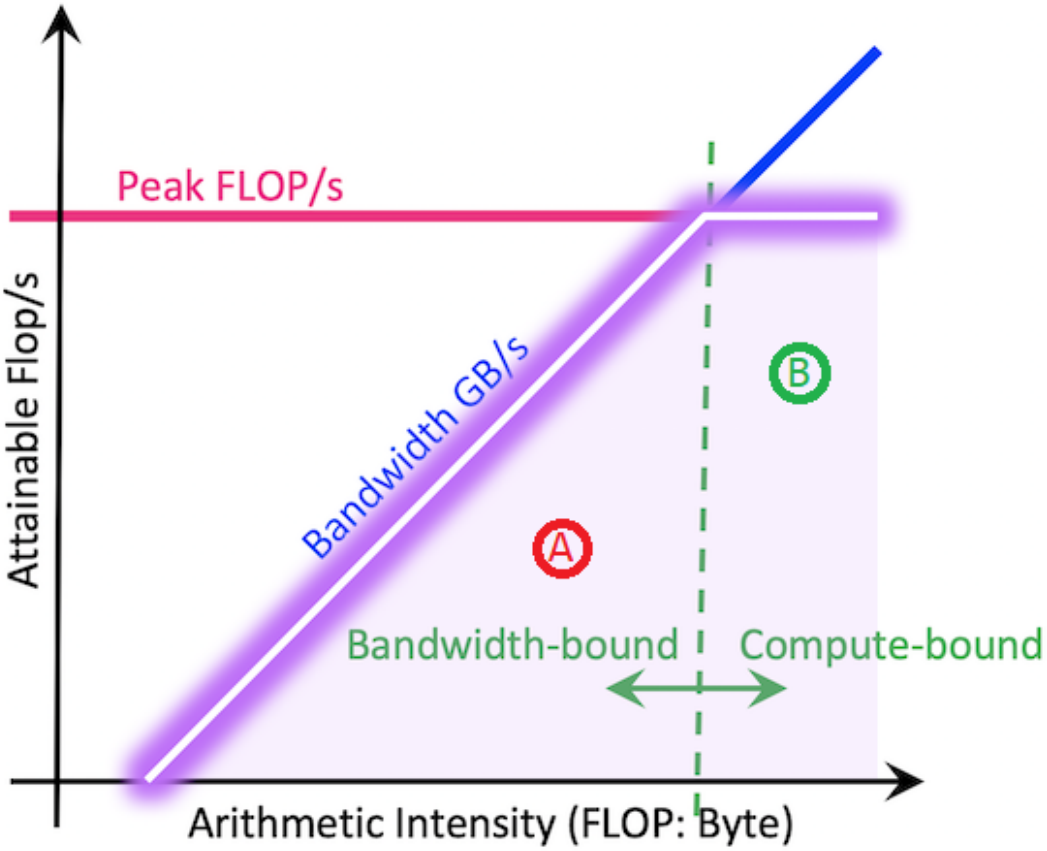


Figure 32: Roofline model. © *Image taken from NERSC Documentation.*

threads. Conversely, if an application has low arithmetic intensity, we should seek ways to improve memory accesses (see Chapter 8). The ultimate goal of optimizing performance using the Roofline model is to move the points up. Vectorization and threading move the dot up while optimizing memory accesses by increasing arithmetic intensity will move the dot to the right and also likely improve performance.

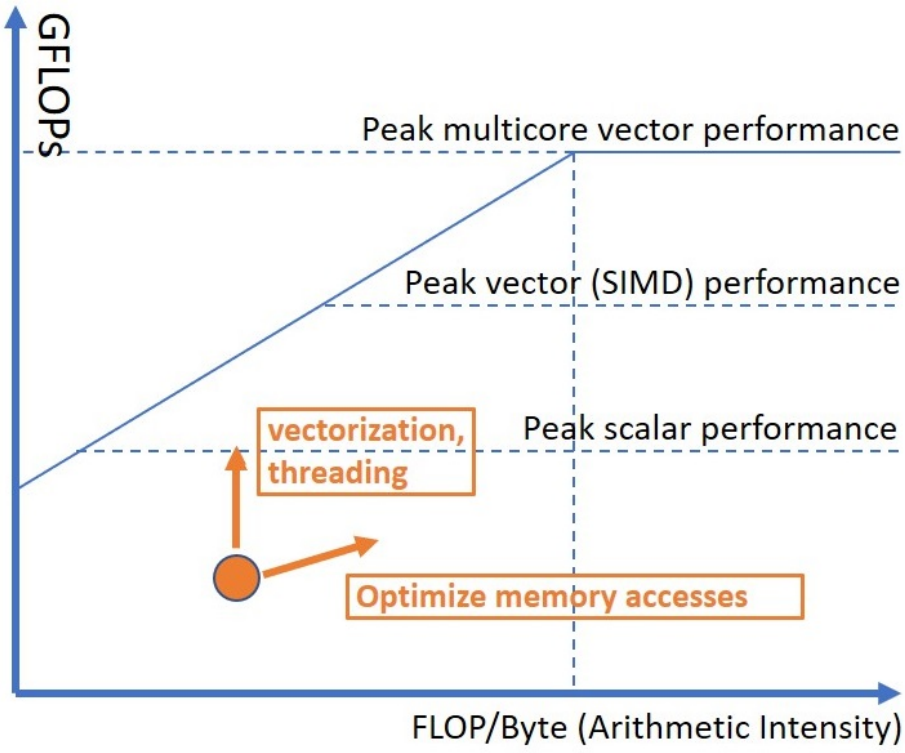


Figure 33: Roofline model.

Theoretical maximums (roof lines) are often presented in a device specification and can be easily looked up. Also, theoretical maximums can be calculated based on the characteristics of the machine you are using. Usually, it is not hard to do once you know the parameters of your machine. For Intel Core i5-8259U processor, the maximum number of FLOPs (single-precision floats) with AVX2 and 2 Fused Multiply Add (FMA) units can be calculated as:

$$\begin{aligned}\text{Peak FLOPS} = 8\ \text{(number of logical cores)} \times \frac{256\ \text{(AVX bit width)}}{32\ \text{bit (size of float)}} \times \\ 2\ \text{(FMA)} \times\ 3.8\ \text{GHz (Max Turbo Frequency)} \\ = 486.4\ \text{GFLOPs}\end{aligned}$$

The maximum memory bandwidth of Intel NUC Kit NUC8i5BEH, which I used for experiments, can be calculated as shown below. Remember, that DDR technology allows transfers of 64 bits or 8 bytes per memory access.

$$\begin{aligned}\text{Peak Memory Bandwidth} = 2400\ \text{(DDR4 memory transfer rate)} \times\ 2\ \text{(memory channels)} \times \\ 8\ \text{(bytes per memory access)} \times 1\ \text{(socket)} = 38.4\ \text{GiB/s}\end{aligned}$$

Automated tools like Empirical Roofline Tool[73]and Intel Advisor[74] are capable of empirically determining theoretical maximums by running a set of prepared benchmarks. If a calculation can reuse the data in cache, much higher FLOP rates are possible. Roofline can account for that by introducing a dedicated roofline for each level of the memory hierarchy (see Figure 34).

After hardware limitations are determined, we can start assessing the performance of an application against the roofline. The two most frequently used methods for automated collection of Roofline data are sampling (used by

[73] Empirical Roofline Tool - https://bitbucket.org/berkeleylab/cs-roofline-toolkit/src/master/.
[74] Intel Advisor - https://software.intel.com/content/www/us/en/develop/tools/advisor.html.

likwid[75]tool) and binary instrumentation (used by Intel Software Development Emulator (SDE[76])). Sampling incurs the lower overhead of collecting data, while binary instrumentation gives more accurate results.[77] Intel Advisor automatically builds a Roofline chart and provides hints for performance optimization of a given loop. An example of a Roofline chart generated by Intel Advisor is presented in Figure 34. Notice, Roofline charts have logarithmic scales.

Roofline methodology enables tracking optimization progress by printing "before" and "after" points on the same chart. So, it is an iterative process that guides developers to help their applications to fully utilize HW capabilities. Figure 34 shows performance gains from making the following two changes to the code shown earlier in Listing 10:

- Interchange the two innermost loops (swap lines 4 and 5). This enables cache-friendly memory accesses (see Chapter 8).
- Enable autovectorization of the innermost loop using AVX2 instructions.

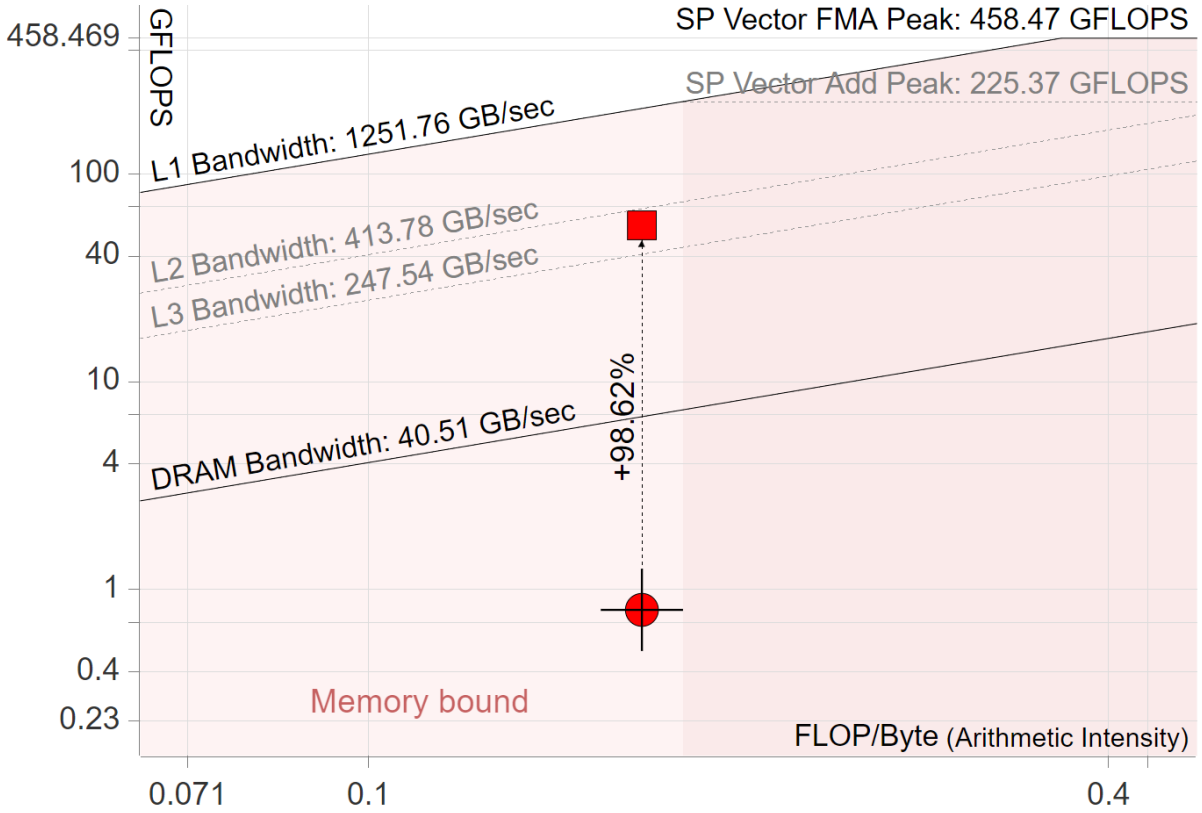


Figure 34: Roofline analysis for matrix multiplication on Intel NUC Kit NUC8i5BEH with 8GB RAM using clang 10 compiler.

In summary, the Roofline performance model can help to:

- Identify performance bottlenecks.
- Guide software optimizations.
- Determine when we're done optimizing.
- Assess performance relative to machine capabilities.

Additional resources and links:

- NERSC Documentation, URL: https://docs.nersc.gov/development/performance-debugging-tools/roofline/.
- Lawrence Berkeley National Laboratory research, URL: https://crd.lbl.gov/departments/computer-science/par/research/roofline/

[75] Likwid - https://github.com/RRZE-HPC/likwid.
[76] Intel SDE - https://software.intel.com/content/www/us/en/develop/articles/intel-software-development-emulator.html.
[77] See a more detailed comparison between methods of collecting roofline data in this presentation: https://crd.lbl.gov/assets/Uploads/ECP20-Roofline-4-cpu.pdf.

- Collection of video presentations about Roofline model and Intel Advisor, URL: https://techdecoded.intel.io/ (search "Roofline").
- `Perfplot` is a collection of scripts and tools that enable a user to instrument performance counters on a recent Intel platform, measure them, and use the results to generate roofline and performance plots. URL: https://github.com/GeorgOfenbeck/perfplot

5.6 Static Performance Analysis

Today we have extensive tooling for static code analysis. For C and C++ languages we have such well known tools like Clang Static Analyzer, Klocwork, Cppcheck and others. They aim at checking the correctness and semantics of the code. Likewise, there are tools that try to address the performance aspect of code. Static performance analyzers don't execute or profile the program. Instead, they simulate the code as if it is executed on a real HW. Statically predicting performance is almost impossible, so there are many limitations to this type of analysis.

First, it is not possible to statically analyze C/C++ code for performance since we don't know the machine code to which it will be compiled. So, static performance analysis works on assembly code.

Second, static analysis tools simulate the workload instead of executing it. It is obviously very slow, so it's not possible to statically analyze the entire program. Instead, tools take a snippet of assembly code and try to predict how it will behave on real hardware. The user should pick specific assembly instructions (usually small loop) for analysis. So, the scope of static performance analysis is very narrow.

The output of static performance analyzers is fairly low-level and sometimes breaks execution down to CPU cycles. Usually, developers use it for fine-grained tuning of a critical code region in which every CPU cycle matters.

Static vs. Dynamic Analyzers

Static tools: don't run the actual code but try to simulate the execution, keeping as many microarchitectural details as they can. They are not capable of doing real measurements (execution time, performance counters) because they don't run the code. The upside here is that you don't need to have the real HW and can simulate the code for different CPU generations. Another benefit is that you don't need to worry about consistency of the results: static analyzers will always give you deterministic output because simulation (in comparison with the execution on real hardware) is not biased in any way. The downside of static tools is that they usually can't predict and simulate everything inside a modern CPU: they are based on a model that may have bugs and limitations. Examples of static performance analyzers are UICA[78] and llvm-mca[79].

Dynamic tools: are based on running the code on the real HW and collecting all sorts of information about the execution. This is the only 100% reliable method of proving any performance hypothesis. As a downside, usually, you are required to have privileged access rights to collect low-level performance data like PMCs. It's not always easy to write a good benchmark and measure what you want to measure. Finally, you need to filter the noise and different kinds of side effects. Examples of dynamic microarchitectural performance analyzers are nanoBench,[80] uarch-bench[81] and a few others.

A bigger collection of tools both for static and dynamic microarchitectural performance analysis is available here[82].

5.6.1 Case Study: Using UICA to Optimize FMA Throughput

One of the questions developers often ask is: "Latest processors have 10+ execution units; How do I write my code to keep them busy all the time?" This is indeed one the hardest questions to tackle. Sometimes it requires looking under the microscope at how the program is running. One of such microscopes is UICA simulator that allows you to have insights into how your code could be flowing through a modern processor.

Let's look at the code in Listing 11. We intentionally try to make the examples as simple as possible. Though real-world codes are of course usually more complicated than this. The code scales every element of array `a` by the constant `B` and accumulates scaled values into `sum`. On the right, we present the machine code for the loop generated

[78] UICA - https://uica.uops.info/
[79] LLVM MCA - https://llvm.org/docs/CommandGuide/llvm-mca.html
[80] nanoBench - https://github.com/andreas-abel/nanoBench
[81] uarch-bench - https://github.com/travisdowns/uarch-bench
[82] Collection of links for C++ performance tools - https://github.com/MattPD/cpplinks/blob/master/performance.tools.md#microarchitecture.

by Clang-16 when compiled with `-O3 -ffast-math -march=core-avx2`. The assembly code look very compact, let's understand it better.

This is a reduction loop, i.e. we need to sum up all the products and in the end return a single float value. The way this code is written, there is a loop-carry dependency over `sum`. You cannot overwrite `sum` until you accumulate the previous product. A smart way to parallelize this is to have multiple accumulators and roll them up in the end. So, instead a single `sum`, we could have `sum1` to accumulate results from even iterations and `sum2` from odd iterations. This is what Clang-16 has done: it has 4 vectors (`ymm2`-`ymm5`) each holding 8 floating point accumulators, plus it used FMA to fuse multiplcation and addition into single instruction. The constant `B` is broadcasted into the `ymm1` register. The `-ffast-math` option allows a compiler to reassociate floating point operations, we will discuss it later in the book. By the way, the multiplication can be done only once after the loop. Definitely an oversight by the programmer, but hopefully compilers will be able to handle it in the future.

The code looks good, but is it really optimal? Let's find out. We took the assembly snippet from Listing 11 to UICA and run simulations. At the time of writing, Alderlake (Intel's 12th gen, based on GoldenCove) is not supported by UICA, so we ran it on the latest available, which is RocketLake (Intel's 11th gen, based on SunnyCove). Although the architectures differ, the issue exposed by this experiment is equally visible on both. The result of simulation is shown on Figure 35. This is a pipeline diagram similar to what we have shown in Chapter 3. We skipped the first two iterations, and show only iterations 2 and 3 (leftmost column "It."). This is when the execution reaches a steady state, all further iterations look very similar.

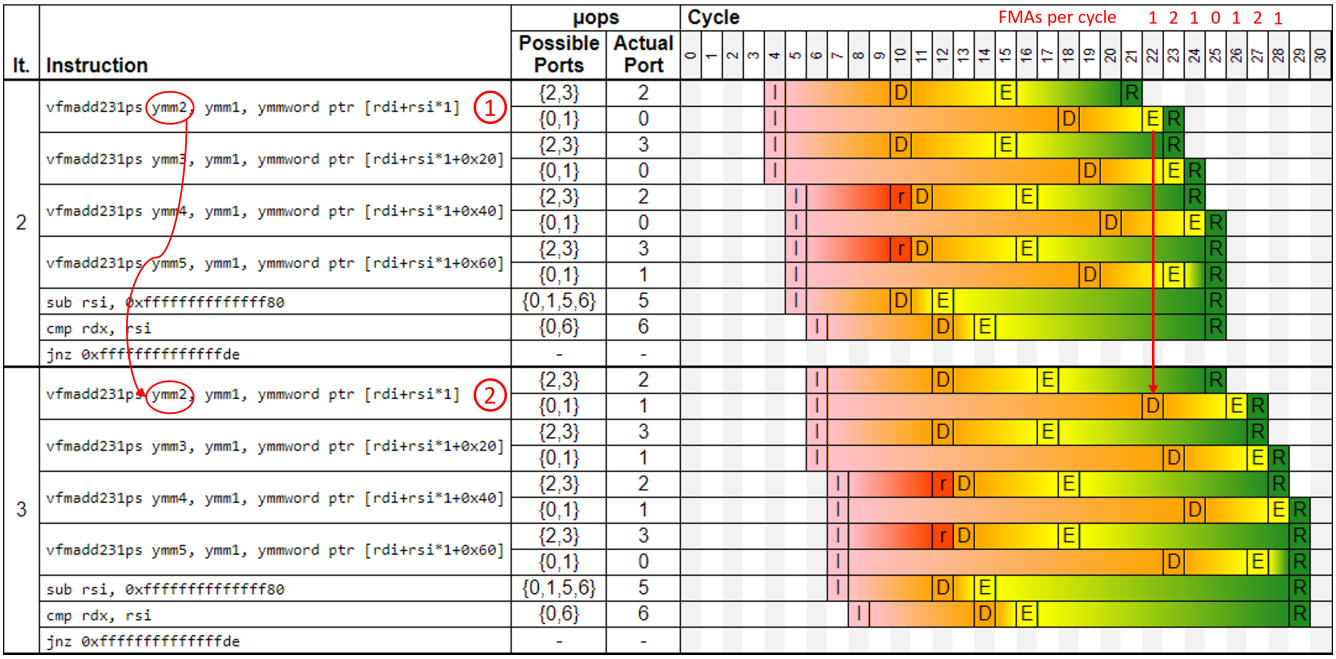


Figure 35: UICA pipeline diagram. `I` = issued, `r` = ready for dispatch, `D` = dispatched, `E` = executed, `R` = retired.

UICA is a very simplified model of the actual CPU pipeline. For example, you may notice that instruction fetch and decode stages are missing. Also, UICA doesn't account for cache misses and branch mispredictions, so it assumes that all memory accesses always hit in L1 cache and branches are always predicted correctly. Which we know is not the case in modern processors. Again, this is irrelevant for our experiment as we could still use the simulation results to find a way to improve the code. Can you see the issue?

Let's examine the diagram. First of all, every `FMA` instruction is broken into two uops (see (1)): a load uop that goes to ports `{2,3}` and an FMA uop that can go to ports `{0,1}`. The load uop has the latency of 5 cycles: starts at cycle 11 and finished at cycle 15. The FMA uop has the latency of 4 cycles: starts at cycle 19 and finishes at cycle 22. All FMA uops depend on load uops, we can clearly see this on the diagram: FMA uops always start after the corresponding load uop finishes. Now find two `r` cells at cycle 10, they are ready to be dispatched, but RocketLake has only two load ports, and both are already occupied in the same cycle. So, these two loads are issued in the next cycle.

The loop has four cross-iteration dependencies over `ymm2-ymm5`. The FMA uop from instruction (2) that writes into `ymm2` cannot start execution before instruction (1) from previous iteration finishes. Notice that the FMA uop from instruction (2) was dispatched right in the same cycle 22 as instruction (1) finished its execution. You can observe this pattern for other FMA instructions as well.

So, "what is the problem?", you ask. Look at the top right corner of the image. For each cycle, we added the number of executed FMA uops, this is not printed by UICA. It goes like `1,2,1,0,1,2,1,...`, or an average of one FMA uop per cycle. Most of the recent Intel processors have two FMA execution units, thus can issue two FMA uops per cycle. The diagram clearly shows the gap as every forth cycle there are no FMAs executed. As we figured out before, no FMA uops can be dispatched because their inputs (`ymm2-ymm5`) are not ready.

To increase the utilization of FMA execution units from 50% to 100%, we need to unroll the loop by a factor of two. This will double the number of accumulators from 4 to 8. Also, instead of 4 independent data flow chains, we would have 8. We will not shown the simulations of unrolled version here, you can experiment on your own. Instead, let us confirm the hypothesis by running two version on a real HW. By the way, this is always a good idea to verify since static performance analyzers like UICA are not accurate models. Below, we show the output of two nanobench tests that we ran on a recent Alderlake processor. The tool takes provided assembly instructions (`-asm` options) and creates a benchmark kernel. Readers can look up the meaning of other parameters in the nanobench documentation. The original code on the left executes 4 instructions in 4 cycles, while improved version can execute 8 instructions in 4 cycles. Now we can be sure we maximized the FMA execution throughput, the code on the right keeps the FMA units busy all the time.

```
# ran on Intel Core i7-1260P (Alderlake)
$ sudo ./kernel-nanoBench.sh -f -unroll 10
 -loop 100 -basic -warm_up_count 10 -asm "
VFMADD231PS YMM0, YMM1, ymmword [R14];
VFMADD231PS YMM2, YMM1, ymmword [R14+32];
VFMADD231PS YMM3, YMM1, ymmword [R14+64];
VFMADD231PS YMM4, YMM1, ymmword [R14+96];"
-asm_init "<not shown>"

Instructions retired: 4.20
Core cycles: 4.02
```

```
$ sudo ./kernel-nanoBench.sh -f -unroll 10
 -loop 100 -basic -warm_up_count 10 -asm "
VFMADD231PS YMM0, YMM1, ymmword [R14];
VFMADD231PS YMM2, YMM1, ymmword [R14+32];
VFMADD231PS YMM3, YMM1, ymmword [R14+64];
VFMADD231PS YMM4, YMM1, ymmword [R14+96];
VFMADD231PS YMM5, YMM1, ymmword [R14+128];
VFMADD231PS YMM6, YMM1, ymmword [R14+160];
VFMADD231PS YMM7, YMM1, ymmword [R14+192];
VFMADD231PS YMM8, YMM1, ymmword [R14+224]"
-asm_init "<not shown>"

Instructions retired: 8.20
Core cycles: 4.02
```

As a rull of thumb, in such situations, the loop must be unrolled by a factor `T * L`, where `T` is the throughput of an instruction, and `L` is its latency. In our case, we should have unrolled it by `2 * 4 = 8` to achieve maximum FMA port utilization since the throughput of FMA on Alderlake is 2 and latency of FMA is 4 cycles. This creates 8 separate data flow chains that can be executed independently.

It's worth to mention that you will not always see 2x speedup in practice. It can only be achieved in an idealized environment like UICA or nanobench. In a real application, even though you maximized the execution throughput of FMA, the gains may be hindered by eventual cache misses and other pipeline hazards. When that happens, the effect of cache misses outweights the effect of suboptimal FMA port utilization. Which could easily result in much more disappointing 5% speedup. But don't worry you've still done the right thing.

As a closing thought, let us remind you that UICA or any other static performance analyzer is not suitable for analyzing large portions of code. But they are great for exploring microarchitectural effects. Also, they help you building up a mental model of how CPU works. Another very important use case for UICA is to find critical dependency chains in a loop as described in the post[83] on the easyperf blog.

5.7 Compiler Optimization Reports

Nowadays, software development relies very much on compilers to do performance optimizations. Compilers play a critical role in speeding up software. Majority of developers leave the job of optimizing code to compilers, interfering

[83] Easyperf blog - https://easyperf.net/blog/2022/05/11/Visualizing-Performance-Critical-Dependency-Chains

only when they see an opportunity to improve something compilers cannot accomplish. Fair to say, this is a good default strategy. But it doesn't work well when you're looking for the best performance possible. What if compiler failed to perform a critical optimization like vectorizing a loop? How you would know about this? Luckily, all major compilers provide optimization reports which we will discuss now.

Suppose you want to know if a critical loop was unrolled or not. If it was unrolled, what is the unroll factor? There is a hard way to know this: by studying generated assembly instructions. Unfortunately, not all people are comfortable at reading assembly language. This can be especially difficult if the function is big, it calls other functions or has many loops that were also vectorized, or if the compiler created multiple versions of the same loop. Most compilers, including `GCC`, `Clang`, and Intel compiler, (not `MSVC`) provide optimization reports to check what optimizations were done for a particular piece of code.

Let's take a look at Listing 12 that shows an example of a loop that is not vectorized by `clang 16.0`.

To emit an optimization report in clang, you need to use -Rpass* flags:

```
$ clang -O3 -Rpass-analysis=.* -Rpass=.* -Rpass-missed=.* a.c -c
a.c:5:3: remark: loop not vectorized [-Rpass-missed=loop-vectorize]
  for (unsigned i = 1; i < N; i++) {
  ^
a.c:5:3: remark: unrolled loop by a factor of 8 with run-time trip count [-Rpass=loop-unroll]
  for (unsigned i = 1; i < N; i++) {
  ^
```

By checking the optimization report above, we could see that the loop was not vectorized, but it was unrolled instead. It's not always easy for a developer to recognize the existence of a loop-carry dependency in the loop on line 6 in Listing 12. The value that is loaded by `c[i-1]` depends on the store from the previous iteration (see operations ② and ③ in Figure 36). The dependency can be revealed by manually unrolling the first few iterations of the loop:

```
// iteration 1
  a[1] = c[0];
  c[1] = b[1]; // writing the value to c[1]
// iteration 2
  a[2] = c[1]; // reading the value of c[1]
  c[2] = b[2];
...
```

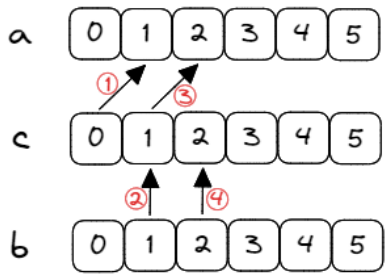


Figure 36: Visualizing the order of operations in Listing 12.

If we were to vectorize the code in Listing 12, it would result in the wrong values written in the array `a`. Assuming a CPU SIMD unit can process four floats at a time, we would get the code that can be expressed with the following pseudocode:

```
// iteration 1
  a[1..4] = c[0..3]; // oops!, a[2..4] get wrong values
  c[1..4] = b[1..4];
...
```

The code in Listing 12 cannot be vectorized because the order of operations inside the loop matters. This example can be fixed by swapping lines 6 and 7 as shown in Listing 13. This does not change the semantics of the code, so it's a perfectly legal change. Alternatively, the code can be improved by splitting the loop into two separate loops.

In the optimization report, we can now see that the loop was vectorized successfully:

```
$ clang -O3 -Rpass-analysis=.* -Rpass=.* -Rpass-missed=.* a.c -c
a.cpp:5:3: remark: vectorized loop (vectorization width: 8, interleaved count: 4)
    [-Rpass=loop-vectorize]
  for (unsigned i = 1; i < N; i++) {
  ^
```

This was just one example of using optimization reports, we have more in the second part of the book when we will discuss discovering vectorization opportunities. Compiler optimization reports can help you find missed optimization opportunities, and understand why those opportunities were missed. In addition, compiler optimization reports are useful for testing a hypothesis. Compilers often decides whether a certain transformation will be beneficial based on their cost model analysis. But compilers don't always make the optimal choice. Once you detect a key missing optimization in the report, you can attempt to rectify it by changing the source code or by providing hints to the compiler in the form of, say, a `#pragma`, an attribute, a compiler built-in, etc. As always, verify your hypothesis by measuring it in a practical environment.

Compiler reports can be quite large, and a separate report is generated for each source-code file. Sometimes, finding relevant records in the output file can become a challenge. We should mention that initially these reports were explicitly designed for use by compiler writers to improve optimization passes. Over the years there has been a number of tools that made them more accessible and actionable by application developers. Most notably, opt-viewer[84] and optview2[85]. Also, the Compiler Explorer website has the "Optimization Output" tool for the LLVM-based compilers that reports performed transformations when you hover your mouse over the corresponding line of source code. All of these tools help visualizing successful and failed code transformations by the LLVM-based compilers.

In LTO[86] mode, some optimizations are made during linking stage. To emit compiler reports from both compilation and linking stages, one should pass dedicated options to both the compiler and the linker. See LLVM "Remarks" guide[87] for more information.

A slightly different way of reporting missing optimizations is taken by Intel® ISPC[88] compiler (discussed in Section 9.4.2.5). It issues warnings for code constructs that compile to relatively inefficient code. Either way, compiler optimization reports should be one of the key tools in your toolbox. It is a fast way to check what optimizations were done for a particular hotspot and see if some important ones failed. Many improvement opportunities were found thanks to compiler optimization reports.

Questions and Exercises

1. Which approaches you would use in the following scenarios?

- scenario 1: client support team reports a customer issue: after upgrading to a new version of the application, performance of a certain operation drops by 10%.
- scenario 2: client support team reports a customer issue: some transactions take 2x longer time to finish than usual with no particular pattern.
- scenario 3: you're evaluating three different compression algorithms and you want to know what types of performance bottlenecks (memory latency/bandwidth, branch mispredictions, etc) each of them has.
- scenario 4: there is a new shiny library that claims to be faster than the one you currently have integrated in your project; you've decided to compare their performance.
- scenario 5: you were asked to analyze performance of an unfamiliar code; you want to know how frequently a certain branch is taken and how many iterations the loop is doing.

[84] opt-viewer - https://github.com/llvm/llvm-project/tree/main/llvm/tools/opt-viewer
[85] optview2 - https://github.com/OfekShilon/optview2
[86] Link-Time optimizations, also called InterProcedural Optimizations (IPO). Read more here: https://en.wikipedia.org/wiki/Interprocedural_optimization
[87] LLVM compiler remarks - https://llvm.org/docs/Remarks.html
[88] ISPC - https://ispc.github.io/ispc.html

2. Run the application that you're working with on a daily basis. Practice doing performance analysis using approaches we discussed in this chapter. Collect raw counts for various CPU performance events, find hotspots, collect roofline data, generate and study the compiler optimization report fot the hot function(s) in your program.

Chapter Summary

- Latency and throughput are often the ultimate metrics of the program performance. When seeking ways to improve them, we need to get more detailed information on how the application executes. Both HW and SW provide data that can be used for performance monitoring.
- Code instrumentation allows us to track many things in the program but causes relatively large overhead both on the development and runtime side. While developers do not manually instrument their code these days very often, this approach is still relevant for automated processes, e.g., PGO.
- Tracing is conceptually similar to instrumentation and is useful for exploring anomalies in the system. Tracing allows us to catch the entire sequence of events with timestamps attached to each event.
- Workload Characterization is a way to compare and group applications based on their runtime behavior. Once characterized, specific recipes could be followed to find optimization headrooms in the program. Profiling tools with marker APIs are useful for analyzing performance of a specific code region.
- Sampling skips the large portion of the program execution and take just one sample that is supposed to represent the entire interval. Despite this, sampling usually yields precise enough distributions. The most well-known use case of sampling is finding hotspots in the code. Sampling is the most popular analysis approach since it doesn't require recompilation of the program and has very little runtime overhead.
- Generally, counting and sampling incur very low runtime overhead (usually below 2%). Counting gets more expensive once you start multiplexing between different events (5-15% overhead), sampling gets more expensive with increasing sampling frequency [Nowak & Bitzes, 2014]. Consider using user-mode sampling for analyzing long-running workloads or when you don't need very accurate data.
- The Roofline performance model is a throughput-oriented performance model that is heavily used in the High Performance Computing (HPC) world. It allows plotting the performance of an application against hardware limitations. Roofline model helps to identify performance bottlenecks, guides software optimizations, and keeps track of optimization progress.
- There are tools that try to statically analyze the performance of code. Such tools simulate a piece of code instead of executing it. Many limitations and constraints apply to this approach, but you get a very detailed and low-level report in return.
- Compiler Optimization reports help to find missing compiler optimizations. These reports also guide developers in composing new performance experiments.

Listing 9 Using libpfm4 marker API on the C-Ray benchmark

```
+#include <perfmon/pfmlib.h>
+#include <perfmon/pfmlib_perf_event.h>
...
/* render a frame of xsz/ysz dimensions into the provided framebuffer */
void render(int xsz, int ysz, uint32_t *fb, int samples) {
   ...
+  pfm_initialize();
+  struct perf_event_attr perf_attr;
+  memset(&perf_attr, 0, sizeof(perf_attr));
+  perf_attr.size = sizeof(struct perf_event_attr);
+  perf_attr.read_format = PERF_FORMAT_TOTAL_TIME_ENABLED |
+                          PERF_FORMAT_TOTAL_TIME_RUNNING | PERF_FORMAT_GROUP;
+
+  pfm_perf_encode_arg_t arg;
+  memset(&arg, 0, sizeof(pfm_perf_encode_arg_t));
+  arg.size = sizeof(pfm_perf_encode_arg_t);
+  arg.attr = &perf_attr;
+
+  pfm_get_os_event_encoding("instructions", PFM_PLM3, PFM_OS_PERF_EVENT_EXT, &arg);
+  int leader_fd = perf_event_open(&perf_attr, 0, -1, -1, 0);
+  pfm_get_os_event_encoding("cycles", PFM_PLM3, PFM_OS_PERF_EVENT_EXT, &arg);
+  int event_fd = perf_event_open(&perf_attr, 0, -1, leader_fd, 0);
+  pfm_get_os_event_encoding("branches", PFM_PLM3, PFM_OS_PERF_EVENT_EXT, &arg);
+  event_fd = perf_event_open(&perf_attr, 0, -1, leader_fd, 0);
+  pfm_get_os_event_encoding("branch-misses", PFM_PLM3, PFM_OS_PERF_EVENT_EXT, &arg);
+  event_fd = perf_event_open(&perf_attr, 0, -1, leader_fd, 0);
+
+  struct read_format { uint64_t nr, time_enabled, time_running, values[4]; };
+  struct read_format before, after;

  for(j=0; j<ysz; j++) {
    for(i=0; i<xsz; i++) {
      double r = 0.0, g = 0.0, b = 0.0;
+     // capture counters before ray tracing
+     read(event_fd, &before, sizeof(struct read_format));

      for(s=0; s<samples; s++) {
        struct vec3 col = trace(get_primary_ray(i, j, s), 0);
        r += col.x;
        g += col.y;
        b += col.z;
      }
+     // capture counters after ray tracing
+     read(event_fd, &after, sizeof(struct read_format));

+     // save deltas in separate arrays
+     nanosecs[j * xsz + i] = after.time_running - before.time_running;
+     instrs  [j * xsz + i] = after.values[0] - before.values[0];
+     cycles  [j * xsz + i] = after.values[1] - before.values[1];
+     branches[j * xsz + i] = after.values[2] - before.values[2];
+     br_misps[j * xsz + i] = after.values[3] - before.values[3];

      *fb++ = ((uint32_t)(MIN(r * rcp_samples, 1.0) * 255.0) & 0xff) << RSHIFT |
              ((uint32_t)(MIN(g * rcp_samples, 1.0) * 255.0) & 0xff) << GSHIFT |
              ((uint32_t)(MIN(b * rcp_samples, 1.0) * 255.0) & 0xff) << BSHIFT;
  } }
+ // aggregate statistics and print it
  ...
}
```

Listing 10 Naive parallel matrix multiplication.

```
void matmul(int N, float a[][2048], float b[][2048], float c[][2048]) {
    #pragma omp parallel for
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++) {
            for(int k = 0; k < N; k++) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
    }
}
```

Listing 11 FMA throughput

```
float foo(float * a, float B, int N){
  float sum = 0;
  for (int i = 0; i < N; i++)
    sum += a[i] * B;
  return sum;
}
```

```
.loop:
 vfmadd231ps ymm2, ymm1, ymmword [rdi + rsi]
 vfmadd231ps ymm3, ymm1, ymmword [rdi + rsi + 32]
 vfmadd231ps ymm4, ymm1, ymmword [rdi + rsi + 64]
 vfmadd231ps ymm5, ymm1, ymmword [rdi + rsi + 96]
 sub rsi, -128
 cmp rdx, rsi
 jne .loop
```

Listing 12 a.c

```
void foo(float* __restrict__ a,
         float* __restrict__ b,
         float* __restrict__ c,
         unsigned N) {
  for (unsigned i = 1; i < N; i++) {
    a[i] = c[i-1]; // value is carried over from previous iteration
    c[i] = b[i];
  }
}
```

Listing 13 a.c

```
void foo(float* __restrict__ a,
         float* __restrict__ b,
         float* __restrict__ c,
         unsigned N) {
  for (unsigned i = 1; i < N; i++) {
    c[i] = b[i];
    a[i] = c[i-1];
  }
}
```

6 CPU Features for Performance Analysis

The ultimate goal of performance analysis is to identify performance bottlenecks and locate parts of the code that are associated with them. Unfortunately, there are no predetermined steps to follow, so it can be approached in many different ways.

Usually, profiling an application can give quick insights about the hotspots of the application. Sometimes it is everything developers need to do to fix performance inefficiencies. Especially high-level performance problems can often be revealed by profiling. For example, consider a situation when you've just made a change to the function `foo` in your application and suddenly see a noticeable performance degradation. So, you decide to profile the application. According to your mental model of the application, you expect that `foo` is a cold function and it doesn't show up in the top-10 list of hot functions. But when you open the profile, you see it consumes a lot more time than before. You quickly realize the mistake you've made in the code and fix it. If all issues in performance engineering were that easy to fix, this book would not exist.

When you embark on a journey to squeeze the last bit of performance from your application, the most basic list of hotspots is not enough. Unless you have a crystall ball or an accurate model of an entire CPU in your head, you need additional support to understand what the performance bottlenecks are. However, before using the information presented in this chapter, make sure that the application you are trying to optimize does not suffer from major performance flaws. Because if it does, using CPU performance monitoring features for low-level tuning doesn't make sense. It will likely steer you in the wrong direction, and instead of fixing real high-level performance problems, you will be tuning bad code, which is just a waste of time.

Some developers rely on their intuition and proceed with random experiments, trying to force various compiler optimizations like loop unrolling, vectorization, inlining, you name it. Indeed, sometimes you can be lucky and enjoy a portion of compliments from your colleagues and maybe even claim an unofficial title of performance guru on your team. But usually, you need to have a very good intuition and luck. In this book we don't teach you how to be lucky. Instead, we show methods that have proved to be working in practice.

Modern CPUs are constantly getting new features that enhance performance analysis in different ways. Using those features greatly simplifies finding low-level issues like cache-misses, branch mispredictions, etc. In this chapter, we will take a look at a few HW performance monitoring capabilities available on modern CPUs. Processors from different vendors do not necessarily have the same set of features. In this chapter, we will focus on performance monitoring capabilities available in Intel, AMD, and ARM processors. RISC-V ecosystem does not yet have a mature performance monitoring infrastructure, so we will not cover it here.

- **Top-down Microarchitecture Analysis** (TMA) methodology, discussed in Section 6.1. This is a powerful technique for identifying ineffective usage of CPU microarchitecture by a program. It characterizes the bottleneck of a workload and allows locating the exact place in the source code where it occurs. It abstracts away intricacies of the CPU microarchitecture and is relatively easy to use even for inexperienced developers.
- **Last Branch Record** (LBR), discussed in Section 6.2. This is a mechanism that continuously logs the most recent branch outcomes in parallel with executing the program. It is used for collecting call stacks, identify hot branches, calculating misprediction rates of individual branches, and more.
- **Processor Event-Based Sampling** (PEBS), discussed in Section **??**. This is a feature that enhances sampling. Its primary benefits include: lowering the overhead of sampling; and providing "Precise Events" capability, that enables pinpointing of the exact instruction that caused a particular performance event.
- **Intel Processor Traces** (PT), discussed in Appendix D. It is a facility to record and reconstruct the program execution with a timestamp on *every* instruction. Its main usages are postmortem analysis and root-causing performance glitches.

The Intel PT feature is covered in Appendix D. Intel PT was supposed to be an "end game" for performance analysis. With its low runtime overhead, it is a very powerful analysis feature. But it turns out to be not very popular among performance engineers. Partially because the support in the tools is not mature, partially because in many cases it is an overkill, and it's just easier to use a sampling profiler. Also, it produces a lot of data, which is not practical for long-running workloads.

The features mentioned above provide insights on the efficiency of a program from the CPU perspective. In the next chapter we will discuss how profiling tools leverage them to provide many different types of performance analysis.

6.1 Top-down Microarchitecture Analysis

Top-down Microarchitecture Analysis (TMA) methodology is a very powerful technique for identifying CPU bottlenecks in the program. It is a robust and formal methodology that is easy to use even for inexperienced developers. The best part of this methodology is that it does not require a developer to have a deep understanding of the microarchitecture and PMCs in the system and still efficiently find CPU bottlenecks.

At a conceptual level, TMA identifies what was stalling the execution of a program. Figure 37 illustrates the core idea of TMA. This is not how the analysis works in practice, because analyzing every single microoperation (Uop) would be terribly slow. Nevertheless, the diagram is helpful for understanding the methodology.

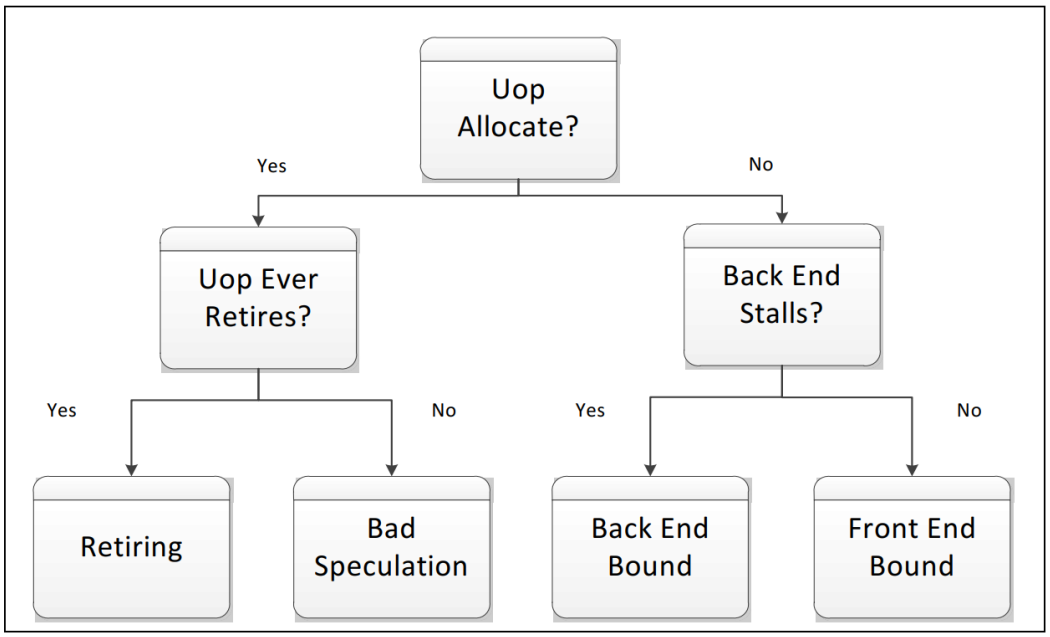


Figure 37: The concept behind TMA's top-level breakdown. © *Image from [Yasin, 2014]*

Here is a short guide on how to read this diagram. As we know from Chapter 3, there are internal buffers in the CPU that keep track of information about uops that are being executed. Whenever a new instruction is fetched and decoded, new entries in those buffers are allocated. If a uop for the instruction was not allocated during a particular cycle of execution, it could be for one of two reasons: either we were not able to fetch and decode it (`Front End Bound`); or the Back End was overloaded with work, and resources for the new uop could not be allocated (`Back End Bound`). If a Uop was allocated and scheduled for execution but never retired, this means it came from a mispredicted path (`Bad Speculation`). Finally, `Retiring` represents a normal execution. It is the bucket where we want all our uops to be, although there are exceptions which we will talk about later.

To accomplish its goal, TMA observes the execution of the program by monitoring specific set of performance events and then calculating metrics based on predefined formulas. Based on those metrics, TMA characterizes the program by assigning it to one of the four high-level buckets. Each of the four high-level categories has several nested levels, which CPU vendors may choose to implement differently. Each generation of processors may have different formulas for calculating those metrics, so it's better to rely on tools to do the analysis rather than trying to calculate them yourself.

In the upcoming sections, we will discuss the TMA implementation in AMD, ARM and Intel processors.

A high `Retiring` metric for non-vectorized code may be a good hint for users to vectorize the code (see Section 9.4). Another situation in which we might see a high Retiring value but slow overall performance is in a program that operates on denormalized floating-point values, thus making such operations extremely slow (see Section 12.4).

6.1.1 TMA on Intel Platforms

The TMA methodology was first proposed by Intel in 2014 and is supported starting from the SandyBridge family of processors. Intel's implementation supports nested categories for each high-level bucket that give a better

understanding of the CPU performance bottlenecks in the program (see Figure 38).

The workflow is designed to "drill down" to lower levels of the TMA hierarchy until we get to the very specific classification of a performance bottleneck. For example, at first, we collect metrics for the main four buckets: `Front End Bound`, `Back End Bound`, `Retiring`, `Bad Speculation`. Say, we found out that the big portion of the program execution was stalled by memory accesses (which is a `Back End Bound` bucket, see Figure 38). The next step is to run the workload again and collect metrics specific for the `Memory Bound` bucket only. The process is repeated until we know the exact root cause, for example, `L3 Bound`.

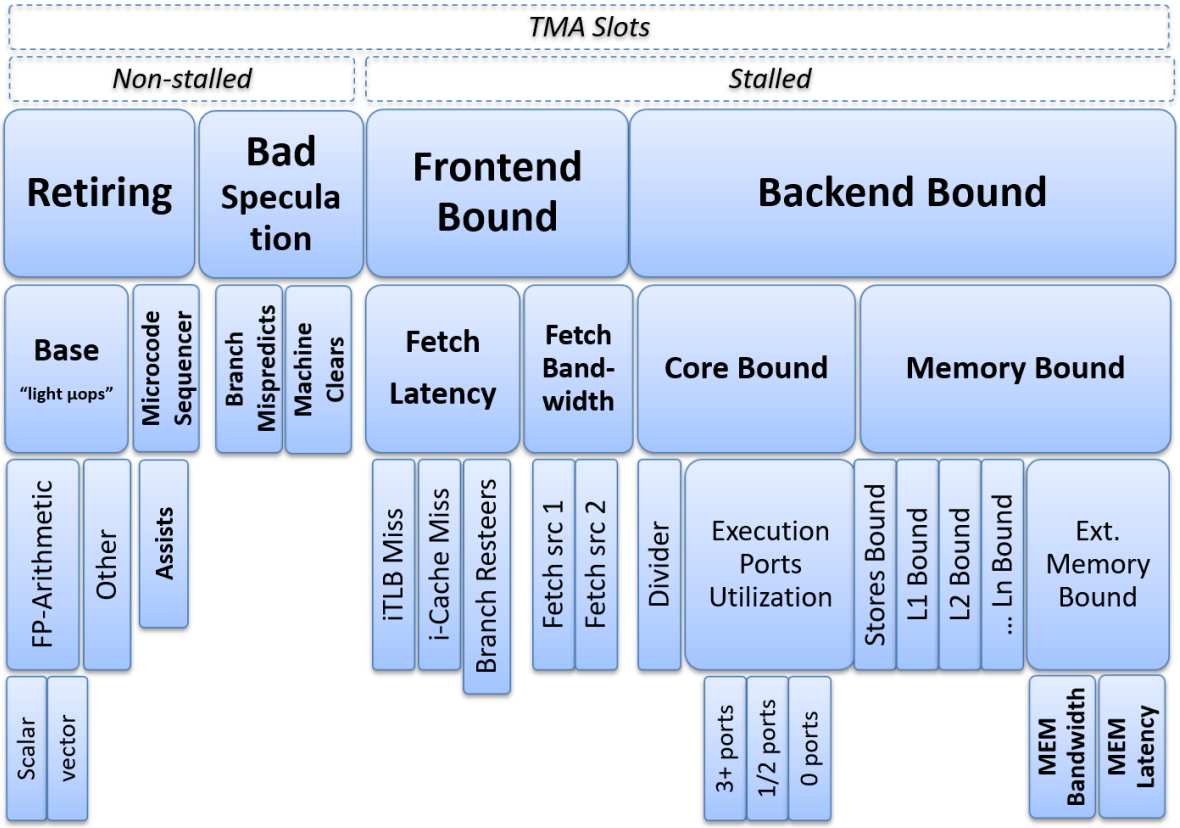


Figure 38: The TMA hierarchy of performance bottlenecks. © *Image by Ahmad Yasin.*

It is absolutely fine to run the workload several times, each time drilling down and focusing on specific metrics. But usually, it is sufficient to run the workload once and collect all the metrics required for all levels of TMA. Profiling tools achieve that by multiplexing between different performance events during a single run (see Section 5.3.3). Also, in a real-world application, performance could be limited by several factors. E.g., it can experience a large number of branch mispredicts (`Bad Speculation`) and cache misses (`Back End Bound`) at the same time. In this case, TMA will drill down into multiple buckets simultaneously and will identify the impact that each type of bottleneck makes on the performance of a program. Analysis tools such as Intel's VTune Profiler, AMD's uProf, and Linux `perf` can calculate all the TMA metrics with a single run of the benchmark. However, this only is acceptable if the workload is steady. Otherwise, you would better fall back to the original strategy of multiple runs and drilling down with each run.

The top two-levels of TMA metrics are expressed in the percentage of all pipeline slots (see Section 4.5) that were available during the execution of the program. It allows TMA to give an accurate representation of CPU microarchitecture utilization, taking into account the full bandwidth of the processor. Up to this point, everything should sum up nicely to 100%. However, starting from Level 3, buckets may be expressed in a different count domain, e.g. clocks and stalls. So they are not necessarily directly comparable with other TMA buckets.

The first step of TMA is to identify the performance bottleneck in the program. After we have done that, we need to know where exactly in the code it is happening. The second step in TMA is to locate the source of the problem down to the exact line of code and assembly instruction. The analysis methodology provides exact performance

event that you should use for each category of the performance problem. Then you can sample on this event to find the line in the source code that contributes to the performance bottleneck identified by the first stage. Don't worry if this process sounds complicated to you, everything becomes clear once you read through the case study.

Case Study: Reduce The Number of Cache Misses with TMA As an example for this case study, we took a very simple benchmark, such that it is easy to understand and change. It is obviously not representative of real-world applications, but it is good enough to demonstrate the workflow of TMA. We have a lot more practical examples in the second part of the book.

Most readers of this book will likely apply TMA to their own applications, with which they are familiar. But TMA is very effective even if you see the application for the first time. For this reason, we don't start with showing you the original source code of the benchmark. But here is a short description: the benchmark allocates a 200 MB array on the heap, then enters a loop of 100M iterations. On every iteration of the loop, it generates a random index into the allocated array, performs some dummy work and then reads the value from that index.

We ran the experiments on the machine equipped with Intel Core i5-8259U CPU (Skylake based) and 16GB of DRAM (DDR4 2400 MT/s), running 64-bit Ubuntu 20.04 (kernel version 5.13.0-27).

Step1: Identify the Bottleneck As a first step, we run our microbenchmark and collect a limited set of events that will help us to calculate Level 1 metrics. Here, we try to identify high-level performance bottlenecks of our application by attributing them to the four L1 buckets: `Front End Bound`, `Back End Bound`, `Retiring`, `Bad Speculation`. It is possible to collect Level 1 metrics using Linux `perf` tool. As of Linux kernel 4.8, `perf` has an option `--topdown` used in `perf stat` command that prints TMA Level 1 metrics. Below is the breakdown for our benchmark. Command outputs in this section are trimmed to save space.

```
$ perf stat --topdown -a -- taskset -c 0 ./benchmark.exe
       retiring  bad speculat  FE bound  BE bound
S0-C0    32.5%        0.2%       13.8%      53.4%  <==
S0-C1    17.4%        2.3%       12.0%      68.2%
S0-C2    10.1%        5.8%       32.5%      51.6%
S0-C3    47.3%        0.3%        2.9%      49.6%
...
```

To get values for high-level TMA metrics, Linux `perf` requires profiling the whole system (`-a`). This is why we see metrics for all cores. But since we have pinned the benchmark to core 0 with `taskset -c 0`, we need focus only on the row corresponding to `S0-C0`. We can discard other rows as they were running other tasks or remained idle. By looking at the output, we can tell that performance of the application is bound by the CPU backend. Without trying to analyze it right now, let us drill one level down.

Linux perf only supports Level 1 TMA metrics, so to get access to TMA metrics Level 2, 3, and further, we will use the `toplev` tool that is a part of pmu-tools[89] written by Andi Kleen. It is implemented in `Python` and invokes Linux `perf` under the hood. Specific Linux kernel settings must be enabled to use `toplev`, check the documentation for more details.

```
$ ~/pmu-tools/toplev.py --core S0-C0 -l2 -v --no-desc taskset -c 0 ./benchmark.exe
...
# Level 1
S0-C0  Frontend_Bound:                13.92 % Slots
S0-C0  Bad_Speculation:                0.23 % Slots
S0-C0  Backend_Bound:                 53.39 % Slots
S0-C0  Retiring:                      32.49 % Slots
# Level 2
S0-C0  Frontend_Bound.FE_Latency:     12.11 % Slots
S0-C0  Frontend_Bound.FE_Bandwidth:    1.84 % Slots
S0-C0  Bad_Speculation.Branch_Mispred: 0.22 % Slots
S0-C0  Bad_Speculation.Machine_Clears: 0.01 % Slots
S0-C0  Backend_Bound.Memory_Bound:    44.59 % Slots <==
```

[89] PMU tools - https://github.com/andikleen/pmu-tools.

```
S0-C0  Backend_Bound.Core_Bound:       8.80 % Slots
S0-C0  Retiring.Base:                 24.83 % Slots
S0-C0  Retiring.Microcode_Sequencer:   7.65 % Slots
```

In this command, we also pinned the process to CPU0 (using `taskset -c 0`), and limited the output of `toplev` to this core only (`--core S0-C0`). The option `-l2` tells the tool to collect Level 2 metrics. The option `--no-desc` disables the description of each metric.

We can see that the application's performance is bound by memory accesses (`Backend_Bound.Memory_Bound`). Almost half of the CPU execution resources were wasted waiting for memory requests to complete. Now let us dig one level deeper: [90]

```
$ ~/pmu-tools/toplev.py --core S0-C0 -l3 -v --no-desc taskset -c 0 ./benchmark.exe
...
# Level 1
S0-C0    Frontend_Bound:                 13.91 % Slots
S0-C0    Bad_Speculation:                 0.24 % Slots
S0-C0    Backend_Bound:                  53.36 % Slots
S0-C0    Retiring:                       32.41 % Slots
# Level 2
S0-C0    FE_Bound.FE_Latency:            12.10 % Slots
S0-C0    FE_Bound.FE_Bandwidth:           1.85 % Slots
S0-C0    BE_Bound.Memory_Bound:          44.58 % Slots
S0-C0    BE_Bound.Core_Bound:             8.78 % Slots
# Level 3
S0-C0-T0 BE_Bound.Mem_Bound.L1_Bound:     4.39 % Stalls
S0-C0-T0 BE_Bound.Mem_Bound.L2_Bound:     2.42 % Stalls
S0-C0-T0 BE_Bound.Mem_Bound.L3_Bound:     5.75 % Stalls
S0-C0-T0 BE_Bound.Mem_Bound.DRAM_Bound:  47.11 % Stalls <==
S0-C0-T0 BE_Bound.Mem_Bound.Store_Bound:  0.69 % Stalls
S0-C0-T0 BE_Bound.Core_Bound.Divider:     8.56 % Clocks
S0-C0-T0 BE_Bound.Core_Bound.Ports_Util: 11.31 % Clocks
```

We found the bottleneck to be in `DRAM_Bound`. This tells us that many memory accesses miss in all levels of caches and go all the way down to the main memory. We can also confirm this if we collect the absolute number of L3 cache misses for the program. For the Skylake architecture, the `DRAM_Bound` metric is calculated using the `CYCLE_ACTIVITY.STALLS_L3_MISS` performance event. Let's collect it manually:

```
$ perf stat -e cycles,cycle_activity.stalls_l3_miss -- ./benchmark.exe
  32226253316  cycles
  19764641315  cycle_activity.stalls_l3_miss
```

The `CYCLE_ACTIVITY.STALLS_L3_MISS` event counts cycles when execution stalls, while the L3 cache miss demand load is outstanding. We can see that there are ~60% of such cycles, which is pretty bad.

Step2: Locate the Place in the Code As the second step in the TMA process, we locate the place in the code where the identified performance event occurs most frequently. To do so, one should sample the workload using an event that corresponds to the type of bottleneck that was identified during Step 1.

A recommended way to find such an event is to run `toplev` tool with the `--show-sample` option that will suggest the `perf record` command line that can be used to locate the issue. For the purpose of understanding the mechanics of TMA, we also present the manual way to find an event associated with a particular performance bottleneck. Correspondence between performance bottlenecks and performance events that should be used for determining the location of bottlenecks in source code can be done with the help of the TMA metrics[91] table. The `Locate-with` column denotes a performance event that should be used to locate the exact place in the code where the issue occurs.

[90] Alternatively, we could use `-l2 --nodes L1_Bound,L2_Bound,L3_Bound,DRAM_Bound,Store_Bound` option instead of `-l3` to limit the collection since we know the application is bound by memory.

[91] TMA metrics - https://github.com/intel/perfmon/blob/main/TMA_Metrics.xlsx.

In our case, to find memory accesses that contribute to such a high value of the `DRAM_Bound` metric (miss in the L3 cache), we should sample on `MEM_LOAD_RETIRED.L3_MISS_PS` precise event. Here is the example command:

```
$ perf record -e cpu/event=0xd1,umask=0x20,name=MEM_LOAD_RETIRED.L3_MISS/ppp ./benchmark.exe

$ perf report -n --stdio
...
# Samples: 33K of event 'MEM_LOAD_RETIRED.'L3_MISS
# Event count (approx.): 71363893
# Overhead   Samples  Shared Object   Symbol
# ........  ........ ..............  ..................
#
    99.95%    33811   benchmark.exe   [.] foo
     0.03%       52   [kernel]        [k] get_page_from_freelist
     0.01%        3   [kernel]        [k] free_pages_prepare
     0.00%        1   [kernel]        [k] free_pcppages_bulk
```

Almost all L3 misses are caused by memory accesses in function `foo` inside executable `benchmark.exe`. Now it's time to look at the source code of the benchmark, which can be found on Github.[92]

To avoid compiler optimizations, function `foo` is implemented in assembly language, which is presented in Listing 14. The "driver" portion of the benchmark is implemented in the `main` function, as shown in Listing 15. We allocate a big enough array `a` to make it not fit in the 6MB L3 cache. The benchmark generates a random index into array `a` and passes this index to the `foo` function along with the address of array `a`. Later the `foo` function reads this random memory location.[93]

Listing 14 Assembly code of function foo.

```
$ perf annotate --stdio -M intel foo
Percent |  Disassembly of benchmark.exe for MEM_LOAD_RETIRED.L3_MISS
------------------------------------------------------------
        :  Disassembly of section .text:
        :
        :  0000000000400a00 <foo>:
        :  foo():
   0.00 :    400a00:  nop  DWORD PTR [rax+rax*1+0x0]
   0.00 :    400a08:  nop  DWORD PTR [rax+rax*1+0x0]
                ...
 100.00 :    400e07:  mov  rax,QWORD PTR [rdi+rsi*1] <==
                ...
   0.00 :    400e13:  xor  rax,rax
   0.00 :    400e16:  ret
```

By looking at Listing 14, we can see that all L3-Cache misses in function `foo` are tagged to a single instruction. Now that we know which instruction caused so many L3 misses, let's fix it.

Step3: Fix the Issue Remember that there is a dummy work emulated with NOPs in the beginning of the `foo` function. This creates a time window between the moment when we get the next address that will be accessed and the actual load instruction. The presence of this time windows gives us an opportunity to start prefetching the memory location in parallel with the dummy work. Listing 16 shows this idea in action. More information about explicit memory prefetching technique can be found in Section 8.2.

This explicit memory prefetching hint decreases execution time from 8.5 seconds to 6.5 seconds. Also, the number of `CYCLE_ACTIVITY.STALLS_L3_MISS` events becomes almost ten times less: it goes from 19B down to 2B.

[92] Case study example - https://github.com/dendibakh/dendibakh.github.io/tree/master/_posts/code/TMAM.
[93] According to x86 calling conventions (https://en.wikipedia.org/wiki/X86_calling_conventions), first 2 arguments land in `rdi` and `rsi` registers respectively.

Listing 15 Source code of function main.

```
extern "C" { void foo(char* a, int n); }
const int _200MB = 1024*1024*200;
int main() {
  char* a = (char*)malloc(_200MB); // 200 MB buffer
  ...
  for (int i = 0; i < 100000000; i++) {
    int random_int = distribution(generator);
    foo(a, random_int);
  }
  ...
}
```

Listing 16 Inserting memory prefetch into main.

```
  for (int i = 0; i < 100000000; i++) {
    int random_int = distribution(generator);
+   __builtin_prefetch ( a + random_int, 0, 1);
    foo(a, random_int);
  }
```

TMA is an iterative process, so once we fixed one problem, we need to repeat the process starting from the Step1. Likely it will move the bottleneck into another bucket, in this case, `Retiring`. This was an easy example demonstrating the workflow of TMA methodology. Analyzing real-world application is unlikely to be that easy. Chapters in the second part of the book are organized to make it convenient for use with the TMA process. In particular, Chapters 8 covers `Memory Bound` category, Chapter 9 covers `Core Bound`, Chapter 10 covers `Bad Speculation`, and Chapter 11 covers `FrontEnd Bound`. The intention of such a structure is to form a checklist that you can use to drive code changes when you encounter a certain performance bottleneck.

Additional resources and links

- Ahmad Yasin's paper "A top-down method for performance analysis and counters architecture" [Yasin, 2014].
- Presentation "Software Optimizations Become Simple with Top-down Analysis on Intel Skylake" by Ahmad Yasin at IDF'15, URL: https://youtu.be/kjufVhyuV_A.
- Andi Kleen's blog: pmu-tools, part II: toplev, URL: http://halobates.de/blog/p/262.
- Toplev manual, URL: https://github.com/andikleen/pmu-tools/wiki/toplev-manual.

6.1.2 TMA on AMD Platforms

[TODO:] Add an example on Zen4 + windows

At the time of this writing, the first level of TMA metrics is also available on AMD processors.

6.1.3 TMA On ARM Platforms

[TODO:] Read https://community.arm.com/arm-community-blogs/b/infrastructure-solutions-blog/posts/arm-neoverse-v1-top-down-methodology https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/neoverse-v1-core-performance-analysis.pdf [TODO:] Which tools I can use to reproduce the results in the paper (TMA breakdown)?

6.1.4 TMA Summary

TMA is great for identifying CPU performance bottlenecks. Ideally, when we run it on an application, we would like to see the Retiring metric at 100%. This would mean that this application fully saturates the CPU. It is possible to achieve results close to this on a toy program. However, real-world applications are far from getting there.

Figure 39 shows top-level TMA metrics for Google's datacenter workloads along with several SPEC CPU2006[94] benchmarks running on Intel's IvyBridge server processors. We can see that that most datacenter workloads have very small fraction in the `Retiring` bucket. This implies that most datacenter workloads spend time stalled on various bottlenecks. `BackendBound` is the primary source of performance issues. `FrontendBound` category represents a bigger problem for datacenter workloads than in SPEC2006 due to the fact that those applications typically have large codebases. Finally, some workloads suffer from branch mispredictions more than others, e.g. `search2` and `445.gobmk`.

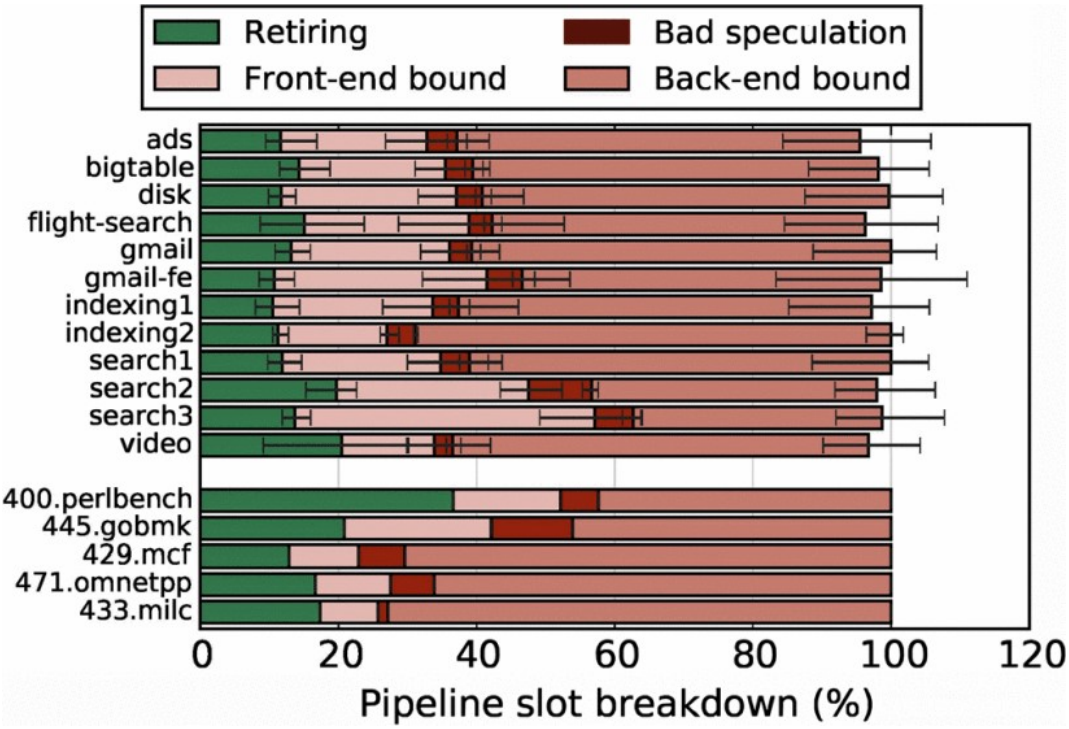


Figure 39: TMA breakdown of Google's datacenter workloads along with several SPEC CPU2006 benchmarks, © *Image from [Kanev et al., 2015]*

Keep in mind that the numbers are likely to change for other CPU generations as architects constantly try to improve the CPU design. The numbers are also likely to change for other instruction set architectures (ISA) and compiler versions.

A few final thoughts before we move on... Using TMA on a code that has major performance flaws is not recommended because it will likely steer you in the wrong direction, and instead of fixing real high-level performance problems, you will be tuning bad code, which is just a waste of time. Similarly, make sure the environment doesn't get in the way of profiling. For example, if you drop filesystem cache and run the benchmark under TMA, it will likely show that your application is Memory Bound, which in fact, may be false when filesystem cache is warmed up.

Workload characterization provided by TMA can increase the scope of potential optimizations beyond source code. For example, if an application is bound by memory bandwidth and all possible ways to speed it up on the software level have been exhausted, it may be possible to improve performance by upgrading the memory subsystem with faster memory chips. This illustrates how using TMA to diagnose performance bottlenecks can support your decision to spend money on a new hardware.

[TODO:] 1. reformat [TODO:] 2. enhance Intel's section [TODO:] 3. read about ARM's BRBE [TODO:] 4. incorporate AMD's LBR

6.2 Branch Recording Mechanisms

Modern high-performance CPUs provide branch recording mechanisms that enable a processor to continuously log a set of previously executed branches. But before going into the details, you may ask: *Why are we so interested*

[94] SPEC CPU 2006 - http://spec.org/cpu2006/.

in branches? Well, because this is how we can determine the control flow of a program. We largely ignore other instructions in a basic block (see Section 11.2) because branches are always the last instruction in a basic block. Since all instructions in a basic block are guaranteed to be executed once, we can only focus on branches that will "represent" the entire basic block. Thus, it's possible to reconstruct the entire line-by-line execution path of the program if we track the outcome of every branch. In fact, this is what the Intel Processor Traces (PT) feature is capable of doing, which is discussed in Appendix D. Branch recording mechanisms that we will discuss here are based on samling, not tracing, and thus have different use cases and capabilities.

Processors designed by Intel, AMD, and ARM all have announced their branch recording extensions. Exact implementations may vary but the idea is the same. Hardware logs the "from" and "to" address of each branch along with some additional data in parallel with executing the program. If we collect a long enough history of source-destination pairs, we will be able to unwind the control flow of our program, just like a call stack, but with limited depth. Such extensions are designed to cause minimal slowdown to a running program often within 1%.

With a branch recording mechanism in place, we can sample on branches (or cycles, it doesn't matter), but during each sample, look at the previous N branches that were executed. This gives us reasonable coverage of the control flow in the hot code paths but does not overwhelm us with too much information, as only a smaller number of the total branches are examined. It is important to keep in mind that this is still sampling, so not every executed branch can be examined. A CPU generally executes too fast for that to be feasible.

It is very important to keep in mind that only taken branches are being logged. Listing 17 shows an example of how branch results are being tracked. This code represents a loop with three instructions that may change the execution path of the program, namely loop backedge `JNE` (1), conditional branch `JNS` (2), function `CALL` (3), and return from this function (4, not shown).

Listing 17 Example of logging branches.

```
----> 4eda10:  mov   edi,DWORD PTR [rbx]
|     4eda12:  test  edi,edi
| --- 4eda14:  jns   4eda1e              <== (2)
| |   4eda16:  mov   eax,edi
| |   4eda18:  shl   eax,0x7
| |   4eda1b:  lea   edi,[rax+rdi*8]
| └─> 4eda1e:  call  4edb26              <== (3)
|     4eda23:  add   rbx,0x4             <== (4)
|     4eda27:  mov   DWORD PTR [rbx-0x4],eax
|     4eda2a:  cmp   rbx,rbp
 ---- 4eda2d:  jne   4eda10              <== (1)
```

Below is one of the possibile branch histories that can be logged with a branch recording mechanism. It shows the last 7 branch outcomes (many more not shown) at the moment we executed the `CALL` instruction. Because on the latest iteration of the loop the `JNS` branch (`4eda14 -> 4eda1e`) was not taken, it is not logged and thus does not appear in the history.

```
    Source Address    Destination Address
    ...               ...
(1) 4eda2d            4eda10    <== next iteration
(2) 4eda14            4eda1e    <== jns taken
(3) 4eda1e            4edb26    <== call a function
(4) 4b01cd            4eda23    <== return from a function
(1) 4eda2d            4eda10    <== next iteration
(3) 4eda1e            4edb26    <== latest branch
```

Untaken branches not being logged might add an additional burden for analysis but usually don't complicate it too much. We can still infer the complete execution path since we know that the control flow was sequential from destination address in entry `N-1` to source address in entry `N`.

Next we will take a look at each vendor's branch recording mechanism and then explore how they can be used in performance analysis.

6.2.1 LBR on Intel Platforms

Intel has first implemented its Last Branch Record (LBR) facility in the Netburst microarchitecture. Initially, it could record only 4 most recent branch outcomes. It was later enhanced to 16 starting with Nehalem and to 32 starting from Skylake. Prior to the Goldencove microarchitecture, LBR was implemented as a set of model-specific registers (MSRs), but now it works within architectural registers. The primary advantage of it is that LBR features are clearly exposed and there is no need to check the exact model number of the current CPU. It makes support in the OS and profiling tools much easier. Also, LBR entries can be configured to be included in the PEBS records (see Section **??**).

The LBR registers act like a ring buffer that is continuously overwritten and provides only 32 most recent branch outcomes. Each LBR entry is comprised of three 64-bit values:

- The source address of the branch (`From IP`).
- The destination address of the branch (`To IP`).
- Metadata for the operation, including mispredict, and elapsed cycle time information.

There are important applications to the additional information saved besides just source and destination addresses, which we will discuss later.

When a sampling counter overflows and a Performance Monitoring Interrupt (PMI) is triggered, the LBR logging freezes until software captures the LBR records and resumes collection.

LBR collection can be limited to a set of specific branch types, for example a user may choose to log only function calls and returns. When applying such filter to the code in Listing 17, we would only see branches (3) and (4) in the history. Users can also filter in/out conditional and unconditional jumps, indirect jumps and calls, system calls, interrupts and others. In Linux perf there is a `-j` option that enables/disables recording of various branch types.

By default, the LBR array works as a ring buffer that captures control flow transitions. However, the depth of the LBR array is limited, which can be a limiting factor when profiling certain applications, in which a transition of the execution flow is accompanied by a large number of leaf function calls. These calls to leaf functions, and their returns, are likely to displace the main execution context from the LBRs. Consider the example in Listing 17 again. Say, we want to unwind the call stack from the history in LBR, and so we configured LBR to capture only function calls and returns. If the loop runs thousands of iterations and taking into account that the LBR array is only 32 entries deep, there is a very high chance we would only see 16 pairs of entries (3) and (4). In such a scenario, the LBR array is cluttered with leaf function calls which don't help us to unwind the current call stack.

This is why LBR supports call-stack mode. With this mode enabled, the LBR array captures function calls as before, but as return instructions are executed the last captured branch (`call`) record is flushed from the array in a last-in first-out (LIFO) manner. Thus, branch information pertaining to completed leaf functions will not be retained, while preserving the call stack information of the main line execution path. When configured in this manner, the LBR array emulates a call stack, where `CALL`s are "pushed" and `RET`s "pop" entries off the stack. If the depth of the call stack in your application never goes beyond 32 nested frames, LBRs will give you a very accurate information.

Users can make sure LBRs are enabled on their system by doing the following command:

```
$ dmesg | grep -i lbr
[    0.228149] Performance Events: PEBS fmt3+, 32-deep LBR, Skylake events, full-width
    counters, Intel PMU driver.
```

With Linux `perf`, you can collect LBR stacks using the following command:

```
$ perf record -b -e cycles ./benchmark.exe
[ perf record: Woken up 68 times to write data ]
[ perf record: Captured and wrote 17.205 MB perf.data (22089 samples) ]
```

LBR stacks can also be collected using `perf record --call-graph lbr` command, but the amount of information collected is less than using `perf record -b`. For example, branch misprediction and cycles data is not collected when running `perf record --call-graph lbr`.

Because each collected sample captures the entire LBR stack (32 last branch records), the size of collected data (`perf.data`) is significantly bigger than sampling without LBRs. Still, runtime overhead for the majority of LBR use cases is below 1%. [Nowak & Bitzes, 2014]

Users can export raw LBR stacks for custom analysis. Below is the Linux perf command you can use to dump the contents of collected branch stacks:

```
$ perf record -b -e cycles ./benchmark.exe
$ perf script -F brstack &> dump.txt
```

The `dump.txt` file, which can be quite large, contains lines like those shown below:

```
...
0x4edaf9/0x4edab0/P/-/-/29
0x4edabd/0x4edad0/P/-/-/2
0x4edadd/0x4edb00/M/-/-/4
0x4edb24/0x4edab0/P/-/-/24
0x4edabd/0x4edad0/P/-/-/2
0x4edadd/0x4edb00/M/-/-/1
0x4edb24/0x4edab0/P/-/-/3
0x4edabd/0x4edad0/P/-/-/1
...
```

In the output above, we present eight entries from the LBR stack, which typically consists of 32 LBR entries. Each entry has `FROM` and `TO` addresses (hexadecimal values), predicted flag (`M` - Mispredicted, `P` - Predicted), and a number of cycles (number in the last position of each entry). Components marked with "`-`" are related to transactional memory extension (TSX), which we won't discuss here. Curious readers can look up the format of a decoded LBR entry in the `perf script` specification[95].

6.2.2 LBR on AMD Platforms

6.2.3 BRBE on ARM Platforms

ARM has introduced its branch recording extension called BRBE in 2020 as a part of ARMv9.2-A ISA. ARM BRBE is very similar to Intel's LBR and provide many similar features. Just like Intel's LBR, BRBE records contain source and destination addresses, misprediction bit and cycle count value. The Branch records only contain information for a branch that is architecturally executed, i.e. not on a mispredicted path. Users can also filter records based on specific branch types. One notable difference is that BRBE supports configurable depth of the BRBE buffer: processors can choose the capacity of the BRBE buffer to be 8, 16, 32 or 64 records. More details are available in [Arm, 2022a, Chapter F1 "Branch Record Buffer Extension"].

At the time of writing, there were no commercially available machines that implement ARMv9.2-A, so it is not possible to test this extension in action.

6.2.4 Capture Call Stacks

There is a number of important use cases that become possible thanks to branch recording. In this and a few later sections, we will cover the most important ones.

One of the most popular use cases for branch recording is capturing call stacks. We already covered why we need to collect them in Section 5.4.3. Branch recording can be used as a light-weight substituion for collecting call-graph information even if you compiled a program without frame pointers or debug information.

At the time of writing (2023), AMD's LBR doesn't support call stack collection, but Intel's LBR does. Here is how you can do it with Intel LBR:

```
$ perf record --call-graph lbr -- ./a.exe
$ perf report -n --stdio
# Children   Self    Samples  Command  Object  Symbol
# ........  .......  .......  .......  ......  .......
  99.96%  99.94%    65447    a.exe    a.exe  [.] bar
            |
             --99.94%--main
```

[95] Linux `perf script` manual page - http://man7.org/linux/man-pages/man1/perf-script.1.html.

```
                    |
                    |--90.86%--foo
                    |          |
                    |           --90.86%--bar
                    |
                     --9.08%--zoo
                               bar
```

As you can see, we've identified the hottest function in the program (which is `bar`). Also, we found out callers that contribute to the most time spent in function `bar`: 91% of the time the tool captured the `main->foo->bar` call stack and 9% it captured `main->zoo->bar`. In other words, 91% of samples in `bar` have `foo` as its caller function.

It's important to mention that we cannot necessarily drive conclusions about function call counts in this case. For example, we cannot say that `foo` calls `bar` 10 times more frequently than `zoo`. It could be the case that `foo` calls `bar` once, but it executes an expensive path inside `bar` while `zoo` calls `bar` many times but returns quickly from it.

6.2.5 Identify Hot Branches

Branch recording also enables us to know what were the most frequently taken branches. Here is an example of using Intel's LBR:

```
$ perf record -e cycles -b -- ./a.exe
[ perf record: Woken up 3 times to write data ]
[ perf record: Captured and wrote 0.535 MB perf.data (670 samples) ]
$ perf report -n --sort overhead,srcline_from,srcline_to -F +dso,symbol_from,symbol_to --stdio
# Samples: 21K of event 'cycles'
# Event count (approx.): 21440
# Overhead  Samples  Object  Source Sym  Target Sym  From Line  To Line
# ........  .......  ......  ..........  ..........  .........  .......
  51.65%      11074   a.exe   [.] bar     [.] bar      a.c:4      a.c:5
  22.30%       4782   a.exe   [.] foo     [.] bar      a.c:10     (null)
  21.89%       4693   a.exe   [.] foo     [.] zoo      a.c:11     (null)
   4.03%        863   a.exe   [.] main    [.] foo      a.c:21     (null)
```

From this example, we can see that more than 50% of taken branches are within the `bar` function, 22% of branches are function calls from `foo` to `bar`, and so on. Notice how `perf` switched from `cycles` events to analyzing LBR stacks: only 670 samples were collected, yet we have an entire LBR stack captured with every sample. This gives us `670 * 32 = 21440` LBR entries (branch outcomes) for analysis.[96]

Most of the time, it's possible to determine the location of the branch just from the line of code and target symbol. However, theoretically, one could write code with two `if` statements written on a single line. Also, when expanding the macro definition, all the expanded code is attributed to the same source line, which is another situation when this might happen. This issue does not totally block the analysis but only makes it a little more difficult. To disambiguate two branches, you likely need to analyze raw LBR stacks yourself (see example on easyperf[97] blog).

Using branch recording, we can also find a *hyper block* (sometimes called *super block*), which is a chain of hot basic blocks in a function that are not necessarily laid out in the sequential physical order but they're executed sequentially. Thus, a hyper block represents a typical hot path through a function, piece of code, or a program.

6.2.6 Analyze Branch Misprediction Rate

Thanks to the mispredict bit in the additional information saved inside each record, it is also possible to know the misprediction rate for hot branches. In this example we take a C-code-only version of the 7-zip benchmark from the LLVM test-suite.[98] The output of perf report is slightly trimmed to fit nicely on a page.

[96] The report header generated by perf might still be confusing because it says `21K of event cycles`. But there are `21K` LBR entries, not `cycles`.

[97] Easyperf: Estimating Branch Probability - https://easyperf.net/blog/2019/05/06/Estimating-branch-probability

[98] LLVM test-suite 7zip benchmark - https://github.com/llvm-mirror/test-suite/tree/master/MultiSource/Benchmarks/7zip

TODO: Check: "Adding `-F +srcline_from,srcline_to` slows down building report. Hopefully, in newer versions of perf, decoding time will be improved".

```
$ perf record -e cycles -b -- ./7zip.exe b
$ perf report -n --sort symbol_from,symbol_to -F +mispredict,srcline_from,srcline_to --stdio
# Samples: 657K of event 'cycles'
# Event count (approx.): 657888
# Overhead  Samples  Mis  From Line  To Line  Source Sym  Target Sym
# ........  .......  ...  .........  .......  ..........  ..........
    46.12%   303391   N   dec.c:36   dec.c:40  LzmaDec     LzmaDec
    22.33%   146900   N   enc.c:25   enc.c:26  LzmaFind    LzmaFind
     6.70%    44074   N   lz.c:13    lz.c:27   LzmaEnc     LzmaEnc
     6.33%    41665   Y   dec.c:36   dec.c:40  LzmaDec     LzmaDec
```

In this example, the lines that correspond to function `LzmaDec` are of particular interest to us. Following a similar analysis from the previous section, we can conclude that the branch on source line `dec.c:36` is the most executed branch in the benchmark. In the output that Linux `perf` provides, we can spot two entries that correspond to the `LzmaDec` function: one with `Y` and one with `N` letters. Analyzing those two entries together gives us a misprediction rate of the branch. In this case, we know that the branch on line `dec.c:36` was predicted `303391` times (corresponds to `N`) and was mispredicted `41665` times (corresponds to `Y`), which gives us `88%` prediction rate.

Linux `perf` calculates the misprediction rate by analyzing each LBR entry and extracting misprediction bits from it. So that for every branch, we have a number of times it was predicted correctly and a number of mispredictions. Again, due to the nature of sampling, it is possible that some branches might have an `N` entry but no corresponding `Y` entry. It could mean there are no LBR entries for the branch being mispredicted, but that doesn't necessarily mean the prediction rate is `100%`.

6.2.7 Precise Timing of Machine Code

As we showed in the Intel's LBR section, starting from Skylake microarchitecture, there is a special `Cycle Count` field in the LBR entry. This additional field specifies the number of elapsed cycles between two taken branches. Since the target address in the previous (N-1) LBR entry is the beginning of a basic block (BB) and the source address of the current (N) LBR entry is the last instruction of the same basic block, then the cycle count is the latency of this basic block. For example:

```
400618:   movb  $0x0, (%rbp,%rdx,1)    <= start of a BB
40061d:   add $0x1, %rdx
400621:   cmp $0xc800000, %rdx
400628:   jnz 0x400644                 <= end of a BB
```

Suppose we have two entries in the LBR stack:

```
FROM_IP   TO_IP    Cycle Count
...       ...      ...
40060a    400618    10
400628    400644     5          <== LBR TOS
```

Given that information, we know that there was one occurrence when the basic block that starts at offset `400618` was executed in 5 cycles. If we collect enough samples, we could plot a probability density function of the latency for that basic block. The chart in Figure 40 was compiled by analyzing all LBR entries that satisfy the rule described above. For example, the basic block was executed in ~75 cycles only 4% of the time, but more often, it was executed in between 260 and 314 cycles. This block has a non-sequential load from a large array that doesn't fit in CPU L3 cache, so the latency of the basic block largely depends on this load. There are two important spikes on the chart: first, around 80 cycles corresponds to the L3 cache hit, and the second spike, around 300 cycles, corresponds to L3 cache miss where the load request goes all the way down to the main memory.

This information can be used for a fine-grained tuning of this basic block. This example might benefit from memory prefetching, which we will discuss in Section 8.2. Also, cycle count information can be used for timing loop iterations, where every loop iteration ends with a taken branch (back edge).

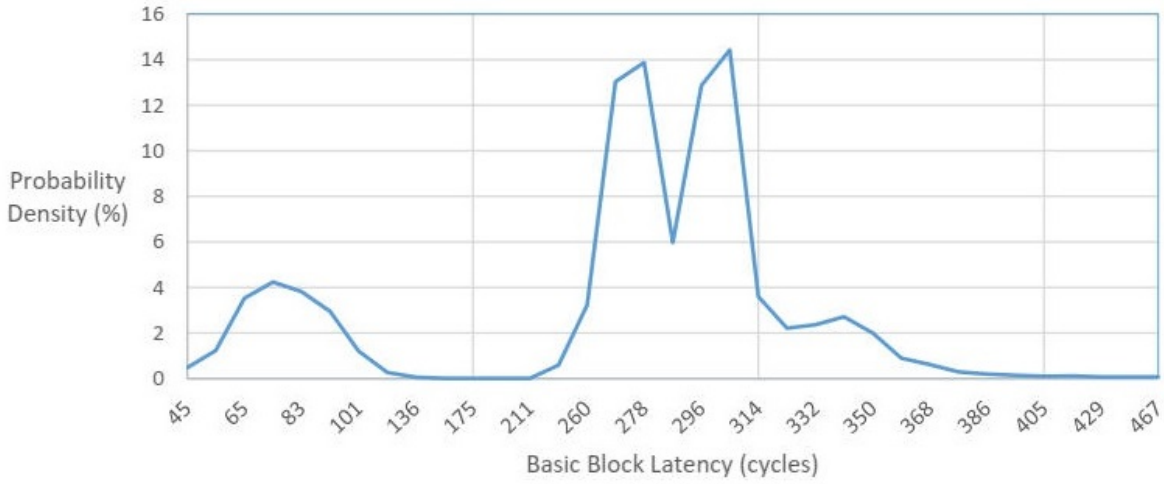


Figure 40: Probability density chart for latency of the basic block that starts at address `0x400618`.

Before the proper support from profiling tools was in place, building probability density graphs similar to Figure 40 required manual parsing of raw LBR dumps. Example of how to do this can be found on the easyperf blog[99]. Luckily, in newer versions of Linux perf, getting this information is much easier. The example below demonstrates this method directly using Linux perf on the same 7-zip benchmark from the LLVM test-suite we introduced earlier:

TODO: Check: "Adding `-F +srcline_from,srcline_to` slows down building report. Hopefully, in newer versions of perf, decoding time will be improved".

```
$ perf record -e cycles -b -- ./7zip.exe b
$ perf report -n --sort symbol_from,symbol_to -F +cycles,srcline_from,srcline_to --stdio
# Samples: 658K of event 'cycles'
# Event count (approx.): 658240
# Overhead  Samples  BBCycles  FromSrcLine  ToSrcLine
# ........  .......  ........  ...........  .........
     2.82%   18581        1      dec.c:325    dec.c:326
     2.54%   16728        2      dec.c:174    dec.c:174
     2.40%   15815        4      dec.c:174    dec.c:174
     2.28%   15032        2      find.c:375   find.c:376
     1.59%   10484        1      dec.c:174    dec.c:174
     1.44%   9474         1      enc.c:1310   enc.c:1315
     1.43%   9392        10      7zCrc.c:15   7zCrc.c:17
     0.85%   5567        32      dec.c:174    dec.c:174
     0.78%   5126         1      enc.c:820    find.c:540
     0.77%   5066         1      enc.c:1335   enc.c:1325
     0.76%   5014         6      dec.c:299    dec.c:299
     0.72%   4770         6      dec.c:174    dec.c:174
     0.71%   4681         2      dec.c:396    dec.c:395
     0.69%   4563         3      dec.c:174    dec.c:174
     0.58%   3804        24      dec.c:174    dec.c:174
```

Notice we've added the `-F +cycles` option to show cycle counts in the output (`BBCycles` column). Several insignificant lines were removed from the output of `perf report` to make it fit on the page. Let's focus on lines in which source and destination is `dec.c:174`, there are seven such lines in the output. In the source code, the line `dec.c:174` expands a macro that has a self-contained branch. That's why the source and destination happen to be on the same line.

Linux `perf` sorts entries by overhead first, so we need to manually filter entries for the branch which we are interested

[99] Easyperf: Building a probability density chart for the latency of an arbitrary basic block - https://easyperf.net/blog/2019/04/03/Precise-timing-of-machine-code-with-Linux-perf.

in. Luckily, they can be grepped very easily. In fact, if we filter them, we will get the latency distribution for the basic block that ends with this branch, as shown in Table 7. This data can be plotted to obtain a chart similar to the one shown in Figure 40.

Table 7: Probability density for basic block latency.

Cycles	Number of samples	Probability density
1	10484	17.0%
2	16728	27.1%
3	4563	7.4%
4	15815	25.6%
6	4770	7.7%
24	3804	6.2%
32	5567	9.0%

Here is how we can interpret the data: from all the collected samples, 17% of the time the latency of the basic block was one cycle, 27% of the time it was 2 cycles, and so on. Notice a distribution mostly concentrates from 1 to 6 cycles, but also there is a second mode of much higher latency of 24 and 32 cycles, which likely corresponds to branch misprediction penalty. The second mode in the distribution accounts for 15% of all samples.

This example that shows that it is feasible to plot basic block latencies not only for tiny microbenchmarks, but for real-world applications as well. Currently, LBR is the most precise cycle-accurate source of timing information on Intel systems.

6.2.8 Estimating Branch Outcome Probability

Later in Chapter 11, we will discuss the importance of code layout for performance. Going forward a little bit, having a hot path in a fall through manner[100] generally improves the performance of a program. Knowing what is the most frequent outcome of a certain branch enables developers and compilers to make better optimization decisions. For example, given that a branch is taken 99% of the time, we can try to invert the condition and convert it to a non-taken branch.

LBR enables us to collect this data without instrumenting the code. As the outcome from the analysis, a user will get a ratio between true and false outcomes of the condition, i.e., how many times the branch was taken and how much not taken. This feature especially shines when analyzing indirect jumps (switch statement) and indirect calls (virtual calls). You can find examples of using it on a real-world application on the easyperf blog[101].

6.2.9 Providing Compiler Feedback Data

We will discuss Profile Guided Optimizations (PGO) later in Section 11.7, so just a quick mention here. Branch recording mechanisms can provide profiling feedback data for optimizing compilers. Imagine that we can feed all the data we discovered in the previous sections back to the compiler. In some cases, this data cannot be obtained using traditional static code instrumentation, so branch recording mechanisms are not only a better choice because of the lower overhead, but also because of richer profiling data. PGO workflows that rely on data collected from the hardware PMU, are becoming more popular and likely will take off sharply once the support in AMD and ARM will mature.

6.3 Hardware-Based Sampling Features

Major CPU vendors provide a set of additional features to enhance performance analysis. Since CPU vendors approach performance monitoring in different ways, those capabilities vary in not only how they are called but also what you can do with them. In Intel processors, it is called Processor Event-Based Sampling (PEBS), first introduced in NetBurst microarchitecture. A similar feature on AMD processors is called Instruction Based Sampling (IBS) and is available starting with the AMD Opteron Family (10h generation) of cores. Next we will discuss those features in more details, including their similarities and differences.

[100] I.e., when the hot branches are not taken.
[101] Easyperf: Estimating Branch Probability - https://easyperf.net/blog/2019/05/06/Estimating-branch-probability

6.3.1 PEBS on Intel Platforms

Similar to Last Branch Record, PEBS is used while profiling the program to capture additional data with every collected sample. When a performance counter is configured for PEBS, the processor saves the set of additional data, which has a defined format and is called the PEBS record. The format of a PEBS record for Intel Skylake CPU is shown in Figure 41. The record contains the state of general-purpose registers (`EAX`, `EBX`, `ESP`, etc.), `EventingIP`, `Data Linear Address`, and `Latency value`, which will discuss later. The content layout of a PEBS record varies across different microarchitectures, see [Intel, 2023b, Volume 3B, Chapter 20 Performance Monitoring].

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	68H	R11
08H	R/EIP	70H	R12
10H	R/EAX	78H	R13
18H	R/EBX	80H	R14
20H	R/ECX	88H	R15
28H	R/EDX	90H	Applicable Counter
30H	R/ESI	98H	Data Linear Address
38H	R/EDI	A0H	Data Source Encoding
40H	R/EBP	A8H	Latency value (core cycles)
48H	R/ESP	B0H	EventingIP
50H	R8	B8H	TX Abort Information (Section 18.3.6.5.1)
58H	R9	C0H	TSC
60H	R10		

Figure 41: PEBS Record Format for 6th Generation, 7th Generation and 8th Generation Intel Core Processor Families. © *Image from [Intel, 2023b, Volume 3B, Chapter 18].*

Since Skylake, the PEBS record has been enhanced to collect XMM registers and LBR records. The format has been restructured where fields are grouped into Basic group, Memory group, GPR group, XMM group, and LBR group. Performance profiling tools have the option to select data groups of interest and thus reduce the record size in memory and record generation latency. By default, the PEBS record will only contain the Basic group.

One of the notable benefits of using PEBS is lower sampling overhead compared to a regular interrupt-based sampling. Recall that when the counter overflows, the CPU generates an interrupt to collect one sample. Frequently generating interrupts and having an analysis tool itself capture program state inside the interrupt service routine is very costly since it involves OS interaction.

On the other hand, PEBS keeps a buffer to temporarily store multiple PEBS records. Suppose, we are sampling on load events using PEBS. When a performance counter is configured for PEBS, an overflow condition in the counter will not trigger an interrupt, instead it will arm the PEBS mechanism. The mechanism will then trap the next load, capture a new record and store it in the dedicated PEBS buffer area. The mechanism also takes care of clearing the counter overflow status and reloading the counter with the initial value. Only when the dedicated buffer is full, the processor raises an interrupt, and the buffer gets flushed to memory. This mechanism lowers the sampling overhead by triggering much fewer interrupts.

Linux users can check if PEBS is enabled by executing `dmesg`:

```
$ dmesg | grep PEBS
[    0.113779] Performance Events: XSAVE Architectural LBR, PEBS fmt4+-baseline,
AnyThread deprecated, Alderlake Hybrid events, 32-deep LBR, full-width counters, Intel PMU
    driver.
```

For LBR, Linux perf dumps entire contents of LBR stack with every collected sample. So, it is possible to analyze raw LBR dumps collected by Linux perf. However, for PEBS, Linux `perf` doesn't export the raw output as it does for LBR. Instead, it processes PEBS records and extracts only the subset of data depending on a particular need. So, it's not possible to access the collection of raw PEBS records with Linux `perf`. However, Linux `perf` provides

some PEBS data processed from raw samples, which can be accessed by `perf report -D`. To dump raw PEBS records, one can use `pebs-grabber`[102] tool.

6.3.2 IBS on AMD Platforms

Instruction-Based Sampling (IBS) is a AMD64 processor feature that can be used to collect specific metrics related to instruction fetch and instruction execution. The processor pipeline of an AMD processor consists of two separate phases: a front-end phase that fetches AMD64 instruction bytes and a back-end phase that executes `ops`. As the phases are logically separated, there are two independent sampling mechanisms: IBS Fetch and IBS Execute.

- IBS Fetch monitors the front-end of the pipeline and provides information about ITLB (hit or miss), I-cache (hit or miss), fetch address, fetch latency and a few other things.
- IBS Execute monitors the back-end of the pipeline and provides information about instruction execution behavior by tracking the execution of a single op. For example: branch (taken or not, predicted or not), load/store (hit or miss in D-caches and DTLB, linear address, load latency).

There is a number of important differences between PMC and IBS in AMD processors. PMC counters are programmable, whereas IBS acts like fixed counters. IBS counters can only be enabled or disabled for monitoring, they can't be programmed to any selective events. IBS Fetch and Op counters can be enabled/disabled independently. With PMC, user has to decide what events to monitor ahead of time. With IBS, a rich set of data is collected for each sampled instruction and then it is up to the user to analyze parts of the data they are interested in. IBS selects and tags an instruction to be monitored and then captures microarchitectural events caused by this instruction during its execution.

Since IBS is integrated into the processor pipeline and acts as a fixed event counter, the sample collection overhead on the processor is minimal. Profilers are required to process the IBS generated data, which could be huge in size depending upon sampling interval, number of threads configured, whether Fetch/Op configured, etc. Until Linux kernel version 6.1, IBS always collects samples for all the cores. This limitation causes huge data collection and processing overhead. From Kernel 6.2 onwards, Linux perf supports IBS sample collection only for the configured cores. And of course, IBS is supported by the AMD uProf profiler.

A more detailed comparison of Intel PEBS and AMD IBS can be found in [Sasongko et al., 2023].

6.3.3 SPE on ARM Platforms

The Arm Statistical Profiling Extension (SPE) is an architectural feature designed for enhanced instruction execution profiling within Arm CPUs. This feature has been available since Neoverse N1 cores introduced in 2019. The SPE feature extension is specified as part of Armv8-A architecture, with support from Arm v8.2 onwards. Compared to other solutions, SPE is more similar to AMD IBS than it is to Intel PEBS. Similar to IBS, SPE is separate from the general performance monitor counters (PMC), but instead of two flavors of IBS (fetch and execute), there is just a single mechanism.

The SPE sampling process is built in as part of the instruction execution pipeline. Sample collection is still based on a configurable interval, but operations are statistically selected. Each sampled operation generates a sample record, which contains various data about execution of this operation. SPE record saves address of the instruction, virtual and physical address for the data accessed by loads and stores, the source of the data access (cache or DRAM) and timestamp to correlate with other events in the system. Also, it can give latency of various pipeline stages, such as Issue latency (from dispatch to execution), Translation latency (cycle count for a virtual-to-physical address translation) and Execution latency (latency of load/stores in the functional unit). The whitepaper [Limited, 2023] describes ARM SPE in more details as well as shows a few optimization examples using it.

Similar to Intel PEBS and AMD IBS, ARM SPE helps to reduce the sampling overhead and enables longer collections. In addition to that, it supports post-filtering of sample records, which helps to reduce the memory required for storage.

SPE profiling is enabled in the Linux `perf` tool and can be used as follows:

```
$ perf record -e arm_spe_0/<controls>/ -- test_program
$ perf report --stdio
$ spe-parser perf.data -t csv
```

[102] PEBS grabber tool - https://github.com/andikleen/pmu-tools/tree/master/pebs-grabber. Requires root access.

, where `<controls>` lets you optionally specify various controls and filters for the collection. `perf report` will give the usual output according to what user asked for with `<controls>` options. `spe-parser`[103] is a tool developed by ARM engineers to parse the captured perf record data and save all the SPE records into a CSV file.

6.3.4 Precise Events

Now that we covered the advanced sampling features, let's discuss **how** they can be used to improve performance analysis. We will start with the notion of precise events.

One of the major problems in profiling is pinpointing the exact instruction that caused a particular performance event. As discussed in Section 5.4, interrupt-based sampling is based on counting a specific performance event and waiting until it overflows. When an overflow interrupt happens, it takes a processor some time to stop the execution and tag the instruction that caused the overflow. This is especially difficult for modern complex out-of-order CPU architectures.

It introduces the notion of a skid, which is defined as the distance between the IP (instruction address) that caused the event to the IP where the event is tagged. Skid makes it difficult to discover the instruction causing the performance issue. Consider an application with a big number of cache misses and the hot assembly code that looks like this:

```
; load1
; load2
; load3
```

The profiler might tag `load3` as the instruction that causes a large number of cache misses, while in reality, `load1` is the instruction to blame. For high-performance processors, this skid can be hundreds of processor instructions. This usually causes a lot of confusion for performance engineers. Interested readers could learn more about underlying reasons for such issues on Intel Developer Zone website[104].

The problem with the skid is mitigated by having the processor itself store the instruction pointer (along with other information). With Intel PEBS, the `EventingIP` field in the PEBS record indicates the instruction that caused the event. This is typically available only for a subset of supported events, called "Precise Events". A complete list of precise events for specific microarchitecture can be found in [Intel, 2023b, Volume 3B, Chapter 20 Performance Monitoring]. An example of using PEBS precise events to mitigate skid can be found on the easyperf blog.[105]

Listed below are precise events for the Skylake Microarchitecture:

```
INST_RETIRED.*          OTHER_ASSISTS.*      BR_INST_RETIRED.*       BR_MISP_RETIRED.*
FRONTEND_RETIRED.*      HLE_RETIRED.*        RTM_RETIRED.*           MEM_INST_RETIRED.*
MEM_LOAD_RETIRED.*      MEM_LOAD_L3_HIT_RETIRED.*
```

, where `.*` means that all sub-events inside a group can be configured as precise events.

Users of Linux `perf` on Intel platforms should add `pp` suffix to the event to enable precise tagging:

```
$ perf record -e cycles:pp -- ./a.exe
```

With AMD IBS and ARM SPE, all the collected samples are precise by design since the HW captures the exact instruction address. In fact, they work in a very similar fashion. Whenever an overflow occurs, the mechanism saves the instruction causing the overflow into a dedicated buffer which is then read by the interrupt handler. As the address is preserved, IBS and SPE samples attribution to the instructions are precise.

Precise events provide a relief for performance engineers. Also, the TMA methodology heavily relies on precise events to locate the exact line of source code where the inefficient execution takes place.

[103] ARM SPE parser - https://gitlab.arm.com/telemetry-solution/telemetry-solution
[104] Hardware event skid - https://software.intel.com/en-us/vtune-help-hardware-event-skid
[105] Performance skid - https://easyperf.net/blog/2018/08/29/Understanding-performance-events-skid

6.3.5 Analyzing Memory Accesses

Memory accesses are a critical factor for the performance of many applications. Both PEBS and IBS enable gathering detailed information about memory accesses in a program. For instance, you can not only sample loads, but also collect their target addresses and access latency. Keep in mind; this does not trace all the stores and loads. Otherwise, the overhead would be too big. Instead, it analyzes only one out of 100'000 accesses or so. You can customize how many sample per second you want. With large enough collection of samples it can give an accurate statistical picture of the memory accesses.

In PEBS, the feature that allows this to happen is called Data Address Profiling (DLA). To provide additional information about sampled loads and stores, it uses the `Data Linear Address` and `Latency Value` fields inside the PEBS facility (see Figure 41). If the performance event supports the DLA facility, and DLA is enabled, the processor will dump the memory address and latency of the sampled memory access. You can also filter memory accesses that have latency higher than a certain threshold. This is useful for finding long-latency memory accesses, which can be a performance bottleneck for many applications.

With the IBS Execute and ARM SPE sampling, you can also do in-depth analysis of memory accesses performed by an application. One approach is to dump collected samples and process them manually. IBS saves the exact linear address, where the access was served from (cache or DRAM) and whether it hit or missed in the DTLB. SPE can be used to estimate latency and bandwidht of the memory subsystem components, estimate memory latencies of individual loads/stores, and more.

One of the most important use cases for these extensions is detecting True and False Sharing, which we will discuss in Section 13.7. The Linux `perf c2c` tool heavily relies on all three mechanisms (PEBS, IBS and SPE) to find contested memory accesses, which could experience True/False sharing: it matches load/store addresses for different threads and checks if the hit occurrs in a cache line modified by other threads.

Questions and Exercises

1. Name the four level-1 categories in TMA performance methodology.
2. What are the benefits of HW event-based sampling helps?
3. What is performance event skid?
4. Study performance analysis features available on the CPU inside the machine you use for development/benchmarking.

Chapter Summary

- Utilizing HW features for low-level tuning is recommended only once all high-level performance issues are fixed. Tuning poorly designed algorithms is a bad investment of a developer's time. Once all the major performance problems get eliminated, one can use CPU performance monitoring features to analyze and further tune their application.
- Top-down Microarchitecture Analysis (TMA) methodology is a very powerful technique for identifying ineffective usage of CPU microarchitecture by the program. It is a robust and formal methodology that is easy to use even for inexperienced developers. TMA is an iterative process that consists of multiple steps, including characterizing the workload and locating the exact place in the source code where the bottleneck occurs. We advise that TMA should be a starting point of analysis for every low-level tuning effort. TMA is available on Intel and AMD[106] processors.
- Last Branch Record (LBR) mechanism continuously logs the most recent branch outcomes in parallel with executing the program, causing a minimal slowdown. It allows us to have a deep enough call stack for every sample we collect while profiling. Also, LBR helps identify hot branches, misprediction rates and allows for precise timing of machine code. LBR is supported on Intel and AMD processors.
- Processor Event-Based Sampling (PEBS) feature is another enhancement for profiling. It lowers the sampling overhead by automatically sampling multiple times to a dedicated buffer without interrupts. However, PEBS are more widely known for introducing "Precise Events", which allow pinpointing exact instruction that caused a particular performance event. The feature is supported on Intel processors. AMD CPUs have a similar feature called Instruction Based Sampling (IBS).
- Intel Processor Traces (PT) is a CPU feature that records the program execution by encoding packets in a highly compressed binary format that can be used to reconstruct execution flow with a timestamp on every

[106] At the time of writing, AMD processors only support the first level of TMA metrics, i.e., Front End Bound, Back End Bound, Retiring, and Bad Speculation.

instruction. PT has extensive coverage and relatively small overhead. Its main usages are postmortem analysis and finding the root cause(s) of performance glitches. Processors based on ARM architecture also have a tracing capability called CoreSight,[107] but it is mostly used for debugging rather than for performance analysis. Intel PT feature is covered in Appendix D.

Performance profilers leverage HW features presented in this chapter to enable many different types of analysis.

[107] ARM CoreSight - https://developer.arm.com/ip-products/system-ip/coresight-debug-and-trace

7 Overview of Performance Analysis Tools

In the previous chapter, we explored the features implemented in modern processors to aid performance analysis. However, if you were to start directly using those features, it would become very nuanced very quickly as it requires a lot of low-level programming to make use of them. Luckily, performance analysis tools take care of all the complexity that is required to effectively use these HW performance monitoring features. It makes profiling goes smoothly, still it's critical to have an intuition of how the tool obtains and interprets the data. That is why we discuss analysis tools after we discussed CPU performance monitoring features.

This chapter gives a quick overview of the most popular tools available on major platforms. The choice will vary depending on which OS and CPU you're using. Some of the tools are cross-platform but majority are not, so it is important to know what tools are available to you. Those profiling tools are usually developed and maintained by the HW vendors themselves because they are the ones who know how to properly use performance monitoring features available on their CPUs. Unfortunately, this creates a situation when you need to install a specialized tool depending on which CPU you are using if you need to do an advanced performance engineering work.

After reading the chapter, take the time to practice using tools that you may eventually use. Familiarize yourself with the interface and workflow of those tools. Profile the application that you work with on a daily basis. Even if you don't find any actionable insights, you will be much better prepared when the actual need comes.

7.1 Intel Vtune

VTune Profiler (formerly VTune Amplifier) is a performance analysis tool for x86-based machines with a rich GUI interface. It can be run on Linux or Windows operating systems. We skip discussion about MacOS support for VTune since it doesn't work on Apple's chips (e.g. M1 and M2), and Intel-based Macbooks quickly become obsolete.

Vtune can be used on both Intel and AMD systems, many features will work. However, advanced hardware-based sampling requires an Intel-manufactured CPU. For example, you won't be able to collect HW performance counters on an AMD system with Intel Vtune.

As of early 2023, VTune is available for free as a stand-alone tool or as part of the Intel oneAPI Base Toolkit.

How to configure it

On Linux, Vtune can use two data collectors: Linux perf and Vtune's own driver called SEP. First type is used for user-mode sampling, but if you want to perform advanced analysis, you need to build and install SEP driver, which is not too hard.

```
# go to the sepdk folder in vtune's installation
$ cd ~/intel/oneapi/vtune/latest/sepdk/src
# build the drivers
$ ./build-driver
# add vtune group and add your user to that group
# create a new shell, or reboot the system
$ sudo groupadd vtune
$ sudo usermod -a -G vtune `whoami`
# install sep driver
$ sudo ./insmod-sep -r -g vtune
```

After you've done with the steps above, you should be able to use adanced analysis types like Microarchitectural Exploration and Memory Access.

Windows does not require any additional configuration after you install Vtune. Collecting hardware performance events requires administrator priveleges.

What you can do with it:

- find hotspots: functions, loops, statements.

- monitor various CPU-specific performance events, e.g. branch mispredictions and L3 cache misses.
- locate lines of code where these events happen.
- characterize CPU performance bottlenecks with TMA methodology.
- filter data for a specific function, process, time period or logical core.
- observe the workload bahavior over time (incuding CPU frequency, memory bandwidth utilization, etc).

VTune can provide very rich information about a running process. It is the right tool for you if you're looking to improve the overall performance of an application. Vtune always provides an aggregated data over some time period, so it can be used for finding optimization opportunities for the "average case".

What you cannot do with it:

- analyze very short execution anomalies.
- observe system-wide complicated SW dynamics.

Due to the sampling nature of the tool, it will eventually miss events with a very short duration (e.g. submicrosecond).

Examples

Below is a series of screenshots of VTune's most interesting features. For the purpose of this example, we took POV-Ray, a ray tracer that is used to create 3D graphics. Figure 42 shows the hotpots analysis of povray 3.7 built-in benchmark, compiled with clang14 compiler with `-O3 -ffast-math -march=native -g` options, and run on Intel Alderlake system (Core i7-1260P, 4 P-cores + 8 E-cores) with 4 worker threads.

At the left part of the image, you can see a list of hot functions in the workload along with corresponding CPU time percentage and the number of retired instructions. On the right panel, you can see one of the most frequent call stacks that lead to calling the function `pov::Noise`. According to that screenshot, `44.4%` of the time function `pov::Noise`, was called from `pov::Evaluate_TPat`, which in turn was called from `pov::Compute_Pigment`. Notice that the call stack doesn't lead all the way to the `main` function. It happens because with HW-based collection, VTune uses LBR to sample call stacks, which has limited depth. Most likely we're dealing with recursive functions here, and to investigate that further users have to dig into the code.

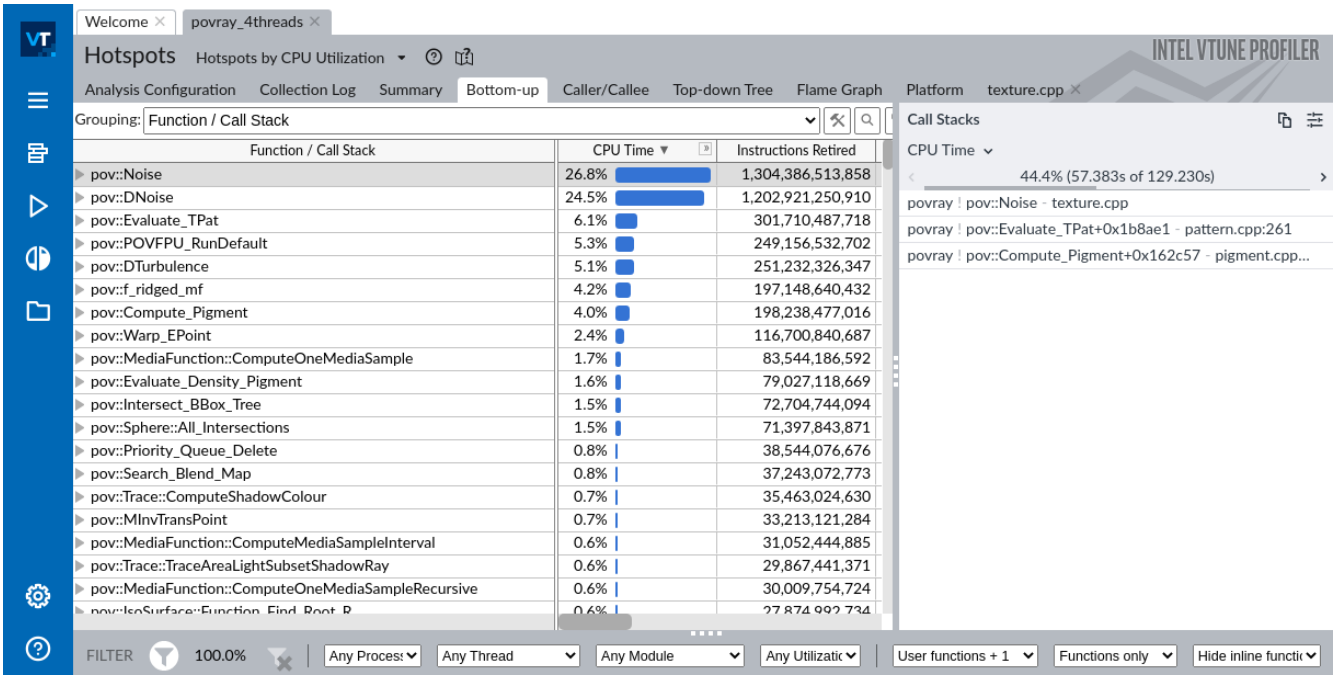


Figure 42: VTune's hotspots view of povray built-in benchmark.

If you double-click on `pov::Noise` function, you will see an image that is shown on Figure 43. For the interest of space, only the most important columns are shown. Left panel shows the source code and CPU time that corresponds to each line of code. On the right, you can see a assembly instructions along with CPU time that was attributed

to them. Highlighted machine instructions correspond to line 476 in the left panel. The sum of all CPU time percentages in both panels equals to the total CPU time attributed to the `pov::Noise`, which is `26.8%`.

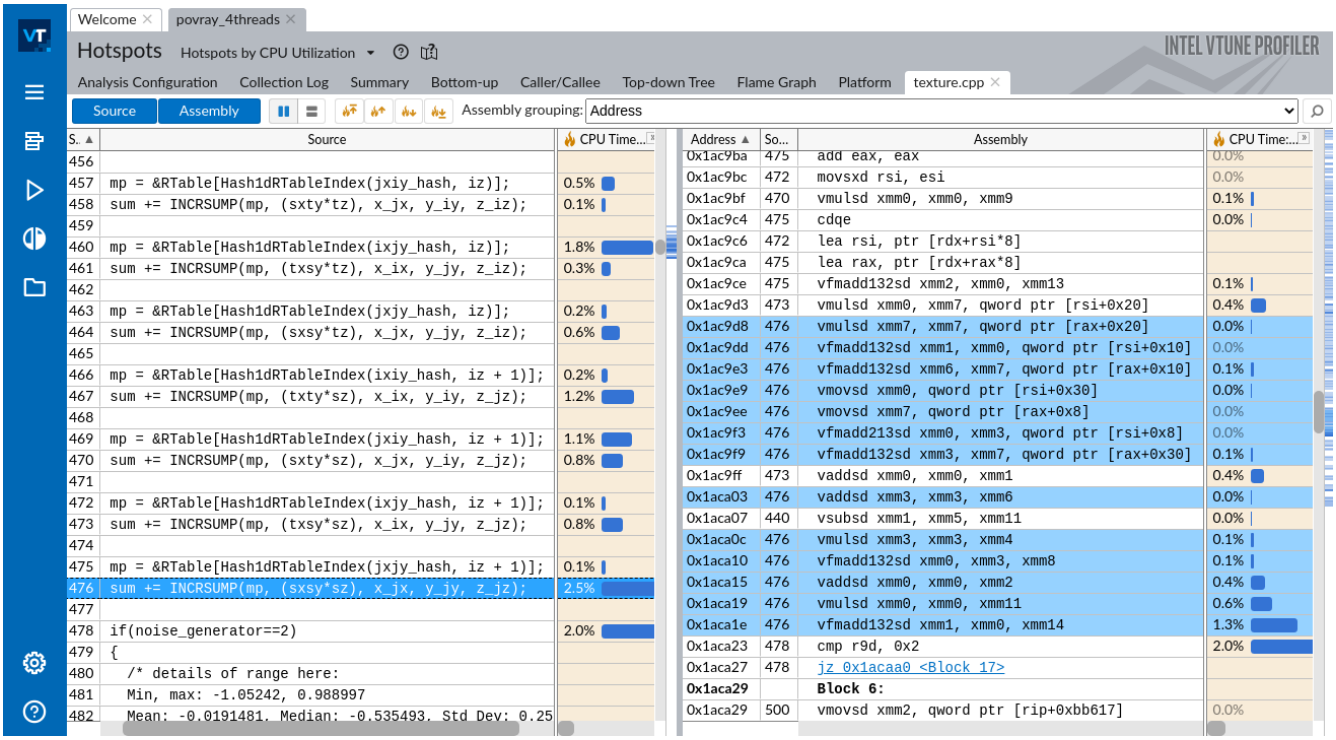


Figure 43: VTune's source code view of povray built-in benchmark.

When you use VTune to profile applications running on Intel CPUs, it can collect many different performance events. To illustrate this, we ran a different analysis type, Microarchitecture Exploration, which we already showed in the previous chapter. At that time we used it for Top-down Microarchitectural Analysis, while we can also use it to observe raw performance events. To access raw event counts, one can switch the view to Hardware Events as shown on Figure 44. To enable switching views, you need to tick the mark in Options -> General -> Show all applicable viewpoints. There are two useful pages, are not shown on the image: Summary page gives you the absolute number of raw performance events as collected from CPU counters, Event Count page gives you the same data with a per-function breakdown. Readers can experiment and look at those views on their own.

Figure 44 is quite busy and requires some explanation. Top panel (region 1) is a timeline view that shows the behavior of our four worker threads over time with respect to L1 cache misses, plus some tiny activity of the main thread (TID: `3102135`), which spawns all the worker threads. The higher the black bar, the more events (L1 cache misses in this case) happend at any given moment. Notice ocassional spikes in L1 misses for all four worker threads. We can use this view to observe different or repeatable phases of the workload. Then to figure out which functions were executed at that time, we can select an interval and click "filter in" to focus just on that portion of the running time. Region 2 is an example of such filtering. To see the updated list of functions, you can go to Bottom Up or, in this case, Event Count view. Such filtering and zooming feature is available on all Vtune timeline views.

Region 3 shows performance events that were collected and their distribution over time. This time it is not a perf-thread view, but rather it shows aggregated data across all the threads. In addition to observing execution phases, you can also visually extract some interesting information. For example, we can see that the number of executed branches is high (`BR_INST_RETIRED.ALL_BRANCHES`), but the misprediction rate is quite low (`BR_MISP_RETIRED.ALL_BRANCHES`). This can lead you to the conclusion that branch misprediction is not a bottleneck for POV-Ray. If you scroll down, you would see that the number of L3 misses is zero, and L2 cache misses are very rare as well. This tells us that 99% of the time, memory access requests are served by L1, and the rest of them are served by L2. Two observations combined, we can conclude that the applications is likely bound by compute, i.e. CPU is busy calculating something, not waiting for memory or recovering from a misprediction.

Finally, the bottom panel (region 4) shows the CPU frequency chart for four hardware threads. Hovering over

different time slices tells us that the frequency of those cores fluctuates in the 3.2GHz - 3.4GHz region. Memory Access analysis type also shows memory bandwidth in GB/s over time.

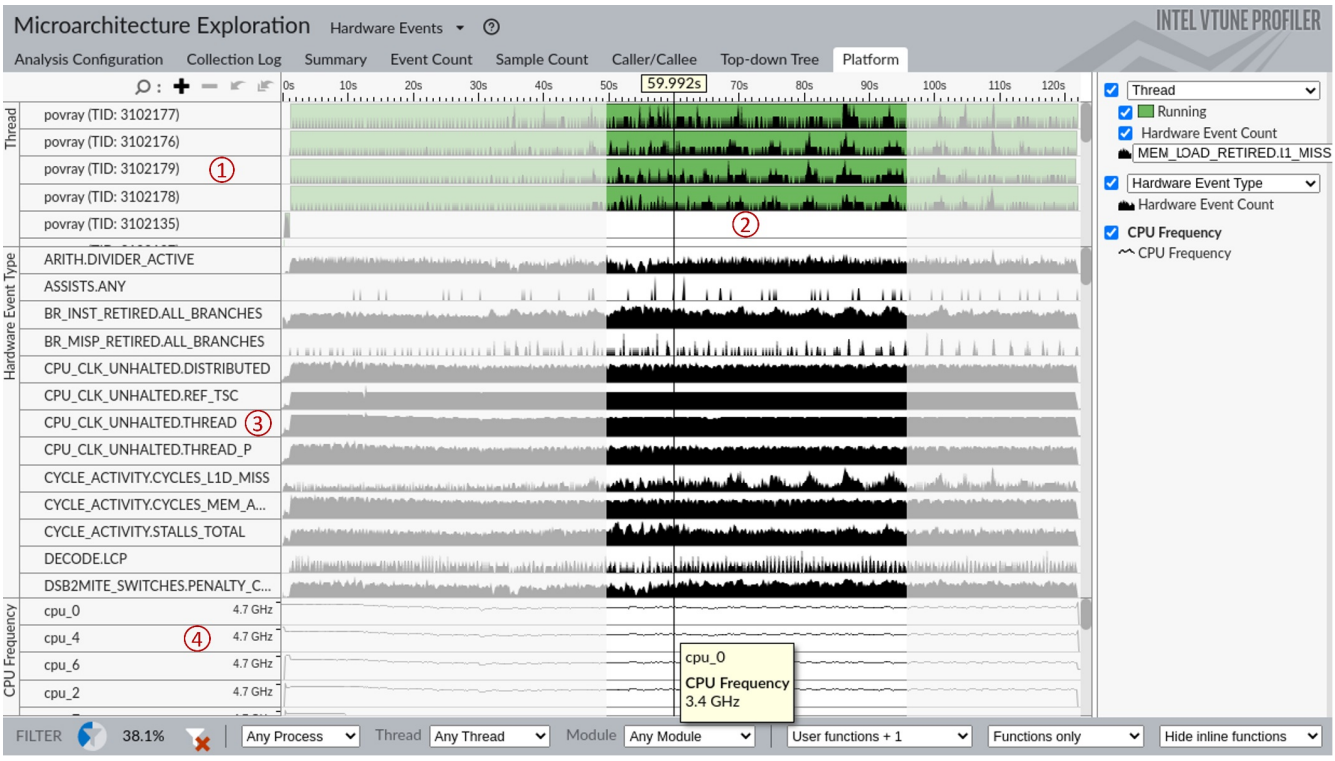


Figure 44: VTune's perf events timeline view of povray built-in benchmark.

TMA in Intel® VTune™ Profiler

TODO: this section needs to be updated (moved from chapter 7).

TMA is featured through the "Microarchitecture Exploration"[108] analysis in the latest Intel VTune Profiler. Figure 45 shows analysis summary for the 7-zip benchmark[109]. On the diagram, you can see that a significant amount of execution time was wasted due to CPU `Bad Speculation` and, in particular, due to mispredicted branches.

The beauty of the tool is that you can click on the metric you are interested in, and the tool will take you to the page showing the top functions contributing to that particular metric. For example, if you click on the `Bad Speculation` metric, you will see something like what is shown in Figure 46. [110]

From there, if you double click on the `LzmaDec_DecodeReal2` function, Intel® VTune™ Profiler will get you to the source level view like the one that is shown in Figure 47. The highlighted line contributes to the biggest number of branch mispredicts in the `LzmaDec_DecodeReal2` function.

7.2 AMD uProf

The uProf profiler is a tool developed by AMD for monitoring performance of applications running on AMD processors. While uProf can be used on Intel processors as well, you will be able to use only CPU-independent features. The profiler is available for free to download and can be used on Windows, Linux and FreeBSD. AMD uProf can be used for profiling on multiple virtual machines (VMs), including Microsoft Hyper-V, KVM, VMware ESXi, Citrix Xen, but not all features are available on all VMs. Also, uProf supports analyzing applications written in various languages, including C, C++, Java, .NET/CLR.

[108] VTune microarchitecture analysis - https://software.intel.com/en-us/vtune-help-general-exploration-analysis. In pre-2019 versions of Intel® VTune Profiler, it was called as "General Exploration" analysis.
[109] 7zip benchmark - https://github.com/llvm-mirror/test-suite/tree/master/MultiSource/Benchmarks/7zip.
[110] Per-function view of TMA metrics is a feature unique to Intel® VTune profiler.

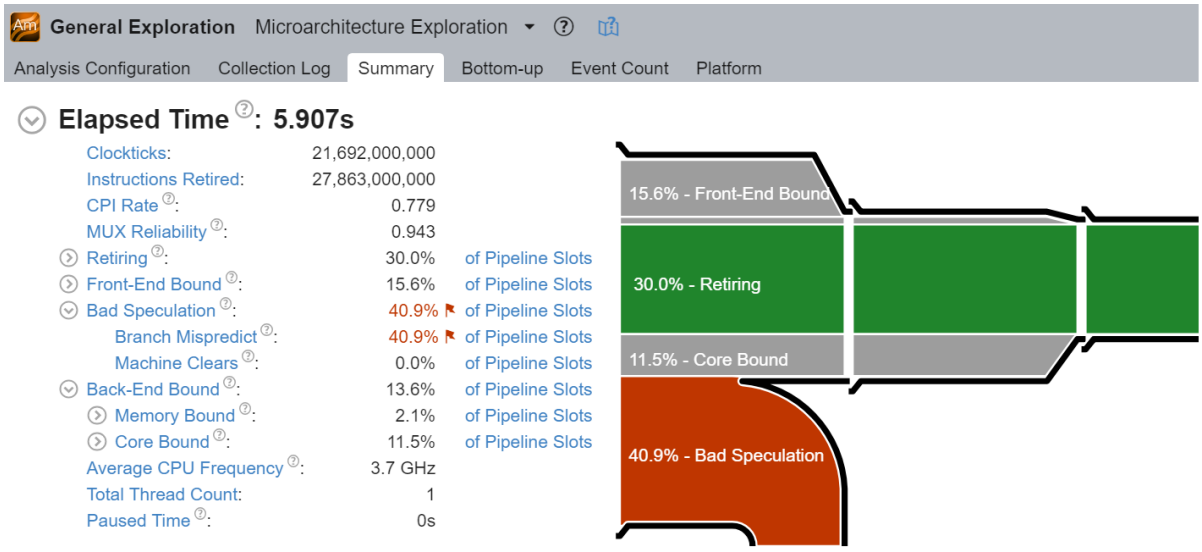


Figure 45: Intel VTune Profiler "Microarchitecture Exploration" analysis.

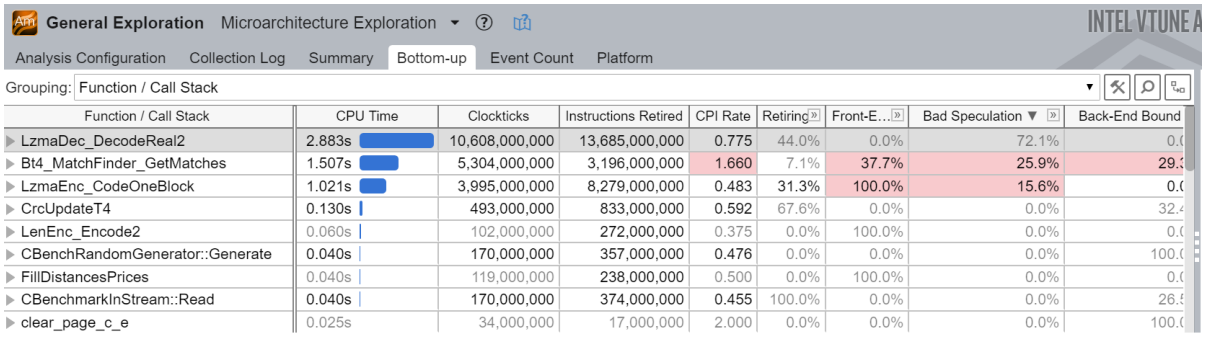


Figure 46: "Microarchitecture Exploration" Bottom-up view.

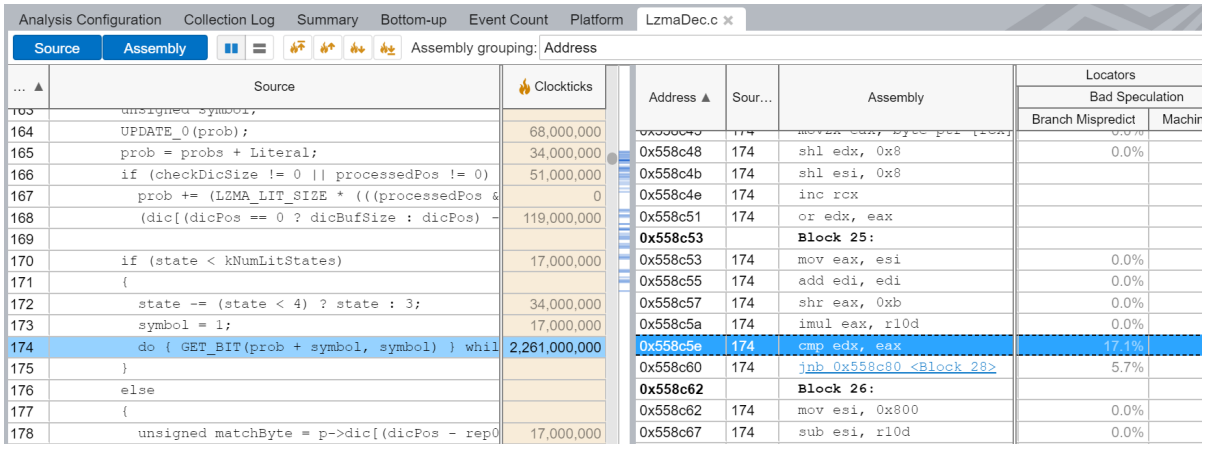


Figure 47: "Microarchitecture Exploration" source code and assembly view.

How to configure it

On Linux, uProf uses Linux perf for data collection. On Windows, uProf uses its own sampling driver that gets installed when you install uProf, no additional configuration is required. AMD uProf supports both command-line interface (CLI) and graphical interface (GUI). The CLI interface requires two separte steps - collect and report, similar to Linux perf.

What you can do with it:

- find hotspots: functions, statements, instructions.
- monitor various HW performance events and locate lines of code where these events happen.
- filter data for a specific function or thread.
- observe the workload behavior over time: view various performance events in timeline chart.
- analyze hot callpaths: call-graph, flame-graph and bottom-up charts.

Besides that uProf can monitor various OS events on Linux - thread state, thread synchronization, system calls, page faults, and others. It also allows to analyze OpenMP applications to detect thread imbalance, and analyze of MPI applications to detect the load imbalance among the nodes of MPI cluster. More details on various features of uProf can be found in the User Guide[111].

What you cannot do with it:

Due to the sampling nature of the tool, it will eventually miss events with a very short duration. The reported samples are statistically estimated numbers, which are most of the time sufficient to analyze the performance but not the exact count of the events.

Example

To demonstrate the look-and-feel of AMD uProf tool, we ran the dense LU matrix factorization component from the Scimark2[112] benchmark on AMD Ryzen 9 7950X, Windows 11, 64 GB RAM.

Figure 48 shows the function hotpots analysis. At the top of the image, you can see event timeline which shows the number of events observed at various times of the application execution. On the right, you can select which metric to plot, we selected `RETIRED_BR_INST_MISP`. Notice a spike in branch mispredictions in the time range from 20s to 40s. You can select this region to analyze closely what's going on there. Once you do that, it will update the bottom panels to show statistics only for that time interval.

Below the timeline graph, you can see a list of hot functions, along with corresponding sampled performance events and calculated metrics. Event counts can be viewed as: sample count, raw event count, and percentage. There are many interesting numbers to look at, however, we will not dive deep into the analysis, but readers are encouraged to figure out performance impact of branch mispredictions and find their source.

Below the functions table, you can see bottom-up callstack view for the selected funtion in the functions table. As we can see, the selected `LU_factor` function is called from `kernel_measureLU`, which in turn is called from `main`. In Scimark2 benchmark, this is the only call stack for `LU_factor`, even though is shows `Call Stacks [5]`, this is an artifact of collection that can be ignored. But in other applications, hot function can be called from many different places, so you would want to examine other call stacks as well.

If you double-click on any function, uProf will open the source/assembly view for that function. We don't show this view for brevity. On the left panel, there are other views available, like Metrics, Flame Graph, Call Graph view, and Thread Concurrency. They are useful for analysis as well, however we decided to skip them. Readers can experiment and look at those views on their own.

7.3 Apple Xcode Instruments

The most convinient way to do the initial performance analisys on MacOS is to use Instruments. It is an application performance analyzer and visualizer, that comes for free with Xcode. Instruments is built on top of the DTrace tracing framework that was ported to MacOS from Solaris. Instruments has many tools to inspect performance of

[111] AMD uProf User Guide - https://www.amd.com/en/developer/uprof.html#documentation
[112] Scimark2 - https://math.nist.gov/scimark2/index.html

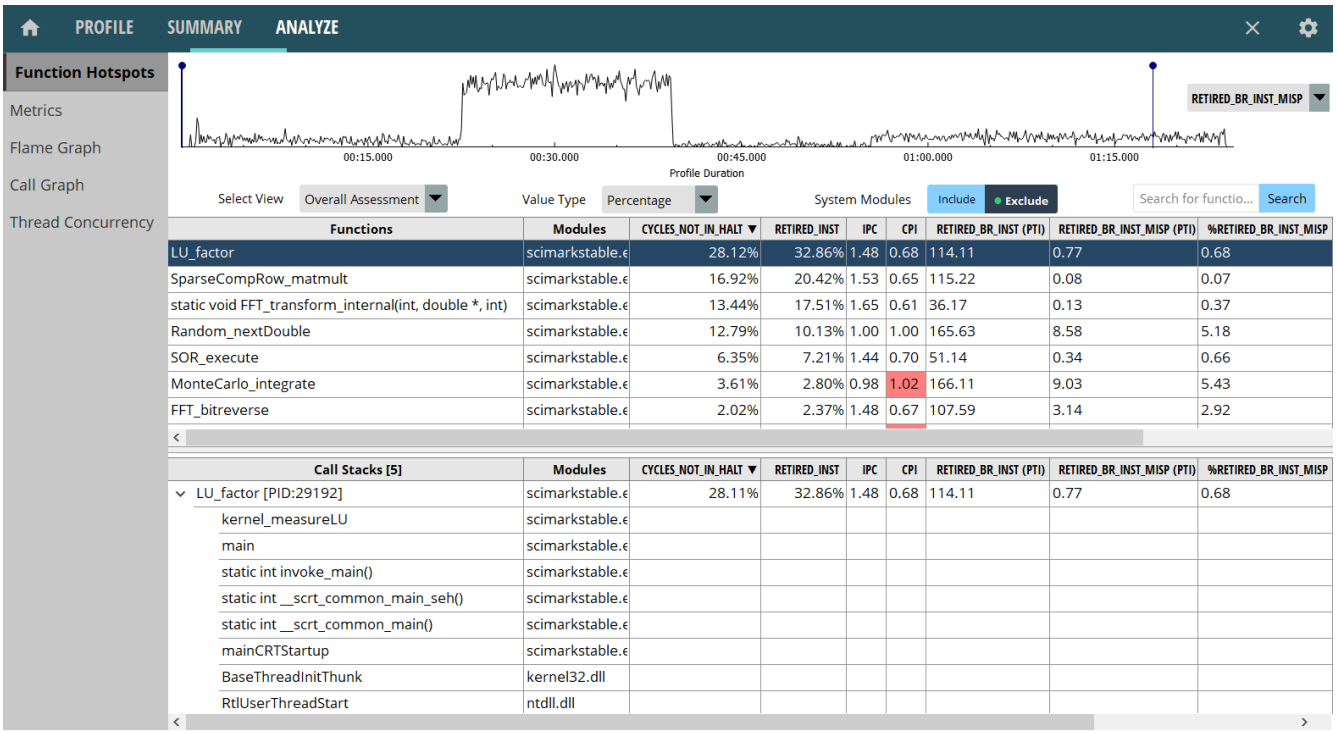


Figure 48: uProf's Function Hotspots view.

an application and allows us to do most of the basic things that other profilers like Intel Vtune can do. The easiest way to get the profiler is to install Xcode from the Apple AppStore. The tool requires no configuration, once you install it you're ready to go.

In Instruments, you use specialized tools, known as instruments, to trace different aspects of your apps, processes, and devices over time. Instruments has a powerfull visualization mechanism. It collects data as it profiles, and presents the results to you in real time. You can gather different types of data and view them side by side which allows you to see patterns in the execution, correlate system events and find very subtle performance issues.

In this chapter we will only showcase the "CPU Counters" instrument, which is the most relevant for this book. Instruments can also visualize GPU, network and disk activity, track memory allocations and releases, capture user events, such as mouse clicks, provide insights into power efficiency, and more. Read more about those use cases in the Instruments documentation[113].

What you can do with it:

- access HW performance counters on Apple M1 and M2 processors
- find hotspots in a program along with call stacks
- inspect generated ARM assembly code side-by-side with the source code
- filter data for a selected interval on the timeline

What you cannot do with it:

Example: Profiling Clang Compilation

As we advertised, in this example we will show how to collect HW performance counters on Apple Mac mini with the M1 processor inside, macOS 13.5.1 Ventura, 16 GB RAM. We took one of the largest file in the LLVM codebase and profile its compilation using Clang C++ compiler, version 15.0. Here is the command line that we will profile:

```
$ clang++ -O3 -DNDEBUG -arch arm64 <other options ...> -c
    llvm/lib/Transforms/Vectorize/LoopVectorize.cpp
```

[113] Instruments documentation - https://help.apple.com/instruments/mac/current

To begin, open Instruments and choose "CPU Counters" analysis type. (Here we need to jump ahead a little bit). It will open the main timeline view shown in Figure 50, ready to start profiling. But before we start, let's configure the collection. Click and hold the red target icon (1), then select "Recording Options..." menu. It will display the dialog window shown in Figure 49. This is where you can add HW performance monitoring events for collection.

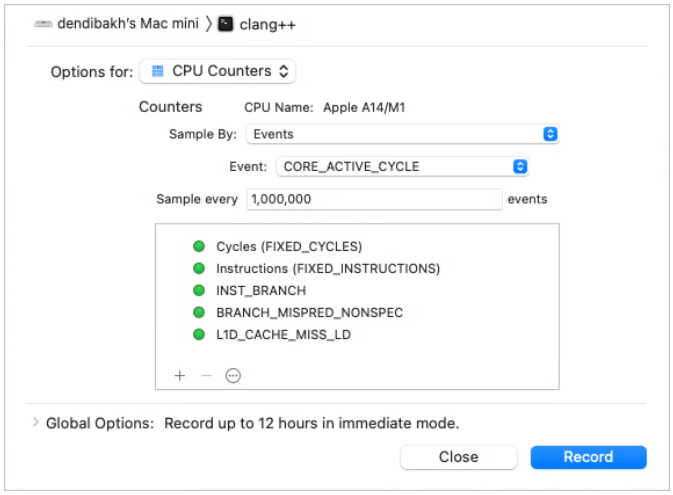


Figure 49: Xcode Instruments: CPU Counters options.

To the best of our knowledge, Apple doesn't document online their HW performance monitoring events, but they provide a list of events with some minimal description in `/usr/share/kpep`. There are `plist` files that you can convert into json. For example, for the M1 processor, one can run:

```
$ plutil -convert json /usr/share/kpep/a14.plist -o a14.json
```

and then open `a14.json` using a simple text editor.

The second step is to set the profiling target. To do that, click and hold the name of an application (marked (2) on Figure 50) and choose the one you're interested in, set the arguments and environment variables if needed. Now, you're ready to start the collection, press the red target icon (1).

Instruments shows a timeline and constantly updates statistics about the running application. Once the program finishes, Instruments will display the image similar to Figure 50. The compilation took 7.3 seconds and we can see how the volume of events changed over time. For example, branch mispredictions become more pronounced towards the end of the runtime. You can zoom in to that interval on the timeline to examine the functions involved.

The bottom panel shows numerical statistics. To inspect the hotspots similar to Intel Vtune's bottom-up view, select "Profile" in the menu (3), then click the "Call Tree" menu (4) and check the "Invert Call Tree" box. This is exactly what we did on Figure 50.

Instruments show raw counts along with the percentages of the total, which is useful if you want to calculate secondary metrics like IPC, MPKI, etc. On the right side, we have the hottest call stack for the function `llvm::FoldingSetBase::FindNodeOrInsertPos`. If you double-click on a function, you can inspect ARM assembly instructions generated for the source code.

To the best of our knowledge, there are no alternative profiling tools of similar quality available on MacOS platforms. Power users could use the `dtrace` framework itself by writing short (or long) command-line scripts, but it is beyond the scope of this book.

7.4 Linux Perf

Linux Perf is probably the most used performance profiler in the world, due to the fact that it is available on most Linux distributions, which makes it accessible for a wide range of users. Perf is natively supported in many popular Linux distributions, including Ubuntu, Red Hat, Debian, and many others. It is included in the kernel, so you can get OS-level statistic (page-faults, cpu-migrations, etc.) on any system that runs Linux. As of mid 2023,

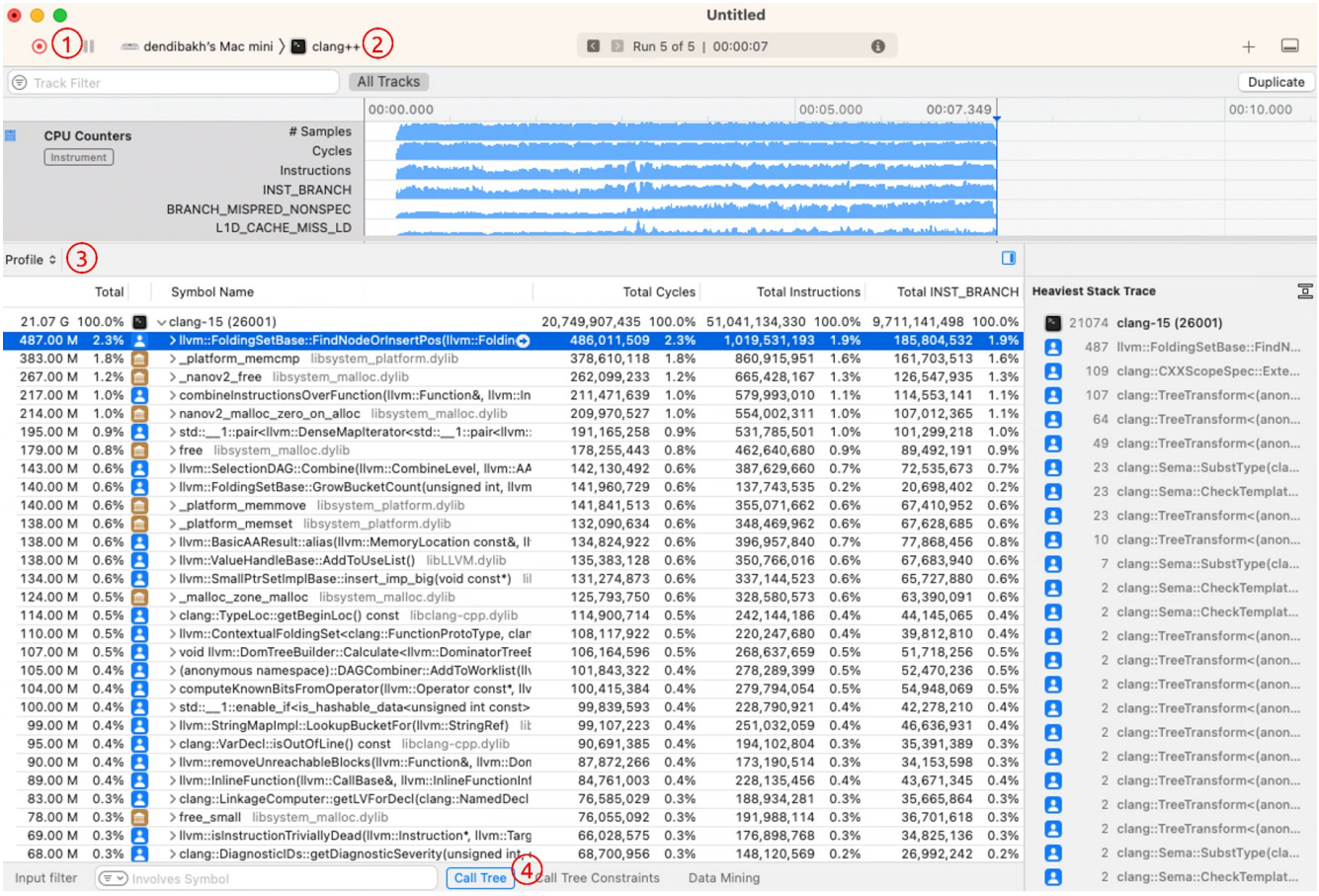


Figure 50: Xcode Instruments: timeline and statistics panels.

the profiler supports x86, ARM, PowerPC64, UltraSPARC, and a few other.[114] That allows to get access to the hardware performance monitoring features, for example, performance counters. More information about Linux `perf` is available on its wiki page[115].

How to configure it

Installing Linux perf is very simple and can be done with a single command:

```
$ sudo apt-get install linux-tools-common linux-tools-generic linux-tools-`uname -r`
```

Also, consider changing the following defaults unless security is a concern:

```
# Allow kernel profiling and access to CPU events for unprivileged users
$ echo 0 | sudo tee /proc/sys/kernel/perf_event_paranoid
$ sudo sh -c 'echo kernel.perf_event_paranoid=0 >> /etc/sysctl.d/local.conf'
# Enable kernel modules symbols resolution for unprivileged users
$ echo 0 | sudo tee /proc/sys/kernel/kptr_restrict
$ sudo sh -c 'echo kernel.kptr_restrict=0 >> /etc/sysctl.d/local.conf'
```

What you can do with it:

Generally, Linux `perf` can do most of the same things that other profilers can do. Hardware vendors prioretize enabling their features in Linux `perf`. so that by the time a new CPU is available on the market, `perf` already supports it. There are two main commands that most users will use: `perf stat` and `perf record + perf report`. First collects the absolute number of performance events in counting mode, second profiles an application or system in sampling mode.

The output of the `perf record` command is a raw dump of samples. Many tools are built on top of Linux `perf` that parse the dump file and provide new analysis types. Here are the most notable ones:

- Flame graphs, see next section.
- KDAB Hotspot,[116] a tool that visualizes Linux `perf` data with an interface very similar to Intel Vtune. If you worked in Intel Vtune, KDAB Hotspot will be very familiar to you. Some people use it as a drop-in replacement for Intel Vtune.
- Netflix Flamescope,[117] displays the heat map of sampled events over application runtime. You can observe different phases and patterns in the behavior of a workload. Netflix engineers found some very subtle performance bugs using this tool. Also, you can select a time range on the heat map and generate a flamegraph just for that time range.

What you cannot do with it:

Linux perf is a command-line tool and lacks GUI, which makes it hard to filter data, observe how the workload bahavior changes over time, zoom into a portion of the runtime, etc. There is a limited console output provided through `perf report` command, which is fine for quick analysis, although not as convenient as other GUI profilers. Luckily, as we just mentioned, there are GUI tools that can postprocess and visualize the raw output of Linux `perf`.

7.5 Flame Graphs

Flame graph is a popular way of visualizing the profiling data and the most frequent code-paths in the program. It allows us to see which function calls take the biggest portion of execution time. Figure 51 shows the example of a flame graph for the x264 video encoding benchmark, generated with open-source scripts[118] developed by Brendan Gregg. Nowadays, nearly all profilers can automatically generate a flame graph as long as the call stacks were collected during the profiling session.

[114] RISCV is not supported yet as a part of the official kernel, although custom tools from vendors exist.
[115] Linux perf wiki - https://perf.wiki.kernel.org/index.php/Main_Page.
[116] KDAB Hotspot - https://github.com/KDAB/hotspot.
[117] Netflix Flamescope - https://github.com/Netflix/flamescope.
[118] Flame Graphs by Brendan Gregg - https://github.com/brendangregg/FlameGraph

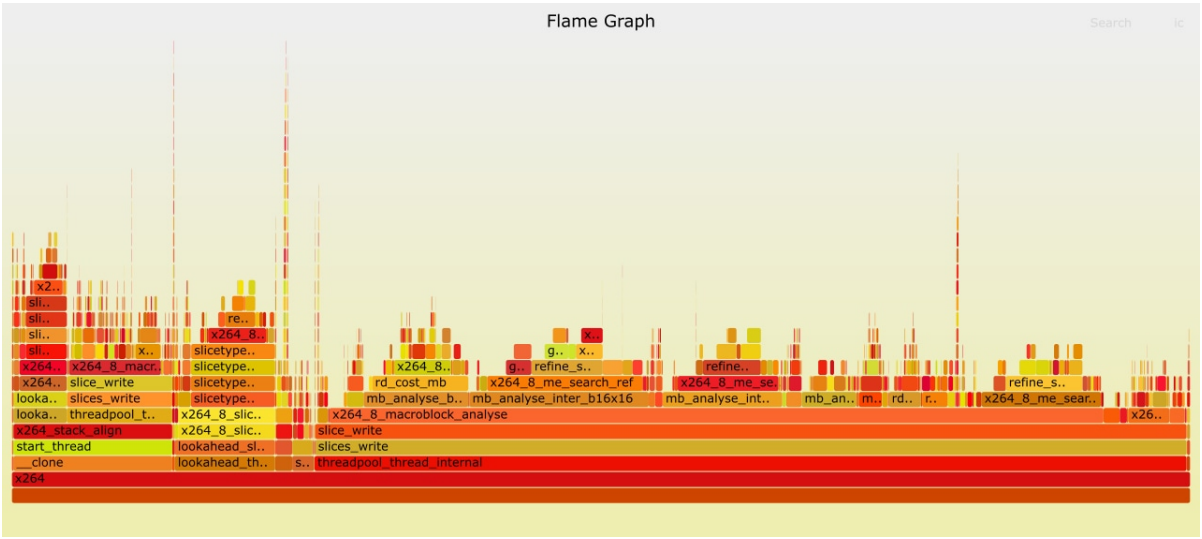


Figure 51: A Flame Graph for x264 benchmark.

On the flame graph, each rectangle (horizontal bar) represents a function call, the width of the rentangle indicates the relative execution time taken by the function itself and by its callees. The function calls happen from bottom to the top, so we can see that the hottest path in the program is `x264 -> threadpool_thread_internal -> ... -> x264_8_macroblock_analyse`. The function `threadpool_thread_internal` and its callees account for 74% of the time spent in the program. But the self time, i.e. time spent in the function itself is rather small. Similarly, we can do the same analysis for `x264_8_macroblock_analyse`, which accounts for 66% of the runtime. This visualization gives you a very good intuition on where the most time is spent.

Flame graphs are interactive, you can click on any bar on the image and it will zoom into that particular code path. You can keep zooming until you find a place that doesn't look according to your expectations or you reach a leaf/tail function - now you have actionable information you can use in your analysis. Another strategy is to figure our what is the hottest function in the program (not immediately clear from this flamegraph) and go bottom-up throught the flame graph, trying to understand from where this hottest function gets called.

7.6 Event Tracing for Windows

Microsoft has developed a system-wide tracing facility named Event Tracing for Windows (ETW). It was originally intended for helping device driver developers, but later found its use in analyzing general-purpose applications as well. ETW is available on all supported Windows platforms (x86, x64 and ARM) with the corresponding platform-dependent installation packages. ETW records structured events in user and kernel code with full call stack trace support which allows to observe SW dynamics in a running system and solve many challenging performance issues.

How to configure it

Recording ETW data is possible without any extra download since Windows 10 with `Wpr.exe`. But to enable system wide profiling you must be administrator and have the `SeSystemProfilePrivilege` enabled. The Windows Performance Recorder tool supports a set of built-in recording profiles which are OK for common performance issues. You can tailor your recording needs by authoring a custom performance recorder profile xml file with the `.wprp` extension.

If you want to not only record but also view the recorded ETW data you need to install the Windows Performance Toolkit (WPT). You can download the Windows Performance Toolkit from the Windows SDK[119] or ADK[120] download page. Windows SDK is huge, you don't necessarily need all its parts. In our case we just enabled the checkbox of the Windows Performance Toolkit. You are allowed to redistribute WPT as a part of your own application.

[119] Windows SDK Downloads https://developer.microsoft.com/en-us/windows/downloads/sdk-archive/
[120] Windows ADK Downloads https://learn.microsoft.com/en-us/windows-hardware/get-started/adk-install#other-adk-downloads

What you can do with it:

- Look at CPU hotspots with a configurable CPU sampling rate from 125 microseconds up to 10 seconds. Default is 1 millisecond which costs approximately 5-10% runtime overhead.
- Who blocks a certain thread and for how long (e.g. late event signals, unnecessary thread sleeps, etc).
- Examine how fast a disk serves read/write requests and who initiates that work.
- Check file access performance and patterns (includes cached read/writes which lead to no disk IO).
- Trace the TCP/IP stack how packets flow between network interfaces and computers.

All the items listed above are recorded system wide for all processes with configurable call stack traces (kernel and user mode call stacks are combined). It's also possible to add your own ETW provider to correlate the system wide traces with your application behavior. You can extend the amount of data collected by instrumenting your code. For example, you can add inject enter/leave ETW tracing hooks in functions in your source code to measure how often a certain method was executed.

What you cannot do with it:

- Examine CPU microarchitectural bottlenecks. For that, use vendor-specific tools like Intel VTune, AMD uProf, Apple Instruments, etc.

ETW traces capture dynamics of all processes at the system level which is great, but it may generate a lot of data. For example, capturing thread context switching data to observe various waits and delays can easily generate 1-2 GB per minute. That's why it is not practical to record high volume events for hours without overriding previously stored traces.

Tools to Record ETW traces

Here is the list of tools one can use to capture ETW traces:

- `wpr.exe`: a command line recording tool, part of Windows 10 and Windows Performance Toolkit.
- `WPRUI.exe`: a simple UI for recording ETW data, part of Windows Performance Toolkit
- `xperf`: a command line predecessor of wpr, part of Windows Performance Toolkit.
- `PerfView`:[121] a graphical recording and analysis tool with the main focus on .NET Applications. Open-source by Microsoft.
- `Performance HUD`:[122] a little known but very powerful GUI tool to track UI delays, User/Handle leaks via live ETW recording all unbalanced resource allocations with a live display of leaking/blocking call stack traces.
- `ETWController`:[123] a recording tool with the ability to record keyboard input and screenshots along with ETW data. Supports also distributed profiling on two machines simultaneously. Open-sourced by Alois Kraus.
- `UIForETW`:[124] a wrapper around xperf with special options to record data for Google Chrome issues. Can also record keyboard and mouse input. Open-sourced by Bruce Dawson.

Tools to View and Analyze ETW traces

- `Windows Performance Analyzer` (WPA): the most powerful UI for viewing ETW data. WPA can visualize and overlay Disk, CPU, GPU, Network, Memory, Process and many more data sources to get a holistic understanding how your system behaves and what it was doing. Although the UI is very powerful, it may also be quite complex for beginners. WPA supports plugins to process data from other sources, not just ETW traces. It's possible to import Linux/Android[125] profiling data that was generated by tools like Linux perf, LTTNG, Perfetto and the following log file formats: dmesg, Cloud-Init, WaLinuxAgent, AndoidLogcat.
- `ETWAnalyzer`:[126] reads ETW data and generates aggregate summary JSON files which can be queried, filtered and sorted at command line or exported to a CSV file.
- `PerfView`: mainly used to troubleshoot .NET applications. The ETW events fired for Garbage Collection and JIT compilation are parsed and easily accessible as reports or CSV data.

[121] PerfView https://github.com/microsoft/perfview
[122] Performance HUD https://www.microsoft.com/en-us/download/100813
[123] ETWController https://github.com/alois-xx/etwcontroller
[124] UIforETW https://github.com/google/UIforETW
[125] Microsoft Performance Tools Linux / Android https://github.com/microsoft/Microsoft-Performance-Tools-Linux-Android
[126] ETWAnalyzer https://github.com/Siemens-Healthineers/ETWAnalyzer

Case Study - Slow Program Start

Next, we will take a look at the example of using ETWController to capture ETW traces and WPA to visualize them.

Problem statement: When double clicking on a downloaded executable in Windows Explorer it is started with a noticeable delay. Something seems to delay process start. What could be the reason for this? Slow disk?

Setup

- Download ETWController to record ETW data and screenshots.
- Download the latest Windows 11 Performance Toolkit[127] to be able to view the data with WPA. Make sure that the newer Win 11 `wpr.exe` comes first in your path by moving the install folder of the WPT before the `C:\\Windows\\system32` in the System Environment dialog. This is how it should look like:

```
C> where wpr
C:\Program Files (x86)\Windows Kits\10\Windows Performance Toolkit\wpr.exe
C:\Windows\System32\wpr.exe
```

Capture traces

- Start ETWController.
- Select the CSwitch profile to track thread wait times along with the other default recording settings. Keep the check boxes *"Record mouse clicks"* and *"Take cyclic screenshots"* enabled to be later able to navigate to the slow spots with the help of the screen shots. See Figure 52.
- Press *"Start Recording"*.
- Download some executable from the internet, unpack it and double click the executable to start it.
- After that you can stop profiling by pressing the *"Stop Recording"* button.

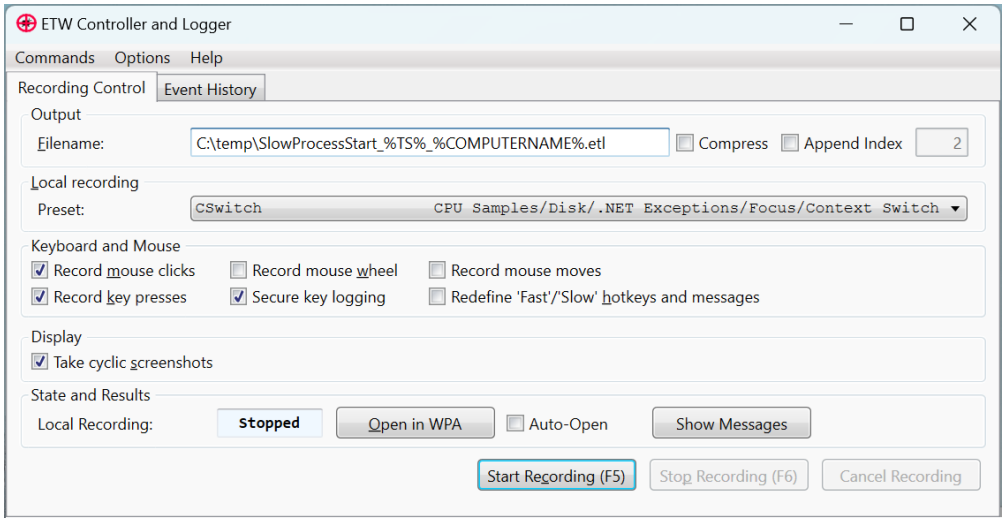


Figure 52: Starting ETW collection with ETWController UI.

Stopping profiling the first time takes a bit longer because for all managed code synthetic pdbs are generated which is a one time operation. After profiling has reached the Stopped state you can press the *"Open in WPA"* button to load the ETL file into the Windows Performance Analyzer with an ETWController supplied profile. The CSwitch profile generates a large amount of data which is stored in a 4 GB ring buffer which allows you to record 1-2 minutes before the oldest events are overwritten. Sometimes it is a bit of an art to stop profiling at the right time point. If you have sporadic issues you can keep recording enabled for hours and stop it when an event like a log entry in a file shows up, which is checked by a polling script, to stop profiling when the issue has occurred.

Windows supports Event Log and Performance Counter triggers which allow one to start a script when a performance counter reaches a threshold value or a specific event is written to an event log. If you need more sophisticated stop

[127] Windows SDK Downloads https://developer.microsoft.com/en-us/windows/downloads/sdk-archive/

triggers you should take a look at PerfView which allows one to define a Performance Counter threshold which must be reached and stay there for x seconds before profiling is stopped. This way random spikes are no longer triggering false positives.

Analysis in WPA Figure 53 shows the recorded ETW data opened in Windows Performance Analyzer (WPA). The WPA view is divided into three parts: *CPU Usage (Sampled)*, *Generic Events* and *CPU Usage (Precise)*. To understand the difference between them, let's dive deeper. The upper graph *CPU Usage (Sampled)* is useful for identifying where the CPU time is spent. The data is collected by sampling all the running threads at a regular time interval. Very similar to the hotspots view in other profiling tools.

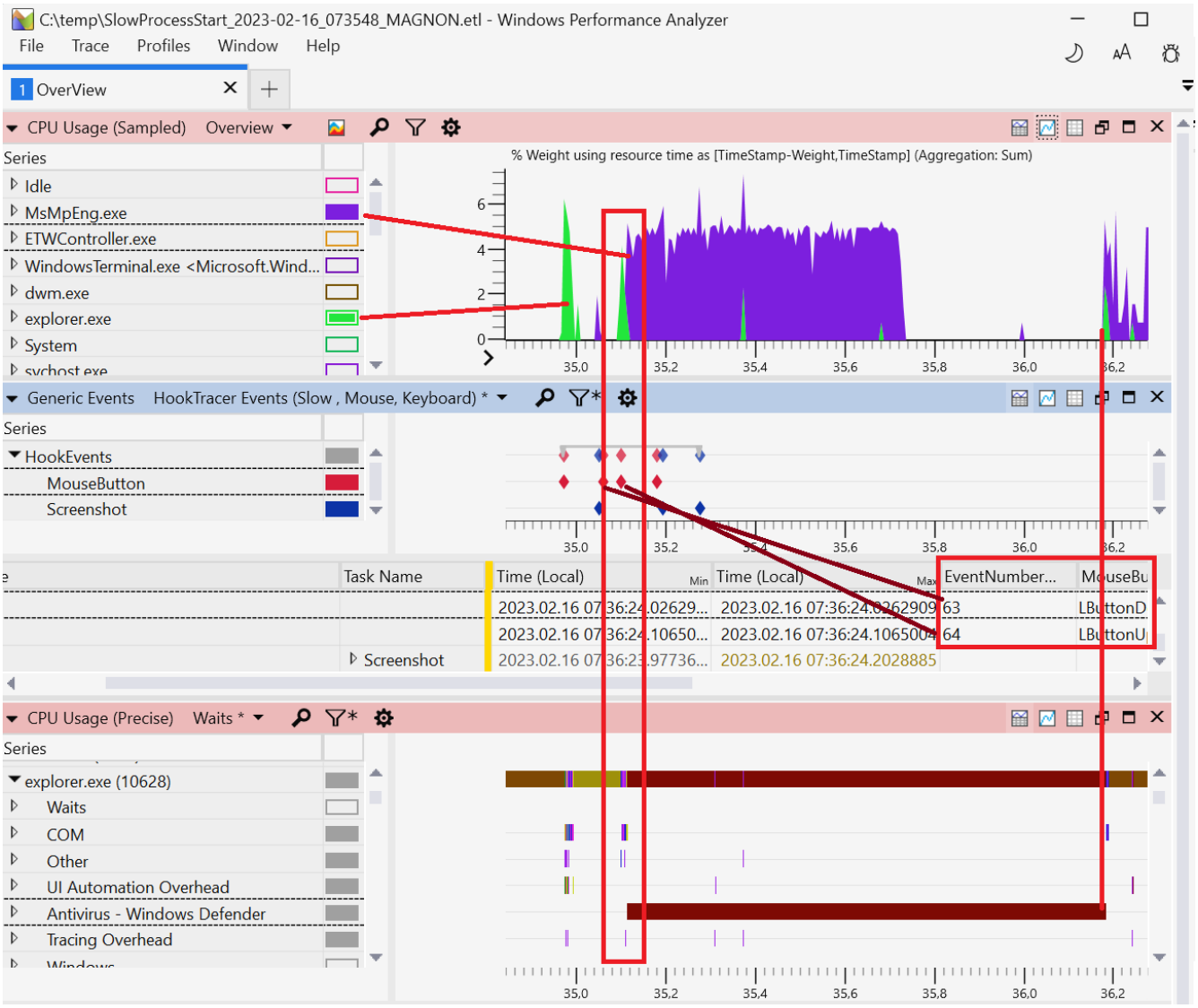


Figure 53: Windows Performance Analyzer overview of a slow start of an application.

Next comes *Generic Events* view which displays such events like mouse clicks and captured screenshots. Remember that we enabled interception of those events in the ETWController window. Because events are placed on the timeline, it is easy to correlate UI interactions with how the system reacts to them.

The bottom Graph *CPU Usage (Precise)* uses different source of data than *Sampled* view. While sampling data only captures running threads, *Precise* collection takes into account time intervals during which a process was not running. The data for precise view comes from the Windows Thread Scheduler. It traces how long and on which CPU a thread was running (CPU Usage), how long it was blocked in a kernel call (Wait), in which priority and how long the thread had been waiting for a CPU to become free (Ready Time), etc. Consequently, precise view

doesn't show the top CPU consumers. But this view is very helpful for understanding for how long and *why* a certain process was blocked.

Now that we familiarized ourselves with the WPA interface, let's observe the charts. First, we can find the `MouseButton` events 63 and 64 on the timeline. ETWController saves all the screenshots taken during collection in a newly created folder. The profiling data itself is saved in the file named `SlowProcessStart.etl` and there is a new folder named `SlowProcessStart.etl.Screenshots`. This folder contains the screenshots and a `Report.html` file which you can view in the browser. Every recorded keyboard/mouse interaction is saved in a file with the event number in its name, e.g. `Screenshot_63.jpg`. Figure 54 (cropped) displays the mouse double-click (events 63 and 64). The mouse pointer position is marked as a green square, except if a click event did occur, then it is red. This makes it easy to spot when and where a mouse click was performed.

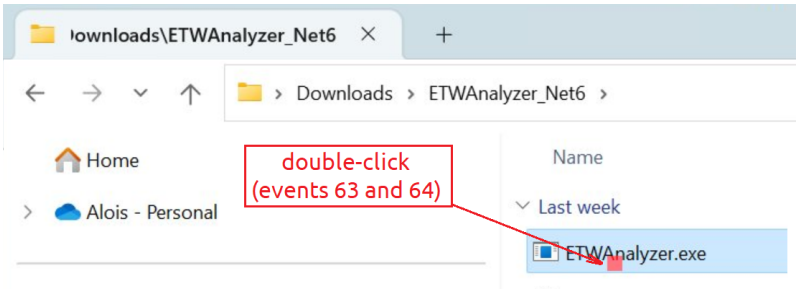


Figure 54: A mouse click screenshot captured with ETWController.

The double click marks the beginning of a 1.2 seconds delay when our application was waiting for something. At timestamp `35.1`, `explorer.exe` is active as it attempts to launch the new application. But then it wasn't doing much work and the application didn't start. Instead, `MsMpEng.exe` takes over the execution up until the time `35.7`. So far it looks like an antivirus scan before the downloaded executable is allowed to start. But we are not 100% sure that `MsMpEng.exe` is blocking the start of a new application.

Since we are dealing with delays, we are interested in wait times which are available on the *CPU Usage (Precise) Waits* Panel. There we find the list of processes that our `explorer.exe` was waiting for, visualized as a bar chart that aligns with the timeline on the upper panel. It's not hard to spot the long bar that corresponds to *Antivirus - Windows Defender* which accounts for a waiting time of 1.068s. So, we can conclude that the delay in starting our application is caused by Defender scanning activity. If you drill into the call stack (not shown), you'll see that `CreateProcess` system call is delayed in the kernel by `WDFilter.sys`, a Windows Defender Filter Driver. It blocks the process from starting until the potentially malicious file contents is scanned. Antivirus software can intercept everything, resulting in unpredictable performance issues that are difficult to diagnose without a comprehensive kernel view, such as with ETW. Mystery solved? Well, not just yet.

Knowing that Defender was the issue is just the first step. If you look at the top panel again, you'll see that the delay is not entirely caused by busy antivirus scanning. The `MsMpEng.exe` process was active from the time `35.1` till `35.7`, but the application didn't start immediately after that. There is an additional delay of 0.5 sec from the time `35.7` till `36.2`, during which the CPU was mostly idle, not doing anything. To root cause it, one needs to follow the thread wakeup history across processes, which we will not present here. In the end you would find a blocking web service call to `MpClient.dll!MpClient::CMpSpyNetContext::UpdateSpynetMetrics` which did wait for some Microsoft Defender web service to respond. If you enable additionally TCP/IP or socket ETW traces you can also find out to which remote endpoint Microsoft Defender was talking to. So, the second part of the delay is caused by the `MsMpEng.exe` process waiting for the network, which also blocked our application from running.

This case study shows only one example of what type of issue you can effectively analyze with WPA, but there are others. The WPA interface is very rich and highly customizable. It supports custom profiles to configure the graphs and tables for visualizing CPU, disk, files, etc. in the way you like best. Originally WPA was developed for device driver developers and there are built-in profiles which do not focus on application development. ETWController brings its own profile (*Overview.wpaprofile*) which you can set as default profile under *Profiles -> Save Startup Profile* to always use the performance overview profile.

7.7 Specialized and Hybrid profilers

Most of the tools explored so far fall under the category of sampling profilers. These are great when you want to identify hot-spots in your code, but in some cases they might not provide enough granularity for analysis. Depending on the profiler sampling frequency and the behavior of your program, most functions could be fast enough that they don't show up in a profiler. In some scenarios you might want to manually define which parts of your program need to be measured consistently. Video games, for instance, render frames (the final image shown on screen) on average at 60 frames per second (FPS); some monitors allow up to 144 FPS. At 60 FPS, each frame has as little as 16 milliseconds to complete the work before moving on to the next one. Developers pay particular attention to frames that go above this threshold, as this causes visible stutter in games and can ruin the player experience. This situation is hard to capture with a sampling profiler, as they usually only provide the total time taken for a given function.

Developers have created profilers that provide features helpful in specific environments, usually with a marker API that you can use to manually instrument your code. This allows you to observe performance of a particular function or a block of code (later refered as a *zone*). Continuing with the game industry, there are a few tools in this space: some are integrated directly into game engines like Unreal, while others are provided as external libraries and tools that can be integrated into your project. Some of the most commonly used profilers are Tracy, RAD Telemetry, Remotery, and Optick (Windows only). Next, we showcase Tracy,[128] as this seems to be one the more popular projects, however these concepts apply to the other profilers as well.

What you can do with Tracy:

- Debug performance anomalies in a program, e.g. slow frames.
- Correlate slow events with other events in a system.
- Find common characteristics among slow events.
- Inspect source code and assembly.
- Do "before-after" comparison after a code change.

What you cannot do with Tracy:

- Examine CPU microarchitectural issues, e.g. collect various performance counters.

Case Study: Analyzing Slow Frames with Tracy

In the example below, we use the ToyPathTracer[129] program, a simple path tracer, a technique similar to ray-tracing that shoots thousands of rays per pixel into the scene to render a realistic image. To process a frame, the implementation distributes the processing of each row of pixels to a separate thread.

To emulate a typical scenario where Tracy can help to root cause the problem, we manually modified the code such that some frames would consume more time than others. Listing 18 shows the sketch of the code along with added Tracy instrumentation. Notice, we randomly select frames to slow down. Also, we included Tracy's header and added `ZoneScoped` and `FrameMark` macros to the functions that we want to track. The `FrameMark` macro can be inserted to identify individual frames in the profiler. The duration of each frame will be visible on the timeline, which is very useful.

Each frame can contain many zones, designated by the `ZoneScoped` macro. Similar to frames, there are many instances of a zone. Every time we enter a zone, Tracy captures statistics for a new instance of that zone. The `ZoneScoped` macro creates an object on the stack that will record the runtime activity of the code within the scope of the object. Tracy refers to this scope as a "zone". At the zone entry, the current timestamp is captured. Once the function exits, the object will record a new timestamp and will store this timing data, along with the function name.

Tracy has two operation modes: it can store all the timing data until the profiler is connected to the application (the default mode), or it can only start recording when a profiler is connected. The latter option can be enabled by specifying the `TRACY_ON_DEMAND` pre-processor macro when compiling the application. This mode should be preferred if you want to distribute an application that can be profiled as needed. With this option, the tracing code can be compiled into the application and it will cause little to no overhead to the running program unless the profiler is attached. The profiler is a separate application that connects to a running application to capture and display the

[128] Tracy - https://github.com/wolfpld/tracy
[129] ToyPathTracer - https://github.com/wolfpld/tracy/tree/master/examples/ToyPathTracer

Listing 18 Tracy Instrumentation

```
#include "tracy/Tracy.hpp"

void TraceRowJob() {
  ZoneScoped;

  if (frameCount == randomlySelected)
    DoExtraWork();

  // ...
}

void RenderFrame() {
  ZoneScoped;
  for (...) {
    TraceRowJob();
  }
  FrameMark;
}
```

live profiling data, aka the "flight recorder" mode. The profiler can be run on a separate machine so that it doesn't interfere with the running application. Note, however, that this doesn't mean that the runtime overhead caused by the instrumentation code disappears - it is still there, but the overhead of visualizing the data is avoided in this case.

We used Tracy to debug the program and find the reason why some frames are slower than others. The data was captured on a Windows 11 machine, equipped with a Ryzen 7 5800X processor. The program was compiled with MSVC 19.36.32532. Tracy graphical interface is quite rich, unfortunately too hard to fit on a single screenshot, so we break it down into pieces. At the top, there is a timeline view as shown on Figure 55, cropped to fit onto the page. It shows only a portion of the frame #76, which took 44.1 ms to render. On that diagram, we see the `Main thread` and five `WorkerThread`s that were active during that frame. All threads, including the main thread, are performing work to advance progress in rendering the final image. As we said earlier, each thread processes a row of pixels inside the `TraceRowJob` zone. Each `TraceRowJob` zone instance contains many smaller zones, that are not visible. Tracy collapses inner zones and only shows the number of collapsed instances - this is what, for example, number `4,109` means under the first `TraceRowJob` in the Main Thread. Notice the instances of `DoExtraWork` zones, nested under `TraceRowJob` zones. This observation already can lead to a discovery, but in the real application it may not be so obvious. Let's leave this for now.

Right above the main panel, there is a histogram that displays the times for all the recorded frames, see Figure 56. It makes it easier to spot a long running frame that could cause stutter. It makes it easier to spot those frames that took longer than average to complete. In this example, most frames take around 33 ms (the yellow bars). However there are some frames that take longer than this and are marked in red. As seen on the screenshot, a tooltip showing the details of a given frame is displayed when hovering the mouse on the bar in the histogram. In this example, we are showing the details for the last frame, highlighted in green.

Figure 57 illustrates the CPU data section of the profiler. This area shows which core a given thread is executing on and it also displays context switches. This section will also display other programs that are running on the CPU. As seen in the image, the details for a given thread are displayed when hovering the mouse on a given section in the CPU data view. Details include the CPU the thread is running on, the parent program, the individual thread and timing information. We can see that the `TestCpu.exe` thread was active for 4.4 ms on CPU 1.

Next comes the panel that provides information on where our program spends its time, aka hotspots. Figure 58 captures the screenshot of the Tracy's statistics window. We can check the recorded data, including the total time a given function was active, how many times it was invoked, etc. It's also possible to select a time range in the main view to filter information that corresponds just to that time interval.

The last set of panels that we show, allow us to analyze individual zone instances in more depth. Once you click on any zone instance, say, on the main timeline view or on CPU data view, Tracy will open a Zone Info window

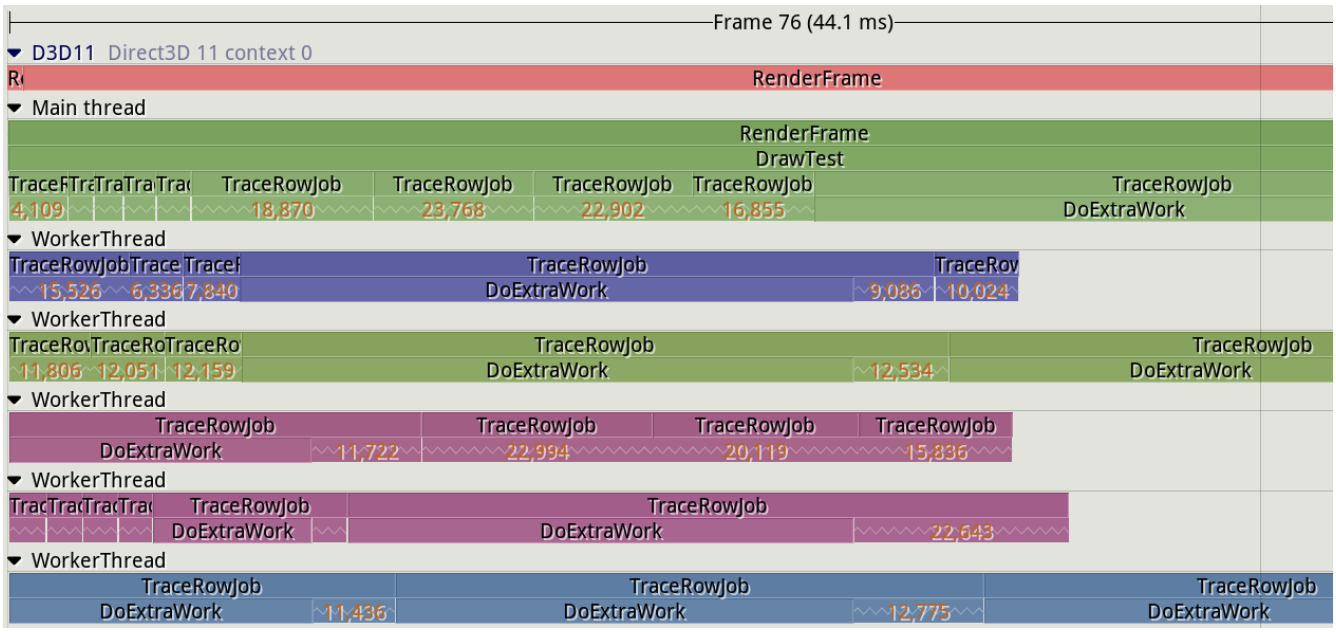


Figure 55: Tracy main timeline view.

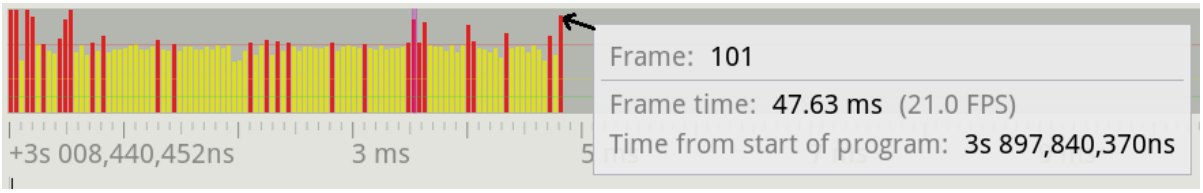


Figure 56: Tracy frame timings.

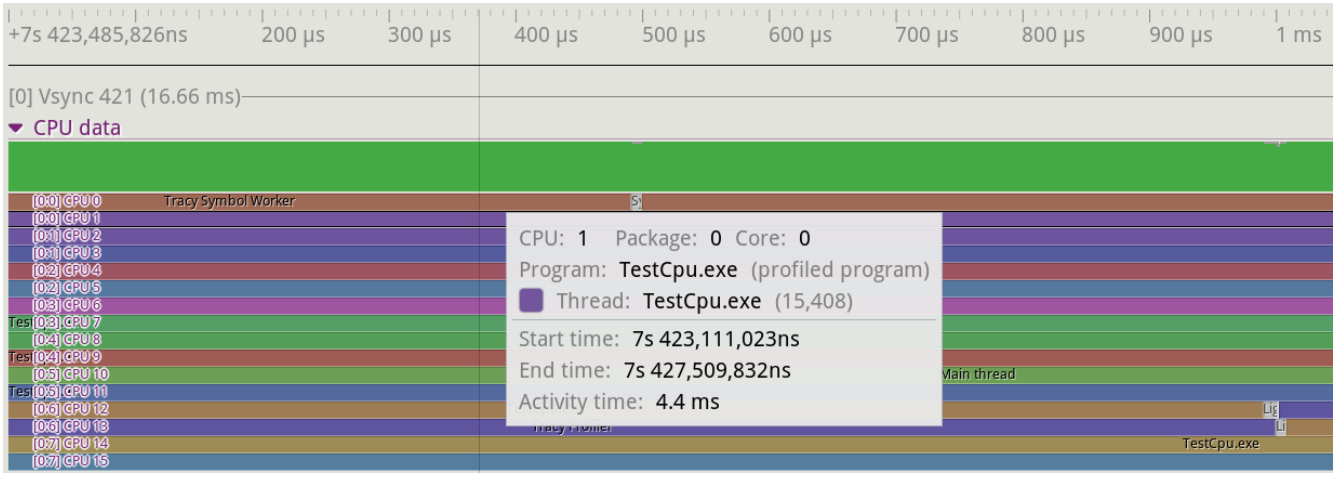


Figure 57: Tracy CPU data view.

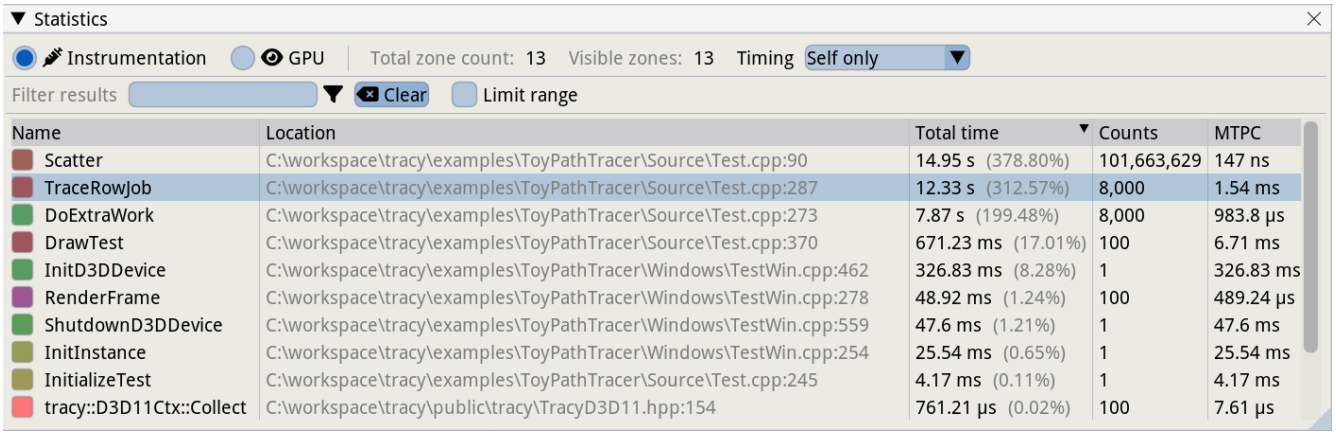


Figure 58: Tracy function statistics.

(see Figure 59, the left panel) with the details for this zone instance. It tells how much of the execution time is consumed by the zone itself or its children. In this example, execution of the `TraceRowJob` function took 19.24 ms, but the time consumed by the function itself without its callees takes 1.36 ms, which is only 7%. The rest of the time is consumed by the child zones.

It's easy to spot a call to `DoExtraWork` that takes the bulk of the time, 16.99 ms out 19.24 ms. Notice that this particular `TraceRowJob` instance runs 4.4 times longer than the average case (find "437.93% of the mean time" on the image). Bingo! We found one of the slow instances where `TraceRowJob` function was slowed down because of some extra work. One way to proceed would be to click on the `DoExtraWork` row to inspect this zone instance. This will update the Zone Info view with the details of the `DoExtraWork` instance so that we can dig down to understand what caused the performance issue. This view also shows the source file and line of code where the zone starts. So, another strategy would be to check the source code to understand why the current `TraceRowJob` instance takes more time than usual.

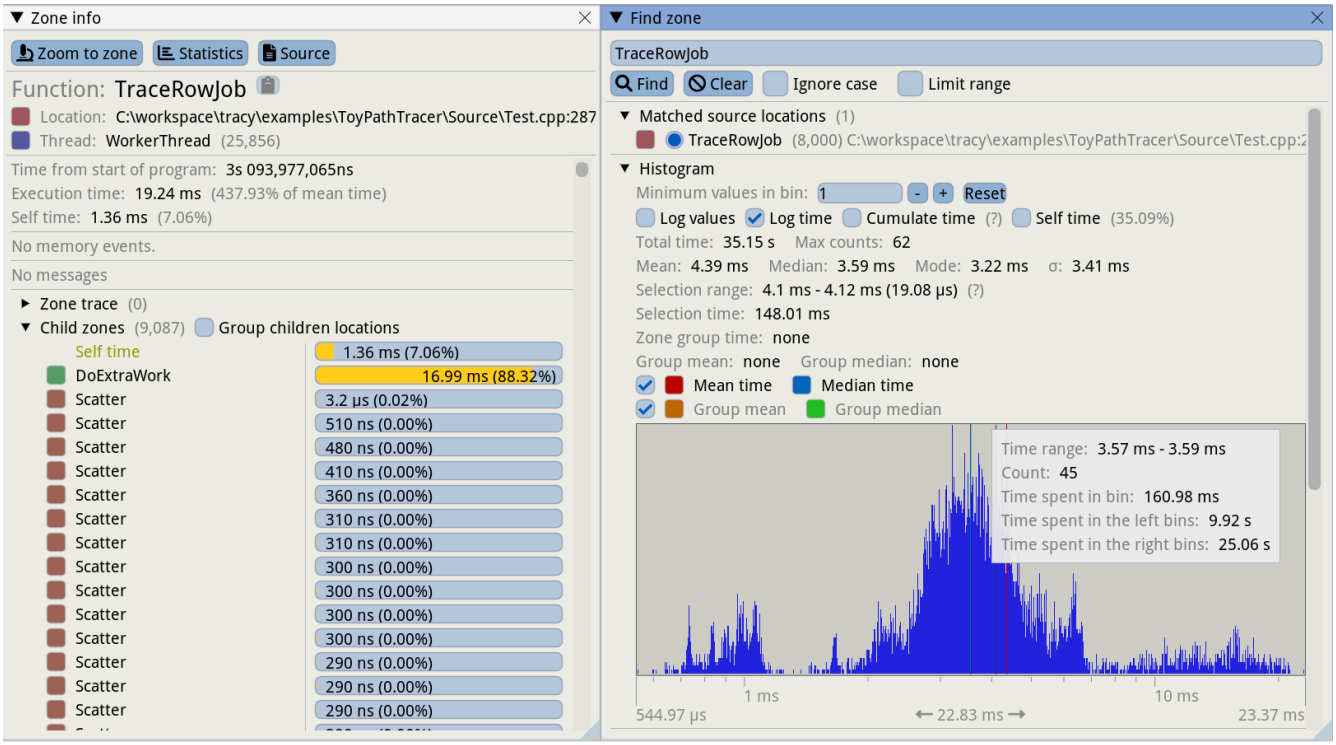


Figure 59: Tracy zone detail windows

Remember, we saw on Figure 56, that there are other slow frames. Let's see if this is the common problem among all the slow frames. If we click on the "Statistics" button, it will display the Find Zone panel (Figure 59, on the right). Here we can see the time histogram that aggregates all zone instances. This is particularly useful to determine how much variation there is when executing a function. Looking at the histogram on the right, we see that the median duration for the `TraceRowJob` function is 3.59 ms, with most calls taking between 1 and 7 ms. However there are a few instances that take longer than 10 ms, with a peak of 23 ms. Note that the time axis is logarithmic. The Find Zone window also provides other data points, including the mean, median and standard deviation for the inspected zone.

Now we can examine other slow instances to find what is common between them, which will help us to root cause the issue. From this view you can select one of the slow zones. This will update the Zone Info window (2) with the details of that zone instance and by clicking the "Zoom to zone" button, the main window will focus on this slow zone. From here we can check if the selected `TraceRowJob` instance has similar characteristics as the one that we just analyzed.

Other Features of Tracy

Tracy monitors performance of the whole system, not just the application itself. It also behaves like a traditional sampling profiler as it reports data for applications that are running concurrently to the profiled program. The tool monitors thread migration and idle time by tracing kernel context switches (administrator privileges are required). Zone statistics (call counts, time, histogram) are exact because Tracy captures every zone entry/exit, but system-level data and source-code-level data are sampled.

In the example in this section, we used manual markup of interesting areas in the code. But it's not a strict requirement to start using Tracy. You may profile an unmodified application and add instrumentation later when you know where it's needed. Tracy provides many other features, too many to cover in this overview. Here are some of the notable ones: - Tracking memory allocations and locks. - Session comparison. This is vital to ensure a change provides the expected benefits. It's possible to load two profiling sessions and compare zone data before and after the change was made. - Source code and assembly view. If debug symbols are available, Tracy can also display hotspots in the source code and related assembly just like Intel Vtune and other profilers.

In comparison with other tools like Intel Vtune and AMD uProf, with Tracy you cannot get the same level of CPU microarchitectural insights (e.g. various performance events). This is because Tracy does not leverage the HW features specific to a particular platform.

The overhead of profiling with Tracy depends on how many zones you have activated. The author of Tracy provides some data points that he measured on a program that does image compression: an overhead of 18% and 34% with two different compression schemes. A total of 200M zones were profiled, with an average overhead of 2.25 ns per zone. This test instrumented a very hot function. In other scenarios the overhead will be much lower. While it's possible to keep the overhead small, you need to be careful about which sections of code you want to instrument, especially if you decide to use it in production.

7.8 Continuous Profiling

In Chapter 6, we covered the various approaches available for conducting a performance analysis, including but not limited to instrumentation, tracing, and sampling. Among these three approaches, sampling imposes relatively minor runtime overhead and requires the least amount of upfront work while still offering valuable insight into application hotspots. But this insight is limited to the specific point in time when the samples are gathered – what if we could add a time dimension to this sampling? Instead of knowing that FunctionA consumes 30% of CPU cycles at one particular point in time, what if we could track changes in FunctionA's CPU usage over days, weeks, or months? Or detect changes in its stack trace over that same timespan, all in production? Continuous Profiling emerged to turn these ideas into reality.

Continuous Profiling (CP) is a systemwide, sample-based profiler that is always on, albeit at a low sample rate to minimize runtime impact. Continuously collecting data from all processes allows comparing why execution of code was different in time and debug incidents even after they have happened. CP tools provide valuable insights into what code uses the most resources, which allows engineers to reduce resource usage in their production environments and thus save money. Unlike typical profilers like Linux perf or Intel VTune, CP can pinpoint a performance

issue from the application stack down to the kernel stack from *any* given date and time, and supports call stack comparisons between any two arbitrary dates/times to highlight performance differences.

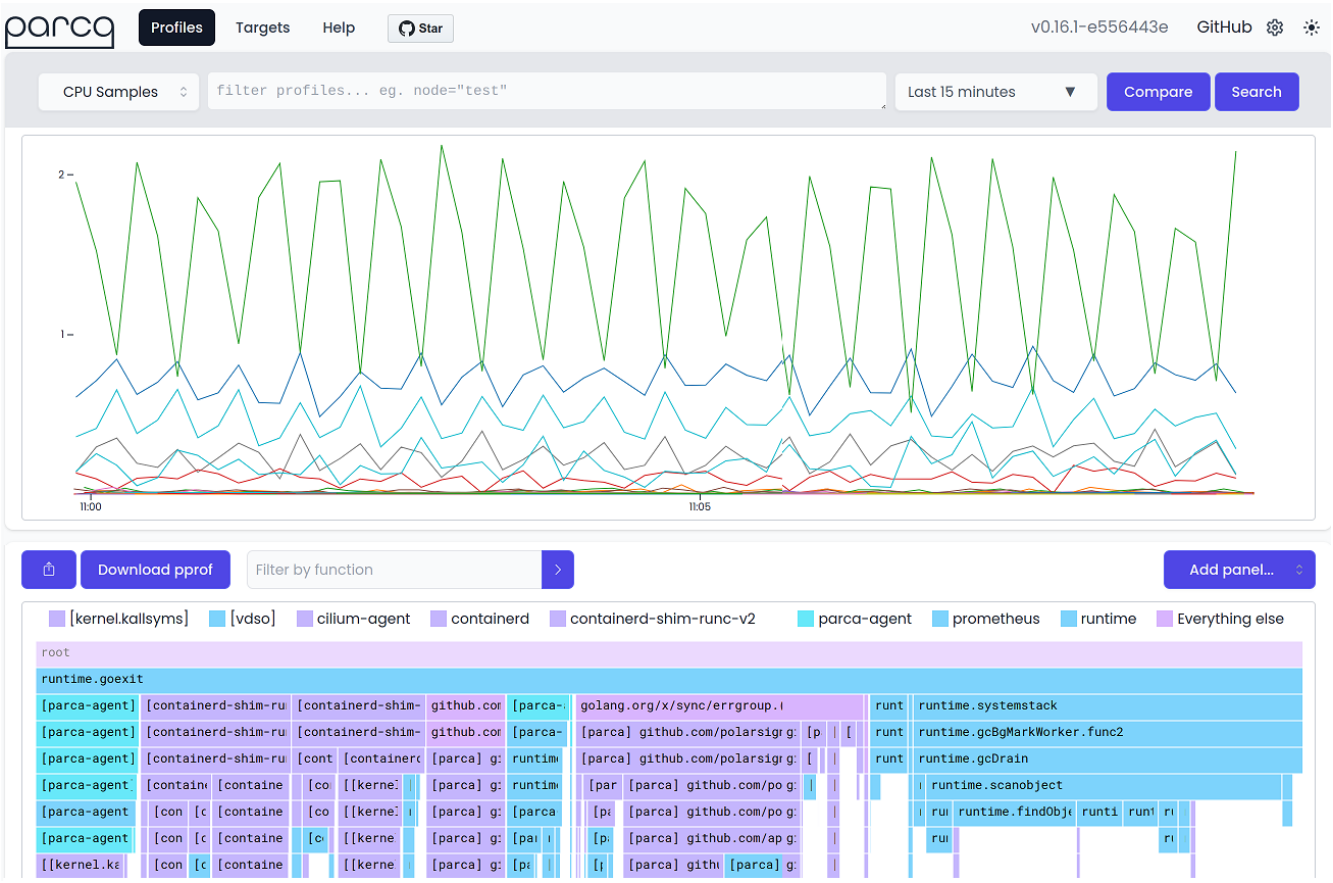


Figure 60: Screenshot of Parca Continuous Profiler Wb UI.

To showcase the look-and-feel of a typical CP, let's look at the Web UI of Parca,[130] one of the open-source CPs, depicted in Figure 60. The top panel displays a timeseries graph of the number of CPU samples gathered from various processes on the machine during the period selected from the time window dropdown list, which in this case is "Last 15 minutes", however, to make it fit on the page, the image was cut to show only the last 10 minutes. By default, Parca collects 19 samples per second. For each sample, it collects stack traces from all the processes that run on the host system. The more samples are attributed to a certain process, the more CPU activity it had during a period of time. In our example, you can see the hottest process (top line) had a bursty behavior with spikes and dips in CPU activity. If you were the lead developer behind this application you would probably be curious why this happens. When you roll out a new version of your application and suddenly see an unexpected spike in the CPU samples attributed to the process, that is an indication that something is going wrong.

Continuous profiling tools make it easier not only to spot the point of performance change, but also to root cause the issue. Once you click on any point of interest on the chart, the tool displays an icicle graph associated with that period in the bottom panel. An icicle graph is the upside-down version of a flamegraph. Using it, you can compare call stacks before and after and find what exactly is causing performance problems.

Imagine, you merged a code change into production and after it has been running for a while, you receive reports of intermittent response time spikes – these may or may not correlate with user traffic or with any particular time of day. This is an area where CP shines. You can pull up the CP Web UI and do a search for stack traces at the dates and times of those response time spikes, and then compare them to stack traces of other dates and times to identify anomalous executions at the application and/or kernel stack level. This type of "visual diff" is supported directly in the UI, like a graphical "perf diff" or a differential flamegraph.

[130] Parca - https://github.com/parca-dev/parca

Google introduced the concept in the 2010 paper "Google-Wide Profiling" [Ren et al., 2010], which championed the value of always-on profiling in production environments. However, it took nearly a decade before it gained traction in the industry:

1. In March 2019, Google Cloud released its Continuous Profiler
2. In July 2020, AWS released CodeGuru Profiler
3. In August 2020, Datadog released its Continuous Profiler
4. In December 2020, New Relic acquired the Pixie Continuous Profiler
5. In Jan 2021, Pyroscope released its open-source Continuous Profiler
6. In October 2021, Elastic acquired Optimize and its Continuous Profiler (Prodfiler); Polar Signals released its open-source Parca Continuous Profiler
7. In December 2021, Splunk releases its AlwaysOn Profiler
8. In March 2022, Intel acquired Granulate and its Continuous Profiler (gProfiler)

And new entrants into this space continue to pop up in both open-source and commercial varieties. Some of these offerings require more hand-holding than others. For example, some require source code or configuration file changes to begin profiling. Others require different agents for different language runtimes (e.g., ruby, python, golang, C/C++/Rust). The best of them have crafted secret sauce around eBPF so that nothing other than simply installing the runtime agent is necessary.

They also differ in the number of language runtimes supported, the work required for obtaining debug symbols for readable stack traces, and the type of system resources that can be profiled aside from CPU (e.g., memory, I/O, locking, etc.). While Continuous Profilers differ in the aforementioned aspects, they all share the common function of providing low-overhead, sample-based profiling for various language runtimes, along with remote stack trace storage for web-based search and query capability.

Where is Continuous Profiling headed? Thomas Dullien, co-founder of Optimyze which developed the innovative Continuous Profiler Prodfiler, delivered the Keynote at QCon London 2023 in which he expressed his wish for a cluster-wide tool that could answer the questions, "Why is this request slow?" or "Why is this request expensive?" In a multithreaded application, one particular function may show up on a profile as the highest CPU and memory consumer, yet its duties might be completely outside the application critical path, e.g. a house-keeping thread. Meanwhile, another function with such insignificant CPU execution time that it barely registers in a profile may exhibit an outsized effect on overall application latency and/or throughput. Typical profilers fail to address this shortcoming. And since CPs are basically profilers which run at all times, they inherit this same blind spot.

Thankfully, a new generation of CPs has emerged that employ AI with LLM-inspired architectures to process profile samples, analyze the relationships between functions, and finally pinpoint with high accuracy the functions and libraries which directly impact overall throughput and latency. One such company that offers this today is Raven.io. And as competition intensifies in this space, innovative capabilities will continue to grow so that CP tooling becomes as powerful and robust as that of typical profilers.

Questions and Exercises

1. Which tools you would use?

- scenario 1: client support team reports a customer issue: after upgrading to a new version of the application, performance of a certain operation drops by 10%.
- scenario 2: client support team reports a customer issue: some transactions take 2x longer time to finish than usual with no particular pattern.
- scenario 3: you're evaluating three different compression algorithms and you want to know what are the performance bottlenecks in each of them?
- scenario 4: there is a new shiny library that claims to be faster than the one you currently have integrated in your project; you've decided to compare their performance.

2. Run the application that you're working with on a daily basis. What is the most appropriate tool for performance analysis of your application according to the improvements you want to make? Practice using this tool.
3. Suppose you run multiple copies of the same program with different inputs on a single machine. Is it enough to profile one of the copies or you need to profile the whole system?

Chapter Summary

- We gave a quick overview of the most popular tools available on three major platforms: Linux, Windows and MacOS. Depending on a CPU vendor, the choice of a profiling tool will vary. For systems with an Intel processor we recommend using Vtune, for systems with an AMD processor use uProf, on Apple platforms use Xcode Instruments.
- Linux perf is probably the most frequently used profiling tool on Linux. It has support for processors from all major CPU vendors. It doesn't have a graphical interface, however, there are free tools that can visualize `perf` profiling data.
- We also discussed Windows Event Tracing (ETW), which is designed to observe SW dynamics in a running system. Linux has a similar tool called KUtrace,[131] which we do not cover here.
- Also, there are hybrid profilers that combine techniques like code instrumentation, sampling and tracing. This takes the best out of these approaches and allows user to get a very detailed information on a specific piece of code. In this chapter we looked at Tracy, which is quite popular among game developers.
- Continuous profilers already become an essential tool for monitoring performance in production environments. They collect system-wide performance metrics with call stacks for days, weeks or even months. Such tools make it easier to spot the point of performance change and root cause an issue.

[131] KUtrace - https://github.com/dicksites/KUtrace

Part2. Source Code Tuning

Welcome to the second part of this book where we will discuss various techniques for low-level source code optimization, aka *tuning*. In the first part, we learned how to find performance bottlenecks in the code, which is only half of the developer's job. Another half is to fix the problem.

Modern CPU is a very complicated device, and it's nearly impossible to predict how fast certain pieces of code will run. SW and HW performance depends on many factors, and the number of moving parts is too big for a human mind to overlook. Hopefully, observing how your code runs from a CPU perspective is possible thanks to all the performance monitoring capabilities we discussed in the first part of the book. We will extensively rely on methods and tools we learned about earlier in the book to guide our performance engineering process.

At a very high level, software optimizations can be divided into five categories.

- **Algorithmic optimizations**. Idea: analyze algorithms and data structures used in the program, and see if you can find better ones. Example: use quicksort instead of bubblesort.
- **Parallelizing computations**. Idea: if an algorithm is highly parallelizable, make the program threaded, or consider running it on a GPU. The goal is to do multiple things at the same time. Concurency is already used in all the layers of the HW and SW stacks. Examples: distribute the work across several threads, balance load between many servers in the data center, use async IO to avoid blocking while waiting for IO operations, keep multiple concurent network connections to overlap the request latency.
- **Eliminating redundant work**. Idea: don't do work that you don't need or have already done. Examples: leverage using more RAM to reduce the amount of CPU and IO you have to use (caching, memoization, look-up tables, compression), move loop invariant computations outside of the loop, pass a C++ object by reference to get rid of excessive copies caused by passing by value.
- **Batching**. Idea: aggregate multiple similar operations and do them in one go, thus reducing the overhead of repeating the action multiple times. Examples: send large TCP packets instead of many small ones, allocate large block of memory rather than allocating space for hundreds of tiny objects.
- **Ordering**. Idea: reorder the sequence of operations in an algorithm. Examples: change the data layout to enable sequential memory accesses, sort an array of C++ polymorphic objects based on their types to allow better prediction of virtual function calls, group hot functions together and place them closer to each other in a binary.

Many optimizations that we will discuss later in the book, fall under multiple categories. For example, we can say that vectorization is a combination of parallelizing and batching; loop blocking (tiling) is a manifestation of batching and eliminating redundant work.

To make the picture complete, let us also list other maybe obvious but still quite reasonable ways to speed up things:

- **Rewrite the code in another language**: if a program is written using interpreted languages (python, javascript, etc.), rewrite its performance-critical portion in a language with less overhead, e.g. C++, Rust, Go, etc.
- **Tune compiler options**: check that you use at least these three compiler flags: `-O3` (enables machine-independent optimizations), `-march` (enables optimizations for particular CPU architecture), `-flto` (enables inter-procedural optimizations). But don't stop here, there are many other options that affect performance. We will look at some of those in the future chapters. One may consider mining the best set of options for an application, commercial products that automate this process are available.
- **Optimize third-party SW packages**: the vast majority of software projects leverage layers of proprietary and open-source code. This includes OS, libraries, and frameworks. You can also seek improvements by replacing, modifying, or reconfiguring one of those pieces.
- **Buy faster hardware**: obviously, it's a business decision that comes with an associated cost, but sometimes it's the only way to improve performance when other options are already exhausted. It is much easier to justify the purchase when you identify performance bottlenecks in your application and communicate that clearly to the upper management. For example, once you find that memory bandwidth is limiting performance of your multithreaded program, you may suggest buying server motherboards and processors with more memory channels and DIMM slots.

Algorithmic Optimizations

Standard algorithms and data structures don't always work well for performance-critical workloads. For example, a linked list is pretty much deprecated in favor of "flat" data structures. Traditionally, every new node of a linked list is dynamically allocated. Besides potentially invoking many costly memory allocations, this will likely result in a situation where all the elements of the list are scattered in memory. Traversing such a data structure is not cache-friendly. Even though algorithmic complexity is still O(N), in practice, the timings will be much worse than of a plain array. Some data structures, like binary trees, have natural linked-list-like representation, so it might be tempting to implement them in a pointer chasing manner. However, more efficient "flat" versions of those data structures exist, see `boost::flat_map`, `boost::flat_set`.

When selecting an algorithm for a problem at hand, you might quickly pick the most popular option and move on... even though it could not be the best for your particular case. For example, you need to find an element in a sorted array. The first option that most developers consider is binary search, right? It is very well-known and is optimal in terms of algorithmic complexity, O(logN). Will you change your decision if I say that the array holds 32-bit integer values and the size of an array is usually very small (less than 20 elements)? In the end, measurements should guide your decision, but binary search suffers from branch mispredictions since every test of the element value has a 50% chance of being true. This is why on a small-sized array, linear scan is usually faster even though it has worse algorithmic complexity.

Data-Driven Optimizations

One of the most important techniques for tuning is called "Data-Driven" optimization that is based on introspecting the data that the program is working on. The approach is to focus on the layout of the data and how it is transformed throughout the program. A classic example of such an approach is Array-Of-Structures to Structure-Of-Array transformation, which is shown in Listing 19.

Listing 19 SOA to AOS transformation.

```
struct S {
  int a;
  int b;
  int c;
  // other fields
};
S s[N];    // AOS

<=>

struct S { // SOA
  int a[N];
  int b[N];
  int c[N];
  // other fields
};
```

The answer to the question of which layout is better depends on how the code is accessing the data. If the program iterates over the data structure and only accesses field `b`, then SOA is better because all memory accesses will be sequential. However, if a program iterates over the data structure and does *extensive* operations on all the fields of the object, then AOS may give better memory bandwidth utilization and in some cases, better performance. In the AOS scenario, members of the struct are likely to reside in the same cache line, and thus require fewer cache line reads and use less cache space. But more often, we see SOA gives better performance as it enables other important transformations, for example vectorization.

The main idea in Data-Driven Development (DDD), is to study how a program accesses data (how it is laid out in memory, observe access patterns), then modify the program accordingly (change the data layout, change the access patterns).

> **Personal Experience:** In fact, we can say that all optimizations are data-driven in a sense. Even the transformations that we will look at in the next sections are based on some feedback we receive from the execution of the program: function call counts, branch taken or not taken, performance counters, etc.

Another wide-spread example of DDD is "Small Size optimization". Its idea is to statically preallocate some amount of memory to avoid dynamic memory allocations. It is especially useful for small and medium-sized containers when the upper limit of elements can be well-predicted. Modern C++ STL implementations of `std::string` keep the first 15-20 characters in the buffer allocated on the stack and only allocate memory on the heap for longer strings. Another instances of this approach can be found in LLVM's `SmallVector` and Boost's `static_vector`.

Low-Level Optimizations

Performance engineering is an art. And like in any art, the set of possible scenarios is endless. It's impossible to cover all various optimizations one can imagine. The upcoming several chapters primarily address optimizations specific to modern CPU architectures.

Before we jump into particular source code tuning techinques, there are a few caution notes to make. First, avoid tuning bad code. If a piece of code has a high-level performance inefficiency, you shouldn't apply machine-specific optimizations to it. Always focus on fixing the major problem first. Only once you're sure that the algorithms and data structures are optimal for the problem you're trying to solve, try applying low-level improvements.

Second, remember that an optimization you implement might not be beneficial on every platform. For example, Loop Blocking depends on characteristics of the memory hierarchy in a system, especially L2 and L3 cache sizes. So, an algorithm tuned for a CPU with particular sizes of L2 and L3 caches might not work well for CPUs with smaller caches. It is important to test the change on the platforms your application will be running on.

The next four chapters are organized according to the TMA classification (see Section 6.1):

- Chapter 8. Optimizing Memory Accesses - `TMA:MemoryBound` category
- Chapter 9. Optimizing Computations - `TMA:CoreBound` category
- Chapter 10. Optimizing Branch Prediction - `TMA:BadSpeculation` category
- Chapter 11. Machine Code Layout Optimizations - `TMA:FrontEndBound` category

The idea behind this classification is to offer a checklist for developers when they are using TMA methodology in their performance engineering work. Whenever TMA attributes a performance bottleneck to one of the categories mentioned above, feel free to consult one of the corresponding chapters to learn about your options.

Chapter 14 covers other optimization areas that do not belong to any of the categories above. Chapter 15 addresses some common problems in optimizing multithreaded applications.

8 Optimizing Memory Accesses

Modern computers are still being built based on the classical Von Neumann architecture with decouples CPU, memory and input/output units. Operations with memory (loads and stores) account for the largest portion of performance bottlenecks and power consumption. It is no surprise that we start with this category first.

The statement that the memory hierarchy performance is very important is backed by Figure 61. It shows the growth of the gap in performance between memory and processors. The vertical axis is on a logarithmic scale and shows the growth of the CPU-DRAM performance gap. The memory baseline is the latency of memory access of 64 KB DRAM chips from 1980. Typical DRAM performance improvement is 7% per year, while CPUs enjoy 20-50% improvement per year.[Hennessy & Patterson, 2017]

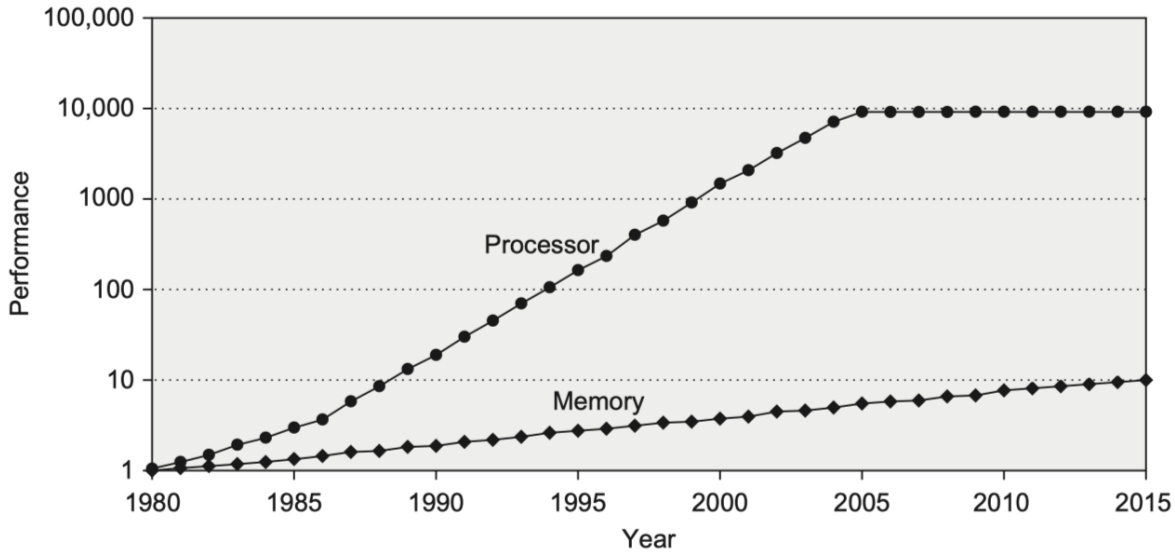


Figure 61: The gap in performance between memory and processors. © *Image from [Hennessy & Patterson, 2017].*

Indeed, a variable can be fetched from the smallest L1 cache in just a few clock cycles, but it can take more than three hundred clock cycles to fetch the variable from DRAM if it is not in the CPU cache. From a CPU perspective, a last level cache miss feels like a *very* long time, especially if the processor is not doing any useful work during that time. Execution threads may also be starved when the system is highly loaded with threads accessing memory at a very high rate and there is no available memory bandwidth to satisfy all loads and stores in a timely manner.

When an application executes a large number of memory accesses and spends significant time waiting for them to finish, such an application is characterized as being bounded by memory. It means that to further improve its performance, we likely need to improve how we access memory, reduce the number of such accesses or upgrade the memory subsystem itself.

In the TMA methodology, `Memory Bound` estimates a fraction of slots where the CPU pipeline is likely stalled due to demand for load or store instructions. The first step to solving such a performance problem is to locate the memory accesses that contribute to the high `Memory Bound` metric (see Section 6.1.1). Once guilty memory access is identified, several optimization strategies could be applied. Below we will discuss a few typical cases.

8.1 Cache-Friendly Data Structures

Writing cache-friendly algorithms and data structures, is one of the key items in the recipe for a well-performing application. The key pillar of cache-friendly code is the principles of temporal and spatial locality that we described in Section 3.6. The goal here is to allow required data to be fetched from caches efficiently. When designing cache-friendly code, it's helpful to think in terms of cache lines, not only individual variables and their location in memory.

8.1.1 Access Data Sequentially.

The best way to exploit the spatial locality of the caches is to make sequential memory accesses. By doing so, we allow the HW prefetcher (see Section 3.6.1.5.1) to recognize the memory access pattern and bring in the next chunk of data ahead of time. An example of a C-code that does such cache-friendly accesses is shown on Listing 20. The code is "cache-friendly" because it accesses the elements of the matrix in the order in which they are laid out in memory (row-major traversal[132]). Swapping the order of indexes in the array (i.e., `matrix[column][row]`) will result in column-major order traversal of the matrix, which does not exploit spatial locality and hurts performance.

Listing 20 Cache-friendly memory accesses.

```
for (row = 0; row < NUMROWS; row++)
  for (column = 0; column < NUMCOLUMNS; column++)
    matrix[row][column] = row + column;
```

The example presented in Listing 20 is classical, but usually, real-world applications are much more complicated than this. Sometimes you need to go an additional mile to write cache-friendly code. For instance, the standard implementation of binary search in a sorted large array does not exploit spatial locality since it tests elements in different locations that are far away from each other and do not share the same cache line. The most famous way of solving this problem is storing elements of the array using the Eytzinger layout [Khuong & Morin, 2015]. The idea of it is to maintain an implicit binary search tree packed into an array using the BFS-like layout, usually seen with binary heaps. If the code performs a large number of binary searches in the array, it may be beneficial to convert it to the Eytzinger layout.

8.1.2 Use Appropriate Containers.

There is a wide variety of ready-to-use containers in almost any language. But it's important to know their underlying storage and performance implications. A good step-by-step guide for choosing appropriate C++ containers can be found in [Fog, 2004, Chapter 9.7 Data structures, and container classes].

Additionally, choose the data storage, bearing in mind what the code will do with it. Consider a situation when there is a need to choose between storing objects in the array versus storing pointers to those objects while the object size is big. An array of pointers take less amount of memory. This will benefit operations that modify the array since an array of pointers requires less memory being transferred. However, a linear scan through an array will be faster when keeping the objects themselves since it is more cache-friendly and does not require indirect memory accesses.[133]

8.1.3 Packing the Data.

Memory hierarchy utilization can be improved by making the data more compact. There are many ways to pack data. One of the classic examples is to use bitfields. An example of code when packing data might be profitable is shown on Listing 21. If we know that `a`, `b`, and `c` represent enum values which take a certain number of bits to encode, we can reduce the storage of the struct `S` (see Listing 22).

Listing 21 Packing Data: baseline struct.

```
struct S {
  unsigned a;
  unsigned b;
  unsigned c;
}; // S is `sizeof(unsigned int) * 3` bytes
```

This greatly reduces the amount of memory transferred back and forth and saves cache space. Keep in mind that this comes with the cost of accessing every packed element. Since the bits of `b` share the same machine word with `a`

[132] Row- and column-major order - https://en.wikipedia.org/wiki/Row-_and_column-major_order.
[133] Blog article "Vector of Objects vs Vector of Pointers" by B. Filipek - https://www.bfilipek.com/2014/05/vector-of-objects-vs-vector-of-pointers.html.

Listing 22 Packing Data: packed struct.

```
struct S {
  unsigned a:4;
  unsigned b:2;
  unsigned c:2;
}; // S is only 1 byte
```

and `c`, compiler need to perform a `>>` (shift right) and `&` (AND) operation to load it. Similarly, `<<` (shift left) and `|` (OR) operations are needed to store the value back. Packing the data is beneficial in places where additional computation is cheaper than the delay caused by inefficient memory transfers.

Also, a programmer can reduce the memory usage by rearranging fields in a struct or class when it avoids padding added by a compiler (see example in Listing 23). The reason for a compiler to insert unused bytes of memory (pads) is to allow efficient storing and fetching of individual members of a struct. In the example, the size of `S1` can be reduced if its members are declared in the order of decreasing their sizes.

Listing 23 Avoid compiler padding.

```
struct S1 {
  bool b;
  int i;
  short s;
}; // S1 is `sizeof(int) * 3` bytes

struct S2 {
  int i;
  short s;
  bool b;
}; // S2 is `sizeof(int) * 2` bytes
```

8.1.4 Aligning and Padding.

Another technique to improve the utilization of the memory subsystem is to align the data. There could be a situation when an object of size 16 bytes occupies two cache lines, i.e., it starts on one cache line and ends in the next cache line. Fetching such an object requires two cache line reads, which could be avoided would the object be aligned properly. Listing 24 shows how memory objects can be aligned using C++11 `alignas` keyword.

Listing 24 Aligning data using the "alignas" keyword.

```
// Make an aligned array
alignas(16) int16_t a[N];

// Objects of struct S are aligned at cache line boundaries
#define CACHELINE_ALIGN alignas(64)
struct CACHELINE_ALIGN S {
  //...
};
```

A variable is accessed most efficiently if it is stored at a memory address, which is divisible by the size of the variable. For example, a double takes 8 bytes of storage space. It should, therefore, preferably be stored at an address divisible by 8. The size should always be a power of 2. Objects bigger than 16 bytes should be stored at an address divisible by 16. [Fog, 2004]

Alignment can cause holes of unused bytes, which potentially decreases memory bandwidth utilization. If, in the

example above, struct `S` is only 40 bytes, the next object of `S` starts at the beginning of the next cache line, which leaves `64 - 40 = 24` unused bytes in every cache line which holds objects of struct `S`.

Sometimes padding data structure members is required to avoid edge cases like cache contentions [Fog, 2004, Chapter 9.10 Cache contentions] and false sharing (see Section 13.7.3). For example, false sharing issues might occur in multithreaded applications when two threads, `A` and `B`, access different fields of the same structure. An example of code when such a situation might happen is shown on Listing 25. Because `a` and `b` members of struct `S` could potentially occupy the same cache line, cache coherency issues might significantly slow down the program. To resolve the problem, one can pad `S` such that members `a` and `b` do not share the same cache line as shown in Listing 26.

Listing 25 Padding data: baseline version.

```
struct S {
  int a; // written by thread A
  int b; // written by thread B
};
```

Listing 26 Padding data: improved version.

```
#define CACHELINE_ALIGN alignas(64)
struct S {
  int a; // written by thread A
  CACHELINE_ALIGN int b; // written by thread B
};
```

When it comes to dynamic allocations via `malloc`, it is guaranteed that the returned memory address satisfies the target platform's minimum alignment requirements. Some applications might benefit from a stricter alignment. For example, dynamically allocating 16 bytes with a 64 bytes alignment instead of the default 16 bytes alignment. To leverage this, users of POSIX systems can use `memalign`[134] API. Others can roll their own like described here[135].

One of the most important areas for alignment considerations is the SIMD code. When relying on compiler auto-vectorization, the developer doesn't have to do anything special. However, when you write the code using compiler vector intrinsics (see Section 9.5), it's pretty common that they require addresses divisible by 16, 32, or 64. Vector types provided by the compiler intrinsic header files are already annotated to ensure the appropriate alignment. [Fog, 2004]

```
// ptr will be aligned by alignof(__m512) if using C++17
__m512 * ptr = new __m512[N];
```

8.1.5 Dynamic Memory Allocation.

First of all, there are many drop-in replacements for `malloc`, which are faster, more scalable,[136] and address fragmentation[137] problems better. You can have a few percent performance improvement just by using a non-standard memory allocator. A typical issue with dynamic memory allocation is when at startup threads race with each other trying to allocate their memory regions at the same time.[138] One of the most popular memory allocation libraries are jemalloc[139] and tcmalloc[140].

Secondly, it is possible to speed up allocations using custom allocators, for example, arena allocators[141]. One of the main advantages is their low overhead since such allocators don't execute system calls for every memory allocation. Another advantage is its high flexibility. Developers can implement their own allocation strategies based on the

[134] Linux manual page for `memalign` - https://linux.die.net/man/3/memalign.
[135] Generating aligned memory - https://embeddedartistry.com/blog/2017/02/22/generating-aligned-memory/.
[136] Typical `malloc` implementation involves synchronization in case multiple threads would try to dynamically allocate the memory
[137] Fragmentation - https://en.wikipedia.org/wiki/Fragmentation_(computing).
[138] The same applies to memory deallocation.
[139] jemalloc - http://jemalloc.net/.
[140] tcmalloc - https://github.com/google/tcmalloc
[141] Region-based memory management - https://en.wikipedia.org/wiki/Region-based_memory_management

memory region provided by the OS. One simple strategy could be to maintain two different allocators with their own arenas (memory regions): one for the hot data and one for the cold data. Keeping hot data together creates opportunities for it to share cache lines, which improves memory bandwidth utilization and spatial locality. It also improves TLB utilization since hot data occupies less amount of memory pages. Also, custom memory allocators can use thread-local storage to implement per-thread allocation and get rid of any synchronization between threads. This becomes useful when an application is based on a thread pool and does not spawn a large number of threads.

8.1.6 Tune the Code for Memory Hierarchy.

The performance of some applications depends on the size of the cache on a particular level. The most famous example here is improving matrix multiplication with loop blocking (tiling). The idea is to break the working size of the matrix into smaller pieces (tiles) such that each tile will fit in the L2 cache.[142] Most of the architectures provide `CPUID`-like instruction,[143] which allows us to query the size of caches. Alternatively, one can use cache-oblivious algorithms[144] whose goal is to work reasonably well for any size of the cache.

Intel CPUs have a Data Linear Address HW feature (see Section 6.3.5) that supports cache blocking as described on an easyperf blog post[145].

8.2 Explicit Memory Prefetching

By now, you should know that memory accesses that are not resolved from caches are often very expensive. Modern CPUs try very hard to lower the penalty of cache misses if the prefetch request is issued sufficiently ahead of time. If the requested memory location is not in the cache, we will suffer the cache miss anyway as we have to go to the DRAM and fetch the data anyway. But if manage to bring that memory location in caches by the time the data is demanded by the program, then we effectively make the penalty of a cache miss to be zero.

Modern CPUs have two mechanisms for solving that problem: hardware prefetching and OOO execution. HW prefetchers help to hide the memory access latency by initiating prefetching requests on repetitive memory access patterns. While OOO engine looks N instructions into the future and issues loads early to allow smooth execution of future instructions that will demand this data.

HW prefetchers fail when data accesses patterns are too complicated to predict. And there is nothing SW developers can do about it as we cannot control the behavior of this unit. On the other hand, OOO engine does not try to predict memory locations that will be needed in the future as HW prefetching does. So, the only measure of success for it is how much latency it was able to hide by scheduling the load in advance.

Consider a small snippet of code in Listing 27, where `arr` is an array of one million integers. The index `idx`, which is assigned to a random value, is immediately used to access a location in `arr`, which almost certainly misses in caches as it is random. It is impossible for a HW prefetcher to predict as every time the load goes to a completely new place in memory. The interval from the time the address of a memory location is known (returned from the function `random_distribution`) until the value of that memory location is demanded (call to `doSomeExtensiveComputation`) is called *prefetching window*. In this example, the OOO engine doesn't have the opportunity to issue the load early since the prefetching window is very small. This leads to the latency of the memory access `arr[idx]` to stand on a critical path while executing the loop as shown in Figure 62. It's visible that the program waits for the value to come back (hatched fill rectangle) without making forward progress.

Listing 27 Random number feeds a subsequent load.

```
for (int i = 0; i < N; ++i) {
  size_t idx = random_distribution(generator);
  int x = arr[idx]; // cache miss
  doSomeExtensiveComputation(x);
}
```

[142] Usually, people tune for the size of the L2 cache since it is not shared between the cores.
[143] In Intel processors `CPUID` instruction is described in [Intel, 2023b, Volume 2]
[144] Cache-oblivious algorithm - https://en.wikipedia.org/wiki/Cache-oblivious_algorithm.
[145] Blog article "Detecting false sharing" - https://easyperf.net/blog/2019/12/17/Detecting-false-sharing-using-perf#2-tune-the-code-for-better-utilization-of-cache-hierarchy.

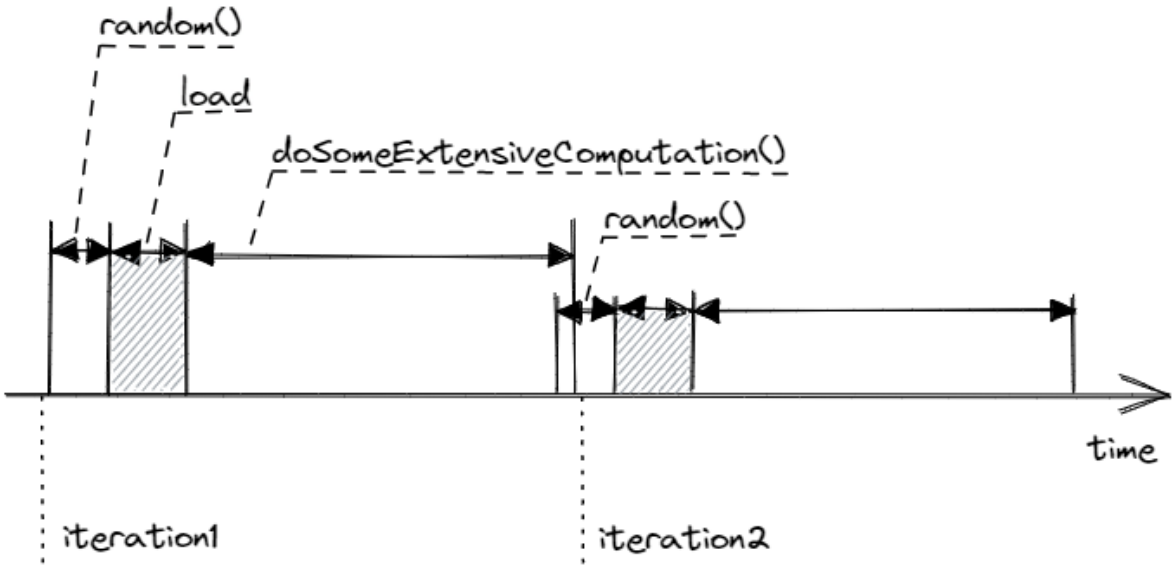


Figure 62: Execution timeline that shows the load latency standing on a critical path.

There is another important observation here. When a CPU gets close to finish running the first iteration, it speculatively starts executing instruction from the second iteration. It creates a positive overlap in the execution between iterations. However, even in modern processors, there are not enough OOO capabilities to fully overlap the latency of a cache miss with executing `doSomeExtensiveComputation` from the iteration1. In other words, in our case a CPU cannot look that far ahead of current execution to issue the load early enough.

Luckily, it's not a dead end as there is a way to speed up this code. To hide the latency of a cache miss, we need to overlap it with execution of `doSomeExtensiveComputation`. We can achieve it if we pipeline generation of random numbers and start prefetching the memory location for the next iteration as shown in Listing 28. Notice the usage of `__builtin_prefetch`,[146] a special hint that developers can use to explicitly request a CPU to prefetch a certain memory location. Graphical illustration of this transformation is illustrated in Figure 63.

Listing 28 Utilizing Exlicit Software Memory Prefetching hints.

```
size_t idx = random_distribution(generator);
for (int i = 0; i < N; ++i) {
  int x = arr[idx];
  idx = random_distribution(generator);
  // prefetch the element for the next iteration
  __builtin_prefetch(&arr[idx]);
  doSomeExtensiveComputation(x);
}
```

Another option to utilize explicit SW prefetching on x86 platforms is to use compiler intrinsics `_mm_prefetch` intrinsic. See Intel Intrinsics Guide for more details. In any case, compiler will compile it down to machine instruction: `PREFETCH` for x86 and `pld` for ARM. For some platforms compiler can skip inserting an instruction, so it is a good idea to check the generated machine code.

There are situations when SW memory prefetching is not possible. For example, when traversing a linked list, prefetching window is tiny and it is not possible to hide the latency of pointer chaising.

In Listing 28 we saw an example of prefetching for the next iteration, but also you may frequently encounter a need to prefetch for 2, 4, 8, and sometimes even more iterations. The code in Listing 29 is one of those cases,

[146] GCC builtins - https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html.

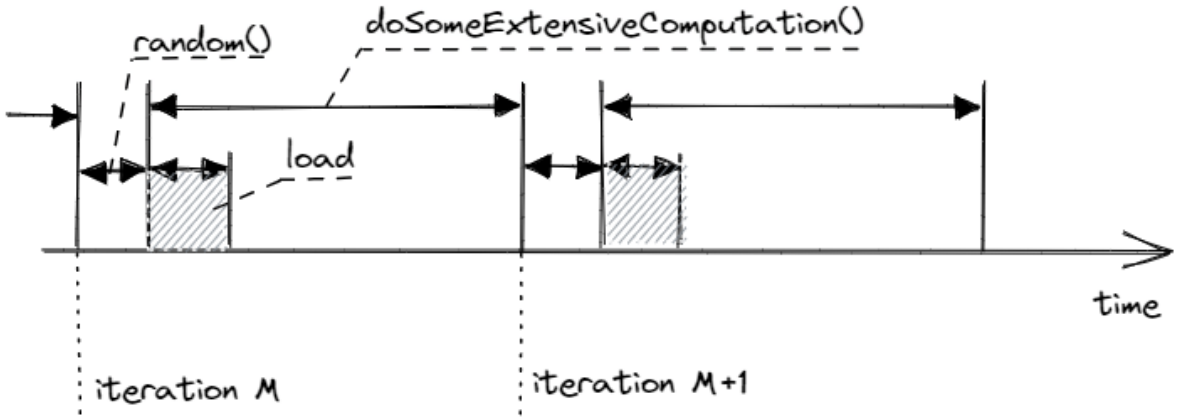


Figure 63: Hiding the cache miss latency by overlapping it with other execution.

when it could be beneficial. If the graph is very sparse and has a lot of verticies, it is very likely that accesses to `this->out_neighbors` and `this->in_neighbors` vectors will miss in caches a lot.

This code is different from the previous example as there are no extensive computations on every iteration, so the penalty of cache misses likely dominates the latency of each iteration. But we can leverage the fact that we know all the elements that will be accessed in the future. The elements of vector `edges` are accessed sequentially and thus are likely to be timely brought to the L1 cache by the HW prefetcher. Our goal here is to overlap the latency of a cache miss with executing enough iterations to completely hide it.

As a general rule, for prefetch hints to be effective, they must be inserted well ahead of time so that by the time the loaded value will be used in other calculations, it will be already in the cache. However, it also shouldn't be inserted too early since it may pollute the cache with the data that is not used for a long time. Notice, in Listing 29, `lookAhead` is a template parameter, which allows to try different values and see which gives the best performance. More advanced users can try to estimate the prefetching window using the method described in Section 6.2.7, example of using such method can be found on easyperf blog. [147]

Listing 29 Example of a SW prefetching for the next 8 iterations.

```
template <int lookAhead = 8>
void Graph::update(const std::vector<Edge>& edges) {
  for(int i = 0; i + lookAhead < edges.size(); i++) {
    VertexID v = edges[i].from;
    VertexID u = edges[i].to;
    this->out_neighbors[u].push_back(v);
    this->in_neighbors[v].push_back(u);

    // prefetch elements for future iterations
    VertexID v_next = edges[i + lookAhead].from;
    VertexID u_next = edges[i + lookAhead].to;
    __builtin_prefetch(this->out_neighbors.data() + v_next);
    __builtin_prefetch(this->in_neighbors.data()  + u_next);
  }
  // process the remainder of the vector `edges` ...
}
```

SW memory prefetching is most frequently used in the loops, but also one can insert those hints into the parent function, again, all depends on the available prefetching window.

[147] "Precise timing of machine code with Linux perf" - https://easyperf.net/blog/2019/04/03/Precise-timing-of-machine-code-with-Linux-perf#application-estimating-prefetch-window.

This technique is a powerful weapon, however, it should be used with extreme care as it is not easy to get it right. First of all, explicit memory prefetching is not portable, meaning that if it gives performance gains on one platform, it doesn't guarantee similar speedups on another platform. It is very implemetation-specific and platforms are not required to honor those hints. In such a case it will likely degrade performance. My recomendation would be to verify that the impact is positive with all available tools. Not only check the performance numbers, but also make sure that the number of cache misses (L3 in particular) went down. Once the change is committed into the code base, monitor performance on all the platforms that you run your application on, as it could be very sensitive to changes in the surrounding code. Consider dropping the idea if the benefits do not overweight the potential maintanance burden.

For some complicated scenarios, make sure that the code actually prefetches the right memory locations. It can get tricky, when a current iteration of a loop depends on the previous iteration, e.g there is `continue` statement or changing the next element to process guarded by an `if` condition. In this case, my recommendation is to instrument the code to test the accuracy of your prefetching hints. Because when used badly, it can degrade the performance of caches by evicting other useful data.

Finally, explicit prefetching increases code size and adds pressure on the CPU Front-End. A prefetch hint is just a fake load that goes into the memory subsystem, but does not have a destination register. And just like any other instruction it consumes CPU resources. Apply it with extreme care, because when used wrong, it can pessimize the performance of a program.

8.3 Memory Profiling

8.4 Reducing DTLB Misses

As described earlier in the book, TLB is a fast but finite per-core cache for virtual-to-physical address translations of memory addresses. Without it, every memory access by an application would require a time-consuming page walk of the kernel page table to calculate the correct physical address for each referenced virtual address. In a system with a 5-level page table, it will require accessing at least 5 different memory locations to obtain an address translation. In section Section 11.8 we will discuss how huge pages can be used for code. Here we will see how they can be used for data.

Any algorithm that does random accesses into a large memory region will likely suffer from DTLB misses. Examples of such applications are: binary search in a big array, accessing a large hash table, traversing a graph. Usage of huge pages has potential for speeding up such applications.

On x86 platforms, the default page size is 4KB. Consider an application that actively references hundreds of MBs of memory. First, it will need to allocate many small pages which is expensive. Second, it will be touching many 4KB-sized pages, each of which will contend for a limited set of TLB entries. For instance, using huge 2MB pages, 20MB of memory can be mapped with just ten pages, whereas with 4KB pages, you will need 5120 pages. This means fewer TLB entries are needed, in turn reducing the number of TLB misses. It will not be a proportional reduction by a factor of 512 since the number of 2MB entries is much less. For example, in Intel's Skylake core families, L1 DTLB has 64 entries for 4KB pages and only 32 entries for 2MB pages. Besides 2MB huge pages, x86-based chips from AMD and Intel also support 1GB gigantic pages, which are only available for data, not for instructions. Using 1GB pages instead of 2MB pages reduces TLB pressure even more.

Utilizing huge pages typically leads to fewer page walks, and the penalty for walking the kernel page table in the event of a TLB miss is reduced since the table itself is more compact. Performance gains of utilizing huge pages can sometimes go as high as 30%, depending on how much TLB pressure an application is experiencing. Expecting 2x speedups would be asking too much, as it is quite rare that TLB misses are the primary bottleneck. The paper [Luo et al., 2015] presents the evaluation of using huge pages on the SPEC2006 benchmark suite. Results can be summarized as follows. Out of 29 benchmarks in the suite, 15 have a speedup within 1%, which can be discarded as noise. Six benchmarks have speedups in the range of 1%-4%. Four benchmarks have speedups in the range from 4% to 8%. Two benchmarks have speedups of 10%, and the two benchmarks that gain the most, enjoyed 22% and 27% speedups respectively.

Many real-world applications already take advantage of huge pages, for example KVM, MySQL, PostgreSQL, Java JVM, and others. Usually, those SW packages provide an option that enable that feature. Whenever you're using a similar application, check its documentation to see if you can enable huge pages.

Both Windows and Linux allow applications to establish huge-page memory regions. Instructions on how to enable huge pages for Windows and Linux can be found in appendix C. On Linux, there are two ways of using huge pages in an application: Explicit and Transparent Huge Pages. Windows supports is not as rich a Linux and will be discussed later.

8.4.1 Explicit Hugepages.

Explicit Huge Pages (EHP) are available as part of the system memory, and are exposed as a huge page file system `hugetlbfs`. As the name implies, EHPs should be reserved either at boot time or at run time. See appendix C for instructions on how to do that. Reserving EHPs at boot time increases the possibility of successfull allocation because the memory has not yet been significantly fragmented. Explicitly preallocated pages reside in a reserved chunk of memory and cannot be swapped out under memory pressure. Also, this memory space cannot be used for other purposes, so users should be careful and reserve only the number of pages they need.

The simplest method of using EHP in an application is to call `mmap` with `MAP_HUGETLB` as shown in Listing 30. In this code, pointer `ptr` will point to a 2MB region of memory that was explicitly reserved for EHPs. Notice, that allocation may fail due to the EHPs were not reserved in advance. Another less popular ways to use EHPs in user code are provided in appendix C. Also, developers can write their own arena-based allocators that tap into EHPs.

Listing 30 Mapping a memory region from an explicitly allocated huge page.

```
void ptr = mmap(nullptr, size, PROT_READ | PROT_WRITE,
                MAP_PRIVATE | MAP_ANONYMOUS | MAP_HUGETLB, -1, 0);
if (ptr == MAP_FAILED)
  throw std::bad_alloc{};
...
munmap(ptr, size);
```

In the past, there was an option to use the libhugetlbfs[148] library, which allowed to override `malloc` calls used in existing dynamically linked executables to allocate memory on top of EHPs. Unfortunately, this project is no longer maintained. It didn't require users to modify the code or to relink the binary. They could simply prepend the command line with `LD_PRELOAD=libhugetlbfs.so HUGETLB_MORECORE=yes <your app command line>` to make use of it. But luckily, there are other libraries that allow to use huge pages (not EHPs) with `malloc` as we will see shortly.

8.4.2 Transparent Hugepages.

Linux also offers Transparent Hugepage Support (THP), which has two modes of operation: system-wide and per-process. When THP is enabled system-wide, the kernel manages huge pages automatically and it is transparent for applications. The OS kernel tries to assign huge pages to any process when large blocks of memory are needed and it is possible to allocate such, so huge pages do not need to be reserved manually. If THP is enabled per-process, the kernel only assigns huge pages to individual processes' memory areas attributed to the `madvise` system call. You can check if THP enabled in the system with:

```
$ cat /sys/kernel/mm/transparent_hugepage/enabled
always [madvise] never
```

If the values are `always` (system-wide) or `madvise` (per-process), then THP is available for your application. A detailed specification for every option can be found in the Linux kernel documentation[149] regarding THP.

When THP is enabled system-wide, huge pages are used automatically for normal memory allocations, without an explicit request from applications. Basically, to observe the effect of huge pages on their application, a user just need to enable system-wide THPs with `echo "always" | sudo tee /sys/kernel/mm/transparent_hugepage/enabled`. It will automatically lauches a daemon process named `khugepaged` which starts scanning application's memory space to promote regular pages to huge pages. Though sometimes the kernel may fail to promote regular pages into huge pages in case it cannot find a contiguous 2MB chunk of memory.

[148] libhugetlbfs - https://github.com/libhugetlbfs/libhugetlbfs.
[149] Linux kernel THP documentation - https://www.kernel.org/doc/Documentation/vm/transhuge.txt

System-wide THPs mode is good for quick experiments to check if huge pages can improve performance. It works automatically, even for applications that are not aware of THPs, so developers don't have to change the code to see the benefit of huge pages for their application.

When hugepages are enabled system wide, applications may end up allocating much more memory resources. An application may mmap a large region but only touch 1 byte of it, in that case a 2M page might be allocated instead of a 4k page for no good. This is why it's possible to disable hugepages system-wide and to only have them inside MADV_HUGEPAGE madvise regions, which we will discuss next. Don't forget to disable system-wide THPs after you've finished your experiments as it may not benefit every application running on the system.

With the `madvise` (per-process) option, THP is enabled only inside memory regions attributed via `madvise` system call with `MADV_HUGEPAGE` flag. As shown in the Listing 31, pointer `ptr` will point to a 2MB region of anonymous (transparent) memory region, which kernel allocates dynamically. The `mmap` call may fail in case the kernel could not find a contiguous 2MB chunk of memory.

Listing 31 Mapping a memory region to a transparent huge page.

```
void ptr = mmap(nullptr, size, PROT_READ | PROT_WRITE | PROT_EXEC,
                MAP_PRIVATE | MAP_ANONYMOUS, -1 , 0);
if (ptr == MAP_FAILED)
  throw std::bad_alloc{};
madvise(ptr, size, MADV_HUGEPAGE);
// use the memory region `ptr`
munmap(ptr, size);
```

Developers can build custom THP allocators based on the code in Listing 31. But also, it's possible to use THPs inside `malloc` calls that their application is making. Many memory allocation libraries provide that feature by overriding the `libc`'s implementation of `malloc`. Here is an example of one of the most popular such libraries `jemalloc`. If you have access to the source code of the application, you can relink the binary with additional `-ljemalloc` option. This will dynamically link your application against the `jemalloc` library, which will handle all the `malloc` calls. Then use the following option to enable THPs for heap allocations:

```
$ MALLOC_CONF="thp:always" <your app command line>
```

If you don't have access to the source code, you can still make use of `jemalloc` by preloading the dynamic library:

```
$ LD_PRELOAD=/usr/local/libjemalloc.so.2 MALLOC_CONF="thp:always" <your app command line>
```

Windows only offers using huge pages in a way similar to the Linux THP per-process mode via WinAPI `VirtualAlloc` system call. See details in appendix C.

8.4.3 Explicit vs. Transparent Hugepages.

Linux users can use huge pages in three different modes: * Explicit Huge Pages * System-wide Transparent Huge Pages * Per-process Transparent Huge Pages

Let's compare those options. First, EHPs are reserved in virtual memory upfront, THPs are not. That makes it harder to ship SW packages that use EHPs, as they rely on specific configuration settings made by an administrator of a machine. Moreover, EHPs statically sit in memory, consuming precious DRAM space for no reason, when they are not used.

Second, system-wide Transparent Huge Pages are great for quick experiments. No changes in the user code are required to test the benefit of using huge pages in your application. However, it will not be wise to ship a SW package to the customers and ask them to enable system-wide THPs, as it may negatively affect other running programs on that system. Usually, developers identify allocations in the code that could benefit from huge pages and use `madvise` hints in these places (per-process mode).

Per-process THPs don't have either of the downsides mentioned above, but they have another one. Previously we discussed that THP allocation by the kernel happens transparently to the user. The allocation process can potentially involve a number of kernel processes responsible for making space in the virtual memory, which may

include swapping memory to the disk, fragmentation, or promoting pages. Background maintenance of transparent huge pages incurs non-deterministic latency overhead from the kernel as it manages the inevitable fragmentation and swapping issues. EHPs are not subject to memory fragmentation and cannot be swapped to the disk, thus have much less latency overhead.

All in all, THPs are easier to use, but incur bigger allocation latency overhead. That is exactly the reason why THPs are not popular in High-Frequency Trading and other ultra low-latency industries, they prefer to use EHPs instead. On the other hand, virtual machine providers and databases tend to use per-process THPs since requiring additional system configuration can become a burden for their users.

Questions and Exercises

1. Solve `perf-ninja::data_packing` lab assignment, in which you need to make the data structure more compact.
2. Solve `perf-ninja::swmem_prefetch_1` lab assignment by implementing explicit memory prefetching for future loop iterations.
3. Solve `perf-ninja::huge_pages_1` lab assignment using methods we discussed in Section 8.4. Observe any changes in performance, huge page allocation in `/proc/meminfo`, and CPU performance counters that measure DTLB loads and misses.
4. Describe what it takes for a piece of code to be cache-friendly?
5. Run the application that you're working with on a daily basis. Measure its memory footprint, profile and identify hot memory accesses. Are they cache-friendly? Is there a way to improve them?

Chapter Summary

- Most of the real-world applications experience performance bottlenecks that can be related to the CPU Backend. It is not surprising since all the memory-related issues, as well as inefficient computations, belong to this category.
- Performance of the memory subsystem is not growing as fast as CPU performance. Yet, memory accesses are a frequent source of performance problems in many applications. Speeding up such programs requires revising the way they access memory.
- In Chapter 8, we discussed some of the popular recipes for cache-friendly data structures, memory prefetching, and utilizing large memory pages to improve DTLB performance.

9 Optimizing Computations

In the previous chapter, we discussed how to clear the path for efficient memory accesses. Once that is done, it's time to look at how well a CPU works with the data it brings from memory. Modern applications demand a large amount of CPU computations, especially those that involve complex graphics, artificial intelligence, cryptocurrency mining, and big data processing. In this chapter, we will focus on optimizing computations which can reduce the amount of work that a CPU needs to do and improve the overall performance of a program.

When the TMA methodology is applied, inefficient computations are usually reflected in the `Core Bound` and to some extent in `Retiring` categories. The `Core Bound` category represents all the stalls inside a CPU out-of-order execution engine that were not caused by memory issues. There are two main categories:

- Data dependencies between software's instructions are limiting the performance. For example, a long sequence of dependent operations may lead to low Instruction Level Parallelism (ILP) and wasting many executions slots. The next section discusses data dependency chains in more detail.
- Shortage in hardware compute resources (aka not enough execution throughput). It indicates that certain execution units are overloaded (aka execution port contention). This can happen when a workload frequently performs many instructions of the same type. For example, AI algorithms typically perform a lot of multiplications, scientific applications may run many divisions and square root operations. But there is limited number of multipliers and dividers in any given CPU core. Thus when port contention occurs, instructions queue up waiting for their turn to be executed. This type of performance bottleneck is very specific to a particular CPU microarchitecture and usually doesn't have a cure.

[TODO:] Give guidance on how to detect port contention.

In Section 6.1, we said that high `Retiring` metric is a good indicator of a well-performing code. The rationale behind it is that execution is not stalled and a CPU is retiring instructions at a high rate. However, sometimes it may hide the real performance problem, that is inefficient computations. A workload may be executing a lot of instructions that are too simple and not doing much useful work. In this case, the high `Retiring` metric won't translate into the high performance.

In this chapter, we will take a look at the well-known techniques like function inlining, vectorization, and loop optimizations. Those code transformations aim at reducing the total amount of executed instructions, or replacing them with a more efficient ones.

9.1 Data Dependencies

When a program statement refers to the data of a preceding statement, we say that there is a *data dependency* between the two statements. Sometimes people also use terms *dependency chain* or *data flow dependencies*. The example we are most familiar with is shown on Figure 64. To access the node `N+1`, we should first dereference the pointer `N->next`. For the loop on the right, this is a *recurrent* data dependency, meaning it spans multiple iterations of the loop. Basically, traversing a linked list is one very long dependency chain.

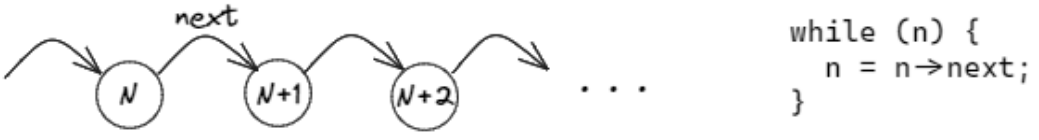


Figure 64: Data dependency while traversing a linked list.

Conventional programs are written assuming the sequential execution model. Under this model, instructions execute one after the other, atomically and in the order specified by the program. However, as we already know, this is not how modern CPUs are built. They are designed to execute instructions out-of-order, in parallel, and in a way that maximizes the utilization of the available execution units.

When long data dependencies do come up, processors are forced to execute code sequentially, utilizing only a part of their full capabilities. Long dependency chains hinder parallelism, which defeats the main advantage of modern

superscalar CPUs. For example, pointer chaising doesn't benefit from OOO execution, and thus will run at the speed of an in-order CPU. As we will see in this section, dependency chains are a major source of performance bottlenecks.

You cannot eliminate data dependencies, they are a fundamental property of programs. Any program takes an input to compute something. In fact, people have developed techniques to discover data dependencies among statements and build data flow graphs. This is called *dependence analysis* and is more appropriate for compiler developers, rather than performance engineers. We are not interested in building data flow graphs for the whole program. Instead, we want to find a critical dependency chain in a hot piece of code (loop or function).

You may wonder: "If you cannot get rid of dependency chains, what *can* you do?". Well, sometimes this will be limiting factor for performance, and unfortunately you will have to live with it. In the last chapter of this book we will touch on one of the possible solutions for breaking dependency chains in the HW, called value prediction. For now, you should seek ways how to break unnecessary data dependency chains or overlap their execution. One such example is shown in Listing 32. Similar to a few other cases, we present source code on the left along with the corresponding ARM aseembly on the right. Also, this code example is included in the Performance Ninja repository on Github, so you can try it yourself.

This small program simulates the random particle movement. We have 1000 particles moving on a 2D surface without constraints, which means they can go as far from their starting position as they want. Each particle is defined by its x and y coordinates on a 2D surface and speed. The initial x and y coordinates are in the range [-1000,1000] and the speed is in the range [0;1], which doesn't change. The program simulates 1000 movement steps for each particle. For each step, we use a random number generator (RNG) to produce an angle, which sets the movement direction for a particle. Then we adjust the coordinates of a particle accordingly.

Given the task at hand, you decide to roll your own RNG, sine and cosine functions to sacrifice some accuracy and make it as fast as possible. After all, this is *random* movement, so it is a good trade-off to make. You choose the medium-quality `XorShift` RNG as it only has 3 shifts and 3 XORs inside. What can be simpler? Also, you quickly searched the web and found sine and cosine approximation using polynomials, which is accurate enough and quite fast.

Let us quickly examine the generated ARM assembly code: * First three `eor` instructions combined with `lsl` or `lsr` correspond to the `XorShift32::gen()` function. * Next `ucvtf` and `fmul` are there to convert the angle from degrees to radians (line 35 in the code). * Sine and Cosine functions both have two `fmul` and one `fmadd` operations. Cosine also has additional `fadd`. * Finally, we have one more pair of `fmadd` to calculate x and y respectively and `stp` instruction to store the pair of coordinates back.

We compiled the code usign Clang-17 C++ compiler and run it on a Mac mini (Apple M1, 2020). You expect this code to "fly", however, there is one very nasty performance problem that slows down the program. Without looking ahead in the text, can you find a recurrent dependency chain in the code?

Congratulations if you've found it. There is a recurrent loop dependency over `XorShift32::val`. To generate the next random number, the generator has to produce the previous number first. The next call of method `gen()` will generate the number based on the previous one. Figure 65 visualizes the problematic loop-carry dependency. Notice, the code for calculating particle coordinates (convert angle to radians, sine, cosine, multiple results by velocity) starts executing as soon as the corresponding random number is ready, but not sooner.

The code that calculates coordinates of each particle does not dependent on each other, so it could be beneficial to pull them left to overlap their execution even more. You probably want to ask: "but how those three (or six) instructions can drag the whole loop down?". Indeed, there are many other "heavy" instructions in the loop, like `fmul` and `fmadd`. However, they are not on the critical path, so they can be executed in parallel with other instructions. And because modern CPUs are very wide, they will execute instructions from multiple iteration at the same time. This allows the OOO engine to effectively find parallelism (independent instructions) within different iterations of the loop.

Let's do some back-of-the-envelope calculations.[150] Each `eor` and `lsl` instruction takes 2 cycle latency, one cycle for shift and one for XOR. We have three dependent `eor + lsl` pairs, so it takes 6 cycles to generate the next random number. This is our absolute minimum for this loop, we cannot run faster than 6 cycles per iteration. The code that follows takes at least 20 cycles latency to finish all the `fmul` and `fmadd` instructions. But it doesn't matter, because

[150] Apple doesn't publish instruction latency and throughput for their products, but there are experiments that shed some light on it, one of such studies is here: https://dougallj.github.io/applecpu/firestorm-simd.html. Since this is unofficial source of data, you should take it with a grain of salt.

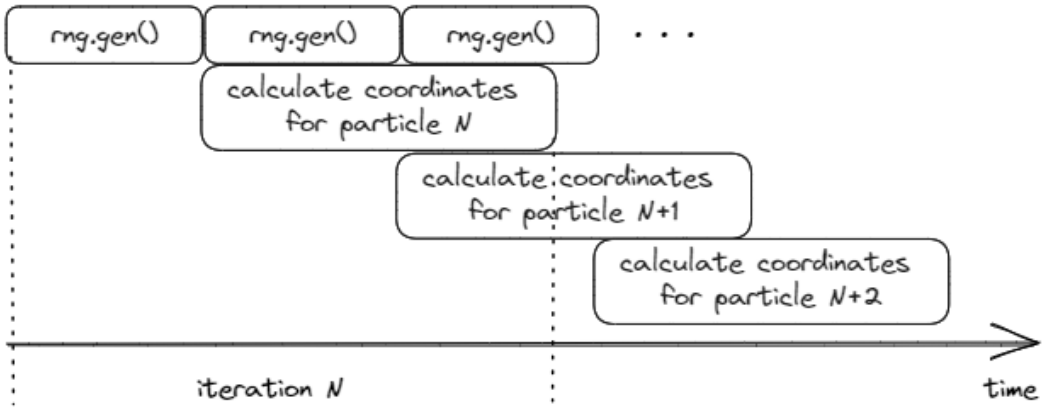


Figure 65: Visualization of dependent execution in Listing 32

they are not on the critical path. The thing that could matter is the throughut of these instructions. The rule of thumb: if an instruction is on a critical path, look at its latency, if it is not on a critical path, look at its throughput. On every loop iteration, we have 5 `fmul` and 4 `fmadd` instructions that are served on the same set of execution units. The M1 processor can run 4 instructions per cycle of this type, so it will take at least `9/4 = 2.25` cycles to issue all the `fmul` and `fmadd` instructions. So, we have two performance limits: the first is imposed by the software (6 cycles per iteration due to dependency chain), and the second is imposed by the hardware (2.25 cycles per iteration due to the throughput of the execution units). Right now we are bound by the first limit, but we can try to break the dependency chain to get closer to the second limit.

One of the ways to solve this would be to employ additional RNG object, so that one of them feeds even iterations and another feeds odd iterations of the loop as shown in Listing 33. Notice, we also manually unrolled the loop. Now we have two separate dependency chains, which can be executed in parallel. One can argue that this changes the functionality of the program, but users would not be able to tell the difference since the motion of particles is random anyway. Alternative solution would be to pick a different RNG that has a less expensive internal dependency chain.

Once you do this transformation, compiler starts autovectorizing the body of the loop, i.e. it glues two chains together and uses SIMD instructions to process them in parallel. To isolate the effect of breaking the dependency chain, we disable compiler vectorization.

To measure the impact of the change, we ran "before" and "after" versions and observed the running time goes down from 19ms per iteration to 10ms per iteration. This is almost a 2x speedup. The `IPC` also goes up from 4.0 to 7.1. To do our due diligence, we also measured other metrics to make sure performance doesn't accidentally improves for other reasons. In the original code, the `MPKI` is 0.01, and `BranchMispredRate` is 0.2%, which means we the program initially did not suffer from cache misses or branch mispredictions. Here is another data point: when running the same code on Intel's Alderlake system, it shows 74% Retiring and 24% Core Bound, which confirms the performance is bound by computations.

With a few additional changes you can generalize this solution to have as many dependency chains as you want. For the M1 processor, the measurements show that having 2 dependency chains is enough to get very close to the hardware limit. Having more than 2 chains brings a negligible performance improvement. However, there is a trend that CPUs are getting wider, i.e. they become increasingly capable of running multiple dependency chains in parallel. That means future processors could benefit from having more than 2 dependency chains. As always you should measure and find the sweet spot for the platforms your code will be running on.

Sometimes it's not enough just to break dependency chains. Imagine for a minute that instead of a simple RNG, you have a very complicated cryptographic algorithm that is a 10'000 instructions long. So, instead of a very short 6 instruction dependency chain, we now have 10'000 instructions standing on the critical path. You immediately do the same change we did above anticipating nice 2x speedup. Only to see a slightly better performance. What's going on?

The problem here is that the CPU simply cannot "see" the second dependency chain to start executing it. Recall from chapter 3, the Reservation Station (RS) capacity is not enough to see 10'000 instructions ahead as it is much

smaller than that. So, the CPU will not be able to overlap the execution of two dependency chains. To fix it, we need to *interleave* those two dependency chains. With this approach you need to change the code so that the RNG object will generate two numbers simultaneously, with *every* statement within the function `gen()` duplicated and interleaved. Even if a compiler inlines all the code and can clearly see both chains, it doesn't automatically interleave them, so you need to watch out for this. Another limitation that you may hit while doing this is register pressure. Running multiple dependency chains in parallel requires keeping more state and thus more registers. If you run out of registers, the compiler will start spilling them to the stack, which will slow down the program.

As a closing thought here, we would like to emphasize the importance of finding that critical dependency chain. It is not always easy, but it is crucial to know what stands on the critical path in your loop, function, or piece of code. Otherwise you may find yourself fixing secondary issues that barely make a difference.

9.2 Inlining Functions

If you're one of those developers who frequently looks into assembly code, you have probably seen `CALL`, `PUSH`, `POP`, and `RET` instructions. In x86 ISA, `CALL` and `RET` instructions are used to call and return from a function. `PUSH` and `POP` instructions are used to save a register value on the stack and restore it back.

The nuances of a function call are described by the *calling convention*, how arguments are passed and in what order, how the result is returned, which registers the called function must preserve and how the work is split between the caller and the callee. Based on a calling convention, when a caller makes a function call, it expects that some registers will hold the same values after the callee returns. Thus, if a callee needs to change one of the registers that should be preserved, it needs to save (`PUSH`) and restore (`POP`) them before returning to the caller. A series of `PUSH` instructions is called a *prologue*, and a series of `POP` instructions is called an *epilogue*.

When a function is small, overhead of calling a function (prologue and epilogue) can be very pronounced. This overhead can be eliminated by inlining a function body into the place where it was called. Function inlining is a process of replacing a call to a function `F` with the code for `F` specialized with the actual arguments of the call. Inlining is one of the most important compiler optimizations. Not only because it eliminates the overhead of calling a function, but also it enables other optimizations. This happens because when a compiler inlines a function, the scope of compiler analysis widens to a much larger chunk of code. However, there are disadvantages as well: inlining can potentially increase the code size and compile time[151].

The primary mechanism for function inlining in many compilers relies on a cost model. For example, in the LLVM compiler, it is based on computing a cost for each function call (callsite). The cost of inlining a function call is based on the number and type of instructions in that function. Inlining happens if the cost is less than a threshold, which is usually fixed; however, it can be varied under certain circumstances.[152] In addition to the generic cost model, there are many heuristics that can overwrite cost model decisions in some cases. For instance:

- Tiny functions (wrappers) are almost always inlined.
- Functions with a single callsite are preferred candidates for inlining.
- Large functions usually are not inlined as they bloat the code of the caller function.

Also, there are situations when inlining is problematic:

- A recursive function cannot be inlined into itself.
- A function that is referred to through a pointer can be inlined in place of a direct call but the functiona has to remain in the binary, i.e., it cannot be fully inlined and eliminated. The same is true for functions with external linkage.

As we said earlier, compilers tend to use a cost model approach when making a decision about inlining a function, which typically works well in practice. In general, it is a good strategy to rely on the compiler for making all the inlining decisions and adjust if needed. The cost model cannot account for every possible situation, which leaves room for improvement. Sometimes compilers require special hints from the developer. One way to find potential candidates for inlining in a program is by looking at the profiling data, and in particular, how hot is the prologue and the epilogue of the function. Below is an example of a function profile with prologue and epilogue consuming `~50%` of the function time:

[151] See the article: https://aras-p.info/blog/2017/10/09/Forced-Inlining-Might-Be-Slow/.
[152] For example, 1) when a function declaration has a hint for inlining, 2) when there is profiling data for the function, or 3) when a compiler optimizes for size (`-Os`) rather than performance (`-O2`).

```
Overhead |  Source code & Disassembly
   (%)   |  of function `foo`
-----------------------------------------
    3.77 :  418be0:  push   r15      # prologue
    4.62 :  418be2:  mov    r15d,0x64
    2.14 :  418be8:  push   r14
    1.34 :  418bea:  mov    r14,rsi
    3.43 :  418bed:  push   r13
    3.08 :  418bef:  mov    r13,rdi
    1.24 :  418bf2:  push   r12
    1.14 :  418bf4:  mov    r12,rcx
    3.08 :  418bf7:  push   rbp
    3.43 :  418bf8:  mov    rbp,rdx
    1.94 :  418bfb:  push   rbx
    0.50 :  418bfc:  sub    rsp,0x8
    ...
    #                               # function body
    ...
    4.17 :  418d43:  add    rsp,0x8  # epilogue
    3.67 :  418d47:  pop    rbx
    0.35 :  418d48:  pop    rbp
    0.94 :  418d49:  pop    r12
    4.72 :  418d4b:  pop    r13
    4.12 :  418d4d:  pop    r14
    0.00 :  418d4f:  pop    r15
    1.59 :  418d51:  ret
```

When you see hot `PUSH` and `POP` instructions, this might be a strong indicator that the time consumed by the prologue and epilogue of the function might be saved if we inline the function. Note that even if the prologue and epilogue are hot, it doesn't necessarily mean it will be profitable to inline the function. Inlining triggers a lot of different changes, so it's hard to predict the outcome. Always measure the performance of the changed code before forcing compiler to inline a function.

For GCC and Clang compilers, one can make a hint for inlining `foo` with the help of C++11 `[[gnu::always_inline]]` attribute as shown in the code example below. For earlier C++ standards one can use `__attribute__((always_inline))`. For the MSVC compiler, one can use the `__forceinline` keyword.

```
[[gnu::always_inline]] int foo() {
    // foo body
}
```

9.3 Loop Optimizations

Loops are at the heart of nearly all high performance programs. Since loops represent a piece of code that is executed a large number of times, they are where the majority of the execution time is spent. Small changes in such a critical piece of code may have a high impact on the performance of a program. That's why it is so important to carefully analyze the performance of hot loops in a program and know possible ways to improve them.

To effectively optimize a loop, it is crucial to understand the performance bottleneck. Once you find a loop that is using most of the time, try to determine what limits its performance. Usually, it will be one or many of the following: memory latency, memory bandwidth, or compute capabilities of a machine. Roofline Performance Model (Section 5.5) is a good starting point for assessing the performance of different loops against the HW theoretical maximums. Top-down Microarchitecture Analysis (Section 6.1) can be another good source of information about the bottlenecks.

In this section, we will take a look at the most well-known loop optimizations that address the types of bottlenecks mentioned above. We first discuss low-level optimizations that only move code around in a single loop. Such optimizations typically help make computations inside the loop more effective. Next, we will take a look at high-level

optimizations that restructure loops, which often affects multiple loops. The second class of optimizations generally aims at improving memory accesses eliminating memory bandwidth and memory latency issues. Note, this is not a complete list of all known loop transformations. For more detailed information on each of the transformations discussed below, readers can refer to [Cooper & Torczon, 2012].

Compilers can automatically recognize an opportunity to perform certain loop transformations. However, sometimes developer's interference is required to reach the desired outcome. In the second part of this section, we will share some thoughts on how to discover loop optimization opportunities. Understanding what transformations were performed on a given loop and what optimizations a compiler failed to do is one of the keys to successful performance tuning. In the end, we will consider an alternative way of optimizing loops with polyhedral frameworks.

9.3.1 Low-level Optimizations.

First, we will consider simple loop optimizations that transform the code inside a single loop: Loop Invariant Code Motion, Loop Unrolling, Loop Strength Reduction, and Loop Unswitching. Such optimizations usually help improve the performance of a loop with high arithmetic intensity (see Section 5.5), i.e., when a loop is bound by CPU compute capabilities. Generally, compilers are good at doing such transformations; however, there are still cases when a compiler might need a developer's support. We will talk about that in subsequent sections.

Loop Invariant Code Motion (LICM): expressions evaluated in a loop that never change are called loop invariants. Since their value doesn't change across loop iterations, we can move loop invariant expressions outside of the loop. We do so by storing the result in a temporary variable and use it inside the loop (see Listing 34). All decent compilers nowadays successfully perform LICM in majority of the cases.

Loop Unrolling: an induction variable is a variable in a loop, whose value is a function of the loop iteration number. For example, `v = f(i)`, where `i` is an iteration number. Modifying the induction variable on each iteration can be unnecessary and expensive. Instead, we can unroll a loop and perform multiple iterations for each increment of the induction variable (see Listing 35).

The primary benefit of loop unrolling is to perform more computations per iteration. At the end of each iteration, the index value must be incremented, tested, and the control is branched back to the top of the loop if it has more iterations to process. This work can be considered as loop "tax", which can be reduced. By unrolling the loop in Listing 35 by a factor of 2, we reduce the number of executed compare and branch instructions by half.

Loop unrolling is a well-known optimization; still, many people are confused about it and try to unroll the loops manually. I suggest that no developer should unroll any loop by hand. First, compilers are very good at doing this and usually do loop unrolling quite optimally. The second reason is that processors have an "embedded unroller" thanks to their out-of-order speculative execution engine (see Chapter 3). While the processor is waiting for the load from the first iteration to finish, it may speculatively start executing the load from the second iteration assuming there are no loop-carry dependencies. This spans to multiple iterations ahead, effectively unrolling the loop in the instruction Reorder Buffer (ROB).

Loop Strength Reduction (LSR): replace expensive instructions with cheaper ones. Such transformation can be applied to all expressions that use an induction variable. Strength reduction is often applied to array indexing. Compilers perform LSR by analyzing how the value of a variable evolves across the loop iterations. In LLVM, it is known as Scalar Evolution (SCEV). In Listing 36, it is relatively easy for a compiler to prove that the memory location `b[i*10]` is a linear function of the loop iteration number `i`, thus it can replace the expensive multiplication with a cheaper addition.

Loop Unswitching: if a loop has a conditional statement inside and it is invariant, we can move it outside of the loop. We do so by duplicating the body of the loop and placing a version of it inside each of the `if` and `else` clauses of the conditional statement (see Listing 37). While the loop unswitching may double the amount of code written, each of these new loops may now be separately optimized.

9.3.2 High-level Optimizations.

There is another class of loop transformations that change the structure of loops and often affect multiple nested loops. We will take a look at Loop Interchange, Loop Blocking (Tiling), and Loop Fusion and Distribution (Fission). This set of transformations aims at improving memory accesses and eliminating memory bandwidth and memory latency bottlenecks. From a compiler perspective, it is very difficult to prove legality of such transformations and justify their performance benefit. In that sense, developers are in a better position since they only have to care

about the legality of the transformation in their particular piece of code, not about every possible scenario that may happen. Unfortunately, that also means that usually we have to do such transformations manually.

Loop Interchange: is a process of exchanging the loop order of nested loops. The induction variable used in the inner loop switches to the outer loop, and vice versa. Listing 38 shows an example of interchanging nested loops for `i` and `j`. The main purpose of loop interchange is to perform sequential memory accesses to the elements of a multi-dimensional array. By following the order in which elements are laid out in memory, we can improve the spatial locality of memory accesses and make our code more cache-friendly. This transformation helps to eliminate memory bandwidth and memory latency bottlenecks.

Loop Interchange is only legal if loops are *perfectly nested*. A perfectly nested loop is one wherein all the statements are in the innermost loop. Interchanging imperfect loop nest is harder to do but still possible, check an example in the Codee[153] catalog.

Loop Blocking (Tiling): the idea of this transformation is to split the multi-dimensional execution range into smaller chunks (blocks or tiles) so that each block will fit in the CPU caches. If an algorithm works with large multi-dimensional arrays and performs strided accesses to their elements, there is a high chance of poor cache utilization. Every such access may push the data that will be requested by future accesses out of the cache (cache eviction). By partitioning an algorithm in smaller multi-dimensional blocks, we ensure the data used in a loop stays in the cache until it is reused.

In the example shown in Listing 39, an algorithm performs row-major traversal of elements of array `a` while doing column-major traversal of array `b`. The loop nest can be partitioned into smaller blocks to maximize the reuse of elements in array `b`.

Loop Blocking is a widely known method of optimizing GEneral Matrix Multiplication (GEMM) algorithms. It enhances the cache reuse of the memory accesses and improves both memory bandwidth and memory latency of an algorithm.

Typically, engineers optimize a tiled algorithm for the size of caches that are private to each CPU core (L1 or L2 for Intel and AMD, L1 for Apple). However, the sizes of private caches are changing from generation to generation, so hardcoding a block size presents its own set of challenges. As an alternative solution, one can use cache-oblivious[154] algorithms whose goal is to work reasonably well for any size of the cache.

Loop Fusion and Distribution (Fission): separate loops can be fused together when they iterate over the same range and do not reference each other's data. An example of a Loop Fusion is shown in Listing 40. The opposite procedure is called Loop Distribution (Fission) when the loop is split into separate loops.

Loop Fusion helps to reduce the loop overhead (similar to Loop Unrolling) since both loops can use the same induction variable. Also, loop fusion can help to improve the temporal locality of memory accesses. In Listing 40, if both `x` and `y` members of a structure happen to reside on the same cache line, it is better to fuse the two loops since we can avoid loading the same cache line twice. This will reduce the cache footprint and improve memory bandwidth utilization.

However, loop fusion does not always improve performance. Sometimes it is better to split a loop into multiple passes, pre-filter the data, sort and reorganize it, etc. By distributing the large loop into multiple smaller ones, we limit the amount of data required for each iteration of the loop, effectively increasing the temporal locality of memory accesses. This helps in situations with a high cache contention, which typically happens in large loops. Loop distribution also reduces register pressure since, again, fewer operations are being done within each iteration of the loop. Also, breaking a big loop into multiple smaller ones will likely be beneficial for the performance of the CPU Front-End because of better instruction cache utilization. Finally, when distributed, each small loop can be further optimized separately by the compiler.

Loop Unroll and Jam: to perform this transformation, one needs to unroll the outer loop first, then jam (fuse) multiple inner loops together as shown in Listing 41. This transformation increases the ILP (Instruction-Level Parallelism) of the inner loop since more independent instructions are executed inside the inner loop. In the code example, inner loop is a reduction operation, which accumulates the deltas between elements of arrays `a` and `b`. When we unroll and jam the loop nest by a factor of 2, we effectively execute 2 iterations of the initial outer loop

[153] Codee: perfect loop nesting - [https://www.codee.com/catalog/glossary-perfect-loop-nesting/]](https://www.codee.com/catalog/glossary-perfect-loop-nesting/)
[154] Cache-oblivious algorithm - https://en.wikipedia.org/wiki/Cache-oblivious_algorithm

simultaneously. This is emphesized by having 2 independent accumulators, which breaks dependency chains over `diffs` in the initial variant.

Loop Unroll and Jam can be performed as long as there are no cross-iteration dependencies on the outer loops, in other words, two iterations of the inner loop can be executed in parallel. Also, this transformation makes sense if inner loop has memory accesses that are strided on the outer loop index (`i` in this case), otherwise other transformations likely apply better. Unroll and Jam is especially useful when the trip count of the inner loop is low, e.g. less than 4. By doing the transformation, we pack more independent operations into the inner loop, which increases the ILP.

Unroll and Jam transformation sometimes could be very useful for outer loop vectorization, which, at the time of writing, compilers cannot do automatically. In a situation when trip count of the inner loop is not visible to a compiler, it could still vectorize the original inner loop, hoping that it will execute enough iterations to hit the vectorized code (more on vectorization in the next section). But if the trip count is low, the program will use a slow scalar version of the loop. Once we do Unroll and Jam, we allow compiler to vectorize the code differently: now "glueing" the independent instructions in the inner loop together (aka SLP vectorization).

9.3.3 Discovering Loop Optimization Opportunities.

As we discussed at the beginning of this section, compilers will do the heavy-lifting part of optimizing your loops. You can count on them on making all the obvious improvements in the code of your loops, like eliminating unnecessary work, doing various peephole optimizations, etc. Sometimes a compiler is clever enough to generate the fast versions of the loops by default, and other times we have to do some rewriting ourselves to help the compiler. As we said earlier, from a compiler's perspective, doing loop transformations legally and automatically is very difficult. Often, compilers have to be conservative when they cannot prove the legality of a transformation.

Consider a code in Listing 42. A compiler cannot move the expression `strlen(a)` out of the loop body. So, the loop checks if we reached the end of the string on each iteration, which is obviously slow. The reason why a compiler cannot hoist the call is that there could be a situation when the memory regions of arrays `a` and `b` overlap. In this case, it would be illegal to move `strlen(a)` out of the loop body. If developers are sure that the memory regions do not overlap, they can declare both parameters of function `foo` with the `restrict` keyword, i.e., `char* __restrict__ a`.

Sometimes compilers can inform us about failed transformations via compiler optimization remarks (see Section 5.7). However, in this case, neither Clang 10 nor GCC 10 were able to explicitly tell that the expression `strlen(a)` was not hoisted out of the loop. The only way to find this out is to examine hot parts of the generated assembly code according to the profile of the application. Analyzing machine code requires the basic ability to read assembly language, but it is a highly rewarding activity.

It is a reasonable strategy to try to get the low-hanging fruits first. Developers could use compiler optimizations reports or examine the machine code of a loop to search for easy improvements. Sometimes, it's possible to adjust compiler transformations using user directives. For example, when we find out that the compiler unrolled our loop by a factor of 4, we may check if using a higher unrolling factor will improve performance. Most compilers support `#pragma unroll(8)`, which will instruct a compiler to use the unrolling factor specified by the user. There are other pragmas that control certain transformations, like loop vectorization, loop distribution, and others. For a complete list of user directives, we invite the user to check compiler's manual.

Next, developers should identify the bottlenecks in the loop and assess performance against the HW theoretical maximum. Start with the Roofline Performance Model (Section 5.5), which will reveal the bottlenecks that developers should try to address. The performance of the loops is limited by one or many of the following factors: memory latency, memory bandwidth, or compute capabilities of a machine. Once the bottlenecks of a loop have been identified, developers can try to apply one of the transformations discussed earlier in this section.

Even though there are well-known optimization techniques for a particular set of computational problems, loop optimizations remain "black art" that comes with experience. We recommend you to rely on a compiler and complement it with manually transforming the code when necessary. Above all, keep the code as simple as possible and do not introduce unreasonably complicated changes if the performance benefits are negligible.

9.3.4 Loop Optimization Frameworks

Over the years, researchers have developed techniques to determine the legality of loop transformations and to transform loops automatically. One such invention is the polyhedral framework[155]. GRAPHITE[156] was among the first set of polyhedral tools that were integrated into a production compiler. GRAPHITE performs a set of classical loop optimizations based on the polyhedral information, extracted from GIMPLE, GCC's low-level intermediate representation. GRAPHITE has demonstrated the feasibility of the approach.

Later LLVM compiler developed its own polyhedral framework called Polly[157]. Polly is a high-level loop and data-locality optimization infrastructure for LLVM. It uses an abstract mathematical representation based on integer polyhedral to analyze and optimize the memory access patterns of a program. Polly performs classical loop transformations, especially tiling and loop fusion, to improve data-locality. This framework has shown significant speedups on a number of well-known benchmarks [Grosser et al., 2012]. Below is an example of how Polly can give an almost 30 times speedup of a GEneral Matrix-Multiply (GEMM) kernel from Polybench 2.0[158] benchmark suite:

```
$ clang -O3 gemm.c -o gemm.clang
$ time ./gemm.clang
real    0m6.574s
$ clang -O3 gemm.c -o gemm.polly -mllvm -polly
$ time ./gemm.polly
real    0m0.227s
```

Polly is a powerful framework for loop optimizations; however, it still misses out on some common and important situations.[159] It is not enabled in the standard optimization pipeline in the LLVM infrastructure and requires that the user provide an explicit compiler option for using it (`-mllvm -polly`). Using polyhedral frameworks is a viable option when searching for a way to speed up your loops.

9.4 Vectorization

On modern processors, the use of SIMD instructions can result in a great speedup over regular un-vectorized (scalar) code. When doing performance analysis, one of the top priorities of the software engineer is to ensure that the hot parts of the code are vectorized. This section guides engineers towards discovering vectorization opportunities. For a recap on the SIMD capabilities of modern CPUs, readers can take a look at Section 3.4.

Often vectorization happens automatically without any user intervention, this is called autovectorization. In such situation, compiler automatically recognizes the opportunity to produce SIMD machine code from the source code. Autovectorization could be a convenient solution because modern compilers generate fast vectorized code for a wide variety of programs.

However, in some cases, autovectorization does not succeed without intervention by the software engineer, perhaps based on the feedback[160] that they get from a compiler or profiling data. In such cases, programmers need to tell the compiler that a particular code region is vectorizable or that vectorization is profitable. Modern compilers have extensions that allow power users to control the autovectorization process and make sure that certain parts of the code are vectorized efficiently. However, this control is limited. There will be several examples of using compiler hints in the subsequent sections.

It is important to note that there is a range of problems where SIMD is important and where autovectorization just does not work and is not likely to work in the near future. One can find an example in [Muła & Lemire, 2019]. Outer loop autovectorization is not currently attempted by compilers. They are less likely to vectorize floating-point code because results will differ numerically (more details later in this section). Code involving permutations or shuffles across vector lanes is also less likely to autovectorize, and this is likely to remain difficult for compilers.

There is one more subtle problem with autovectorization. As compilers evolve, optimizations that they make are changing. The successful autovectorization of the code that was done in the previous compiler version may stop working in the next version and vice versa. Also, during code maintenance or refactoring, the structure of the code

[155] Polyhedral framework - https://en.wikipedia.org/wiki/Loop_optimization#The_polyhedral_or_constraint-based_framework.
[156] GRAPHITE polyhedral framework - https://gcc.gnu.org/wiki/Graphite.
[157] Polly - https://polly.llvm.org/.
[158] Polybench - https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.
[159] Why not Polly? - https://sites.google.com/site/parallelizationforllvm/why-not-polly.
[160] For example, compiler optimization reports, see Section 5.7.

may change, such that autovectorization suddenly starts failing. This may occur long after the original software was written, so it would be more expensive to fix or redo the implementation at this point.

When it is absolutely necessary to generate specific assembly instructions, one should not rely on compiler autovectorization. In such cases, code can instead be written using compiler intrinsics, which we will discuss later in this chapter. In most cases, compiler intrinsics provide a 1-to-1 mapping to assembly instructions. Intrinsics are somewhat easier to use than inline assembly because the compiler takes care of register allocation, and they allow the programmer to retain considerable control over code generation. However, they are still often verbose and difficult to read, and subject to behavioral differences or even bugs in various compilers.

For a middle path between low-effort but unpredictable autovectorization, and verbose/unreadable but predictable intrinsics, one can use a wrapper library around intrinsics. These tend to be more readable, can centralize compiler fixes in a library as opposed to scattering workarounds in user code, and still allow developers control over the generated code. Many such libraries exist, differing in their coverage of recent or 'exotic' operations, and the number of platforms they support. To our knowledge, Highway is currently the only one that fully supports scalable vectors as seen in the SVE and RISC-V V instruction sets. Note that one of the authors is the tech lead for this library. It will be introduced in Section 9.5.

Note that when using intrinsics or a wrapper library, it is still advisable to write the initial implementation using C++. This allows rapid prototyping and verification of correctness, by comparing the results of the original code against the new vectorized implementation.

In the remainder of this section, we will discuss several of these approaches, especially inner loop vectorization because it is the most common type of autovectorization. The other two types, outer loop vectorization, and SLP (Superword-Level Parallelism) vectorization, are mentioned in appendix B.

9.4.1 Compiler Autovectorization.

Multiple hurdles can prevent auto-vectorization, some of which are inherent to the semantics of programming languages. For example, the compiler must assume that unsigned loop-indices may overflow, and this can prevent certain loop transformations. Another example is the assumption that the C programming language makes: pointers in the program may point to overlapping memory regions, which can make the analysis of the program very difficult. Another major hurdle is the design of the processor itself. In some cases, processors don't have efficient vector instructions for certain operations. For example, performing predicated (bitmask-controlled) load and store operations are not available on most processors. Another example is vector-wide format conversion between signed integers to doubles because the result operates on vector registers of different sizes. Despite all of the challenges, the software developer can work around many of the challenges and enable vectorization. Later in the section, we provide guidance on how to work with the compiler and ensure that the hot code is vectorized by the compiler.

The vectorizer is usually structured in three phases: legality-check, profitability-check, and transformation itself:

- **Legality-check**: in this phase, the compiler checks if it is legal to transform the loop (or another type of code region) into using vectors. The loop vectorizer checks that the iterations of the loop are consecutive, which means that the loop progresses linearly. The vectorizer also ensures that all of the memory and arithmetic operations in the loop can be widened into consecutive operations. That the control flow of the loop is uniform across all lanes and that the memory access patterns are uniform. The compiler has to check or ensure somehow that the generated code won't touch memory that it is not supposed to and that the order of operations will be preserved. The compiler needs to analyze the possible range of pointers, and if it has some missing information, it has to assume that the transformation is illegal. The legality phase collects a list of requirements that need to happen for vectorization of the loop to be legal.
- **Profitability-check**: next, the vectorizer checks if a transformation is profitable. It compares different vectorization factors and figures out which vectorization factor would be the fastest to execute. The vectorizer uses a cost model to predict the cost of different operations, such as scalar add or vector load. It needs to take into account the added instructions that shuffle data into registers, predict register pressure, and estimate the cost of the loop guards that ensure that preconditions that allow vectorizations are met. The algorithm for checking profitability is simple: 1) add-up the cost of all of the operations in the code, 2) compare the costs of each version of the code, 3) divide the cost by the expected execution count. For example, if the scalar code costs 8 cycles, and the vectorized code costs 12 cycles, but performs 4 loop iterations at once, then the vectorized version of the loop is probably faster.

- **Transformation**: finally, after the vectorizer figures out that the transformation is legal and profitable, it transforms the code. This process also includes the insertion of guards that enable vectorization. For example, most loops use an unknown iteration count, so the compiler has to generate a scalar version of the loop, in addition to the vectorized version of the loop, to handle the last few iterations. The compiler also has to check if pointers don't overlap, etc. All of these transformations are done using information that is collected during the legality check phase.

9.4.2 Discovering Vectorization Opportunities.

Amdahl's law[161] teaches us that we should spend time analyzing only those parts of code that are used the most during the execution of a program. Thus, performance engineers should focus on hot parts of the code that were highlighted by a profiling tool. As mentioned earlier, vectorization is most frequently applied to loops.

Discovering opportunities for improving vectorization should start by analyzing hot loops in the program and checking what optimizations were performed by the compiler. Checking compiler vectorization remarks (see Section 5.7) is the easiest way to know that. Modern compilers can report whether a certain loop was vectorized, and provide additional details, e.g. vectorization factor (VF). In the case when the compiler cannot vectorize a loop, it is also able to tell the reason why it failed.

An alternative way to using compiler optimization reports is to check assembly output. It is best to analyze the output from a profiling tool that shows the correspondence between the source code and generated assembly instructions for a given loop. That way you only focus on the code that matters, i.e. the hot code. However, understanding assembly language is much more difficult than high-level language like C++. It may take some time to figure out the semantics of the instructions generated by the compiler. But this skill is highly rewarding and often provide valuable insights. Experienced developers can quickly tell whether the code was vectorized or not just by looking at instruction mnemonics and the register names used by those instructions. For example, in x86 ISA, vector instructions operate on packed data (thus have `P` in their name) and use `XMM`, `YMM`, or `ZMM` registers, e.g. `VMULPS XMM1, XMM2, XMM3` multiplies four single precision floats in `XMM2` and `XMM3` and saves the result in `XMM1`. But be careful, often people conclude from seeing `XMM` register being used, that it is vector code – not necessary. For instance, the `VMULSS` instruction will only multiply one single precision floating point value, not four.

There are a few common cases that developers frequently run into when trying to accelerate vectorizable code. Below we present four typical scenarios and give general guidance on how to proceed in each case.

I STOPPED HERE

9.4.2.1 Vectorization Is Illegal. In some cases, the code that iterates over elements of an array is simply not vectorizable. Vectorization remarks are very effective at explaining what went wrong and why the compiler can't vectorize the code. Listing 43 shows an example of dependence inside a loop that prevents vectorization.[162]

While some loops cannot be vectorized due to the hard limitations described above, others could be vectorized when certain constraints are relaxed. There are situations when the compiler cannot vectorize a loop because it simply cannot prove it is legal to do so. Compilers are generally very conservative and only do transformations when they are sure it doesn't break the code. Such soft limitations could be relaxed by providing additional hints to the compiler. For example, when transforming the code that performs floating-point arithmetic, vectorization may change the behavior of the program. The floating-point addition and multiplication are commutative, which means that you can swap the left-hand side and the right-hand side without changing the result: `(a + b == b + a)`. However, these operations are not associative, because rounding happens at different times: `((a + b) + c) != (a + (b + c))`. The code in Listing 44 cannot be auto vectorized by the compiler. The reason is that vectorization would change the variable sum into a vector accumulator, and this will change the order of operations and may lead to different rounding decisions and a different result.

However, if the program can tolerate a bit of inaccuracy in the final result (which usually is the case), we can convey this information to the compiler to enable vectorization. Clang and GCC compilers have a flag, `-ffast-math`,[163] that allows this kind of transformation:

```
$ clang++ -c a.cpp -O3 -march=core-avx2 -Rpass-analysis=.*
```

[161] Amdahl's_law - https://en.wikipedia.org/wiki/Amdahl's_law.
[162] It is easy to spot a read-after-write dependency once you unroll a couple of iterations of the loop. See the example in Section 5.7.
[163] The compiler flag `-Ofast` enables `-ffast-math` as well as the `-O3` compilation mode.

```
...
a.cpp:5:9: remark: loop not vectorized: cannot prove it is safe to reorder floating-point
    operations; allow reordering by specifying '#pragma clang loop vectorize(enable)' before
    the loop or by providing the compiler option '-ffast-math'.
    [-Rpass-analysis=loop-vectorize]
...
$ clang++ -c a.cpp -O3 -ffast-math -Rpass=.*
...
a.cpp:4:3: remark: vectorized loop (vectorization width: 4, interleaved count: 2)
    [-Rpass=loop-vectorize]
...
```

Unfortunately this flag involves subtle and potentially dangerous behavior changes, including for Not-a-Number, signed zero, infinity and subnormals. Because third-party code may not be ready for these effects, this flag should not be enabled across large sections of code without careful validation of the results, including for edge cases.

Let's look at another typical situation when a compiler may need support from a developer to perform vectorization. When compilers cannot prove that a loop operates on arrays with non-overlapping memory regions, they usually choose to be on the safe side. Let's revisit the example from Listing 12 provided in Section 5.7. When the compiler tries to vectorize the code presented in Listing 45, it generally cannot do this because the memory regions of arrays `a`, `b`, and `c` can overlap.

Here is the optimization report (enabled with `-fopt-info`) provided by GCC 10.2:

```
$ gcc -O3 -march=core-avx2 -fopt-info
a.cpp:2:26: optimized: loop vectorized using 32 byte vectors
a.cpp:2:26: optimized:  loop versioned for vectorization because of possible aliasing
```

GCC has recognized potential overlap between memory regions of arrays `a`, `b`, and `c`, and created multiple versions of the same loop. The compiler inserted runtime checks[164] for detecting if the memory regions overlap. Based on that checks, it dispatches between vectorized and scalar[165] versions. In this case, vectorization comes with the cost of inserting potentially expensive runtime checks. If a developer knows that memory regions of arrays `a`, `b`, and `c` do not overlap, it can insert `#pragma GCC ivdep`[166] right before the loop or use the `__restrict__` keyword as shown in Listing 13. Such compiler hints will eliminate the need for the GCC compiler to insert runtime checks mentioned earlier.

By their nature, compilers are static tools: they only reason based on the code they work with. For example, some of the dynamic tools, such as Intel Advisor, can detect if issues like cross-iteration dependence or access to arrays with overlapping memory regions actually occur in a given loop. But be aware that such tools only provide a suggestion. Carelessly inserting compiler hints can cause real problems.

9.4.2.2 Vectorization Is not Beneficial.

In some cases, the compiler can vectorize the loop but figures that doing so is not profitable. In the code presented on Listing 46, the compiler could vectorize the memory access to array `A` but would need to split the access to array `B` into multiple scalar loads. The scatter/gather pattern is relatively expensive, and compilers that can simulate the cost of operations often decide to avoid vectorizing code with such patterns.

Here is the compiler optimization report for the code in Listing 46:

```
$ clang -c -O3 -march=core-avx2 a.cpp -Rpass-missed=loop-vectorize
a.cpp:3:3: remark: the cost-model indicates that vectorization is not beneficial
    [-Rpass-missed=loop-vectorize]
  for (int i = 0; i < n; i++)
  ^
```

Users can force the Clang compiler to vectorize the loop by using the `#pragma` hint, as shown in Listing 47. However, keep in mind that the true fact of whether vectorization is profitable or not largely depends on the runtime data, for

[164] See example on easyperf blog: https://easyperf.net/blog/2017/11/03/Multiversioning_by_DD.
[165] But the scalar version of the loop still may be unrolled.
[166] It is GCC specific pragma. For other compilers, check the corresponding manuals.

example, the number of iterations of the loop. Compilers don't have this information available,[167] so they often tend to be conservative. Developers can use such hints when searching for performance headrooms.

Developers should be aware of the hidden cost of using vectorized code. Using AVX and especially AVX-512 vector instructions could lead to frequency downclocking or startup overhead, which on certain CPUs can also affect subsequent code over a period of several microseconds. The vectorized portion of the code should be hot enough to justify using AVX-512.[168] For example, sorting 80 KiB was found to be sufficient to amortize this overhead and make vectorization worthwhile.[169]

9.4.2.3 Loop Vectorized but Scalar Version Used. In some cases, the compiler can successfully vectorize the code, but the vectorized code does not show in the profiler. When inspecting the corresponding assembly of a loop, it is usually easy to find the vectorized version of the loop body because it uses the vector registers, which are not commonly used in other parts of the program, and the code is unrolled and filled with checks and multiple versions for enabling different edge cases.

If the generated code is not executed, one possible reason for this is that the code that the compiler has generated assumes loop trip counts that are higher than what the program uses. For example, to vectorize efficiently on a modern CPU, programmers need to vectorize and utilize AVX2 and also unroll the loop 4-5 times to generate enough work for the pipelined FMA units. This means that each loop iteration needs to process around 40 elements. Many loops may run with loop trip counts that are below this value and may fall back to use the scalar remainder loop. It is easy to detect these cases because the scalar remainder loop would light up in the profiler, and the vectorized code would remain cold.

The solution to this problem is to force the vectorizer to use a lower vectorization factor or unroll count, to reduce the number of elements that loops process and enable more loops with lower trip counts to visit the fast vectorized loop body. Developers can achieve that with the help of `#pragma` hints. For Clang compiler one can use `#pragma clang loop vectorize_width(N)` as shown in the article[170]on easyperf blog.

9.4.2.4 Loop Vectorized in a Suboptimal Way. When you see a loop being vectorized and is executed at runtime, likely this part of the program already performs well. However, there are exceptions. Sometimes human experts can come up with the code that outperforms the one generated by the compiler.

The optimal vectorization factor can be unintuitive because of several factors. First, it is difficult for humans to simulate the operations of the CPU in their heads, and there is no alternative to actually trying multiple configurations. Vector shuffles that touch multiple vector lanes could be more or less expensive than expected, depending on many factors. Second, at runtime, the program may behave in unpredictable ways, depending on port pressure and many other factors. The advice here is to try to force the vectorizer to pick one specific vectorization factor and unroll factor and measure the result. Vectorization pragmas can help the user enumerate different vectorization factors and figure out the most performant one. There are relatively few possible configurations for each loop, and running the loop on typical inputs is something that humans can do that compilers can't.

Finally, there are situations when the scalar un-vectorized version of a loop performs better than the vectorized one. This could happen due to expensive vector operations like `gather/scatter` loads, masking, shuffles, etc. which compiler is required to use in order to make vectorization happen. Performance engineers could also try to disable vectorization in different ways. For the Clang compiler, it can be done via compiler options `-fno-vectorize` and `-fno-slp-vectorize`, or with a hint specific for a particular loop, e.g. `#pragma clang loop vectorize(enable)`.

9.4.2.5 Languages with Explicit Vectorization. Vectorization can also be achieved by rewriting parts of a program in a programming language that is dedicated to parallel computing. Those languages use special constructs and knowledge of the program's data to compile the code efficiently into parallel programs. Originally such languages were mainly used to offload work to specific processing units such as graphics processing units (GPU), digital signal processors (DSP), or field-programmable gate arrays (FPGAs). However, some of those programming models can also target your CPU (such as OpenCL and OpenMP).

One such parallel language is Intel® Implicit SPMD Program Compiler (ISPC),[171] which we will cover a bit in this

[167] Besides Profile Guided Optimizations (see Section 11.7).
[168] For more details read this blog post: https://travisdowns.github.io/blog/2020/01/17/avxfreq1.html.
[169] Study of AVX-512 downclocking: in VQSort readme
[170] Using Clang's optimization pragmas - https://easyperf.net/blog/2017/11/09/Multiversioning_by_trip_counts
[171] ISPC compiler: https://ispc.github.io/.

section. The ISPC language is based on the C programming language and uses the LLVM compiler infrastructure to emit optimized code for many different architectures. The key feature of ISPC is the "close to the metal" programming model and performance portability across SIMD architectures. It requires a shift from the traditional thinking of writing programs but gives programmers more control over CPU resource utilization.

Another advantage of ISPC is its interoperability and ease of use. ISPC compiler generates standard object files that can be linked with the code generated by conventional C/C++ compilers. ISPC code can be easily plugged in any native project since functions written with ISPC can be called as if it was C code.

Listing 48 shows a simple example of a function that we presented earlier in Listing 44, rewritten with ISPC. ISPC considers that the program will run in parallel instances, based on the target instruction set. For example, when using SSE with `float`s, it can compute 4 operations in parallel. Each program instance would operate on vector values of `i` being `(0,1,2,3)`, then `(4,5,6,7)`, and so on, effectively computing 4 sums at a time. As you can see, a few keywords not typical for C and C++ are used:

- The `export` keyword means that the function can be called from a C-compatible language.
- The `uniform` keyword means that a variable is shared between program instances.
- The `varying` keyword means that each program instance has its own local copy of the variable.
- The `foreach` is the same as a classic `for` loop except that it will distribute the work across the different program instances.

Since function `calcSum` must return a single value (a `uniform` variable) and our `sum` variable is a `varying`, we then need to *gather* the values of each program instance using the `reduce_add` function. ISPC also takes care of generating peeled and remainder loops as needed to take into account the data that is not correctly aligned or that is not a multiple of the vector width.

"Close to the metal" programming model: one of the problems with traditional C and C++ languages is that compiler doesn't always vectorize critical parts of code. Often times programmers resort to using compiler intrinsics (see Section 9.5), which bypasses compiler autovectorization but is generally difficult and requires updating when new instruction sets come along. ISPC helps to resolve this problem by assuming every operation is SIMD by default. For example, the ISPC statement `sum += array[i]` is implicitly considered as a SIMD operation that makes multiple additions in parallel. ISPC is not an autovectorizing compiler, and it does not automatically discover vectorization opportunities. Since the ISPC language is very similar to C and C++, it is much better than using intrinsics as it allows you to focus on the algorithm rather than the low-level instructions. Also, it has reportedly matched[Pharr & Mark, 2012] or beaten[172] hand-written intrinsics code in terms of performance.

Performance portability: ISPC can automatically detect features of your CPU to fully utilize all the resources available. Programmers can write ISPC code once and compile to many vector instruction sets, such as SSE4, AVX, and AVX2. ISPC can also emit code for different architectures like x86 CPU, ARM NEON, and has experimental support for GPU offloading.

9.5 Compiler Intrinsics

There are types of applications that have hotspots worth tuning heavily. However, compilers do not always do what we want in terms of generated code in those hot places. For example, a program does some computation in a loop which the compiler vectorizes in a suboptimal way. It usually involves some tricky or specialized algorithms, for which we can come up with a better sequence of instructions. It can be very hard or even impossible to make the compiler generate the desired assembly code using standard constructs of the C and C++ languages.

Hopefully, it's possible to force the compiler to generate particular assembly instructions without writing in low-level assembly language. To achieve that, one can use compiler intrinsics, which in turn are translated into specific assembly instructions. Intrinsics provide the same benefit as using inline assembly, but also they improve code readability, allow compiler type checking, assist instruction scheduling, and help reduce debugging. Example in Listing 49 shows how the same loop in function `foo` can be coded via compiler intrinsics (function `bar`).

[172] Some parts of the Unreal Engine which used SIMD intrinsics were rewritten using ISPC, which gave speedups: https://software.intel.com/content/www/us/en/develop/articles/unreal-engines-new-chaos-physics-system-screams-with-in-depth-intel-cpu-optimizations.html.

When compiled for the SSE target, both `foo` and `bar` will generate similar assembly instructions. However, there are several caveats. First, when relying on auto-vectorization, the compiler will insert all necessary runtime checks. For instance, it will ensure that there are enough elements to feed the vector execution units. Secondly, function `foo` will have a fallback scalar version of the loop for processing the remainder of the loop. And finally, most vector intrinsics assume aligned data, so `movaps` (aligned load) is generated for `bar`, while `movups` (unaligned load) is generated for `foo`. Keeping that in mind, developers using compiler intrinsics have to take care of safety aspects themselves.

When writing code using non-portable platform-specific intrinsics, developers should also provide a fallback option for other architectures. A list of all available intrinsics for the Intel platform can be found in this reference[173].

9.5.1 Wrapper Libraries for Intrinsics

The write-once, target-many model of ISPC is appealing. However, we may wish for tighter integration into C++ programs, for example interoperability with templates, or avoiding a separate build step and using the same compiler. Conversely, intrinsics offer more control, but at a higher development cost.

We can combine the advantages of both and avoid these drawbacks using a so-called embedded domain-specific language, where the vector operations are expressed as normal C++ functions. You can think of these functions as 'portable intrinsics', for example `Add` or `LoadU`. Even compiling your code multiple times, once per instruction set, can be done within a normal C++ library by using the preprocessor to 'repeat' your code with different compiler settings, but within unique namespaces. One example of this is the previously mentioned Highway library,[174] which only requires the C++11 standard.

Like ISPC, Highway also supports detecting the best available instruction sets, grouped into 'clusters' which on x86 correspond to Intel Core (S-SSE3), Nehalem (SSE4.2), Haswell (AVX2), Skylake (AVX-512), or Icelake/Zen4 (AVX-512 with extensions). It then calls your code from the corresponding namespace. Unlike intrinsics, the code remains readable (without prefixes/suffixes on each function) and portable.

Notice the explicit handling of remainders after the loop processes multiples of the vector sizes `Lanes(d)`. Although this is more verbose, it makes visible what is actually happening, and allows optimizations such as overlapping the last vector instead of relying on `MaskedLoad`, or even skipping the remainder entirely when `count` is known to be a multiple of the vector size.

Highway supports over 200 operations, which can be grouped into the following categories:

- Initialization
- Getting/setting lanes
- Getting/setting blocks
- Printing
- Tuples
- Arithmetic
- Logical
- Masks
- Comparisons
- Memory
- Cache control
- Type conversion
- Combine
- Swizzle/permute
- Swizzling within 128-bit blocks
- Reductions
- Crypto

For the full list of operations, see its documentation [175] and FAQ. You can also experiment with it in the online Compiler Explorer. Other libraries include Eigen, nsimd, SIMDe, VCL, and xsimd. Note that a C++ standardization effort starting with the Vc library resulted in std::experimental::simd, but this provides a very limited set of operations and as of this writing is only supported on the GCC 11 compiler.

[173] Intel Intrinsics Guide - https://software.intel.com/sites/landingpage/IntrinsicsGuide/.
[174] Highway library: https://github.com/google/highway
[175] Highway Quick Reference - https://github.com/google/highway/blob/master/g3doc/quick_reference.md

Questions and Exercises

1. Solve the following lab assignments using techniques we discussed in this chapter:

- `perf-ninja::function_inlining_1`
- `perf-ninja::vectorization` 1 & 2
- `perf-ninja::dep_chains` 1 & 2
- `perf-ninja::compiler_intrinsics` 1 & 2
- `perf-ninja::loop_interchange` 1 & 2
- `perf-ninja::loop_tiling_1`

2. Describe the steps you will take to find out if an application is using all the opportunities for utilizing SIMD code?
3. Practice doing loop optimizations manually on a real code (but don't commit it). Make sure that all the tests are still passing.
4. Suppose you're dealing with an application that has a very low IpCall (instructions per call) metric. What optimizations you will try to apply/force?
5. Run the application that you're working with on a daily basis. Find the hottest loop in the program. Is it vectorized? Is it possible to force compiler autovectorization? Bonus question: is the loop bottlenecked by dependency chains or execution throughput?

Chapter Summary

- Inefficient computations represent a significant portion of the bottlenecks in real-world applications. Modern compilers are very good at removing unnecessary computation overhead by performing many different code transformations. Still, there is a high chance that we can do better than what compilers can offer.
- In Chapter 9, we showed how one could search performance headrooms in a program by forcing certain code optimizations. We discussed such popular transformations as function inlining, loop optimizations, and vectorization.

Listing 32 Random Particle Motion on a 2D Surface

```
struct Particle {
  float x; float y; float velocity;
};

class XorShift32 {
  uint32_t val;
public:
  XorShift32 (uint32_t seed) : val(seed) {}
  uint32_t gen() {
    val ^= (val << 13);
    val ^= (val >> 17);
    val ^= (val << 5);
    return val;
  }
};

static float sine(float x) {
  const float B = 4 / PI_F;
  const float C = -4 / ( PI_F * PI_F);
  return B * x + C * x * std::abs(x);
}
static float cosine(float x) {
  return sine(x + (PI_F / 2));
}

/* Map degrees [0;UINT32_MAX) to radians [0;2*pi)*/
float DEGREE_TO_RADIAN = (2 * PI_D) / UINT32_MAX;

void particleMotion(vector<Particle> &particles,
                    uint32_t seed) {
 XorShift32 rng(seed);
 for (int i = 0; i < STEPS; i++)
  for (auto &p : particles) {
   uint32_t angle = rng.gen();
   float angle_rad = angle * DEGREE_TO_RADIAN;
   p.x += cosine(angle_rad) * p.velocity;
   p.y += sine(angle_rad) * p.velocity;
  }
}
```

```
.loop:
  eor    w0, w0, w0, lsl #13
  eor    w0, w0, w0, lsr #17
  eor    w0, w0, w0, lsl #5
  ucvtf  s1, w0
  fmov   s2, w9
  fmul   s2, s1, s2
  fmov   s3, w10
  fadd   s3, s2, s3
  fmov   s4, w11
  fmul   s5, s3, s3
  fmov   s6, w12
  fmul   s5, s5, s6
  fmadd  s3, s3, s4, s5
  ldp    s6, s4, [x1, #0x4]
  ldr    s5, [x1]
  fmadd  s3, s3, s4, s5
  fmov   s5, w13
  fmul   s5, s1, s5
  fmul   s2, s5, s2
  fmadd  s1, s1, s0, s2
  fmadd  s1, s1, s4, s6
  stp    s3, s1, [x1], #0xc
  cmp    x1, x16
  b.ne   .loop
```

Listing 33 Random Particle Motion on a 2D Surface

```
void particleMotion(vector<Particle> &particles,
                    uint32_t seed1, uint32_t seed2) {
  XorShift32 rng1(seed1);
  XorShift32 rng2(seed2);
  for (int i = 0; i < STEPS; i++) {
    for (int j = 0; j + 1 < particles.size(); j += 2) {
      uint32_t angle1 = rng1.gen();
      float angle_rad1 = angle1 * DEGREE_TO_RADIAN;
      particles[j].x += cosine(angle_rad1) * particles[j].velocity;
      particles[j].y += sine(angle_rad1)   * particles[j].velocity;
      uint32_t angle2 = rng2.gen();
      float angle_rad2 = angle2 * DEGREE_TO_RADIAN;
      particles[j+1].x += cosine(angle_rad2) * particles[j+1].velocity;
      particles[j+1].y += sine(angle_rad2)   * particles[j+1].velocity;
    }
    // remainder (not shown)
  }
}
```

Listing 34 Loop Invariant Code motion

```
for (int i = 0; i < N; ++i)                 for (int i = 0; i < N; ++i) {
  for (int j = 0; j < N; ++j)     =>          auto temp = c[i];
    a[j] = b[j] * c[i];                       for (int j = 0; j < N; ++j)
                                                a[j] = b[j] * temp;
                                            }
```

Listing 35 Loop Unrolling

```
for (int i = 0; i < N; ++i)                 for (int i = 0; i+1 < N; i+=2) {
  a[i] = b[i] * c[i];           =>            a[i]   = b[i]   * c[i];
                                              a[i+1] = b[i+1] * c[i+1];
                                            }
```

Listing 36 Loop Strength Reduction

```
for (int i = 0; i < N; ++i)                 int j = 0;
  a[i] = b[i * 10] * c[i];       =>         for (int i = 0; i < N; ++i) {
                                              a[i] = b[j] * c[i];
                                              j += 10;
                                            }
```

Listing 37 Loop Unswitching

```
for (i = 0; i < N; i++) {                   if (c)
  a[i] += b[i];                               for (i = 0; i < N; i++) {
  if (c)                          =>            a[i] += b[i];
    b[i] = 0;                                   b[i] = 0;
}                                             }
                                            else
                                              for (i = 0; i < N; i++) {
                                                a[i] += b[i];
                                              }
```

Listing 38 Loop Interchange

```
for (i = 0; i < N; i++)                    for (j = 0; j < N; j++)
  for (j = 0; j < N; j++)          =>      for (i = 0; i < N; i++)
    a[j][i] += b[j][i] * c[j][i];            a[j][i] += b[j][i] * c[j][i];
```

Listing 39 Loop Blocking

```
// linear traversal                        // traverse in 8*8 blocks
for (int i = 0; i < N; i++)                for (int ii = 0; ii < N; ii+=8)
  for (int j = 0; j < N; j++)    =>         for (int jj = 0; jj < N; jj+=8)
    a[i][j] += b[j][i];                      for (int i = ii; i < ii+8; i++)
                                              for (int j = jj; j < jj+8; j++)
                                               a[i][j] += b[j][i];
```

Listing 40 Loop Fusion and Distribution

```
for (int i = 0; i < N; i++)                for (int i = 0; i < N; i++) {
  a[i].x = b[i].x;                           a[i].x = b[i].x;
                                  =>         a[i].y = b[i].y;
for (int i = 0; i < N; i++)                }
  a[i].y = b[i].y;
```

Listing 41 Loop Unroll and Jam

```
for (int i = 0; i < N; i++)              for (int i = 0; i+1 < N; i+=2)
  for (int j = 0; j < M; j++)              for (int j = 0; j < M; j++) {
    diffs += a[i][j] - b[i][j];   =>         diffs1 += a[i][j]   - b[i][j];
                                             diffs2 += a[i+1][j] - b[i+1][j];
                                           }
                                         diffs = diffs1 + diffs2;
```

Listing 42 Cannot move strlen out of the loop

```
void foo(char* a, char* b) {
  for (int i = 0; i < strlen(a); ++i)
    b[i] = (a[i] == 'x') ? 'y' : 'n';
}
```

Listing 43 Vectorization: read-after-write dependence.

```
void vectorDependence(int *A, int n) {
  for (int i = 1; i < n; i++)
    A[i] = A[i-1] * 2;
}
```

Listing 44 Vectorization: floating-point arithmetic.

```
// a.cpp
float calcSum(float* a, unsigned N) {
  float sum = 0.0f;
  for (unsigned i = 0; i < N; i++) {
    sum += a[i];
  }
  return sum;
}
```

Listing 45 a.c

```
void foo(float* a, float* b, float* c, unsigned N) {
  for (unsigned i = 1; i < N; i++) {
    c[i] = b[i];
    a[i] = c[i-1];
  }
}
```

Listing 46 Vectorization: not beneficial.

```
// a.cpp
void stridedLoads(int *A, int *B, int n) {
  for (int i = 0; i < n; i++)
    A[i] += B[i * 3];
}
```

Listing 47 Vectorization: not beneficial.

```
// a.cpp
void stridedLoads(int *A, int *B, int n) {
#pragma clang loop vectorize(enable)
  for (int i = 0; i < n; i++)
    A[i] += B[i * 3];
}
```

Listing 48 ISPC version of summing elements of an array.

```
export uniform float calcSum(const uniform float array[],
                             uniform ptrdiff_t count)
{
    varying float sum = 0;
    foreach (i = 0 ... count)
        sum += array[i];
    return reduce_add(sum);
}
```

Listing 49 Compiler Intrinsics

```
void foo(float *a, float *b, float *c, unsigned N) {
  for (unsigned i = 0; i < N; i++)
    c[i] = a[i] + b[i];
}

#include <xmmintrin.h>

void bar(float *a, float *b, float *c, unsigned N) {
  __m128 rA, rB, rC;
  int i = 0;
  for (; i + 3 < N; i += 4){
    rA = _mm_load_ps(&a[i]);
    rB = _mm_load_ps(&b[i]);
    rC = _mm_add_ps(rA,rB);
    _mm_store_ps(&c[i], rC);
  }
  for (; i < N; i++) // remainder
    c[i] = a[i] + b[i];
}
```

Listing 50 Highway version of summing elements of an array.

```
#include <hwy/highway.h>

float calcSum(const float* HWY_RESTRICT array, size_t count) {
  const ScalableTag<float> d;  // type descriptor; no actual data
  auto sum = Zero(d);
  size_t i = 0;
  for (; i + Lanes(d) <= count; i += Lanes(d)) {
    sum = Add(sum, LoadU(d, array + i));
  }
  sum = Add(sum, MaskedLoad(FirstN(d, count - i), d, array + i));
  return ReduceSum(d, sum);
}
```

10 Optimizing Branch Prediction

So far we've been talking about optimizing memory accesses and computations. However, there is another important category of performance bottlenecks that we haven't discussed yet. It is related to speculative execution, a feature that is present in all modern high-performance CPU cores. To refresh your memory, turn to Section 3.3.3 where we discussed how speculative execution can be used to improve performance. In this chapter, we will explore techniques to reduce the number of branch mispredictions.

In general, modern processors are very good at predicting branch outcomes. They not only follow static prediction rules but also detect dynamic patterns. Usually, branch predictors save the history of previous outcomes for the branches and try to guess what will be the next result. However, when the pattern becomes hard for the CPU branch predictor to follow, it may hurt performance.

Mispredicting a branch can add a significant speed penalty when it happens regularly. When such an event happens, a CPU is required to clear all the speculative work that was done ahead of time and later was proven to be wrong. It also needs to flush the pipeline and start filling it with instructions from the correct path. Typically, modern CPUs experience from 10 to 20 cycles penalty as a result of a branch misprediction. The exact number of cycles depends on the microarchitecture design, namely, on the depth of the pipeline and the mechanism used to recover from the mispredicts.

Branch predictors use caches and history registers and therefore are susceptible to the issues pertaining to caches, namely three C's:

- **Compulsory misses**: mispredictions may happen on the first dynamic occurence of the branch when static prediction is employed and no dynamic history is available.
- **Capacity misses**: mispredictions arising from the loss of dynamic history due to very high number of branches in the program or exceedingly long dynamic pattern.
- **Conflict misses**: branches are mapped into cache buckets (associative sets) using a combination of their virtual and/or physical addresses. If too many active branches are mapped to the same set, the loss of history can occur. Another instance of a conflict miss is false sharing when two independent branches are mapped to the same cache entry and interfere with each other potentially degrading the prediction history.

A program will always take a non-zero number of branch mispredictions. You can find out how much a program suffers from branch mispredictions by looking at TMA `Bad Speculation` metric. It is normal for a general purpose application to have a `Bad Speculation` metric in the range of 5-10%. Our recommendation is to pay a close attention once this metric goes higher than 10%.

Since branch predictors are good at finding patterns, old advice for optimizing branch prediction is no longer valid. In the past, developers had an option of providing a prediction hint to the processor in the form of an encoding prefix to the branch instruction (`0x2E: Branch Not Taken`, `0x3E: Branch Taken`). This could potentially improve performance on older microarchitectures, like Pentium 4. While using those branch prefixes still gives valid x86/x64 assembly, it won't produce gains on modern processors. [TODO]

One indirect way to reduce branch mispredictions is to straighten the code using source-based and compiler-based techniques. PGO and BOLT are effective at reducing branch mispredictions thanks to improving fallthrough rates that alleviates the pressure on branch predictor structures. We will discuss those techniques in the next chapter.

So perhaps the only direct way to get rid of branch mispredictions is to get rid of the branch itself. In the two subsequent sections, we will take a look at how branches can be replaced with lookup tables and predication.

There is a conventional wisdom that never taken branches are transparent to the branch prediction and can't affect performance, and therefore it doesn't make much sense to remove them, at least from prediction perspective. However, contrary to the wisdom, an experiment conducted by authors of BOLT optimizer demonstrated that replacing never taken branches with equal-sized no-ops in a large code footprint application, such as Clang C++ compiler, leads to approximately 5% speedup on modern Intel CPUs. So it still pays to try to eliminate all branches.

10.1 Replace Branches with Lookup

One way to avoid frequently mispredicted branches is to use lookup tables. An example of code when such transformation might be profitable is shown in Listing 51. As usual, the original version is on the left while the improved version is on the right. Function `mapToBucket` maps values in the `[0-50)` range into corresponding five buckets, and returns `-1` for values that are out of this range. For uniformly distributed values of `v`, we will have an equal probability for `v` to fall into any of the buckets. In the generated assembly for the original version, we will likely see many branches, which could have high misprediction rates. Hopefully, it's possible to rewrite the function `mapToBucket` using a single array lookup, as shown on the right.

Listing 51 Replacing branches with lookup tables.

```
int8_t mapToBucket(unsigned v) {
  if (v >= 0  && v < 10) return 0;
  if (v >= 10 && v < 20) return 1;
  if (v >= 20 && v < 30) return 2;
  if (v >= 30 && v < 40) return 3;
  if (v >= 40 && v < 50) return 4;
  return -1;
}
```

=>

```
int8_t buckets[50] = {
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
  4, 4, 4, 4, 4, 4, 4, 4, 4, 4};

int8_t mapToBucket(unsigned v) {
  if (v < (sizeof(buckets) / sizeof(int8_t)))
    return buckets[v];
  return -1;
}
```

For the improved version of `mapToBucket` on the right, a compiler will likely generate a single branch instruction that guards against out-of-bounds access to the `buckets` array. A typical hot path through this function will execute the untaken branch and one load instruction. The branch will be well-predicted by the CPU branch predictor since we expect most of the input values to fall into the range covered by the `buckets` array. And the lookup will also be fast since the `buckets` array is small and likely to be in the L1-d cache.

If we need to map a bigger range of values, say `[0-1M)`, allocating a very large array is not practical. In this case, we might use interval map data structures that accomplish that goal using much less memory but logarithmic lookup complexity. Readers can find existing implementations of interval map container in Boost[176] and LLVM[177].

10.2 Replace Branches with Arithmetic

In some scenarios, branches can be replaced with arithmetic. The code in Listing 51, can also be rewritten using a simple arithmetic formula, as shown in Listing 52. For this code, Clang-17 compiler replaces expensive division with a much cheaper multiplication operation.

Listing 52 Replacing branches with arithmetic.

```
int8_t mapToBucket(unsigned v) {
  constexpr unsigned BucketRangeMax = 50;
  if (v < BucketRangeMax)
    return v / 10;
  return -1;
}
```

As of year 2023, compilers are usually unable to find these shortcuts on their own, so it is up to the programmer to do it manually. If you can find a way to replace a branch with arithmetic, you will likely see a performance improvement. Unfortunately, this is not always possible.

[176] C++ Boost `interval_map` - https://www.boost.org/doc/libs/1_65_0/libs/icl/doc/html/boost/icl/interval_map.html
[177] LLVM's `IntervalMap` - https://llvm.org/doxygen/IntervalMap_8h_source.html

10.3 Replace Branches with Predication

Some branches could be effectively eliminated by executing both parts of the branch and then selecting the right result (*predication*). Example of code when such transformation might be profitable is shown on Listing 53. If TMA suggests that the `if (cond)` branch has a very high number of mispredictions, you can try to eliminate the branch by doing the transformation shown on the right.

Listing 53 Predicating branches.

```
int a;                                                  int x = computeX();
if (cond) { /* frequently mispredicted */     =>        int y = computeY();
  a = computeX();                                       int a = cond ? x : y;
} else {
  a = computeY();
}
```

For the code on the right, the compiler can replace the branch that comes from the ternary operator, and generate a `CMOV` x86 instruction instead. A `CMOVcc` instruction checks the state of one or more of the status flags in the `EFLAGS` register (`CF, OF, PF, SF` and `ZF`) and performs a move operation if the flags are in a specified state or condition. Similar transformation can be done for floating-point numbers with `FCMOVcc,VMAXSS/VMINSS` instructions. Listing 54 shows assembly listings for the original and the branchless version.

Listing 54 Predicating branches - x86 assembly code.

```
# original version                      # branchless version
400504: test edi,edi                    400537: mov eax,0x0
400506: je 400514                       40053c: call <computeX> # compute x; a = x
400508: mov eax,0x0                     400541: mov ebp,eax     # ebp = x
40050d: call <computeX>         =>      400543: mov eax,0x0
400512: jmp 40051e                      400548: call <computeY> # compute y; a = y
400514: mov eax,0x0                     40054d: test ebx,ebx    # test cond
400519: call <computeY>                 40054f: cmovne eax,ebp  # override a with x if needed
40051e: mov edi,eax
```

In contrast with the original version, the branchless version doesn't have jump instructions. However, the branchless version calculates both `x` and `y` independently, and then selects one of the values and discards the other. While this transformation eliminates the penalty of a branch misprediction, it is potentially doing more work than the original code. Performance improvement, in this case, very much depends on the characteristics of `computeX` and `computeY` functions. If the functions are small and the compiler is able to inline them, then it might bring noticeable performance benefits. If the functions are big, it might be cheaper to take the cost of a branch mispredict than to execute both functions.

It is important to note that predication does not always benefit the performance of the application. The issue with predication is that it limits the parallel execution capabilities of the CPU. For the original version of the code, the CPU can predict that the branch will be taken, speculatively call `computeX` and continue executing the rest of the program. This type of speculation is not possible for the branchless version as the CPU has to wait for the result of the `CMOVNE` instruction to proceed.

The typical example of the tradeoffs involved when choosing between the regular and the branchless versions of the code is binary search:[178]

- For a search over a large array that doesn't fit in CPU caches, a branch-based binary search version performs better because the penalty of a branch misprediction is low comparing to the latency of memory accesses (which are high because of the cache misses). Because of the branches in place, the CPU can speculate on their outcome, which allows loading the array element from the current iteration and the next one at the same time. It doesn't end there: the speculation continues, and you might have multiple loads in flight at the same time.

[178] Discussion on branchless binary search - https://stackoverflow.com/a/54273248.

- The situation is reversed for small arrays that fit in CPU caches. The branchless search still has all the memory accesses serialized, as explained earlier. But this time, the load latency is small (only a handful of cycles) since the array fits in CPU caches. The branch-based binary search suffers constant mispredictions, which cost roughly 10-20 cycles. In this case, the cost of a mispredict is much more than the cost of a memory access, so the benefits of speculative execution are hindered. The branchless version usually ends up being faster in this case.

The binary search is a neat example that shows how one can reason about when choosing between standard and branchless implementation. The real-world scenario can be more difficult to analyze, so again, measure to find out if it would be beneficial to replace branches in your case.

Without profiling data, compilers don't have visibility into the misprediction rates. As a result, compilers usually prefer to generate branches, i.e. original version, by default. They are conservative at using predication and may resist generating `CMOV` instructions even in simple cases. Again, the tradeoffs are complicated, and it is hard to make the right decision without the runtime data. HW-based PGO (see [#sec:secPGO]) will be a huge step forward here. Also, there is a way to indicate to the compiler that a branch condition is unpredictable by hardware mechanisms. Starting from Clang-17, the compiler now respect a `__builtin_unpredictable`, which can be very effective at replacing unpredictable branches with `CMOV` x86 instructions. For example:

```
if (__builtin_unpredictable(x != 2))
  y = 0;
if (__builtin_unpredictable(x == 3))
  y = 1;
```

Questions and Exercises

1. Solve the following lab assignments using techniques we discussed in this chapter:

- `perf-ninja::branches_to_cmov_1`
- `perf-ninja::lookup_tables_1`
- `perf-ninja::virtual_call_mispredict`
- `perf-ninja::conditional_store_1`

2. Run the application that you're working with on a daily basis. Collect the TMA breakdown and check the `BadSpeculation` metric. Look at the code that is attributed with the most number of branch mispredictions. Is there a way to avoid branches using techniques we discussed in this chapter?

Coding exercise: write a microbenchmark that will experience 50% misprediction rate or get as close as possible. Your goal is to write a code in which half of all branch instructions are mispredicted. That is not as simple as you may think. Some hints and ideas: - Branch misprediction rate is measured as `BR_MISP_RETIRED.ALL_BRANCHES / BR_INST_RETIRED.ALL_BRANCHES`. - If you're coding in C++, you can use 1) google benchmark similar to perf-ninja, or 2) write a regular console program and collect CPU counters with Linux `perf`, or 3) integrate libpfm into the microbenchmark (see Section 5.3.4). - There is no need to invent some complicated algorithm. A simple approach would be to generate a pseudo-random number in the range `[0;100)` and check if it is less than 50. Random numbers can be pregenerated ahead of time. - Keep in mind that modern CPUs can remember long (but still limited) sequences of branch outcomes.

Chapter Summary

- Modern processors are very good at predicting branch outcomes. So, we recommend starting the work on fixing branch mispredictions only when the TMA report points to a high `Bad Speculation` metric.
- When branch outcome patterns become hard for the CPU branch predictor to follow, the performance of the application may suffer. In this case, the branchless version of an algorithm can be more performant. In this chapter, we showed how branches could be replaced with lookup tables, arithmetic, and predication. In some situations, it is also possible to use compiler intrinsics to eliminate branches, as shown in [Kapoor, 2009].
- Branchless algorithms are not universally beneficial. Always measure to find out what works better in your specific case.

11 Machine Code Layout Optimizations

The CPU Front-End (FE) is responsible for fetching and decoding instructions and delivering them to the out-of-order Back-End. As the newer processors get more execution "horsepower", CPU FE needs to be as powerful to keep the machine balanced. If the FE cannot keep up with supplying instructions, the BE will be underutilized, and the overall performance will suffer. That's why the FE is designed to always run well ahead of the actual execution to smooth out any hiccups that may occur and always have instructions ready to be executed. For example, Intel Skylake, released in 2016, can fetch up to 16 instructions per cycle.

Most of the time, inefficiencies in the CPU FE can be described as a situation when the Back-End is waiting for instructions to execute, but the FE is not able to provide them. As a result, CPU cycles are wasted without doing any actual useful work. Recall that modern CPUs can process multiple instructions every cycle, nowadays ranging from 4- to 8-wide. Situations when not all available slots are filled happen very often. This represents a source of inefficiency for applications in many domains, such as databases, compilers, web browsers, and many others.

The TMA methodology captures FE performance issues in the `Front-End Bound` metric. It represents the percentage of cycles when the CPU FE is not able to deliver instructions to the BE, while it could have accepted them. Most of the real-world applications experience a non-zero 'Front-End Bound' metric, meaning that some percentage of running time will be lost on suboptimal instruction fetching and decoding. Below 10% is the norm. If you see the "Front-End Bound" metric being more than 20%, it's definitely worth to spend time on it.

There could be many reasons why FE cannot deliver instructions to the execution units. Most of the time, it is due to suboptimal code layout, which leads to the poor I-cache and ITLB utilization. Applications with a large codebase, e.g. millions lines of code, are especially vulnerable to FE performance issues. In this chapter, we will take a look at some typical optimizations to improve machine code layout and increase the overall performance of the program.

11.1 Machine Code Layout

When a compiler translates source code into machine code, it generates a linear byte sequence. Listing 55 shows an example of a binary layout for a small snippet of C++ code. Once compiler finished generating assembly instructions, it needs to encode them and lay out in memory sequentially.

Listing 55 Example of machine code layout

```
  C++ Code        Assembly Listing      Disassembled Machine Code
  ........        ................      .........................
if (a <= b)     |   ; a is in edi     | 401125 cmp esi, edi
  bar();        |   ; b is in esi     | 401128 jb 401131
else            |   cmp esi, edi      | 40112a call bar
  baz();        |   jb .label1        | 40112f jmp 401136
                |   call bar()        | 401131 call baz
                |   jmp .label2       | 401136 ...
                | .label1:            |
                |   call baz()        |
                | .label2:            |
                |   ...               |
```

The way code is placed in a binary is called *machine code layout*. Note that for the same program, it's possible to lay out the code in many different ways. For the code in Listing 55, compiler may decide to reverse the branch in such a way that a call to `baz` will come first. Also, bodies of the functions `bar` and `baz` can be placed in two different orders: we can place `bar` first in the binary and then `baz` or reverse the order. This affects offsets at which instructions will be placed in memory, which in turn may affect the performance of the generated binary as you will see later. In the following sections of this chapter, we will take a look at some typical optimizations for the machine code layout.

11.2 Basic Block

A basic block is a sequence of instructions with a single entry and a single exit. Figure 66 shows a simple example of a basic block, where `MOV` instruction is an entry, and `JA` is an exit instruction. While a basic block can have one or many predecessors and successors, no instruction in the middle can enter or exit a basic block.

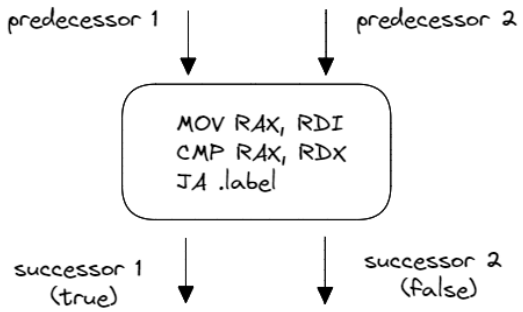


Figure 66: Basic Block of assembly instructions.

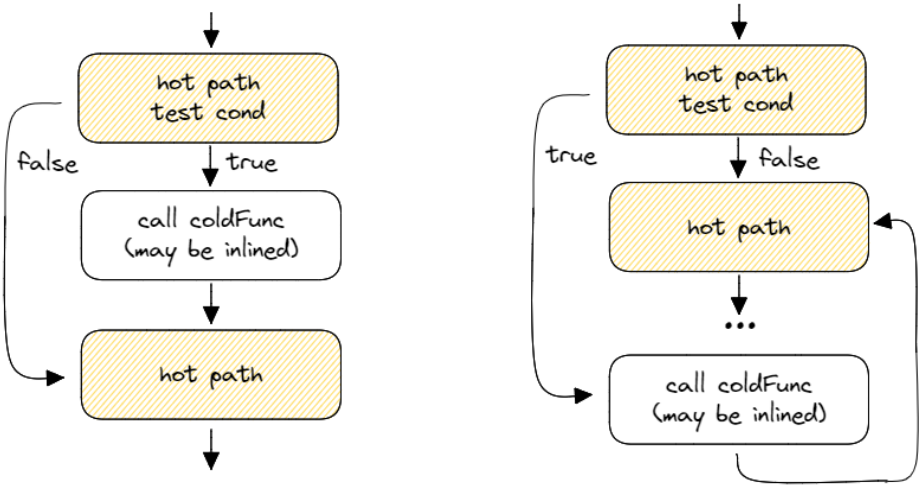
It is guaranteed that every instruction in the basic block will be executed exactly once. This is an important property that is leveraged by many compiler transformations. For example, it greatly reduces the problem of control flow graph analysis and transformations since, for some class of problems, we can treat all instructions in the basic block as one entity.

11.3 Basic Block Placement

Suppose we have a hot path in the program that has some error handling code (`coldFunc`) in between:

```
// hot path
if (cond)
  coldFunc();
// hot path again
```

Figure 67 shows two possible physical layouts for this snippet of code. Figure 67a is the layout most compiler will emit by default, given no hints provided. The layout that is shown in Figure 67b can be achieved if we invert the condition `cond` and place hot code as fall through.



(a) default layout

(b) improved layout

Figure 67: Two versions of machine code layout for the snippet of code above.

Which layout is better? Well, it depends on whether `cond` is usually true or false. If `cond` is usually true, then we would better choose the default layout because otherwise, we would be doing two jumps instead of one. Also, in the general case, if `coldFunc` is a relatively small function, we would want to have it inlined. However, in this particular example, we know that coldFunc is an error handling function and is likely not executed very often. By choosing layout 67b, we maintain fall through between hot pieces of the code and convert taken branch into not taken one.

There are a few reasons why the layout presented in Figure 67b performs better. First of all, layout in Figure 67b makes better use of the instruction and uop-cache (DSB, see Section 3.8.1). With all hot code contiguous, there is no cache line fragmentation: all the cache lines in the L1I-cache are used by hot code. The same is true for the uop-cache since it caches based on the underlying code layout as well. Secondly, taken branches are also more expensive for the fetch unit. The Front-End of a CPU fetches contiguous chunks of bytes, so every taken jump means the bytes after the jump are useless. This reduces the maximum effective fetch throughput. Finally, on some architectures, not taken branches are fundamentally cheaper than taken. For instance, Intel Skylake CPUs can execute two untaken branches per cycle but only one taken branch every two cycles.[179]

To suggest a compiler to generate an improved version of the machine code layout, one can provide a hint using `[[likely]]` and `[[unlikely]]` attributes, which is available since C++20. The code that uses this hint will look like this:

```
// hot path
if (cond) [[unlikely]]
  coldFunc();
// hot path again
```

In the code above, `[[unlikely]]` hint will instruct the compiler that `cond` is unlikely to be true, so compiler should adjust the code layout accordingly. Prior to C++20, developers could have used `__builtin_expect`[180] construct and they usually created `LIKELY` wrapper hints themselves to make the code more readable. For example:

```
#define LIKELY(EXPR)   __builtin_expect((bool)(EXPR), true)
#define UNLIKELY(EXPR) __builtin_expect((bool)(EXPR), false)
// hot path
if (UNLIKELY(cond)) // NOT
  coldFunc();
// hot path again
```

Optimizing compilers will not only improve code layout when they encounter "likely/unlikely" hints. They will also leverage this information in other places. For example, when `[[unlikely]]` attribute is applied, the compiler will prevent inlining `coldFunc` since it now knows that it is unlikely to be executed often and it's more beneficial to optimize it for size, i.e., just leave a `CALL` to this function. Inserting `[[likely]]` attribute is also possible for a switch statement as presented in Listing 56.

Listing 56 Likely attribute used in a switch statement

```
for (;;) {
  switch (instruction) {
              case NOP: handleNOP(); break;
    [[likely]] case ADD: handleADD(); break;
              case RET: handleRET(); break;
    // handle other instructions
  }
}
```

Using this hint, a compiler will be able to reorder code a little bit differently and optimize the hot switch for faster processing of `ADD` instructions.

[179] Though, there is a special small loop optimization that allows very small loops to have one taken branch per cycle.
[180] More about builtin-expect here: https://llvm.org/docs/BranchWeightMetadata.html#builtin-expect.

11.4 Basic Block Alignment

Sometimes performance can significantly change depending on the offset at which instructions are laid out in memory. Consider a simple function presented in Listing 57 along with a corresponding machine code when compiled with `-O3 -march=core-avx2 -fno-unroll-loops`. Loop unrolling is disabled for illustrating the idea.

Listing 57 Basic block alignment

```
void benchmark_func(int* a) {
  for (int i = 0; i < 32; ++i)
    a[i] += 1;
}
```

```
00000000004046a0 <_Z14benchmark_funcPi>:
4046a0: mov rax,0xffffffffffffff80
4046a7: vpcmpeqd ymm0,ymm0,ymm0
4046ab: nop DWORD [rax+rax+0x0]
4046b0: vmovdqu ymm1,YMMWORD [rdi+rax+0x80] # loop begins
4046b9: vpsubd ymm1,ymm1,ymm0
4046bd: vmovdqu YMMWORD [rdi+rax+0x80],ymm1
4046c6: add rax,0x20
4046ca: jne 4046b0                          # loop ends
4046cc: vzeroupper
4046cf: ret
```

The code itself is pretty reasonable, but its layout is not perfect (see Figure 68a). Instructions that correspond to the loop are highlighted with yellow hachure. As well as for data caches, instruction cache lines are 64 bytes long. On Figure 68 thick boxes denote cache line borders. Notice that the loop spans multiple cache lines: it begins on the cache line `0x80-0xBF` and ends in the cache-line `0xC0-0xFF`. To fetch instructions that are executed in the loop, a processor needs to read two cache lines. These kinds of situations usually cause performance problems for the CPU Front-End, especially for the small loops like presented above.

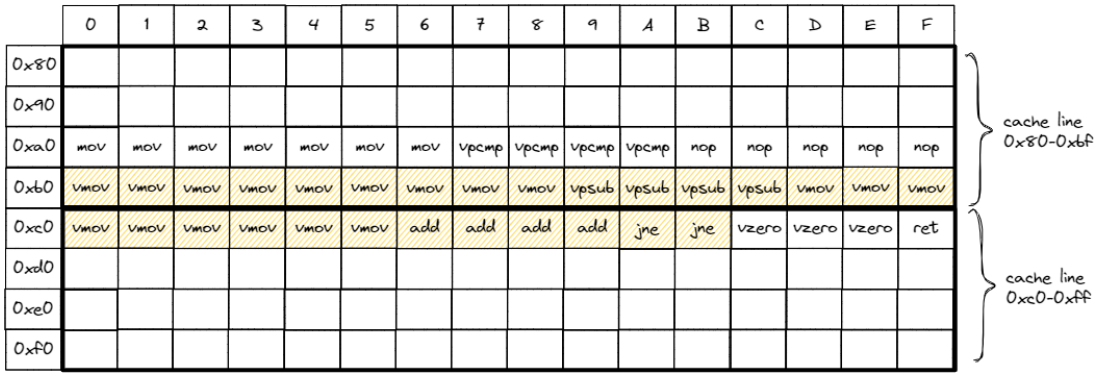
To fix this, we can shift the loop instructions forward by 16 bytes using NOPs so that the whole loop will reside in one cache line. Figure 68b shows the effect of doing this with NOP instructions highlighted in blue. Interestingly, the performance impact is visible even you run nothing but this hot loop in a microbenchmark. It is somewhat puzzling since the amount of code is tiny and it shouldn't saturate the L1I-cache size on any modern CPU. The reason for the better performance of the layout in Figure 68b is not trivial to explain and will involve a fair amount of microarchitectural details, which we don't discuss in this book. Interested readers can find more information in the article "Code alignment issues" on the easyperf blog.[181]

By default, the LLVM compiler recognizes loops and aligns them at 16B boundaries, as we saw in Figure 68a. To reach the desired code placement for our example, as shown in Figure 68b, one can use the `-mllvm -align-all-blocks=5` option that will align every basic block in an object file at a 32 bytes boundary. However, be careful with using this option, as it can easily degrade performance in other places. This option inserts NOPs on the executed path, which can add overhead to the program, especially if they stand on a critical path. NOPs do not require execution; however, they still require to be fetched from memory, decoded, and retired. The latter additionally consumes space in FE data structures and buffers for bookkeeping, similar to all other instructions. There are other less intrusive options in the LLVM compiler that can be used to control basic block alignment, which you can check in the easyperf blog post.[182]

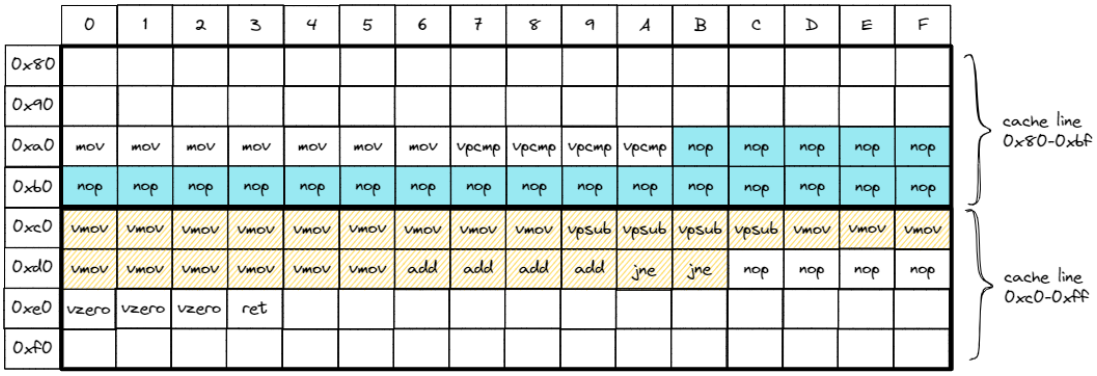
A recent addition to the LLVM compiler is the new `[[clang::code_align()]]` loop attribute, which allows developers to specify the alignment of a loop in the source code. This gives a very fine-grained control over machine code layout. Before this attribute was introduced, developers had to resort to some less practical solutions like injecting `asm(".align 64;")` statements of inline assembly in the source code. The following code shows how the new Clang attribute can be used to align a loop at a 64 bytes boundary:

```
void benchmark_func(int* a) {
  [[clang::code_align(64)]]
  for (int i = 0; i < 32; ++i)
    a[i] += 1;
}
```

[181] "Code alignment issues" - https://easyperf.net/blog/2018/01/18/Code_alignment_issues
[182] "Code alignment options in llvm" - https://easyperf.net/blog/2018/01/25/Code_alignment_options_in_llvm



(a) default layout



(b) improved layout

Figure 68: Two different code layouts for the loop in Listing 57.

Even though CPU architects work hard to minimize the impact of machine code layout, there are still cases when code placement (alignment) can make a difference in performance. Machine code layout is also one of the main sources of noise in performance measurements. It makes it harder to distinguish a real performance improvement or regression from the accidental one, that was caused by the change in the code layout.

11.5 Function Splitting

The idea behind function splitting is to separate hot code from the cold. Such transformation is also often called *function outlining*. This optimization is beneficial for relatively big functions with complex control flow graph and large chunks of cold code inside a hot path. An example of code when such transformation might be profitable is shown in Listing 58. To remove cold basic blocks from the hot path, we cut and paste them into a new function and create a call to it.

Notice, we disable inlining of cold functions by using `noinline` attribute. Because without it, a compiler may decide to inline it, which will effectively undo our transformation. Alternatively, we could apply the `[[unlikely]]` macro (see Section 11.3) on both `cond1` and `cond2` branches to convey to the compiler that inlining `cold1` and `cold2` functions is not desired.

Figure 69 gives a graphical representation of this transformation. Because we left just a `CALL` instruction inside the hot path, it's likely that the next hot instruction will reside in the same cache line as the previous one. This improves the utilization of CPU Front-End data structures such as I-cache and DSB.

Outlined functions should be created outside of `.text` segment, for example in `.text.cold`. This improves memory footprint if the function is never called since it won't be loaded into memory at runtime.

11.6 Function Reordering

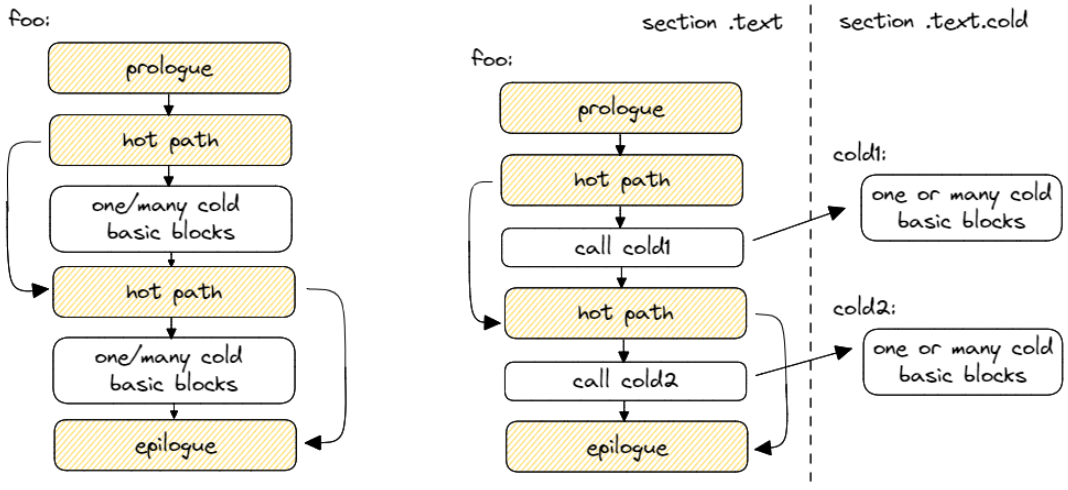
Following the principles described in previous sections, hot functions can be grouped together to further improve the utilization of caches in the CPU Front-End. When hot functions are grouped together, they start sharing cache

Listing 58 Function splitting: cold code outlined to the new functions.

```
void foo(bool cond1, bool cond2) {
  // hot path
  if (cond1) {
    /* large amount of cold code (1) */
  }
  // hot path
  if (cond2) {
    /* large amount of cold code (2) */
  }
}
```

=>

```
void foo(bool cond1, bool cond2) {
  // hot path
  if (cond1) {
    cold1();
  }
  // hot path
  if (cond2) {
    cold2();
  }
}
void cold1() __attribute__((noinline))
  { /* large amount of cold code (1) */ }
void cold2() __attribute__((noinline))
  { /* large amount of cold code (2) */ }
```



(a) default layout (b) improved layout

Figure 69: Splitting cold code into a separate function.

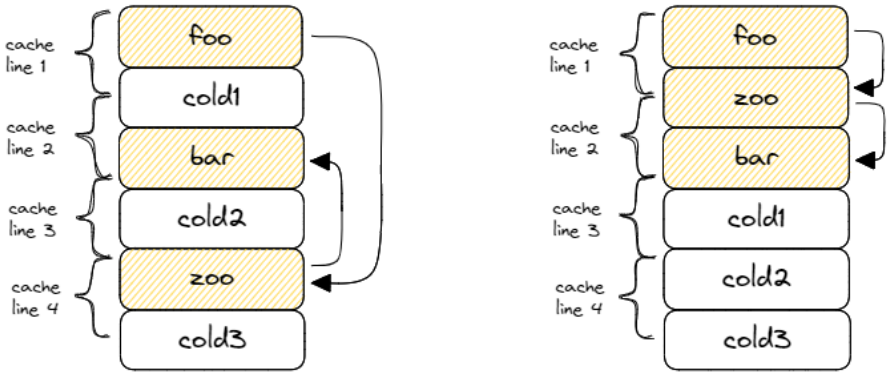
lines, which reduces the *code footprint*, total number of cache lines a CPU needs to fetch.

Figure 70 gives a graphical representation of reordering hot functions `foo`, `bar`, and `zoo`. The arrows on the image show the most frequent call pattern, i.e. `foo` calls `zoo`, which in turn calls `bar`. In the default layout (see Figure 70a), hot functions are not adjacent to each other with some cold functions placed between them. Thus the sequence of two function calls (`foo` -> `zoo` -> `bar`) requires four cache line reads.

We can rearrange the order of the functions such that hot functions are placed close to each other (see Figure 70b). In the improved version, the code of `foo`, `bar` and `zoo` functions fits in three cache lines. Also, notice that function `zoo` now is placed between `foo` and `bar` according to the order in which function calls are being made. When we call `zoo` from `foo`, the beginning of `zoo` is already in the I-cache.

Similar to previous optimizations, function reordering improves the utilization of I-cache and DSB-cache. This optimization works best when there are many small hot functions.

The linker is responsible for laying out all the functions of the program in the resulting binary output. While developers can try to reorder functions in a program themselves, there is no guarantee on the desired physical layout. For decades people have been using linker scripts to achieve this goal. Still, this is the way to go if you are using the GNU linker. The Gold linker (`ld.gold`) has an easier approach to this problem. To get the desired ordering of functions in the binary with the Gold linker, one can first compile the code with the `-ffunction-sections` flag,



(a) default layout (b) improved layout

Figure 70: Reordering hot functions.

which will put each function into a separate section. Then use `--section-ordering-file=order.txt` option to provide a file with a sorted list of function names that reflects the desired final layout. The same feature exists in the LLD linker, which is a part of LLVM compiler infrastructure and is accessible via the `--symbol-ordering-file` option.

An interesting approach to solving the problem of grouping hot functions together was introduced in 2017 by engineers from Meta. They implemented a tool called HFSort[183], that generates the section ordering file automatically based on profiling data [Ottoni & Maher, 2017]. Using this tool, they observed a 2% performance speedup of large distributed cloud applications like Facebook, Baidu, and Wikipedia. HFSort has been integrated into Meta's HHVM, LLVM BOLT, and LLD linker[184]. Since then, the algorithm has been superseded first by HFSort+, and most recently by Cache-Directed Sort (CDSort[185]), with more improvements for workloads with large code footprint.

11.7 Profile Guided Optimizations

Compiling a program and generating optimal assembly is all about heuristics. Code transformation algorithms have many corner cases that aim for optimal performance in specific situations. For a lot of decisions that a compiler makes, it tries to guess the best choice based on some typical cases. For example, when deciding whether a particular function should be inlined, the compiler could take into account the number of times this function will be called. The problem is that compiler doesn't know that beforehand. It first needs to run the program to find out. Without any runtime information, the compiler will have to make a guess.

Here is when profiling information becomes handy. Given profiling information, compiler can make better optimization decisions. There is a set of transformations in most compilers that can adjust their algorithms based on profiling data fed back to them. This set of transformations is called Profile Guided Optimizations (PGO). When profiling data is available, a compiler can use it to direct optimizations. Otherwise, it will fall back to using its standard algorithms and heuristics. Sometimes in literature, you can find the term Feedback Directed Optimizations (FDO), which refers to the same thing as PGO.

Figure 71 shows a traditional workflow of using PGO, also called *instrumented PGO*. First, you compile your program and tell the compiler to automatically instrument the code. This will insert some bookkeeping code into functions to collect runtime statistics. Second step is to run the instrumented binary with an input data that represents a typical workload for your application. This will generate the profiling data, a new file with runtime statistics. It is a raw dump file with information about function call counts, loop iteration counts, and other basic block hit counts. The final step in this workflow is to recompile the program with the profiling data to produce optimized executable.

Developers can enable PGO instrumentation (step 1) in the LLVM compiler by building the program with the `-fprofile-instr-generate` option. This will instruct the compiler to instrument the code, which will collect profiling information at runtime. After that, the LLVM compiler can consume profiling data with the

[183] HFSort - https://github.com/facebook/hhvm/tree/master/hphp/tools/hfsort
[184] HFSort in LLD - https://github.com/llvm-project/lld/blob/master/ELF/CallGraphSort.cpp
[185] Cache-Directed Sort in LLVM - https://github.com/llvm/llvm-project/blob/main/llvm/lib/Transforms/Utils/CodeLayout.cpp

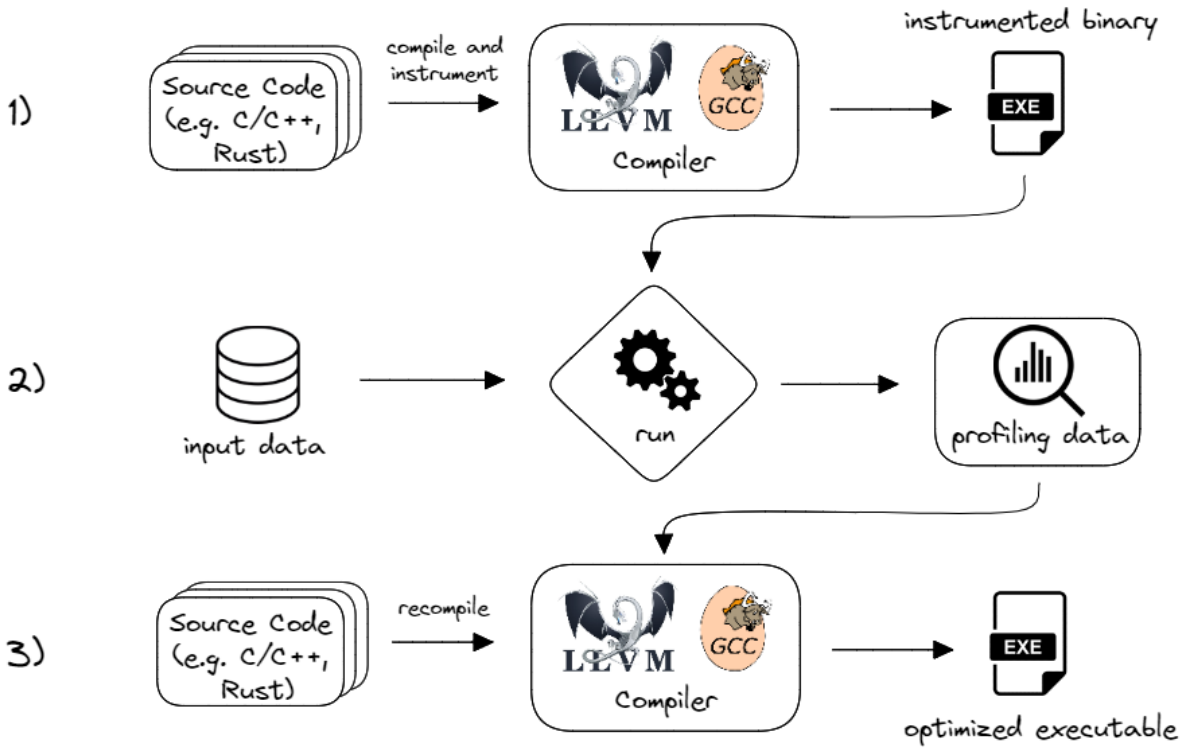


Figure 71: Instrumented PGO workflow.

`-fprofile-instr-use` option to recompile the program and output a PGO-tuned binary. The guide for using PGO in clang is described in the documentation.[186] GCC compiler uses different set of options: `-fprofile-generate` and `-fprofile-use` as described in the documentation.[187]

PGO helps the compiler to improve function inlining, code placement, register allocation, and other code transformations. PGO is primarily used in projects with a large codebase, for example, Linux kernel, compilers, databases, web browsers, video games, productivity tools and others. For applications with millions lines of code, it is the only practical way to improve machine code layout. It is not uncommon to see performance of production workloads increase by 10-25% from using Profile Guided Optimizations.

While many software projects adopted instrumented PGO as a part of their build process, the rate of adoption is still very low. There are a few reasons for that. The primary reason is a huge runtime overhead of instrumented executables. Running an instrumented binary and collecting profiling data frequently incurs 5-10x slowdown, which makes the build step longer and prevents profile collection directly from production systems, whether on client devices or in the cloud. Unfortunately, you cannot collect the profiling data once and use it for all the future builds. As the source code of an application evolves, the profile data becomes stale (out of sync) and needs to be recollected.

Another caveat in the PGO flow is that a compiler should only be trained using representative scenarios of how your application will be used. Otherwise, you may end up degrading program's preformance. The compiler "blindly" uses the profile data that you provided. It assumes that the program will always behave the same no matter what the input data is. Users of PGO should be careful about choosing the input data they will use for collecting profiling data (step 2) because while improving one use case of the application, others may be pessimized. Luckily, it doesn't have to be exactly a single workload since profile data from different workloads can be merged together to represent a set of use cases for the application.

An alternative solution was pioneered by Google in 2016 with sample-based PGO. [Chen et al., 2016] Instead of instrumenting the code, the profiling data can be obtained from the output of a standard profiling tool such as Linux `perf`. Google developed an open-source tool called AutoFDO[188] that converts sampling data generated by Linux

[186] PGO in Clang - https://clang.llvm.org/docs/UsersManual.html#profiling-with-instrumentation
[187] PGO in GCC - https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options
[188] AutoFDO - https://github.com/google/autofdo

`perf` into a format that compilers like GCC and LLVM can understand.

This approach has a few advantages over instrumented PGO. First of all, it eliminates one step from the PGO build workflow, namely step 1 since there is no need to build an instrumented binary. Secondly, profiling data collection runs on an already optimized binary, thus it has a much lower runtime overhead. This makes it possible to collect profiling data in production environment for a longer period of time. Since this approach is based on HW collection, it also enables new kinds of optimizations that are not possible with instrumented PGO. One example is branch-to-cmov conversion, which is a transformation that replaces conditional jumps with conditional moves to avoid the cost of a branch misprediction (see Section 10.3). To effectively perform this transformation, a compiler needs to know how frequently the original branch was mispredicted. This information is available with sample-based PGO on modern CPUs (Intel Skylake+).

The next innovative idea came from Meta in the mid-2018, when it open-sourced its binary optimization tool called BOLT.[189] BOLT works on the already compiled binary. It first disassembles the code, then it uses the profile information collected by sampling profiler, such as Linux perf, to do various layout transformations and then relinks the binary again. [Panchenko et al., 2018] As of today, BOLT has more than 15 optimization passes, including basic blocks reordering, function splitting and reordering, and others. Similar to traditional PGO, primary candidates for BOLT optimizations are programs that suffer from many instruction cache and iTLB misses. Since January 2022, BOLT is a part of the LLVM project and is available as a standalone tool.

A few years after BOLT was introduced, Google open-sourced its binary relinking tool called Propeller. It serves a similar purpose but instead of disassembling the original binary, it relies on linker input, and thus can be distributed across several machines for better scaling and less memory consumption. Post-link optimizers such as BOLT and Propeller can be used in combination with traditional PGO (and LTO) and often provide additional 5-10% performance speedup. Such techniques open up new kinds of binary rewriting optimizations that are based on HW telemetry.

11.8 Reducing ITLB Misses

Another important area of tuning FE efficiency is virtual-to-physical address translation of memory addresses. Primarily those translations are served by TLB (see Section 3.7.1), which caches most recently used memory page translations in dedicated entries. When TLB cannot serve the translation request, a time-consuming page walk of the kernel page table takes place to calculate the correct physical address for each referenced virtual address. Whenever you see a high percentage of ITLB overhead in the TMA summary, the advice in this section may become handy.

In general, relatively small applications are not susceptible to ITLB misses. For example, Golden Cove microarchitecture can cover memory space up to 1MB in its ITLB. If machine code of your application fits in 1MB you should not be affected by ITLB misses. The problem start to appear when frequently executed parts of an application are scattered around the memory. When many functions begin to frequently call each other, they start competing for the entries in the ITLB. One of the examples is the Clang compiler, which at the time of writing, has a code section of ~60MB. ITLB overhead running on a laptop with a mainstream Intel CoffeeLake processor is ~7%, which means that 7% of cycles are wasted handling ITLB misses: doing demanding page walks and populating TLB entries.

Another set of large memory applications that frequently benefit from using huge pages include relational databases (e.g., MySQL, PostgreSQL, Oracle), managed runtimes (e.g. Javascript V8, Java JVM), cloud services (e.g. web search), web tooling (e.g. node.js). Mapping code sections onto the huge pages can reduce the number of ITLB misses by up to 50% [Suresh Srinivas, 2019], which yields speedups of up to 10% for some applications. However, as it is with many other features, huge pages are not for every application. Small programs with an executable file of only a few KB in size would be better off using regular 4KB pages rather than 2MB huge pages; that way, memory is used more efficiently.

The general idea of reducing ITLB pressure is by mapping the portions of the performance-critical code of an application onto 2MB (huge) pages. But usually, the entire code section of an application gets remapped for simplicity or if you don't know which functions are hot. The key requirement for that transformation to happen is to have code section aligned on 2MB boundary. When on Linux, this can be achieved in two different ways: relinking the binary with additional linker option or remapping the code sections at runtime. Both options are showcased on

[189] BOLT - https://code.fb.com/data-infrastructure/accelerate-large-scale-applications-with-bolt/

easyperf.net[190] blog. To the best of our knowledge, it is not possible on Windows, so we will only show how to do it on Linux.

The first option can be achieved by linking the binary with `-Wl,-zcommon-page-size=2097152 -Wl,-zmax-page-size=2097152` options. These options instruct the linker to place the code section at the 2MB boundary in preparation for it to be placed on 2MB pages by the loader at startup. The downside of such placement is that linker will be forced to insert up to 2MB of padded (wasted) bytes, bloating the binary even more. In the example with Clang compiler, it increased the size of the binary from 111 MB to 114 MB. After relinking the binary, we set a special bit in the ELF binary header that determines if the text segment should be backed with huge pages by default. The simplest way to do it is using the `hugeedit` or `hugectl` utilities from libhugetlbfs[191] package. For example:

```
# Permanently set a special bit in the ELF binary header.
$ hugeedit --text /path/to/clang++
# Code section will be loaded using huge pages by default.
$ /path/to/clang++ a.cpp

# Overwrite default behavior at runtime.
$ hugectl --text /path/to/clang++ a.cpp
```

The second option is to remap the code section at runtime. This option does not require the code section to be aligned to 2MB boundary, thus can work without recompiling the application. This is especially useful when you don't have access to the source code. The idea behind this method is to allocate huge pages at the startup of the program and transfer all the code section there. The reference implementation of that approach is implemented in the iodlr[192]. One option would be to call that functionality from your `main` function. Another option, which is simpler, to build the dynamic library and preload it in the command line:

```
$ LD_PRELOAD=/usr/lib64/liblppreload.so clang++ a.cpp
```

While the first method only works with explicit huge pages, the second approach which uses `iodlr` works both with explicit and transparent huge pages. Instructions on how to enable huge pages for Windows and Linux can be found in appendix C.

Besides from employing huge pages, standard techniques for optimizing I-cache performance can be used for improving ITLB performance. Namely, reordering functions so that hot functions are collocated better, reducing the size of hot regions via Link-Time Optimizations (LTO/IPO), using Profile-Guided Optimizations (PGO) and BOLT, and less aggressive inlining.

BOLT provides the `-hugify` option to automatically use huge pages for hot code based on profile data. When this option is used, `llvm-bolt` will inject the code to put hot code on 2MB pages at runtime. The implementation leverages Linux Transparent Huge Pages (THP). The benefit of this approach is that only a small portion of the code is mapped to the huge pages and the number of required huge pages is minimized, and as a consequence, page fragmentation is reduced.

11.9 Measuring Code Footprint

1. Say why we need to measure code footprint
2. Large code footprint in itself doesn't necessary mean there is impact on performance. Say that it should be analyzed in conjunction with TMA. And be used as an additional data point.
3. estimating hot code footprint in non-ambiguous way is not trivial. Similar to mem footprint it quickly becomes very involved. The question like: how many times the code needs to be executed what is hot code
4. Start with high-level analysis of the `.text` segment. Say that it is not very accurate but it is a good starting point.
5. Not many tools can estimate hot code footprint at runtime.
6. Give example of `perf-tools` for 4 benchmarks: `clang`, `stockfish`, `blender`, `cloverleaf`.

[190] "Performance Benefits of Using Huge Pages for Code" - https://easyperf.net/blog/2022/09/01/Utilizing-Huge-Pages-For-Code.
[191] libhugetlbfs - https://github.com/libhugetlbfs/libhugetlbfs/blob/master/HOWTO.
[192] iodlr library - https://github.com/intel/iodlr.

- build clang with relocations, apply BOLT, check the perf difference rerun perf-tools
- collect blender

7. Mention code heatmap from BOLT.

Questions and Exercises

1. Solve `perf-ninja::pgo` lab assignment.
2. Experiment with using Huge Pages for code section. Take a large application (access to source code is a plus but not necessary), with a binary size of more than 100MB. Try to remap its code section onto huge pages using one of the methods described in Section 11.8. Observe any changes in performance, huge page allocation in `/proc/meminfo`, CPU performance counters that measure ITLB loads and misses.
3. Suppose you have a code that has a C++ switch statement in a loop. You instrumented the code and figured out that one particular case in a switch statement is used 70% of the time. The other 40 cases are used <3% of the time each and other 20 cases never happen. What will you do to optimize performance of that switch/loop?
4. Run the application that you're working with on a daily basis. Apply PGO, llvm-bolt or Propeller. Compare "before" and "after" profiles to understand where the speedups are coming from.

Chapter Summary

Summary of CPU Front-End optimizations is presented in Table 8.

Table 8: Summary of CPU Front-End optimizations.

Transform	How transformed?	Why helps?	Works best for	Done by
Basic block placement	maintain fall through hot code	not taken branches are cheaper; better cache utilization	any code, especially with a lot of branches	compiler
Basic block alignment	shift the hot code using NOPS	better cache utilization	hot loops	compiler
Function splitting	split cold blocks of code and place them in separate functions	better cache utilization	functions with complex CFG when there are big blocks of cold code between hot parts	compiler
Function reorder	group hot functions together	better cache utilization	many small hot functions	linker

- Code layout improvements are often underestimated and end up being omitted and forgotten. CPU Front-End performance issues like I-cache and ITLB misses represent a large portion of wasted cycles, especially for applications with large codebases. But even small- and medium-sized applications can benefit from optimizing the machine code layout.
- It is usually not the first thing developers turn their attention to when trying to improve the performance of their application. They prefer to start with low hanging fruits like loop unrolling and vectorization. However, knowing that you might get an extra 5-10% just from better machine code layout is still useful.
- It is usually the best option to use LTO, PGO, BOLT, and other tools if you can come up with a set of typical use cases for your application. For large applications, it is the only practical way to improve machine code layout.

12 Other Tuning Areas

In this chapter, we will take a look at some of the optimization topics not specifically related to any of the categories covered in the previous three chapters, still important enough to find their place in this book.

12.1 Optimizing Input-Output

To be written

- network and storage IO kernel bypass
- use OS-specific APIs, for example, mmap (read a file into the address space) vs. C++ streams.

12.2 Compile-Time Computations

If a portion of a program does some calculations that don't depend on the input, it can be precomputed ahead of time instead of doing it in the runtime. Modern optimizing compilers already move a lot of computations into compile-time, especially trivial cases like `int x = 2 * 10` into `int x = 20`. Although, they cannot handle more complicated calculations at compile time if they involve branches, loops, function calls. C++ language provides features that allow us to make sure that certain calculations happen at compile time.

In C++, it's possible to move computations into compile-time with various metaprogramming techniques. Before `C++11/14`, developers were using templates to achieve this result. It is theoretically possible to express any algorithm with template metaprogramming; however, this method tends to be syntactically obtuse and often compile quite slowly. Still, it was a success that enabled a new class of optimizations. Fortunately, metaprogramming gradually becomes a lot simpler with every new C++ standard. The `C++14` standard allows having `constexpr` functions, and the `C++17` standard provides compile-time branches with the `if constexpr` keyword. This new way of metaprogramming allows doing many computations in compile-time without sacrificing code readability. [Fog, 2004, Chapter 15 Metaprogramming]

An example of optimizing an application by moving computations into compile-time is shown in Listing 59. Suppose a program involves a test for a number being prime. If we know that a large portion of tested numbers is less than `1024`, we can precompute the results ahead of time and keep them in a `constexpr` array `primes`. At runtime, most of the calls of `isPrime` will involve just one load from the `primes` array, which is much cheaper than computing it at runtime.

Architecture-Specific Optimizations

Performance considerations on x86, ARM, and RISC-V

Major differences between ISAs

Know capabilities of your ISA

CISC vs RISC code density

Microarchitecture-specific issues

Memory ordering

Memory alignment

4K aliasing

Cache trashing

Non-temporal stores

Listing 59 Precomputing prime numbers in compile-time

```
constexpr unsigned N = 1024;

// function pre-calculates first N primes in compile-time
constexpr std::array<bool, N> sieve() {
  std::array<bool, N> Nprimes{true};
  Nprimes[0] = Nprimes[1] = false;
  for(long i = 2; i < N; i++)
    Nprimes[i] = true;
  for(long i = 2; i < N; i++) {
    if (Nprimes[i])
      for(long k = i + i; k < N; k += i)
        Nprimes[k] = false;
  }
  return Nprimes;
}

constexpr std::array<bool, N> primes = sieve();

bool isPrime(unsigned value) {
  // primes is accessible both in compile-time and runtime
  static_assert(primes[97], "");
  static_assert(!primes[98], "");
  if (value < N)
    return primes[value];
  // fall back to computing in runtime
}
```

Instruction latencies and throughput

12.3 Low Latency Tuning Techniques

So far we have discussed a variety of software optimizations that aim at improving overall performance of an application. In this section, we will discuss additional tuning techniques used in low-latency systems, such as real-time processing and high-frequency trading (HFT). In such an environment, the primary optimization goal is to make a certain portion of a program to run as fast as possible. When you work in the HFT industry, every microsecond and nanosecond count as it has a direct impact on profits. Usually, the low-latency portion implements a critical loop of a real-time or an HFT system, such as moving a robotic arm or sending an order to the exchange. Optimizing latency of a critical path is sometimes done at the expense of other portions of a program. And some techniques even sacrifice the overall throughput of a system.

When developers optimize for latency, they avoid any unnecessary cost they need to pay on a hot path. That usually involves system calls, memory allocation, I/O, and anything else that has non-deterministic latency. To reach the lowest possible latency, the hot path needs to have all the resources ready and available for it ahead of time.

One relatively simple technique is to precompute some of the operations you would do on the hot path. That comes with a cost of using more memory which will be unavailable to other processes in the system but it may save you some precious cycles on a critical path. However, keep in mind that sometimes it is faster to compute the thing than to fetch the result from memory.

Since this a book about low-level CPU performance, we will skip talking about higher-level techniques similar to the one we just mentioned. Instead, we will discuss how to avoid page faults, cache misses, TLB shootdowns, and core throttling on a critical path.

12.3.1 Avoid Minor Page Faults

While the term contains the word "minor", there's nothing minor about the impact of minor page faults on runtime latency. Recall that when a user code allocates memory, OS only commits to provide a page, but it doesn't immediately execute on the committment by giving us a zeroed physical page. Instead, it will wait until the first time the user code will access it and only then the OS fulfills its duties. The very first access to a newly allocated page triggers a minor page fault, a HW interrupt that is handled by the OS. Latency impact of minor faults can range from just under a microsecond up to several microseconds, especially if you're using a Linux kernel with 5-level page tables instead of 4-level page tables.

How do you detect runtime minor page faults in your application? One simple way is by using the `top` utility (add the `-H` option for a thread-level view). Add the `vMn` field to the default selection of display columns to view the number of minor page faults occurring per display refresh interval. Listing 60 shows a dump of `top` command with the top-10 processes while compiling a large C++ project. The additional `vMn` column shows the number of minor page faults occurred during the last 3 seconds.

Listing 60 A dump of Linux top command with additional vMn field while compiling large C++ project.

```
   PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND  vMn
341763 dendiba+  20   0  303332 165396  83200 R  99.3   1.0   0:05.09 c++      13k
341705 dendiba+  20   0  285768 153872  87808 R  99.0   1.0   0:07.18 c++       5k
341719 dendiba+  20   0  313476 176236  83328 R  94.7   1.1   0:06.49 c++       8k
341709 dendiba+  20   0  301088 162800  82944 R  93.4   1.0   0:06.46 c++       2k
341779 dendiba+  20   0  286468 152376  87424 R  92.4   1.0   0:03.08 c++      26k
341769 dendiba+  20   0  293260 155068  83072 R  91.7   1.0   0:03.90 c++      22k
341749 dendiba+  20   0  360664 214328  75904 R  88.1   1.3   0:05.14 c++      18k
341765 dendiba+  20   0  351036 205268  76288 R  87.1   1.3   0:04.75 c++      18k
341771 dendiba+  20   0  341148 194668  75776 R  86.4   1.2   0:03.43 c++      20k
341776 dendiba+  20   0  286496 147460  82432 R  76.2   0.9   0:02.64 c++      25k
```

Another way of detecting runtime minor page faults involves attaching to the running process with `perf stat -e page-faults`.

In the HFT world, anything more than `0` is a problem. But for low latency applications in other business domains, a constant occurrence in the range of 100-1000 faults per second should prompt further investigation. Investigating the root cause of runtime minor page faults can be as simple as firing up `perf record -e page-faults` and then `perf report` to locate offending source code lines.

To avoid page fault penalties during runtime, you should pre-fault all the memory for the application at startup time. A toy example might look something like this:

```
char *mem = malloc(size);
int pageSize = sysconf(_SC_PAGESIZE)
for (int i = 0; i < size; i += pageSize)
  mem[i] = 0;
```

First, this sample code allocates `size` amount of memory on the heap as usual. However, immediately after that, it steps by and touches each page of newly allocated memory to ensure each one is brought into RAM. This method helps to avoid runtime delays caused by minor page faults during future accesses.

Take a look at Listing 61 with a more comprehensive approach of tuning the glibc allocator in conjunction with `mlock/mlockall` syscalls (taken from the "Real-time Linux Wiki" [193]).

The code in Listing 61 tunes three glibc malloc settings: `M_MMAP_MAX`, `M_TRIM_THRESHOLD`, and `M_ARENA_MAX`.

- Setting `M_MMAP_MAX` to `0` disables underlying `mmap` syscall usage for large allocations – this is necessary because the `mlockall` can be undone by library usage of `munmap` when it attempts to release `mmap`-ed segments back to the OS, defeating the purpose of our efforts.

[193] The Linux Foundation Wiki: Memory for Rael-time Applications - https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/memory

Listing 61 Tuning the glibc allocator to lock pages in RAM and prevent releasing them to the OS.

```
#include <malloc.h>
#include <sys/mman.h>

mallopt(M_MMAP_MAX, 0);
mallopt(M_TRIM_THRESHOLD, -1);
mallopt(M_ARENA_MAX, 1);

mlockall(MCL_CURRENT | MCL_FUTURE);

char *mem = malloc(size);
for (int i = 0; i < size; i += sysconf(_SC_PAGESIZE))
    mem[i] = 0;
//...
free(mem);
```

- Setting `M_TRIM_THRESHOLD` to `-1` prevents glibc from returning memory to the OS after calls to `free`. As indicated before, this option has no effect on `mmap`-ed segments.
- Finally, setting `M_ARENA_MAX` to `1` prevents glibc from allocating multiple arenas via `mmap` to accommodate multiple cores. Keep in mind, the latter hinders the glibc allocator's multithreaded scalability feature.

Combined, these settings force glibc into heap allocations which will not release memory back to the OS until the application ends. As a result, the heap will remain the same size after the final call to `free(mem)` in the code above. Any subsequent runtime calls to `malloc` or `new` simply will reuse space in this pre-allocated/pre-faulted heap area if it is sufficiently sized at initialization.

More importantly, all that heap memory that was pre-faulted in the `for`-loop will persist in RAM due to the previous `mlockall` call – the option `MCL_CURRENT` locks all pages which are currently mapped, while `MCL_FUTURE` locks all pages that will become mapped in the future. An added benefit of using `mlockall` this way is that any thread spawned by this process will have its stack pre-faulted and locked, as well. For the finer control of page locking, developers should use `mlock` system call which gives you the option to choose which pages should persist in RAM. A downside of this technique is that it reduces the amount of memory available to other processes running on the system.

Developers of applications for Windows should look into the following APIs: lock pages with `VirtualLock`, avoid immediate release of memory with `VirtualFree` with `MEM_DECOMMIT`, but not `MEM_RELEASE` flag.

These are just two example methods for preventing runtime minor faults. Some or all of these techniques may be already integrated into memory allocation libraries such as jemalloc, tcmalloc, or mimalloc. Check the documentation of your chosen library to see what is available.

12.3.2 Cache Warming

In some applications, the portions of code that are most latency-sensitive are the least frequently executed. An example of such an application might be an HFT application that continuously reads market data signals from the stock exchange and, once a favorable market signal is detected, sends a BUY order to the exchange. In the aforementioned workload, the code paths involved with reading the market data is most commonly executed, while the code paths for executing a BUY order is rarely executed.

Since other players in the market are likely to catch the same market signal, the success of the strategy largely relies on how fast we can react, in other words, how fast we send the order to the exchange. When we want our BUY order to reach the exchange as fast as possible and to take advantage of the favorable signal detected in the market data, the last thing we want is to meet roadblocks right at the moment we decide to take off.

When a certain code path is not exercised for a while, its instructions and associated data are likely to be evicted from of the I-cache and D-cache. Then, just when we need that critical piece of rarely executed code to run, we take I-cache and D-cache miss penalties, which may cause us loose the race. This is where the technique of *cache warming* would be helpful.

Cache warming involves periodically exercising the latency-sensitive code to keep it in the cache while ensuring it does not follow all the way through with any unwanted actions. Exercising the latency-sensitive code also "warms up" the D-cache by bringing latency-sensitive data into it. This technique is routinely employed for HFT applications. While we will not provide an example implementation, you can get a taste of it in a CppCon 2018 lightning talk[194].

12.3.3 Avoid TLB Shootdowns

We learned from earlier chapters that the TLB is a fast but finite per-core cache for virtual-to-physical memory address translations that reduces the need for time-consuming kernel page table walks. When a process is scheduled off a core to make way for a new process with an entirely different virtual address space, the TLB that belongs to the core needs to flushed. In addition to wholesale TLB flushes, there is a more selective procedure for invalidating TLB entries called *TLB shootdowns*.

Unlike the case with MESI-based protocols and per-core CPU caches (i.e., L1, L2, and LLC), the HW itself is incapable of maintaining core-to-core TLB coherency. Therefore, this task must be performed in software by the kernel. The kernel fulfills this role by means of a specific type of Inter Processor Interrupts (IPI), called TLB shootdowns, which on x86 platforms are implemented via the `INVLPG` assembly instruction.

TLB shootdowns are one of the most overlooked pitfalls to achieving low latency with multithreaded applications. Why? Because in a multithreaded application, process threads share the virtual address space. Therefore, the kernel must communicate specific types of updates to that shared address space among the TLBs of the cores on which any of the participating threads execute. For example, commonly used syscalls such as `munmap` (which can be disabled from glibc allocator usage, see Section 12.3.1), `mprotect`, and `madvise` effect the types of address space changes that the kernel must communicate among the constituent threads of a process.

Though a developer may avoid explicitly using these syscalls in his/her code, TLB shootdowns may still erupt from external sources – e.g., allocator shared libraries or OS facilities. Not only will this type of IPI disrupt runtime application performance, but the magnitude of its impact grows with the number of threads involved since the interrupts are delivered in software.

How do you detect TLB shootdowns in your multithreaded application? One simple way is to check the TLB row in `/proc/interrupts`. A useful method of detecting continuous TLB interrupts during runtime is to use the `watch` command while viewing this file. For example, you might run `watch -n5 -d 'grep TLB /proc/interrupts'`, where the `-n 5` option refreshes the view every 5 seconds while `-d` highlights the delta between each refresh output.

Listing 62 shows a dump of `/proc/interrupts` with a large number of TLB shootdowns on the `CPU2` processor that ran the latency-critical thread. Notice the order of magnitude difference between other cores. In that scenario, the culprit of such a behavior was a Linux kernel feature called Automatic NUMA Balancing, that can be easily disarmed with `sysctl -w numa_balancing=0`.

Listing 62 A dump of /proc/interrupts that shows a large number of TLB shootdowns on CPU2

```
           CPU0       CPU1       CPU2       CPU3
...
NMI:          0          0          0          0   Non-maskable interrupts
LOC:     552219    1010298    2272333    3179890   Local timer interrupts
SPU:          0          0          0          0   Spurious interrupts
...
IWI:          0          0          0          0   IRQ work interrupts
RTR:          7          0          0          0   APIC ICR read retries
RES:      18708       9550        771        528   Rescheduling interrupts
CAL:        711        934       1312       1261   Function call interrupts
TLB:       4493       6108      73789       5014   TLB shootdowns
```

But that's not the only source of TLB shootdowns. Others include Transparent Huge Pages, memory compaction, page migration, and page cache writeback. Garbage collectors also can initiate TLB shootdowns. These features either relocate pages and/or alter permissions on pages in the process of fulfilling its duties, which require page table updates and, thus, TLB shootdowns.

[194] Cache Warming technique - https://www.youtube.com/watch?v=XzRxikGgaHI

Preventing TLB shootdowns requires limiting the number of updates made to the shared process address space. On the source code level, you should avoid runtime execution of the aforementioned list of syscalls, namely `munmap`, `mprotect`, and `madvise`. On the OS level, disable kernel features which induce TLB shootdowns as a consequence of its function, such as Transparent Huge Pages and Automatic NUMA Balancing. For more nuanced discussion on TLB shootdowns, along with their detection and prevention, read the article[195] on the JabPerf blog.

12.3.4 Prevent Unintentional Core Throttling

C/C++ compilers are a wonderful feat of engineering. However, they sometimes generate surprising results that may lead you on a wild goose chase. A real-life example is an instance where the compiler optimizer emits heavy AVX instructions that you never intended. While less of an issue on more modern chips, many older generations of CPUs (which remain in active usage on-prem and in the cloud) exhibit heavy core throttling/downclocking when executing heavy AVX instructions. If your compiler produces these instructions without your explicit knowledge or consent, you may experience unexplained latency anomalies during application runtime.

For this specific case, if heavy AVX instruction usage is not desired, include "-mprefer-vector-width=###" to your compilation flags to pin the highest width instruction set to either 128 or 256. Again, if your entire server fleet runs on the latest chips then this is much less of a concern since the throttling impact of AVX instruction sets is negligible nowadays.

12.4 Slow Floating-Point Arithmetic

Some applications that do extensive computations with floating-point values, are prone to one very subtle issue that can cause performance slowdown. This issue arises when an application hit *subnormal* FP value, which we will discuss in this section. You can also find a term *denormal* FP value, which refers to the same thing. According to the IEEE Standard 754,[196] a subnormal is a non-zero number with exponent smaller than the smallest normal number.[197] Listing 63 shows a very simple instantiation of a subnormal value.

In real-world applications, a subnormal value usually represents a signal so small that it is indistinguishable from zero. In audio, it can mean a signal so quiet that it is out of the human hearing range. In image processing, it can mean any of the RGB color components of a pixel to be very close to zero and so on. Interestingly, subnormal values are present in many production software packages, including weather forecasting, ray tracing, physics simulations and modeling and others.

Listing 63 Instantiating a normal and subnormal FP value

```
unsigned usub = 0x80200000; // -2.93873587706e-39 (subnormal)
unsigned unorm = 0x411a428e; // 9.641248703 (normal)
float sub = *((float*)&usub);
float norm = *((float*)&unorm);
assert(std::fpclassify(sub) == FP_SUBNORMAL);
assert(std::fpclassify(norm) != FP_SUBNORMAL);
```

Without subnormal values, the subtraction of two FP values `a - b` can underflow and produce zero even though the values are not equal. Subnormal values allow calculations to gradually lose precision without rounding the result to zero. Although, it comes with a cost as we shall see later. Subnormal values also may occur in production software when a value keeps decreasing in a loop with subtraction or division.

From the hardware perspective, handling subnormals is more difficult than handling normal FP values as it requires special treatment and generally, is considered as an exceptional situation. The application will not crash, but it will get a performance penalty. Calculations that produce or consume subnormal numbers are much slower than similar calculations on normal numbers and can run 10 times slower or more. For instance, Intel processors currently handle operations on subnormals with a microcode *assist*. When a processor recognizes subnormal FP value, Microcode Sequencer (MSROM) will provide the necessary microoperations (UOPs) to compute the result.

[195] JabPerf blog: TLB Shootdowns - https://www.jabperf.com/how-to-deter-or-disarm-tlb-shootdowns/
[196] IEEE Standard 754 - https://ieeexplore.ieee.org/document/8766229
[197] Subnormal number - https://en.wikipedia.org/wiki/Subnormal_number

In many cases, subnormal values are generated naturally by the algorithm and thus are unavoidable. Luckily, most processors give an option to flush subnormal value to zero and not generate subnormals in the first place. Indeed, many users rather choose to have slightly less accurate results rather than slowing down the code. Although, the opposite argument could be made for finance software: if you flush a subnormal value to zero, you lose precision and cannot scale it up as it will remain zero. This could make some customers angry.

Suppose you are OK without subnormal values, how to detect and disable them? While one can use runtime checks as shown in Listing 63, inserting them all over the codebase is not practical. There is better way to detect if your application is producing subnormal values using PMU (Performance Monitoring Unit). On Intel CPUs, you can collect the `FP_ASSIST.ANY` performance event, which gets incremented every time a subnormal value is used. TMA methodology classifies such bottlenecks under the `Retiring` category, and yes, this is one of the situations when high `Retiring` doesn't mean a good thing.

Once you confirmed subnormal values are there, you can enable the FTZ and DAZ modes:

- **DAZ** (Denormals Are Zero). Any denormal inputs are replaced by zero before use.
- **FTZ** (Flush To Zero). Any outputs that would be denormal are replaced by zero.

When they are enabled, there is no need for a costly handling of subnormal value in a CPU floating-point arithmetic. In x86-based platforms, there are two separate bit fields in the `MXCSR`, global control and status register. In ARM Aarch64, two modes are controled with `FZ` and `AH` bits of the `FPCR` control register. If you compile your application with `-ffast-math`, you have nothing to worry about, the compiler will automatically insert the required code to enable both flags at the start of the program. The `-ffast-math` compiler option is a little overloaded, so GCC developers created a separate `-mdaz-ftz` option that only controls the behavior of subnormal values. If you'd rather control it from the source code, Listing 64 shows example that you can use. If you choose this option, avoid frequent changes to the `MXCSR` register because the operation is relatively expensive. A read of the MXCSR register has a fairly long latency, and a write to the register is a serializing instruction.

Listing 64 Enabling FTZ and DAZ modes manually

```
unsigned FTZ = 0x8000;
unsigned DAZ = 0x0040;
unsigned MXCSR = _mm_getcsr();
_mm_setcsr(MXCSR | FTZ | DAZ);
```

Keep in mind, both `FTZ` and `DAZ` modes are incompatible with the IEEE Standard 754. They are implemented in hardware to improve performance for applications where underflow is common and generating a denormalized result is unnecessary. Usually, we have observed a 3%-5% speedup on some production floating-point applications that were using subnormal values and sometimes even up to 50%.

12.5 System Tuning

After successfully completing all the hard work of tuning an application to exploit all the intricate facilities of the CPU microarchitecture, the last thing we want is for the system firmware, the OS, or the kernel to destroy all our efforts. The most highly tuned application will mean very little if it is intermittently disrupted by a System Management Interrupt (SMI), a BIOS interrupt that halts the entire OS to execute firmware code. Such interrupt might run for up to 10s to 100s of milliseconds at a time.

Fair to say, developers usually have little to no control over the environment in which the application is executed. When we ship the product, it's unrealistic to tune every setup a customer might have. Usually, large-enough organizations have separate Operations (Ops) Teams, which handles such sort of issues. Nevertheless, when communicating with members of such teams, it's important to understand what else can limit the application to show its best performance.

As shown in Section 2.1, there are many things to tune in the modern system, and avoiding system-based interference is not an easy task. An example of a performance tuning manual of x86-based server deployments is Red Hat guidelines[198]. There, you will find tips for eliminating or significantly minimizing cache disrupting interrupts from

[198] Red Hat low latency tuning guidelines - https://access.redhat.com/sites/default/files/attachments/201501-perf-brief-low-latency-tuning-rhel7-v2.1.pdf

sources like the system BIOS, the Linux kernel, and from device drivers, among many other sources of application interference. These guidelines should serve as a baseline image for all new server builds before any application is deployed into a production environment.

When it comes to tuning a specific system setting, it is not always an easy 'yes' or 'no' answer. For example, it's not clear upfront whether your application will benefit from the Simultaneous Multi-Threading (SMT) feature enabled in the environment in which your SW is running. The general guideline is to enable SMT only for heterogenous workloads[199] that exhibit a relatively low IPC. On the other hand, CPU manufacturers these days offer processors with such high core counts that SMT is far less necessary than it was in the past. However, this is just a general guideline, and as with everything stressed so far in this book, it is best to measure for yourself.

Most out-of-the-box platforms are configured for optimal throughput while saving power when it's possible. But there are industries with real-time requirements, which care more about having lower latency than everything else. An example of such an industry can be robots operating in automotive assembly lines. Actions performed by such robots are triggered by external events and usually have a predetermined time budget to finish because the next interrupt will come shortly (it is usually called "control loop"). Meeting real-time goals for such a platform may require sacrificing the overall throughput of the machine or allowing it to consume more energy. One of the popular techniques in that area is to disable processor sleeping states[200] to keep it ready to react immediately. Another interesting approach is called Cache Locking,[201] where portions of the CPU cache are reserved for a particular set of data; it helps to streamline the memory latencies within an application.

Questions and Exercises

1. Solve `perf-ninja::lto` and `perf-ninja::io_opt1` lab assignments.
2. Run the application that you're working with on a daily basis. Find the hotspot. Check if it can benefit from any of the techniques we discussed in this chapter.

Chapter Summary

[199] I.e., when sibling threads execute differing instruction patterns

[200] Power Management States: P-States, C-States. See details here: https://software.intel.com/content/www/us/en/develop/articles/power-management-states-p-states-c-states-and-package-c-states.html.

[201] Cache Locking. Survey of cache locking techniques [Mittal, 2016]. An example of pseudo-locking a portion of the cache, which is then exposed as a character device in the Linux file system and made available for `mmap`ing: https://events19.linuxfoundation.org/wp-content/uploads/2017/11/Introducing-Cache-Pseudo-Locking-to-Reduce-Memory-Access-Latency-Reinette-Chatre-Intel.pdf.

13 Optimizing Multithreaded Applications

Modern CPUs are getting more and more cores each year. As of 2020, you can buy an x86 server processor which will have more than 50 cores! And even a mid-range desktop with 8 execution threads is a pretty usual setup. Since there is so much processing power in every CPU, the challenge is how to utilize all the HW threads efficiently. Preparing software to scale well with a growing amount of CPU cores is very important for the future success of the application.

Multithreaded applications have their own specifics. Certain assumptions of single-threaded execution get invalidated when we're dealing with multiple threads. For example, we can no longer identify hotspots by looking at a single thread since each thread might have its own hotspot. In a popular producer-consumer[202] design, the producer thread may sleep during most of the time. Profiling such a thread won't shed light on the reason why our multithreaded application is not scaling well.

[TODO:] redo the scaling study

13.1 Performance Scaling and Overhead

When dealing with a single-threaded application, optimizing one portion of the program usually yields positive results on performance. However, it's not necessarily the case for multithreaded applications. There could be an application in which thread `A` executes a long-running operation, while thread `B` finishes its task early and just waits for thread `A` to finish. No matter how much we improve thread `B`, application latency will not be reduced since it will be limited by a longer-running thread `A`.

This effect is widely known as Amdahl's law,[203] which constitutes that the speedup of a parallel program is limited by its serial part. Figure 72 illustrates the theoretical speedup limit as a function of the number of processors. For a program, 75% of which is parallel, the speedup factor converges to 4.

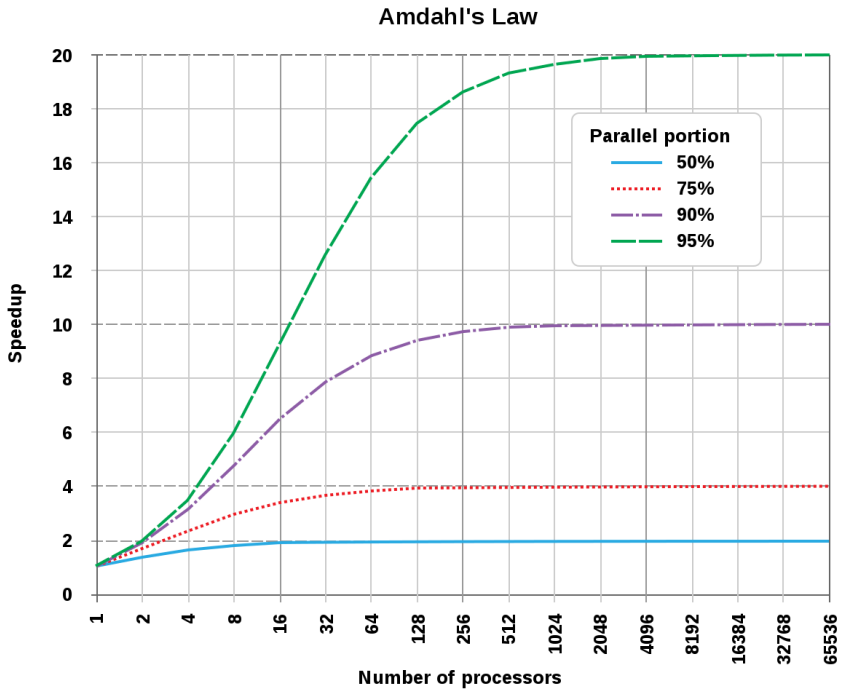
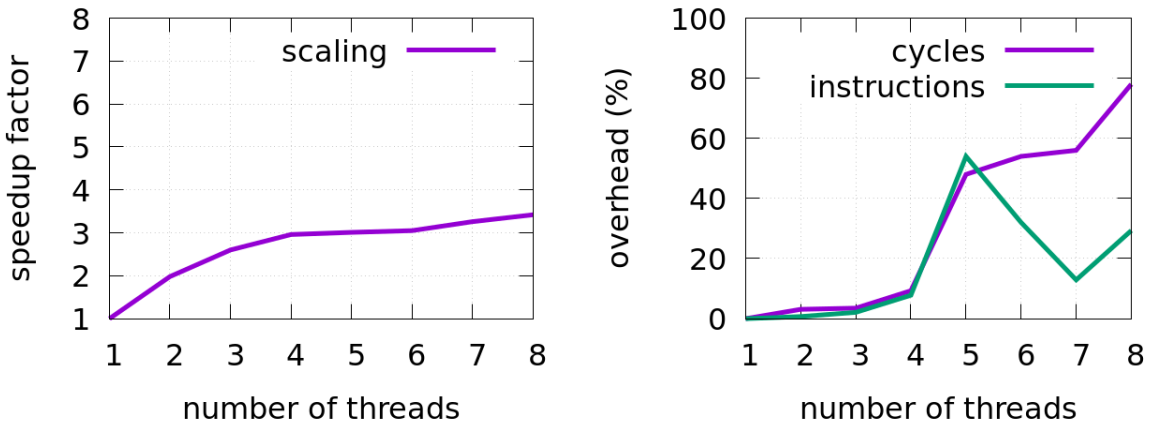


Figure 72: The theoretical speedup of the latency of the execution of a program as a function of the number of processors executing it, according to Amdahl's law. *© Image by Daniels220 via Wikipedia.*

[202] Producer-consumer pattern - https://en.wikipedia.org/wiki/Producer-consumer_problem
[203] Amdahl's law - https://en.wikipedia.org/wiki/Amdahl's_law.

Figure 73a shows performance scaling of the `h264dec` benchmark from Starbench parallel benchmark suite. I tested it on Intel Core i5-8259U, which has 4 cores/8 threads. Notice that after using 4 threads, performance doesn't scale much. Likely, getting a CPU with more cores won't improve performance. [204]



(a) Performance scaling with different number of threads.

(b) Overhead of using different number of threads.

Figure 73: Performance scaling and overhead of h264dec benchmark on Intel Core i5-8259U.

In reality, further adding computing nodes to the system may yield retrograde speed up. This effect is explained by Neil Gunther as Universal Scalability Law[205] (USL), which is an extension of Amdahl's law. USL describes communication between computing nodes (threads) as yet another gating factor against performance. As the system is scaled up, overheads start to hinder the gains. Beyond a critical point, the capability of the system starts to decrease (see Figure 74). USL is widely used for modeling the capacity and scalability of the systems.

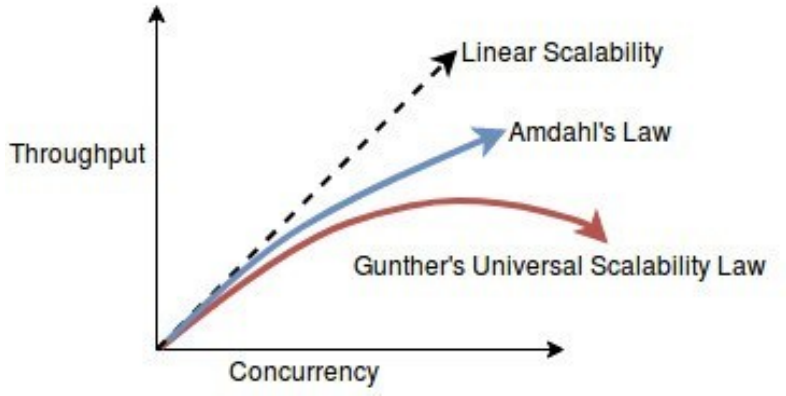


Figure 74: Universal Scalability Law and Amdahl's law. © *Image by Neha Bhardwaj via Knoldus Blogs.*

Slowdowns described by USL are driven by several factors. First, as the number of computing nodes increases, they start to compete for resources (contention). This results in additional time being spent on synchronizing those accesses. Another issue occurs with resources that are shared between many workers. We need to maintain a consistent state of the shared resource between many workers (coherence). For example, when multiple workers frequently change a globally visible object, those changes need to be broadcasted to all nodes that use that object. Suddenly, usual operations start getting more time to finish due to the additional need to maintain coherence. The communication overhead of the h264dec benchmark on Intel Core i5-8259U can be observed in Figure 73b. Notice that the graph indicates increased overhead in terms of elapsed core cycles as we assign more than 4 threads to the task.[206]

[204] However, it will benefit from a CPU with a higher frequency.
[205] USL law - http://www.perfdynamics.com/Manifesto/USLscalability.html#tth_sEc1.
[206] There is an interesting spike in the number of retired instruction when using 5 and 6 worker threads. This should be investigated by profiling the workload.

Optimizing multithreaded applications not only involves all the techniques described in this book so far but also involves detecting and mitigating the aforementioned effects of contention and coherence. The following subsections will describe techniques for addressing these additional challenges for tuning multithreaded programs.

13.2 Parallel Efficiency Metrics

When dealing with multithreaded applications, engineers should be careful analyzing basic metrics like CPU utilization and IPC (see Chapter 4). One of the threads might show high CPU utilization and high IPC, but it could turn out that the thread was just spinning on a lock. That's why, when evaluating the parallel efficiency of an application, it's recommended to use Effective CPU Utilization, which is based only on the Effective time.[207]

13.2.1 Effective CPU Utilization

This metric represents how efficiently the application utilized the CPUs available. It shows the percent of average CPU utilization by all logical CPUs on the system. CPU utilization metric is based only on the Effective time and does not include the overhead introduced by the parallel runtime system[208] and Spin time. A CPU utilization of 100% means that your application keeps all the logical CPU cores busy for the entire time that it runs[Intel, 2023a].

For a specified time interval T, Effective CPU Utilization can be calculated as

$$\text{Effective CPU Utilization} = \frac{\sum_{i=1}^{\text{ThreadsCount}} \text{Effective Cpu Time(T,i)}}{\text{T} \times \text{ThreadsCount}}$$

13.2.2 Thread Count

Applications usually have a configurable number of threads, which allows them to run efficiently on platforms with a different number of cores. Obviously, running an application using a lower number of threads than is available on the system underutilizes its resources. On the other hand, running an excessive number of threads can cause a higher CPU time because some of the threads may be waiting on others to complete, or time may be wasted on context switches.

Besides actual worker threads, multithreaded applications usually have helper threads: main thread, input and output threads, etc. If those threads consume significant time, they require dedicated HW threads themselves. This is why it is important to know the total thread count and configure the number of worker threads properly.

To avoid a penalty for threads creation and destruction, engineers usually allocate a pool of threads[209] with multiple threads waiting for tasks to be allocated for concurrent execution by the supervising program. This is especially beneficial for executing short-lived tasks.

13.2.3 Wait Time

Wait Time occurs when software threads are waiting due to APIs that block or cause synchronization. Wait Time is per-thread; therefore, the total Wait Time can exceed the application Elapsed Time[Intel, 2023a].

A thread can be switched off from execution by the OS scheduler due to either synchronization or preemption. So, Wait Time can be further divided into Sync Wait Time and Preemption Wait Time. A large amount of Sync Wait Time likely indicates that the application has highly contended synchronization objects. We will explore how to find them in the following sections. Significant Preemption Wait Time can signal a thread oversubscription[210] problem either because of a big number of application threads or a conflict with OS threads or other applications on the system. In this case, the developer should consider reducing the total number of threads or increasing task granularity for every worker thread.

13.2.4 Spin Time

Spin time is Wait Time, during which the CPU is busy. This often occurs when a synchronization API causes the CPU to poll while the software thread is waiting [Intel, 2023a]. In reality, the implementation of kernel synchronization

[207] Performance analysis tools such as Intel VTune Profiler can distinguish profiling samples that were taken while the thread was spinning. They do that with the help of call stacks for every sample (see Section 5.4.3).
[208] Threading libraries and APIs like `pthread`, `OpenMP`, and `Intel TBB` have their own overhead for creating and managing threads.
[209] Thread pool - https://en.wikipedia.org/wiki/Thread_pool.
[210] Thread oversubscription - https://software.intel.com/en-us/vtune-help-thread-oversubscription.

primitives prefer to spin on a lock for some time to the alternative of doing an immediate thread context switch (which is expensive). Too much Spin Time, however, can reflect the lost opportunity for productive work.

13.3 Analysis with Intel VTune Profiler

Intel VTune Profiler has a dedicated type of analysis for multithreaded applications called Threading Analysis. Its summary window (see Figure 75) displays statistics on the overall application execution, identifying all the metrics we described in Section 13.2. From the Effective CPU Utilization histogram, we can learn several interesting facts about the captured application behavior. First, on average, only 5 HW threads (logical cores on the diagram) were utilized at the same time. Second, it was very rare for all 8 HW threads to be active at the same time.

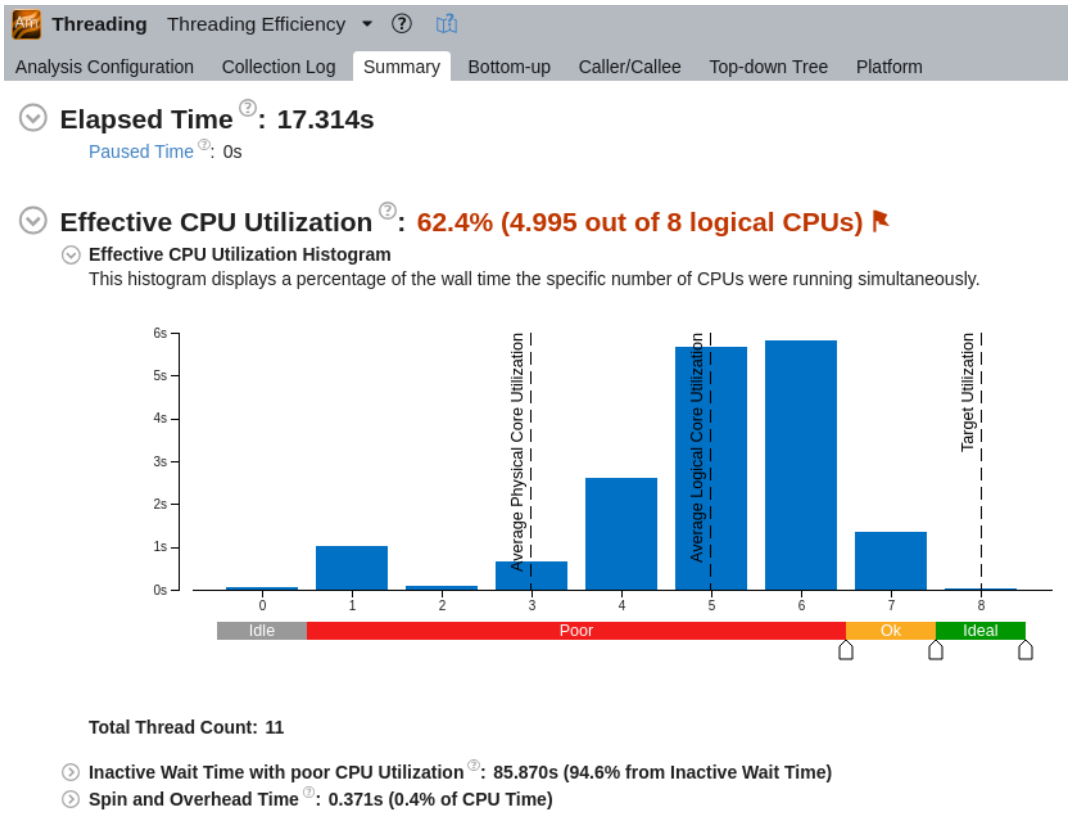


Figure 75: Intel VTune Profiler Threading Analysis summary for x264 benchmark from Phoronix test suite.

13.3.1 Find Expensive Locks

Next, the workflow suggests that we identify the most contended synchronization objects. Figure 76 shows the list of such objects. We can see that `__pthread_cond_wait` definitely stands out, but since we might have dozens of conditional variables in the program, we need to know which one is the reason for poor CPU utilization.

To find out, we can simply click on `__pthread_cond_wait`, which will get us to the Bottom-Up view that is shown on Figure 77. We can see the most frequent path (47% of wait time) that leads to threads waiting on conditional variable: `__pthread_cond_wait <- x264_8_frame_cond_wait <- mb_analyse_init`.

We can next jump into the source code of `x264_8_frame_cond_wait` by double-clicking on the corresponding row in the analysis (see Figure 78). Next, we can study the reason behind the lock and possible ways to make thread communication in this place more efficient. [211]

[211] I don't claim that it will be necessary an easy road, and there is no guarantee that you will find a way to make it better.

Inactive Wait Time with poor CPU Utilization: 85.870s (94.6% from Inactive Wait Time)

Inactive Sync Wait Time: 85.508s

Preemption Wait Time: 0.361s

Top functions by Inactive Wait Time with Poor CPU Utilization.

This section lists the functions sorted by the time spent waiting on synchronization or thread preemption with poor CPU Utilization.

Function	Module	Inactive Wait Time	Inactive Sync Wait Time	Inactive Sync Wait Count
__pthread_cond_wait	libpthread-2.27.so	84.596s	84.594s	24,903
__GI___pthread_mutex_lock	libpthread-2.27.so	0.889s	0.889s	79
[vmlinux]	vmlinux	0.374s	0.016s	453
[vtsspp]	vtsspp	0.005s	0.005s	126
__GI___pthread_mutex_unlock	libpthread-2.27.so	0.002s	0s	0
[Others]		0.004s	0.003s	58

Figure 76: Intel VTune Profiler Threading Analysis showing the most contended synchronization objects for x264 benchmark.

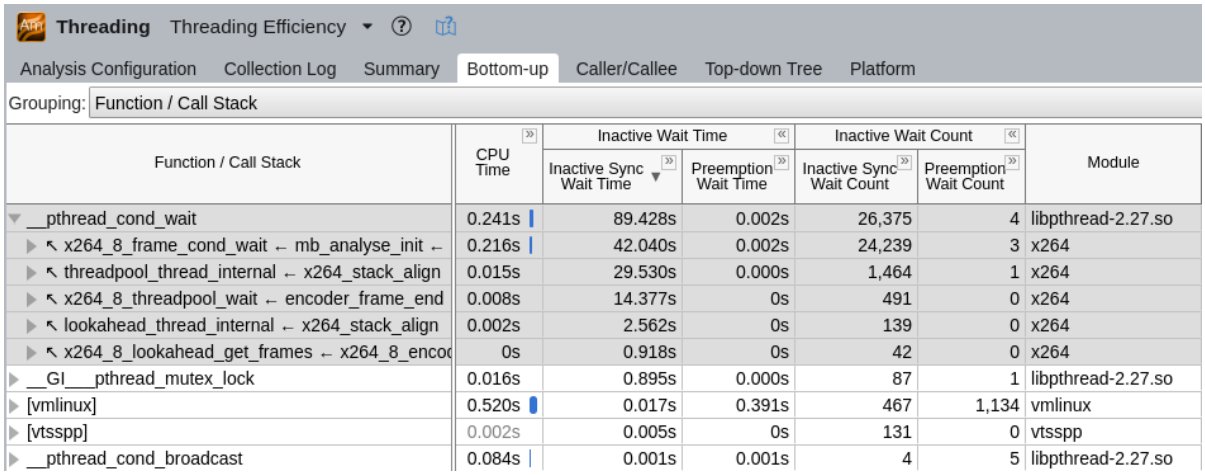


Figure 77: Intel VTune Profiler Threading Analysis showing the call stack for the most contended conditional variable in x264 benchmark.

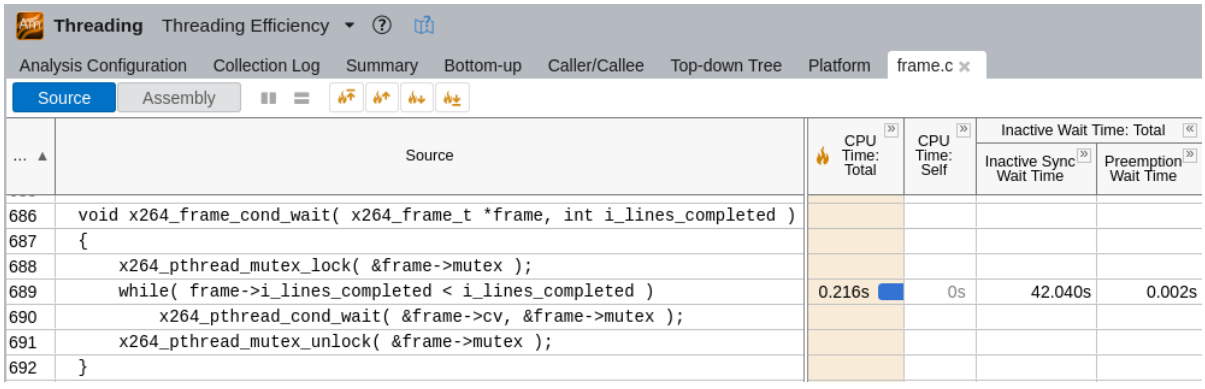


Figure 78: Source code view for x264_8_frame_cond_wait function in x264 benchmark.

13.3.2 Platform View

Another very useful feature of Intel VTune Profiler is Platform View (see Figure 79), which allows us to observe what each thread was doing in any given moment of program execution. This is very helpful for understanding the behavior of the application and finding potential performance headrooms. For example, we can see that during the time interval from 1s to 3s, only two threads were consistently utilizing ~100% of the corresponding CPU core (threads with TID 7675 and 7678). CPU utilization of other threads was bursty during that time interval.

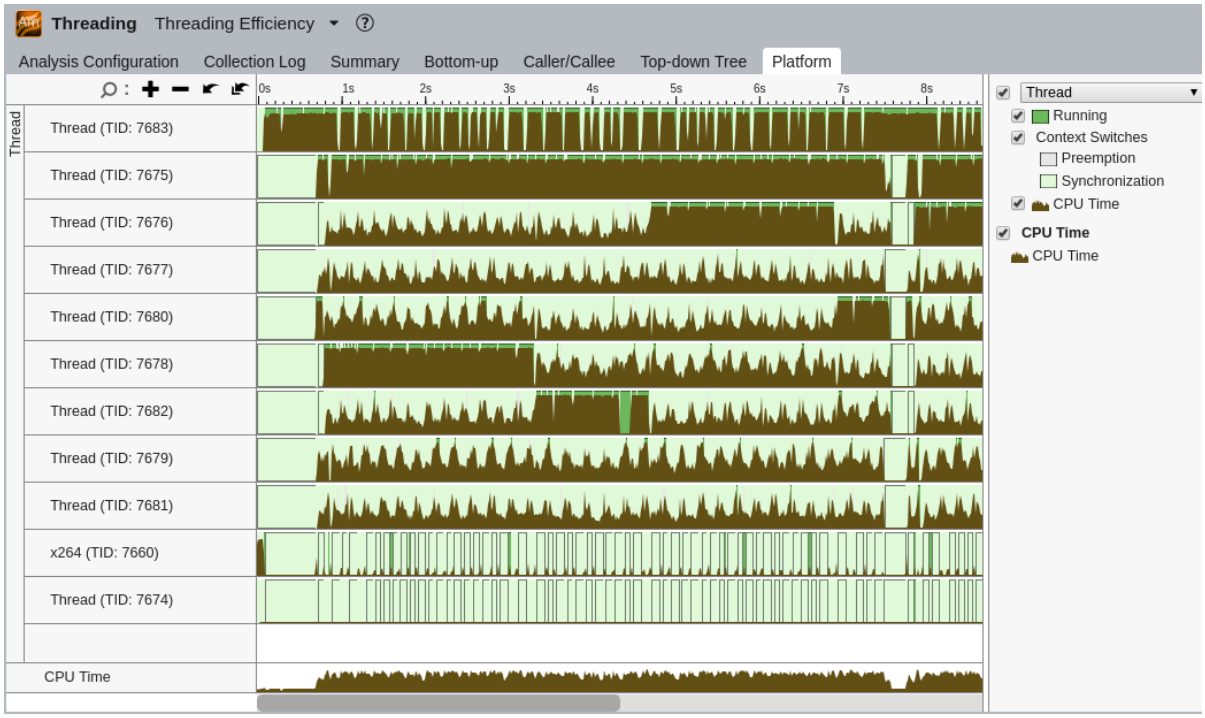


Figure 79: Vtune Platform view for x264 benchmark.

Platform View also has zooming and filtering capabilities. This allows us to understand what each thread was executing during a specified time frame. To see this, select the range on the timeline, right-click and choose Zoom In and Filter In by Selection. Intel VTune Profiler will display functions or sync objects used during this time range.

13.4 Analysis with Linux Perf

Linux `perf` tool profiles all the threads that the application might spawn. It has the `-s` option, which records per-thread event counts. Using this option, at the end of the report, `perf` lists all the thread IDs along with the number of samples collected for each of them:

```
$ perf record -s ./x264 -o /dev/null --slow --threads 8
    Bosphorus_1920x1080_120fps_420_8bit_YUV.y4m
$ perf report -n -T
...
#  PID    TID    cycles:ppp
  6966   6976  41570283106
  6966   6971  25991235047
  6966   6969  20251062678
  6966   6975  17598710694
  6966   6970  27688808973
  6966   6972  23739208014
  6966   6973  20901059568
  6966   6968  18508542853
  6966   6967     48399587
  6966   6966   2464885318
```

To filter samples for a particular software thread, one can use the `--tid` option:

```
$ perf report -T --tid 6976 -n
# Overhead  Samples  Shared Object  Symbol
# ........  .......  .............  ..................................
     7.17%   19877       x264       get_ref_avx2
     7.06%   19078       x264       x264_8_me_search_ref
     6.34%   18367       x264       refine_subpel
     5.34%   15690       x264       x264_8_pixel_satd_8x8_internal_avx2
     4.55%   11703       x264       x264_8_pixel_avg2_w16_sse2
     3.83%   11646       x264       x264_8_pixel_avg2_w8_mmx2
```

Linux `perf` also automatically provides some of the metrics we discussed in Section 13.2:

```
$ perf stat ./x264 -o /dev/null --slow --threads 8 Bosphorus_1920x1080_120fps_420_8bit_YUV.y4m
         86,720.71 msec task-clock        #    5.701 CPUs utilized
            28,386      context-switches  #    0.327 K/sec
             7,375      cpu-migrations    #    0.085 K/sec
            38,174      page-faults       #    0.440 K/sec
   299,884,445,581      cycles            #    3.458 GHz
   436,045,473,289      instructions      #    1.45  insn per cycle
    32,281,697,229      branches          #  372.249 M/sec
       971,433,345      branch-misses     #    3.01% of all branches
```

13.4.1 Find Expensive Locks

To find the most contended synchronization objects with Linux `perf`, one needs to sample on scheduler context switches (`sched:sched_switch`), which is a kernel event and thus requires root access:

```
$ sudo perf record -s -e sched:sched_switch -g --call-graph dwarf -- ./x264 -o /dev/null
    --slow --threads 8 Bosphorus_1920x1080_120fps_420_8bit_YUV.y4m
$ sudo perf report -n --stdio -T --sort=overhead,prev_comm,prev_pid --no-call-graph -F
    overhead,sample
# Samples: 27K of event 'sched:sched_switch'
# Event count (approx.): 27327
# Overhead       Samples         prev_comm    prev_pid
# ........  ............  ................  .........
    15.43%          4217              x264        2973
    14.71%          4019              x264        2972
    13.35%          3647              x264        2976
    11.37%          3107              x264        2975
    10.67%          2916              x264        2970
    10.41%          2844              x264        2971
     9.69%          2649              x264        2974
     6.87%          1876              x264        2969
     4.10%          1120              x264        2967
     2.66%           727              x264        2968
     0.75%           205              x264        2977
```

The output above shows which threads were switched out from the execution most frequently. Notice, we also collect call stacks (`--call-graph dwarf`, see Section 5.4.3) because we need it for analyzing paths that lead to the expensive synchronization events:

```
$ sudo perf report -n --stdio -T --sort=overhead,symbol -F overhead,sample -G
# Overhead       Samples  Symbol
# ........  ............  ........................................
   100.00%         27327  [k] __sched_text_start
   |
   |--95.25%--0xffffffffffffffff
```

```
|  |
|  |--86.23%--x264_8_macroblock_analyse
|  |  |
|  |   --84.50%--mb_analyse_init (inlined)
|  |     |
|  |      --84.39%--x264_8_frame_cond_wait
|  |        |
|  |         --84.11%--__pthread_cond_wait (inlined)
|  |                   __pthread_cond_wait_common (inlined)
|  |                   |
|  |                    --83.88%--futex_wait_cancelable (inlined)
|  |                              entry_SYSCALL_64
|  |                              do_syscall_64
|  |                              __x64_sys_futex
|  |                              do_futex
|  |                              futex_wait
|  |                              futex_wait_queue_me
|  |                              schedule
|  |                              __sched_text_start
...
```

The listing above shows the most frequent path that leads to waiting on a conditional variable (`__pthread_cond_wait`) and later context switch. This path is `x264_8_macroblock_analyse -> mb_analyse_init -> x264_8_frame_cond_wait`. From this output, we can learn that 84% of all context switches were caused by threads waiting on a conditional variable inside `x264_8_frame_cond_wait`.

13.5 Analysis with Coz

In Section 13.1, we defined the challenge of identifying parts of code that affects the overall performance of a multithreaded program. Due to various reasons, optimizing one part of a multithreaded program might not always give visible results. Coz[212] is a new kind of profiler that addresses this problem and fills the gaps left behind by traditional software profilers. It uses a novel technique called "causal profiling", whereby experiments are conducted during the runtime of an application by virtually speeding up segments of code to predict the overall effect of certain optimizations. It accomplishes these "virtual speedups" by inserting pauses that slow down all other concurrently running code. [Curtsinger & Berger, 2018]

Example of applying Coz profiler to C-Ray benchmark from Phoronix test suite is shown on 80. According to the chart, if we improve the performance of line 540 in c-ray-mt.c by 20%, Coz expects a corresponding increase in application performance of C-Ray benchmark overall of about 17%. Once we reach ~45% improvement on that line, the impact on the application begins to level off by COZ's estimation. For more details on this example, see the article[213] on easyperf blog.

13.6 Analysis with eBPF and GAPP

Linux supports a variety of thread synchronization primitives – mutexes, semaphores, condition variables, etc. The kernel supports these thread primitives via the `futex` system call. Therefore, by tracing the execution of the `futex` system call in the kernel while gathering useful metadata from the threads involved, contention bottlenecks can be more readily identified. Linux provides kernel tracing/profiling tools that make this possible, none more powerful than Extended Berkley Packet Filter[214] (eBPF).

eBPF is based around a sandboxed virtual machine running in the kernel that allows the execution of user-defined programs safely and efficiently inside the kernel. The user-defined programs can be written in C and compiled into BPF bytecode by the BCC compiler[215] in preparation for loading into the kernel VM. These BPF programs can

[212] COZ source code - https://github.com/plasma-umass/coz.
[213] Blog article "COZ vs Sampling Profilers" - https://easyperf.net/blog/2020/02/26/coz-vs-sampling-profilers.
[214] eBPF docs - https://prototype-kernel.readthedocs.io/en/latest/bpf/
[215] BCC compiler - https://github.com/iovisor/bcc

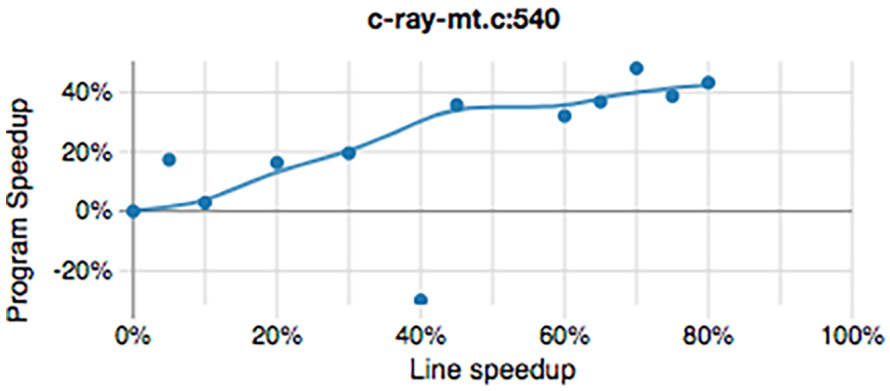


Figure 80: Coz profile for C-Ray benchmark.

be written to launch upon the execution of certain kernel events and communicate raw or processed data back to userspace via a variety of means.

The open0source community has provided many eBPF programs for general use. One such tool is the Generic Automatic Parallel Profiler (GAPP), which helps to track multithreaded contention issues. GAPP uses eBPF to track contention overhead of a multithreaded application by ranking the criticality of identified serialization bottlenecks, collects stack traces of threads that were blocked and the one that caused the blocking. The best thing about GAPP is that it does not require code changes, expensive instrumentation, or recompilation. Creators of the GAPP profiler were able to confirm known bottlenecks and also expose new, previously unreported bottlenecks on Parsec 3.0 Benchmark Suite[216]and some large open-source projects. [Nair & Field, 2020]

13.7 Cache Coherence Issues

13.7.1 Cache Coherency Protocols

Multiprocessor systems incorporate Cache Coherency Protocols to ensure data consistency during shared usage of memory by each individual core containing its own, separate cache entity. Without such a protocol, if both CPU `A` and CPU `B` read memory location `L` into their individual caches, and processor B subsequently modified its cached value for `L`, then the CPUs would have inconsistent values of the same memory location `L`. Cache Coherency Protocols ensure that any updates to cached entries are dutifully updated in any other cached entry of the same location.

One of the most well-known cache coherency protocols is MESI (**M**odified **E**xclusive **S**hared **I**nvalid), which is used to support writeback caches like those used in modern CPUs. Its acronym denotes the four states with which a cache line can be marked (see Figure 81):

- **Modified**: cache line is present only in the current cache and has been modified from its value in RAM
- **Exclusive**: cache line is present only in the current cache and matches its value in RAM
- **Shared**: cache line is present here and in other cache lines and matches its value in RAM
- **Invalid**: cache line is unused (i.e., does not contain any RAM location)

When fetched from memory, each cache line has one of the states encoded into its tag. Then the cache line state keeps transiting from one state to another.[217] In reality, CPU vendors usually implement slightly improved variants of MESI. For example, Intel uses MESIF,[218] which adds a Forwarding (F) state, while AMD employs MOESI,[219] which adds the Owning (O) state. But these protocols still maintain the essence of the base MESI protocol.

As an earlier example demonstrates, the cache coherency problem can cause sequentially inconsistent programs. This problem can be mitigated by having snoopy caches to watch all memory transactions and cooperate with each

[216] Parsec 3.0 Benchmark Suite - https://parsec.cs.princeton.edu/index.htm
[217] Readers can watch and test animated MESI protocol here: https://www.scss.tcd.ie/Jeremy.Jones/vivio/caches/MESI.htm.
[218] MESIF - https://en.wikipedia.org/wiki/MESIF_protocol
[219] MOESI - https://en.wikipedia.org/wiki/MOESI_protocol

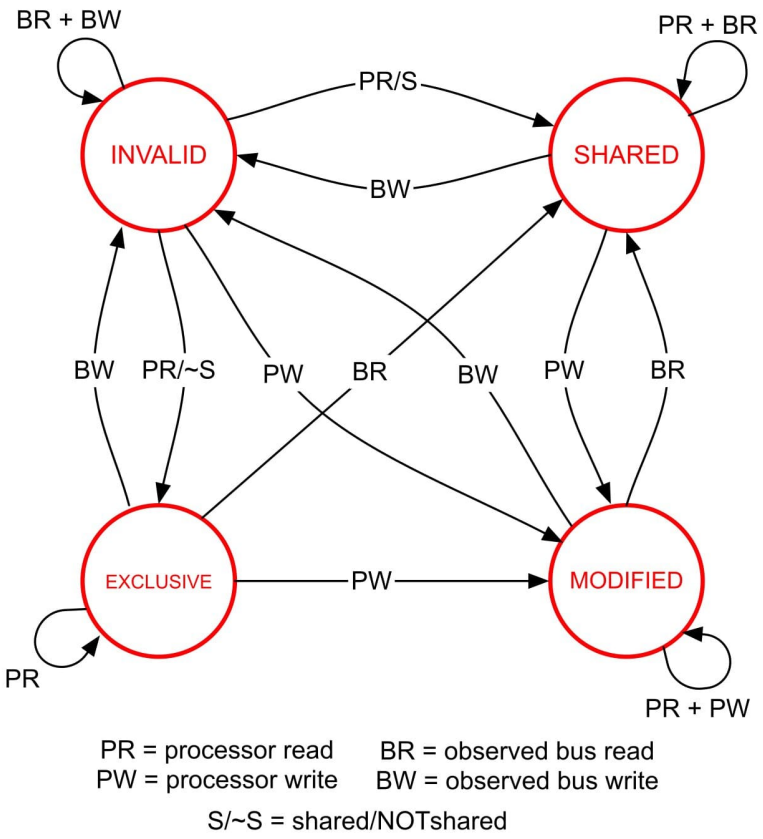


Figure 81: MESI States Diagram. © *Image by University of Washington via courses.cs.washington.edu.*

other to maintain memory consistency. Unfortunately, it comes with a cost since modification done by one processor invalidates the corresponding cache line in another processor's cache. This causes memory stalls and wastes system bandwidth. In contrast to serialization and locking issues, which can only put a ceiling on the performance of the application, coherency issues can cause retrograde effects as attributed by USL in Section 13.1. Two widely known types of coherency problems are "True Sharing" and "False Sharing", which we will explore next.

13.7.2 True Sharing

True sharing occurs when two different processors access the same variable (see Listing 65).

Listing 65 True Sharing Example.

```
unsigned int sum;
{ // parallel section
  for (int i = 0; i < N; i++)
    sum += a[i]; // sum is shared between all threads
}
```

First of all, true sharing implies data races that can be tricky to detect. Fortunately, there are tools that can help identify such issues. Thread sanitizer[220] from Clang and helgrind[221] are among such tools. To prevent data race in Listing 65 one should declare `sum` variable as `std::atomic<unsigned int> sum`.

Using C++ atomics can help to solve data races when true sharing happens. However, it effectively serializes accesses to the atomic variable, which may hurt performance. Another way of solving true sharing issues is by using Thread Local Storage (TLS). TLS is the method by which each thread in a given multithreaded process can allocate memory to store thread-specific data. By doing so, threads modify their local copies instead of contending for a globally available memory location. The example in Listing 65 can be fixed by declaring `sum` with a TLS class specifier: `thread_local unsigned int sum` (since C++11). The main thread should then incorporate results from all the local copies of each worker thread.

13.7.3 False Sharing

False Sharing[222] occurs when two different processors modify different variables that happen to reside on the same cache line (see Listing 66). Figure 82 illustrates the false sharing problem.

Listing 66 False Sharing Example.

```
struct S {
  int sumA; // sumA and sumB are likely to
  int sumB; // reside in the same cache line
};
S s;

{ // section executed by thread A
  for (int i = 0; i < N; i++)
    s.sumA += a[i];
}

{ // section executed by thread B
  for (int i = 0; i < N; i++)
    s.sumB += b[i];
}
```

[220] Clang's thread sanitizer tool: https://clang.llvm.org/docs/ThreadSanitizer.html.
[221] Helgrind, a thread error detector tool: https://www.valgrind.org/docs/manual/hg-manual.html.
[222] It's worth saying that false sharing is not something that can be observed only in low-level languages, like C/C++/Ada, but also in higher-level ones, like Java/C#.

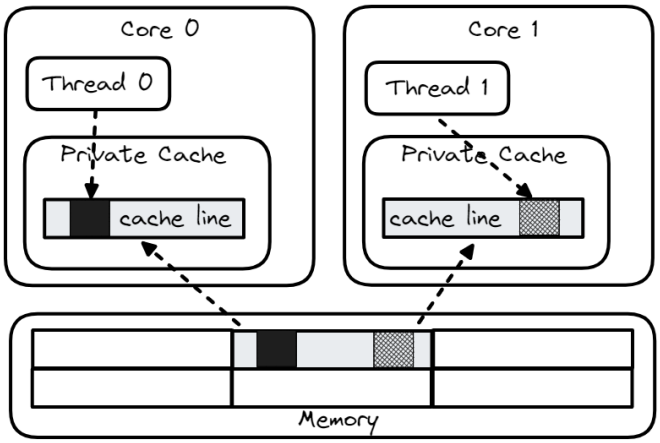


Figure 82: False Sharing: two threads access the same cache line. © *Image by Intel Developer Zone via software.intel.com.*

False sharing is a frequent source of performance issues for multithreaded applications. Because of that, modern analysis tools have built-in support for detecting such cases. TMA characterizes applications that experience true/false sharing as Memory Bound. Typically, in such cases, you would see a high value for Contested Accesses[223] metric.

When using Intel VTune Profiler, the user needs two types of analysis to find and eliminate false sharing issues. Firstly, run Microarchitecture Exploration[224] analysis that implements TMA methodology to detect the presence of false sharing in the application. As noted before, the high value for the Contested Accesses metric prompts us to dig deeper and run the Memory Access analysis with the "Analyze dynamic memory objects" option enabled. This analysis helps in finding out accesses to the data structure that caused contention issues. Typically, such memory accesses have high latency, which will be revealed by the analysis. See an example of using Intel VTune Profiler for fixing false sharing issues on Intel Developer Zone[225].

Linux `perf` has support for finding false sharing as well. As with Intel VTune Profiler, run TMA first (see Section 6.1.1) to find out that the program experience false/true sharing issues. If that's the case, use the `perf c2c` tool to detect memory accesses with high cache coherency cost. `perf c2c` matches store/load addresses for different threads and sees if the hit in a modified cache line occurred. Readers can find a detailed explanation of the process and how to use the tool in a dedicated blog post[226].

It is possible to eliminate false sharing with the help of aligning/padding memory objects. Example in Section 13.7.2 can be fixed by ensuring `sumA` and `sumB` do not share the same cache line (see details in Section 8.1.4).

From a general performance perspective, the most important thing to consider is the cost of the possible state transitions. Of all cache states, the only ones that do not involve a costly cross-cache subsystem communication and data transfer during CPU read/write operations are the Modified (M) and Exclusive (E) states. Thus, the longer the cache line maintains the `M` or `E` states (i.e., the less sharing of data across caches), the lower the coherence cost incurred by a multithreaded application. An example demonstrating how this property has been employed can be found in Nitsan Wakart's blog post "Diving Deeper into Cache Coherency"[227].

Questions and Exercises

1. Solve `perf-ninja::false_sharing` lab assignment.
2. Run the application that you're working with on a daily basis. Is it multithreaded? If not, pick up some multithreaded benchmark. Calculate parallel efficiency metrics. Run the scaling study. Look at the timeline

[223] Contested accesses - https://software.intel.com/en-us/vtune-help-contested-accesses.
[224] Vtune general exploration analysis - https://software.intel.com/en-us/vtune-help-general-exploration-analysis.
[225] Vtune cookbook: false-sharing - https://software.intel.com/en-us/vtune-cookbook-false-sharing.
[226] An article on `perf c2c` - https://joemario.github.io/blog/2016/09/01/c2c-blog/.
[227] Blog post "Diving Deeper into Cache Coherency" - http://psy-lob-saw.blogspot.com/2013/09/diving-deeper-into-cache-coherency.html

diagram, are there any scheduling issues? Identify the hot locks and which code paths lead to those locks. Can you improve locking? Check if the application performance suffers from true/false sharing.

3. Bonus question: what are the benefits of multithreaded vs. multiprocessed applications?

Chapter Summary

- Applications not taking advantage of modern multicore CPUs are lagging behind their competitors. Preparing software to scale well with a growing amount of CPU cores is very important for the future success of the application.
- When dealing with the single-threaded application, optimizing one portion of the program usually yields positive results on performance. However, it's not necessarily the case for multithreaded applications. This effect is widely known as Amdahl's law, which constitutes that the speedup of a parallel program is limited by its serial part.
- Threads communication can yield retrograde speedup as explained by Universal Scalability Law. This poses additional challenges for tuning multithreaded programs. Optimizing the performance of multithreaded applications also involves detecting and mitigating the effects of contention and coherence.
- Intel VTune Profiler is a "go-to" tool for analyzing multithreaded applications. But during the past years, other tools emerged with a unique set of features, e.g., Coz and GAPP.

14 Current And Future Trends in SW and HW performance

14.1 Processing In Memory

[TODO]

14.2 Traditional Elements of CPU Design

[TODO]

14.3 Machine Programming

[TODO]

Questions and Exercises

Chapter Summary

Epilog

Thanks for reading through the whole book. I hope you enjoyed it and found it useful. I would be even happier if the book will help you solve a real-world problem. In such a case, I would consider it a success and proof that my efforts were not wasted. Before you continue with your endeavors, let me briefly highlight the essential points of the book and give you final recommendations:

- HW performance is not increasing as rapidly as it used to a few decades ago. Performance tuning is becoming more critical than it has been for the last 40 years. It will be one of the key drivers for performance gains in the near future.
- Software doesn't have an optimal performance by default. Certain limitations exist that prevent applications to reach their full performance potential. Both HW and SW components have such limitations.
- There is a famous quote by Donald Knuth: "Premature optimization is the root of all evil". But the opposite is often true as well. Postponed performance engineering work may be too late and cause as much evil as premature optimization. Do not neglect performance aspects when designing your future product.
- Performance of modern CPUs is not deterministic and depends on many factors. Meaningful performance analysis should account for noise and use statistical methods to analyze performance measurements.
- Knowledge of the CPU microarchitecture might become handy in understanding the results of experiments you conduct. However, don't rely on this knowledge too much when you make a specific change in your code. Your mental model can never be as accurate as the actual design of the CPU internals. Predicting the performance of a particular piece of code is nearly impossible. *Always measure!*
- Performance is hard because there are no predetermined steps you should follow, no algorithm. Engineers need to tackle problems from different angles. Know performance analysis methods and tools (both HW and SW) that are available to you. I strongly suggest embracing the Roofline model and TMA methodology if they are available on your platform. It will help you to steer your work in the right direction. Also, know when you can leverage other HW performance monitoring features like LBR, PEBS, and PT in your work.
- Understand the limiting factor for the performance of your application and possible ways to fix that. Part 2 covers some of the essential optimizations for every type of CPU performance bottleneck: Front End Bound, Back End Bound, Retiring, Bad Speculation. Use chapters 8-11 to see what options are available when your application falls into one of the four categories mentioned above.
- If the benefit of the modification is negligible, you should keep the code in its most simple and clean form.
- Sometimes modifications that improve performance on one system slow down execution on another system. Make sure you test your changes on all the platforms that you care about.

[TODO]: Performance metrics: be carefull about drawing conclusions just by looking at the aggregate numbers. Don't fall in the trap of "excel performance engineering", i.e. only collect performance metrics and never look at the code. Always seek for a second source of data (e.g. performance profiles, discussed later) that will confirm your hypothesis.

I hope this book will help you better understand your application's performance and CPU performance in general. Of course, it doesn't cover every possible scenario you may encounter while working on performance optimization. My goal was to give you a starting point and to show you potential options and strategies for dealing with performance analysis and tuning on modern CPUs.

If you enjoyed reading this book, make sure to pass it to your friends and colleagues. I would appreciate your help in spreading the word about the book by endorsing it on social media platforms.

I would love to hear your feedback on my email dendibakh@gmail.com. Let me know your thoughts, comments, and suggestions for the book. I will post all the updates and future information about the book on my blog easyperf.net.

Happy performance tuning!

Glossary

AOS Array Of Structures

BB Basic Block

BIOS Basic Input Output System

CI/CD Contiguous Integration/ Contiguous Development

CPI Clocks Per Instruction

CPU Central Processing Unit

DSB Decoded Stream Buffer

DRAM Dynamic Random-Access Memory

DTLB Data Translation Lookaside Buffer

EBS Event-Based Sampling

FLOPS FLoating-point Operations Per Second

FPGA Field-Programmable Gate Array

GPU Graphics processing unit

HFT High-Frequency Trading

HPC High Performance Computing

HW Hardware

I/O Input/Output

IDE Integrated Development Environment

ILP Instruction-Level Parallelism

IPC Instructions Per Clock cycle

IPO Inter-Procedural Optimizations

ITLB Instruction Translation Lookaside Buffer

LBR Last Branch Record

LLC Last Level Cache

LSD Loop Stream Detector

MSR Model Specific Register

MS-ROM Microcode Sequencer Read-Only Memory

NUMA Non-Uniform Memory Access

OS Operating System

PEBS Processor Event-Based Sampling

PGO Profile Guided Optimizations

PMC Performance Monitoring Counter

PMI Performance Monitoring Interrupt

PMU Performance Monitoring Unit

PT Processor Traces

RAT Register Alias Table

ROB ReOrder Buffer

SIMD Single Instruction Multiple Data

SMT Simultaneous MultiThreading

SOA Structure Of Arrays

SW Software

TLB Translation Lookaside Buffer

TMA Top-down Microarchitecture Analysis

TSC Time Stamp Counter

UOP MicroOperation

List of the Major CPU Microarchitectures

In the tables below we present the most recent ISAs and microarchitectures from Intel, AMD, and ARM-based vendors. Of course, not all the designs are listed here. We only include those that we reference in the book or if they represent a big transition in the evolution of the platform.

Table 9: List of the most recent Intel Core microarchitectures.

Name	Three-letter acronym	Year released	Supported ISA client/server chips
Nehalem	NHM	2008	SSE4.2
Sandy Bridge	SNB	2011	AVX
Haswell	HSW	2013	AVX2
Skylake	SKL	2015	AVX2 / AVX512
Sunny Cove	SNC	2019	AVX512
Golden Cove	GLC	2021	AVX2 / AVX512
Redwood Cove	RWC	2023	AVX2 / AVX512

Table 10: List of the most recent AMD microarchitectures.

Name	Year released	Supported ISA
Streamroller	2014	AVX
Excavator	2015	AVX2
Zen	2017	AVX2
Zen2	2019	AVX2
Zen3	2020	AVX2
Zen4	2022	AVX512

Table 11: List of ARM ISAs along with their own and third-party implementations.

ISA	Year released	ARM uarchs (latest)	Third-party uarchs
ARMv7-A (32bit)	2005	Cortex-A17	Apple A6; Quallcomm Scorpion
ARMv8-A	2011	Cortex-A73	Apple A7-A10; Quallcomm Kryo; Samsung M1/M2/M3
ARMv8.2-A	2016	Neoverse N1; Cortex-X1	Apple A11 Samsung M4
ARMv8.4-A	2017	Neoverse V1	Apple A13, M1
ARMv9.0-A (64bit-only)	2018	Neoverse V2; Neoverse N2; Cortex X3	—
ARMv8.6-A (64bit-only)	2019	—	Apple A15,A16,M2
ARMv9.2-A	2020	Cortex X4	—
ARMv9.4-A	2022	—	—

References

[Akinshin, 2019] Akinshin, A. (2019). *Pro .NET Benchmarking* (1 ed.). Apress. https://doi.org/10.1007/978-1-4842-4941-3

[Alam et al., 2019] Alam, M., Gottschlich, J., Tatbul, N., Turek, J. S., Mattson, T., & Muzahid, A. (2019). A zero-positive learning approach for diagnosing software performance regressions. *Advances in Neural Information Processing Systems 32*, 11627–11639. Curran Associates, Inc. http://papers.nips.cc/paper/9337-a-zero-positive-learning-approach-for-diagnosing-software-performance-regressions.pdf

[AMD, 2023] AMD (2023). *AMD64 Architecture Programmer's Manual*. Advanced Micro Devices, Inc. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf

[Arm, 2022a] Arm (2022a). *Arm Architecture Reference Manual Supplement Armv9*. Arm Limited. https://documentation-service.arm.com/static/632dbdace68c6809a6b41710?token=

[Arm, 2022b] Arm (2022b). *Arm Neoverse™ V1 PMU Guide, Revision: r1p2*. Arm. https://developer.arm.com/documentation/PJDOC-1063724031-605393/2-0/?lang=en

[Chen et al., 2016] Chen, D., Li, D. X., & Moseley, T. (2016). Autofdo: Automatic feedback-directed optimization for warehouse-scale applications. *CGO 2016 Proceedings of the 2016 International Symposium on Code Generation and Optimization*, 12–23. https://ieeexplore.ieee.org/document/7559528

[Cooper & Torczon, 2012] Cooper, K. & Torczon, L. (2012). *Engineering a Compiler*. Morgan Kaufmann. Morgan Kaufmann. https://books.google.co.in/books?id=CGTOlAEACAAJ

[Curtsinger & Berger, 2013] Curtsinger, C. & Berger, E. D. (2013). Stabilizer: Statistically sound performance evaluation. *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, 219–228. https://doi.org/10.1145/2451116.2451141

[Curtsinger & Berger, 2018] Curtsinger, C. & Berger, E. D. (2018). Coz: Finding code that counts with causal profiling. *Commun. ACM*, 61(6), 91–99. https://doi.org/10.1145/3205911

[Daly et al., 2020] Daly, D., Brown, W., Ingo, H., O'Leary, J., & Bradford, D. (2020). The use of change point detection to identify software performance regressions in a continuous integration system. *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE '20, 67–75. https://doi.org/10.1145/3358960.3375791

[domo.com, 2017] domo.com (2017). *Data Never Sleeps 5.0*. Domo, Inc. https://www.domo.com/learn/data-never-sleeps-5?aid=ogsm072517_1&sf100871281=1

[Du et al., 2010] Du, J., Sehrawat, N., & Zwaenepoel, W. (2010). Performance profiling in a virtualized environment. *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, 2. https://www.usenix.org/legacy/event/hotcloud10/tech/full_papers/Du.pdf

[Fog, 2004] Fog, A. (2004). *Optimizing software in c++: An optimization guide for windows, linux and mac platforms*. https://www.agner.org/optimize/optimizing_cpp.pdf

[Fog, 2012] Fog, A. (2012). The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers. *Copenhagen University College of Engineering*. https://www.agner.org/optimize/microarchitecture.pdf

[Gregg, 2013] Gregg, B. (2013). *Systems Performance: Enterprise and the Cloud* (1st ed.). Prentice Hall Press.

[Grosser et al., 2012] Grosser, T., Größlinger, A., & Lengauer, C. (2012). Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Process. Lett.*, 22.

[Hennessy, 2018] Hennessy, J. L. (2018). *The future of computing*. https://youtu.be/Azt8Nc-mtKM?t=329

[Hennessy & Patterson, 2017] Hennessy, J. L. & Patterson, D. A. (2017). *Computer Architecture, Sixth Edition: A Quantitative Approach* (6th ed.). Morgan Kaufmann Publishers Inc.

[Ingo & Daly, 2020] Ingo, H. & Daly, D. (2020). Automated system performance testing at mongodb. *Proceedings of the Workshop on Testing Database Systems*, DBTest '20. https://doi.org/10.1145/3395032.3395323

[Intel, 2023a] Intel (2023a). *CPU Metrics Reference*. Intel® Corporation. https://software.intel.com/en-us/vtune-help-cpu-metrics-reference

[Intel, 2023b] Intel (2023b). *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel® Corporation. https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html

[Jimenez & Lin, 2001] Jimenez, D. & Lin, C. (2001). Dynamic branch prediction with perceptrons. *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, 197–206. https://doi.org/10.1109/HPCA.2001.903263

[Jin et al., 2012] Jin, G., Song, L., Shi, X., Scherpelz, J., & Lu, S. (2012). Understanding and detecting real-world performance bugs. *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, 77–88. https://doi.org/10.1145/2254064.2254075

[Kanev et al., 2015] Kanev, S., Darago, J. P., Hazelwood, K., Ranganathan, P., Moseley, T., Wei, G.-Y., & Brooks, D. (2015). Profiling a warehouse-scale computer. *SIGARCH Comput. Archit. News*, 43(3S), 158–169. https://doi.org/10.1145/2872887.2750392

[Kapoor, 2009] Kapoor, R. (2009). Avoiding the cost of branch misprediction. https://software.intel.com/en-us/articles/avoiding-the-cost-of-branch-misprediction

[Khuong & Morin, 2015] Khuong, P.-V. & Morin, P. (2015). *Array layouts for comparison-based searching*. https://arxiv.org/ftp/arxiv/papers/1509/1509.05053.pdf

[Leiserson et al., 2020] Leiserson, C. E., Thompson, N. C., Emer, J. S., Kuszmaul, B. C., Lampson, B. W., Sanchez, D., & Schardl, T. B. (2020). There's plenty of room at the top: What will drive computer performance after moore's law? *Science*, 368(6495). https://doi.org/10.1126/science.aam9744

[Limited, 2023] Limited, A. (2023). *Arm Statistical Profiling Extension: Performance Analysis Methodology*. Arm Limited. https://developer.arm.com/documentation/109429/latest/

[Liu et al., 2019] Liu, M., Sun, X., Varshney, M., & Xu, Y. (2019). *Large-scale online experimentation with quantile metrics*. https://arxiv.org/abs/1903.08762

[Luo et al., 2015] Luo, T., Wang, X., Hu, J., Luo, Y., & Wang, Z. (2015). Improving tlb performance by increasing hugepage ratio. *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 1139–1142. https://doi.org/10.1109/CCGrid.2015.36

[Matteson & James, 2014] Matteson, D. S. & James, N. A. (2014). A nonparametric approach for multiple change point analysis of multivariate data. *Journal of the American Statistical Association*, 109(505), 334–345. https://doi.org/10.1080/01621459.2013.849605

[Mittal, 2016] Mittal, S. (2016). A survey of techniques for cache locking. *ACM Transactions on Design Automation of Electronic Systems*, 21. https://doi.org/10.1145/2858792

[Muła & Lemire, 2019] Muła, W. & Lemire, D. (2019). Base64 encoding and decoding at almost the speed of a memory copy. *Software: Practice and Experience*, 50(2), 89–97. https://doi.org/10.1002/spe.2777

[Mytkowicz et al., 2009] Mytkowicz, T., Diwan, A., Hauswirth, M., & Sweeney, P. F. (2009). Producing wrong data without doing anything obviously wrong! *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, 265–276. https://doi.org/10.1145/1508244.1508275

[Nair & Field, 2020] Nair, R. & Field, T. (2020). Gapp: A fast profiler for detecting serialization bottlenecks in parallel linux applications. *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. https://doi.org/10.1145/3358960.3379136

[Nowak & Bitzes, 2014] Nowak, A. & Bitzes, G. (2014). The overhead of profiling using pmu hardware counters. https://zenodo.org/record/10800/files/TheOverheadOfProfilingUsingPMUhardwareCounters.pdf

[Ottoni & Maher, 2017] Ottoni, G. & Maher, B. (2017). Optimizing function placement for large-scale data-center applications. *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, 233–244. https://ieeexplore.ieee.org/document/7863743

[Panchenko et al., 2018] Panchenko, M., Auler, R., Nell, B., & Ottoni, G. (2018). BOLT: A practical binary optimizer for data centers and beyond. *CoRR*, abs/1807.06735. http://arxiv.org/abs/1807.06735

[Paoloni, 2010] Paoloni, G. (2010). *How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures*. Intel® Corporation. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf

[Pharr & Mark, 2012] Pharr, M. & Mark, W. R. (2012). ispc: A spmd compiler for high-performance cpu programming. *2012 Innovative Parallel Computing (InPar)*, 1–13. https://doi.org/10.1109/InPar.2012.6339601

[Ren et al., 2010] Ren, G., Tune, E., Moseley, T., Shi, Y., Rus, S., & Hundt, R. (2010). Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, 65–79. http://www.computer.org/portal/web/csdl/doi/10.1109/MM.2010.68

[Sasongko et al., 2023] Sasongko, M. A., Chabbi, M., Kelly, P. H. J., & Unat, D. (2023). Precise event sampling on amd versus intel: Quantitative and qualitative comparison. *IEEE Transactions on Parallel and Distributed Systems*, 34(5), 1594–1608. https://doi.org/10.1109/TPDS.2023.3257105

[Seznec & Michaud, 2006] Seznec, A. & Michaud, P. (2006). A case for (partially) tagged geometric history length branch prediction. *J. Instr. Level Parallelism*, 8. https://inria.hal.science/hal-03408381/document

[Sharma, 2016] Sharma, S. D. (2016). Hardware-assisted instruction profiling and latency detection. *The Journal of Engineering*, 2016, 367–376(9). https://digital-library.theiet.org/content/journals/10.1049/joe.2016.0127

[statista.com, 2018] statista.com (2018). *Volume of data/information created worldwide from 2010 to 2025*. Statista, Inc. https://www.statista.com/statistics/871513/worldwide-data-created/

[Suresh Srinivas, 2019] Suresh Srinivas, e. a. (2019). *Runtime performance optimization blueprint: Intel® architecture optimization with large code pages*. https://www.intel.com/content/www/us/en/develop/articles/runtime-performance-optimization-blueprint-intel-architecture-optimization-with-large-code.html

[Yasin, 2014] Yasin, A. (2014). A top-down method for performance analysis and counters architecture. 35–44. https://doi.org/10.1109/ISPASS.2014.6844459

Appendix A. Reducing Measurement Noise

Below are some examples of features that can contribute to increased non-determinism in performance measurements. See complete discussion in Section 2.1.

Dynamic Frequency Scaling

Dynamic Frequency Scaling[228] (DFS) is a technique to increase the performance of the system by automatically raising CPU operating frequency when it runs demanding tasks. As an example of DFS implementation, Intel CPUs have a feature called Turbo Boost[229] and AMD CPUs employ Turbo Core[230]functionality.

Example of an impact Turbo Boost can make for a single-threaded workload running on Intel® Core™ i5-8259U:

```
# TurboBoost enabled
$ cat /sys/devices/system/cpu/intel_pstate/no_turbo
0
$ perf stat -e task-clock,cycles -- ./a.exe
    11984.691958  task-clock (msec) #    1.000 CPUs utilized
  32,427,294,227  cycles            #    2.706 GHz
      11.989164338 seconds time elapsed

# TurboBoost disabled
$ echo 1 | sudo tee /sys/devices/system/cpu/intel_pstate/no_turbo
1
$ perf stat -e task-clock,cycles -- ./a.exe
    13055.200832  task-clock (msec) #    0.993 CPUs utilized
  29,946,969,255  cycles            #    2.294 GHz
      13.142983989 seconds time elapsed
```

The average frequency is much higher when Turbo Boost is on.

DFS can be permanently disabled in BIOS.[231] To programmatically disable the DFS feature on Linux systems, you need root access. Here is how one can achieve this:

```
# Intel
echo 1 > /sys/devices/system/cpu/intel_pstate/no_turbo
# AMD
echo 0 > /sys/devices/system/cpu/cpufreq/boost
```

Simultaneous Multithreading

Modern CPU cores are often made in the simultaneous multithreading (SMT[232]) manner. It means that in one physical core, you can have two threads of simultaneous execution. Typically, architectural state[233] is replicated, but the execution resources (ALUs, caches, etc.) are not. That means that if we have two separate processes running on the same core "simultaneously" (in different threads), they can steal resources from each other, for example, cache space.

SMT can be permanently disabled in BIOS.[234] To programmatically disable SMT on Linux systems, you need root access. Here is how one can turn down a sibling thread in each core:

[228] Dynamic frequency scaling - https://en.wikipedia.org/wiki/Dynamic_frequency_scaling.
[229] Intel Turbo Boost - https://en.wikipedia.org/wiki/Intel_Turbo_Boost.
[230] AMD Turbo Core - https://en.wikipedia.org/wiki/AMD_Turbo_Core.
[231] Intel Turbo Boost FAQ - https://www.intel.com/content/www/us/en/support/articles/000007359/processors/intel-core-processors.html.
[232] SMT - https://en.wikipedia.org/wiki/Simultaneous_multithreading.
[233] Architectural state - https://en.wikipedia.org/wiki/Architectural_state.
[234] "How to disable hyperthreading" - https://www.pcmag.com/article/314585/how-to-disable-hyperthreading.

```
echo 0 > /sys/devices/system/cpu/cpuX/online
```

The sibling pairs of CPU threads can be found in the following files:

```
/sys/devices/system/cpu/cpuN/topology/thread_siblings_list
```

For example, on Intel® Core™ i5-8259U, which has 4 cores and 8 threads:

```
# all 8 HW threads enabled:
$ lscpu
...
CPU(s):              8
On-line CPU(s) list: 0-7
...
$ cat /sys/devices/system/cpu/cpu0/topology/thread_siblings_list
0,4
$ cat /sys/devices/system/cpu/cpu1/topology/thread_siblings_list
1,5
$ cat /sys/devices/system/cpu/cpu2/topology/thread_siblings_list
2,6
$ cat /sys/devices/system/cpu/cpu3/topology/thread_siblings_list
3,7

# Disabling SMT on core 0
$ echo 0 | sudo tee /sys/devices/system/cpu/cpu4/online
0
$ lscpu
CPU(s):               8
On-line CPU(s) list:  0-3,5-7
Off-line CPU(s) list: 4
...
$ cat /sys/devices/system/cpu/cpu0/topology/thread_siblings_list
0
```

Scaling Governor

Linux kernel is able to control CPU frequency for different purposes. One such purpose is to save the power, in which case the scaling governor[235] inside the Linux Kernel can decide to decrease CPU operating frequency. For performance measurements, it is recommended to set the scaling governor policy to `performance` to avoid sub-nominal clocking. Here is how we can set it for all the cores:

```
for i in /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor
do
  echo performance > $i
done
```

CPU Affinity

Processor affinity[236] enables the binding of a process to a certain CPU core(s). In Linux, one can do this with `taskset`[237] tool. Here

```
# no affinity
$ perf stat -e context-switches,cpu-migrations -r 10 -- a.exe
               151      context-switches
                10      cpu-migrations
```

[235] Documentation for Linux CPU frequency governors: https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.
[236] Processor affinity - https://en.wikipedia.org/wiki/Processor_affinity.
[237] `taskset` manual - https://linux.die.net/man/1/taskset.

```
# process is bound to the CPU0
$ perf stat -e context-switches,cpu-migrations -r 10 -- taskset -c 0 a.exe
               102      context-switches
                 0      cpu-migrations
```

notice the number of `cpu-migrations` gets down to `0`, i.e., the process never leaves the `core0`.

Alternatively, you can use cset[238] tool to reserve CPUs for just the program you are benchmarking. If using `Linux perf`, leave at least two cores so that `perf` runs on one core, and your program runs in another. The command below will move all threads out of N1 and N2 (`-k on` means that even kernel threads are moved out):

```
$ cset shield -c N1,N2 -k on
```

The command below will run the command after `--` in the isolated CPUs:

```
$ cset shield --exec -- perf stat -r 10 <cmd>
```

Process Priority

In Linux, one can increase process priority using the `nice` tool. By increasing the priority process gets more CPU time, and the Linux scheduler favors it more in comparison with processes with normal priority. Niceness ranges from `-20` (highest priority value) to `19` (lowest priority value) with the default of `0`.

Notice in the previous example, execution of the benchmarked process was interrupted by the OS more than 100 times. If we increase process priority by run the benchmark with `sudo nice -n -N`:

```
$ perf stat -r 10 -- sudo nice -n -5 taskset -c 1 a.exe
    0   context-switches
    0   cpu-migrations
```

Notice the number of context-switches gets to `0`, so the process received all the computation time uninterrupted.

Filesystem Cache

Usually, an area of main memory is assigned to cache the file system contents, including various data. This reduces the need for an application to go all the way down to the disk. Here is an example of how file system cache can affect the running time of simple `git status` command:

```
# clean fs cache
$ echo 3 | sudo tee /proc/sys/vm/drop_caches && sync && time -p git status
real 2,57
# warmed fs cache
$ time -p git status
real 0,40
```

One can drop the current filesystem cache by running the following two commands:

```
$ echo 3 | sudo tee /proc/sys/vm/drop_caches
$ sync
```

Alternatively, you can make one dry run just to warm up the filesystem cache and exclude it from the measurements. This dry iteration can be combined with the validation of the benchmark output.

[238] `cpuset` manual - https://github.com/lpechacek/cpuset.

Appendix B. The LLVM Vectorizer

This section describes the state of the LLVM Loop Vectorizer inside the Clang compiler as of the year 2020. Innerloop vectorization is the process of transforming code in the innermost loops into code that uses vectors across multiple loop iterations. Each lane in the SIMD vector performs independent arithmetic on consecutive loop iterations. Usually, loops are not found in a clean state, and the Vectorizer has to guess and assume missing information and check for details at runtime. If the assumptions are proven wrong, the Vectorizer falls back to running the scalar loop. The examples below highlight some of the code patterns that the LLVM Vectorizer supports.

Loops with unknown trip count

The LLVM Loop Vectorizer supports loops with an unknown trip count. In the loop below, the iteration start and finish points are unknown, and the Vectorizer has a mechanism to vectorize loops that do not start at zero. In this example, `n` may not be a multiple of the vector width, and the Vectorizer has to execute the last few iterations as scalar code. Keeping a scalar copy of the loop increases the code size.

```
void bar(float* A, float* B, float K, int start, int end) {
  for (int i = start; i < end; ++i)
    A[i] *= B[i] + K;
}
```

Runtime Checks of Pointers

In the example below, if the pointers A and B point to consecutive addresses, then it is illegal to vectorize the code because some elements of A will be written before they are read from array B.

Some programmers use the `restrict` keyword to notify the compiler that the pointers are disjointed, but in our example, the LLVM Loop Vectorizer has no way of knowing that the pointers A and B are unique. The Loop Vectorizer handles this loop by placing code that checks, at runtime, if the arrays A and B point to disjointed memory locations. If arrays A and B overlap, then the scalar version of the loop is executed.

```
void bar(float* A, float* B, float K, int n) {
  for (int i = 0; i < n; ++i)
    A[i] *= B[i] + K;
}
```

Reductions

In this example, the sum variable is used by consecutive iterations of the loop. Normally, this would prevent vectorization, but the Vectorizer can detect that `sum` is a reduction variable. The variable `sum` becomes a vector of integers, and at the end of the loop, the elements of the array are added together to create the correct result. The LLVM Vectorizer supports a number of different reduction operations, such as addition, multiplication, XOR, AND, and OR.

```
int foo(int *A, int n) {
  unsigned sum = 0;
  for (int i = 0; i < n; ++i)
    sum += A[i] + 5;
  return sum;
}
```

The LLVM Vectorizer supports floating-point reduction operations when `-ffast-math` is used.

Inductions

In this example, the value of the induction variable i is saved into an array. The LLVM Loop Vectorizer knows to vectorize induction variables.

```
void bar(float* A, int n) {
  for (int i = 0; i < n; ++i)
    A[i] = i;
}
```

If Conversion

The LLVM Loop Vectorizer is able to "flatten" the IF statement in the code and generate a single stream of instructions. The Vectorizer supports any control flow in the innermost loop. The innermost loop may contain complex nesting of IFs, ELSEs, and even GOTOs.

```
int foo(int *A, int *B, int n) {
  unsigned sum = 0;
  for (int i = 0; i < n; ++i)
    if (A[i] > B[i])
      sum += A[i] + 5;
  return sum;
}
```

Pointer Induction Variables

This example uses the `std::accumulate` function from the standard c++ library. This loop uses C++ iterators, which are pointers, and not integer indices. The LLVM Loop Vectorizer detects pointer induction variables and can vectorize this loop. This feature is important because many C++ programs use iterators.

```
int baz(int *A, int n) {
  return std::accumulate(A, A + n, 0);
}
```

Reverse Iterators

The LLVM Loop Vectorizer can vectorize loops that count backward.

```
int foo(int *A, int n) {
  for (int i = n; i > 0; --i)
    A[i] +=1;
}
```

Scatter / Gather

The LLVM Loop Vectorizer can vectorize code that becomes a sequence of scalar instructions that scatter/gathers memory.

```
int foo(int * A, int * B, int n) {
  for (intptr_t i = 0; i < n; ++i)
      A[i] += B[i * 4];
}
```

In many situations, the cost model will decide that this transformation is not profitable.

Vectorization of Mixed Types

The LLVM Loop Vectorizer can vectorize programs with mixed types. The Vectorizer cost model can estimate the cost of the type conversion and decide if vectorization is profitable.

```
int foo(int *A, char *B, int n) {
  for (int i = 0; i < n; ++i)
    A[i] += 4 * B[i];
}
```

Vectorization of function calls

The LLVM Loop Vectorizer can vectorize intrinsic math functions. See the table below for a list of these functions.

```
pow        exp        exp2
sin        cos        sqrt
log        log2       log10
fabs       floor      ceil
fma        trunc      nearbyint
fmuladd
```

Partial unrolling during vectorization

Modern processors feature multiple execution units, and only programs that contain a high degree of parallelism can fully utilize the entire width of the machine. The LLVM Loop Vectorizer increases the instruction-level parallelism (ILP) by performing partial-unrolling of loops.

In the example below, the entire array is accumulated into the variable `sum`. This is inefficient because only a single execution port can be used by the processor. By unrolling the code, the Loop Vectorizer allows two or more execution ports to be used simultaneously.

```
int foo(int *A, int n) {
  unsigned sum = 0;
  for (int i = 0; i < n; ++i)
      sum += A[i];
  return sum;
}
```

The LLVM Loop Vectorizer uses a cost model to decide when it is profitable to unroll loops. The decision to unroll the loop depends on the register pressure and the generated code size.

SLP vectorization

SLP (Superword-Level Parallelism) vectorizer tries to glue multiple scalar operations together into vector operations. It processes the code bottom-up, across basic blocks, in search of scalars to combine. The goal of SLP vectorization is to combine similar independent instructions into vector instructions. Memory accesses, arithmetic operations, comparison operations can all be vectorized using this technique. For example, the following function performs very similar operations on its inputs (a1, b1) and (a2, b2). The basic-block vectorizer may combine the following function into vector operations.

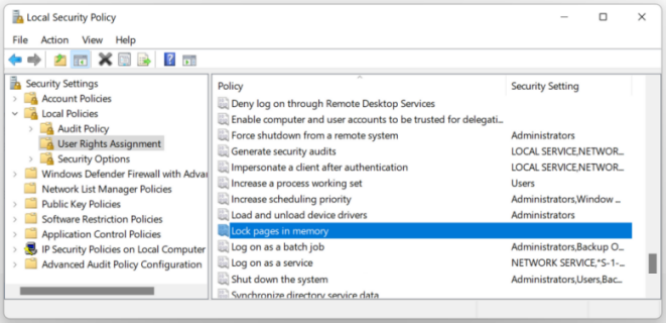
```
void foo(int a1, int a2, int b1, int b2, int *A) {
  A[0] = a1*(a1 + b1);
  A[1] = a2*(a2 + b2);
  A[2] = a1*(a1 + b1);
  A[3] = a2*(a2 + b2);
}
```

Appendix C. Enable Huge Pages

14.4 Windows

To utilize huge pages on Windows, one needs to enable `SeLockMemoryPrivilege` security policy. This can be done programatically via the Windows API, or alternatively via the security policy GUI.

1. Hit start -> search "secpol.msc", launch it.
2. On the left select "Local Policies" -> "User Rights Assignment", then double-click on "Lock pages in memory".



Windows security: Lock pages in memory

3. Add your user and reboot the machine.
4. Check that huge pages are used at runtime with RAMMap tool.

Use huge pages in the code with:

```
void* p = VirtualAlloc(NULL, size, MEM_RESERVE |
                                   MEM_COMMIT |
                                   MEM_LARGE_PAGES,
                       PAGE_READWRITE);
...
VirtualFree(ptr, 0, MEM_RELEASE);
```

14.5 Linux

On Linux OS, there are two ways of using huge pages in an application: Explicit and Transparent Huge Pages.

Explicit hugepages

Explicit huge pages can be reserved at boot time or at run time. To make a permanent change to force the Linux kernel to allocate 128 huge pages at the boot time, run the following command:

```
$ echo "vm.nr_hugepages = 128" >> /etc/sysctl.conf
```

To explicitly allocate a fixed number of huge pages, one can use libhugetlbfs. The following command preallocates 128 huge pages.

```
$ sudo apt install libhugetlbfs-bin
$ sudo hugeadm --create-global-mounts
$ sudo hugeadm --pool-pages-min 2M:128
```

This is roughly the equivalent of executing the following commands which do not require libhugetlbfs (see the kernel docs):

```
$ echo 128 > /proc/sys/vm/nr_hugepages
$ mount -t hugetlbfs                                                    \
    -o uid=<value>,gid=<value>,mode=<value>,pagesize=<value>,size=<value>,\
    min_size=<value>,nr_inodes=<value> none /mnt/huge
```

You should be able to observe the effect in `/proc/meminfo`. Note that it is a system-wide view and not per-process:

```
$ watch -n1 "cat /proc/meminfo  | grep huge -i"
AnonHugePages:      2048 kB
ShmemHugePages:        0 kB
FileHugePages:         0 kB
HugePages_Total:     128    <== 128 huge pages allocated
HugePages_Free:      128
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
Hugetlb:          262144 kB <== 256MB of space occupied
```

Developers can utilize explicit huge pages in the code by calling `mmap` with `MAP_HUGETLB` flag (full example[239]):

```
void ptr = mmap(nullptr, size, PROT_READ | PROT_WRITE,
                MAP_PRIVATE | MAP_ANONYMOUS | MAP_HUGETLB, -1, 0);
...
munmap(ptr, size);
```

Other alternatives include:

- `mmap` using a file from a mounted `hugetlbfs` filesystem (exampe code[240]).
- `shmget` using the `SHM_HUGETLB` flag (exampe code[241]).

Transparent hugepages

To allow application use Transparent Huge Pages (THP) on Linux one should make sure that `/sys/kernel/mm/transparent_hugep` is `always` or `madvise`. The former enables system wide usage of THPs, while the latter gives control to the user code which memory regions should use THPs, thus avoids the risk of consuming more memory resources. Below is the example of using the `madvise` approach:

```
void ptr = mmap(nullptr, size, PROT_READ | PROT_WRITE | PROT_EXEC,
                MAP_PRIVATE | MAP_ANONYMOUS, -1 , 0);
madvise(ptr, size, MADV_HUGEPAGE);
...
munmap(ptr, size);
```

You can observe the system-wide effect in `/proc/meminfo` under `AnonHugePages`:

```
$ watch -n1 "cat /proc/meminfo  | grep huge -i"
AnonHugePages:     61440 kB     <== 30 transparent huge pages are in use
HugePages_Total:     128
HugePages_Free:      128        <== explicit huge pages are not used
```

Also, developers can observe how their application utilizes EHPs and/or THPs by looking at `smaps` file specific to their process:

```
$ watch -n1 "cat /proc/<PID_OF_PROCESS>/smaps"
```

[239] MAP_HUGETLB example - https://github.com/torvalds/linux/blob/master/tools/testing/selftests/vm/map_hugetlb.c.
[240] Mounted `hugetlbfs` filesystem - https://github.com/torvalds/linux/blob/master/tools/testing/selftests/vm/hugepage-mmap.c.
[241] SHM_HUGETLB example - https://github.com/torvalds/linux/blob/master/tools/testing/selftests/vm/hugepage-shm.c.

Appendix D. Intel Processor Traces

The Intel Processor Traces (PT) is a CPU feature that records the program execution by encoding packets in a highly compressed binary format that can be used to reconstruct execution flow with a timestamp on every instruction. PT has extensive coverage and relatively small overhead,[242] which is usually below `5%`. Its main usages are postmortem analysis and root-causing performance glitches.

Workflow

Similar to sampling techniques, PT does not require any modifications to the source code. All you need to collect traces is just to run the program under the tool that supports PT. Once PT is enabled and the benchmark launches, the analysis tool starts writing tracing packets to DRAM.

Similar to LBR (Last Branch Records), Intel PT works by recording branches. At runtime, whenever a CPU encounters any branch instruction, PT will record the outcome of this branch. For a simple conditional jump instruction, a CPU will record whether it was taken (`T`) or not taken (`NT`) using just 1 bit. For an indirect call, PT will record the destination address. Note that unconditional branches are ignored since we statically know their targets.

An example of encoding for a small instruction sequence is shown in Figure 83. Instructions like `PUSH`, `MOV`, `ADD`, and `CMP` are ignored because they don't change the control flow. However, `JE` instruction may jump to `.label`, so its result needs to be recorded. Later there is an indirect call for which destination address is saved.

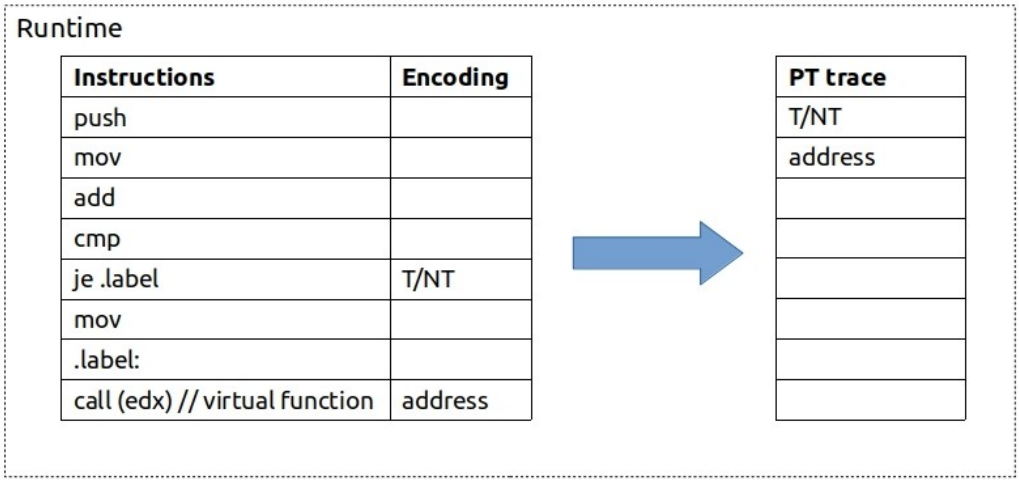


Figure 83: Intel Processor Traces encoding

At the time of analysis, we bring together the application binary and the collected PT trace. A SW decoder needs the application binary file to reconstruct the execution flow of the program. It starts from the entry point and then uses collected traces as a lookup reference to determine the control flow. Figure 84 shows an example of decoding Intel Processor Traces. Suppose that the `PUSH` instruction is an entry point of the application binary file. Then `PUSH`, `MOV`, `ADD`, and `CMP` are reconstructed as-is without looking into encoded traces. Later, the SW decoder encounters a `JE` instruction, which is a conditional branch and for which we need to look up the outcome. According to the traces in Figure 84, `JE` was taken (`T`), so we skip the next `MOV` instruction and go to the `CALL` instruction. Again, `CALL(edx)` is an instruction that changes the control flow, so we look up the destination address in encoded traces, which is `0x407e1d8`. Instructions highlighted in yellow were executed when our program was running. Note that this is *exact* reconstruction of program execution; we did not skip any instruction. Later we can map assembly instructions back to the source code by using debug information and have a log of source code that was executed line by line.

[242] See more information about Intel PT overhead in [Sharma, 2016].

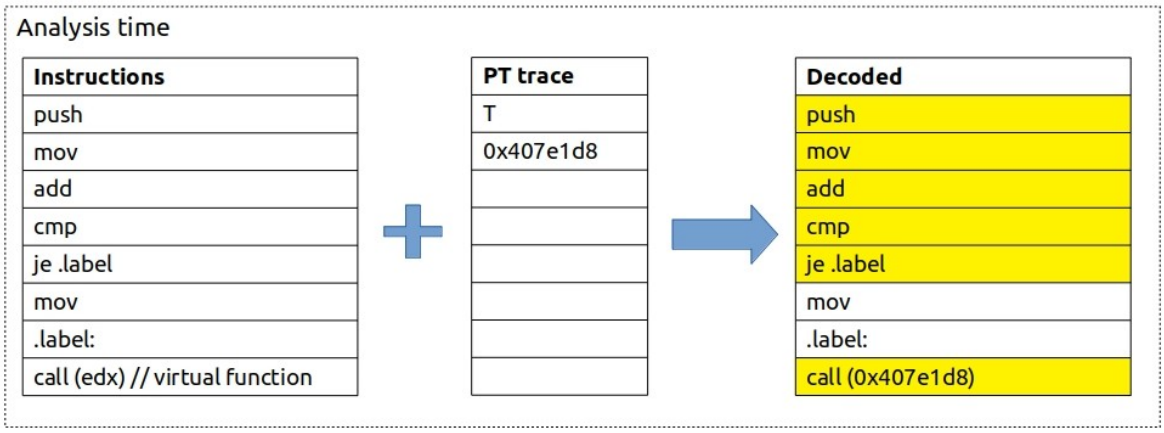


Figure 84: Intel Processor Traces decoding

Timing Packets

With Intel PT, not only execution flow can be traced but also timing information. In addition to saving jump destinations, PT can also emit timing packets. Figure 85 provides a visualization of how time packets can be used to restore timestamps for instructions. As in the previous example, we first see that `JNZ` was not taken, so we update it and all the instructions above with timestamp 0ns. Then we see a timing update of 2ns and `JE` being taken, so we update it and all the instructions above `JE` (and below `JNZ`) with timestamp 2ns. After that, there is an indirect call, but no timing packet is attached to it, so we do not update timestamps. Then we see that 100ns elapsed, and `JB` was not taken, so we update all the instructions above it with the timestamp of 102ns.

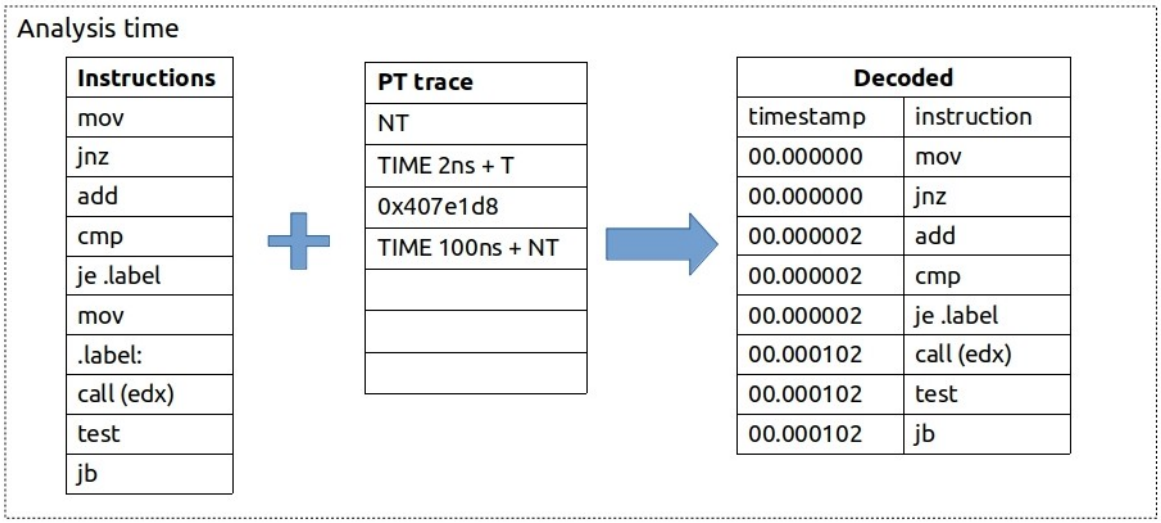


Figure 85: Intel Processor Traces timings

In the example shown in Figure 85, instruction data (control flow) is perfectly accurate, but timing information is less accurate. Obviously, `CALL(edx)`, `TEST`, and `JB` instructions were not happening at the same time, yet we do not have more accurate timing information for them. Having timestamps enables us to align the time interval of our program with another event in the system, and it's easy to compare to wall clock time. Trace timing in some implementations can further be improved by a cycle-accurate mode, in which the hardware keeps a record of cycle counts between normal packets (see more details in [Intel, 2023b, Volume 3C, Chapter 36]).

Collecting and Decoding Traces

Intel PT traces can be easily collected with the Linux `perf` tool:

```
$ perf record -e intel_pt/cyc=1/u ./a.out
```

In the command line above, we asked the PT mechanism to update timing information every cycle. But likely, it will not increase our accuracy greatly since timing packets will only be sent when paired with another control flow packet.

After collecting, raw PT traces can be obtained by executing:

```
$ perf report -D > trace.dump
```

PT bundles up to 6 conditional branches before it emits a timing packet. Since the Intel Skylake CPU generation, timing packets have cycle count elapsed from the previous packet. If we then look into the `trace.dump`, we might see something like the following:

```
000073b3: 2d 98 8c  TIP 0x8c98     // target address (IP)
000073b6: 13        CYC 0x2        // timing update
000073b7: c0        TNT TNNNNN (6) // 6 conditional branches
000073b8: 43        CYC 0x8        // 8 cycles passed
000073b9: b6        TNT NTTNTT (6)
```

Above we showed the raw PT packets, which are not very useful for performance analysis. To decode processor traces to human-readable form, one can execute:

```
$ perf script --ns --itrace=i1t -F time,srcline,insn,srccode
```

Below is the example of decoded traces one might get:

```
timestamp       srcline   instruction      srccode
...
253.555413143:  a.cpp:24  call 0x35c       foo(arr, j);
253.555413143:  b.cpp:7   test esi, esi    for (int i = 0; i <= n; i++)
253.555413508:  b.cpp:7   js 0x1e
253.555413508:  b.cpp:7   movsxd rsi, esi
...
```

Above is shown just a small snippet from the long execution log. In this log, we have traces of *every* instruction executed while our program was running. We can literally observe every step that was made by the program. It is a very strong foundation for further functional and performance analysis.

Use Cases

1. **Analyze performance glitches**: because PT captures the entire instruction stream, it is possible to analyze what was going on during the small-time period when the application was not responding. More detailed examples can be found in an article[243] on easyperf blog.
2. **Postmortem debugging**: PT traces can be replayed by traditional debuggers like `gdb`. In addition to that, PT provides call stack information, which is *always* valid even if the stack is corrupted.[244] PT traces could be collected on a remote machine once and then analyzed offline. This is especially useful when the issue is hard to reproduce or access to the system is limited.
3. **Introspect execution of the program**:
   - We can immediately tell if a code path was never executed.
   - Thanks to timestamps, it's possible to calculate how much time was spent waiting while spinning on a lock attempt, etc.
   - Security mitigation by detecting specific instruction pattern.

[243] Analyze performance glitches with Intel PT - https://easyperf.net/blog/2019/09/06/Intel-PT-part3.
[244] Postmortem debugging with Intel PT - https://easyperf.net/blog/2019/08/30/Intel-PT-part2.

Disk Space and Decoding Time

Even taking into account the compressed format of the traces, encoded data can consume a lot of disk space. Typically, it's less than 1 byte per instruction, however taking into account the speed at which CPU executes instructions, it is still a lot. Depending on a workload, it's very common for CPU to encode PT at a speed of 100 MB/s. Decoded traces might easily be ten times more (~1GB/s). This makes PT not practical for using on long-running workloads. But it is affordable to run it for a small period of time, even on a big workload. In this case, the user can attach to the running process just for the period of time when the glitch happened. Or they can use a circular buffer, where new traces will overwrite old ones, i.e., always having traces for the last 10 seconds or so.

Users can limit collection even further in several ways. They can limit collecting traces only on user/kernel space code. Also, there is an address range filter, so it's possible to opt-in and opt-out of tracing dynamically to limit the memory bandwidth. This allows us to trace just a single function or even a single loop.

Decoding PT traces can take a long time. On an Intel Core i5-8259U machine, for a workload that runs for 7 milliseconds, encoded PT trace consumes around 1MB of disk space. Decoding this trace using `perf script` takes ~20 seconds. The decoded output from `perf script -F time,ip,sym,symoff,insn` takes ~1.3GB of disk space. As of February 2020, decoding traces with `perf script -F` with `+srcline` or `+srccode` gets extremely slow and is not practical for daily usage. The implementation of Linux perf should be improved.

Intel PT References and links

- Intel® 64 and IA-32 Architectures Software Developer Manuals [Intel, 2023b, Volume 3C, Chapter 36].
- Whitepaper "Hardware-assisted instruction profiling and latency detection" [Sharma, 2016].
- Andi Kleen article on LWN, URL: https://lwn.net/Articles/648154.
- Intel PT Micro Tutorial, URL: https://sites.google.com/site/intelptmicrotutorial/.
- simple_pt: Simple Intel CPU processor tracing on Linux, URL: https://github.com/andikleen/simple-pt/.
- Intel PT documentation in the Linux kernel, URL: https://github.com/torvalds/linux/blob/master/tools/perf/Documentation/intel-pt.txt.
- Cheatsheet for Intel Processor Trace, URL: http://halobates.de/blog/p/410.