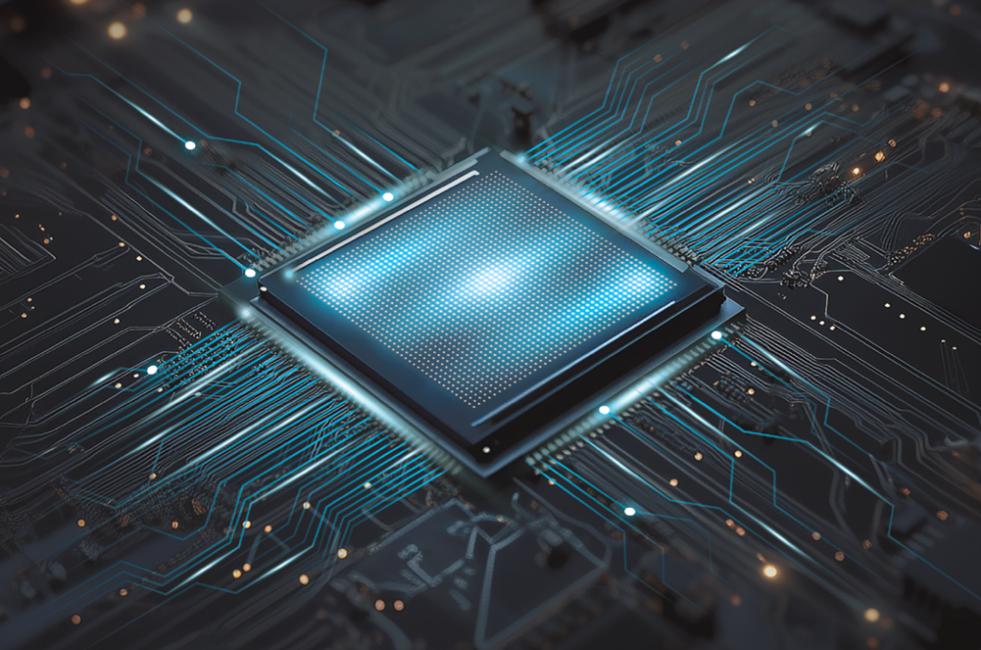


PERFORMANCE ANALYSIS AND TUNING ON MODERN CPUs

Denis Bakhvalov



second edition

- **SOFTWARE DEVELOPERS GUIDE**

for discovering and implementing
low-level optimizations



- Curated by experts from Intel, AMD, ARM, Google, and leading HFT firms

Notices

Responsibility. Knowledge and best practices 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 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, contributors, nor editors, assume any liability for any injury and/or damage to persons or property as a matter of product 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 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

Copyright © 2024 by Denis Bakhvalov.

Preface

About The Author

Denis Bakhvalov is a performance engineer at Intel, where he works on optimizing production applications and benchmarks. Before that, he was a part of the Intel compiler team, that develops C++ compilers for a variety of different architectures. Denis started his career as a software developer in 2008, working on a large C++ enterprise financial application. Before joining Intel, he worked for three years at Nokia, where he was writing embedded software.

Performance engineering and compilers were always among his primary interests. In 2016 Denis started [easypf.net](#) blog where he writes about low-level performance optimizations, C/C++ compilers, and CPU microarchitecture. Away from work, Denis enjoys soccer, chess, and traveling.

Contacts:

- Email: dendibakh@gmail.com
- X (formerly Twitter): [@dendibakh](#)
- LinkedIn: [@dendibakh](#)

From The Author

I started this book with a simple goal: educate software developers to better understand their applications' performance. I know how difficult the topic of low-level performance engineering might be for a beginner or even for an experienced developer. I remember the days when I was starting with performance analysis. I was looking at unfamiliar metrics trying to match 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... even though sometimes I still struggle with the same problems.

When I was taking my first steps in performance engineering, the only good sources of information on the topic were software developer manuals, which are not what mainstream developers like to read. Frankly, I wish I had this book when I was trying to learn low-level performance analysis. In 2016 I started sharing things that I learned on my blog, and received some positive feedback from my readers. Some of them suggested I aggregate this information into a book. This book is their fault.

Many people have asked me why I decided to self-publish. In fact, I initially tried to pitch it to several reputable publishers, but they didn't see the financial benefits of making such a book. However, I really wanted to write it, so I decided to do it anyway. In the end, it turned out quite well, so I decided to self-publish the second edition too.

The first edition was released in November 2020. It was well-received by the community, however, readers gave me a lot of constructive criticism. The most popular feedback was to include exercises for experimentation. Some readers complained that it was too focused on Intel CPUs and didn't cover other architectures like AMD and ARM. Other readers suggested that I should cover system performance, not just CPU performance.

The second edition expands in all these and many other directions. It came out to be twice as big as the first book.

Specifically, I want to highlight the exercises that I added to the second edition of the book. I created a supplementary online course “Performance Ninja” with more than twenty lab assignments. I recommend you use these small puzzles to practice optimization techniques and check your understanding of the material. I consider it the best part that differentiates this book from others. I hope it will make your learning process more entertaining. More details about the online course can be found in Section 1.7.

I know firsthand that low-level performance optimization is not easy. I tried to explain everything as clearly as possible, but the topic is very complex. It requires experimentation and practice to fully understand the material. I encourage you to take your time, read through the chapters, and experiment with examples provided in the online course.

During my career, I never shied away from software optimization tasks. I always got a dopamine hit whenever I managed to make my program run faster. The excitement of discovering something and feeling proud left me even more curious and craving for more. My initial performance work was very unstructured. Now it is my profession, yet I still feel very happy when I make software run faster. I hope you also experience the joy of discovering performance issues, and the satisfaction of fixing them.

I sincerely hope that this book will help you learn low-level performance analysis. If you make your application faster as a result, I will consider my mission accomplished.

You will find that I use “we” instead of “I” in some places in the book. This is because I received a lot of help from other people. The full list of contributors can be found at the end of the book in the “Acknowledgements” section.

The PDF version of this book and the “Performance Ninja” online course are available for free. This is my way to give back to the community.

Target Audience

If you’re working with performance-critical applications, this book is right for you. It is primarily targeted at software developers in High-Performance Computing (HPC), AI, game development, data center applications (like those at Meta, Google, etc.), High-Frequency Trading (HFT), and other industries where the value of performance optimizations is well known and appreciated.

This book will also be useful for any developer who wants to understand the performance of their application better and know how it can be improved. You may just be enthusiastic about performance engineering and want to learn more about it. Or you may want to be the smartest engineer in the room; that’s also fine. I hope that the material presented in this book will help you develop new skills that can be applied in your daily work and potentially move your career forward.

A minimal background in the C and C++ programming languages is necessary to understand the book’s examples. The ability to read basic x86/ARM assembly is desirable, but not a strict requirement. I also expect familiarity with basic concepts

of computer architecture and operating systems like “CPU”, “memory”, “process”, “thread”, “virtual” and “physical memory”, “context switch”, etc. If any of these terms are new to you, I suggest studying these prerequisites first.

I suggest you read the book chapter by chapter, starting from the beginning. If you consider yourself a beginner in performance analysis, I do not recommend skipping chapters. After you finish reading, you can use this book as a source of ideas whenever you face a performance issue and are unsure how to fix it. You can skim through the second part of the book to see which optimization techniques can be applied to your code.

I will post errata and other information about the book on my blog at the following URL: <https://easyperf.net/blog/2024/11/11/Book-Updates-Errata>.

Table Of Contents

Table Of Contents	6
1 Introduction	11
1.1 Why Is Software Slow?	12
1.2 Why Care about Performance?	14
1.3 What Is Performance Analysis?	17
1.4 What Is Performance Tuning?	17
1.5 What Is Discussed in this Book?	18
1.6 What Is Not Discussed in this Book?	20
1.7 Exercises	21
Part 1. Performance Analysis on a Modern CPU	23
2 Measuring Performance	23
2.1 Noise in Modern Systems	24
2.2 Measuring Performance in Production	26
2.3 Continuous Benchmarking	26
2.4 Manual Performance Testing	30
2.5 Software and Hardware Timers	33
2.6 Microbenchmarks	34
2.7 Active Benchmarking	35
3 CPU Microarchitecture	38
3.1 Instruction Set Architecture	38
3.2 Pipelining	39
3.3 Exploiting Instruction Level Parallelism (ILP)	41
3.3.1 Out-Of-Order (OOO) Execution	42
3.3.2 Superscalar Engines	44
3.3.3 Speculative Execution	44
3.3.4 Branch Prediction	46
3.4 SIMD Multiprocessors	47
3.5 Exploiting Thread-Level Parallelism	49
3.5.1 Multicore Systems	50
3.5.2 Simultaneous Multithreading	50
3.5.3 Hybrid Architectures	52
3.6 Memory Hierarchy	53
3.6.1 Cache Hierarchy	54
3.6.2 Main Memory	57
3.7 Virtual Memory	61
3.7.1 Translation Lookaside Buffer (TLB)	63

3.7.2	Huge Pages	63
3.8	Modern CPU Design	64
3.8.1	CPU Frontend	64
3.8.2	CPU Backend	66
3.8.3	Load-Store Unit	68
3.8.4	TLB Hierarchy	71
3.9	Performance Monitoring Unit	72
3.9.1	Performance Monitoring Counters	73
4	Terminology and Metrics in Performance Analysis	77
4.1	Retired vs. Executed Instruction	77
4.2	CPU Utilization	78
4.3	CPI and IPC	78
4.4	Micro-operations	80
4.5	Pipeline Slot	81
4.6	Core vs. Reference Cycles	82
4.7	Cache Miss	83
4.8	Mispredicted Branch	84
4.9	Performance Metrics	85
4.10	Memory Latency and Bandwidth	88
4.11	Case Study: Analyzing Performance Metrics of Four Benchmarks	91
5	Performance Analysis Approaches	98
5.1	Code Instrumentation	99
5.2	Tracing	103
5.3	Collecting Performance Monitoring Events	104
5.3.1	Multiplexing and Scaling Events	105
5.3.2	Using Marker APIs	107
5.4	Sampling	110
5.4.1	User-Mode and Hardware Event-based Sampling	111
5.4.2	Finding Hotspots	111
5.4.3	Collecting Call Stacks	114
5.5	The Roofline Performance Model	115
5.6	Static Performance Analysis	119
5.6.1	Case Study: Using UICA to Optimize FMA Throughput	120
5.7	Compiler Optimization Reports	124
6	CPU Features for Performance Analysis	129
6.1	Top-down Microarchitecture Analysis	130
6.1.1	TMA on Intel Platforms	131
6.1.2	TMA on AMD Platforms	136
6.1.3	TMA On Arm Platforms	138
6.1.4	TMA Summary	141
6.2	Branch Recording Mechanisms	143
6.2.1	LBR on Intel Platforms	144
6.2.2	LBR on AMD Platforms	146
6.2.3	BRBE on Arm Platforms	146
6.2.4	Capture Call Stacks	147
6.2.5	Identify Hot Branches	147

6.2.6	Analyze Branch Misprediction Rate	148
6.2.7	Precise Timing of Machine Code	149
6.2.8	Estimating Branch Outcome Probability	152
6.2.9	Providing Compiler Feedback Data	152
6.3	Hardware-Based Sampling Features	152
6.3.1	PEBS on Intel Platforms	152
6.3.2	IBS on AMD Platforms	154
6.3.3	SPE on Arm Platforms	155
6.3.4	Precise Events	156
6.3.5	Analyzing Memory Accesses	157
7	Overview of Performance Analysis Tools	159
7.1	Intel VTune Profiler	159
7.2	AMD uProf	163
7.3	Apple Xcode Instruments	165
7.4	Linux Perf	168
7.5	Flame Graphs	169
7.6	Event Tracing for Windows	170
7.7	Specialized and Hybrid Profilers	171
7.8	Memory Profiling	177
7.8.1	Memory Usage	178
7.8.2	Case Study: Analyzing Stockfish’s Heap Allocations	179
7.8.3	Memory Intensity and Footprint	182
7.9	Continuous Profiling	184
Part 2.	Source Code Tuning	189
8	Optimizing Memory Accesses	193
8.1	Cache-Friendly Data Structures	194
8.1.1	Access Data Sequentially	194
8.1.2	Use Appropriate Containers	195
8.1.3	Packing the Data	195
8.1.4	Field Reordering	196
8.1.5	Other Data Structure Reorganization Techniques	197
8.2	Dynamic Memory Allocation	198
8.3	Workaround Memory Bandwidth Limitations	199
8.4	Reducing DTLB Misses	200
8.4.1	Explicit Huge Pages	201
8.4.2	Transparent Huge Pages	202
8.4.3	Explicit vs. Transparent Huge Pages	204
8.5	Explicit Memory Prefetching	204
9	Optimizing Computations	210
9.1	Data Dependencies	210
9.2	Inlining Functions	215
9.2.1	Tail Call Optimization	217
9.3	Loop Optimizations	218
9.3.1	Low-level Optimizations	219
9.3.2	High-level Optimizations	220

9.3.3	Discovering Loop Optimization Opportunities	223
9.4	Vectorization	224
9.4.1	Compiler Autovectorization	224
9.4.2	Discovering Vectorization Opportunities	225
9.5	Compiler Intrinsics	231
9.5.1	Wrapper Libraries for Intrinsics	232
10	Optimizing Branch Prediction	236
10.1	Replace Branches with Lookup	237
10.2	Replace Branches with Arithmetic	238
10.3	Replace Branches with Selection	238
10.4	Multiple Tests Single Branch	240
11	Machine Code Layout Optimizations	244
11.1	Machine Code Layout	244
11.2	Basic Block	245
11.3	Basic Block Placement	246
11.4	Basic Block Alignment	247
11.5	Function Splitting	249
11.6	Function Reordering	251
11.7	Profile Guided Optimizations	252
11.8	Reducing ITLB Misses	255
11.9	Case Study: Measuring Code Footprint	256
12	Other Tuning Areas	261
12.1	CPU-Specific Optimizations	261
12.1.1	ISA Extensions	262
12.1.2	CPU Dispatch	263
12.1.3	Instruction Latencies and Throughput	264
12.2	Microarchitecture-Specific Performance Issues	266
12.2.1	Memory Order Violations	266
12.2.2	Misaligned Memory Accesses	268
12.2.3	Cache Aliasing	270
12.2.4	Slow Floating-Point Arithmetic	272
12.3	Low Latency Tuning Techniques	274
12.3.1	Avoid Minor Page Faults	274
12.3.2	Cache Warming	276
12.3.3	Avoid TLB Shootdowns	277
12.3.4	Prevent Unintentional Core Throttling	278
12.4	System Tuning	279
12.5	Case Study: Sensitivity to Last Level Cache Size	280
13	Optimizing Multithreaded Applications	286
13.1	Parallel Efficiency Metrics	287
13.2	Performance Scaling in Multithreaded Programs	288
13.2.1	Thread Count Scaling Case Study	289
13.3	Task Scheduling	300
13.4	Cache Coherence	303
13.4.1	Cache Coherency Protocols	303

13.4.2 True Sharing	305
13.4.3 False Sharing	305
13.5 Advanced Analysis Tools	307
13.5.1 Coz	307
13.5.2 eBPF and GAPP	308
Epilog	310
Acknowledgments	312
Glossary	316
List of the Major CPU Microarchitectures	317
References	319
Appendix A. Reducing Measurement Noise	324
Appendix B. Enable Huge Pages	327
Windows	327
Linux	327
Appendix C. Intel Processor Traces	330
Appendix D. Event Tracing for Windows Analysis	335

1 Introduction

Performance is king: this was true a decade ago, and it certainly is now. According to [[domo.com, 2017](#)], in 2017 the world has been creating 2.5 quintillion¹ bytes of data every day. [[statista.com, 2024](#)] predicts that number to reach 400 quintillion bytes per day in 2024. In our increasingly data-centric world, the growth of information exchange requires both faster software and faster hardware.

Software programmers have had an “easy ride” for decades, thanks to Moore’s law. Software vendors could rely on new generations of hardware to speed up their software products, even if they did not spend human resources on making improvements in their code. This strategy doesn’t work any longer. By looking at Figure 1.1, we can see that single-threaded² performance growth is slowing down. From 1990 to 2000, single-threaded performance on SPECint benchmarks increased by a factor of approximately 25 to 30, driven largely by higher CPU frequencies and improved microarchitecture.

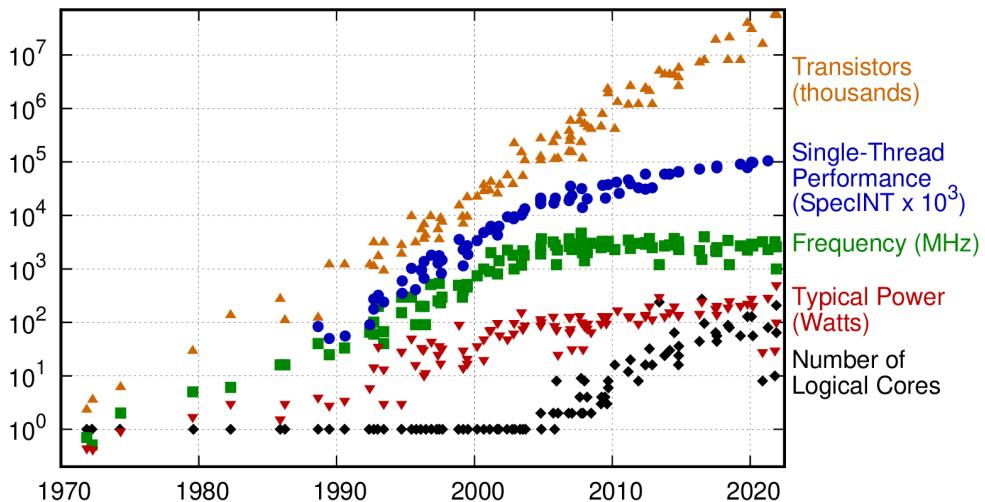


Figure 1.1: 50 Years of Microprocessor Trend Data. © Image by K. Rupp via [karlrupp.net](#). Original data up to the year 2010 was collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten. New plot and data collected for 2010-2021 by K. Rupp.

Single-threaded CPU performance growth was more modest from 2000 to 2010 (a factor between four and five). At that time, clock speeds topped out around 4GHz due to power consumption, heat dissipation challenges, limitations in voltage scaling (Dennard Scaling³), and other fundamental problems. Despite clock speed stagnation,

¹ A quintillion is a thousand raised to the power of six (10^{18}).

² Single-threaded performance is the performance of a single hardware thread inside a CPU core when measured in isolation.

³ Dennard Scaling - https://en.wikipedia.org/wiki/Dennard_scaling

architectural advancements continued: better branch prediction, deeper pipelines, larger caches, and more efficient execution units.

From 2010 to 2020, single-threaded performance grew only by a factor between two and three. During this period, CPU manufacturers began to focus more on multi-core processors and parallelism rather than solely increasing single-threaded performance.

Transistor counts continue to increase in modern processors. For instance, the number of transistors in Apple chips grew from 16 billion in M1 to 20 billion in M2, to 25 billion in M3, to 28 billion in M4 in a span of roughly four years. The growth in transistor count enables manufacturers to add more cores to a processor. As of 2024, you can buy a high-end server processor that will have more than 100 logical cores on a single CPU socket. This is very impressive. Unfortunately, it doesn't always translate into better performance. Very often, application performance doesn't scale with extra CPU cores.

As it's no longer the case that each hardware generation provides a significant performance boost, we must start paying more attention to how fast our code runs. When seeking ways to improve performance, developers should not rely on hardware. 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)

1.1 Why Is Software Slow?

If all the software in the world utilized all available hardware resources efficiently, then this book would not exist. We would not need any changes on the software side and would rely on what existing processors have to offer. But you already know that the reality is different, right? The reality is that modern software is *massively* inefficient. A regular server system in a public cloud typically runs poorly optimized code, consuming more power than it could have consumed (increasing carbon emissions and contributing to other environmental issues). If we could make all software run two times faster, we would potentially reduce the carbon footprint of computing by a factor of two.

The authors of the paper [Leiserson et al., 2020] provide an excellent example that illustrates the performance gap between “default” and highly optimized software. Table 1.1 summarizes speedups from performance engineering a program that multiplies two 4096-by-4096 matrices. The end result of applying several 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. The majority of programs are within rows 1–5. The potential for source-code-level improvements is significant.

Table 1.1: 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

So, let's talk about what prevents systems from achieving optimal performance by default. Here are some of the most important factors:

1. **CPU limitations:** It's so tempting to ask: "*Why doesn't hardware solve all our problems?*" Modern CPUs execute instructions incredibly quickly, 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 bubble sort, a CPU will not make any attempts to recognize it and use better alternatives (e.g. quicksort). It will blindly execute whatever it was told to do.
2. **Compiler limitations:** "*But isn't that 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 vectorization, they may not generate the best possible code. Performance experts often can come up with clever ways to vectorize loops beyond the capabilities of compilers. When compilers have to make a decision whether to perform a code transformation or not, they rely on complex cost models and heuristics, which may not work for every possible scenario. 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. Additionally, compilers cannot perform optimizations unless they are absolutely certain it is safe to do so. It may be very difficult for a compiler to prove that an optimization is correct under all possible circumstances, disallowing some transformations. Finally, compilers generally do not attempt "heroic" optimizations, like transforming data structures used by a program.
3. **Algorithmic complexity analysis limitations:** Some developers are overly obsessed with algorithmic complexity analysis, which leads them to choose a popular algorithm with the optimal algorithmic complexity, even though it may not be the most efficient for a given problem. Considering two sorting algorithms, insertion sort and quicksort, the latter clearly wins in terms of Big O notation for the average case: insertion sort is $O(N^2)$ while quickSort is only

$O(N \log N)$. Yet for relatively small sizes of N (up to 50 elements), insertion sort outperforms quickSort. Complexity analysis cannot account for all the low-level performance effects of various algorithms, so people just encapsulate them in an implicit constant C , which sometimes can make a large impact on performance. Only counting comparisons and swaps that are used for sorting, ignores cache misses and branch mispredictions, which, today, are actually very costly. 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.

In addition to the limitations described above, there are overheads created by programming paradigms. Coding practices that prioritize code clarity, readability, and maintainability can reduce performance. Highly generalized and reusable code can introduce unnecessary copies, runtime checks, function calls, memory allocations, etc. For instance, polymorphism in object-oriented programming is usually implemented using virtual functions, which introduce a performance overhead.⁴

All the factors mentioned above assess a “performance tax” on the software. There are very often substantial opportunities for tuning the performance of our software to reach its full potential.

1.2 Why Care about Performance?

In addition to the slowing growth of hardware single-threaded performance, there are a couple of other business reasons to care about performance. During the PC era,⁵ the costs of slow software were paid by the users, as inefficient software was running on user computers. Software vendors were not directly incentivized to optimize the code of their applications. With the advent of SaaS (software as a service) and cloud computing, the costs of slow software are put back on the software providers, not their users. If you’re a SaaS company like Meta or Netflix,⁶ it doesn’t matter if you run your service on-premise hardware or you use the public cloud, you pay for the electricity your servers consume. Inefficient software cuts right into your margins and market valuation. According to Synergy Research Group,⁷ worldwide spending on cloud services topped \$100 billion in 2020, and according to Gartner,⁸ it will surpass \$675 billion in 2024.

For many years performance engineering was a nerdy niche, but now it’s becoming mainstream. Many companies have already realized the importance of performance engineering and are willing to pay well for this work.

It is fairly easy to reach performance level 4 in Table 1.1. In fact, you don’t need this

⁴ I do not dismiss design patterns and clean code principles, but I encourage a more nuanced approach where performance is also a key consideration in the development process.

⁵ The late 1990s and early 2000s, a time when personal computers dominated the market of computing devices.

⁶ In 2024, Meta uses mostly on-premise cloud, while Netflix uses AWS public cloud.

⁷ Worldwide spending on cloud services in 2020 - <https://www.srgresearch.com/articles/2020-the-year-that-cloud-service-revenues-finally-dwarfed-enterprise-spending-on-data-centers>

⁸ Worldwide spending on cloud services in 2024 - <https://www.gartner.com/en/newsroom/press-releases/2024-05-20-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-surpass-675-billion-in-2024>

book to get there. Write your program in one of the native programming languages, distribute work among multiple threads, pick a good optimizing compiler and you'll get there. Unfortunately, the performance of your program will be about 200 times slower than the optimal target.

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 1.1. 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. SQLite is commonplace today not because its developers one day made it 50% faster, but because they meticulously made hundreds of 0.1% improvements over the years. The cumulative effect of these small improvements is what makes the difference.

The impact of small improvements is very relevant for large distributed applications running in the cloud. According to [Hennessy, 2018], in the year 2018, Google spent roughly the same amount of money on actual computing servers that run the cloud as it spent on power and cooling infrastructure. Energy efficiency is a very important problem, which can be improved by optimizing software.

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

In addition to cloud costs, there is another factor at play: how people perceive slow software. Google reported that a 500-millisecond delay in search caused a 20% reduction in traffic.⁹ For Yahoo! 400 milliseconds faster page load caused 5-9% more traffic.¹⁰ In the game of big numbers, small improvements can make a significant impact. Such examples prove that the slower a service works, the fewer people will use it.

Outside cloud services, there are many other performance-critical industries where performance engineering does not need to be justified, such as Artificial Intelligence (AI), High-Performance Computing (HPC), High-Frequency Trading (HFT), game development, etc. Moreover, performance is not only required in highly specialized areas, it is also relevant for 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++ IntelliSense¹¹ features that are integrated into Microsoft Visual Studio IDE have very tight performance constraints. For the IntelliSense autocomplete feature to work, it must parse the entire source codebase in milliseconds.¹² Nobody will use a source code editor if it takes several seconds to suggest autocomplete options. Such a feature has to be very responsive and provide valid continuations as the user types new code.

⁹ Google I/O '08 Keynote by Marissa Mayer - <https://www.youtube.com/watch?v=6x0cAzQ7PVs>

¹⁰ Slides by Stoyan Stefanov - <https://www.slideshare.net/stoyan/dont-make-me-wait-or-building-highperformance-web-applications>

¹¹ Visual C++ IntelliSense - <https://docs.microsoft.com/en-us/visualstudio/ide/visual-cpp-intellisense>

¹² 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 have 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>

“Not all fast software is world-class, but all world-class software is fast. Performance is *the killer feature.*” —Tobi Lutke, CEO of Shopify.

I hope it goes without saying that people hate using slow software, especially when their productivity goes down because of it. Table 1.2 shows that most people consider a delay of 2 seconds or more to be a “long wait,” and would switch to something else after 10 seconds of waiting (I think much sooner). If you want to keep users’ attention, your application must react quickly.

Table 1.2: Human-software interaction classes. *Source: Microsoft Windows Blogs.*¹³

Interaction Class	Human Perception	Response Time
Fast	Minimally noticeable delay	100ms–200ms
Interactive	Quick, but too slow to be described as Fast	300ms–500ms
Pause	Not quick but still feels responsive	500ms–1 sec
Wait	Not quick due to amount of work for scenario	1 sec–3 sec
Long Wait	No longer feels responsive	2 sec–5 sec
Captive	Reserved for unavoidably long/complex scenarios	5 sec–10 sec
Long-running	User will probably switch away during operation	10 sec–30 sec

Application performance can drive your customers to a competitor’s product. By emphasizing performance, you can give your product a competitive advantage.

Sometimes fast tools find applications for which they were not initially designed. For example, game engines like Unreal and Unity are used in architecture, 3D visualization, filmmaking, and other areas. Because game engines are so performant, they are a natural choice for applications that require 2D and 3D rendering, physics simulation, 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 his blog.¹⁴

Before starting performance-related work, 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.¹⁵ Mindful performance engineering starts with clearly defined performance goals. Understand clearly what you are trying to achieve, and justify the work. Establish metrics that you will use to measure success.

Now that we’ve talked about the value of performance engineering, let’s uncover what it consists of. When you’re trying to improve the performance of a program, you need to find problems (performance analysis) and then improve them (tuning), a task very similar to a regular debugging activity. This is what we will discuss next.

¹³ Microsoft Windows Blogs - <https://blogs.windows.com/windowsdeveloper/2023/05/26/delivering-delightful-performance-for-more-than-one-billion-users-worldwide/>

¹⁴ Reflections on software performance by N. Elhage - <https://blog.nelhage.com/post/reflections-on-performance/>

¹⁵ Unless you just want to practice performance optimizations, which is fine too.

1.3 What Is Performance Analysis?

Have you 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 small tweaks to code can trigger noticeable performance changes. Relying on intuition when optimizing an application typically results in random “fixes” without real performance impact.

Inexperienced developers sometimes make changes in their code and claim it *should* run faster. One such example is replacing `i++` (post-increment) with `++i` (pre-increment) 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: every decent optimizing compiler will recognize that the previous value of `i` is not used and will eliminate redundant copies anyway. The first piece of advice in this book is: don’t solely rely on your intuition. *Always measure.*

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 such example is the XOR swap idiom.¹⁶ In reality, simple `std::swap` produces equivalent or faster code. Such accidental changes likely won’t improve the performance of an application. Finding the right place to tune should be the result of careful performance analysis, not intuition or guessing.

Performance analysis is a process of collecting information about how a program executes and interpreting it to find optimization opportunities. Any change that ends up being made in the source code of a program should be driven by analyzing and interpreting collected data. We will show you how to use performance analysis techniques to discover optimization opportunities even in a large and unfamiliar codebase. There are many performance analysis methodologies. Depending on the problem, some will be more efficient than others. With experience, you will develop your own strategies about when to use each approach.

1.4 What Is Performance Tuning?

Locating a performance bottleneck is only half of an engineer’s job. The second half is to fix it properly. Sometimes changing one line in the source code of a program can yield a drastic performance boost. Missing such opportunities can be quite wasteful. Performance analysis and tuning are all about finding and fixing this line.

To take advantage of all the computing power of modern CPUs, you need to understand how they work. Or as performance engineers like to say, you need to have “mechanical sympathy”. This term was borrowed from the car racing world. It means that a racing driver with a good understanding of how the car works has an edge over its competitors who don’t. The same applies to performance engineering. It is not possible to know all the details of how a modern CPU operates, but you need to have a good mental model of it to squeeze the last bit of performance.

This is what I mean by *low-level optimizations*. This is a type of optimization that takes into account the details of the underlying hardware capabilities. It is different

¹⁶ XOR-based swap idiom - https://en.wikipedia.org/wiki/XOR_swap_algorithm

from *high-level optimizations* which are more about application-level logic, algorithms, and data structures. As you will see in the book, the majority of low-level optimizations can be applied to a wide variety of modern processors. To successfully implement low-level optimizations, you need to have a good understanding of the underlying hardware.

“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]

In the past, software developers had more mechanical sympathy, as they often had to deal with nuances of the hardware implementation. During the PC era, 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 software stack grew deeper, broader, and more complex. The top layer of the stack (on which most developers work) has moved further away from the hardware. The negative side of such evolution is that developers of modern applications have less affinity for the actual hardware on which their software is running. This book will help you build a strong connection with modern processors.

There is a famous quote by Donald Knuth: “Premature optimization is the root of all evil”.[Knuth, 1974] But the opposite is often true as well. Postponed performance engineering 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 hardware works. In such roles, program development without a hardware focus is a failure from the beginning.¹⁷ Performance characteristics of software must be a primary objective alongside correctness and security from day one. Poor performance can kill a product just as easily as security vulnerabilities.

Performance engineering is important and rewarding work, but it may be very time-consuming. In fact, performance optimization is a game with no end. There will always be something to optimize. Inevitably, a developer will reach the point of diminishing returns at which further improvement is not justified by expected engineering costs. Knowing when to stop optimizing is a critical aspect of performance work.

1.5 What Is Discussed in this Book?

This book is written to help developers better understand the performance of their applications, learn to find inefficiencies, and eliminate them.

- Why did my change cause a 2x performance drop?
- Our customers complain about the slowness of our application. How should I investigate it?
- Why does my handwritten compression algorithm perform slower than the conventional one?
- Have I optimized my program to its full potential?
- What performance analysis tools are available on my platform?
- What are techniques to reduce the number of cache misses and branch mispredictions?

¹⁷ ClickHouse DB is an example of a successful software product that was built around a small but very efficient core.

I hope that by the end of this book, you will be able to answer those questions.

The book is split into two parts. The first part (chapters 2–7) teaches you how to find performance problems, and the second part (chapters 8–13) teaches you how to fix them.

- Chapter 2 discusses fair performance experiments and their analysis. It introduces the best practices for performance testing and comparing results.
- Chapter 3 introduces CPU microarchitecture, with a close look at Intel’s Golden Cove microarchitecture.
- Chapter 4 covers terminology and metrics used in performance analysis. At the end of the chapter, we present a case study that features various performance metrics collected on four real-world applications.
- Chapter 5 explores the most popular performance analysis approaches. We describe how profiling tools work and what sort of data they can collect.
- Chapter 6 examines features provided by modern Intel, AMD, and ARM-based CPUs to support and enhance performance analysis. It shows how they work and what problems they help to solve.
- Chapter 7 gives an overview of the most popular performance analysis tools available on Linux, Windows, and MacOS.
- Chapter 8 is about optimizing memory accesses, cache-friendly code, data structure reorganization, and other techniques.
- Chapter 9 is about optimizing computations; it explores data dependencies, function inlining, loop optimizations, and vectorization.
- Chapter 10 is about branchless programming, which is used to avoid branch misprediction.
- Chapter 11 is about machine code layout optimizations, such as basic block placement, function splitting, and profile-guided optimizations.
- Chapter 12 contains optimization topics not considered in the previous four chapters, but still important enough to find their place in this book. In this chapter, we discuss CPU-specific optimizations, examine several microarchitecture-related performance problems, explore techniques used for optimizing low-latency applications, and give you advice on tuning your system for the best performance.
- Chapter 13 discusses techniques for analyzing multithreaded applications. It digs into some of the most important challenges of optimizing multithreaded applications. We provide a case study of five real-world multithreaded applications, where we explain why their performance doesn’t scale with the number of CPU threads. We also discuss cache coherency issues (e.g., “false sharing”) and a few tools that are designed to analyze multithreaded applications.

At the end of the book, there is a glossary and a list of microarchitectures for major CPU vendors. Whenever you see an unfamiliar acronym or you need to refresh your memory on recent Intel, AMD, and ARM chip families, refer to these resources.

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 various open-source applications and benchmarks¹⁸ that you can build and run. The reason is not only the popularity of these projects but also the fact that their source code is open, which enables us to better understand the underlying mechanism of how they work. This is especially useful for learning the concepts presented in this book. This doesn't mean that we will never showcase proprietary tools. For example, we extensively use Intel® VTune™ Profiler.

Sometimes it's possible to obtain attractive speedups by forcing the compiler to generate desired machine code through various hints. You will find many such examples throughout the book. While prior compiler experience helps a lot in performance work, most of the time you don't have to be a compiler expert to drive performance improvements in your application. The majority of optimizations can be done at a source code level without the need to dig down into compiler sources.

1.6 What Is Not Discussed in this Book?

System performance depends on different components: CPU, DRAM, I/O, and network devices, etc. Applications may benefit from tuning various components of the system, depending on where a bottleneck is. In general, engineers should analyze the performance of the whole system. However, the biggest factor in a system's performance is its heart, the CPU. This is why this book primarily focuses on performance analysis from a CPU perspective. We also discuss the memory subsystem quite extensively, but we don't explore I/O and network performance.

Likewise, the software stack includes many layers, e.g., firmware, BIOS, OS, libraries, and the source code of an application. However, since most of the lower layers are not under our direct control, the primary focus will be on the source code level.

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.

I tried to make this book to be applicable to most modern CPUs, including Intel, AMD, Apple, and other ARM-based processors. I'm sorry if it doesn't cover your favorite architecture. Nevertheless, many of the principles discussed in this book apply well to other processors. Similarly, most examples in this book were run on Linux, but again, most of the time it doesn't matter since the same techniques benefit applications that run on Windows and macOS operating systems.

Code snippets in this book are written in C or C++, 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, I assume that readers have full control over the software that they develop, including the choice of libraries and compilers they use. Hence, this book is not about tuning purchased commercial packages, e.g., tuning SQL database queries.

¹⁸ Some people don't like when their application is called a "benchmark". They think that a benchmark is something that is synthesized and contrived, and does a poor job of representing real-world scenarios. In this book, we use the terms "benchmark", "workload", and "application" interchangeably and don't mean to offend anyone.

1.7 Exercises

As supplemental material for this book, I developed “Performance Ninja”, a free online course where you can practice low-level performance analysis and tuning. It is available at the following URL: <https://github.com/dendibakh/perf-ninja>. It has a collection of lab assignments that focus on a specific performance problem. Each lab assignment can take anywhere from 30 minutes up to 4 hours depending on your background and the complexity of the lab assignment itself.

Following the name of the GitHub repository, we will use `perf-ninja` to refer to the online course. In the “Questions and Exercises” section at the end of each chapter, you may find assignments from `perf-ninja`. For example, when you see `perf-ninja::warmup`, this corresponds to the lab assignment with the name “Warmup” in the GitHub repository. We encourage you to solve these puzzles to solidify your knowledge.

You can solve assignments on your local machine or submit your code changes to GitHub for automated verification and benchmarking. If you choose the latter, follow the instructions on the “Get Started” page of the repository. We also use examples from `perf-ninja` throughout the book. This enables you to reproduce a specific performance problem on your own machine and experiment with it.

Chapter Summary

- Single-threaded CPU performance is not increasing as rapidly as it used to a few decades ago. When it’s no longer the case that each hardware generation provides a significant performance boost, developers should start optimizing the code of their software.
- Modern software is massively inefficient. A regular server system in a public cloud, typically runs poorly optimized code, consuming more power than it could have consumed, which increases carbon emissions and contributes to other environmental issues.
- Certain limitations exist that prevent applications from reaching their full performance potential. CPUs cannot magically speed up slow algorithms. Compilers are far from generating optimal code for every program. Big O notation is not always a good indicator of performance as it doesn’t account for hardware specifics.
- For many years performance engineering was a nerdy niche. But now it’s becoming mainstream as software vendors realize the impact that their poorly optimized software has on their bottom line.
- People absolutely hate using slow software, especially when their productivity goes down because of it. Not all fast software is world-class, but all world-class software is fast. Performance is *the* killer feature.
- Software tuning is becoming more important than it has been for the last 40 years and it will be one of the key drivers for performance gains in the near future. The importance of low-level performance tuning should not be underestimated, even if it’s just a 1% improvement. The cumulative effect of these small improvements is what makes the difference.
- To squeeze the last bit of performance you need to have a good mental model of how modern CPUs work.

- 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 software optimizations, developers should not rely on intuition but use careful performance analysis instead.

Part 1. Performance Analysis on a Modern CPU

2 Measuring Performance

The first step to understanding the performance of an application is to measure it. Anyone ever concerned with performance evaluations likely knows how hard it is sometimes to conduct fair performance measurements and draw accurate conclusions from them. 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 the performance of the program. For various reasons, measurements may consistently overestimate or underestimate the true performance, which leads to distorted results that do not accurately reflect reality. This phenomenon is called *measurement bias*.

Performance problems are often harder to reproduce and root cause than most functional issues. Every run of a program is usually functionally the same but somewhat different from a 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 the same CPU cycle-by-cycle performance profile of this operation.

Conducting fair performance experiments is an essential step towards getting accurate and meaningful results. You need to ensure you're looking at the right problem and are not debugging some unrelated issue. Designing performance tests and configuring the environment are both important components in the process of evaluating performance.

Because of the measurement bias, performance evaluations often involve statistical methods, which deserve a whole book just for themselves. There are many corner cases and a huge amount of research done in this field. We will not dive into statistical methods for evaluating performance measurements. Instead, we only discuss high-level ideas and give basic directions to follow. We encourage you to research deeper on your own.

In this chapter, we:

- Give a brief introduction to why modern systems yield noisy performance measurements and what you can do about it.
- Explain why it is important to measure performance in production deployments.
- Provide general guidance on how to properly collect and analyze performance measurements.
- Explore how to automatically detect performance regressions as you implement changes in your codebase.
- Describe software and hardware timers that can be used by developers in time-based measurements.

- Discuss how to write a good microbenchmark and some common pitfalls you may encounter while doing it.

2.1 Noise in Modern Systems

There are many features in hardware and software that are designed to increase performance, but not all of them have deterministic behavior. Consider Dynamic Frequency Scaling (DFS), a feature that allows a CPU to increase its frequency far above the base frequency, allowing it to run significantly faster. DFS is also frequently referred to as *turbo mode*. Unfortunately, a CPU cannot stay in the turbo mode for a long time, otherwise it may face the risk of overheating. So after some time, it decreases its frequency to stay within its thermal limits. DFS usually depends a lot on the current system load and external factors, such as core temperature, which makes it hard to predict the impact on performance measurements.

Figure 2.1 shows a typical example where DFS can cause variance in performance. In our scenario, we started two runs of a benchmark, one right after another on a “cold” processor.¹⁹ During the first second, the first iteration of the benchmark was running on the maximum turbo frequency of 4.4 GHz but later the CPU had to decrease its frequency below 4 GHz. The second run did not have the advantage of boosting the CPU frequency and did not enter the turbo mode. Even though we ran the exact same version of the benchmark two times, the environment in which they ran was not the same. As you can see, the first run is 200 milliseconds faster than the second run due to the fact that it was running with a higher CPU frequency in the beginning. Such a scenario can frequently happen when you benchmark software on a laptop since laptops have limited heat dissipation.²⁰

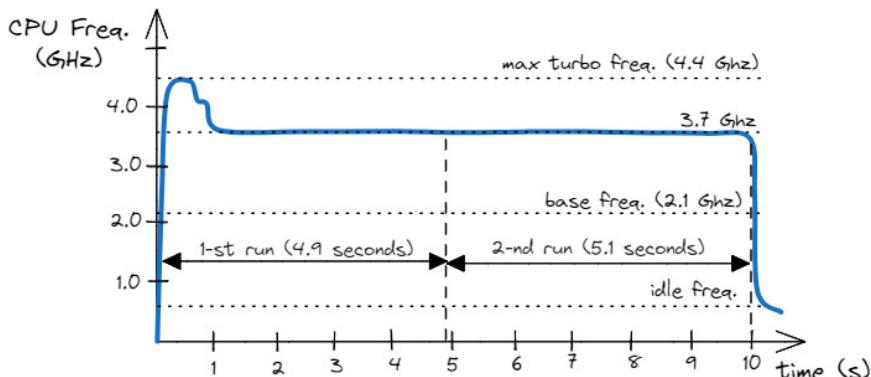


Figure 2.1: Variance in performance caused by dynamic frequency scaling: the first run is 200 milliseconds faster than the second.

Frequency Scaling is an example of how a hardware feature can cause variations in our

¹⁹ By cold processor, we mean the CPU that stayed in idle mode for a while, allowing it to cool down its temperature.

²⁰ Remember that even running Windows task manager or Linux top programs, can affect measurements since an additional CPU core will be activated and assigned to it. This might affect the frequency of the core that is running the actual benchmark.

measurements, however, they could also come from software. Consider benchmarking a `git status` command, which accesses many files on the disk. The filesystem plays a big role in performance in this scenario; in particular, the filesystem cache. On the first run, the required entries in the filesystem cache are missing. The filesystem cache is not effective and our `git status` command runs very slowly. However, the second time, the filesystem cache will be warmed up, making it much faster than the first run.

You’re probably thinking about including a dry run before taking measurements. That certainly helps, unfortunately, measurement bias can persist through the runs as well. [Mytkowicz et al., 2009] paper demonstrates that UNIX environment size (i.e., the total number of bytes required to store the environment variables) or the link order (the order of object files that are given to the linker) can affect performance in unpredictable ways. There are numerous other ways how memory layout may affect performance measurements.²¹

Having consistent performance requires running all iterations of the benchmark with the same conditions. It is impossible to achieve 100% consistent results on every run of a benchmark, but perhaps you can get close by carefully controlling the environment. Eliminating nondeterminism in a system is helpful for well-defined, stable performance tests, e.g., microbenchmarks.

Consider a situation when you implemented a code change and want to know the relative speedup ratio by benchmarking the “before” and “after” versions of the program. This is a scenario in which you can control most of the variability in a system, including HW configuration, OS settings, background processes, etc. Disabling features with nondeterministic performance impact will help you get a more consistent and accurate comparison. You can find examples of such features 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`²².

However, it is not possible to replicate the exact same environment and eliminate bias completely: there could be different temperature conditions, power delivery spikes, unexpected system interrupts, etc. Chasing all potential sources of noise and variation in a system can be a never-ending story. Sometimes it cannot be achieved, for example, when you’re benchmarking a large distributed cloud service.

You should not eliminate system nondeterministic behavior when you want to measure the real-world performance impact of your change. Yes, these features may contribute to performance instabilities, but they are designed to improve the overall performance of the system. Users of your application are likely to have them enabled to provide better performance. So, when you analyze the performance of a production application, you should try to replicate the target system configuration, which you are optimizing for. Introducing any artificial tuning to the system will change results from what users

²¹ One approach to enable statistically sound performance analysis was presented in [Curtsinger & Berger, 2013]. This work showed that it’s possible to eliminate measurement bias that comes from memory layout by 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.

²² Temci - <https://github.com/parttimenerd/temci>.

of your service will see in practice.²³

2.2 Measuring Performance in Production

When an application runs in a shared infrastructure, e.g., in a public cloud, there usually will be 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. When your application shares resources with neighbor processes, its performance can become very unpredictable.

Analyzing production workloads by recreating a specific scenario in a lab can be quite tricky. Sometimes it's not possible to reproduce exact behavior for “in-house” performance testing. This is why cloud providers and hyperscalers provide tools to monitor performance directly on production systems. A code change that performs well in a lab environment does not necessarily always perform well in production. Consult with your cloud service provider to see how you can enable performance monitoring of production instances. We provide an overview of continuous profilers in Section 7.9.

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 Icarus²⁴ telemetry service, which runs on thousands of different devices spread around the world. Such a telemetry system helps Netflix understand how users perceive Netflix's app performance. It enables Netflix engineers to analyze data collected from many devices and to find issues that otherwise would be impossible to find. This kind of data enables making better-informed decisions on where to focus optimization efforts.

One important caveat of monitoring production deployments is measurement overhead. Because any kind of monitoring affects the performance of a running service, we recommended using 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 capturing data samples less frequently.

2.3 Continuous Benchmarking

We just discussed why you should monitor performance in production. On the other hand, it is still beneficial to set up continuous “in-house” testing to catch performance problems early, even though not every performance regression can be caught in a lab.

Software vendors constantly seek ways to accelerate the pace of delivering their products to the market. Many companies deploy newly written code every couple of months or weeks. Unfortunately, software products don't get better performance with

²³ Another downside of disabling nondeterministic performance features is that it makes a benchmark run longer. This is especially important for Continuous Integration and Continuous Delivery (CI/CD) performance testing when there are time limits for how long it should take to run the whole benchmark suite.

²⁴ Presented at CMG 2019, https://www.youtube.com/watch?v=4RG2DUK03_0.

each new release. Performance defects tend to leak into production software at an alarming rate [Jin et al., 2012]. A large number of code changes pose a challenge to thorough analysis of their performance impact.

Performance regressions are defects that make the software run slower compared to the previous version. Catching performance regressions (or improvements) requires detecting the commit that has changed the performance of the program. 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. It may be impossible to entirely avoid performance regressions during software development, but with proper testing and diagnostic tools, the likelihood of such defects silently leaking into production code can be reduced significantly.

It is useful to track the performance of your application with charts, like the one shown in Figure 2.2. Using such a chart you can see historical trends and find moments where performance improved or degraded. Typically, you will have a separate line for each performance test you’re tracking. Do not include too many benchmarks on a single chart as it will become very noisy.

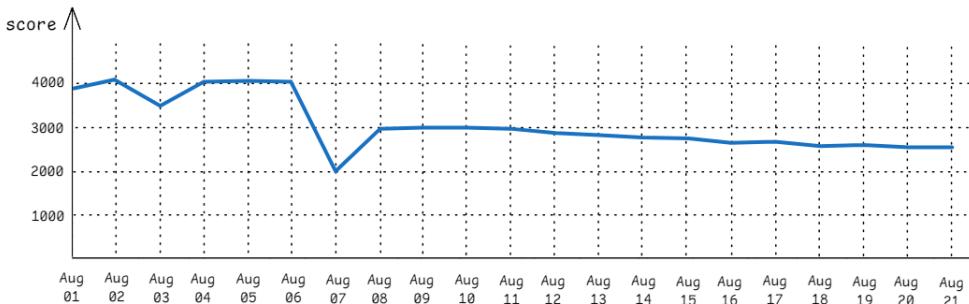


Figure 2.2: Performance graph (higher better) for an application showing a big drop in performance on August 7th and smaller ones later.

Let’s consider some potential solutions for detecting performance regressions. The first option that comes to mind is: having humans look at the graphs. For the chart in Figure 2.2, humans will likely catch performance regression that happened on August 7th, but it’s not obvious that they will detect later smaller regressions. People tend to lose focus quickly and can miss regressions, especially on a busy chart. In addition to that, it is a time-consuming and boring job that must be performed daily.

There is another interesting performance drop on August 3rd. A developer will also likely catch it, however, most of us would be tempted to dismiss it since performance recovered the next day. But are we sure that it was merely a glitch in measurements? What if this was a real regression that was compensated by an optimization on August 4th? If we could fix the regression *and* keep the optimization, we would have a performance score of around 4500. Do not dismiss such cases. One way to proceed here would be to repeat the measurements for the dates Aug 02–Aug 04 and inspect code changes during that period.

The second option is to have a threshold, say, 2%. Every code modification that

has performance within that 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 can trigger performance variation in a benchmark.²⁵ 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 regressions can pile up slowly into a bigger regression, which can be left unnoticed. Going back to Figure 2.2, notice a downward trend that lasted from Aug 11 to Aug 21. The period started with a score of 3000 and ended with 2600. That is roughly a 15% regression over 10 days or 1.5% per day on average. If we set a 2% threshold all regressions will be filtered out. However, as we can see, the accumulated regression is much bigger than the threshold.

Nevertheless, this option works reasonably well for many projects, especially if the level of noise in a benchmark is very low. Also, you can adjust the threshold for each test. An example of a Continuous Integration (CI) system where each test requires setting explicit threshold values for alerting a regression is LUCI,²⁶ which is a part of the Chromium project.

It's worth mentioning that tracking performance results over time requires that you maintain the same configuration of the machine(s) that you use to run benchmarks. A change in the configuration may invalidate all the previous performance results. You may decide to recollect all historical measurements with a new configuration, but this is very expensive.

Another option that recently became popular uses a statistical approach to identify performance regressions. It leverages an algorithm called “Change Point Detection” (CPD, see [Matteson & James, 2014]), which utilizes historical data and identifies points in time where performance has changed. Many performance monitoring systems embraced the CPD algorithm, including several open-source projects. You can search the web to find the one that better suits your needs.

The notable advantage of CPD is that it does not require setting thresholds. The algorithm evaluates a large window of recent results, which allows it to ignore outliers as noise and produce fewer false positives. The downside for CPD is the lack of immediate feedback. For example, consider a performance test with the following historical measurements of running time: 5 sec, 6 sec, 5 sec, 5 sec, 7 sec. If the next benchmark result comes at 11 seconds, then the threshold would likely be exceeded and an alert would be generated immediately. However, in the case of using the CPD algorithm, it wouldn't do anything at this point. If in the next run, performance is restored to 5 seconds, then it would likely dismiss it as a false positive and not generate an alert. Conversely, if the next run or two resulted in 10 sec and 12 sec respectively, only then would the CPD algorithm trigger an alert.

²⁵ 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

²⁶ LUCI - https://chromium.googlesource.com/chromium/src.git/+/master/docs/tour_of_luci_ui.md

There is no clear answer to which approach is better. If your development flow requires immediate feedback, e.g., evaluating a pull request before it gets merged, then using thresholds is a better choice. Also, if you can remove a lot of noise from your system and achieve stable performance results, then using thresholds is more appropriate. In a very quiet system, the 11 second measurement mentioned before likely indicates a real performance regression, thus we need to flag it as early as possible. In contrast, if you have a lot of noise in your system, e.g., you run distributed macro-benchmarks, then that 11 second result may just be a false positive. In this case, you may be better off using Change Point Detection.

A typical CI performance tracking system should automate the following actions:

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

Another desirable feature of a CI performance tracking system is to allow developers to submit performance evaluation jobs for their patches before they commit them to the codebase. This greatly simplifies the developer's job and facilitates quicker turnaround of experiments. The performance impact of a code change is frequently included in the list of check-in criteria.

If, for some reason, a performance regression has slipped into the codebase, it is very important to detect it promptly. First, because fewer changes were merged since it happened. This allows us to have a person responsible for the regression 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 improves performance by 10% in the automated tracking harness. Your initial instinct may be to celebrate this fortuitous performance boost and proceed with your day. However, while this commit may have passed all functional tests in your CI pipeline, chances are that this unexpected improvement uncovered a gap in functional testing which only manifested itself in performance regression results. For instance, the change caused the application to skip some parts of work, which was not covered by functional tests. This scenario occurs often enough that it warrants explicit mention: treat the automated performance regression harness as part of a holistic software testing framework.

To wrap it up, we 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

In the previous section, we discussed how CI systems can help with evaluating the performance impact of a code change. However, it may not always be possible to leverage such a system due to reasons such as hardware unavailability, setup being too complicated for the testing infrastructure, a need to collect additional metrics, etc. In this section, we provide basic advice for local performance evaluations.

We typically measure the performance impact of our code change by 1) measuring the baseline performance, 2) measuring the performance of the modified program, and 3) comparing them with each other. For example, we had a program that calculates Fibonacci numbers recursively (baseline), and we decided to rewrite it with a loop (modified). Both versions are functionally correct and yield the same Fibonacci numbers. Now we need to compare the performance of the two versions of the program.

It is highly recommended to get not just a single measurement but to run the benchmark multiple times. If you make comparisons based on a single measurement, you're increasing the risk of having your numbers skewed by the measurement bias that we discussed in Section 2.1. So, we collected N performance measurements for the baseline and N measurements for the modified version of the program. We call a set of performance measurements a *performance distribution*. Now we need to aggregate and compare those two distributions to decide which version of the program is faster.

The most straightforward way to compare two performance distributions is to take the average of N measurements from both distributions and calculate the ratio. For the types of code improvements we discuss in this book, this simple method works well in most cases. However, comparing performance distributions is quite nuanced, and there are many ways how you can be fooled by measurements and potentially derive wrong conclusions. We will not get into the details of statistical analysis, instead, we recommend you read a textbook on the subject. A good reference specifically for performance engineers is a book by Dror G. Feitelson, “Workload Modeling for Computer Systems Performance Evaluation”,²⁷ that has more information on modal distributions, skewness, and other related topics.

Data scientists often present measurements by plotting them. This eliminates biased conclusions and allows readers to interpret the data for themselves. One of the popular ways to plot distributions is by using box plots (also known as a box-and-whisker plot). In Figure 2.3, we visualized performance distributions of two versions of the same functional program (“before” and “after”). There are 70 performance data points in each distribution.

Let's describe the terms indicated on the image:

- The *mean* (often referred to as the *average*) is the sum of all values in a dataset divided by the number of values. Indicated with X.
- The *median* is the middle value of a dataset when the values are sorted. The same as *50th percentile* (p50).
- The *25th percentile* (p25) divides the lowest 25% of the data from the highest 75%.

²⁷ Book “Workload Modeling for Computer Systems Performance Evaluation” - <https://www.cs.huji.ac.il/~feit/wlmod/>

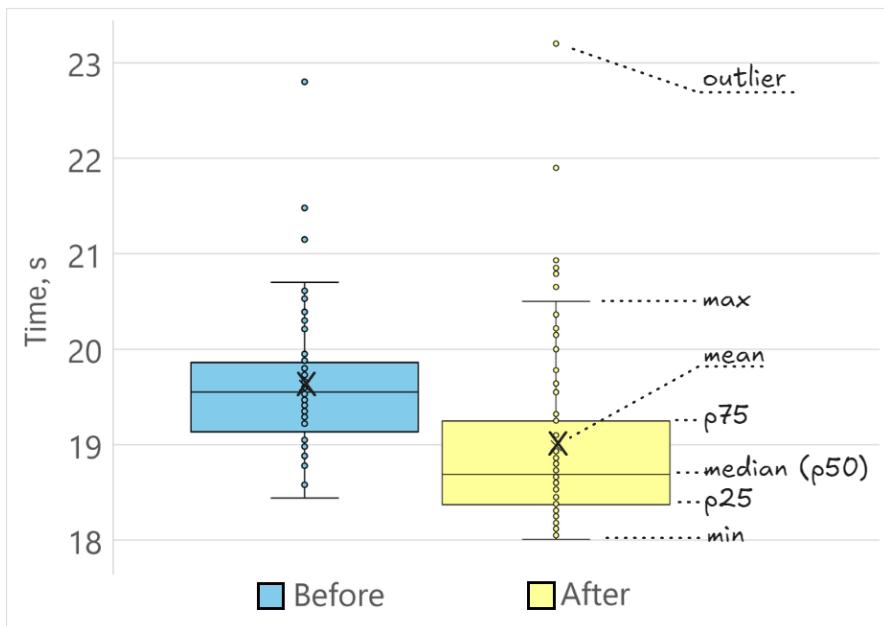


Figure 2.3: Performance measurements (lower is better) of “Before” and “After” versions of a program presented as box plots.

- The *75th percentile* (p75) divides the lowest 75% of the data from the highest 25%.
- An *outlier* is a data point that differs significantly from other samples in the dataset. Outliers can be caused by variability in the data or experimental errors.
- The *min* and *max* (whiskers) represent the most extreme data points that are not considered outliers.

By looking at the box plot in Figure 2.3, we can sense that our code change has a positive impact on performance since “after” samples are generally faster than “before”. However, there are some “before” measurements that are faster than “after”. Box plots allow comparisons of multiple distributions on the same chart. The benefits of using box plots for visualizing performance distributions are described in a blog post by Stefan Marr.²⁸

Performance speedups can be calculated by taking a ratio between the two means. In some cases, you can use other metrics to calculate speedups, including median, min, and 95th percentile, depending on which one is more representative of your distribution.

Standard deviation quantifies how much the values in a dataset deviate from the mean on average. A low standard deviation indicates that the data points are close to the mean, while a high standard deviation indicates that they are spread out over a wider range. Unless distributions have low standard deviation, do not calculate speedups. If

²⁸ Stefan Marr’s blog post about box plots - <https://stefan-marr.de/2024/06/5-reasons-for-box-plots-as-default/>

the standard deviation in the measurements is on the same order of magnitude as the mean, the average is not a representative metric. Consider taking steps to reduce noise in your measurements. If that is not possible, present your results as a combination of the key metrics such as mean, median, standard deviation, percentiles, min, max, etc.

Performance gains are usually represented in two ways: as a speedup factor or percentage improvement. If a program originally took 10 seconds to run, and you optimized it down to 1 second, that's a 10x speedup. We shaved off 9 seconds of running time from the original program, that's a 90% reduction in time. The formula to calculate percentage improvement is shown below. In the book, we will use both ways of representing speedups.

$$\text{Percentage Speedup} = \left(1 - \frac{\text{New Time}}{\text{Old Time}}\right) \times 100\%$$

One of the most important factors in calculating accurate speedup ratios is collecting a rich collection of samples, i.e., running a 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**²⁹ 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.

If obtaining new measurements is expensive, don't rush to collect many samples. Often you can learn a lot from just three runs. If you see a very low standard deviation within those three samples, you will probably learn nothing new from collecting more measurements. This is very typical of programs with underlying consistency (e.g., static benchmarks). However, if you see an abnormally high standard deviation, I do not recommend launching new runs and hoping to have "better statistics". You should figure out what is causing performance variance and how to reduce it.

In an automated setting, you 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. The lower the standard deviation in the distribution, the lower the number of samples you need. Once you have a standard deviation lower than the threshold, you can stop collecting measurements. This strategy is explained in more detail in [Akinshin, 2019, Chapter 4].

An important thing to watch out for is the presence of outliers. It is OK to discard some samples (for example, cold runs) as outliers, but do not deliberately discard unwanted samples from the measurement set. Outliers can be one of the most important metrics for some types of benchmarks. For example, when benchmarking software that has real-time constraints, the 99 percentile could be very interesting.

I recommend using benchmarking tools as they automate performance measurements. For example, `Hyperfine`³⁰ is a popular cross platform command-line benchmarking tool that automatically determines the number of runs, and can visualize the results as a table with mean, min, max, or as a box plot.

²⁹ SPEC CPU 2017 benchmarks - <http://spec.org/cpu2017/Docs/overview.html#benchmarks>

³⁰ `hyperfine` - <https://github.com/sharkdp/hyperfine>

In the next two sections, we will discuss how to measure wall clock time (latency), which is the most common case. However, sometimes we also may want to measure other things, like the number of requests per second (throughput), heap allocations, context switches, etc.

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`³¹. This clock is monotonic; i.e., it always goes up. System time can be retrieved from the OS with a system call. Accessing the system timer on Linux systems is possible via the `clock_gettime` system call. The system timer has a nanosecond resolution, is consistent between all the CPUs, and is independent of the CPU frequency. Even though the system timer can return timestamps with a nanosecond 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 standard way of accessing the system timer in C++ is using `std::chrono` as shown in Listing 2.1.

Listing 2.1 Using C++ `std::chrono` to access system timer

```
#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;
}
```

- **Time Stamp Counter (TSC):** this is a hardware timer that is implemented as a hardware 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 up to one minute. On x86 platforms, the value of TSC can be retrieved by using the compiler's built-in function `_rdtsc` as shown in Listing 2.2, which uses the RDTSC assembly instruction under the hood. More low-level details on benchmarking the code using the RDTSC assembly instruction can be found in the white paper [Paoloni, 2010]. On ARM platforms you can read `CNTVCT_ELO`, Counter-timer Virtual Count Register.

³¹ Unix epoch starts on 1 January 1970 00:00:00 UT: https://en.wikipedia.org/wiki/Unix_epoch.

Listing 2.2 Using `__rdtsc` compiler builtins to access TSC

```
#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 short 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 need cycle accuracy, the system timer should be enough for a large proportion of cases. It's important to keep in mind that accessing the system timer usually has a higher latency than accessing TSC. Making a `clock_gettime` system call can be much slower than executing RDTSC instruction. The latter takes about 5 ns (20 CPU cycles), while the former takes about 500 ns. 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 the wiki page of the CppPerformanceBenchmarks repository.³²

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++, you can use the Google [benchmark](#)³³ library, C# has the [BenchmarkDotNet](#)³⁴ library, Julia has the [BenchmarkTools](#)³⁵ package, Java has [JMH](#)³⁶ (Java Microbenchmark Harness), Rust has the Criterion³⁷ package, 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 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");
```

³² CppPerformanceBenchmarks wiki - <https://gitlab.com/chriscox/CppPerformanceBenchmarks/-/wikis/ClockTimeAnalysis>

³³ Google benchmark library - <https://github.com/google/benchmark>

³⁴ BenchmarkDotNet - <https://github.com/dotnet/BenchmarkDotNet>

³⁵ Julia BenchmarkTools - <https://github.com/JuliaCI/BenchmarkTools.jl>

³⁶ Java Microbenchmark Harness - <http://openjdk.java.net/projects/code-tools/jmh/etc>

³⁷ Criterion.rs - <https://github.com/bheisler/criterion.rs>

Blunders like that one are nicely captured in the paper “Always Measure One Level Deeper” [Ousterhout, 2018], where the author advocates for a more scientific approach, and measuring performance from different angles. Following advice from the paper, we should inspect the performance profile of the benchmark and make sure 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`-like³⁸ 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. A good benchmark tests performance in realistic conditions. In contrast, 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, 2023b, 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.

2.7 Active Benchmarking

As you have seen in the previous sections, measuring performance is a complex task with many pitfalls along the way. As human beings, we tend to welcome favorable results and ignore unfavorable ones. This often leads to benchmarking done in a “run and forget” style, with no additional analysis, and overlooking any potential problems. Measurements done in this way are likely incomplete, misleading, or even erroneous. Consider the following two scenarios:

- Developer A on a team meeting: “If we add the `final` keyword to the class declaration across our entire C++ codebase, it will make our code 5% faster, with some tests showing up to 30% speedup.”

³⁸ For JMH, this is known as the `Blackhole.consume()`.

- Developer B on the next team meeting: “I looked closely at the performance impact of adding the `final` keyword to the class declarations. First, I performed longer tests and haven’t measured speedups larger than 5%. The initially observed 30% speedups were outliers caused by test instability. I also noticed that the two machines used for measurements have different configurations: while the CPUs are identical, one of the machines has faster memory modules. I reran the tests on the same machine and observed a performance difference within 1%. I compared the generated machine code before and after the change and found no significant differences. Also, I compared the number of instructions executed, cache misses, page faults, context switches, etc., and haven’t found any anomalies. At this point, I concluded that the performance impact of the `final` keyword is negligible compared to other optimizations we could make.”

Benchmarking done by developer A was done in a passive way. The results were presented without any technical explanation, and the performance impact was exaggerated. In contrast, developer B performed *active benchmarking*.³⁹ She ensured proper machine configuration, ran extensive testing, looked one level deeper, and collected as many metrics as possible to support her conclusions. Her analysis explains the underlying technical reason for the performance results she observed.

You should have a good intuition to spot suspicious benchmark results. Whenever you see publications that present benchmark results that look too good to be true and without any technical explanation, you should be skeptical. There is nothing wrong with presenting the results of your measurements, but as John Ousterhout said, “Performance measurements should be considered guilty until proven innocent.” [Ousterhout, 2018] The best way to verify the results is through active benchmarking. Active benchmarking requires much more effort than passive benchmarking, but it is the only way to get reliable results.

Questions and Exercises

1. Is it always safe to take a mean of a series of measurements to determine the running time of a program? What are the pitfalls?
2. Suppose you’ve identified a performance bug that you’re now trying to fix in your development environment. How would you reduce the noise in the system to have more pronounced benchmarking results?
3. Is it OK to track the overall performance of a program with function-level unit tests?
4. Does your organization have a performance regression system in place? If yes, can it be improved? If not, think about the strategy of installing one. Take into consideration: what is changing and what isn’t (source code, compiler, hardware configuration, etc.), how often a change occurs, what is the measurement variance, what is the running time of the benchmark suite, and how many iterations you can run.

³⁹ A term coined by Brendan Gregg - <https://www.brendangregg.com/activebenchmarking.html>.

Chapter Summary

- Modern systems have nondeterministic performance. Eliminating nondeterminism in a system is helpful for well-defined, stable performance tests, e.g., microbenchmarks.
- Measuring performance in production is required to assess how users perceive the responsiveness of your services. However, this requires dealing with noisy environments and using statistical methods for analyzing results.
- It is beneficial to employ an automated performance tracking system to prevent performance regressions from leaking into production software. Such CI systems are supposed to run automated performance tests, visualize results, and alert on discovered performance anomalies.
- Visualizing performance distributions helps compare performance results. It is a safe way of presenting performance results to a wide audience.
- To benchmark execution time, engineers can use two different timers: the system-wide high-resolution timer and the Time Stamp Counter. The former is suitable for measuring events whose duration is more than a microsecond. The latter can be used for measuring short events with high accuracy.
- Microbenchmarks are good for quick experiments, but you should always verify your ideas on a real application in practical conditions. Make sure that you are benchmarking the right code by checking performance profiles.
- Always measure one level deeper, collect as many metrics as possible to support your conclusions, and be ready to explain the underlying technical reasons for the performance results you observe.

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], [Shen & Lipasti, 2013]. I provide a recap of features that are present in modern processors to prepare the reader for what comes next in the book.

3.1 Instruction Set Architecture

The instruction set architecture (ISA) is the contract between the software and the hardware, which defines the rules of communication. Intel x86-64,⁴⁰ Armv8-A, and RISC-V are examples of current-day ISAs that are widely deployed. All of these are 64-bit architectures, i.e., all address computations use 64 bits. ISA developers and CPU architects typically ensure that software or firmware conforming to the specification will execute on any processor built using the specification. Widely deployed ISAs 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, such as RISC-V and ARM where the operands are explicitly specified, and memory is accessed only using load and store instructions. The X86 ISA is a register-memory architecture, where operations can be performed on registers, as well as memory operands. In addition to providing the basic functions in an ISA such as load, store, control, and scalar arithmetic operations using integers and floating-point, the widely deployed architectures continue to augment their ISAs to support new computing paradigms. These include enhanced vector processing instructions (e.g., Intel AVX2, AVX512, ARM SVE, RISC-V “V” vector extension) and matrix/tensor instructions (Intel AMX, ARM SME). Applications that extensively use these advanced instructions typically see large performance gains.

Modern CPUs support 32-bit and 64-bit precision for floating-point and integer arithmetic operations. With the fast-evolving fields of machine learning and AI, the industry has a renewed interest in alternative numeric formats to drive significant performance improvements. Research has shown that machine learning models perform just as well using fewer bits to represent variables, saving on both compute and memory bandwidth. As a result, the majority of mainstream ISAs have recently added support for lower precision data types such as 8-bit and 16-bit integer and floating-point types (int8, fp8, fp16, bf16), in addition to the traditional 32-bit and 64-bit formats for arithmetic operations.

⁴⁰In the book I often write x86 for shortness, but I assume x86-64, which is a 64-bit version of the x86 instruction set, first announced in 1999. Also, I use ARM to refer to the ISA, and Arm to refer to the company.

3.2 Pipelining

Pipelining is a foundational technique used to make CPUs fast, wherein multiple instructions overlap during their execution. Pipelining in CPUs drew inspiration from 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 a relatively simple architecture designed by John L. Hennessy and David A. Patterson in 1994. As defined in [Hennessy & Patterson, 2017], it has a 5-stage pipeline which consists of:

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

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 3.1: Simple 5-stage pipeline diagram.

Figure 3.1 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, the instruction x+1 couldn't start its execution until after instruction x had finished its work.

Modern high-performance CPUs have multiple pipeline stages, often ranging from 10 to 20 (sometimes more), depending on the architecture and design goals. This involves a much more complicated design than a simple 5-stage pipeline introduced earlier. For example, the decode stage may be split into several new stages. We may also add new stages before the execute stage to buffer decoded instructions and so on.

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 next 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 of work that can be done in a stage as this

directly affects the frequency of operation of the CPU.

In real implementations, pipelining introduces several constraints that limit the nicely-flowing execution illustrated in Figure 3.1. 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, i.e., when there are two instructions competing for the same resource. An example of such a hazard is when two 32-bit addition instructions are ready to execute in the same cycle, but there is only one execution unit available in that cycle. In this case, we need to choose which one of the two instructions to execute and which one will be executed in the next cycle. To a large extent, they could be eliminated by replicating hardware resources, such as using multiple execution units, instruction decoders, multi-ported register files, etc. However, this 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:

A *read-after-write* (RAW) hazard requires a dependent read to execute after a write. It occurs when instruction $x+1$ reads a source before 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 the addition `R0 ADD 1` is done (from the EXE pipeline stage), we don't need to wait until the WB stage finishes (when 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.

A *write-after-read* (WAR) hazard requires a dependent write to execute after a read. It occurs when an instruction writes a register before an earlier instruction reads the source, resulting in the wrong new value being read. A WAR hazard is not a true dependency and can be eliminated by a technique called *register renaming*. 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 the architectural state, solving WAR hazards is simple: we just need to use a different physical register for the write operation. For example:

<pre>; machine code, WAR hazard ; (architectural registers) R1 = R0 ADD 1 R0 = R2 ADD 2</pre>	<pre>; after register renaming ; (physical registers) => R101 = R100 ADD 1 R103 = R102 ADD 2</pre>
---	--

In the original assembly code, there is a WAR dependency for register R0. For the code on the left, we cannot reorder the execution of the instructions, because it could leave the wrong value in R1. However, we can leverage our large pool of physical registers to overcome this limitation. To do that we need to rename all the occurrences of the R0 register starting from the write operation ($R0 = R2 \text{ ADD } 2$) and below to use a free register. After renaming, we give these registers new names that correspond to physical registers, say R103. By renaming registers, we eliminated a WAR hazard in the initial code, and we can safely execute the two operations in any order.

A *write-after-write* (WAW) hazard requires a dependent write to execute after a write. It occurs when an instruction writes to a register before an earlier instruction writes to the same register, resulting in the wrong value being stored. WAW hazards are also eliminated by register renaming, allowing both writes to execute in any order while preserving the correct final result. Below is an example of eliminating WAW hazards.

<pre>; machine code, WAW hazard (architectural registers) R1 = R0 ADD 1 R2 = R1 SUB R3 ; RAW R1 = R0 MUL 3 ; WAW and WAR</pre>	\Rightarrow <pre>; after register renaming (physical registers) R101 = R100 ADD 1 R102 = R101 SUB R103 ; RAW R104 = R100 MUL 3</pre>
--	--

You will see similar code in many production programs. In our example, R1 keeps the temporary result of the ADD operation. Once the SUB instruction is complete, R1 is immediately reused to store the result of the MUL operation. The original code on the left features all three types of data hazards. There is a RAW dependency over R1 between ADD and SUB, and it must survive register renaming. Also, we have WAW and WAR hazards over the same R1 register for the MUL operation. Again, we need to rename registers to eliminate those two hazards. Notice that after register renaming we have a new destination register (R104) for the MUL operation. Now we can safely reorder MUL with the other two operations.

- **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 mitigate 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 numerous additional hardware features to exploit such instruction-level parallelism (ILP), i.e., parallelism within a single stream of instructions. Working in concert with advanced compiler techniques, these hardware features provide significant performance improvements.

3.3.1 Out-Of-Order (OOO) Execution

The pipeline example in Figure 3.1 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, also known as *program order*. Most modern CPUs support *out-of-order* (OOO) execution, where sequential instructions can enter the execution stage in any arbitrary order only limited by their dependencies and resource availability. CPUs with OOO execution must still give the same result as if all instructions were executed in the program order.

An instruction is called *retired* after 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 execution is primarily used to avoid underutilization of CPU resources due to stalls caused by dependencies, especially in superscalar engines, which we will discuss shortly.

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 3.2: The concept of out-of-order execution.

Figure 3.2 illustrates the concept of out-of-order execution. Let's assume that instruction $x+1$ cannot be executed during cycles 4 and 5 due to some conflict. An in-order CPU would stall all subsequent instructions from entering the EXE pipeline stage, so instruction $x+2$ would begin executing only at cycle 7. In a CPU with OOO execution, an instruction can begin executing as long as it does not have any conflicts (e.g., its inputs are available, the execution unit is not occupied, etc.). As you can see on the diagram, instruction $x+2$ started executing before instruction $x+1$. The instruction $x+3$ cannot enter the EXE stage at cycle 6, because it is already occupied by instruction $x+1$. All instructions still retire in order, i.e., the instructions complete the WB stage in the program order.

OOO execution usually brings large performance improvements over in-order execution. However, it also introduces additional complexity and power consumption.

The process of reordering instructions is often called instruction *scheduling*. The goal of scheduling is to issue instructions in such a way as to minimize pipeline hazards and maximize the utilization of CPU resources. Instruction scheduling can be done at compile time (static scheduling), or at runtime (dynamic scheduling). Let's unpack both options.

3.3.1.1 Static scheduling The Intel Itanium was an example of static scheduling. With static scheduling of a superscalar, multi-execution unit machine, the scheduling

is moved 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 and loop unrolling to look farther ahead than can be reasonably supported by hardware structures to find the right ILP.

The Intel Itanium never managed to become a success for a few reasons. One of them was a lack of compatibility with x86 code. Another was the fact that it was very difficult for compilers to schedule instructions in such a way as to keep the CPU busy due to variable load latencies. The 64-bit extension of the x86 ISA (x86-64) was introduced in the same time window by AMD, was compatible with IA-32 (the 32-bit version of the x86 ISA) and eventually became its real successor. The last Intel Itanium processors were shipped in 2021.

3.3.1.2 Dynamic scheduling To overcome the problem with static scheduling, modern processors use dynamic scheduling. The two most important algorithms for dynamic scheduling are [Scoreboarding⁴¹](#) and the [Tomasulo algorithm⁴²](#).

Scoreboarding was first implemented in the CDC6600 processor in the 1960s. Its main drawback is that it not only preserves true dependencies (RAW), but also false dependencies (WAW and WAR), and therefore it provides suboptimal ILP. False dependencies are caused by the small number of architectural registers, which is typically between 16 and 32 in modern ISAs. That is why all modern processors have adopted the Tomasulo algorithm for dynamic scheduling. The Tomasulo algorithm was invented in the 1960s by Robert Tomasulo and first implemented in the IBM360 Model 91.

To eliminate false dependencies, the Tomasulo algorithm makes use of register renaming, which we discussed in the previous section. Because of that, performance is greatly improved compared to scoreboarding. However, a sequence of instructions that carry a RAW dependency, also known as a *dependency chain*, is still problematic for OOO execution, because there is no increase in the ILP after register renaming since all the RAW dependencies are preserved. Dependency chains are often found in loops (loop carried dependency) where the current loop iteration depends on the results produced on the previous iteration.

Other two key components for implementing dynamic scheduling are the Reorder Buffer (ROB) and the Reservation Station (RS). The ROB is a circular buffer that keeps track of the state of each instruction, and in modern processors, it has several hundred entries. Typically, the size of the ROB determines how far ahead the hardware can look for scheduling instructions independently. Instructions are inserted in the ROB in program order, can execute out of order, and retire in program order. Register renaming is done when instructions are placed in the ROB.

From the ROB, instructions are inserted in the RS, which has much fewer entries. Once instructions are in the RS, they wait for their input operands to become available. When inputs are available, instructions can be issued to the appropriate execution unit.

⁴¹ Scoreboarding - <https://en.wikipedia.org/wiki/Scoreboarding>.

⁴² Tomasulo algorithm - https://en.wikipedia.org/wiki/Tomasulo_algorithm.

So instructions can be executed in any order once their operands become available and are not tied to the program order any longer. Modern processors are becoming wider (can execute many instructions in one cycle) and deeper (larger ROB, RS, and other buffers), which demonstrates that there is a lot of potential to uncover more ILP in production applications.

3.3.2 Superscalar Engines

Most modern CPUs are superscalar, i.e., they can issue more than one instruction in any given cycle. *Issue width* is the maximum number of instructions that can be issued during the same cycle (see Section 4.5). The typical issue width of a mainstream CPU in 2024 ranges from 6 to 9. To ensure the right balance, such superscalar engines also have more than one 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 3.3: A pipeline diagram of a code executing in a 2-way superscalar CPU.

Figure 3.3 shows a pipeline diagram of a CPU that supports 2-wide issue. Notice that two instructions can be processed in each stage of the pipeline every cycle. For example, both instructions x and $x+1$ started their execution during cycle 3. This could be two instructions of the same type (e.g., two additions) or two different instructions (e.g., an addition and a branch). Superscalar processors replicate execution resources to keep instructions in the pipeline flowing through without structural conflicts. For instance, to support the decoding of two instructions simultaneously, we need to have 2 independent decoders.

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*. Using this technique, a CPU predicts the likely target of branches and starts executing instructions from the predicted path (known as *speculative execution*).

Let's consider an example in Listing 3.1. 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 is determined, as shown in Figure 3.4a.

Listing 3.1 Speculative execution

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

Instruction	Clock cycle								
	1	2	3	4	5	6	7	8	9
BRANCH ($a < b$)	IF	ID	EXE	MEM	WB				
CALL foo				IF	ID	EXE	MEM	WB	
// Instr from foo					IF	ID	EXE	MEM	WB

(a) No speculation

Instruction	Clock cycle								
	1	2	3	4	5	6	7	8	9
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. Speculative work is marked with *.

Figure 3.4: The concept of speculative execution.

With speculative execution, the CPU guesses 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 3.4b). State changes to the machine cannot be committed until the condition is resolved to ensure that the architectural state of the machine is never impacted by speculatively executing instructions.

In the example above, the branch instruction compares two scalar values, which is fast. But in reality, a branch instruction can depend 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 will discuss in Section 4.8.

An instruction that is executed speculatively is marked as such in the ROB. Once it is not speculative any longer, it can retire in program order. Here is where the architectural state is committed, and architectural registers are updated. Because the results of the speculative instructions are not committed, it is easy to roll back when a misprediction happens.

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 sophisticated dynamic branch prediction mechanisms that 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 they frequently represent error-checking code. Backward conditional jumps are frequently seen in loops and are 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 call. 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 every cycle to generate the next address from which to fetch instructions. 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 look up the target address in the BTB. Every cycle the BPU needs to generate the next address from which to fetch instructions to avoid 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 the branch will be taken or not. If it is not taken, then we fall through and there is no need to look up the target. Otherwise, we look up 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. This is also known as local correlation.

- **Spatial correlation:** several adjacent branches may resolve in a highly correlated manner (a preferred path of execution). This is also known as global correlation.

The best accuracy is often achieved by leveraging local and global correlations together. So, not only do we look at the outcome history of the current branch, but also we correlate it with the outcomes of adjacent 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 a complex predictor and pollute its data structures. A much simpler mechanism can be used instead. 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] predictors. Championship⁴³ branch predictors make less than 3 mispredictions per 1000 instructions. Modern CPUs routinely reach a >95% prediction rate on most workloads.

3.4 SIMD Multiprocessors

Another technique to facilitate parallel processing is called Single Instruction Multiple Data (SIMD), which is used in nearly all high-performance processors. As the name indicates, in a SIMD processor, 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. A SIMD architecture enables more efficient processing of a large amount of data and works best for data-parallel applications that involve vector operations.

Figure 3.5 shows scalar and SIMD execution modes for the code in Listing 3.2. In a traditional Single Instruction Single Data (SISD) mode, also known as *scalar* mode, the 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 that 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 fewer instructions and can potentially gain a 4x speedup over four scalar computations.

Listing 3.2 SIMD execution

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

For regular integer SISD instructions, processors utilize general-purpose registers. Similarly, for SIMD instructions, CPUs have a set of SIMD registers to keep the

⁴³ 5th Championship Branch Prediction competition - <https://jilp.org/cbp2016>.

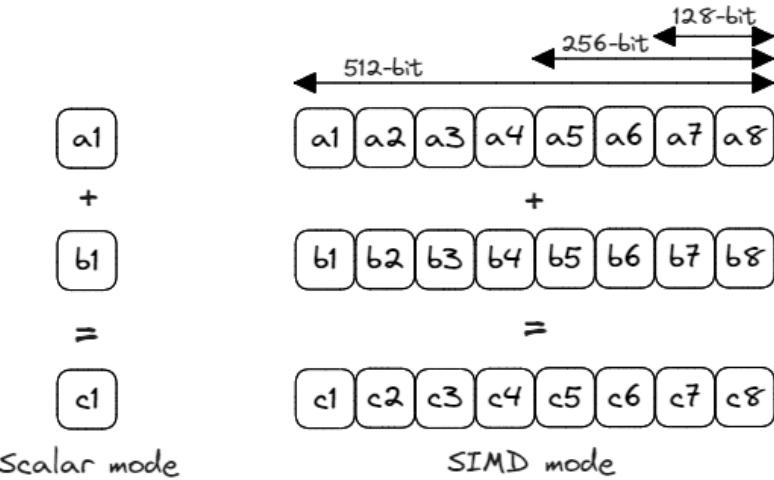


Figure 3.5: Example of scalar and SIMD operations.

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 the 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 corresponding to array c . Note, that the data elements can be either integers or floating-point numbers.

A vector execution unit is logically divided into *lanes*. In the context of SIMD, a lane refers to a distinct data pathway within the SIMD execution unit and processes one element of the vector. In our example, each lane processes 64-bit elements (double-precision), so there will be 4 lanes in a 256-bit register.

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, and 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*. Initially, SIMD instructions were programmed in assembly. Later, special compiler intrinsics, which are small functions providing a one-to-one mapping to SIMD instructions, were introduced. Today all the major compilers support autovectorization for the popular processors, i.e., they can generate SIMD instructions straight from high-level code written in C/C++, Java, Rust, and other languages.

To enable 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 16384-bit vectors, which greatly reduces the number of instructions executed. At each loop iteration, SVE 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 enables a programmer to query/set the `number_of_lanes`.

Going back to the example in Listing 3.2, if N equals 5, and we have a 256-bit vector, we cannot process all the elements in a single iteration. We can process the first four elements using a single SIMD instruction, but the 5th element needs to be processed individually. This is known as the *loop remainder*. Loop remainder is a portion of a loop that must process fewer elements than the vector width, requiring additional scalar code to handle the leftover elements. Scalable vector ISA extensions do not have this problem, as they can process any number of elements in a single instruction. Another solution to the loop remainder problem is to use *masking*, which allows selectively enabling or disabling SIMD lanes based on a condition.

Also, CPUs increasingly accelerate the matrix multiplications often used in machine learning. Intel's AMX extension, supported in server processors since 2023, 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 uses 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 in Figure 3.5, newer use cases of SIMD include:

- String processing: finding characters, validating UTF-8,⁴⁴ parsing JSON⁴⁵ and CSV;⁴⁶
- Hashing,⁴⁷ random generation,⁴⁸ cryptography(AES);
- Columnar databases (bit packing, filtering, joins);
- Sorting built-in types (VQSort,⁴⁹ QuickSelect);
- Machine Learning and Artificial Intelligence (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 a CPU. Next, we will discuss three

⁴⁴ UTF-8 validation - <https://github.com/rusticstuff/simdutf8>

⁴⁵ Parsing JSON - <https://github.com/simdjson/simdjson>

⁴⁶ Parsing CSV - <https://github.com/geofflangdale/simdcsv>

⁴⁷ SIMD hashing - <https://github.com/google/highwayhash>

⁴⁸ Random generation - abseil library

⁴⁹ Sorting - VQSort

techniques to exploit Thread-Level Parallelism (TLP): multicore systems, simultaneous multithreading, and hybrid architectures. Such techniques make it possible 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 that attempted to increase core counts for each generation. The idea was to replicate multiple processor cores on a single chip and let them serve different programs at the same time. For example, one of the cores could run a web browser, another core could render a video, and yet another could play music, all at the same time. For a server machine, requests from different clients could be processed on separate cores, which could greatly increase the throughput of such a system.

The first consumer-focused dual-core processor was the Intel Core 2 Duo, released in 2005, which 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 contain more than ten physical cores and server processors contain more than 100 cores on a single socket.

It may sound very impressive, but we cannot add cores infinitely. 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. This means that when more cores are running, heat can quickly exceed cooling capability. In such a situation, multicore processors will reduce clock speeds. This is one of the reasons you can see server chips with a large 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 shared resources, such as last-level cache and memory controllers. Such a communication channel is called an *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 count grows. 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 the throughput of other shared resources, e.g., interconnect bandwidth, last-level cache size and bandwidth, and memory bandwidth. 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 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 that happened to be scheduled on the same physical core.

An example of execution on a non-SMT and a 2-way SMT (SMT2) processor is shown in Figure 3.6. In both cases, the width of the processor pipeline is four, and each slot represents an opportunity to issue a new instruction. 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 slots, so the available resources are not utilized well. This may happen for a variety of reasons. For instance, in cycle 3, thread 1 cannot make forward progress because all instructions are waiting for their inputs to become available. Non-SMT processors would simply stall, while SMT-enabled processors take this opportunity to schedule useful work from another thread. The goal here is to occupy unused slots by another thread to improve hardware utilization and multithreaded performance.

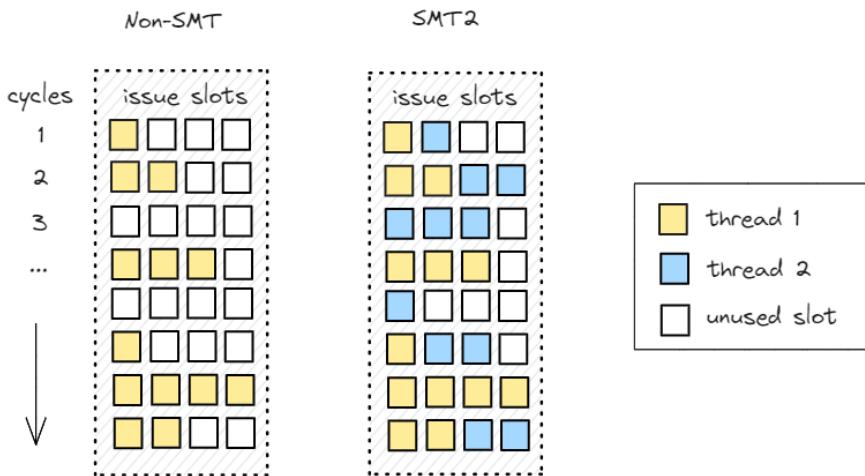


Figure 3.6: Execution on a 4-wide non-SMT and a 4-wide SMT2 processor.

With an SMT2 implementation, each physical core is represented by 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 an SMT-enabled core and each consistently utilizes only two out of four available slots, then there is a high chance they will run as fast as if they were 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 contexts which helps preserve the correctness of execution. To support SMT, a CPU must replicate the 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 Frontend, they fetch instructions in an alternating order (every cycle or a few cycles). In the backend, each cycle a processor selects instructions for execution from all threads. Instruction execution is mixed as the processor dynamically schedules execution units among both threads.

So, SMT is a very flexible setup that makes it possible to recover unused CPU issue slots. SMT provides equal single-thread performance, in addition to its benefits for multiple threads. Modern CPUs that choose to support SMT, usually implement two-way (SMT2), and sometimes four-way (SMT4) SMT.

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 likely SMT penalty is caused by competition for 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 it harder to predict and measure the performance of an application that runs on an SMT core. Imagine you're running performance-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 them with someone else. This problem is especially pronounced in a cloud environment when you cannot predict whether your application will 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 one application could 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 topic since hardware security is outside the scope of this book.

3.5.3 Hybrid Architectures

Computer architects also developed a hybrid CPU design in which two (or more) types of cores 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 tasks and small cores provide reduced power consumption. 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 portions of single-threaded bursty execution as well as portions of work that can be scaled 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 which has four high-performance "Firestorm" and four energy-efficient

“Icestorm” cores. Intel introduced its Alder Lake 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 they come with their 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, scheduling becomes restricted. For example, if a big core features some fancy instructions that are not available on small cores, then you can only assign big cores to run workloads that use such instructions. That’s why usually vendors use the “greatest common denominator” approach when choosing the ISA for a hybrid processor.

Even with ISA-compatible cores, scheduling becomes challenging. Different types of workloads call for a specific scheduling scheme, e.g., bursty execution vs. steady execution, low IPC vs. high IPC,⁵⁰ low importance 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 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 the case of 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.

3.6 Memory Hierarchy

To effectively utilize all the hardware resources provisioned in a CPU, the machine needs to be fed with the right data at the right time. Failing to do so requires fetching a variable from the main memory, which takes around 100 ns. From a CPU perspective, it is a very long time. Understanding the memory hierarchy is critically important to delivering 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, the same location will likely be accessed again soon. Ideally, we want this information to be in the cache next time we need it.
- **Spatial locality:** when a given memory location is accessed, nearby locations will likely be accessed soon. 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 (a 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.

⁵⁰ IPC - Instructions Per Cycle, see Section 4.3.

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 core, while other levels can be shared among cores.

Caches are organized as blocks with a defined size, also known as *cache lines*. The typical cache line size in modern CPUs is 64 bytes. However, the notable exception here is the L2 cache in Apple processors (such as M1, M2, and later), which operates on 128B cache lines. Caches closest to the execution pipeline typically range in size from 32 KB to 128 KB. Mid-level caches tend to have 1MB and above. Last-level caches in modern CPUs can be tens or even hundreds of megabytes.

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. Direct-mapped caches are relatively easy to build and have fast access time, however, they have a high miss rate.

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

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

In a *fully associative* cache, a given block can be placed in any location in the cache. This approach involves high hardware complexity and slow access time, thus considered impractical for most use cases.

An intermediate option between 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:

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

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

Consider an example of an L1 cache, whose size is 32 KB with 64 bytes cache lines, 64 sets, and 8 ways. The total number of cache lines in such a cache is $32 \text{ KB} / 64 \text{ bytes} = 512 \text{ lines}$. A new line can only be inserted in its appropriate set (one of the 64 sets). Once the set is determined, a new line can go to one of the 8 ways in this set. Similarly, when you later search for this cache line, you determine the set first, and then you only need to examine up to 8 ways in the set.

Here is another example of the cache organization of the Apple M1 processor. The L1 data cache inside each performance core can store 128 KB, has 256 sets with 8 ways in each set, and operates on 64-byte lines. Performance cores form a cluster and share the L2 cache, which can keep 12 MB, is 12-way set-associative, and operates on 128-byte lines. [Apple, 2024]

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 a bit to indicate whether the data is valid. Tags can also contain additional bits to indicate access information, sharing information, etc.

Block Address		Block offset
Tag	Index	

Figure 3.7: Address organization for cache lookup.

Figure 3.7 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 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 cache 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 new data. Another alternative is to randomly select one of the blocks as the victim block.

3.6.1.4 Managing Writes. Write accesses to caches are less frequent than data reads. 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* 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 extracting performance from any application. From the perspective of the CPU 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.

3.6.1.6 Hardware and Software Prefetching. One method to avoid cache misses and subsequent stalls is to prefetch data into caches prior to when the pipeline demands it. 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 provide 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 an application, such as varying data sets, and does not require support from an optimizing compiler. Also, the hardware prefetching works without the overhead of additional address generation and prefetch instructions. However, hardware prefetching works for a limited set of commonly used data access patterns.

Software memory prefetching complements prefetching done by hardware. Developers can specify which memory locations are needed ahead of time via dedicated hardware

instruction (see Section 8.5). 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 in the northbridge chip on the motherboard. 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, along with its price of course. Memory density defines the capacity of the module measured in GB. Obviously, the more available memory the better as it is a precious resource used by the OS and applications.

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

3.6.2.1 DDR (Double Data Rate) 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 increased. Table 3.1 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 in millions of transfers per second (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. You can see an example of measuring observed memory latency and bandwidth in Section 4.10.

Table 3.1: Performance characteristics for the last three generations of DDR technologies.

DDR Generation	Year	Highest Data Rate(MT/s)	Peak Bandwidth (GB/s)	In-device Read Latency(ns)
DDR3	2007	2133	17.1	10.3
DDR4	2014	3200	25.6	12.5
DDR5	2020	6400	51.2	14

It is worth mentioning that DRAM chips require their memory cells to be refreshed periodically. This is because the bit value is stored as the presence of an electric

charge on a tiny capacitor, so it can lose its charge over time. To prevent this, there is 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.

A DRAM module is organized as a set 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: $\times 4$, $\times 8$, or $\times 16$, and defines how wide is the bus that goes to each chip. As an example, Figure 3.8 shows the organization of a 2Rx16 dual-rank DRAM DDR4 module, with a total of 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.

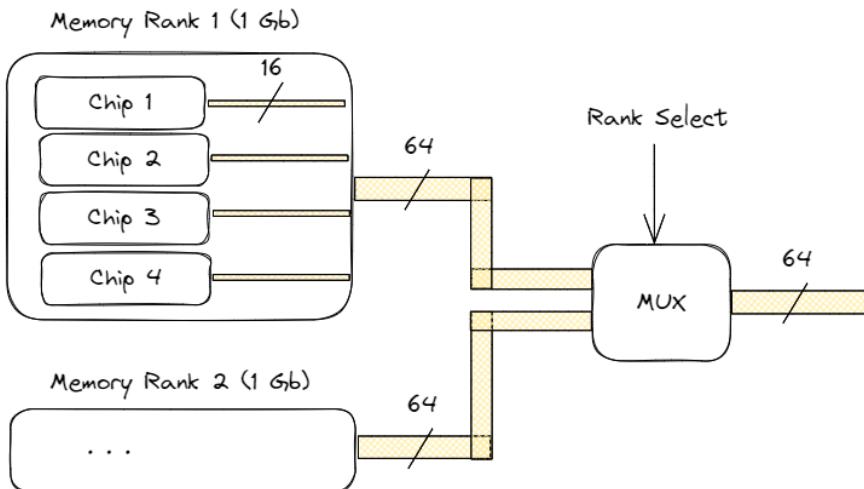


Figure 3.8: Organization of a 2Rx16 dual-rank DRAM DDR4 module with a total capacity of 2GB.

There is no direct answer as to whether the performance of single-rank or dual-rank is better as it depends on the type of application. Single-rank modules generally produce less heat and are less likely to fail. Also, multi-rank modules require a rank select signal to switch from one rank to another, which needs additional clock cycles and 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.

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 bits to 128 bits, doubling the available bandwidth, see Figure 3.9. 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 or eight memory channels.

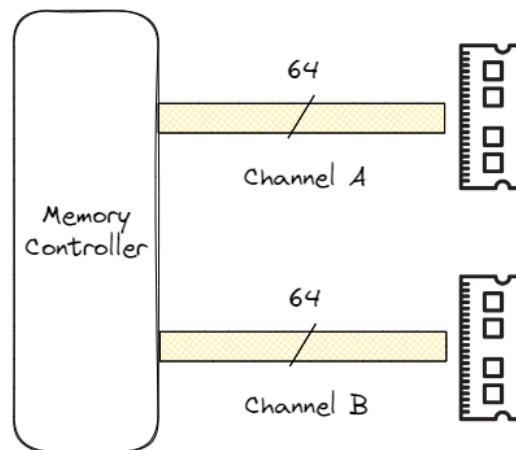


Figure 3.9: Organization of a dual-channel DRAM setup.

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 calculation to determine the maximum memory bandwidth for a given memory technology, using the simple formula below:

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

For example, for a single-channel DDR4 configuration with a data rate of 2400 MT/s and 64 bits (8 bytes) per transfer, the maximum bandwidth equals $2400 * 8 = 19.2$ GB/s. Dual-channel or dual memory controller setups double the bandwidth to 38.4 GB/s. Remember though, 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 motherboard that support such an 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, you can use the `dmidecode` command. Alternatively, 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. An example of a 2-way interleaving for sequential memory accesses is shown in Figure 3.10. As before, we have a dual-channel memory configuration (channels A and B) with two independent memory controllers. Modern processors interleave per four cache lines (256 bytes), i.e., the first four adjacent cache lines go to channel A, and then the next set of four cache lines go to channel B.

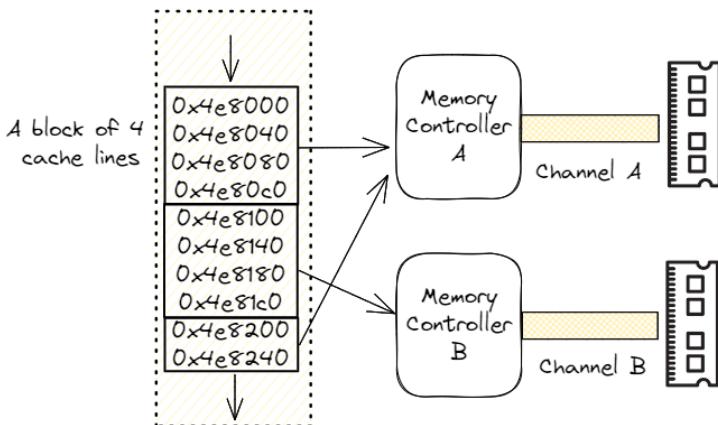


Figure 3.10: 2-way interleaving for sequential memory access.

Without interleaving, consecutive adjacent accesses would be sent to the same memory controller, not utilizing the second available controller. In contrast, interleaving enables hardware parallelism to better utilize available memory bandwidth. 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, and physics simulation, but also 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 primarily designed for graphics and nowadays it is used on virtually every high-performance graphics card. While GDDR shares some characteristics 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 words (64 bits in total) 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 the HBM memory bus is very wide: 1024 bits for each HBM stack. This enables 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 want to maximize data transfer throughput. 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 a CPU. In fact, the first x86 general-purpose server chips with integrated HBM are now available.

3.7 Virtual Memory

Virtual memory is the mechanism to share the physical memory attached to a CPU with all the processes executing on it. Virtual memory provides a protection mechanism that prevents access to the memory allocated to a given process from other processes. Virtual memory also provides relocation, which is the ability to load a program anywhere in physical memory without changing the addresses 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 data from memory requires physical addresses. 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.

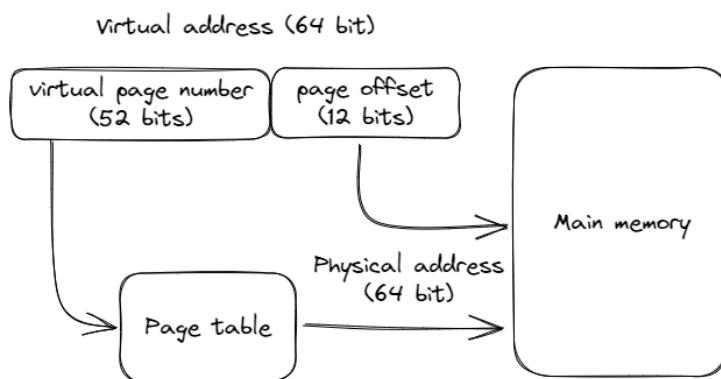


Figure 3.11: Virtual-to-physical address translation for 4KB pages.

Virtual-to-physical address translation is required for accessing data as well as code (instructions). The translation mechanism for a system with a page size of 4KB is shown in Figure 3.11. The virtual address is split into two parts. The virtual page number (52 most significant bits) is used to index into the page table to produce a mapping between the virtual page number and the corresponding physical page. The 12 least significant bits are used to offset within a 4KB page. These bits do not require translation and are used “as-is” to access the physical memory location.

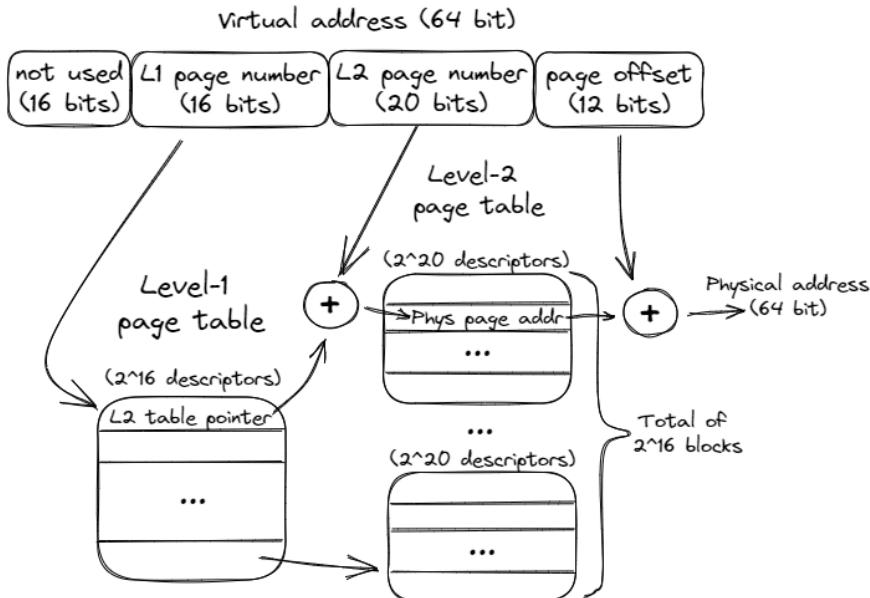


Figure 3.12: Example of a 2-level page table.

The page table can be either single-level or nested. Figure 3.12 shows one example of a 2-level page table. Notice how the address gets split into more pieces. The first thing to mention is that the 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*.

A nested page table is a radix tree that keeps physical page addresses along with some metadata. To find a translation within a 2-level page table, we first use bits 32..47 as an index into the Level-1 page table also known as the *page 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 dictated by the CPU for reasons we will discuss a few paragraphs later. Thus the variations of page table organization are limited by what a CPU supports. Modern CPUs support both 4-level page tables with 48-bit pointers (256 TB of total memory) and 5-level page tables with 57-bit pointers (128 PB of total memory).

Breaking the page table into multiple levels doesn't change the total addressable memory. However, a nested approach does not require storing 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) the operating system committed to allocating a page but hasn't yet backed it with a physical page, and 2) an 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 a traversal is called a *page walk*. To reduce the address translation time, CPUs support a hardware structure called a 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), and L1 DTLB (Data), followed by a shared (instructions and data) L2 STLB.

To lower memory access latency, the L1 cache lookup can be partially overlapped with the DTLB lookup thanks to a constraint on the cache associativity and size that allows the L1 set selection without the physical address.⁵¹ However, higher level caches (L2 and L3) - which are also typically Physically Indexed and Physically Tagged (PIPT) but cannot benefit from this optimization - therefore require the address translation before the cache lookup.

The TLB hierarchy keeps translations for a relatively large memory space. Still, misses in the TLB can be very costly. To speed up the handling of TLB misses, CPUs have a mechanism called a *hardware page walker*. Such a unit can perform a page walk directly in hardware 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 operating systems must comply. High-end processors have several hardware page walkers that can handle multiple TLB misses simultaneously. However, even with all the acceleration offered by modern CPUs, TLB misses still cause performance bottlenecks for many applications.

3.7.2 Huge Pages

Having a small page size makes it possible to manage the available memory more efficiently and reduce fragmentation. The drawback though is that it requires having more page table entries to cover the same memory region. Consider two page sizes: 4KB, which is a default on x86, and a 2MB *huge page* size. For an application that operates on 10MB of data, we need 2560 entries in the first case, but 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. Throughout the rest of this book, we will refer to them as Huge Pages.

⁵¹ Minimum associativity for a PIPT L1 cache to also be VIPT, accessing a set without translating the index to physical - <https://stackoverflow.com/a/59288185>.

An example of an address that points to data within a huge page is shown in Figure 3.13. Just like with a default page size, the exact address format when using huge pages is dictated by the hardware, but luckily we as programmers usually don't have to worry about it.



Figure 3.13: 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. We will discuss how to use huge pages to reduce the frequency of TLB misses in Section 8.4 and Section 11.8. The downsides of using huge pages are memory fragmentation and, in some cases, nondeterministic page allocation latency because 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 this cannot be found, the OS needs to reorganize the pages, resulting in a longer allocation latency.

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, Golden Cove, which became available in 2021. This core is used as the P-core inside the Alder Lake and Sapphire Rapids platforms. Figure 3.14 shows the block diagram of the Golden Cove core. Notice that this section only describes a single core, not the entire processor. So, we will skip the discussion about frequencies, core counts, L3 caches, core interconnects, memory latency, and bandwidth.

The core is split into an in-order frontend that fetches and decodes x86 instructions into μ ops⁵² and a 6-wide superscalar, out-of-order backend. The Golden Cove 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) 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 Frontend

The CPU Frontend consists of several functional units that fetch and decode instructions from memory. Its main purpose is to feed prepared instructions to the CPU Backend, 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, the branch predictor unit (BPU) steers the work of the CPU Frontend. That is indicated by the arrow that goes from the BPU to the

⁵² Complex CISC instructions are translated into simple RISC microoperations, see Section 4.4.

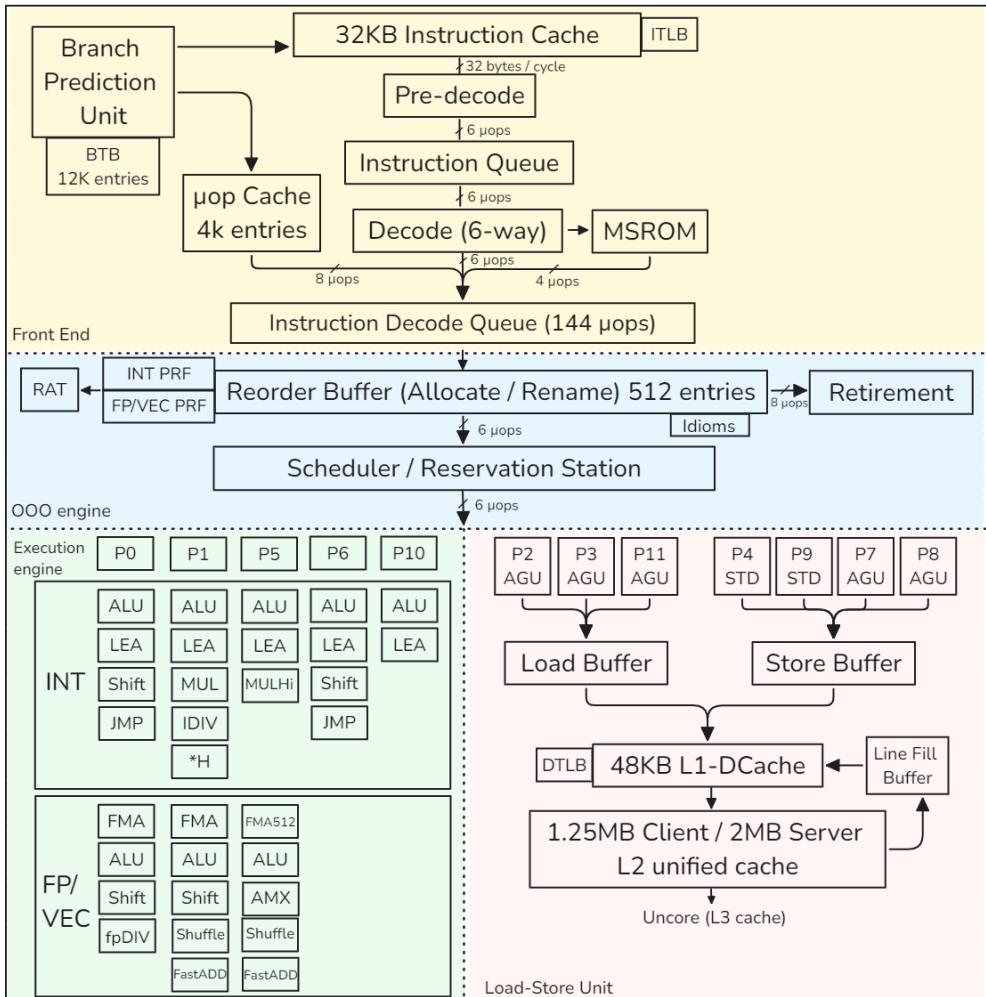


Figure 3.14: Block diagram of a CPU core in the Intel Golden Cove Microarchitecture.

instruction cache. The BPU predicts the target 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 containing information about branches and their targets. This information is used by the prediction algorithms. Every cycle, the BPU generates the next fetch address and passes it to the CPU Frontend.

The CPU Frontend fetches 32 bytes per cycle of x86 instructions from the L1 I-cache. This is shared among the two threads (if SMT is enabled), so each thread gets 32 bytes every other cycle. These are complex, variable-length x86 instructions. First, the pre-decode stage determines and marks the boundaries of the variable instructions by inspecting the chunk. In x86, the instruction length can range from 1 to 15 bytes. This stage also identifies branch instructions. The pre-decode stage moves up to 6

instructions (also referred to as *macroinstructions*) to the Instruction Queue that is split between the two threads. The instruction queue also supports a macro-op fusion unit that detects when two macroinstructions can be fused into a single micro-operation (μ op). 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 μ ops. Decoded μ ops are queued into the Instruction Decode Queue (IDQ), labeled as “ μ op Queue” on the diagram.

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

Some very complicated instructions may require more μ ops than decoders can handle. μ ops for such instruction are served from the Microcode Sequencer (MSROM). Examples of such instructions include hardware 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 μ ops per cycle into the IDQ.

The Instruction Decode Queue (IDQ) provides the interface between the in-order frontend and the out-of-order backend. The IDQ queues up the μ ops in order and can hold 144 μ ops per logical processor in single thread mode, or 72 μ ops per thread when SMT is active. This is where the in-order CPU Frontend finishes and the out-of-order CPU Backend starts.

3.8.2 CPU Backend

The CPU Backend employs an OOO engine that executes instructions and stores results. I repeated a part of the diagram that depicts the Golden Cove OOO engine in Figure 3.15.

The heart of the OOO engine is the 512-entry ReOrder Buffer (ROB). It serves a few purposes. First, it provides register renaming.⁵³ There are only 16 general-purpose integer and 32 floating-point/SIMD architectural registers, however, the number of physical registers is much higher.⁵⁴ Physical registers are located in a structure called the Physical Register File (PRF). There are separate PRFs for integer and floating-point/SIMD registers. The mappings from architecture-visible registers to the physical registers are kept in the register alias table (RAT).

⁵³ Renaming must be done in program order.

⁵⁴ There are around 300 physical general-purpose registers (GPRs) and a similar number of vector registers. The actual number of registers is not disclosed.

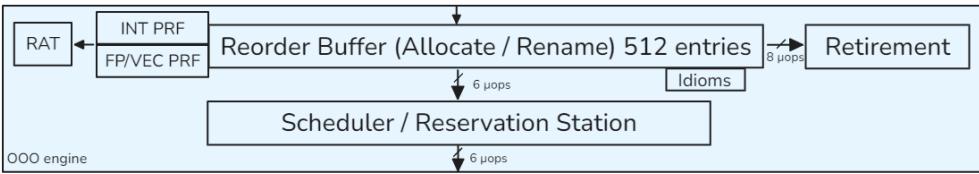


Figure 3.15: Block diagram of the CPU Backend of the Intel Golden Cove Microarchitecture.

Second, the ROB allocates execution resources. When an instruction enters the ROB, a new entry is allocated and resources are assigned to it, mainly an execution unit and the destination physical register. The ROB can allocate up to $6 \mu\text{ops}$ per cycle.

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

There are certain operations that 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 than 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 EAX, EAX`, which are preferred by compilers instead of the equivalent `MOV EAX, 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 frontend, which saves execution resources. The instruction later retires as usual.
- **Move elimination:** similar to the previous one, register-to-register `mov` operations, e.g., `MOV EAX, EBX`, 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 to the reservation station.
- **Other bypasses:** CPU architects also optimized certain arithmetic operations. For example, multiplying any number by one will always yield the same number. The same goes for dividing any number by one. Multiplying any number by zero always yields zero, etc. Some CPUs can recognize such cases at runtime and execute 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 μop and dispatches the μop to an *execution port* once it is ready. An execution port is a pathway that connects the scheduler to its execution units. Each execution port may be connected to multiple execution units. When an instruction enters the RS, the scheduler starts tracking its data dependencies. Once all the source operands become available, the RS attempts to dispatch the μop to a free execution port. The RS has fewer entries⁵⁵ than the ROB. It can dispatch up to $6 \mu\text{ops}$ per cycle.

⁵⁵ People have measured ~200 entries in the RS, however the actual number of entries is not disclosed.

I repeated a part of the diagram that depicts the Golden Cove execution engine and Load-Store unit in Figure 3.16. There are 12 execution ports:

- Ports 0, 1, 5, 6, and 10 provide integer (INT) operations, and some of them handle floating-point and vector (FP/VEC) 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.

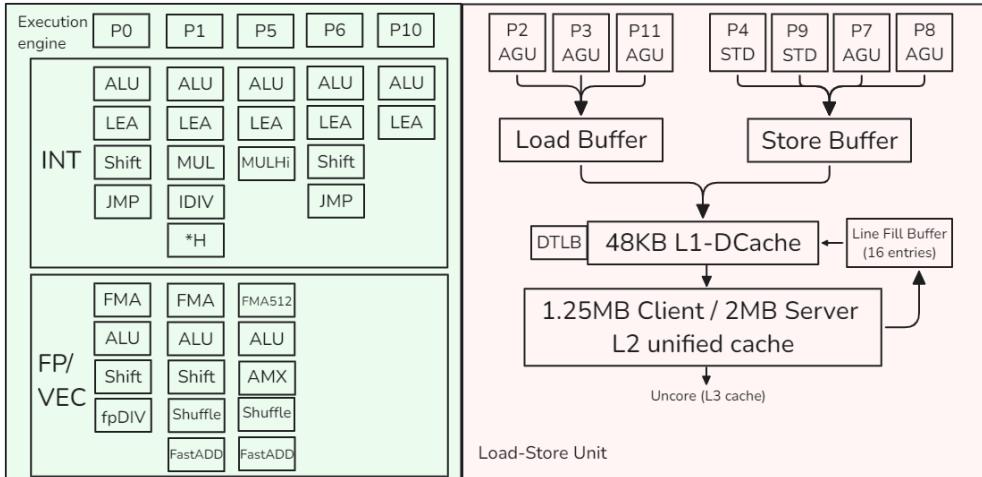


Figure 3.16: Block diagram of the execution engine and the Load-Store unit in the Intel Golden Cove Microarchitecture.

Instructions that require memory operations are handled by the Load-Store unit (ports 2, 3, 11, 4, 9, 7, and 8) which we will discuss in the next section. If an operation does not involve loading or storing data, then it will be dispatched to the execution engine (ports 0, 1, 5, 6, and 10). Some instructions may require two μ ops that must be executed on different execution ports, e.g., load and add.

For example, an **Integer Shift** operation can go only to either port 0 or 6, while a **Floating-Point Divide** operation can only be dispatched to port 0. In a situation when a scheduler has to dispatch two operations that require the same execution port, one of them will have to be delayed.

The FP/VEC stack does floating-point scalar and *all* packed (SIMD) operations. For instance, ports 0, 1, and 5 can handle ALU operations of the following types: packed integer, packed floating-point, and scalar floating-point. Integer and Vector/FP register files are located separately. Operations that move values from the INT stack to FP/VEC and vice-versa (e.g., convert, extract, or insert) incur additional penalties.

3.8.3 Load-Store Unit

The Load-Store Unit (LSU) is responsible for operations with memory. The Golden Cove core can issue up to three loads (three 256-bit or two 512-bit) by using ports 2, 3, and 11. AGU stands for Address Generation Unit, which is required to access a

memory location. It can also issue up to two stores (two 256-bit or one 512-bit) per cycle via ports 4, 9, 7, and 8. STD stands for Store Data.

Notice that the AGU is required for both load and store operations to perform dynamic address calculation. For example, in the instruction `vmovss DWORD PTR [rsi+0x4], xmm0`, the AGU will be responsible for calculating `rsi+0x4`, which will be used to store data from `xmm0`.

Once a load or a store leaves the scheduler, the LSU is responsible for accessing the data. Load operations save the fetched value in a register. Store operations transfer value from a register to a location in memory. LSU has a Load Buffer (also known as Load Queue) and a Store Buffer (also known as Store Queue); their sizes are not disclosed.⁵⁶ Both Load Buffer and Store Buffer receive operations at dispatch from the scheduler.

When a memory load request comes, the LSU 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 the L1 D-cache is 48KB. If both operations result in a hit, the load delivers data to the integer or floating-point register and leaves the Load Buffer. Similarly, a store would write the data to the data cache and exit the Store Buffer.

In case of an L1 miss, the hardware initiates a query of the (private) L2 cache tags. While the L2 cache is being queried, a 64-byte wide fill buffer (FB) entry is allocated, which will keep the cache line once it arrives. The Golden Cove core has 16 fill buffers. As a way to lower the latency, a speculative query is sent to the L3 cache in parallel with the L2 cache lookup. Also, 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.

In case the L2 miss is confirmed, the load continues to wait for the results of the L3 cache, which incurs much higher latency. From that point, the request leaves the core and enters the *uncore*, the term you may sometimes see in profiling tools. The outstanding misses from the core are tracked in the Super Queue (SQ, not shown on the diagram), 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 modifies 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 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 (also known as *streaming stores*) that modify an 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 an entire cache line. If this succeeds no memory read operation is needed.

Second, write combining enables multiple stores to be assembled and written further

⁵⁶ Load Buffer and Store Buffer sizes are not disclosed, but people have measured 192 and 114 entries respectively.

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. All these optimizations are done inside the Store Buffer. 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). Also, the LSU supports store-to-load forwarding when there is an older store containing all of the load's bytes, and the store's data has been produced and is available in the store queue.

Finally, there are cases when we can improve cache utilization by using so-called *non-temporal* memory accesses. If we execute a partial store (e.g., we overwrite 8 bytes in a cache line), we need to read the cache line first. This new cache line will displace another line in the cache. However, if we know that we won't need this data again, then it would be better not to allocate space in the cache for that line. Non-temporal memory accesses are special CPU instructions that do not keep the fetched line in the cache and drop it immediately after use.

During a typical program execution, there could be dozens of memory accesses in flight. In most high-performance processors, the order of load and store operations is not necessarily required to be the same as the program order, which is known as a *weakly ordered memory model*. For optimization purposes, the processor can reorder memory read and write operations. Consider a situation when a load runs into a cache miss and has to wait until the data comes from memory. The processor allows subsequent loads to proceed ahead of the load that is waiting for data. This allows later loads to finish before the earlier load and doesn't unnecessarily block the execution. Such load/store reordering enables memory units to process multiple memory accesses in parallel, which translates directly into higher performance.

The LSU dynamically reorders operations, supporting both loads bypassing older loads and loads bypassing older non-conflicting stores. However, there are a few exceptions. Just like with dependencies through regular arithmetic instructions, there are memory dependencies through loads and stores. In other words, a load can depend on an earlier store and vice-versa. First of all, stores cannot be reordered with older loads:

```
Load R1, MEM_LOC_X
Store MEM_LOC_X, 0
```

If we allow the store to go before the load, then the R1 register may read the wrong value from the memory location `MEM_LOC_X`.

Another interesting situation happens when a load consumes data from an earlier store:

```
Store MEM_LOC, 0
Load R1, MEM_LOC
```

If a load consumes data from a store that hasn't yet finished, we should not allow the load to proceed. But what if we don't yet know the address of the store? In this case, the processor predicts whether there will be any potential data forwarding between the load and the store and if reordering is safe. This is known as *memory*

disambiguation. When a load starts executing, it has to be checked against all older stores for potential store forwarding. There are four possible scenarios:

- Prediction: Not dependent; Outcome: Not dependent. This is a case of a successful memory disambiguation, which yields optimal performance.
- Prediction: Dependent; Outcome: Not dependent. In this case, the processor was overly conservative and did not let the load go ahead of the store. This is a missed opportunity for performance optimization.
- Prediction: Not dependent; Outcome: Dependent. This is a *memory order violation*. Similar to the case of a branch misprediction, the processor has to flush the pipeline, roll back the execution, and start over. It is very costly.
- Prediction: Dependent; Outcome: Dependent. There is a memory dependency between the load and the store, and the processor predicted it correctly. No missed opportunities.

It's worth mentioning that forwarding from a store to a load occurs in real code quite often. In particular, any code that uses read-modify-write accesses to its data structures is likely to trigger these sorts of problems. Due to the large out-of-order window, the CPU can easily attempt to process multiple read-modify-write sequences at once, so the read of one sequence can occur before the write of the previous sequence is complete. One such example is presented in Section 12.2.

3.8.4 TLB Hierarchy

Recall from Section 3.7.1 that translations from virtual to physical addresses are cached in the TLB. Golden Cove's TLB hierarchy is presented in Figure 3.17. 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 1MB of memory, while L1 DTLB has 96 entries that cover 384 KB.

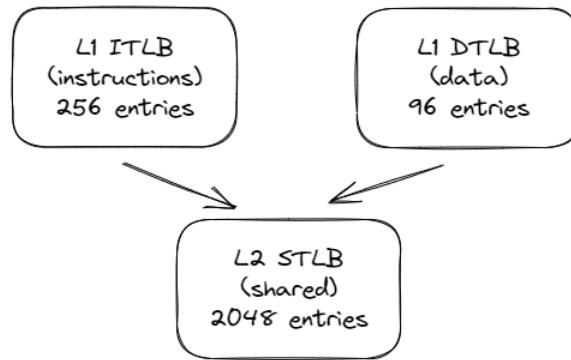


Figure 3.17: TLB hierarchy of Intel's Golden Cove microarchitecture.

The second level of the hierarchy (STLB) caches translations for both instructions and data. It is a larger storage for serving requests that missed in the L1 TLBs. L2 STLB can accommodate 2048 recent data and instruction page address translations, which covers a total of 8MB of memory space. There are fewer entries available for 2MB huge pages: L1 ITLB has 32 entries, L1 DTLB has 32 entries, and L2 STLB can

only use 1024 entries that are also shared with regular 4KB 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. Recall that the page table is built as a radix tree of subtables, with each entry of the subtable 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 Caches⁵⁷ that cache 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 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 levels 1 and 2 of the address (bits 47:30), we only have to do the remaining 2 loads.

The Golden Cove microarchitecture has four dedicated page walkers, which allows it to process 4 page walks simultaneously. In the event of a TLB miss, these hardware 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.

The Golden Cove 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, Load Buffer, and the Store Buffer. PRF is also replicated.

3.9 Performance Monitoring Unit

Every modern CPU provides facilities to monitor performance, which are combined into the Performance Monitoring Unit (PMU). This unit incorporates features that help developers analyze the performance of their applications. An example of a PMU in a modern Intel CPU is provided in Figure 3.18. 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, topics to which Chapter 6 is devoted.

As CPU design evolves with every new generation, so do their PMUs. On Linux, it is possible to determine the version of the PMU in your CPU using the `cpuid` command, as shown in Listing 3.3. Similar information can be extracted from the kernel message buffer by checking the output of the `dmesg` command. Characteristics

⁵⁷ AMD’s equivalent is called Page Walk Caches.

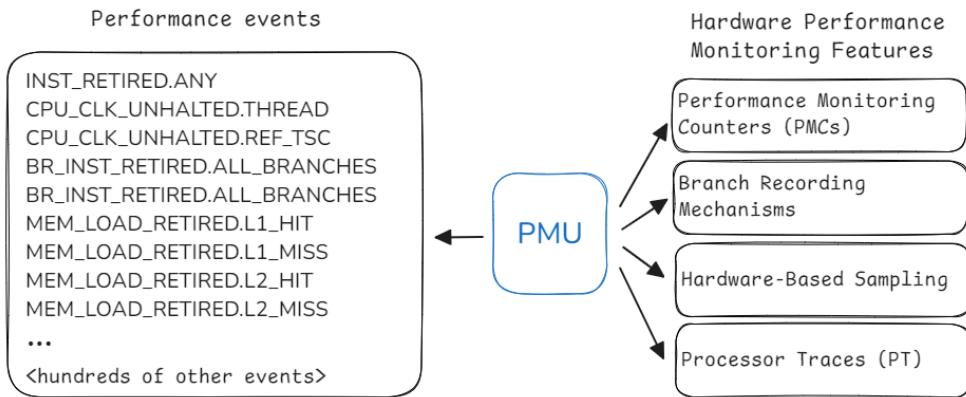


Figure 3.18: Performance Monitoring Unit of a modern Intel CPU.

of each Intel PMU version, as well as changes from the previous version, can be found in [Intel, 2023, Volume 3B, Chapter 20].

Listing 3.3 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 a processor, it may look something like what is shown in Figure 3.19. 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 were executed, how many cache misses or branch mispredictions happened during that time, and other performance events.

Typically, PMCs are 48-bit wide, which enables analysis tools to run for a long time without interrupting a program's execution.⁵⁸ Performance counter is a hardware register implemented as a Model-Specific Register (MSR). That means the number of counters and their width can vary from model to model, and you cannot 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 `RDM MSR` and `WRMSR` instructions,

⁵⁸ When the value of PMCs overflows, the execution of a program must be interrupted. A profiling tool then should save the fact of an overflow. We will discuss it in more detail in Chapter 5.

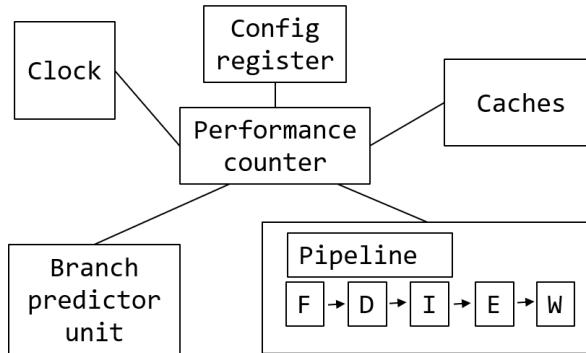


Figure 3.19: Simplified view of a CPU with a performance monitoring counter.

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 very 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 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 3.3), 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, with no fixed counters.

It’s not unusual for the PMU to provide more than one hundred events available for monitoring. Figure 3.18 shows just a small subset of the performance monitoring 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 5.3.1).

- For Intel CPUs, the complete list of performance events can be found in [Intel, 2023, Volume 3B, Chapter 20] or at perfmon-events.intel.com.
- AMD doesn’t publish a list of performance monitoring events for every AMD processor. Curious readers may find some information in the Linux `perf` source code⁵⁹. Also, you can list performance events available for monitoring using the 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 so well defined. Vendors implement

⁵⁹ Linux source code for AMD cores - <https://github.com/torvalds/linux/blob/master/arch/x86/events/amd/core.c>

cores following an ARM architecture, but performance events vary, both in what they mean and what events are supported. For the Arm Neoverse V1 core, that Arm designs themselves, the list of performance events can be found in [Arm, 2022b]. For the Arm Neoverse V2 and V3 microarchitectures, the list of performance events can be found on Arm’s website.⁶⁰

Questions and Exercises

1. Describe pipelining, out-of-order, and speculative execution.
2. How does register renaming help to speed up execution?
3. Describe spatial and temporal locality.
4. What is the size of the cache line in the majority of modern processors?
5. Name the components that constitute 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 the TLB (Translation Lookaside Buffer) play?

Chapter Summary

- Instruction Set Architecture (ISA) is a fundamental contract between software and hardware. ISA is an abstract model of a computer that defines the set of available operations and data types, a 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 encapsulated in the term CPU “microarchitecture”. This topic has been 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 single-threaded performance.
- In parallel with single-threaded performance, hardware designers began pushing multi-threaded performance. The vast majority of modern client-facing devices have a processor containing multiple cores. Some processors double the number of observable CPU cores with the help of Simultaneous Multithreading (SMT). SMT enables multiple software threads to run simultaneously on the same physical core using shared resources. A more recent technique in this direction is called “hybrid” processors which combine different types of cores in a single package to better support a diversity of workloads.
- The memory hierarchy in modern computers includes several levels of cache that reflect different tradeoffs in speed of access vs. size. L1 cache tends to be closest to a core, fast but small. The L3/LLC cache is slower but also bigger. DDR is the predominant DRAM technology used in most platforms. DRAM modules vary in the 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.

⁶⁰ Arm telemetry - <https://developer.arm.com/telemetry>

- Virtual memory is the mechanism for sharing 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, and on ARM is 16KB. Only the page address gets translated, the offset within the page is used as-is. The OS keeps the translation in the page table, which is implemented as a radix tree. There are hardware features that improve the performance of address translation: mainly the Translation Lookaside Buffer (TLB) and hardware page walkers. Also, developers can utilize Huge Pages to mitigate the cost of address translation in some cases (see Section 8.4).
- We looked at the design of Intel’s recent Golden Cove microarchitecture. Logically, the core is split into a Frontend and a Backend. The Frontend consists of a Branch Predictor Unit (BPU), L1 I-cache, instruction fetch and decode logic, and the IDQ, which feeds instructions to the CPU Backend. The Backend consists of the OOO engine, execution units, the load-store unit, the L1 D-cache, and the TLB hierarchy.
- Modern processors have performance monitoring features that are encapsulated into a Performance Monitoring Unit (PMU). This unit is built around a concept of Performance Monitoring Counters (PMC) that enables observation of specific events that happen while a 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` or Intel VTune Profiler. Those tools juggle many complex terms and metrics, however, these metrics are “must-knowns” if you’re set to do any serious performance engineering work.

Since I have mentioned Linux `perf`, let me briefly introduce the tool as I 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. I will use Linux `perf` extensively throughout the book as it is one of the most popular performance analysis tools. Another reason why I prefer showcasing Linux `perf` is because it is open-source software, which enables 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, μ ops, 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 instructions are executed speculatively, as discussed in Section 3.3.3. For most instructions, the CPU commits results once they are available, and all preceding instructions have already been 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 the speculation turns out 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. Some examples of this are NOP, move elimination, and zeroing, as discussed in 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 performance monitoring 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 *μ ops* 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 run:
$ perf stat -- ./a.exe
```

4.2 CPU Utilization

CPU utilization is the percentage of time the core 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 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 13.1, 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:

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

$$IPC = \frac{INST_RETIREDA.NY}{CPU_CLK_UNHALTED.THREAD},$$

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

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

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

Using one or another is a matter of preference. I prefer to use IPC as it is easier to compare. With IPC, we want as many instructions per cycle as possible, so the higher the IPC, the better. With CPI, it's the opposite: we want as few cycles per instruction as possible, so the lower the 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.

The relationship 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 can be calculated as **total cycles / frequency**:

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

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

From the perspective of benchmarking, IPC and frequency are two independent metrics. I've seen some engineers mistakenly mixing them up and thinking that if you increase the frequency, the IPC will also go up. But that's not true. If you clock a processor at 1 GHz instead of 5 GHz, for many applications you will still get the same IPC.⁶¹ It may sound very confusing, especially since IPC has everything to do with CPU clocks. However, frequency only tells us how fast a single clock cycle is, whereas IPC 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 have a 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 hardware architect, they will certainly tell you there is a dependency between IPC and frequency. From the CPU design perspective, you can deliberately downclock the processor, which will make every cycle longer and make it possible to squeeze more work into each cycle. In the end, you will get a higher IPC but a lower frequency. Hardware vendors approach this performance equation in different ways. For example, Intel and AMD chips usually have very high frequencies, with the recent Intel 13900KS processor providing a 6 GHz turbo frequency out of the box with no overclocking required. On the other hand, Apple M1/M2 chips have lower frequency but compensate with a higher IPC. Lower frequency facilitates lower power consumption. Higher IPC, on the other hand, usually requires a more complicated design, more transistors, and a larger die size. We will not go into all the design tradeoffs here, as they are topics for a different book.

IPC is useful for evaluating both hardware and software efficiency. Hardware engineers use this metric to compare CPU generations and CPUs from different vendors. Since

⁶¹ When you lower CPU frequency, memory speed becomes faster relative to the CPU. This may hide actual memory bottlenecks and artificially increase IPC.

IPC is the measure of the performance of a CPU microarchitecture, engineers and media use it to express gains over the previous generation. However, to make a fair comparison, you need to run both systems at the same frequency.

IPC is also a useful metric for evaluating software. It gives you an intuition for how quickly instructions in your application progress through the CPU pipeline. Later in this chapter, you will see several production applications with varying IPCs. Memory-intensive applications usually have a low IPC (0–1), while computationally intensive workloads tend to have a 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 Micro-operations

Microprocessors with the x86 architecture translate complex CISC instructions into simple RISC microoperations, abbreviated as μ ops. A simple register-to-register addition instruction such as `ADD rax, rbx` generates only one μ op, while a more complex instruction like `ADD rax, [mem]` may generate two: one for loading from the `mem` memory location into a temporary (unnamed) register, and one for adding it to the `rax` register. The instruction `ADD [mem], rax` generates three μ ops: one for loading from memory, one for adding, and one for storing the result back to memory.

The main advantage of splitting instructions into micro-operations is that μ ops can be executed:

- **Out of order:** consider the `PUSH rbx` instruction, which 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 decoding:

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

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

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

```
xmm1[63:0] = xmm2[127:64] + xmm1[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 μ ops 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, μ ops can also be fused together. There are two types of fusion in modern x86 CPUs:

- **Microfusion:** fuse μ ops 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 μ ops in this instruction: 1) read the memory location `mem`, and 2) add it to `eax`. With microfusion, two μ ops are fused into one at the decoding step.

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

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

With macrofusion, two μ ops from the DEC and JNZ instructions are fused into one. The Zen4 microarchitecture also added support for DIV/IDIV and NOP macrofusion [Advanced Micro Devices, 2023, sections 2.9.4 and 2.9.5].

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 μ op uses only one entry. Such a fused ROB entry is later dispatched to two different execution ports but is retired again as a single unit. Readers can learn more about μ op fusion in [Fog, 2023a].

To collect the number of issued, executed, and retired μ ops 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, a lower number of μ ops used for an instruction means that hardware has 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 only one μ op. Latency, throughput, port usage, and the number of μ ops for x86 instructions on recent microarchitectures can be found at the uops.info⁶² website.

4.5 Pipeline Slot

Another important metric that some performance tools use is the concept of a *pipeline slot*. A pipeline slot represents the hardware resources needed to process one μ op. Figure 4.1 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

⁶² x86 instruction latency and throughput - <https://uops.info/table.html>

and destination registers, execution port, ROB entries, etc.) to 4 new μ ops 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 (highlighted in yellow). From a microarchitecture perspective, the efficiency of executing such code is only 50%.

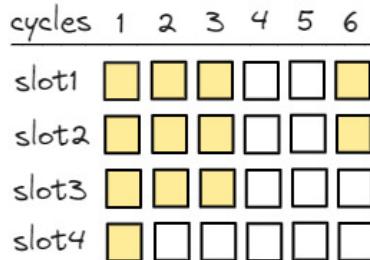


Figure 4.1: Pipeline diagram of a 4-wide CPU.

Intel's Skylake and AMD Zen3 cores have a 4-wide allocation. Intel's Sunny Cove microarchitecture was a 5-wide design. As of the end of 2023, the most recent Golden Cove and Zen4 architectures both have a 6-wide allocation. Apple M1 and M2 designs are 8-wide, and Apple M3 is 9- μ op execution bandwidth see [Apple, 2024, Table 4.10]. The width of a machine puts a cap on the IPC. This means that the maximum achievable IPC of a processor equals its width.⁶³ For example, when your calculations show more than 6 IPC on a Golden Cove core, you should be suspicious.

Very few applications can achieve the maximum IPC of a machine. For example, Intel Golden Cove core can theoretically execute four integer additions/subtractions, plus one load, plus one store (for a total of six instructions) per clock, but an application is highly unlikely to have the appropriate mix of independent instructions adjacent to each other to exploit all that potential parallelism.

Pipeline slot utilization is one of the core metrics in Top-down Microarchitecture Analysis (see Section 6.1). For example, Frontend Bound and Backend Bound metrics are expressed as a percentage of unutilized pipeline slots due to various bottlenecks.

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.

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

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,

⁶³ Although there are some exceptions. For instance, macrofused compare-and-branch instructions only require a single pipeline slot but are counted as two instructions. In some extreme cases, this may cause IPC to be greater than the machine width.

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. The reference clock event counts cycles as if the processor is running at the base frequency. Let's take a look at an experiment on a 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
```

The **ref-cycles** event counts cycles as if there were no frequency scaling. The external clock on the platform 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 the 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 (i.e., 3.4 GHz).

The **cycles** event counts real CPU cycles and takes into account frequency scaling. Using the formula above we can confirm that the average operating frequency was $43340884632 \text{ cycles} / 10.899 \text{ sec} = 3.97 \text{ GHz}$. When you compare the 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 4.1. Performance greatly suffers when a memory request misses in the Last Level Cache (LLC) and goes all the way down to the main memory.⁶⁴

Table 4.1: 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)

Both instruction and data fetches can miss in the cache. According to Top-down Microarchitecture Analysis (see Section 6.1), an instruction cache (I-cache) miss is characterized as a Frontend stall, while a data cache (D-cache) miss is characterized

⁶⁴ There is also an interactive view that visualizes the latency of different operations in modern systems - https://colin-scott.github.io/personal_website/research/interactive_latency.html

as a Backend stall. Instruction cache misses happen very early in the CPU pipeline during instruction fetch. Data cache misses happen 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`), hit the L1 cache (`l1_hit`), or miss both (`l1_miss`), thus $\text{all_loads} = \text{fb_hit} + \text{l1_hit} + \text{l1_miss}$.⁶⁵ 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%. A breakdown for the L3 cache can be made similarly.

4.8 Mispredicted Branch

Modern CPUs try to predict the outcome of a conditional 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 conditional branch. To increase performance, modern processors will try to guess the outcome every time they see a branch instruction. This is called *Speculative Execution* which 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.

A mispredicted branch typically involves a penalty between 10 and 25 clock cycles. First, all the instructions that were fetched and executed based on the incorrect prediction need to be flushed from the pipeline. After that, some buffers may require cleanup to restore the state from where the bad speculation started.

⁶⁵ Careful readers may notice a discrepancy in the numbers: `fb_hit + l1_hit + l1_miss = 545,820`, which doesn't exactly match `all_loads`. Most likely it's due to slight inaccuracy in hardware event collection, but we did not investigate this since the numbers are very close.

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

Being able to collect various performance events is very helpful in performance analysis. However, there is a caveat. Say, you ran a program and collected the `MEM_LOAD_RETIRE.L3_MISS` event, which counts the LLC misses, and it shows you a value of one billion. Sure it sounds like a lot, so you decided to investigate where these cache misses are coming from. Wrong! Are you sure it is an issue? If a program only does two billion loads, then yes, it is a problem as half of the loads miss in the LLC. In contrast, if a program does one trillion loads, then only one in a thousand loads results in a L3 cache miss.

That's why in addition to the hardware performance events, performance engineers frequently use metrics, which are built on top of raw events. Table 4.2 shows a list of metrics for Intel's 12th-gen Golden Cove architecture along with descriptions and formulas. The list is not exhaustive, but it shows the most important metrics. A complete list of metrics for Intel CPUs and their formulas can be found in `TMA_metrics.xlsx`.⁶⁶ Section 4.11 shows how performance metrics can be used in practice.

Table 4.2: A list (not exhaustive) of performance metrics along with descriptions and formulas for the Intel Golden Cove architecture.

Metric Name	Description	Formula
L1MPKI	L1 cache true misses per kilo instruction for retired demand loads.	$\frac{1000 * \text{MEM_LOAD_RETIRE.L1_MISS_PS}}{\text{INST_RETIRE.ANY}}$
L2MPKI	L2 cache true misses per kilo instruction for retired demand loads.	$\frac{1000 * \text{MEM_LOAD_RETIRE.L2_MISS_PS}}{\text{INST_RETIRE.ANY}}$
L3MPKI	L3 cache true misses per kilo instruction for retired demand loads.	$\frac{1000 * \text{MEM_LOAD_RETIRE.L3_MISS_PS}}{\text{INST_RETIRE.ANY}}$
Branch Mispr. Ratio	Ratio of all branches which mispredict	$\frac{\text{BR_MISP_RETIRE.ALL_BRANCHES}}{\text{BR_INST_RETIRE.ALL_BRANCHES}}$

⁶⁶ TMA metrics - https://github.com/intel/perfmon/blob/main/TMA_Metrics.xlsx.

Metric Name	Description	Formula
Code STLB MPKI	STLB (2nd level TLB) code speculative misses per kilo instruction (misses of any page size that complete the page walk)	$\frac{1000 * \text{ITLB_MISSES.WALK_COMPLETED}}{\text{INST_RETIRED.ANY}}$
Load STLB MPKI	STLB data load speculative misses per kilo instruction	$\frac{1000 * \text{DTLB_LD_MISSES.WALK_COMPLETED}}{\text{INST_RETIRED.ANY}}$
Store STLB MPKI	STLB data store speculative misses per kilo instruction	$\frac{1000 * \text{DTLB_ST_MISSES.WALK_COMPLETED}}{\text{INST_RETIRED.ANY}}$
Load Miss Real Latency	Average latency for L1 D-cache miss demand load operations (in core cycles)	$\frac{\text{L1D_PEND_MISS.PENDING}}{\text{MEM_LD_COMPLETED.L1_MISS_ANY}}$
ILP	Instr. level parallelism per core (average number of μ ops executed when there is execution)	$\frac{\text{UOPS_EXECUTED.THREAD}}{\text{UOPS_EXECUTED.CORE_CYCLES_GE1}}$, divide by 2 if SMT is enabled
MLP	Memory level parallelism per-thread (average number of L1 miss demand loads when there is at least one such miss.)	$\frac{\text{L1D_PEND_MISS.PENDING}}{\text{L1D_PEND_MISS.PENDING_CYCLES}}$
DRAM BW Use GB/sec	Average external memory bandwidth use for reads and writes	$\frac{(64 * (\text{UNC_M_CAS_COUNT.RD} + \text{UNC_M_CAS_COUNT.WR})) / 1GB}{\text{Time}}$
IpCall	Instructions per near call (lower number means higher occurrence rate)	$\frac{\text{INST_RETIRED.ANY}}{\text{BR_INST_RETIRED.NEAR_CALL}}$
Ip Branch	Instructions per branch	$\frac{\text{INST_RETIRED.ANY}}{\text{BR_INST_RETIRED.ALL_BRANCHES}}$
IpLoad	Instructions per load	$\frac{\text{INST_RETIRED.ANY}}{\text{MEM_INST_RETIRED.ALL_LOADS_PS}}$
IpStore	Instructions per store	$\frac{\text{INST_RETIRED.ANY}}{\text{MEM_INST_RETIRED.ALL_STORES_PS}}$
IpMispredict	Number of instructions per non-speculative branch misprediction	$\frac{\text{INST_RETIRED.ANY}}{\text{BR_MISP_RETIRED.ALL_BRANCHES}}$

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	$\text{INST_RETIRED.ANY} / \text{FP_ARITH_INST.SCALAR_SINGLE}$
IpArith Scalar DP	Instructions per FP arith. scalar double-precision instruction	$\text{INST_RETIRED.ANY} / \text{FP_ARITH_INST.SCALAR_DOUBLE}$
Ip Arith AVX128	Instructions per arithmetic AVX/SSE 128-bit instruction	$\text{INST_RETIRED.ANY} / (\text{FP_ARITH_INST.128B_PACKED_DOUBLE} + \text{FP_ARITH_INST.128B_PACKED_SINGLE})$
Ip Arith AVX256	Instructions per arithmetic AVX* 256-bit instruction	$\text{INST_RETIRED.ANY} / (\text{FP_ARITH_INST.256B_PACKED_DOUBLE} + \text{FP_ARITH_INST.256B_PACKED_SINGLE})$
Ip SWPF	Instructions per software prefetch instruction (of any type)	$\text{INST_RETIRED.ANY} / \text{SW_PREFETCH_ACCESS.T0:u0xF}$

A few notes on those metrics. First, the ILP and MLP metrics do not represent theoretical maximums for an application; rather they measure the actual ILP and MLP of an application on a given machine. On an ideal machine with infinite resources, these numbers would 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 them in context. For the former, we would like to know if a program saturates the memory bandwidth or not. The latter gives you an idea of the average cost of a cache miss, which is useless by itself unless you know the latencies of each component in the cache hierarchy. We will discuss how to find out cache latencies and peak memory bandwidth in the next section.

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. Table 4.2 provides formulas for the Intel Golden Cove architecture, but you can build similar metrics on another platform as long as underlying performance events are available.

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 the peak performance of the memory subsystem components.

One of the tools that can become helpful on x86 platforms is Intel Memory Latency Checker (MLC),⁶⁷ 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. Examples of such projects are `lmbench`⁶⁸, `bandwidth`⁶⁹ and `Stream`.⁷⁰

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, the access latency increases as there may be more queueing for resources at various points. MLC measures idle latency by doing dependent loads (also known as pointer chasing). A measuring thread allocates a very large buffer and initializes it so that each (64-byte) cache line within the buffer contains a pointer to another, but non-adjacent, cache line within the buffer. By appropriately sizing the buffer, we can ensure that almost all the loads are hitting a certain level of the cache or in the main memory.

My system under test is an Intel Alder Lake computer with a Core i7-1260P CPU and 16GB DDR4 @ 2400 MT/s dual-channel memory. The processor has 4P (Performance) hyperthreaded and 8E (Efficient) cores. Every P-core has 48 KB of L1 data cache and 1.25 MB of L2 cache. Every E-core has 32 KB of L1 data cache, and four E-cores form a cluster that has access to a shared 2 MB L2 cache. All cores in the system are backed by an 18 MB L3 cache. If we use a 10 MB buffer, we can be nearly certain that repeated accesses to that buffer would miss in L2 but hit in L3. Here is the example `mlc` command:

```
$ sudo ./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
Each iteration took 31.1 base frequency clocks ( 12.5 ns)
```

The option `--idle_latency` measures read latency without loading the system. Also, 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

⁶⁷ Intel MLC tool - <https://www.intel.com/content/www/us/en/download/736633/intel-memory-latency-checker-intel-mlc.html>

⁶⁸ lmbench - <https://sourceforge.net/projects/lmbench>

⁶⁹ Memory bandwidth benchmark by Zack Smith - <https://zsmith.co/bandwidth.php>

⁷⁰ Stream - <https://github.com/jeffhammond/STREAM>

logical CPU 0, which is on a P-core. The option `-L` enables huge pages to limit TLB effects in our measurements. The option `-b10m` tells MLC to use a 10MB buffer, which will fit in the L3 cache on our system.

Figure 4.2 shows the read latencies of L1, L2, and L3 caches. There are four different regions on the chart. The first region on the left from 1 KB to 48 KB buffer size corresponds to the L1 D-cache, which is private to each physical core. We can observe 0.9 ns latency for the E-core and a slightly higher 1.1 ns for the 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 32 KB but P-core latency stays constant up to 48 KB. That confirms that the L1 D-cache size in the E-core is 32 KB, and in the P-core it is 48 KB.

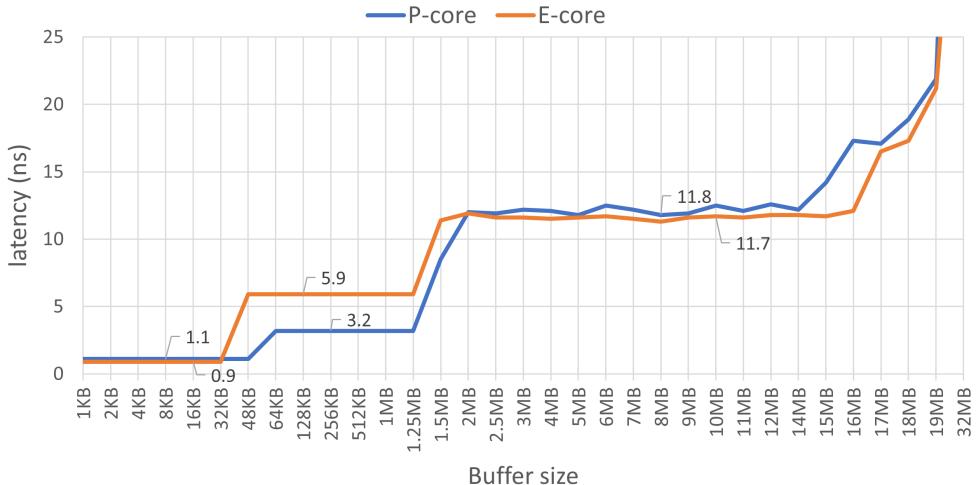


Figure 4.2: L1/L2/L3 cache read latencies (lower better) on Intel Core i7-1260P, measured with the MLC tool, huge pages enabled.

The second region shows the L2 cache latencies, which for E-core is almost two times higher than for P-core (5.9 ns vs. 3.2 ns). For P-core, the latency increases after we cross the 1.25 MB buffer size, which is expected. We expect E-core latency to stay the same until we hit 2 MB, however, according to our measurements, it happens sooner.

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

I don't show the part of the chart that corresponds to memory latency, which begins after we cross the 18MB boundary. The latency starts climbing very steeply and levels off at 24 MB for the E-core and 64 MB for the P-core. With a much larger buffer size, e.g., 500 MB, 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 the 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 the 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 determines whether MLC is measuring L1/L2/L3 cache bandwidth or memory bandwidth.

```
$ sudo ./mlc --max_bandwidth -k0-15 -Y -L -u -b18m
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      :      349670.42
```

There are a couple of new options here. The `-k` option specifies a list of CPU cores used for measurements. The `-Y` option tells MLC to use AVX2 loads, i.e., 32 bytes at a time. The options With the `-u` flag, each thread shares the same buffer and does not allocate its own. This option must be used to measure the L3 bandwidth (notice we used an 18 MB buffer, which equals to the size of the L3 cache).

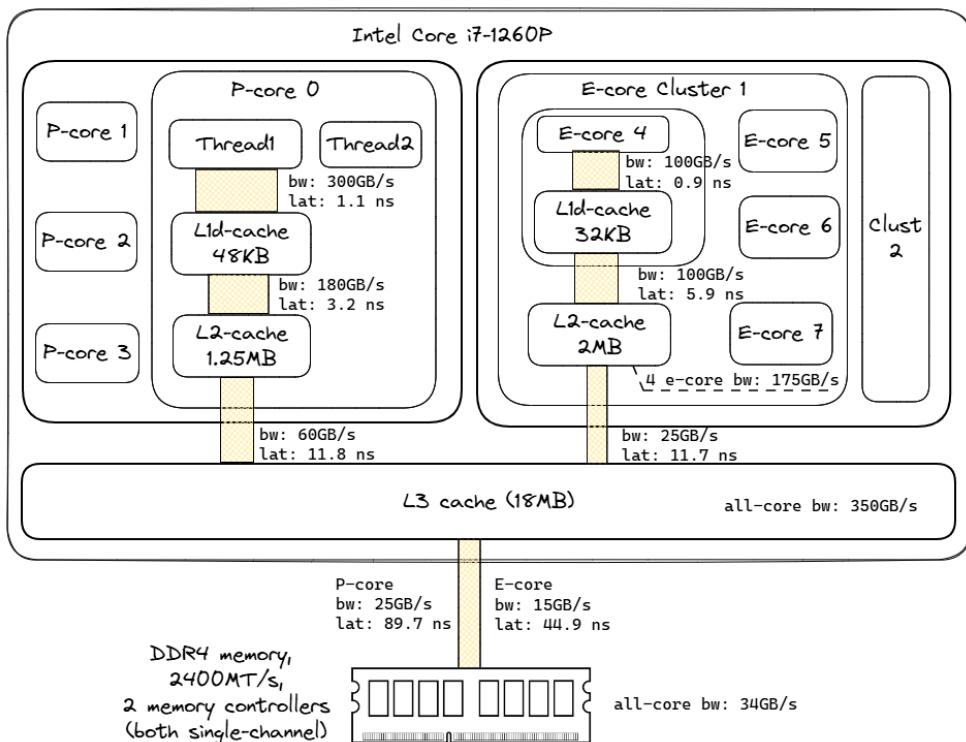


Figure 4.3: Block diagram of the memory hierarchy of Intel Core i7-1260P and external DDR4 memory.

Combined latency and bandwidth numbers for our system under test, as measured with Intel MLC, are shown in Figure 4.3. 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, a 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 the 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, the L3 cache can sustain a bandwidth of 300 GB/s. Reading data from memory can be done at 33.7 GB/s, while the theoretical maximum bandwidth is 38.4 GB/s on my platform.

Knowledge of the primary characteristics of a machine is fundamental to assessing how well a program utilizes available resources. We will return to this topic in Section 5.5 when discussing the Roofline performance model. If you constantly analyze performance on a single platform, it is a good idea to memorize the 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

To put together everything we discussed so far in this chapter, let's look at some real-world examples. We ran four benchmarks from different domains and calculated their performance metrics. First of all, let's introduce the benchmarks.

1. Blender 3.4 - an open-source 3D creation and modeling software project. This test is of Blender's Cycles performance with the BMW27 blend file. All hardware 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 hardware thread is used. URL: <https://stockfishchess.org>. Command line: `./stockfish bench 128 1 24 default depth`.
3. Clang 15 self-build - this test uses Clang 15 to build the Clang 15 compiler from sources. All hardware threads are used. URL: <https://www.llvm.org>. Command line: `ninja -j16 clang`.
4. CloverLeaf 2018 - a Lagrangian-Eulerian hydrodynamics benchmark. All hardware threads are used. This test uses the clover_bm.in input file (Problem 5). URL: <http://uk-mac.github.io/CloverLeaf>. Command line: `./clover_leaf`.

For this exercise, I ran all four benchmarks on the machine with the following characteristics:

- 12th Gen Alder Lake Intel® Core™ 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)
- Clang-15 C++ compiler with the following options: `-O3 -march=core-avx2`

To collect performance metrics, I used the `toplev.py` script from Andi Kleen's `pmu-`

tools:⁷¹

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

Table 4.3 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.

Table 4.3: 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	P-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
IpArith AVX256	All	30.3	0.0	0.0	39.6
IpSWPF	All	90.2	2,565	105,933	172,348

Here are the hypotheses we can make about the performance of the benchmarks:

- **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 presents a minor bottleneck: the Br. Misp. Ratio metric is at 2%; we get 1 misprediction for every 610

⁷¹ pmu-tools - <https://github.com/andikleen/pmu-tools>

instructions (see `IpMispredict` metric), which is quite good. TLB is not a bottleneck as we very rarely miss in STLB. We ignore the `Load Miss Latency` metric since the number of cache misses is very low. The ILP is reasonably high. Golden Cove is a 6-wide architecture; an ILP of 3.67 means that the algorithm utilizes almost 2/3 of the core resources every cycle. Memory bandwidth demand is low (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), a 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 Blender's source code. Preliminary conclusion: Blender's performance is bound by FP compute.

- **Stockfish.** We ran it using only one hardware 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. The branch misprediction ratio is high; we pay the misprediction penalty every 215 instructions. We can estimate that we get one mispredict every $215 \text{ (instructions)} / 1.80 \text{ (IPC)} = 120$ cycles, which is very frequent. 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 are almost no FP operations in the workload (see `IpFLOP` metric). Preliminary 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 that has a very flat performance profile, i.e., there are no big hotspots. You will see that the running time is attributed to many different functions. The 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 don't look troubling at 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 differences with any other benchmark we test. This is due to another aspect of the Clang compiler (and other compilers as well): the size of the binary is relatively big (more than 100 MB). The code constantly jumps to distant places causing high pressure on the TLB subsystem. As you can see the problem exists both for instructions (see `Code st1b MPKI`) and data (see `Ld st1b MPKI`). Let's proceed with our analysis. DRAM bandwidth use is higher than for the two previous benchmarks, but still is not reaching even half of the maximum memory bandwidth on our platform (which is ~34 GB/s). Another concern for us is the very small number of instructions per call (`IpCall`): only ~41 instructions per function call. This is unfortunately the nature of the compiler's codebase: it has thousands of small functions. The compiler needs to be more aggressive with inlining all those functions and wrappers.⁷² Yet, we suspect that the performance overhead associated with making a function call remains an issue for the Clang compiler. Also, one can spot the high `ipBranch` and `IpMispredict` metrics. For Clang compilation, every fifth instruction is a

⁷² Perhaps by using Link Time Optimizations (LTO).

branch and one of every ~35 branches gets mispredicted. There are almost no FP or vector instructions, but this is not surprising. Preliminary conclusion: Clang has a large codebase, flat profile, many small functions, and “branchy” code; performance is affected by data cache misses 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.⁷³ The L*MPKI metrics are high, especially the number of L3 misses per kilo instructions. The load miss latency ($LdMissLat$) is off the 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 consumption is near its limits. That’s the problem: all the cores in the system share the same memory bus, so they compete for access to the main 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. Preliminary conclusion: multi-threaded CloverLeaf is bound by memory bandwidth.

As you can see from this study, there is a lot one can learn about the 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 a performance profile, which we will introduce in later chapters. In Part 2 of this book, we will discuss how to mitigate the performance issues we suspect to exist in the four benchmarks that we have analyzed.

Keep in mind that the summary of performance metrics in Table 4.3 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 an IPC of 0.1, and the second with an 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 4.4 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 the `--xlsx` and `--xchart` options. The `-I 10000` option aggregates collected samples with 10-second intervals.

```
$ ~/workspace/pmu-tools/toplev.py -m --global --no-desc -v --xlsx workload.xlsx
-xchart -I 10000 -- ./clover_leaf
```

Even though the deviation from the average values reported in the summary is not very big, we can see that the workload is not stable. After looking at the IPC chart

⁷³ A possible explanation for that is because CloverLeaf is very memory-bandwidth bound. All P- and E-cores are equally stalled waiting on memory. Because P-cores have a higher frequency, they waste more CPU clocks than E-cores.

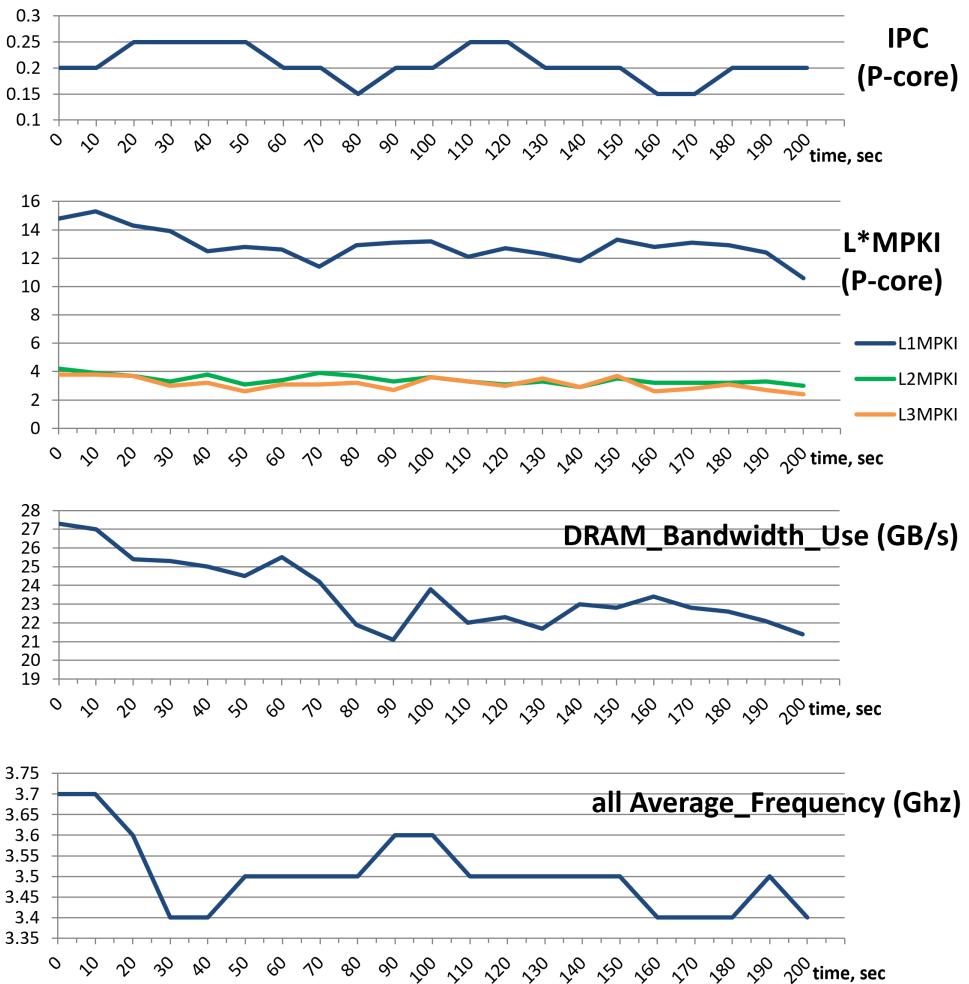


Figure 4.4: Performance metrics charts for the CloverLeaf benchmark with 10 second intervals.

for P-core we can hypothesize that there are no distinct 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 was 10 seconds). The chart that plots L*MPKI suggests that all three metrics hover around their average numbers without much deviation. The DRAM bandwidth utilization chart indicates that there are periods with varying pressure on the main memory. The last chart shows the average frequency of all CPU cores. As you may observe on this chart, throttling starts after the first 10 seconds. I recommend being careful when drawing conclusions just from looking at the aggregate numbers since they may not be a good representation of the workload behavior.

Remember that collecting performance metrics is not a substitute for looking into the code. Always try to find explanation for the numbers that you see by checking relevant parts of the code.

In summary, performance metrics help you build the right mental model about what is and what is *not* happening in a program. Going further into analysis, these data will serve you well.

Questions and Exercises

1. What is the difference between the CPU core clock and the reference clock?
2. What is the difference between retired and executed instructions?
3. When you increase the frequency, does IPC go up, down, or stay the same?
4. Take a look at the DRAM BW Use formula in Table 4.2. Why do you think there is a constant 64?
5. Measure the bandwidth and latency of the cache hierarchy and memory on the machine you use for development/benchmarking using Intel MLC, Stream, or other tools.
6. Run the application that you're working with on a daily basis. Collect performance metrics. Does anything surprise you?

Capacity Planning Exercise: 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 being maximum performance (throughput). The 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 choice 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 parameters you will prioritize? Where will you go with the most expensive part? Where you can save money? Try to describe it in as much detail as possible, and search the web for exact components and their prices. Account for all the components of the system: motherboard, disk drive, cooling solution, power delivery unit, rack/case/tower, etc. What additional performance experiments would you 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, μ ops, 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 you can collect. For instance, cache misses per kilo instructions (MPKI), instructions per function call, branch, load, etc. (I_p^*), ILP, MLP, and others.

The case studies in this chapter show how you can get actionable insights from analyzing these metrics.

- Be careful about drawing conclusions just by looking at the aggregate numbers. Don't fall into the trap of "Excel performance engineering", i.e., only collecting performance metrics and never looking at the code. Always seek a second source of data (e.g., performance profiles, discussed later) to verify your ideas.
- Memory bandwidth and latency are crucial factors in the performance of many production software 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. The majority of modern client-facing systems fall in the range of 70–110 ns latency per memory access. Server platforms may have higher memory latencies.

5 Performance Analysis Approaches

When you're working on a high-level optimization, e.g., integrating a better algorithm into an application, it is usually easy to tell whether the performance improves or not since the benchmarking results are usually pronounced. Big speedups, like 2x, 3x, etc., are relatively obvious from a 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 for 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 work. To solve a performance mystery, you need to gather all the data that you can 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 pieces of evidence as possible to confirm or refute 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 those measurements you conclude that the program became slower by X percent. If you know that the slowdown occurred right after a certain commit, that may already give you enough information to fix the problem. But if you don't have good reference points, then the set of possible reasons for the slowdown is endless, and 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 have proven to be useful in performance engineering.

The next question comes: "What performance data are available and how to collect them?" Both hardware and software layers of the stack have facilities to track performance events and record them while a program is running. In this context, by hardware, we mean the CPU, which executes the program, and by software, we mean the OS, libraries, the application itself, and other tools used for the analysis. Typically, the software stack provides high-level metrics like time, number of context switches, and page faults, while the 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 hardware metrics will always give us a more precise overview of the program execution. They are just different. 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. This chapter is mostly about collecting hardware-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 that 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 5.1 shows the simplest example of inserting a `printf` statement at the beginning of a function to indicate if this function is called. After that, you run the program and count the number of times you see “foo is called” in the output. Perhaps every programmer in the world did this at some point in their career at least once.

Listing 5.1 Code Instrumentation

```
int foo(int x) {  
+ printf("foo is called\n");  
// 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; rather, it’s for collecting the needed data and later can be deleted.

A more interesting example of code instrumentation is presented in Listing 5.2. In this made-up code example, the function `findObject` searches for the coordinates of an object with some properties `p` on a map. All objects are guaranteed to eventually be located. The function `getNewCoords` returns new coordinates within a bigger area that is provided as an argument. 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 call `zoomIn` to find a more precise location of the object. Otherwise, we get the new coordinates within the `searchArea` to try our search next time.

The 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 occurrence 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 and can be adjusted to your specific needs quickly.⁷⁴

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

⁷⁴ I have a slightly more advanced version of this code which I usually copy-paste into whatever project I’m working on, and later delete.

Listing 5.2 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 searchArea) {
+ incrementor inc;
  Coords c = getNewCoords(searchArea);
  while (true) {
+   inc.tripCount++;
    float match = findObj(p, c);
    if (exactMatch(match))
      return c;
    if (match > threshold) {
      searchArea = zoomIn(searchArea, c);
+      inc.zoomCount++;
    } else {
      c = getNewCoords(searchArea);
    }
  }
  return c;
}

```

how many times we reduce the search area (call to `zoomIn`). We always expect `inc.zoomCount` to be less or equal to `inc.tripCount`.

The `findObject` function is called many times with various inputs. Here is a possible output we may observe after running the instrumented program:

```
// [tripCount][zoomCount]: occurrences
[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
...
```

The first number in the square bracket is the trip count of the loop, and the second is the number of `zoomIns` we made within the same loop. The number after the column sign is the number of occurrences of that particular combination of the numbers. For example, two times we observed 7 loop iterations and 6 `zoomIns`. 251004 times the loop ran 7 iterations and no `zoomIns`, and so on. You can then plot the data for better visualization, or employ some other statistical methods, but the main point we can make is that `zoomIns` are not frequent. The total number of calls to `findObject` is approximately 400k; we can calculate it by summing up all the buckets in the histogram. If we sum up all the buckets with a non-zero `zoomCount`, we get approximately 10k; this is the number of times the `zoomIn` function was called. So, for every `zoomIn` call, we make 40 calls of the `findObject` function.

Later chapters of this book contain many examples of how such information can be used for optimizations. In our case, we conclude that `findObj` 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 area. Knowing that, we could attempt a number of optimizations: 1) run multiple searches in parallel, and synchronize if any of them succeeded; 2) precompute certain things for the current search region, thus eliminating repetitive work inside `findObj`; 3) write a software pipeline that calls `getNewCoords` to generate the next set of required coordinates and prefetch the corresponding map locations from memory. Part 2 of this book looks more deeply into some of these techniques.

Code instrumentation provides very detailed information when you need specific knowledge about the execution of a program. It allows us to track any information about every variable in a 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 and then drilling down to its callees) to better understand the behavior of an application. Code instrumentation enables developers to observe the architecture and flow of an application. This technique is especially helpful for someone working with an unfamiliar codebase.

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

While code instrumentation is powerful in many cases, it does not provide any information about how 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 of 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.

A more important 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 and increase analysis time. Unfortunately, there are additional downsides. Since you usually 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 an instrumented program. 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 the 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 an entire program (except third-party libraries) to collect interesting statistics about the execution. The most widely known use cases for automated instrumentation are code coverage analysis and Profile-Guided Optimization (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 rather than on source code. There are two types of binary instrumentation: static (done ahead of time) and dynamic (instrumented code is 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, instead of instrumenting the entire program.

Binary instrumentation is very useful in performance analysis and debugging. One of the most popular tools for binary instrumentation is the Intel Pin⁷⁵ tool. Pin intercepts the execution of a program at the occurrence of an interesting event and generates new instrumented code starting at this point in the program. This enables the collection of various runtime information. One of the most popular tools that is built on top of Pin is Intel SDE (Software Development Emulator).⁷⁶ Another well-known binary instrumentation tool is called DynamoRIO.⁷⁷ Here are some of the things you can collect using a binary instrumentation tool:

- instruction count and function call counts,
- instruction mix analysis,
- intercepting function calls and execution of any instruction in an application,
- memory intensity and footprint (see Section 7.8.3).

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

⁷⁵ Pin - <https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool>

⁷⁶ Intel SDE - <https://www.intel.com/content/www/us/en/developer/articles/tool/software-development-emulator.html>

⁷⁷ DynamoRIO - <https://github.com/DynamoRIO/dynamorio>. It supports Linux and Windows operating systems, and runs on x86 and ARM hardware.

5.2 Tracing

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

An example of tracing system calls with the Linux `strace` tool is provided in Listing 5.3, 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 (after the = sign), and the duration of each system call (in angle brackets).

Listing 5.3 Tracing system calls with strace.

```
$ strace -tt -T -- git status
17:46:16.798861 execve("/usr/bin/git", ["git", "status"], 0x7ffe705dc78
                           /* 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 depends on what exactly we try to trace. For example, if we trace a program that rarely 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.⁷⁸ 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.

Similar to instrumentation, tracing can be used for exploring anomalies in a 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 leads to the program being

⁷⁸ An article about `strace` by B. Gregg - <http://www.brendangregg.com/blog/2014-05-11/strace-wow-much-syscall.html>

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 rr⁷⁹ debugger, which performs record and replay of processes, supports backward single stepping, and much more. Most tracing tools are capable of decorating events with timestamps, which enables us to find correlations 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 Collecting Performance Monitoring Events

Performance Monitoring Counters (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” or “Sampling”. The counting mode is primarily used for calculating various performance metrics that we discussed in Section 4.9. The sampling mode is used for finding hotspots, which we will discuss soon.

The idea behind counting is very simple: we want to count the total number of certain performance monitoring events while our program is running. PMCs are heavily used in the Top-down Microarchitecture Analysis (TMA) methodology, which we will closely look at in Section 6.1. Figure 5.1 illustrates the process of counting performance events from the start of a program to the end of a program.

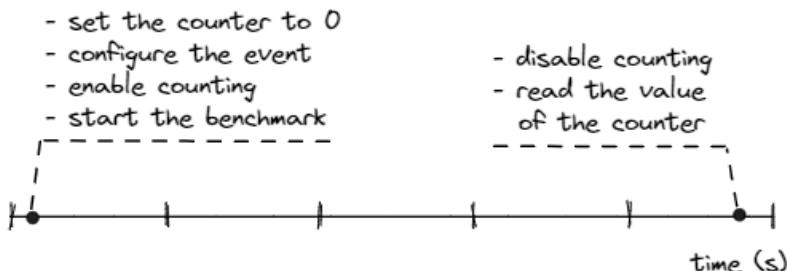


Figure 5.1: Counting performance events.

The steps outlined in Figure 5.1 roughly represent what a typical analysis tool will do to count performance events. A similar process is implemented in the `perf stat` tool, which can be used to count various hardware events, like the number of instructions, cycles, cache misses, etc. Below is an example of the output from `perf stat`:

```
$ perf stat -- ./my_program.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
```

⁷⁹ Mozilla rr debugger - <https://rr-project.org/>.

This data may become quite handy. First of all, it enables us to quickly spot some anomalies, such as 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 relevant events might help you justify or reject the code change. The `perf stat` utility can be used as a lightweight benchmark wrapper. It may serve as a first step in performance investigation. Sometimes anomalies can be spotted right away, which can save you some analysis time.

A full list of available event names can be viewed with `perf list`:

```
$ perf list
cycles          [Hardware event]
ref-cycles      [Hardware event]
instructions    [Hardware event]
branches        [Hardware event]
branch-misses   [Hardware event]
...
cache:
mem_load_retired.l1_hit
mem_load_retired.l1_miss
...
```

Modern CPUs have hundreds of observable performance events. It's very hard to remember all of them and their meanings. Understanding when to use a particular event is even harder. That is why generally, I don't recommend manually collecting a specific event unless you really know what you are doing. Instead, I recommend using tools like Intel VTune Profiler that automatically collect required events to calculate various metrics.

Performance events 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 monitoring 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 enable⁸⁰ virtual Performance Monitoring Counters (vPMC). The AWS EC2 cloud has also enabled⁸¹ PMCs for dedicated hosts.

5.3.1 Multiplexing and Scaling Events

There are situations when we want to count many different events at the same time. However, with one counter, it's possible to count only one event at a time. That's why PMUs contain multiple counters (in Intel's recent Golden Cove microarchitecture there are 12 programmable PMCs, 6 per hardware thread). Even then, the number of fixed and programmable counters 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.

⁸⁰ VMware PMCs - <https://www.vladan.fr/what-are-vmware-virtual-cpu-performance-monitoring-counters-vpmcs/>

⁸¹ Amazon EC2 PMCs - <http://www.brendangregg.com/blog/2017-05-04/the-pmcbs-of-ec2.html>

If you need to collect more events than the number of available PMCs, the analysis tool uses time multiplexing to give each event a chance to access the monitoring hardware. Figure 5.2a shows an example of multiplexing between 8 performance events with only 4 counters available.

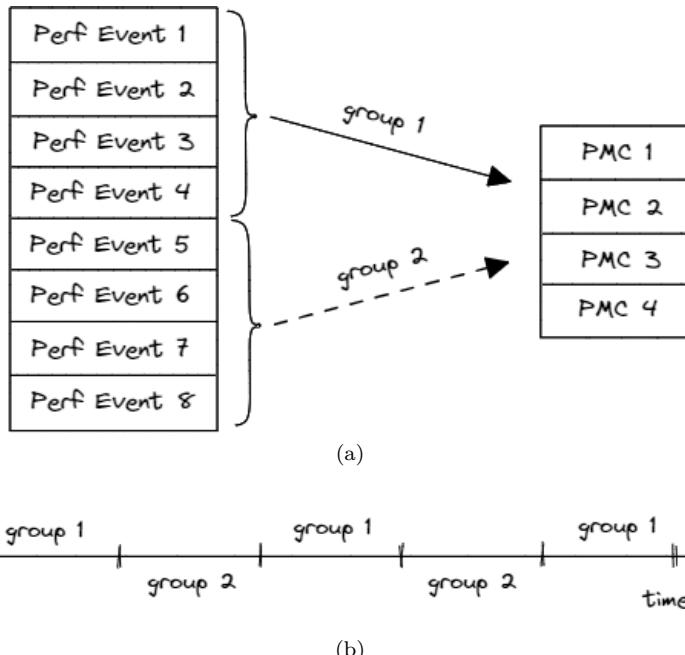


Figure 5.2: Multiplexing between 8 performance events with only 4 PMCs available.

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 the total time enabled:

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

Let's take Figure 5.2b 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 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:

$$\text{final count} = 10,000 \times (500ms/(100ms \times 3)) = 16,666$$

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, you can reduce the number of events to no more than the number of physical PMCs available. However, you'll have to run the benchmark multiple times to measure all the events.

5.3.2 Using Marker APIs

In certain scenarios, we might be interested in analyzing the 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 two examples:

- Intel VTune has `--itt_task_begin` / `--itt_task_end` functions.
- AMD uProf has `amdProfileResume` / `amdProfilePause` functions.

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

Below we provide a very basic example of using `libpfm4`,⁸² 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 event counters directly. The `perf_events` subsystem is rather low-level, so the `libpfm4` 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. The code listing below shows how you can use `libpfm4` to instrument the `render` function of the C-Ray⁸³ 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);
```

⁸² libpfm4 - <https://sourceforge.net/p/perfmon2/libpfm4/ci/master/tree/>

⁸³ C-Ray benchmark - <https://openbenchmarking.org/test/pts/c-ray>

```

+ 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
...
}

```

In this code example, we first initialize the libpfm library and then 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 careful 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 Alder Lake machine, we get the output shown below. Root privileges are not required, but `/proc/sys/kernel/perf_event_paranoia` should be set to less than 1. When reading counters for a thread, the values are for that thread alone. It can optionally include kernel code that 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 our instrumentation 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 measurements, 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 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 a hypothesis by studying the source code or extending the instrumentation to capture more data for the “slow” pixels.

The additional instrumentation code in our example 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 a 17% slowdown. Managing the overhead of your instrumentation is critical, especially if you choose to enable it in a production environment.

Overhead is usefully calculated as the occurrence rate per unit of time or work (RPC, database query, loop iteration, etc.). If a `read` system call on our system takes roughly 1.6 microseconds of CPU time, and we call 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. It will otherwise interfere with the measurements. The instrumentation 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, i.e., randomly select which RPCs to monitor for each sample. 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 collect, process, and retain only as 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 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 implementation⁸⁴ uses less than 50 bytes of memory.

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

⁸⁴ Accurately computing running variance - https://www.johndcook.com/blog/standard_deviation/

In our example, 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 both indicate duration in nanoseconds. The `time_enabled` field indicates how many nanoseconds the event group has been enabled. The `time_running` field indicates how much of that enabled time the events have been collected. If you had two event groups enabled simultaneously that couldn't fit together on the hardware counters, you might see the running time for both groups converge to `time_running = 0.5 * time_enabled`.

Capturing multiple events simultaneously makes it possible 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`) with 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 it against wall-clock time. Also, we can normalize the numbers to get the event rate per second/clock/instruction. For instance, by measuring `MEM_LOAD_RETIRED.L3_MISS` and `INSTRUCTIONS_RETIRED` we can get the L3MPKI metric. As you can see, this gives a lot of flexibility.

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. For example, let's assume 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 this event contributes to the low IPC that we're dealing with. If it doesn't, we can 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 a decrease in IPC (e.g., when there is a busy sibling hardware thread running) or decreasing CPU frequency (e.g., system throttling due to heavy load). When such an event occurs, the 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 a program. To put it more broadly, sampling helps to find places in the code that contribute to the highest number of certain performance events. If we want to find hotspots, the problem can be reformulated as: "find a place in the code that consumes the biggest number of CPU cycles". People often use the term *profiling* for what is technically called *sampling*. According to Wikipedia,⁸⁵ profiling is a much broader term and includes a wide variety

⁸⁵ Profiling(wikipedia) - [https://en.wikipedia.org/wiki/Profiling_\(computer_programming\)](https://en.wikipedia.org/wiki/Profiling_(computer_programming)).

of techniques to collect data, including sampling, code instrumentation, tracing, and others.

It may come as a surprise, but the simplest sampling profiler one can imagine is a debugger. In fact, you can identify hotspots as follows: 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 collected 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 of the hottest places in a program.

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 an interrupt, the profiler collects the snapshot of the program state, which constitutes one sample. Information collected for every sample may include an instruction address that was executed at the time of the 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 the 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 hardware event-based sampling (EBS). User-mode sampling is a pure software approach that embeds an agent library into the profiled application. The agent sets up an OS timer for each thread in the application. Upon timer expiration, the application receives the `SIGPROF` signal that is handled by the agent. 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, Top-down Microarchitecture Analysis (see Section 6.1), etc.

User-mode sampling incurs higher runtime overhead than EBS. The average overhead of the user-mode sampling is about 5% when sampling with an 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 5.3 illustrates the counter overflow feature of the PMU, which is used to trigger a Performance Monitoring Interrupt (PMI), also known as `SIGPROF`. At the start of a benchmark, we configure the event that we want to sample. Sampling on cycles is a default for many profiling tools since we want to know where the program spends most of the time. However, it is 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_RETIRE.L3_MISS`.

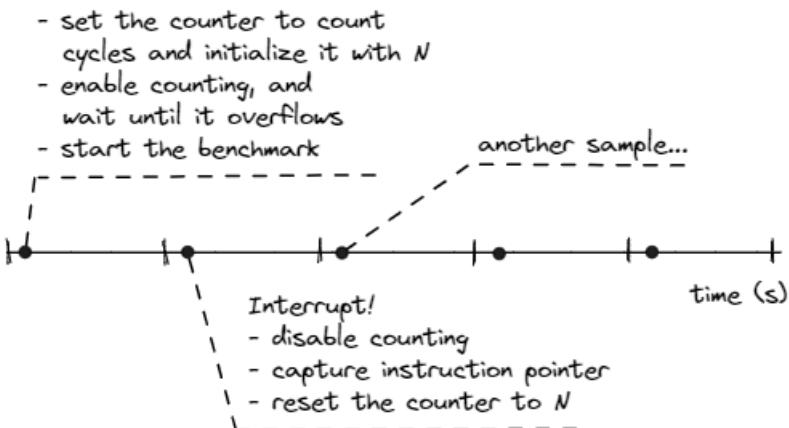


Figure 5.3: Using performance counter for sampling

After we have initialized the register, we start counting and let the benchmark run. Since we have configured a PMC to count cycles, it will be incremented every cycle. Eventually, it will overflow. At the time the register overflows, the hardware 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 the ISR: first of all, we disable counting; after that, we record the instruction that 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 cycles. To achieve this, we can set the counter to `(unsigned) -1,000,000` so that it will overflow after every 1 million cycles. 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 in 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`⁸⁶ benchmark from the [Phoronix test suite](#)⁸⁷ 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
```

⁸⁶ x264 benchmark - <https://openbenchmarking.org/test/pts/x264>.

⁸⁷ Phoronix test suite - <https://www.phoronix-test-suite.com/>.

#	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_mm2	
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 there were the same number of process interrupts. 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, that 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 4500000 to collect 1000 samples per second. The number N can be adjusted by Linux `perf` dynamically according to the actual CPU frequency.

And of course, 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).

Linux `perf` doesn't have rich graphic support, so viewing hot parts of source code is not very convenient, 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];
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 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 detail in Chapter 7.

Sampling gives a good statistical representation of a program's execution, however, one of the downsides of this technique is that it has blind spots and is not suitable for detecting abnormal behaviors. Each sample represents an aggregated view of a

portion of a program's execution. Aggregation doesn't give us enough details of what exactly happened during that time interval. We cannot zoom in to learn more about execution nuances. When we squash time intervals into samples, we lose valuable information and it becomes useless for analyzing events with a very short duration. For instance, profiling a program that reacts to network packets (such as stock trading software) may not be very informative as it will attribute most samples to the busy wait loop. Increasing the sampling interval, e.g., more than 1000 samples per second may give you a better picture, but may still not be enough. As a solution, you should use tracing as it doesn't skip events of interest.

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 5.4. 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 is responsible for it.

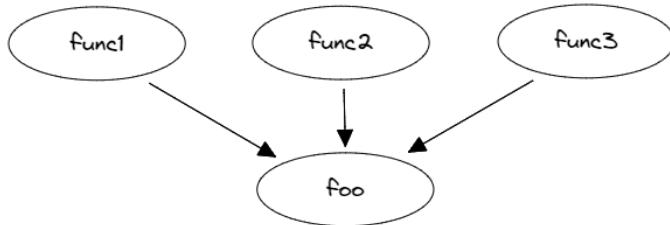


Figure 5.4: Control Flow Graph: hot function “foo” has multiple callers.

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`). It requires that the binary be built with `--fno-omit-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 (also known as *stack unwinding*). The frame pointer can tell the return address immediately. It enables very cheap stack unwinding, which reduces profiling overhead, however, it consumes one additional register just for this purpose. At the time when the number of architectural registers was small, using frame pointers was expensive in terms of runtime performance. Nowadays, the Linux community is moving back to using

frame pointers, because it provides better quality call stacks and low profiling overhead.

2. DWARF debug info (`perf record --call-graph dwarf`). It requires that the binary be built with DWARF debug information (-g). It also obtains call stacks through the stack unwinding procedure, but this method is more expensive than using frame pointers.
3. Intel Last Branch Record (LBR). This method makes use of a hardware feature, and is accessed with the following command: `perf record --call-graph lbr`. It obtains call stacks by parsing the LBR stack (a set of hardware registers). The resulting call graph is not as deep as those produced by the first two methods. See more information about the LBR call-stack mode in Section 6.2.

Below is an 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           [.]
|              |
--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, you 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.

5.5 The 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 [Williams et al., 2009]. The term “roofline” in this model expresses the fact that the performance of an application cannot exceed the capabilities of a machine. Every function and every loop in a program is limited by either the

computing or memory bandwidth capacity of a machine. This concept is represented in Figure 5.5. The performance of an application will always be limited by a certain “roofline” function.

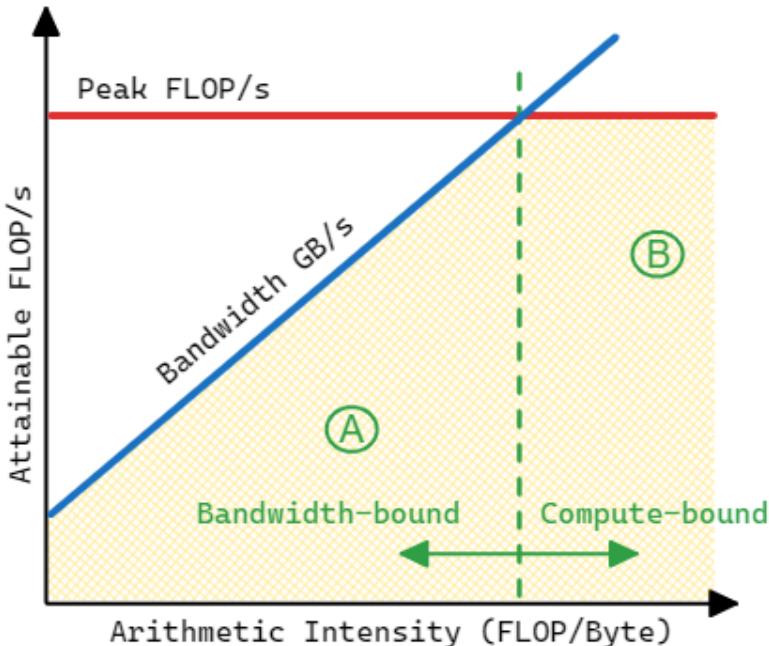


Figure 5.5: The Roofline Performance Model. The maximum performance of an application is limited by the minimum between peak FLOPS (horizontal line) and the platform bandwidth multiplied by arithmetic intensity (diagonal line).

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 5.5 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. However, keep in mind that the Roofline performance model is mainly applicable for HPC applications that have few compute-intensive loops. I do not recommend using it for general-purpose applications, such as compilers, web browsers, or databases.

Arithmetic Intensity is a ratio between Floating-point operations (FLOPs)⁸⁸ and bytes, and it can be calculated for every loop in a program. Let's calculate the arithmetic intensity of code in Listing 5.4. In the innermost loop body, we have a floating-point addition and a multiplication; thus, we have 2 FLOPs. Also, we have three read operations and one write operation; thus, we transfer $4 \text{ operations} * 4 \text{ bytes} = 16$ bytes. The arithmetic intensity of that code is $2 / 16 = 0.125$. Arithmetic intensity is the X-axis on the Roofline chart, while the Y-axis measures the performance of a given program.

Listing 5.4 Naive parallel matrix multiplication.

```

1 void matmul(int N, float a[] [2048], float b[] [2048], float c[] [2048]) {
2     #pragma omp parallel for
3     for(int i = 0; i < N; i++) {
4         for(int j = 0; j < N; j++) {
5             for(int k = 0; k < N; k++) {
6                 c[i][j] = c[i][j] + a[i][k] * b[k][j];
7             }
8         }
9     }
10 }
```

Traditional ways to speed up an application's performance is to fully utilize the SIMD and multicore capabilities of a machine. Usually, we need to optimize for many aspects: vectorization, memory, and 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 5.6). This will give us an understanding of the scope for improving the performance of an application. If we find 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 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 on the chart. 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.

Theoretical maximums (rooflines) 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 the Intel Core i5-8259U processor, the maximum number of FLOPS (single-precision floats) with AVX2 and 2 Fused Multiply

⁸⁸ The Roofline performance model is not only applicable to floating-point calculations but can be also used for integer operations. However, the majority of HPC applications involve floating-point calculations, thus the Roofline model is mostly used with FLOPs.

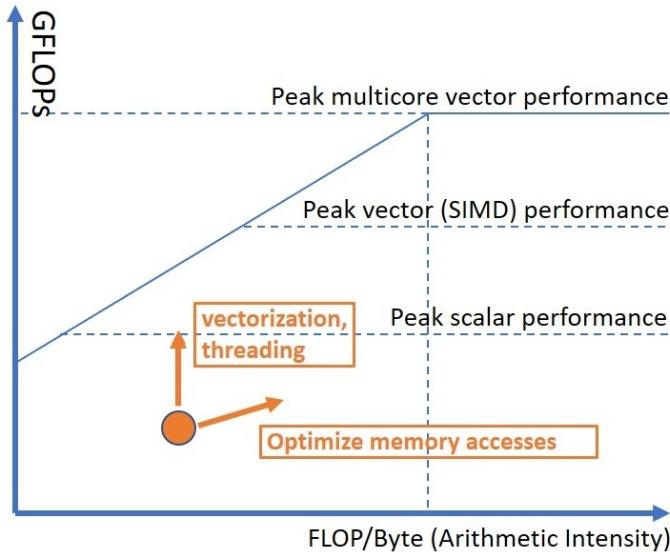


Figure 5.6: Roofline analysis of a program and potential ways to improve its performance.

Add (FMA) units can be calculated as:

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

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.

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

Automated tools like [Empirical Roofline Tool](#)⁸⁹ and [Intel Advisor](#)⁹⁰ are capable of empirically determining theoretical maximums by running a set of prepared benchmarks. If a calculation can reuse the data in the 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 5.7).

After hardware limitations are determined, we can start assessing the performance of an application against the roofline. 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 5.7. Notice, that Roofline charts have logarithmic scales.

⁸⁹ Empirical Roofline Tool - <https://bitbucket.org/berkeleylab/cs-roofline-toolkit/src/master/>.

⁹⁰ Intel Advisor - <https://software.intel.com/content/www/us/en/develop/tools/advisor.html>.

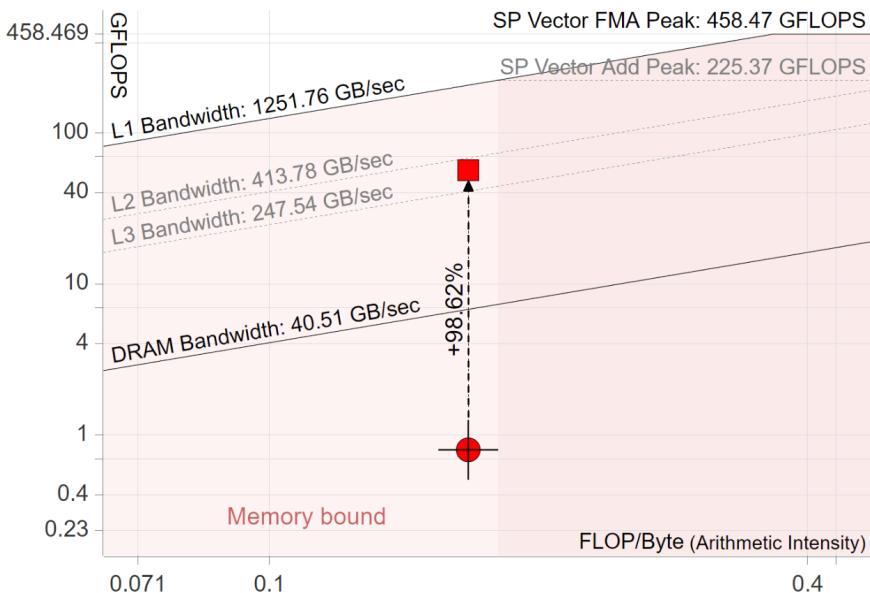


Figure 5.7: Roofline analysis for “before” and “after” versions of matrix multiplication on Intel NUC Kit NUC8i5BEH with 8GB RAM using the Clang 10 compiler.

Roofline methodology enables tracking optimization progress by plotting “before” and “after” points on the same chart. So, it is an iterative process that guides developers to help their applications fully utilize hardware capabilities. Figure 5.7 shows performance gains from making the following two changes to the code shown earlier in Listing 5.4:

- 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.

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.

5.6 Static Performance Analysis

Today we have extensive tooling for static code analysis. For the C and C++ languages, we have well-known tools like Clang static analyzer, Klocwork, Cppcheck, and others. Such tools aim at checking the correctness and semantics of code. Likewise, some tools 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 real hardware. 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 a 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 often 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. They don't run 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 real hardware and can simulate the code for different CPU generations. Another benefit is that you don't need to worry about the 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 [UICA⁹¹](#) and [llvm-mca⁹²](#).

Dynamic tools. They are based on running code on real hardware 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. Two examples of dynamic microarchitectural performance analyzers are [nanoBench⁹³](#) and [uarch-bench⁹⁴](#).

A bigger collection of tools both for static and dynamic microarchitectural performance analysis is available [here⁹⁵](#).

5.6.1 Case Study: Using UICA to Optimize FMA Throughput

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

⁹¹ UICA - <https://uica.uops.info/>

⁹² LLVM MCA - <https://llvm.org/docs/CommandGuide/llvm-mca.html>

⁹³ nanoBench - <https://github.com/andreas-abel/nanoBench>

⁹⁴ uarch-bench - <https://github.com/travisdowns/uarch-bench>

⁹⁵ Collection of links for C++ performance tools - <https://github.com/MattPD/cplinks/blob/master/performance.tools.md#microarchitecture>.

Let's look at the code in Listing 5.5. I 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 floating-point value `B` and accumulates products into `sum`. On the right, I present the machine code for the loop generated by Clang-16 when compiled with `-O3 -ffast-math -march=core-avx2`.

Listing 5.5 FMA throughput

<pre>float foo(float * a, float B, int N){ float sum = 0; for (int i = 0; i < N; i++) sum += a[i] * B; return sum; }</pre>	<pre>.loop: vfmadd231ps ymm2, ymm1, [rdi + rsi] vfmadd231ps ymm3, ymm1, [rdi + rsi + 32] vfmadd231ps ymm4, ymm1, [rdi + rsi + 64] vfmadd231ps ymm5, ymm1, [rdi + rsi + 96] sub rsi, -128 cmp rdx, rsi jne .loop</pre>
---	---

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 of 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 multiplication and addition into a single instruction. The constant `B` is broadcast into the `ymm1` register. The `-ffast-math` option allows a compiler to reassociate floating-point operations; we will discuss how this option can aid optimizations in Section 9.4.⁹⁶

The code looks good, but is it optimal? Let's find out. We took the assembly snippet from Listing 5.5 to UICA and ran simulations. At the time of writing, Alder Lake (Intel's 12th gen, based on Golden Cove) is not supported by UICA, so we ran it on the latest available, which is Rocket Lake (Intel's 11th gen, based on Sunny Cove). Although the architectures differ, the issue exposed by this experiment is equally visible in both. The result of the simulation is shown in Figure 5.8. 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, and all further iterations look very similar.

UICA is a very simplified model of the actual CPU pipeline. For example, you may notice that the 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 the L1 cache and branches are always predicted correctly, which we know is not the case in modern processors. Again, this is irrelevant to our experiment as we could still use the simulation results to find a way to improve the code.

Can you see the performance issue? Let's examine the diagram. First of all, every

⁹⁶ By the way, it would be possible to sum all the elements in `a` and then multiply it by `B` once outside of the loop. This is an oversight by the programmer, but hopefully, compilers will be able to handle it in the future.

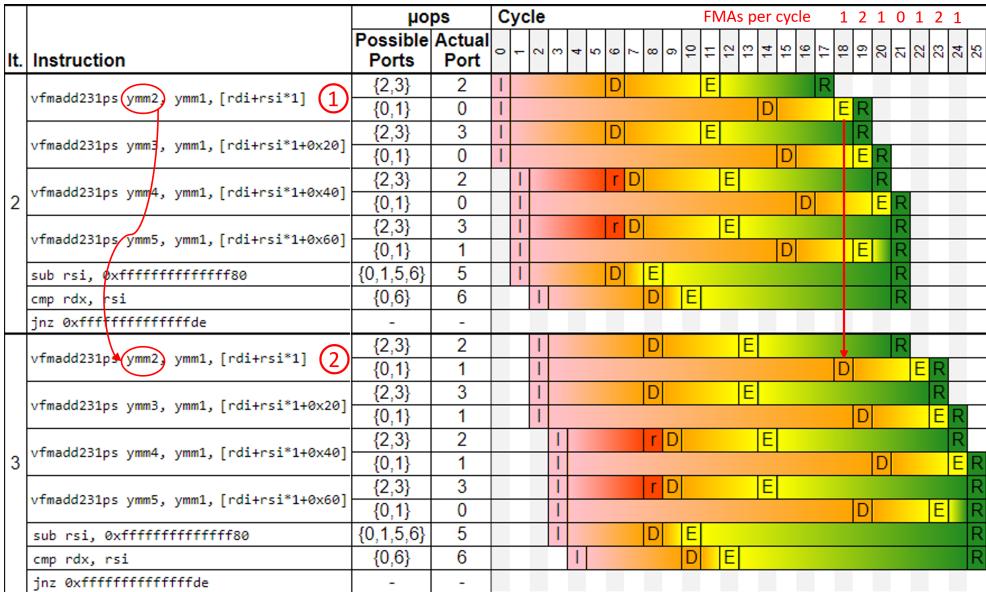


Figure 5.8: UICA pipeline diagram. I = issued, r = ready for dispatch, D = dispatched, E = executed, R = retired.

FMA instruction is broken into two μ ops (see ① in Figure 5.8): a load μ op that goes to ports {2,3} and an FMA μ op that can go to ports {0,1}. The load μ op has a latency of 5 cycles: it starts at cycle 7 and finishes at cycle 11. The FMA μ op has a latency of 4 cycles: it starts at cycle 15 and finishes at cycle 18. All FMA μ ops depend on load μ ops, as we can see in the diagram: FMA μ ops always start after the corresponding load μ op finishes. Now find two r cells at cycle 6, they are ready to be dispatched, but Rocket Lake 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 *yymm2*-*yymm5*. The FMA μ op from instruction ② that writes into *yymm2* cannot start execution before instruction ① from the previous iteration finishes. Notice that the FMA μ op from instruction ② was dispatched in the same cycle 18 as instruction ① finished its execution. There is a data dependency between instruction ① and instruction ②. 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 μ ops (this is not printed by UICA). It goes like 1,2,1,0,1,2,1, ..., or an average of one FMA μ op per cycle. Most of the recent Intel processors have two FMA execution units, thus can issue two FMAs μ ops per cycle. Thus, we utilize only half of the available FMA execution throughput. The diagram clearly shows the gap as every fourth cycle there are no FMAs executed. As we figured out before, no FMA μ ops can be dispatched because their inputs (*yymm2*-*yymm5*) are not ready.

To increase the utilization of FMA execution units from 50% to 100%, we need to

double the number of accumulators from 4 to 8, effectively unrolling the loop by a factor of two. Instead of 4 independent data flow chains, we will have 8. I'm not showing simulations of the unrolled version here; you can experiment on your own. Instead, let us confirm the hypothesis by running both versions on real hardware. By the way, it is always a good idea to verify, because static performance analyzers like UICA are not accurate models. Below, we show the output of two `nanoBench` tests that we ran on an Alder Lake processor. The tool takes provided assembly instructions (`-asm` option) 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 the improved version can execute 8 instructions in 4 cycles. Now we can be sure we have maximized the FMA execution throughput since the code on the right keeps the FMA units busy all the time.

<pre># ran on Intel Core i7-1260P (Alder Lake) \$ sudo ./kernel-nanoBench.sh -f -basic -loop 100 -unroll 1000 -warm_up_count 10 -asm " VFMADD231PS YMM0, YMM1, [R14]; VFMADD231PS YMM2, YMM1, [R14+32]; VFMADD231PS YMM3, YMM1, [R14+64]; VFMADD231PS YMM4, YMM1, [R14+96];" -asym_init "<not shown>" Instructions retired: 4.00 Core cycles: 4.00</pre>	<pre>\$ sudo ./kernel-nanoBench.sh -f -basic -loop 100 -unroll 1000 -warm_up_count 10 -asm " VFMADD231PS YMM0, YMM1, [R14]; VFMADD231PS YMM2, YMM1, [R14+32]; VFMADD231PS YMM3, YMM1, [R14+64]; VFMADD231PS YMM4, YMM1, [R14+96]; VFMADD231PS YMM5, YMM1, [R14+128]; VFMADD231PS YMM6, YMM1, [R14+160]; VFMADD231PS YMM7, YMM1, [R14+192]; VFMADD231PS YMM8, YMM1, [R14+224];" -asym_init "<not shown>" Instructions retired: 8.00 Core cycles: 4.00</pre>
---	---

As a rule of thumb, in such situations, the loop must be unrolled by a factor of $T * L$, 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 Alder Lake is 2 and the latency of FMA is 4 cycles. This creates 8 separate data flow chains that can be executed independently.

It's worth mentioning that you will not always see a 2x speedup in practice. This can be achieved only 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 outweighs the effect of suboptimal FMA port utilization, which could easily result in a 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 to build up a mental model of how a CPU works. Another very important use case for UICA is to find critical dependency chains in a loop as described in a post⁹⁷ on the Easyperf blog.

⁹⁷ Easyperf blog - <https://easyperf.net/blog/2022/05/11/Visualizing-Performance-Critical-Dependency-Chains>

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. The majority of developers leave the job of optimizing code to compilers, interfering 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 the 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 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, Intel compiler, and MSVC⁹⁸ provide optimization reports to check what optimizations were done for a particular piece of code.

Let's take a look at Listing 5.6, which shows an example of a loop that is not vectorized by clang 16.0.

Listing 5.6 a.c

```

1 void foo(float* __restrict__ a,
2         float* __restrict__ b,
3         float* __restrict__ c,
4         unsigned N) {
5     for (unsigned i = 1; i < N; i++) {
6         a[i] = c[i-1]; // value is carried over from previous iteration
7         c[i] = b[i];
8     }
9 }
```

To emit an optimization report in the Clang compiler, 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 5.6. The value that is loaded by `c[i-1]` depends on the store from the previous iteration (see operations ② and ③ in Figure 5.9). The dependency can be revealed by manually unrolling the first few iterations of the loop:

⁹⁸ At the time of writing (2024), MSVC provides only vectorization reports.

```
// 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];
...
```

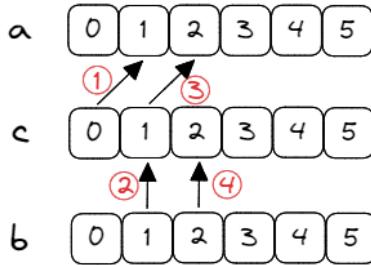


Figure 5.9: Visualizing the order of operations in Listing 5.6.

If we were to vectorize the code in Listing 5.6, 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 5.6 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 5.7. 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. Doing this would double the loop overhead, but this drawback would be outweighed by the performance improvement gained through vectorization.

Listing 5.7 a.c

```
1 void foo(float* __restrict__ a,
2         float* __restrict__ b,
3         float* __restrict__ c,
4         unsigned N) {
5     for (unsigned i = 1; i < N; i++) {
6         c[i] = b[i];
7         a[i] = c[i-1];
8     }
9 }
```

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 will provide more examples in Section 9.4.2, where we discuss how to discover 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 decide 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 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 to be used by compiler writers to improve optimization passes. Over the years, several tools have been developed to make optimization reports more accessible and actionable by application developers, most notably `opt-viewer`⁹⁹ and `optview2`.¹⁰⁰ Also, the [Compiler Explorer](#) website has the “Optimization Output” tool for LLVM-based compilers that reports performed transformations when you hover your mouse over the corresponding line of source code. All of these tools help visualize successful and failed code transformations by LLVM-based compilers.

In the Link-Time Optimization (LTO)¹⁰¹ mode, some optimizations are made during the linking stage. To emit compiler reports from both the compilation and linking stages, you should pass dedicated options to both the compiler and the linker. See the LLVM “Remarks” guide¹⁰² for more information.

A slightly different way of reporting missing optimizations is taken by the Intel® ISPC¹⁰³ 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. They are a fast way to check what optimizations were done for a particular hotspot and see if some important ones failed. I have found many improvement opportunities thanks to compiler optimization reports.

Questions and Exercises

1. Which performance analysis approaches would you use in the following scenarios?
 - scenario 1: the client support team reports a customer issue: after upgrading to a new version of the application, the performance of a certain operation drops by 10%.
 - scenario 2: the client support team reports a customer issue: some transactions run 2x longer than others with no particular pattern.

⁹⁹ `opt-viewer` - <https://github.com/llvm/llvm-project/tree/main/llvm/tools/opt-viewer>

¹⁰⁰ `optview2` - <https://github.com/OfekShilon/optview2>

¹⁰¹ Link-Time Optimizations, also called InterProcedural Optimizations (IPO). Read more here: https://en.wikipedia.org/wiki/Interprocedural_optimization

¹⁰² LLVM compiler remarks - <https://llvm.org/docs/Remarks.html>

¹⁰³ ISPC - <https://ispc.github.io/ispc.html>

- scenario 3: you’re evaluating three different compression algorithms and you want to know what types of performance bottlenecks (memory latency, computations, 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 into your project; you’ve decided to compare their performance.
 - scenario 5: you were asked to analyze the performance of some unfamiliar code, which involves a hot loop; you want to know how many iterations the loop is doing.
2. Run the application that you’re working with daily. Practice doing performance analysis using the approaches we discussed in this chapter. Collect raw counts for various CPU performance events, find hotspots, collect roofline data, and generate and study the compiler optimization report for 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 hardware and software provide data that can be used for performance monitoring.
- Code instrumentation enables us to track many things in a program but causes relatively large overhead both on the development and runtime side. While most developers are not in the habit of manually instrumenting their code, this approach is still relevant for automated processes, e.g., Profile-Guided Optimizations (PGO).
- Tracing is conceptually similar to instrumentation and is useful for exploring anomalies in a system. Tracing enables us to catch the entire sequence of events with timestamps attached to each event.
- Performance monitoring counters are a very important instrument of low-level performance analysis. They are generally used in two modes: “Counting” or “Sampling”. The counting mode is primarily used for calculating various performance metrics.
- Sampling skips most of a program’s execution and takes 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 a program. Sampling is the most popular analysis approach since it doesn’t require recompilation of a 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), while sampling gets more expensive with increasing sampling frequency [Nowak & Bitzes, 2014].
- The Roofline Performance Model is a throughput-oriented performance model that is heavily used in the High Performance Computing (HPC) world. It visual-

izes the performance of an application against hardware limitations. The 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.

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 into the hotspots of the application. Sometimes it's all that's needed to help developers find and fix the performance problems. 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, simply looking at the list of hotspots is not enough. Unless you have a crystal ball or an accurate model of an entire CPU in your head, you need additional support to understand what the performance bottlenecks are.

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 hardware performance monitoring capabilities available on modern CPUs. Processors from different vendors do not necessarily have the same set of features. We will explore performance monitoring capabilities available in Intel, AMD, and Arm processors.¹⁰⁴

- **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 the intricacies of the CPU microarchitecture and is relatively easy to use even for inexperienced developers.
- **Branch Recording**, discussed in Section 6.2. This is a mechanism that continuously logs the most recent branch outcomes in parallel with executing the

¹⁰⁴ The RISC-V ecosystem does not yet have a mature performance monitoring infrastructure, so we will not cover it in this book.

program. It is used for collecting call stacks, identifying hot branches, calculating misprediction rates of individual branches, and more.

- **Hardware-Based Sampling**, discussed in Section 6.3.1. 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 C. 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 C. 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, and partially because in many cases it is 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. Nevertheless, it is popular in some industries, such as high-frequency trading (HFT).

The hardware performance monitoring features mentioned above provide insights into the efficiency of a program from the CPU perspective. In the next chapter, we will discuss how profiling tools use these features to provide many different types of analysis.

6.1 Top-down Microarchitecture Analysis

Top-down Microarchitecture Analysis (TMA) methodology is a very powerful technique for identifying CPU bottlenecks in a program. 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 is stalling the execution of a program. Figure 6.1 illustrates the core idea of TMA. 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 μ ops that are being executed. Whenever a new instruction is fetched and decoded, new entries in those buffers are allocated. If a μ op 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 (**Frontend Bound**), or the Backend was overloaded with work, and resources for the new μ op could not be allocated (**Backend Bound**). If a μ op 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 μ ops to be, although there are exceptions which we will talk about later.

This is not how the analysis works in practice because analyzing every single micro-operation (μ op) would be terribly slow. Instead, TMA observes the execution of a program by monitoring a specific set of performance events and then calculates metrics based on predefined formulas. Using these 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,

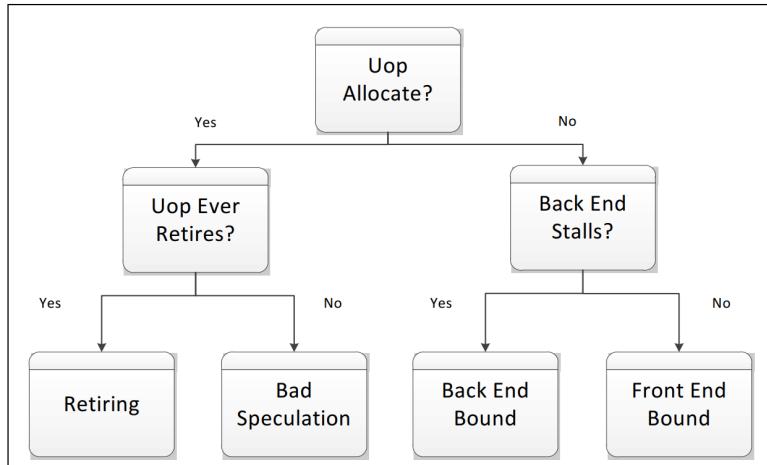


Figure 6.1: The concept behind TMA’s top-level breakdown. © Source: [[Yasin, 2014](#)]

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.

6.1.1 TMA on Intel Platforms

The TMA methodology was first proposed by Intel in 2014 and is supported starting from the Sandy Bridge 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 6.2).

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. First, we collect metrics for the main four buckets: **Frontend Bound**, **Backend Bound**, **Retiring**, and **Bad Speculation**. Say, we found out that a big portion of the program execution was stalled by memory accesses (which is a **Backend Bound** bucket, see Figure 6.2). The next step is to run the workload again and collect metrics specific to the **Memory Bound** bucket only. The process is repeated until we know the exact root cause, for example, **L3 Bound**.

It is 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 (see Section 5.3.1) between different performance events during a single run. Also, in a real-world application, performance could be limited by several factors. For example, it can experience a large number of branch mispredictions (**Bad Speculation**) and cache misses (**Backend 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

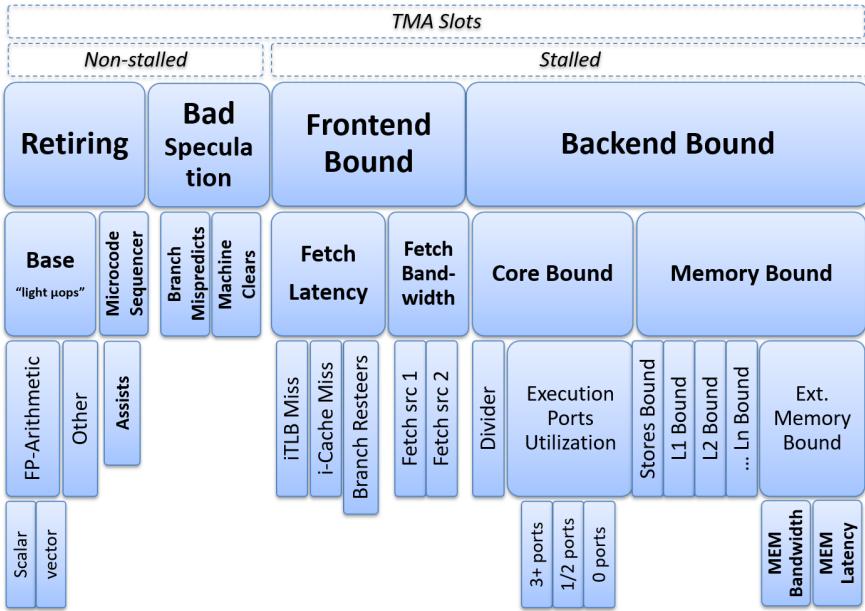


Figure 6.2: The TMA hierarchy of performance bottlenecks. © Image by Ahmad Yasin.

a single benchmark run. 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 a program. It allows TMA to give an accurate representation of CPU microarchitecture utilization, taking into account the full bandwidth of a 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.

Once we identify the performance bottleneck, 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 corresponding assembly instructions. The analysis methodology provides performance events 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 will become 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, I took a very simple benchmark, such that it is easy to understand and change. It is not representative of real-world applications, but it is good enough to demonstrate the workflow of TMA. I 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, I don't start by showing you the 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.

I ran the experiments on the machine equipped with an 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).

Step 1: Identify the Bottleneck

As a first step, we run our microbenchmark and collect a limited set of events that will help us calculate Level 1 metrics. Here, we try to identify high-level performance bottlenecks of our application by attributing them to the four L1 buckets: **Frontend Bound**, **Backend Bound**, **Retiring**, and **Bad Speculation**. It is possible to collect Level 1 metrics using the Linux `perf` tool. The `perf stat` command has a dedicated `--topdown` option. In more recent versions it will output these metrics by default. Below is the breakdown for our benchmark. The output of all commands in this section is trimmed to save space.

```
$ perf stat -- ./benchmark.exe
...
TopdownL1 (cpu_core)  # 53.4 % tma_backend_bound    <==
# 0.2 % tma_bad_speculation
# 13.8 % tma_frontend_bound
# 32.5 % tma_retiring
...
...
```

By looking at the output, we can tell that the performance of the application is bound by the CPU backend. Let's drill one level down. To obtain TMA metrics Level 2, 3, and further, I will use the `toplev` tool that is a part of [pmu-tools¹⁰⁵](#) written by Andi Kleen. It is implemented in Python and uses 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 <==
S0-C0 Backend_Bound.Core_Bound: 8.80 % Slots
```

¹⁰⁵ PMU tools - <https://github.com/andikleen/pmu-tools>.

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

In this command, we pinned the process to CPU0 (using `taskset -c 0`) and limited the output of `toplev` to this core only (`--core S0-C0`). The option `-12` 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: ¹⁰⁶

```
$ ~/pmu-tools/toplev.py --core S0-C0 -13 -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.

Step 2: Locate the Place in the Code

The second step in the TMA process is to locate the place in the code where the identified performance event occurs most frequently. To do so, you 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

¹⁰⁶ Alternatively, we could use the `-12 --nodes L1_Bound,L2_Bound,L3_Bound,DRAM_Bound,Store_Bound` option instead of `-13` to limit the collection since we know the application is bound by memory.

used to locate the issue. To understand 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 found with the help of the [TMA metrics¹⁰⁷](#) 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. 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_RETIRE.L3_MISS_PS` precise event. Here is the example command:

```
$ perf record -e cpu/event=0xd1,umask=0x20,name=MEM_LOAD_RETIRE.L3_MISS/ppp --
./benchmark.exe
$ perf report -n --stdio
...
# Samples: 33K of event 'MEM_LOAD_RETIRE.'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_pcpages_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](#).¹⁰⁸

To avoid compiler optimizations, function `foo` is implemented in assembly language, which is presented in Listing 6.1. The “driver” portion of the benchmark is implemented in the `main` function, as shown in Listing 6.2. 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.¹⁰⁹

Listing 6.1 Assembly code of function `foo`.

```
$ perf annotate --stdio -M intel foo
Percent | Disassembly of benchmark.exe for MEM_LOAD_RETIRE.L3_MISS
-----
: Disassembly of section .text:
:
: 000000000400a00 <foo>:
: foo():
0.00 : 400a00: nop  DWORD PTR [rax+rax*1+0x0]
0.00 : 400a08: nop  DWORD PTR [rax+rax*1+0x0]
... # more NOPs
100.00 : 400e07: mov   rax,QWORD PTR [rdi+rsi*1] <=
...
0.00 : 400e13: xor   rax,rax
0.00 : 400e16: ret
```

¹⁰⁷ TMA metrics - https://github.com/intel/perfmon/blob/main/TMA_Metrics.xlsx.

¹⁰⁸ Case study example - https://github.com/dendibakh/dendibakh.github.io/tree/master/_posts/code/TMAM.

¹⁰⁹ According to x86 Linux calling conventions (https://en.wikipedia.org/wiki/X86_calling_conventions), the first 2 arguments land in `rdi` and `rsi` registers respectively.

Listing 6.2 Source code of function main.

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

By looking at Listing 6.1, 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.

Step 3: Fix the Issue

There is dummy work emulated with NOPs at 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 the time window allows us to start prefetching the memory location in parallel with the dummy work. Listing 6.3 shows this idea in action. More information about the explicit memory prefetching technique can be found in Section 8.5.

Listing 6.3 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);
}
```

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 19 billion down to 2 billion.

TMA is an iterative process, so once we fix one problem, we need to repeat the process starting from Step 1. 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 applications is unlikely to be that easy. Chapters in the second part of the book are organized to make them convenient for use with the TMA process. In particular, Chapter 8 covers the `Memory Bound` category, Chapter 9 covers `Core Bound`, Chapter 10 covers `Bad Speculation`, and Chapter 11 covers `Frontend Bound`. Such a structure is intended to form a checklist that you can use to drive code changes when you encounter a certain performance bottleneck.

6.1.2 TMA on AMD Platforms

Starting from Zen4, AMD processors support Level-1 and Level-2 TMA analysis. According to AMD documentation, it is called “Pipeline Utilization” analysis, but the

idea remains the same. The L1 and L2 buckets are also very similar to Intel's. Since kernel 6.2, Linux users can utilize the `perf` tool to collect the pipeline utilization data.

Next, we will examine the [Crypto++¹¹⁰](#) implementation of SHA-256 (Secure Hash Algorithm 256), the fundamental cryptographic algorithm in Bitcoin mining. Crypto++ is an open-source C++ class library of cryptographic algorithms and contains an implementation of many algorithms, not just SHA-256. However, for our example, I disabled benchmarking of other algorithms by commenting out corresponding lines in the `BenchmarkUnkeyedAlgorithms` function in `bench1.cpp`.

I ran the test on an AMD Ryzen 9 7950X machine with Ubuntu 22.04, Linux kernel 6.5. I compiled Crypto++ version 8.9 using GCC 12.3 C++ compiler. I used the default `-O3` optimization option, but it doesn't impact performance much since the code is written with x86 intrinsics (see Section 9.5) and utilizes the SHA x86 ISA extension.

Below is the command I used to obtain L1 and L2 pipeline utilization metrics. The output was trimmed and some statistics were dropped to remove unnecessary distraction.

```
$ perf stat -M PipelineL1,PipelineL2 -- ./cryptest.exe b1 10
0.0 % bad_speculation_mispredicts      (20.08%)
0.0 % bad_speculation_pipeline_restarts (20.08%)
0.0 % bad_speculation                 (20.08%)
6.1 % frontend_bound                  (20.00%)
6.1 % frontend_bound_bandwidth        (20.00%)
0.1 % frontend_bound_latency          (20.00%)
65.9 % backend_bound_cpu             (20.00%)
1.7 % backend_bound_memory           (20.00%)
67.5 % backend_bound                (20.00%)
26.3 % retiring                     (20.08%)
20.2 % retiring_fastpath            (19.99%)
6.1 % retiring_microcode            (19.99%)
```

In the output, numbers in brackets indicate the percentage of runtime duration, when a metric was monitored. As we can see, each metric was monitored only 20% of the time due to multiplexing. In our case it is likely not a concern as SHA256 has consistent behavior, however it may not always be the case. To minimize the impact of multiplexing, you can collect a limited set of metrics in a single run, e.g., `perf stat -M frontend_bound,backend_bound`.

A description of pipeline utilization metrics shown above can be found in [AMD, 2024, Chapter 2.8 Pipeline Utilization]. By looking at the metrics, we can see that branch mispredictions are not happening in SHA256 (`bad_speculation` is 0%). Only 26.3% of the available dispatch slots were used (`retiring`), which means the remaining 73.7% were wasted due to frontend and backend stalls.

Advanced cryptography instructions are not trivial, so internally they are broken into smaller pieces (μ ops). Once a processor encounters such an instruction, it retrieves μ ops for it from the microcode. Microoperations are fetched from the microcode sequencer with a lower bandwidth than from regular instruction decoders, making it a potential source of performance bottlenecks. Crypto++ SHA256 implementation

¹¹⁰ Crypto++ - <https://github.com/weidai11/cryptopp>

heavily uses instructions such as `SHA256MSG2`, `SHA256RNDS2`, and others which consist of multiple μ ops according to uops.info¹¹¹ website.

The `retiring_microcode` metric indicates that 6.1% of dispatch slots were used by microcode operations that eventually retired. When comparing with its sibling metric `retiring_fastpath`, we can say that roughly every 4th instruction was a microcode operation. If we now look at the `frontend_bound_bandwidth` metric, we will see that 6.1% of dispatch slots were unused due to bandwidth bottleneck in the CPU frontend. This suggests that 6.1% of dispatch slots were wasted because the microcode sequencer has not been providing μ ops while the backend could have consumed them. In this example, the `retiring_microcode` and `frontend_bound_bandwidth` metrics are tightly connected, however, the fact that they are equal is merely a coincidence.

The majority of cycles are stalled in the CPU backend (`backend_bound`), but only 1.7% of cycles are stalled waiting for memory accesses (`backend_bound_memory`). So, we know that the benchmark is mostly limited by the computing capabilities of the machine. As you will know from Part 2 of this book, it could be related to either data flow dependencies or execution throughput of certain cryptographic operations. They are less frequent than traditional ADD, SUB, CMP, and other instructions and thus can be often executed only on a single execution unit. A large number of such operations may saturate the execution throughput of this particular unit. Further analysis should involve a closer look at the source code and generated assembly, checking execution port utilization, finding data dependencies, etc.

In summary, the Crypto++ implementation of SHA-256 on AMD Ryzen 9 7950X utilizes only 26.3% of the available dispatch slots; 6.1% of the dispatch slots were wasted due to the microcode sequencer bandwidth, and 65.9% were stalled due to lack of computing resources of the machine. The algorithm certainly hits a few hardware limitations, so it's unclear if its performance can be improved or not.

When it comes to Windows, at the time of writing, TMA methodology is only supported on AMD server platforms (codename Genoa), and not on client systems (codename Raphael). TMA support was added in AMD uProf version 4.1, but only in the command line tool `AMDuProfPcm` tool which is part of AMD uProf installation. You can consult [AMD, 2024, Chapter 2.8 Pipeline Utilization] for more details on how to run the analysis. The graphical version of AMD uProf doesn't have the TMA analysis yet.

6.1.3 TMA On Arm Platforms

Arm CPU architects also have developed a TMA performance analysis methodology for their processors, which we will discuss next. Arm calls it “Topdown” in their documentation [Arm, 2023a], so we will use their naming. At the time of this writing (late 2023), Topdown is only supported on cores designed by Arm, e.g. Neoverse N1 and Neoverse V1,¹¹² and their derivatives, e.g. Ampere Altra and AWS Graviton3. Refer to the list of major CPU microarchitectures at the end of this book if you need to refresh your memory on Arm chip families. Processors designed by Apple don't

¹¹¹ uops.info - <https://uops.info/table.html>

¹¹² In September 2024, Arm published performance analysis for Neoverse V2 and V3 cores - <https://developer.arm.com/telemetry>

support the Arm Topdown performance analysis methodology yet.

Arm Neoverse V1 is the first CPU in the Neoverse family that supports a full set of level 1 Topdown metrics: **Bad Speculation**, **Frontend Bound**, **Backend Bound**, and **Retiring**. Prior to V1 cores, Neoverse N1 supported only two L1 categories: **Frontend Stalled Cycles** and **Backend Stalled Cycles**.¹¹³

To demonstrate Arm’s Topdown analysis on a V1-based processor, I launched an AWS EC2 `m7g.metal` instance powered by the AWS Graviton3. Note that Topdown might not work on other non-metal instance types due to virtualization. I used 64-bit ARM Ubuntu 22.04 LTS with Linux kernel 6.2 managed by AWS. The provided `m7g.metal` instance had 64 vCPUs and 256 GB of RAM.

I applied the Topdown methodology to **AI Benchmark Alpha**,¹¹⁴ which is an open-source Python library for evaluating AI performance of various hardware platforms, including CPUs, GPUs, and TPUs. The benchmark relies on the TensorFlow machine learning library to measure inference and training speed for key Deep Learning models. In total, AI Benchmark Alpha consists of 42 tests, including classification, image segmentation, text translation, and others.

Arm engineers have developed the **topdown-tool**¹¹⁵ that we will use below. The tool works both on Linux and Windows on ARM. On Linux, it utilizes a standard perf tool, while on Windows it uses **WindowsPerf**¹¹⁶, which is a Windows on ARM performance profiling tool. Similar to Intel’s TMA, the Arm methodology employs the “drill-down” concept, i.e. you first determine the high-level performance bottleneck and then drill down for a more nuanced root cause analysis. Here is the command we used:

```
$ topdown-tool --all-cpus -m Topdown_L1 -- python -c "from ai_benchmark import
    AIBenchmark; results = AIBenchmark(use_CPU=True).run()"
Stage 1 (Topdown metrics)
=====
[Topdown Level 1]
Frontend Bound... 16.48% slots
Backend Bound.... 54.92% slots
Retiring......... 27.99% slots
Bad Speculation.. 0.59% slots
```

where the `--all-cpus` option enables system-wide collection for all CPUs, and `-m Topdown_L1` collects Topdown Level 1 metrics. Everything that follows `--` is the command line to run the AI Benchmark Alpha suite.

From the output above, we conclude that the benchmark doesn’t suffer from branch mispredictions. Also, without a deeper understanding of the workloads involved, it’s hard to say if the Frontend Bound of 16.5% is worth looking at, so we turn our attention to the Backend Bound metric, which is the main source of stalled cycles. Neoverse V1-based chips don’t have the second-level breakdown, instead, the methodology suggests further exploring the problematic category by collecting a set of corresponding metrics. Here is how we drill down into the more detailed **Backend Bound** analysis:

¹¹³ Performance analysis guides for Neoverse V2 and V3 cores became available after I wrote this section. See the following page - <https://developer.arm.com/telemetry>.

¹¹⁴ AI Benchmark Alpha - <https://ai-benchmark.com/alpha.html>

¹¹⁵ Arm **topdown-tool** - <https://learn.arm.com/install-guides/topdown-tool/>

¹¹⁶ WindowsPerf - <https://gitlab.com/Linaro/WindowsPerf/windowsperf>

```
$ topdown-tool --all-cpus -n BackendBound -- python -c "from ai_benchmark import
    AIBenchmark; results = AIBenchmark(use_CPU=True).run()"
Stage 1 (Topdown metrics)
=====
[Topdown Level 1]
Backend Bound..... 54.70% slots

Stage 2 (uarch metrics)
=====
[Data TLB Effectiveness]
DTLB MPKI..... 0.413 misses per 1,000 instructions
L1 Data TLB MPKI..... 3.779 misses per 1,000 instructions
L2 Unified TLB MPKI..... 0.407 misses per 1,000 instructions
DTLB Walk Ratio..... 0.001 per TLB access
L1 Data TLB Miss Ratio..... 0.013 per TLB access
L2 Unified TLB Miss Ratio..... 0.112 per TLB access

[L1 Data Cache Effectiveness]
L1D Cache MPKI..... 13.114 misses per 1,000 instructions
L1D Cache Miss Ratio..... 0.046 per cache access

[L2 Unified Cache Effectiveness]
L2 Cache MPKI..... 1.458 misses per 1,000 instructions
L2 Cache Miss Ratio..... 0.027 per cache access

[Last Level Cache Effectiveness]
LL Cache Read MPKI..... 2.505 misses per 1,000 instructions
LL Cache Read Miss Ratio..... 0.219 per cache access
LL Cache Read Hit Ratio..... 0.783 per cache access

[Speculative Operation Mix]
Load Operations Percentage..... 25.36% operations
Store Operations Percentage..... 2.54% operations
Integer Operations Percentage..... 29.60% operations
Advanced SIMD Operations Percentage. 10.93% operations
Floating Point Operations Percentage 6.85% operations
Branch Operations Percentage..... 10.04% operations
Crypto Operations Percentage..... 0.00% operations
```

In the command above, the option `-n BackendBound` collects all the metrics associated with the `Backend Bound` category as well as its descendants. The description for every metric in the output is given in [Arm, 2023a]. Note, that they are quite similar to what we have discussed in Section 4.9, so you may want to revisit it as well.

We don't have a goal of optimizing the benchmark, rather we want to characterize performance bottlenecks. However, if given such a task, here is how our analysis could continue. There are a substantial number of L1 Data TLB misses (3.8 MPKI), but then 90% of those misses hit in L2 TLB (see `L2 Unified TLB Miss Ratio`). All in all, only 0.1% of all TLB accesses result in a page table walk (see `DTLB Walk Ratio`), which suggests that it is not our primary concern, although a quick experiment that utilizes huge pages is still worthwhile.

Looking at the L1/L2/LL Cache Effectiveness metrics, we can spot a potential problem with data cache misses. One in ~22 accesses to the L1D cache results in a miss (see `L1D Cache Miss Ratio`), which is tolerable but still expensive. For L2, this number is one in 37 (see `L2 Cache Miss Ratio`), which is much better. However for LLC, the LL Cache Read Miss Ratio is unsatisfying: every 5th access results in a miss. Since this is an AI benchmark, where the bulk of the time is likely spent in matrix

multiplication, code transformations like loop blocking may help (see Section 9.3). AI algorithms are known for being “memory hungry”, however, Neoverse V1 Topdown doesn’t show if there are stalls that can be attributed to saturated memory bandwidth.

The final category provides the operation mix, which can be useful in some scenarios. The percentages give us an estimate of how many instructions of a certain type were executed, including speculatively executed instructions. The numbers don’t sum up to 100%, because the rest is attributed to the implicit “Others” category (not printed by the `topdown-tool`), which is about 15% in our case. We should be concerned by the low percentage of SIMD operations, especially given that the highly optimized `Tensorflow` and `numpy` libraries are used. In contrast, the percentage of integer operations and branches seems high. I checked that the majority of executed branch instructions are loop backward jumps to the next loop iteration. The high percentage of integer operations could be caused by lack of vectorization, or due to thread synchronization. [Arm, 2023a] gives an example of discovering a vectorization opportunity using data from the Speculative Operation Mix category.

In our case study, we ran the benchmark two times, however, in practice, one run is usually sufficient. Running the `topdown-tool` without options will collect all the available metrics using a single run. Also, the `-s combined` option will group the metrics by the L1 category, and output data in a format similar to Intel VTune, `toplev`, and other tools. The only practical reason for making multiple runs is when a workload has a bursty behavior with very short phases that have different performance characteristics. In such a scenario, you would like to avoid event multiplexing (see Section 5.3.1) and improve collection accuracy by running the workload multiple times.

The AI Benchmark Alpha has various tests that could exhibit different performance characteristics. The output presented above aggregates all 42 tests and gives an overall breakdown. This is generally not a good idea if the individual tests indeed have different performance bottlenecks. You need to have a separate Topdown analysis for each of the tests. One way the `topdown-tool` can help is to use the `-i` option which will output data per configurable interval of time. You can then compare the intervals and decide on the next steps.

6.1.4 TMA Summary

TMA is great for identifying CPU performance bottlenecks. Ideally, we would like to see the `Retiring` metric at 100%. Although there are exceptions. Having the `Retiring` metric at 100% means a CPU is fully saturated and it crunches instructions at full speed. But it doesn’t say anything about the quality of those instructions. A program can spin in a tight loop waiting for a lock; that would show a high `Retiring` metric, but it doesn’t do any useful work.

Another example in which you might see a high `Retiring` metric but slow overall performance is when a program has a hotspot that was not vectorized. You give a processor an “easy” time by letting it run simple non-vectorized operations, but is it an optimal way of using available CPU resources? Of course, no. If a CPU doesn’t have problems executing your code, that doesn’t mean performance cannot be improved. Watch out for such cases and remember that TMA identifies CPU performance bottlenecks but doesn’t correlate them with the performance of your

program. You will find it out once you do the necessary experiments.

While it is possible to achieve **Retiring** close to 100% on a toy program, real-world applications are far from getting there. Figure 6.3 shows top-level TMA metrics for Google's datacenter workloads along with several **SPEC CPU2006**¹¹⁷ benchmarks running on Intel's Ivy Bridge server processors. We can see that most data center workloads have a very small fraction in the **Retiring** bucket. This implies that most data center workloads spend time stalled on various bottlenecks. **BackendBound** is the primary source of performance issues. The **FrontendBound** category represents a bigger problem for data center workloads than in SPEC2006 because those applications typically have large codebases with poor locality. Finally, some workloads suffer from branch mispredictions more than others, e.g., **search2** and **445.gobmk**.

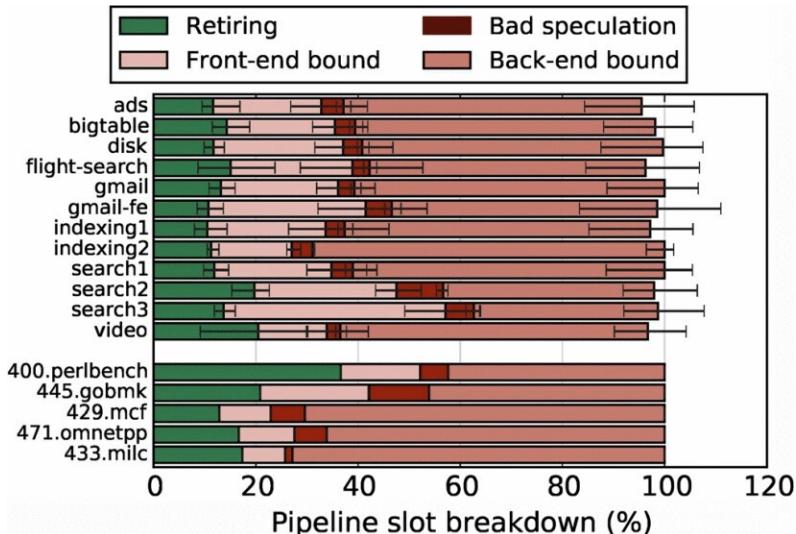


Figure 6.3: TMA breakdown of Google's datacenter workloads along with several SPEC CPU2006 benchmarks, © Source: [Kanev et al., 2015]

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... As we mentioned at the beginning of this chapter, using TMA on 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 the filesystem cache and run the benchmark under TMA, it will likely show that your application is Memory Bound, which may in fact be false when the 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

¹¹⁷ SPEC CPU 2006 - <http://spec.org/cpu2006/>.

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 demonstrates how using TMA to diagnose performance bottlenecks can support your decision to spend money on new hardware.

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 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 a branch is 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. This is what the Intel Processor Traces (PT) feature is capable of doing, which is discussed in Appendix C. Branch recording mechanisms that we will discuss in this section are based on sampling, 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” addresses 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 small number of 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 6.4 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 back edge JNE (1), conditional branch JNS (2), function CALL (3), and return address from this function (4).

Listing 6.5 shows one possible branch history 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.

The fact that untaken branches are not logged might add a burden for analysis but usually, it doesn’t complicate it too much. We can still infer the complete execution

Listing 6.4 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]
| L-> 4eda1e: call   4edb26          <== (3)
|   4eda23: add    rbx,0x4          <== (4)
|   4eda27: mov    DWORD PTR [rbx-0x4],eax
|   4eda2a: cmp    rbx,rbp
----- 4eda2d: jne    4eda10          <== (1)

```

Listing 6.5 Possible branch 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

↓ time
V

path since we know that the control flow was sequential from the destination address in the entry N-1 to the source address in the 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 first implemented its Last Branch Record (LBR) facility in the NetBurst microarchitecture. Initially, it could record only the 4 most recent branch outcomes. It was later enhanced to 16 starting with Nehalem and to 32 starting from Skylake. Prior to the Golden Cove microarchitecture, LBR was implemented as a set of model-specific registers (MSRs), but now it works within architectural registers.¹¹⁸

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.

¹¹⁸ Its primary advantage is that LBR features are clearly defined 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 6.3.1).

When a sampling counter overflows and a Performance Monitoring Interrupt (PMI) is triggered, the LBR logging freezes until the 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. Applying such a filter to the code in Listing 6.4, 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, etc. In Linux perf, there is a `-j` option that enables/disables the 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 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 6.4 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, then 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 records with 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 a `CALL` instruction pushes and a `RET` instruction pops entry from the stack. If the depth of the call stack in your application never goes beyond 32 nested frames, LBRs will give you very accurate information. [Intel, 2023, Volume 3B, Chapter 19 Last Branch Records]

You can make sure LBRs are enabled on your system with 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 the `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 are 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, the 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:

```
...
0x4edadf9/0x4edab0/P/-/-/29
0x4edabd/0x4edad0/P/-/-/2
0x4edad0/0x4edb00/M/-/-/4
0x4edb24/0x4edab0/P/-/-/24
0x4edad0/0x4edad0/P/-/-/2
0x4edad0/0x4edb00/M/-/-/1
0x4edb24/0x4edab0/P/-/-/3
0x4edad0/0x4edad0/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), a predicted flag (this one single branch outcome was M - Mispredicted, P - Predicted), and the number of cycles since the previous record (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 specification¹¹⁹](#).

6.2.2 LBR on AMD Platforms

AMD processors also support Last Branch Record (LBR) on AMD Zen4 processors. Zen4 can log 16 pairs of “from” and “to” addresses along with some additional metadata. Similar to Intel LBR, AMD processors can record various types of branches. The main difference from Intel LBR is that AMD processors don’t support call stack mode yet, hence the LBR feature can’t be used for call stack collection. Another noticeable difference is that there is no cycle count field in the AMD LBR record. For more details see [\[AMD, 2023, 13.1.1.9 Last Branch Stack Registers\]](#).

Since Linux kernel 6.1 onwards, Linux perf on AMD Zen4 processors supports the branch analysis use cases we discuss below unless explicitly mentioned otherwise. The Linux perf commands use the same **-b** and **-j** options as on Intel processors.

Branch analysis is also possible with the AMD uProf CLI tool. The example command below dumps collected raw LBR records and generates a CSV report:

```
$ AMDuProfCLI collect --branch-filter -o /tmp/ ./AMDTClassicMatMul-bin
```

6.2.3 BRBE on Arm Platforms

Arm introduced its branch recording extension called BRBE in 2020 as a part of the ARMv9.2-A ISA. Arm BRBE is very similar to Intel’s LBR and provides many similar features. Just like Intel’s LBR, BRBE records contain source and destination addresses, a misprediction bit, and a cycle count value. According to the latest available BRBE specification, the call stack mode is not supported. The Branch records only contain information for a branch that is architecturally executed, i.e.,

¹¹⁹ Linux perf script manual page - <http://man7.org/linux/man-pages/man1/perf-script.1.html>.

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 the BRBE extension in action.

Several use cases become possible thanks to branch recording. Next, we will cover the most important ones.

6.2.4 Capture Call Stacks

One of the most popular use cases for branch recording is capturing call stacks. I already covered why we need to collect them in Section 5.4.3. Branch recording can be used as a lightweight substitution for collecting call graph information even if you compiled a program without frame pointers or debug information.

At the time of writing (late 2023), AMD’s LBR and Arm’s BRBE don’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
|          |
|          |--90.86%--foo
|          |          |
|          |          |--90.86%--bar
|          |
--9.08%--zoo
          bar
```

As you can see, we identified the hottest function in the program (which is `bar`). Also, we identified 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% of the time 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 identify the most frequently taken branches. It is supported on Intel and AMD. According to Arm’s BRBE specification, it can be supported, but due to the unavailability of processors that implement this extension, it is not possible to verify. Here is an example:

```
$ 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` event to analyzing LBR stacks: only 670 samples were collected, yet we have an entire 32-entry LBR stack captured with every sample. This gives us $670 * 32 = 21440$ LBR entries (branch outcomes) for analysis.¹²⁰

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, you 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 completely 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](#)¹²¹ blog).

Using branch recording, we can also find a *hyperblock* (sometimes called *superblock*), which is a chain of hot basic blocks in a function that are not necessarily laid out in the sequential physical order but are executed sequentially. Thus, a hyperblock represents a typical hot path inside a function.

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.¹²² The output of the `perf report` command is slightly trimmed to fit nicely on a page. The following use case is supported on Intel and AMD. According to Arm's BRBE specification, it can be supported, but due to the unavailability of processors that implement this extension, it is not possible to verify.

```
$ 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
```

¹²⁰ 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`.

¹²¹ Easyperf: Estimating Branch Probability - <https://easyperf.net/blog/2019/05/06/Estimating-branch-probability>

¹²² LLVM test-suite 7zip benchmark - <https://github.com/llvm-mirror/test-suite/tree/master/MultiSource/Benchmarks/7zip>

#	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. 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. We can conclude that the branch on source line `dec.c:36` is the most executed in the benchmark since more than 50% of samples are attributed to it. Analyzing those two entries together gives us a misprediction rate for the branch. 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 an 88% prediction rate.

Linux `perf` calculates the misprediction rate by analyzing each LBR entry and extracting a misprediction bit from it. So 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, some branches might have an N entry but no corresponding Y entry. It means 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 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.

This type of analysis is not supported on AMD platforms since they don't record a cycle count in the LBR record. According to Arm's BRBE specification, it can be supported, but due to the unavailability of processors that implement this extension, it is not possible to verify. However, Intel supports it. Here is an example:

```
400618:  movb $0x0, (%rbp,%rdx,1)    <= start of the BB
40061d:  add $0x1, %rdx
400621:  cmp $0xc800000, %rdx
400628:  jnz 0x400644                <= end of the BB
```

Suppose we have the following entries in the LBR stack:

FROM_IP	TO_IP	Cycle Count	
...	<== 26 entries
40060a	400618	10	
400628	400644	80	<== occurrence 1
40064e	400600	3	
40060a	400618	9	
400628	400644	300	<== occurrence 2
40064e	400600	3	<== LBR TOS

Given that information, we have two occurrences of the basic block that starts at offset 400618. The first one was completed in 80 cycles, while the second one took 300 cycles. If we collect enough samples like that, we could plot an occurrence rate chart of latency for that basic block.

An example of such a chart is shown in Figure 6.4. It was compiled by analyzing relevant LBR entries. The way to read this chart is as follows: it tells what was the rate of occurrence of a given latency value. For example, the basic block latency was measured as 100 cycles roughly 2% of the time, 14% of the time we measured 280 cycles, and never saw anything between 150 and 200 cycles. Another way to read is: based on the collected data, what is the probability of seeing a certain basic block latency if you were to measure it?

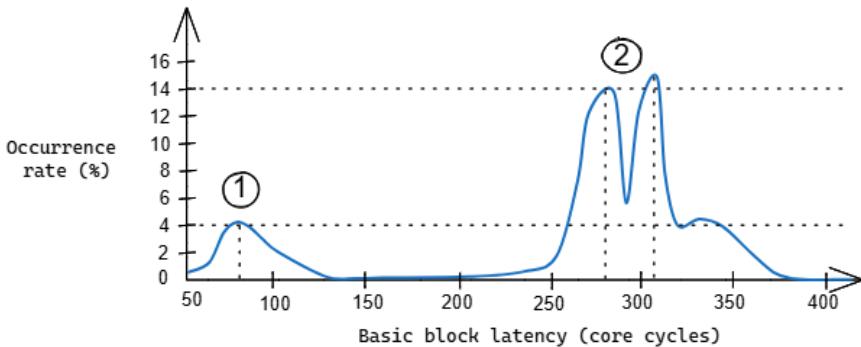


Figure 6.4: Occurrence rate chart for latency of the basic block that starts at address 0x400618.

We can see two humps: a small one around 80 cycles (1) and two bigger ones at 280 and 305 cycles (2). The block has a random load from a large array that doesn't fit in the CPU L3 cache, so the latency of the basic block largely depends on this load. Based on the chart we can conclude that the first spike (1) corresponds to the L3 cache hit and the second spike (2) corresponds to the 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 8.5. Also, cycle count information can be used for timing loop iterations, where every loop iteration ends with a taken branch (back edge).

Before the proper support from profiling tools was in place, building graphs similar to Figure 6.4 required manual parsing of raw LBR dumps. An example of how to do this can be found on the [easyperf blog](#)¹²³. Luckily, in newer versions of Linux perf, obtaining 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:

```
$ 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
```

¹²³ 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>.

#	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 the `perf report` to make it fit on the page. Let's focus on lines in which the source and destination are `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 point to the same line.

Linux `perf` sorts entries by overhead first, so we need to manually filter entries for the branch in which we are interested. 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 6.1. This data can be plotted to obtain a chart similar to the one shown in Figure 6.4.

Table 6.1: Occurrence rate for basic block latency.

Cycles	Number of samples	Occurrence rate
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 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 fall through¹²⁴ generally improves the performance of a program. Knowing 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 was not taken. This feature especially shines when analyzing indirect jumps (switch statements) and indirect calls (virtual calls). You can find examples of using it on a real-world application on the [easyperf blog](#)¹²⁵.

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 matures.

6.3 Hardware-Based Sampling Features

Major CPU vendors provide a set of additional features to enhance sampling. 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 detail, including their similarities and differences.

6.3.1 PEBS on Intel Platforms

Similar to the Last Branch Record feature, 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 the Intel Skylake CPU is shown in Figure 6.5. It contains the state of general-purpose

¹²⁴I.e., when the hot branches are not taken.

¹²⁵Easyperf: Estimating Branch Probability - <https://easyperf.net/blog/2019/05/06/Estimating-branch-probability>

registers (EAX, EBX, ESP, etc.), EventingIP, Data Linear Address, and Latency value, and a few other fields. The content layout of a PEBS record varies across different microarchitectures, see [Intel, 2023, 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 6.5: PEBS Record Format for 6th Generation, 7th Generation and 8th Generation Intel Core Processor Families. © Source: [Intel, 2023, Volume 3B, Chapter 20].

Since Skylake, the PEBS record has been enhanced to collect XMM registers and Last Branch Record (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 recording overhead. 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 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 the 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 load instructions using PEBS. When a performance counter is configured for PEBS, an overflow condition in the counter will not trigger an interrupt, instead, it will activate 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 does the processor raise an interrupt and the buffer gets flushed to memory. This mechanism lowers the sampling overhead by triggering 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 the entire contents of the 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 a 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, you can use `pebs-grabber`¹²⁶.

6.3.2 IBS on AMD Platforms

Instruction-Based Sampling (IBS) is an AMD64 processor feature that can be used to collect specific metrics related to instruction fetch and instruction execution. The pipeline of an AMD processor consists of two separate phases: a Frontend phase that fetches AMD64 instruction bytes and a Backend 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 Frontend 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 Backend 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), and load/store (hit or miss in D-caches and DTLB, linear address, load latency).

There are several 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 Execute counters can be enabled/disabled independently. With PMC, the 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. A more detailed comparison of Intel PEBS and AMD IBS can be found in [Sasongko et al., 2023].

Since IBS is integrated into the processor pipeline and acts as a fixed event counter, the sample collection overhead 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/Execute 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.

IBS is supported by Linux `perf` and the AMD uProf profiler. Here are sample commands to collect IBS Execute and Fetch samples:

```
$ perf record -a -e ibs_op/cnt_ctl=1,l3missonly=1/ -- benchmark.exe
```

¹²⁶ PEBS grabber tool - <https://github.com/andikleen/pmu-tools/tree/master/pebs-grabber>. Requires root access.

```
$ perf record -a -e ibs_fetch/l3missonly=0/ -- benchmark.exe
$ perf report
```

where `cnt_ctl=0` counts clock cycles, `cnt_ctl=1` counts dispatched ops for an interval period; `l3missonly=1` only keeps the samples that had an L3 miss. These two parameters and a few others are described in more detail in [AMD, 2024, Table 25. AMDuProfCLI Collect Command Options]. Note that in both of the commands above, the `-a` option is used to collect IBS samples for all cores, otherwise `perf` would fail to collect samples on Linux kernel 6.1 or older. From version 6.2 onwards, the `-a` option is no longer needed unless you want to collect IBS samples for all cores. The `perf report` command will show samples attributed to functions and source code lines similar to regular PMU events but with added features that we will discuss later. AMD uProf command line tool can generate IBS raw data, which later can be converted to a CSV file for later postprocessing with MS Excel as described in [AMD, 2024, Section 7.10 ASCII Dump of IBS Samples].

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. The SPE feature extension is specified as part of Armv8-A architecture, with support from Arm v8.2 onwards. Arm SPE extension is architecturally optional, which means that Arm processor vendors are not required to implement it. Arm Neoverse cores have supported SPE since Neoverse N1 cores, which were introduced in 2019.

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 the execution of this operation. SPE record saves the address of the instruction, the virtual and physical address for the data accessed by loads and stores, the source of the data access (cache or DRAM), and the 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 [Arm, 2023b] describes Arm SPE in more detail 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 postfiltering of sample records, which helps to reduce the memory required for storage. SPE profiling is supported in Linux `perf` and can be used as follows:¹²⁷

```
$ perf record -e arm_spe_0/<controls>/ -- test_program
```

¹²⁷ Linux `perf` driver for `arm_spe` should be installed first (see <https://developer.arm.com/documentation/ka005362/latest/>). On Amazon Linux 2 and 2023, the SPE PMU is available by default on Graviton metal instances (see https://github.com/aws/aws-graviton-getting-started/blob/main/perfrunbook/debug_hw_perf.md)

```
$ perf report --stdio
$ spe-parser perf.data -t csv
```

where <controls> lets you optionally specify various controls and filters for the collection. `perf report` will give the usual output according to what the user asked for with <controls> options. `spe-parser`¹²⁸ is a tool developed by Arm engineers to parse the captured perf record data and save all the SPE records into a CSV file.

Now that we covered the advanced sampling features, let's discuss how they can be used to improve performance analysis.

6.3.4 Precise Events

One of the major problems in sampling 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 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 large number of cache misses and a 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 the underlying reasons for such issues on [Intel Developer Zone website](#)¹²⁹.

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 a specific microarchitecture can be found in [Intel, 2023, Volume 3B, Chapter 20 Performance Monitoring]. An example of using PEBS precise events to mitigate skid can be found on the [easyperf blog](#).¹³⁰

Listed below are precise events for the Intel Skylake Microarchitecture:

<code>INST_RETIRE.*</code>	<code>OTHER_ASSISTS.*</code>	<code>BR_INST_RETIRE.*</code>	<code>BR_MISP_RETIRE.*</code>
<code>FRONTEND_RETIRE.*</code>	<code>HLE_RETIRE.*</code>	<code>RTM_RETIRE.*</code>	<code>MEM_INST_RETIRE.*</code>
<code>MEM_LOAD_RETIRE.*</code>	<code>MEM_LOAD_L3_HIT_RETIRE.*</code>		

, where `.*` means that all sub-events inside a group can be configured as precise events.

¹²⁸ Arm SPE parser - <https://gitlab.arm.com/telemetry-solution/telemetry-solution>

¹²⁹ Hardware event skid - <https://software.intel.com/en-us/vtune-help-hardware-event-skid>

¹³⁰ Performance skid - <https://easyperf.net/blog/2018/08/29/Understanding-performance-events-skid>

With AMD IBS and Arm SPE, all the collected samples are precise by design since the hardware captures the exact instruction address. They both 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, the IBS and SPE sample's instruction attribution is precise.

Users of Linux `perf` on Intel and AMD platforms must add the `pp` suffix to one of the events listed above to enable precise tagging as shown below. However, on Arm platforms, it has no effect, so users must use the `arm_spe_0` event.

```
$ perf record -e cycles:pp -- ./a.exe
```

Precise events provide relief for performance engineers as they help to avoid misleading data that often confuses beginners and even senior developers. The TMA methodology heavily relies on precise events to locate the exact line of source code where the inefficient execution takes place.

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 sample loads and collect their target addresses and access latency. Keep in mind, that 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 samples per second you want. With a large enough collection of samples, it can give an accurate statistical picture.

In PEBS, such a feature 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 6.5). 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 an 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, its latency, where the memory location was fetched from (cache or DRAM), and whether it hit or missed in the DTLB. SPE can be used to estimate the latency and bandwidth 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 13.4. 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 occurs in a cache line modified by other threads.

Questions and Exercises

1. Name the four level-1 categories in the TMA performance methodology.
2. What are the benefits of hardware event-based sampling?
3. What is a performance event skid?
4. Study performance analysis features available on the CPU inside the machine you use for development/benchmarking.

Chapter Summary

- Modern processors provide features that enhance performance analysis. Using those features greatly simplifies finding opportunities for low-level optimization.
- Top-down Microarchitecture Analysis (TMA) methodology is a powerful technique for identifying ineffective usage of CPU microarchitecture by a program, and 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 one of the starting points for every low-level tuning effort.
- Branch Record mechanisms such as Intel’s LBR, AMD’s LBR, and ARM’s BRBE continuously log the most recent branch outcomes in parallel with executing the program, causing a minimal slowdown. One of the primary usages of these facilities is to collect call stacks. Also, they help identify hot branches, and misprediction rates and enable precise timing of machine code.
- Modern processors often provide Hardware-Based Sampling features for advanced profiling. Such features lower the sampling overhead by storing multiple samples in a dedicated buffer without software interrupts. They also introduce “Precise Events” that enable pinpointing the exact instruction that caused a particular performance event. In addition, there are several other less important use cases. Example implementations of such Hardware-Based Sampling features include Intel’s PEBS, AMD’s IBS, and ARM’s SPE.
- 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 a relatively small overhead. Its main usages are postmortem analysis and finding the root cause(s) of performance glitches. The Intel PT feature is covered in Appendix C. Processors based on ARM architecture also have a tracing capability called Arm CoreSight,¹³¹ but it is mostly used for debugging rather than for performance analysis.

¹³¹ 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 hardware performance monitoring features. This makes profiling go smoothly, but it's still critical to have an intuition of how such tools obtain and interpret the data. That is why we now discuss analysis tools only after we have discussed CPU performance monitoring features.

This chapter gives a quick overview of the most popular performance analysis tools available on major platforms. Some of the tools are cross-platform but the majority are not, so it is important to know what tools are available to you. Profiling tools are usually developed and maintained by hardware vendors themselves because they are the ones who know how to properly use performance monitoring features available on their processors. So, the choice of a tool for advanced performance engineering work depends on which operating system and CPU you're using.

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 daily. Even if you don't find any actionable insights, you will be much better prepared when the actual need arises.

7.1 Intel VTune Profiler

VTune Profiler (formerly VTune Amplifier) is a performance analysis tool for x86-based machines with a rich graphical 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 are quickly becoming obsolete.

VTune can be used on both Intel and AMD systems. However, advanced hardware-based sampling requires an Intel-manufactured CPU. For example, you won't be able to collect hardware 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.¹³²

How to configure it

On Linux, VTune can use two data collectors: Linux perf and VTune's own driver called SEP. The first type is used for user-mode sampling, but if you want to perform advanced analysis, you need to build and install the SEP driver, which is not too hard.

```
# go to the sepdk folder in vtune's installation
$ cd ~/intel/oneapi/vtune/latest/sepdk/src
```

¹³² Intel oneAPI Base Toolkit - <https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit.html>

```
# 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 advanced 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 privileges.

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 behavior over time (including 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 aggregated data over a period of time, 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 software dynamics.

Due to the sampling nature of the tool, it will eventually miss events with a very short duration (e.g., sub-microsecond).

Example

Below is a series of screenshots of VTune's most interesting features. For this example, I took POV-Ray, which is a ray tracer used to create 3D graphics. Figure 7.1 shows the hotpots analysis of the built-in POV-Ray 3.7 benchmark, compiled with clang14 compiler with `-O3 -ffast-math -march=native -g` options, and run on an Intel Alder Lake 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 the corresponding CPU time percentage and the number of retired instructions. On the right panel, you can see the most frequent call stack that leads to calling the function `pov::Noise`. According to that screenshot, 44.4% of the time function

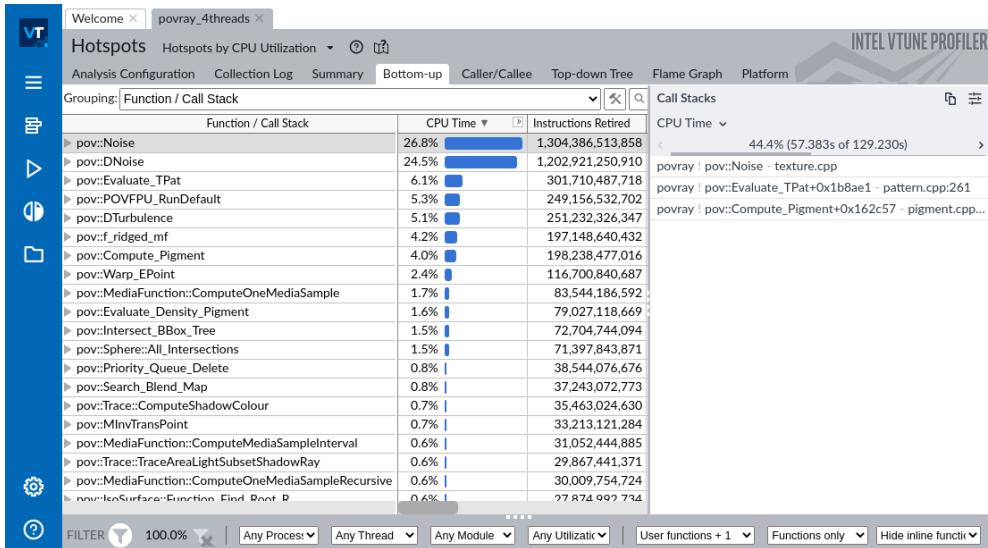


Figure 7.1: VTune’s hotspots view of povray built-in benchmark.

`pov::Noise`, was called from `pov::Evaluate_TPat`, which in turn was called from `pov::Compute_Pigment`.¹³³

If you double-click on the `pov::Noise` function, you will see an image that is shown in Figure 7.2. For the interest of space, only the most important columns are shown. The left panel shows the source code and CPU time that corresponds to each line of code. On the right, you can see assembly instructions along with the 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 (not just the ones that are visible) in each panel equals to the total CPU time attributed to the `pov::Noise` function, which is 26.8%.

When you use VTune to profile applications running on Intel CPUs, it can collect many different performance events. To illustrate this, I ran a different analysis type, Microarchitecture Exploration. To access raw event counts, you can switch the view to Hardware Events as shown in Figure 7.3. To enable switching views, you need to tick the mark in *Options → General → Show all applicable viewpoints*. Near the top of Figure 7.3, you can see that the *Platform* tab is selected. Two other pages are also useful. The *Summary* page gives you the absolute number of raw performance events as collected from CPU counters. The *Event Count* page gives you the same data with a per-function breakdown.

Figure 7.3 is quite busy and requires some explanation. The top panel, indicated with (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

¹³³ Notice that the call stack doesn’t lead all the way to the `main` function. This happens because, with the hardware-based collection, VTune uses LBR to sample call stacks, which provides limited depth. Most likely we’re dealing with recursive functions here, and to investigate that further users will have to dig into the code.

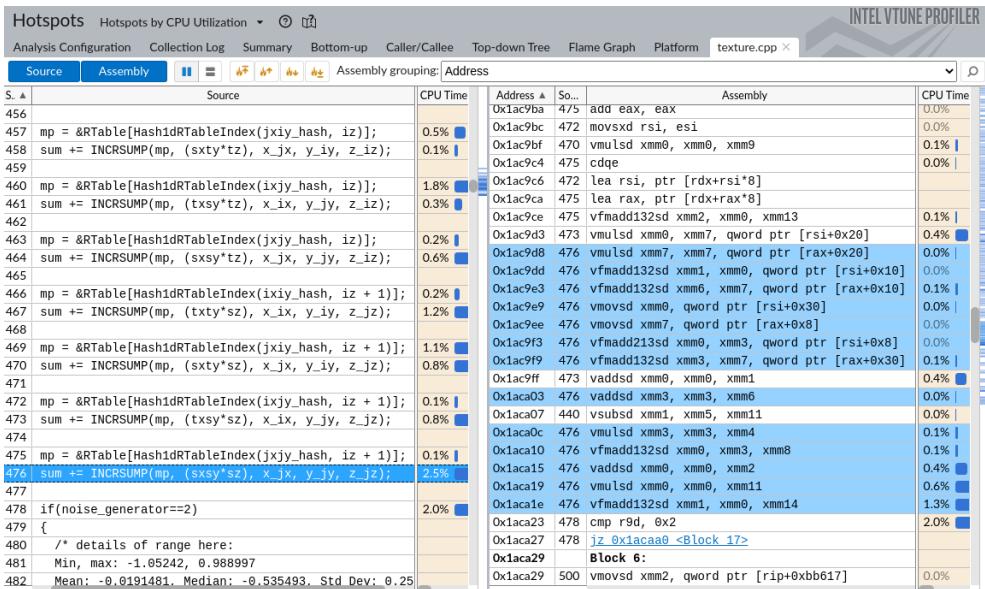


Figure 7.2: VTune’s source code view of povray built-in benchmark.

(TID: 3102135), which spawns all the worker threads. The higher the black bar, the more events (L1 cache misses in this case) happened at any given moment. Notice occasional 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. The region indicated with ② is an example of such filtering. To see the updated list of functions, you can go to the *Event Count* view. Such filtering and zooming features are available on all VTune timeline views.

The region indicated with ③ shows performance events that were collected and their distribution over time. This time it is not a per-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_RETIRE_ALL_BRANCHES**), but the misprediction rate is quite low (**BR_MISP_RETIRE_ALL_BRANCHES**). This can lead you to the conclusion that branch misprediction is not a bottleneck for POV-Ray. If you scroll down, you will see that the number of L3 misses is zero, and L2 cache misses are very rare as well. This tells us that 99% of memory access requests are served by L1, and the rest of them are served by L2. By combining these two observations, we can conclude that the application is likely bound by compute, i.e., the CPU is busy calculating something, not waiting for memory or recovering from a misprediction.

Finally, the bottom panel 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 3.2–3.4GHz region.

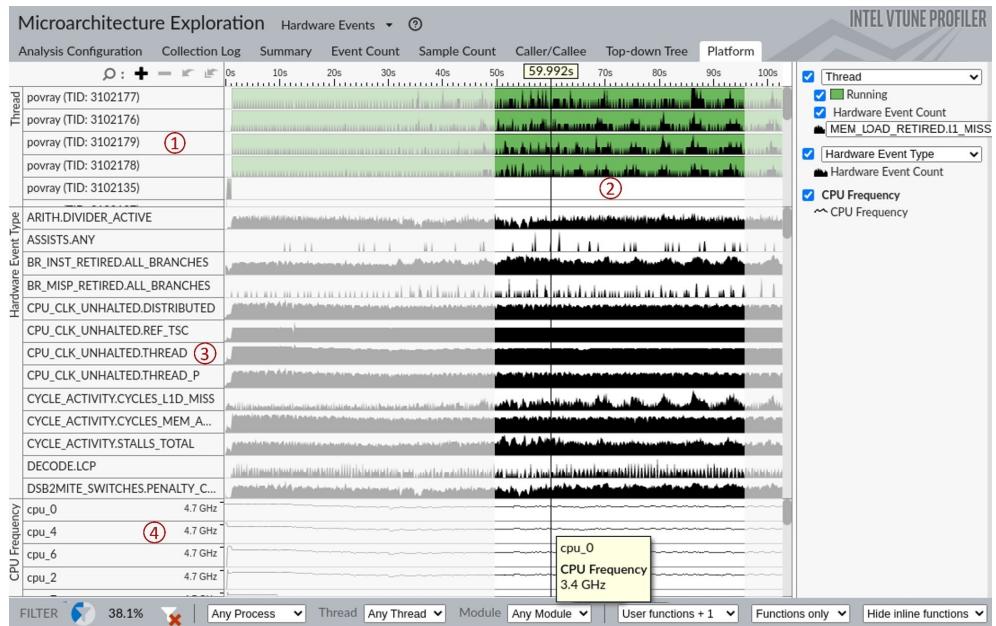


Figure 7.3: VTune's perf events timeline view of povray built-in benchmark.

7.2 AMD uProf

The [uProf](#) profiler is a tool developed by AMD for monitoring the 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, and 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.

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. The CLI interface requires two separate steps—collect and report, similar to Linux perf.

What you can do with it:

- Find hotspots: functions, statements, instructions.
- Monitor various hardware 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

the timeline chart.

- Analyze hot callpaths: call graph, flame graph, and bottom-up charts.

In addition, uProf can monitor various OS events on Linux: thread state, thread synchronization, system calls, page faults, and others. You can use it to analyze OpenMP applications to detect thread imbalance and analyze MPI¹³⁴ applications to detect the load imbalance among the nodes of the MPI cluster. More details on various features of uProf can be found in the [User Guide](#)¹³⁵.

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 the AMD uProf tool, we ran the dense LU matrix factorization component from the [Scimark2](#)¹³⁶ benchmark on an AMD Ryzen 9 7950X, running Windows 11, with 64 GB RAM.

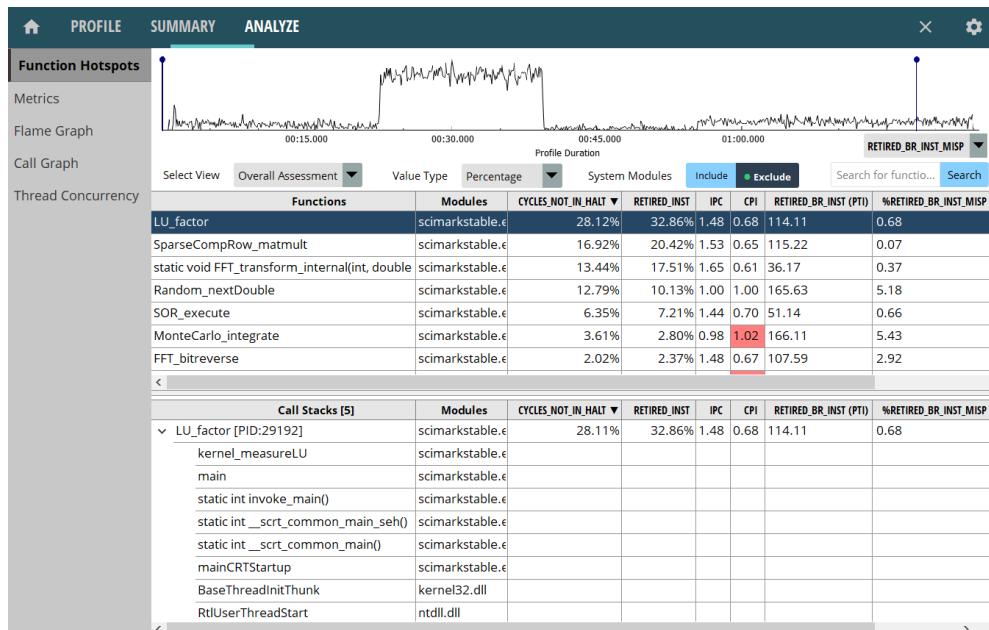


Figure 7.4: uProf's Function Hotspots view.

Figure 7.4 shows *Function Hotspots* analysis (selected in the menu list on the left side of the image). At the top of the image, you can see an event timeline showing the

¹³⁴ MPI - Message Passing Interface, a standard for parallel programming on distributed memory systems.

¹³⁵ AMD uProf User Guide - <https://www.amd.com/en/developer/uprof.html#documentation>

¹³⁶ Scimark2 - <https://math.nist.gov/scimark2/index.html>

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, or percentage. There are many interesting numbers to look at, but we will not dive deep into the analysis. Instead, readers are encouraged to figure out the performance impact of branch mispredictions and find their source.

Below the functions table, you can see a bottom-up call stack view for the selected function 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 the Scimark2 benchmark, this is the only call stack for LU_factor, even though it shows Call Stacks [5]. This is an artifact of collection that can be ignored. But in other applications, a 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 convenient way to do similar performance analysis on MacOS is to use Xcode Instruments. This is an application performance analyzer and visualizer that comes for free with Xcode. The Instruments profiler is built on top of the DTrace tracing framework that was ported to MacOS from Solaris. It has many tools to inspect the performance of an application and enables 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 App Store. 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 powerful 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 enables 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. You can read more about those use cases in the Instruments documentation.¹³⁷

¹³⁷ Instruments documentation - <https://help.apple.com/instruments/mac/current>

What you can do with it:

- Access hardware performance counters on Apple processors.
- Find hotspots in a program along with their 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:

Similar to other sampling-based profilers, Xcode Instruments has the same blind spots as VTune and uProf.

Example: Profiling Clang Compilation

In this example, I will show how to collect hardware performance counters on an Apple Mac mini with the M1 processor, macOS 13.5.1 Ventura, and 16 GB RAM. I took one of the largest files in the LLVM codebase and profiled its compilation using version 15.0 of the Clang C++ compiler.

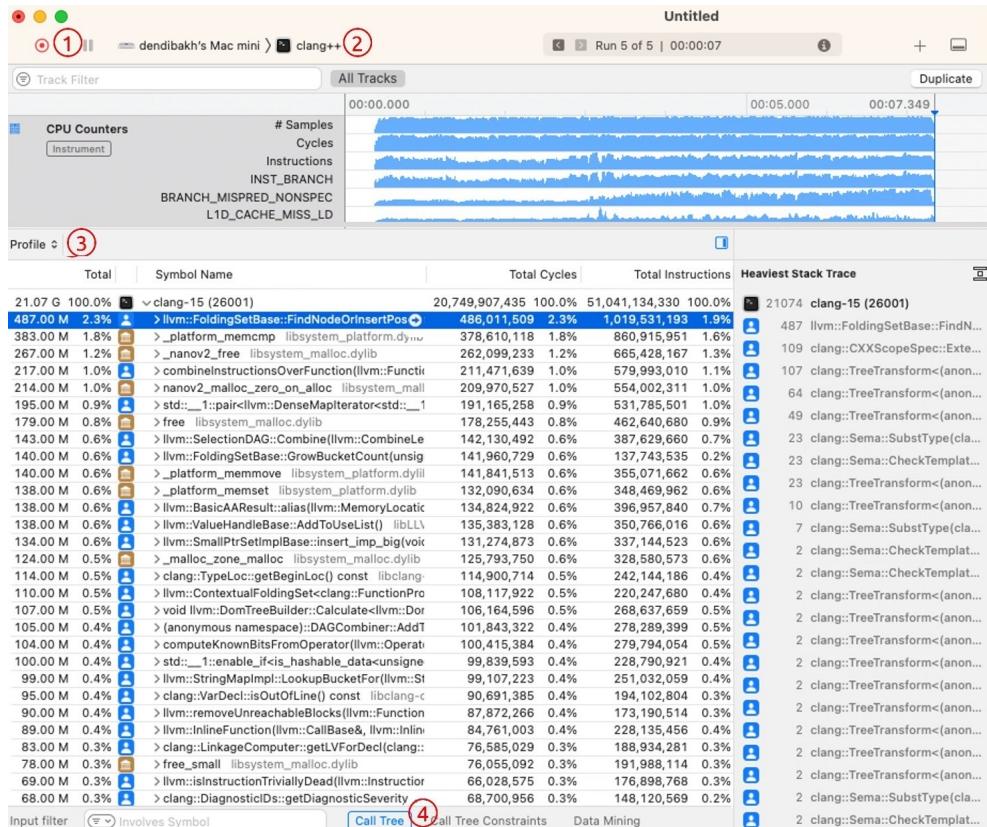


Figure 7.5: Xcode Instruments: timeline and statistics panels.

Here is the command line that I used:

```
$ clang++ -O3 -DNDEBUG -arch arm64 <other options ...> -c
    llvm/lib/Transforms/Vectorize/LoopVectorize.cpp
```

Figure 7.5 shows the main timeline view of Xcode Instruments. This screenshot was taken after the compilation had finished. We will get back to it a bit later, but first, let us show how to start the profiling session.

To begin, open *Instruments* and choose the *CPU Counters* analysis type. The first step you need to do is configure the collection. Click and hold the red target icon (see ① in Figure 7.5), then select *Recording Options...* from the menu. It will display the dialog window shown in Figure 7.6. This is where you can add hardware performance monitoring events for collection. Apple has documented its hardware performance monitoring events in its manual [Apple, 2024, Section 6.2 Performance Monitoring Events].

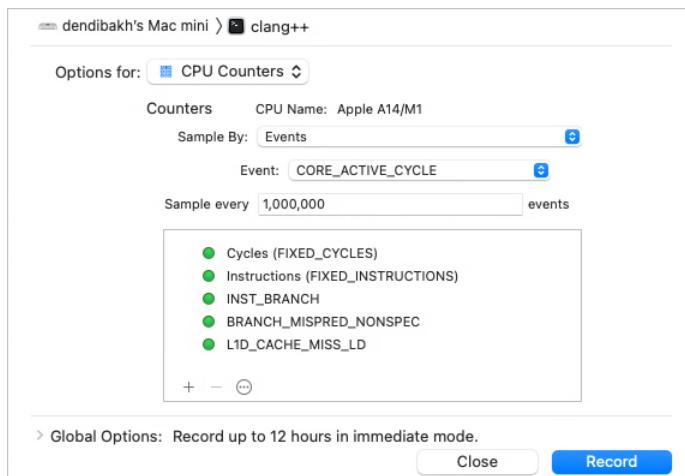


Figure 7.6: Xcode Instruments: CPU Counters options.

The second step is to set the profiling target. To do that, click and hold the name of an application (marked ② in Figure 7.5) 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 results like those shown in Figure 7.5. The compilation took 7.3 seconds and we can see how the volume of events changed over time. For example, the number of executed branch instructions and mispredictions increased 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 in Figure 7.5.

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 my 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 a discussion of how to do so is beyond the scope of this book.

7.4 Linux Perf

Linux Perf is probably the most used performance profiler in the world since it is available on most Linux distributions, which makes it accessible to a wide range of users. Perf is natively supported in many popular Linux distributions, including Ubuntu, Red Hat, and Debian. It is included in the kernel, so you can get OS-level statistics (page faults, CPU migrations, etc.) on any system that runs Linux. As of mid-2024, the profiler supports x86, ARM, PowerPC64, UltraSPARC, and a few other CPU types.¹³⁸ On such platforms, `perf` provides access to the hardware performance monitoring features, for example, performance counters. More information about Linux `perf` is available on its [wiki page](#)¹³⁹.

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
$ echo kernel.perf_event_paranoid=0 | sudo tee -a /etc/sysctl.d/local.conf
# Enable kernel modules symbols resolution for unprivileged users
$ echo 0 | sudo tee /proc/sys/kernel/kptr_restrict
$ echo kernel.kptr_restrict=0 | sudo tee -a /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 prioritize 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 people use. The first, `perf stat`, reports a count of specified performance events. The second, `perf record`, profiles an application or system in sampling mode and is often followed by `perf report` to generate a report from the sampling data.

The output of the `perf record` command is a raw dump of samples. Many tools, built on top of Linux `perf`, parse raw dump files and provide new analysis types. Here are the most notable ones:

- Flame graphs, discussed in Section 7.5.

¹³⁸ RISCV is not supported yet as a part of the official kernel, although custom tools from vendors exist.

¹³⁹ Linux `perf` wiki - https://perf.wiki.kernel.org/index.php/Main_Page.

- KDAB Hotspot,¹⁴⁰ a tool that visualizes Linux `perf` data with an interface very similar to Intel VTune. If you have worked with Intel VTune, KDAB Hotspot will seem very familiar to you.
- Netflix `Flamescope`.¹⁴¹ This tool displays a 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 flame graph for that time range.

What you cannot do with it:

Linux perf is a command-line tool and lacks a Graphical User Interface (GUI), which makes it hard to filter data, observe how the workload behavior changes over time, zoom into a portion of the runtime, etc. There is a limited console output provided through the `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 post-process and visualize the raw output of Linux `perf`.

7.5 Flame Graphs

A flame graph is a popular way of visualizing profiling data and the most frequent code paths in a program. It enables us to see which function calls take the largest portion of execution time. Figure 7.7 shows an example of a flame graph for the `x264` video encoding benchmark, generated with open-source `scripts`¹⁴² developed by Brendan Gregg. Nowadays, nearly all profilers can automatically generate a flame graph as long as the call stacks are collected during the profiling session.

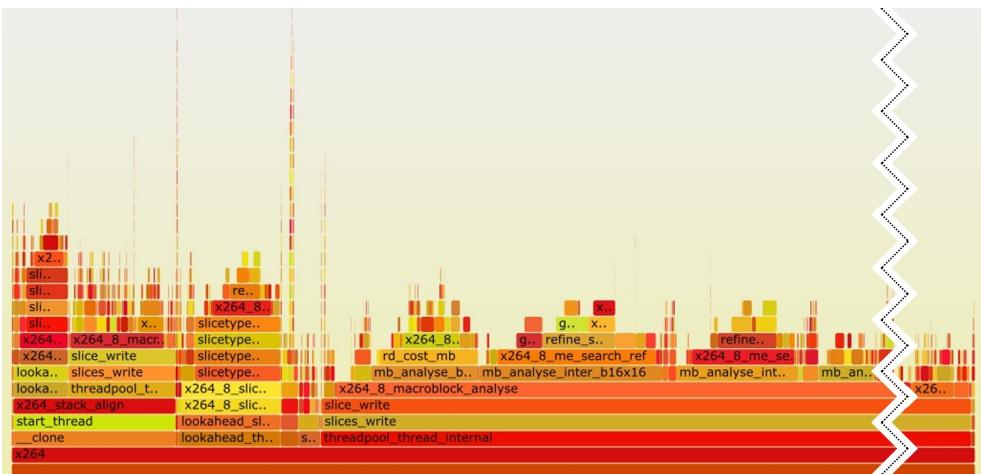


Figure 7.7: A flame graph for the `x264` benchmark.

On the flame graph, each rectangle (horizontal bar) represents a function call, and the

¹⁴⁰ KDAB Hotspot - <https://github.com/KDAB/hotspot>.

¹⁴¹ Netflix Flamescope - <https://github.com/Netflix/flamescope>.

¹⁴² Flame graphs by Brendan Gregg - <https://github.com/brendangregg/FlameGraph>

width of the rectangle indicates the relative execution time taken by the function itself and by its callees. The function calls happen from the 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 out what is the hottest function in the program (not immediately clear from this flame graph) and go bottom-up through the flame graph, trying to understand from where this hottest function gets called.

Some tools prefer to use an *icicle graph*, which is the upside-down version of a flame graph (see an example in Section 7.9).

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 use in analyzing general-purpose applications as well. ETW is available on all supported Windows platforms (x86 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 enables you to observe software 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 an administrator and have the `SeSystemProfilePrivilege` enabled. The `Windows Performance Recorder` tool supports a set of built-in recording profiles that are suitable 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 it from the Windows SDK¹⁴³ or ADK¹⁴⁴ download page. The 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.

¹⁴³ Windows SDK Downloads - <https://developer.microsoft.com/en-us/windows/downloads/sdk-archive/>

¹⁴⁴ Windows ADK Downloads - <https://learn.microsoft.com/en-us/windows-hardware/get-started/adk-install#other-adk-downloads>

What you can do with it:

- Identify hotspots with a configurable CPU sampling rate from 125 microseconds up to 10 seconds. The default is 1 millisecond which costs approximately 5–10% runtime overhead.
- Determine what blocks a certain thread and for how long (e.g., late event signals, unnecessary thread sleep, etc).
- Examine how fast a disk serves read/write requests and discover what initiates that work.
- Check file access performance and patterns (including cached read/writes that lead to no disk IO).
- Trace the TCP/IP stack to see 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 inject enter/leave ETW tracing hooks in functions into your source code to measure how often a certain function was executed.

What you cannot do with it:

ETW traces are not useful for examining CPU microarchitectural bottlenecks. For that, use vendor-specific tools like Intel VTune, AMD uProf, Apple Instruments, etc.

ETW traces capture the dynamics of all processes at the system level, however, 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 overwriting previously stored traces.

If you'd like to learn more about ETW, there is a more detailed discussion in Appendix D. We explore tools to record and analyze ETW and present a case study of debugging a slow start of a program.

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 hotspots in your code, but in some cases, they might not provide the required 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.

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 enables you to observe the performance of a particular function or a block of code (later referred to as a *zone*). Continuing with the game industry, there are several 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,¹⁴⁵ as this seems to be one of the most 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 a “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 this example, we will use the ToyPathTracer¹⁴⁶ program, a simple path tracer, which is a simplified ray-tracing technique 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 diagnose the root cause of the problem, we have manually modified the code so that some frames will consume more time than others. Listing 7.1 shows an outline of the code along with added Tracy instrumentation. Notice, that we randomly select frames to slow down. Also, we included Tracy’s header and added the `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 a C++ object on the stack that will record the runtime activity of the code within the scope of the object’s lifetime. Tracy refers to this scope as a *zone*. At the zone entry, the current timestamp is captured. Once the function exits, the object’s destructor will record a new timestamp and store this timing data, along with the function name.

¹⁴⁵ Tracy - <https://github.com/wolfpld/tracy>

¹⁴⁶ ToyPathTracer - <https://github.com/wolfpld/tracy/tree/master/examples/ToyPathTracer>

Listing 7.1 Tracy Instrumentation

```
#include "tracy/Tracy.hpp"

void DoExtraWork() {
    ZoneScoped;
    // imitate useful work
}

void TraceRowJob() {
    ZoneScoped;
    if (frameCount == randomlySelected)
        DoExtraWork();
    // ...
}

void RenderFrame() {
    ZoneScoped;
    for (...) {
        TraceRowJob();
    }
    FrameMark;
}
```

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 live profiling data, also known as 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’s graphical interface is quite rich, but unfortunately contains too much detail to fit on a single screenshot, so we break it down into pieces. At the top, there is a timeline view as shown in Figure 7.8, cropped to fit onto the page. It shows only a portion of frame 76, which took 44.1 ms to render. On that diagram, we see the `Main thread` and five `WorkerThreads` 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 a real application, it may not be so obvious. Let's leave this for now.

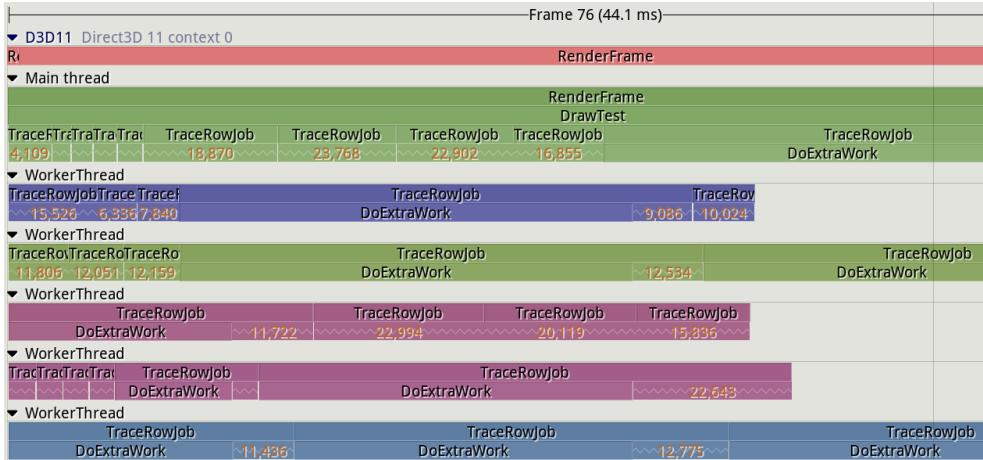


Figure 7.8: Tracy main timeline view. It shows the main thread and five worker threads while rendering a frame.

Right above the main panel, there is a histogram that displays the times for all the recorded frames (see Figure 7.9). 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, some frames take longer than this and are marked in red. As seen in the screenshot, a tooltip showing the details of a given frame is displayed when you point the mouse at a bar in the histogram. In this example, we are showing the details for the last frame.



Figure 7.9: Tracy frame timings. You can find frames that take more time to render than other frames.

Figure 7.10 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 on CPU 1 only for 4.4 ms during the entire run of the program.

Next comes the panel that provides information on where our program spends its time (hotspots). Figure 7.11 is a screenshot of Tracy's statistics window. We can check the recorded data, including the total time a given function was active, how many times

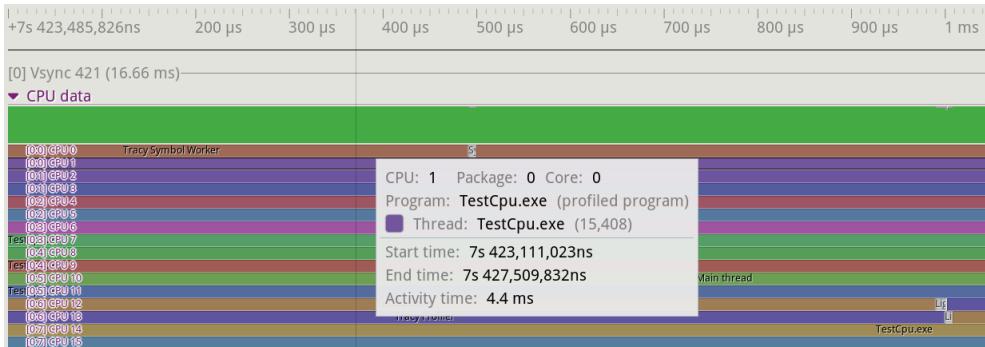


Figure 7.10: Tracy CPU data view. You can see what each CPU core was doing at any given moment.

it was invoked, etc. It’s also possible to select a time range in the main view to filter information corresponding to a time interval.

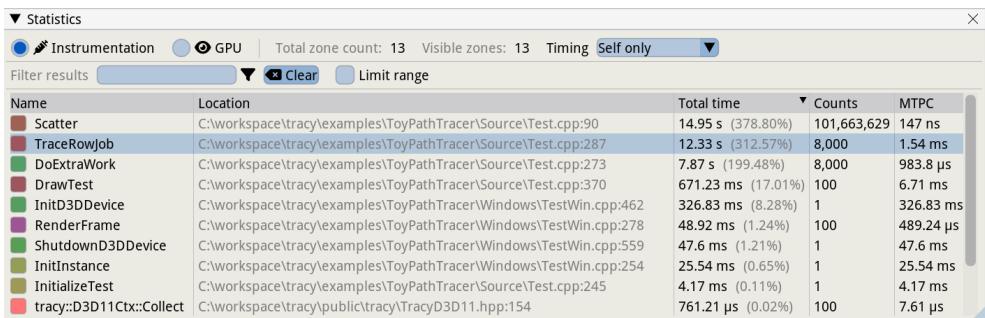


Figure 7.11: Tracy function statistics. A regular “hotspot” view that provides information where a program spends time.

The last set of panels that we show, enables us to analyze individual zone instances in more depth. Once you click on any zone instance, say, on the main timeline view or on the *CPU data* view, Tracy will open a *Zone Info* window (see the left panel in Figure 7.12) with the details for this zone instance. It shows 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 (self time) takes 1.36 ms, which is only 7%. The rest of the time is consumed by child zones.

It’s easy to spot a call to `DoExtraWork` that takes the bulk of the time, 16.99 ms out of 19.24 ms (see the left panel in Figure 7.12). Notice that this particular `TraceRowJob` instance runs almost 4.4 times as long as the average case (indicated by “437.93% of the mean time” on the image). Bingo! We found one of the slow instances where the `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.

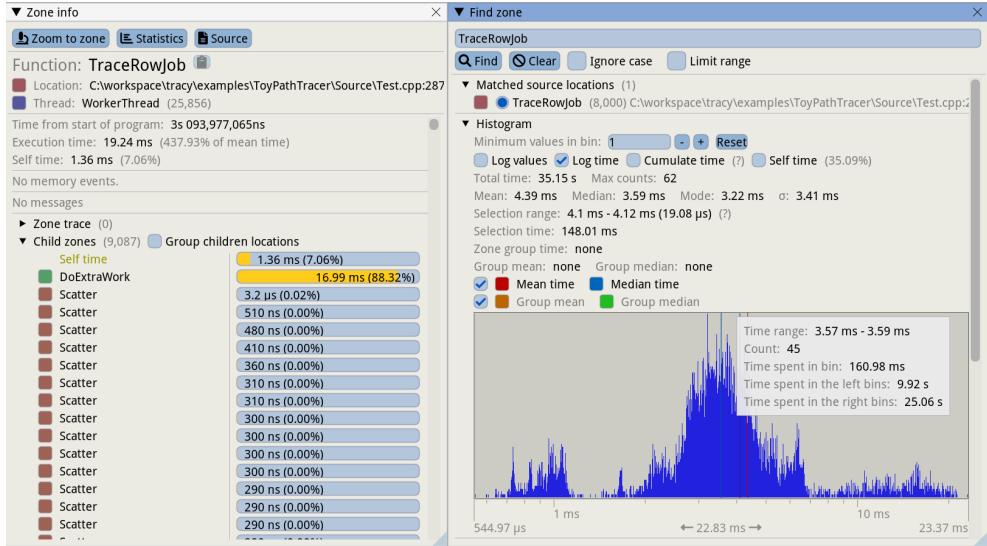


Figure 7.12: Tracy zone detail windows. It shows statistics for a slow instance of the `TraceRowJob` zone.

Remember, we saw in Figure 7.9, 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 (on the right of Figure 7.12). 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 determine the root cause of the issue. From this view, you can select one of the slow zones. This will update the *Zone Info* window 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 the 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 with 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, we used manual markup of interesting areas in the code. However, doing this is not a strict requirement to start using Tracy. You can 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 hardware 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 Memory Profiling

So far in this chapter, we have discussed tools that identify places where a program spends most of its time. In this section, we will focus on a program's interaction with memory. This is usually called *memory profiling*. In particular, we will learn how to collect memory usage, profile heap allocations and measure memory footprint. Memory profiling helps you understand how an application uses memory over time and helps you build an accurate mental model of a program's interaction with memory. Here are some questions it can answer:

- What is a program's total virtual memory consumption and how does it change over time?
- Where and when does a program make heap allocations?
- What are the code places with the largest amount of allocated memory?
- How much memory does a program access every second?
- What is the total memory footprint of a program?

7.8.1 Memory Usage

Memory usage is frequently described by Virtual Memory Size (VSZ) and Resident Set Size (RSS). VSZ includes all memory that a process can access, e.g., stack, heap, the memory used to encode instructions of an executable, and instructions from linked shared libraries, including the memory that is swapped out to disk/SSD. On the other hand, RSS measures how much memory allocated to a process resides in RAM. Thus, RSS does not include memory that is swapped out or was never touched yet by that process. Also, RSS does not include memory from shared libraries that were not loaded to memory. Files that are mapped to memory with `mmap` also contribute to VSZ and RSS usage.

Consider an example. Process A has 200K of stack and heap allocations of which 100K resides in the main memory; the rest is swapped out or unused. It has a 500K binary, from which only 400K was touched. Process A is linked against 2500K of shared libraries and has only loaded 1000K in the main memory.

VSZ: $200\text{K} + 500\text{K} + 2500\text{K} = 3200\text{K}$

RSS: $100\text{K} + 400\text{K} + 1000\text{K} = 1500\text{K}$

Developers can observe both RSS and VSZ on Linux with a standard `top` utility, however, both metrics can change very rapidly. Luckily, some tools can record and visualize memory usage over time. Figure 7.13 shows the memory usage of the PSPNet image segmentation algorithm, which is a part of the AI Benchmark Alpha.¹⁴⁷ This chart was created based on the output of a tool called `memory_profiler`¹⁴⁸, a Python library built on top of the cross-platform `psutil`¹⁴⁹ package.

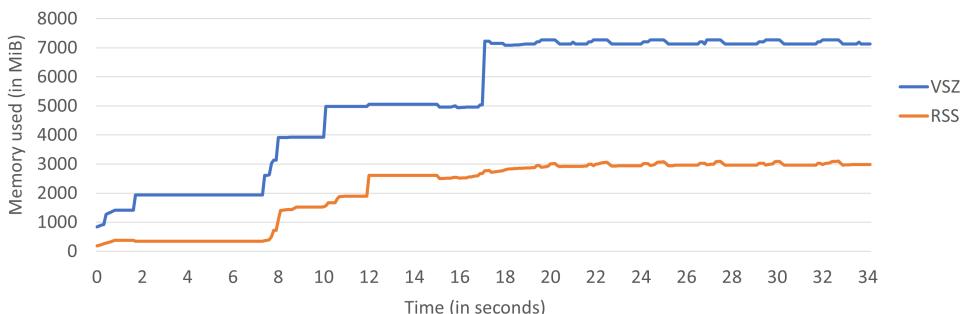


Figure 7.13: RSS and VSZ memory utilization of AI_bench PSPNet image segmentation.

In addition to standard RSS and VSZ metrics, people have developed a few more sophisticated metrics. Since RSS includes both the memory that is unique to the process and the memory shared with other processes, it's not clear how much memory a process has for itself. The USS (Unique Set Size) is the memory that is unique to a process and which would be freed if the process was terminated right now. The PSS (Proportional Set Size) represents unique memory plus the amount of shared memory, evenly divided between the processes that share it. E.g. if a process has 10

¹⁴⁷ AI Benchmark Alpha - <https://ai-benchmark.com/alpha.html>

¹⁴⁸ Easyper blog: Measuring memory footprint with SDE - <https://easyper.net/blog/2024/02/12/Memory-Profiling-Part3>

¹⁴⁹ psutil - <https://github.com/giampaolo/psutil>

MB all to itself (USS) and 10 MBs shared with another process, its PSS will be 15 MBs. The `psutil` library supports measuring these metrics (Linux-only), which can be visualized by `memory_profiler`.

On Windows, similar concepts are defined by Committed Memory Size and Working Set Size. They are not direct equivalents to VSZ and RSS but can be used to effectively estimate the memory usage of Windows applications. The `RAMMap`¹⁵⁰ tool provides a rich set of information about memory usage for the system and individual processes.

When developers talk about memory consumption, they implicitly mean heap usage. Heap is, in fact, the biggest memory consumer in most applications as it accommodates all dynamically allocated objects. But heap is not the only memory consumer. For completeness, let's mention others:

- Stack: Memory used by stack frames in an application. Each thread inside an application gets its own stack memory space. Usually, the stack size is only a few MB, and the application will crash if it exceeds the limit. For example, the default size of stack memory on Linux is usually 8MB, although it may vary depending on the distribution and kernel settings. The default stack size on macOS is also 8MB, but on Windows, it's only 1 MB. The total stack memory consumption is proportional to the number of threads running in the system.
- Code: Memory that is used to store the code (instructions) of an application and its libraries. In most cases, it doesn't contribute much to memory consumption, but there are exceptions. For example, the Clang 17 C++ compiler has a 33 MB code section, while the latest Windows Chrome browser has 187MB of its 219MB `chrome.dll` dedicated to code. However, not all parts of the code are frequently exercised while a program is running. We show how to measure code footprint in Section 11.9.

Since the heap is usually the largest consumer of memory resources, it makes sense for developers to focus on this part of memory when they analyze the memory utilization of their applications. In the following section, we will examine heap consumption and memory allocations in a popular real-world application.

7.8.2 Case Study: Analyzing Stockfish's Heap Allocations

In this case study, I use `heaptrack`¹⁵¹, an open-sourced heap memory profiler for Linux developed by KDE. Ubuntu users can install it very easily with `apt install heaptrack heaptrack-gui`. `Heaptrack` can find places in the code where the largest and most frequent allocations happen among many other things. On Windows, you can use `MTuner`¹⁵² which has similar capabilities to `Heaptrack`.

I analyzed `Stockfish`'s¹⁵³ chess engine built-in benchmark, which we have already looked at in Section 4.11. As before, I compiled it using the Clang 15 compiler with `-O3 -mavx2` options. I collected the `Heaptrack` memory profile of a single-threaded Stockfish built-in benchmark on an Intel Alder Lake i7-1260P processor using the following command:

¹⁵⁰ RAMMap - <https://learn.microsoft.com/en-us/sysinternals/downloads/rammap>

¹⁵¹ Heaptrack - <https://github.com/KDE/heaptrack>

¹⁵² MTuner - <https://github.com/milostotic/MTuner>

¹⁵³ Stockfish - <https://github.com/official-stockfish/Stockfish>

```
$ heaptrack ./stockfish bench 128 1 24 default depth
```

Figure 7.14 shows us a summary view of the Stockfish memory profile. Here are some interesting facts we can learn from it:

- The total number of allocations is 10614.
- Almost half of the allocations are temporary, i.e., allocations that are directly followed by their deallocation.
- Peak heap memory consumption is 204 MB.
- `Stockfish::std_aligned_alloc` is responsible for the largest portion of the allocated heap space (182 MB). But it is not among the most frequent allocation spots (middle table), so it is likely allocated once and stays alive until the end of the program.
- Almost half of all the allocation calls come from `operator new`, which are all temporary allocations. Can we get rid of temporary allocations? You will find out later.
- Leaked memory is not a concern for this case study.

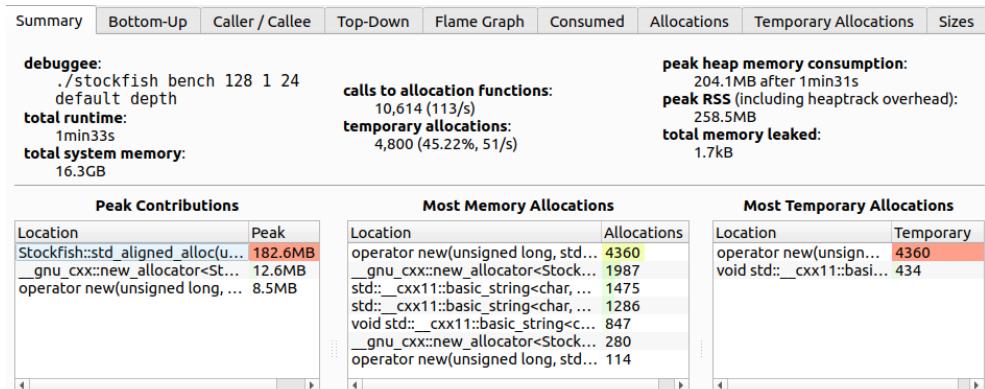


Figure 7.14: Stockfish memory profile with Heaptrack, summary view.

Notice, that there are many tabs on the top of the image; we will explore some of them. Figure 7.15 shows the memory usage of the Stockfish built-in benchmark. The memory usage stays constant at 200 MB throughout the entire run of the program. Total consumed memory is broken into slices, e.g., regions ① and ② on the image. Each slice corresponds to a particular allocation. Interestingly, it was not a single big 182 MB allocation that was done through `Stockfish::std_aligned_alloc` as we thought earlier. Instead, there are two: slice ① 134.2 MB and slice ② 48.4 MB. Both allocations stay alive until the very end of the benchmark.

Does it mean that there are no memory allocations after the startup phase? Let's find out. Figure 7.16 shows the accumulated number of allocations over time. Similar to the consumed memory chart (Figure 7.15), allocations are sliced according to the accumulated number of memory allocations attributed to each function. As we can see, new allocations keep coming from not just a single place, but many. The most frequent allocations are done through `operator new` that corresponds to region ① on the image.

Notice there are new allocations at a steady pace throughout the life of the program.

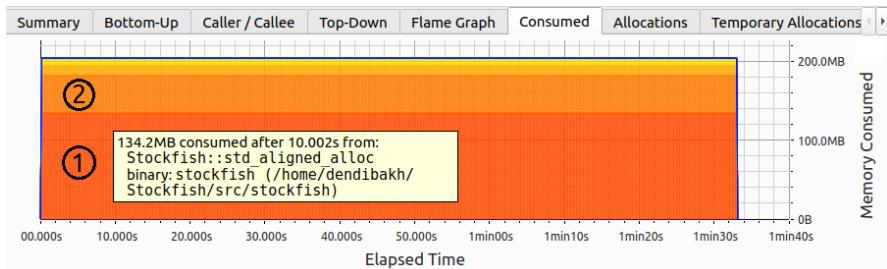


Figure 7.15: Stockfish memory profile with Heaptrack, memory usage over time stays constant.

However, as we just saw, memory consumption doesn't change; how is that possible? Well, it can be possible if we deallocate previously allocated buffers and allocate new ones of the same size (also known as *temporary allocations*).

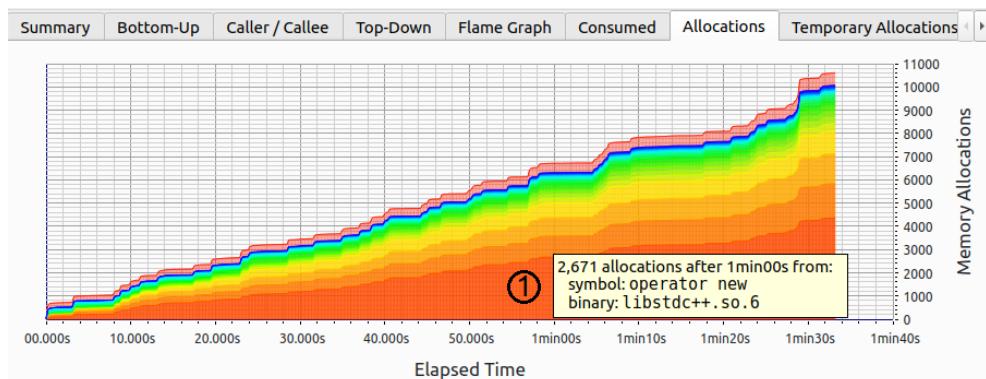


Figure 7.16: Stockfish memory profile with Heaptrack, the number of allocations is growing.

Since the number of allocations is growing but the total consumed memory doesn't change, we are dealing with temporary allocations. Let's find out where in the code they are coming from. It is easy to do with the help of a flame graph shown in Figure 7.17. There are 4800 temporary allocations in total with 90.8% of those coming from `operator new`. Thanks to the flame graph we know the entire call stack that leads to 4360 temporary allocations. Interestingly, those temporary allocations are initiated by `std::stable_sort` which allocates a temporary buffer to do the sorting. One way to get rid of those temporary allocations would be to use an in-place stable sorting algorithm. However, by doing so I observed an 8% drop in performance, so I discarded this change.

Similar to temporary allocations, you can also find the paths that lead to the largest allocations in a program. In the dropdown menu at the top of Figure 7.17, you would need to select the “Consumed” flame graph.

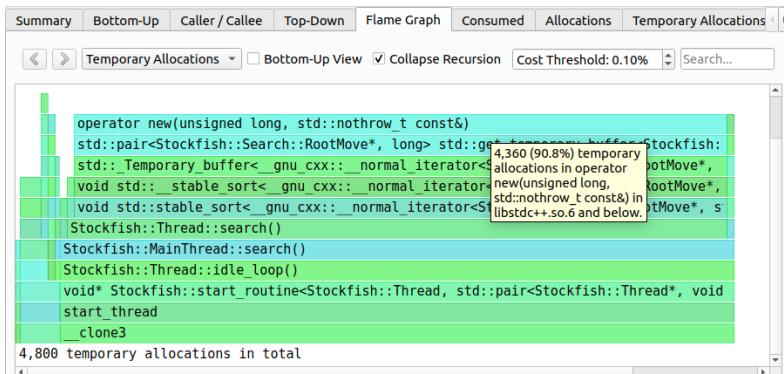


Figure 7.17: Stockfish memory profile with Heaptrack, temporary allocations flamegraph.

7.8.3 Memory Intensity and Footprint

In this section, I will show how to measure the memory *intensity* and memory *footprint* of a program. Memory intensity refers to the size of data being accessed by a program during an interval, measured, for example, in MB per second. A program with high memory intensity makes heavy use of the memory system, often accessing large amounts of data. On the other hand, a program with low memory intensity makes relatively fewer memory accesses and may be more compute-bound, meaning it spends more time performing calculations rather than waiting for data from memory. Measuring memory intensity during short time intervals enables us to observe how it changes over time.

Memory footprint measures the total number of bytes accessed by an application. While calculating memory footprint, we only consider unique memory locations. That is, if a memory location was accessed twice during the lifetime of a program, we count the touched memory only once.

Figure 7.18 shows the memory intensity and footprint of four workloads: Blender ray tracing, Stockfish chess engine, Clang++ compilation, and AI_bench PSPNet segmentation. We collect data for the chart with the Intel SDE (Software Development Emulator) tool using the method described on the Easyperf blog¹⁵⁴ with intervals of one billion instructions.

The solid line (Intensity) tracks the number of bytes accessed during each interval of 1B instructions. Here, we don't count how many times a certain memory location was accessed. If a memory location was loaded twice during an interval I , we count the touched memory only once. However, if this memory location is accessed the third time in the subsequent interval $I+1$, it contributes to the memory intensity of the interval $I+1$. Because of this, we cannot aggregate time intervals. For example, we can see that, on average, the Blender benchmark touches roughly 20MB every interval. We cannot aggregate its 150 consecutive intervals and say that the memory footprint of Blender was $150 * 20\text{MB} = 3\text{GB}$. It would be true only if a program never repeats memory accesses across intervals.

¹⁵⁴ Easyperf blog: Measuring memory footprint with SDE - <https://easyperf.net/blog/2024/02/12/Memory-Profiling-Part3>

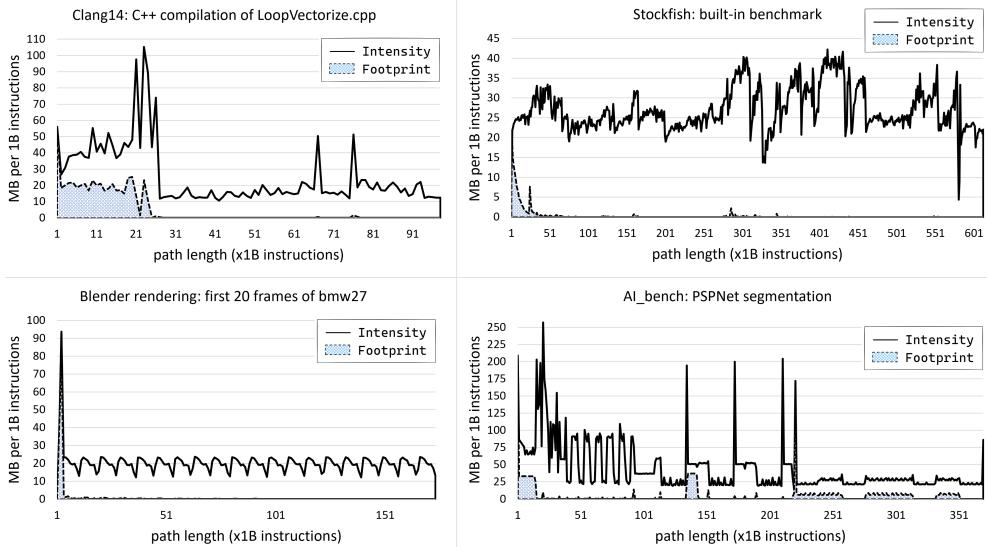


Figure 7.18: Memory intensity and footprint of four workloads. Intensity: total memory accessed during 1B instructions interval. Footprint: accessed memory that has not been seen before.

The dashed line (Footprint) tracks the size of the new data accessed every interval since the start of the program. Here, we count the number of bytes accessed during each 1B instruction interval that have never been touched before by the program. Aggregating all the intervals (cross-hatched area under that dashed line) gives us the total memory footprint for a program. Here are the memory footprint numbers for our benchmarks: Clang C++ compilation (487MB); Stockfish (188MB); PSPNet (1888MB); and Blender (149MB). Keep in mind, that these unique memory locations can be accessed many times, so the overall pressure on the memory subsystem may be high even if the footprint is relatively small.

As you can see our workloads have very different behavior. Clang compilation has high memory intensity at the beginning, sometimes spiking to 100MB per 1B instructions, but after that, it decreases to about 15MB per 1B instructions. Any of the spikes on the chart may be concerning to a Clang developer: are they expected? Could they be related to some memory-hungry optimization pass? Can the accessed memory locations be compacted?

The Blender benchmark is very stable; we can see the start and the end of each rendered frame. This enables us to focus on just a single frame, without looking at the entire workload of 1000+ frames. The Stockfish benchmark is a lot more chaotic, probably because the chess engine crunches different positions which require different amounts of resources. Finally, the PSPNet segmentation memory intensity is very interesting as we can spot repetitive patterns. After the initial startup, there are five or six sine waves from 40B to 95B, then three regions that end with a sharp spike to 200MB, and then again three mostly flat regions hovering around 25MB per 1B instructions. This is actionable information that can be used to optimize an application.

Such charts help us estimate memory intensity and measure the memory footprint of an application. By looking at the chart, you can observe phases and correlate them with the underlying algorithm. Ask yourself: “Does it look according to your expectations, or the workload is doing something sneaky?” You may encounter unexpected spikes in memory intensity. On many occasions, memory profiling helped identify a problem or served as an additional data point to support the conclusions that were made during regular profiling.

In some scenarios, memory footprint helps us estimate the pressure on the memory subsystem. For instance, if the memory footprint is small (several megabytes), we might suspect that the pressure on the memory subsystem is low since the data will likely reside in the L3 cache; remember that available memory bandwidth in modern processors ranges from tens to hundreds of GB/s and is getting close to 1 TB/s. On the other hand, when we’re dealing with an application that accesses 10 GB/s and the memory footprint is much bigger than the size of the L3 cache, then the workload might put significant pressure on the memory subsystem.

While memory profiling techniques discussed in this chapter certainly help better understand the behavior of a workload, this is not enough to fully assess the temporal and spatial locality of memory accesses. We still have no visibility into how many times a certain memory location was accessed, what is the time interval between two consecutive accesses to the same memory location, and whether the memory is accessed sequentially, with strides, or completely random. The topic of data locality of applications has been researched for a long time. Unfortunately, as of early 2024, there are no production-quality tools available that would give us such information. Further details are beyond the scope of this book, however, curious readers are welcome to read the related [article¹⁵⁵](#) on the Easyperf blog.

7.9 Continuous Profiling

In Chapter 5, we covered the various approaches available for conducting 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. However, 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 has emerged to turn these goals 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 facilitates analysis of why the execution of code was different at different times and also aids debugging of incidents even after they have happened. CP tools provide valuable insights into which code uses the most resources, and this helps engineers 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

¹⁵⁵ Easyperf blog: Data Locality and Reuse Distances - <https://easyperf.net/blog/2024/02/12/Memory-Profiling-Part5>

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.

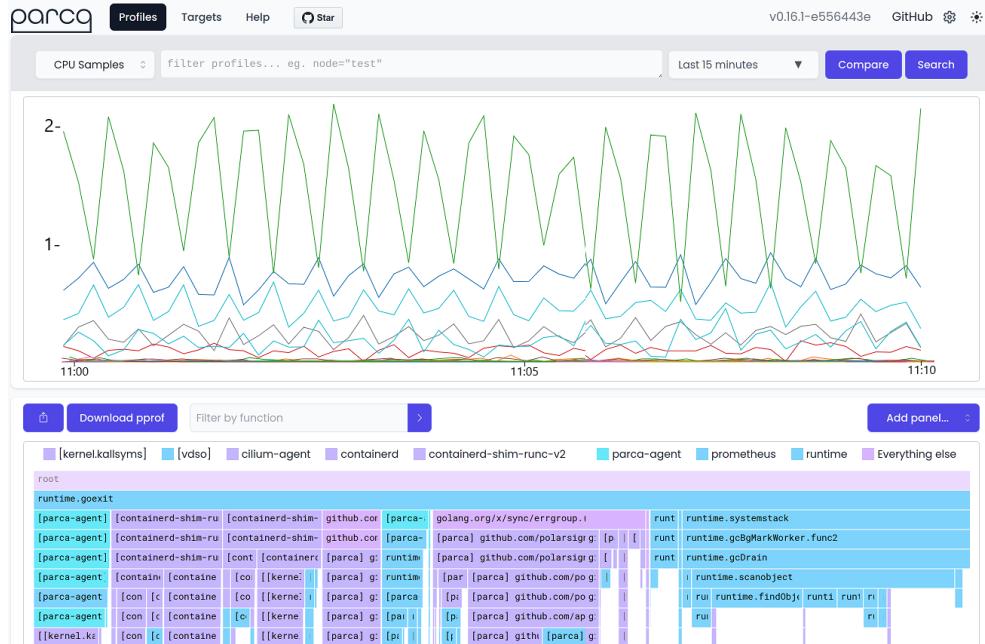


Figure 7.19: Screenshot of the Parca Continuous Profiler Web UI.

To showcase the look-and-feel of a typical CP tool, let's look at the Web UI of [Parca](#),¹⁵⁶ one of the open-source CP tools, depicted in Figure 7.19. The top panel displays a time series 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 in time when performance change occurred but also to determine the root cause of 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

¹⁵⁶ Parca - <https://github.com/parca-dev/parca>

version of a flame graph. Using it, you can compare call stacks before and after to help you find what 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.¹⁵⁷

Google introduced the CP 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 Optimyze and its Continuous Profiler (Prod-filer); Polar Signals released its Parca Continuous Profiler. It was open-source in April 2024.
7. In December 2021, Splunk released its AlwaysOn Profiler.
8. In March 2022, Intel acquired Granulate and its Continuous Profiler (gProfiler). It was made open-source in March 2024.

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 a 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 the CPU (e.g., memory, I/O, or locking). 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 Prod-filer, 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 an

¹⁵⁷ Differential flamegraph - <https://www.brendangregg.com/blog/2014-11-09/differential-flamegraphs.html>

application's critical path, e.g., a housekeeping 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 CP tools are basically profilers that run at all times, they inherit this same blind spot.

Thankfully, a new generation of CP tools has emerged that employ AI with Large Language Model-inspired architectures to process profile samples, analyze the relationships between functions, and finally pinpoint with high accuracy the functions and libraries that directly impact overall throughput and latency. One such company that offers this today is Raven.io. 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. Recall all the tools we used in this chapter. Which ones would you use in the following scenarios?
 - Scenario 1: the client support team reports a customer issue: after upgrading to a new version of the application, the performance of a certain operation drops by 10%.
 - Scenario 2: the 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 into your project; you've decided to compare their performance.
2. Run the application that you're working with daily. 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 do 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 the 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 tools that can visualize `perf`'s profiling data.
- We also discussed Windows Event Tracing (ETW), which is designed to observe software dynamics in a running system. Linux has a similar tool called

KUtrace,¹⁵⁸ which is covered in the book [Sites, 2022].

- There are hybrid profilers that combine techniques like code instrumentation, sampling, and tracing. This takes the best of these approaches and allows users to get very detailed information on a specific piece of code. In this chapter, we looked at Tracy, which is quite popular among game developers.
- Memory profilers provide information about memory usage, heap allocations, memory footprint, and other metrics. Memory profiling helps you understand how an application uses memory over time.
- Continuous profiling tools have already become an essential part of 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 in time when a performance change started and determine the root cause of an issue.

¹⁵⁸ KUtrace - <https://github.com/dicksites/KUtrace>

Part 2. Source Code Tuning

In Part 1 of this book, we discussed how to find performance bottlenecks in code. In Part 2 we will discuss how to fix such bottlenecks through the use of techniques for low-level source code optimization, also known as *tuning*.

A modern CPU is a very complicated device, and it's nearly impossible to predict how fast certain pieces of code will run. Software and hardware performance depends on many factors, and the number of moving parts is too big for a human mind to contain. 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.** Analyze algorithms and data structures used in a program, and see if you can find better ones. Example: use quicksort instead of bubble sort.
- **Parallelizing computations.** If an algorithm is highly parallelizable, make the program multithreaded, or consider running it on a GPU. The goal is to do multiple things at the same time. Concurrency is already used in all the layers of the hardware and software 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 concurrent network connections to overlap the request latency.
- **Eliminating redundant work.** 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, look-up tables, compression); pre-compute values known at compile-time; 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.** 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 a large block of memory rather than allocating space for hundreds of tiny objects.
- **Ordering.** Reorder the sequence of operations in an algorithm. Examples: change 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 discuss in this book fall under multiple categories. For example, we can say that vectorization is a combination of parallelizing and batching; and loop blocking (tiling) is a manifestation of batching and eliminating redundant work.

To complete the picture, let us also list other possibly 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), `-fipa` (enables inter-procedural optimizations). But don't stop here; there are many other options that affect performance. We will look at some of these in later chapters. You may consider mining the best set of options for an application; commercial products that automate this process are available.
- **Optimize third-party software packages:** the vast majority of software projects leverage layers of proprietary and open-source code. This includes OS, libraries, and frameworks. You can seek improvements by replacing, modifying, or reconfiguring one of those pieces.
- **Buy faster hardware:** obviously, this is a business decision that comes with an associated cost, but sometimes it's the only way to improve performance when other options have already been exhausted. It is much easier to justify a purchase when you identify performance bottlenecks in your application and communicate them clearly to the upper management. For example, once you find that memory bandwidth is limiting the 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, 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 those of a plain array. Some data structures, like binary trees, have a 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, e.g., `boost::flat_map` and `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. Let's assume 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, $O(\log N)$. 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 array, a linear scan is usually faster even though it has worse algorithmic complexity.¹⁵⁹

¹⁵⁹ In addition, linear scan does not require elements to be sorted.

Data-Driven Development

The main idea in Data-Driven Development (DDD), is to study how a program accesses data: how it is laid out in memory and how it is transformed throughout the program; then modify the program accordingly (change the data layout, change the access patterns). A classic example of such an approach is the Array-of-Structures (AOS) to Structure-of-Array (SOA) transformation, which is shown in Listing 7.2.

Listing 7.2 SOA to AOS transformation.

<pre>// Array of Structures (AOS) struct S { int a; int b; int c; // other fields };</pre>	<=>	<pre>// Structure of Arrays (SOA) struct S { int a[N]; int b[N]; int c[N]; // other arrays };</pre>
--	------------------	---

The answer to the question of which layout is better depends on how the code is accessing the data. If it iterates over the data structure `S` 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.

Another widespread example of DDD is small-size optimization. The 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 an internal buffer allocated in the stack memory and allocate memory on the heap only for longer strings. Other 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 the various optimizations one can imagine. The chapters in Part 2 primarily address optimizations specific to modern CPU architectures.

Before we jump into particular source code tuning techniques, 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 should you try applying low-level improvements.

Second, remember that an optimization you implement might not be beneficial on every platform. For example, Loop Blocking (tiling), which is discussed in Section 9.3.2,

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—the TMA:MemoryBound category.
- Chapter 9. Optimizing Computations—the TMA:CoreBound category.
- Chapter 10. Optimizing Branch Prediction—the TMA:BadSpeculation category.
- Chapter 11. Machine Code Layout Optimizations—the 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 12 covers optimization topics not specifically related to any of the categories covered in the previous four chapters. In this chapter, we will discuss CPU-specific optimizations, examine several microarchitecture-related performance problems, explore techniques used for optimizing low-latency applications, and give you advice on tuning your system for the best performance.

Chapter 13 addresses some common problems in optimizing multithreaded applications. It provides a case study of five real-world multithreaded applications, where we explain why their performance doesn't scale with the increasing number of CPU threads. We also discuss cache coherency issues, such as "False Sharing" and a few tools that are designed to analyze multithreaded applications.

8 Optimizing Memory Accesses

Modern computers are still being built based on the classical Von Neumann architecture which decouples CPU, memory, and input/output units. Nowadays, 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.

The statement that memory hierarchy performance is critical can be illustrated by Figure 8.1. 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. According to this picture, processor performance has plateaued, but even then, the gap in performance remains. [Hennessy & Patterson, 2017]

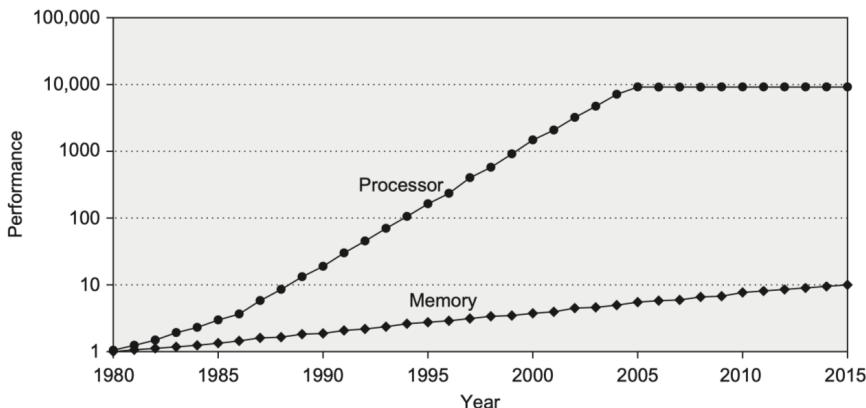


Figure 8.1: The gap in performance between memory and processors. © Source: [Hennessy & Patterson, 2017].

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 a 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 promptly.

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, the `Memory Bound` metric estimates a fraction of slots where

a 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 a troublesome memory access issue is identified, several optimization strategies might be applied. In this chapter, we will discuss techniques to improve memory access patterns.

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 pillars of cache-friendly code are the principles of temporal and spatial locality that we introduced in Section 3.6. The goal here is to have a predictable memory access pattern and store data efficiently.

The cache line is the smallest unit of data that can be transferred between the cache and the main memory. When designing cache-friendly code, it's helpful to think not only of individual variables and their locations in memory but also of cache lines.

Next, we will discuss several techniques to make data structures more cache-friendly.

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 enable the hardware prefetching mechanism (see Section 3.6.1.6) to recognize the memory access pattern and bring in the next chunk of data ahead of time. An example of Row-major versus Column-Major traversal is shown in Listing 8.1. Notice, that there is only one tiny change in the code (swapped `col` and `row` subscripts), but it has a large impact on performance.

The code on the left is not cache-friendly because it skips the `NCOLS` elements on every iteration of the inner loop. This results in a very inefficient use of caches: we aren't making full use of the entire prefetched cache line before it gets evicted. In contrast, the code on the right accesses elements of the matrix in the order in which they are laid out in memory. This guarantees that the cache line will be fully used before it gets evicted. Row-major traversal exploits spatial locality and is cache-friendly. Figure 8.2 illustrates the difference between the two traversal patterns.

Listing 8.1 Cache-friendly memory accesses.

<pre>// Column-major order for (row = 0; row < NROWS; row++) for (col = 0; col < NCOLS; col++) matrix[col][row] = row + col;</pre>	=>	<pre>// Row-major order for (row = 0; row < NROWS; row++) for (col = 0; col < NCOLS; col++) matrix[row][col] = row + col;</pre>
--	----	---



Figure 8.2: Column-major versus Row-major traversal.

The example presented above 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. If the data is not laid out in memory in a way that is optimal for the algorithm, it may require to rearrange the data first.

Consider a standard implementation of binary search in a large sorted array, where on each iteration, you access the middle element, compare it with the value you're searching for, and go either left or right. This algorithm 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 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. Keep in mind how the data will be accessed and manipulated. You should consider not only the time and space complexity of operations with a data structure but also the hardware effects associated with them.

By default, stay away from data structures that rely on pointers, e.g. linked lists or trees. When traversing elements, they require additional memory accesses to follow the pointers. If the maximum number of elements is relatively small and known at compile time, C++ `std::array` might be a better option than `std::vector`. If you need an associative container but don't need to store the elements in sorted order, `std::unordered_map` should be faster than `std::map`. A good step-by-step guide for choosing appropriate C++ containers can be found in [Fog, 2023b, Section 9.7 Data structures, and container classes].

Sometimes, it's more efficient to store pointers to contained objects, instead of objects themselves. Consider a situation when you need to store many objects in an array while the size of each object is big. In addition, the objects are frequently shuffled, removed, and inserted. Storing objects in an array will require moving large chunks of memory every time the order of objects is changed, which is expensive. In this case, it's better to store pointers to objects in the array. This way, only the pointers are moved, which is much cheaper. However, this approach has its drawbacks. It requires additional memory for the pointers and introduces an additional level of indirection.

8.1.3 Packing the Data

The utilization of data caches can be also improved by making 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 in Listing 8.2. If we know that `a`, `b`, and `c` represent enum values that take a certain number of bits to encode, we can reduce the storage of the struct `S`.

Notice the three times less space required to store an object of the packed version of

Listing 8.2 Data Packing

```
// S is 3 bytes           // S is 1 byte
struct S {               struct S {
    unsigned char a;     unsigned char a:4;
    unsigned char b;     =>   unsigned char b:2;
    unsigned char c;     unsigned char c:2;
};                      };
```

S. This greatly reduces the amount of memory transferred back and forth and saves cache space. However, using bitfields comes with additional costs.¹⁶⁰ Since the bits of **a**, **b**, and **c** are packed into a single byte, the compiler needs to perform additional bit manipulation operations to extract and insert them. For example, to load **b**, you need to shift the byte value right (`>>`) by 2 and do logical AND (`&`) with `0x3`. Similarly, shift left (`<<`) and logical OR (`|`) operations are needed to store the updated value back into the packed format. Data packing is beneficial in places where additional computation is cheaper than the delay caused by inefficient memory transfers.

Also, a programmer can reduce memory usage by rearranging fields in a struct or class when it avoids padding added by a compiler. Inserting unused bytes of memory (pads) enables efficient storing and fetching of individual members of a struct. In the example in Listing 8.3, the size of **S** can be reduced if its members are declared in the order of decreasing size. Figure 8.3 illustrates the effect of rearranging the fields in struct **S**.

Listing 8.3 Avoid compiler padding.

```
// S is `sizeof(int) * 3` bytes      // S is `sizeof(int) * 2` bytes
struct S {                         struct S {
    bool b;                         int i;
    int i;                           short s;
    short s;                         bool b;
};                                };
```



Figure 8.3: Avoid compiler padding by rearranging the fields. Blank cells represent compiler padding.

8.1.4 Field Reordering

Reordering fields in a data structure can also be beneficial for another reason. Consider an example in Listing 8.4. Suppose that the **Soldier** structure is used to track each one of the thousands of units on the battlefield in a game. The game has three phases: battle, movement, and trade. During the battle phase, the **attack**, **defense**, and **health** fields are used. During the movement phase, the **coords**, and **speed** fields are used. During the trade phase, only the **money** field is used.

¹⁶⁰ Also, you cannot take the address of a bitfield.

The problem with the organization of the `Soldier` struct in the code on the left is that the fields are not grouped according to the phases of the game. For example, during the battle phase, the program needs to access two different cache lines to fetch the required fields. The fields `attack` and `defense` are very likely to reside on the same cache line, but the `health` field is always pushed to the next cache line. The same applies to the movement phase (`speed` and `coords` fields).

We can make the `Soldier` struct more cache-friendly by reordering the fields as shown in Listing 8.4 on the right. With that change, the fields that are accessed together are grouped together.

Listing 8.4 Field Reordering.

<pre>struct Soldier { 2DCoords coords; /* 8 bytes */ unsigned attack; unsigned defense; /* other fields */ /* 64 bytes */ unsigned speed; unsigned money; unsigned health; };</pre>	<pre>struct Soldier { unsigned attack; // 1. battle unsigned defense; // 1. battle unsigned health; // 1. battle 2DCoords coords; // 2. move unsigned speed; // 2. move /* other fields */ unsigned money; // 3. trade };</pre>
---	---

Since Linux kernel 6.8, there is a new functionality in the `perf` tool that allows you to find data structure reordering opportunities. The `perf mem record` command can now be used to profile data structure access patterns. The `perf annotate --data-type` command will show you the data structure layout along with profiling samples attributed to each field of the data structure. Using this information you can identify fields that are accessed together.¹⁶¹

Data-type profiling is very effective at finding opportunities to improve cache utilization. Recent Linux kernel history contains many examples of commits that reorder structures,¹⁶² pad fields,¹⁶³ or pack¹⁶⁴ them to improve performance.

8.1.5 Other Data Structure Reorganization Techniques

To close the topic of cache-friendly data structures, we will briefly mention two other techniques that can be used to improve cache utilization: *structure splitting* and *pointer inlining*.

Structure splitting. Splitting a large structure into smaller ones can improve cache utilization. For example, if you have a structure that contains a large number of fields, but only a few of them are accessed together, you can split the structure into two or more smaller ones. This way, you can avoid loading unnecessary data into the cache. An example of structure splitting is shown in Listing 8.5. By splitting the `Point` structure into `PointCoords` and `PointInfo`, we can avoid loading the `PointInfo` data into caches when we only need `PointCoords`. This way, we can fit more points on a single cache line.

¹⁶¹ Linux `perf` data-type profiling - <https://lwn.net/Articles/955709/>

¹⁶² Linux commit 54ff8ad69c6e93c0767451ae170b41c000e565dd

¹⁶³ Linux commit aee79d4e5271cee4ffa89ed830189929a6272eb8

¹⁶⁴ Linux commit e5598d6ae62626d261b046a2f19347c38681ff51

Listing 8.5 Structure Splitting.

```

struct Point {
    int X;
    int Y;
    int Z;
    /*many other fields*/
};  

std::vector<Point> points;  

struct PointCoords {
    int X;
    int Y;
    int Z;
};  

struct PointInfo {
    /*many other fields*/
};  

std::vector<PointCoords> pointCoords;  

std::vector<PointInfo> pointInfos;

```

Pointer inlining. Inlining a pointer into a structure can improve cache utilization. For example, if you have a structure that contains a pointer to another structure, you can inline the pointer into the first structure. This way, you can avoid additional memory access to fetch the second structure. An example of pointer inlining is shown in Listing 8.6. The `weight` parameter is used in many graph algorithms, and thus, it is frequently accessed. However, in the original version on the left, retrieving the edge weight requires additional memory access, which can result in a cache miss. By moving the `weight` parameter into the `GraphEdge` structure, we avoid such issues.

Listing 8.6 Moving the `weight` parameter into the parent structure.

```

struct GraphEdge {
    unsigned int from;
    unsigned int to;
    GraphEdgeProperties* prop;
};  

struct GraphEdgeProperties {
    float weight;
    std::string label;
    // ...
};  

struct GraphEdge {
    unsigned int from;
    unsigned int to;
    float weight;
    GraphEdgeProperties* prop;
};  

struct GraphEdgeProperties {
    std::string label;
    // ...
};  


```

8.2 Dynamic Memory Allocation

Developers should realize that dynamic memory allocation has many associated costs. Allocating an object on the stack is done instantly, regardless of the size of the object: you just need to move the stack pointer. However, dynamic memory allocation is a more complex operation. It involves calling a standard library function like `malloc`, which potentially may delegate the allocation to the operating system. Avoiding unnecessary dynamic memory allocation is the first step to avoiding these costs. The easy target in this process is temporary allocations, i.e., allocations that are directly followed by their deallocation. In Section 7.8.2 we showed how you can use `heaptrack` to find sources of dynamic memory allocations.

You can amortize the cost of many small allocations by allocating one large block. This is the core idea behind [arena allocators](#)¹⁶⁵ and memory pools. It gives more flexibility for manual memory management. You can take a memory region allocated

¹⁶⁵ Region-based memory management - https://en.wikipedia.org/wiki/Region-based_memory_management

by the OS and design your own allocation strategy on top of that region. One simple strategy could be to divide that region into two parts: one for hot data and one for cold data. And provide two allocation methods that will tap into their own arenas. Keeping hot data together creates opportunities for better cache utilization. It is also likely to improve TLB utilization since hot data will be more compact and will occupy fewer memory pages.

Another cost of dynamic memory allocation appears when an application is using multiple threads. When two threads are trying to allocate memory at the same time, the OS has to synchronize them. In a highly concurrent application, threads may spend a significant amount of time waiting for a common lock to allocate memory. The same applies to memory deallocation. Again, custom allocators can help to avoid this problem, for example, by employing a separate arena for each thread.

There are many drop-in replacements for the standard dynamic memory allocation routines (`malloc` and `free`) that are faster, more scalable, and address fragmentation problems better. Some of the most popular memory allocation libraries are `jemalloc`¹⁶⁶ and `tcmalloc`¹⁶⁷. Some projects adopted `jemalloc` and `tcmalloc` as their default memory allocator, and they have seen significant performance improvements.

Finally, some costs of dynamic memory allocation are hidden¹⁶⁸ and cannot be easily measured. In all major operating systems, the pointer returned by `malloc` is just a promise – the OS commits that when pages are touched it will provide the required memory, but the actual physical pages are not allocated until the virtual addresses are accessed. This is called *demand paging*, which incurs a cost of a minor page fault for every newly allocated page. We discuss how to mitigate this cost in Section 12.3.1. Also, for security reasons, all modern operating systems erase the contents (write zeros) of a page before giving it to the next process. The OS maintains a pool of zeroed pages to have them ready for allocation. But when this pool runs out of available zeroed pages, the OS has to zero a page on demand. This process isn't super expensive, but it isn't free either and may increase the latency of a memory allocation call.

8.3 Workaround Memory Bandwidth Limitations

As discussed in Section 3.6.2, a processor gets the data from memory through a memory bus. With the latest DDR5 memory technology, the maximum theoretical memory bandwidth is 51.2 GB/s per channel. Modern systems have multiple memory channels; for example, a typical laptop usually has two memory channels, while server systems can have from 4 to 12 channels. It may seem that even a laptop can move a lot of data back and forth each second, but in reality, memory bandwidth becomes a limitation in many applications.

We should keep in mind that memory channels are shared between all the cores in a system. Once many cores engage in a memory-intensive activity, the traffic flowing through the memory bus can become congested. This may lead to increased wait times for memory requests to return. Modern systems are designed to accommodate multiple

¹⁶⁶ jemalloc - <http://jemalloc.net/>.

¹⁶⁷ tcmalloc - <https://github.com/google/tcmalloc>

¹⁶⁸ Bruce Dawson: Hidden Costs of Memory Allocation - <https://randomascii.wordpress.com/2014/12/10/hidden-costs-of-memory-allocation/>.

memory-demanding threads working at the same time, so usually you need multiple threads to saturate the memory bandwidth. Emerging AI workloads are known to be extremely “memory hungry” and highly parallelized, so memory bandwidth is the number one bottleneck for them.

The first step in addressing memory bandwidth limitations is to determine the maximum theoretical and expected memory bandwidth. The theoretical maximum memory bandwidth can be calculated from the memory technology specifications as we have shown in Section 5.5. The expected memory bandwidth can be measured using tools like Intel Memory Latency Checker or `lmbench`, which we discussed in Section 4.10. Intel VTune can automatically measure memory bandwidth before the analyzed application starts.

The second step is to measure the memory bandwidth utilization while your application is running. If the amount of memory traffic is close to the maximum measured bandwidth, then the performance of your application is likely to be bound by memory bandwidth. It is a good idea to plot the memory bandwidth utilization over time to see if there are different phases where memory intensity spikes or takes a dip. Intel VTune can provide such a chart if you tick the “Evaluate max DRAM bandwidth” checkbox in the analysis configuration.

If you have determined that your application is memory bandwidth bound, the first suggestion is to see if you can decrease the memory intensity of your application. It is not always possible, but you can consider disabling some memory-hungry features of your application, recomputing data on the fly instead of caching results, or compressing your data. In the AI space, most Large Language Models (LLMs) are supplied in fp32 precision, which means that each parameter takes 4 bytes. The biggest performance gain can be achieved with quantization techniques, which reduce the precision of the parameters to fp16 or int8. This will reduce the memory traffic by 2x or 4x, respectively. Sometimes, 4-bit and even 5-bit quantization is used, to reduce memory traffic and strike the right balance between inference performance and quality.

It is important to mention that for workloads that saturate available memory bandwidth, code optimizations don’t play a big role as they do for compute-bound workloads. For compute-bound applications, code optimizations like vectorization usually translate into large performance gains. However, for memory-bound workloads, vectorization may not have a similar effect since a processor can’t make forward progress, simply because it lacks data to work with. We cannot make the memory bus run faster, this is why memory bandwidth is often a hard limitation to overcome.

Finally, if all the options have been exhausted, and the memory bandwidth is still a limitation, the only way to improve the situation is to buy better hardware. You can invest money in a server with more memory channels, or DRAM modules with faster transfer speed. This could be an expensive but still, a viable option to speed up your application.

8.4 Reducing DTLB Misses

As discussed in Section 3.7.1, 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, and traversing a graph. The usage of huge pages has the potential to speed up such applications.

On x86 platforms, the default page size is 4KB. Consider an application that frequently references memory space of 20 MBs. With 4KB pages, the OS needs to allocate many small pages. Also, the process will be touching many 4KB-sized pages, each of which will contend for a limited number of TLB entries. In contrast, using huge 2MB pages, 20MB of memory can be mapped with just ten pages, whereas with 4KB pages, you would need 5120 pages. This means fewer TLB entries are needed when using huge pages, which in turn reduces 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 for data, but 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 from 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's JVM, and others. Usually, those software packages provide an option to 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 B](#). On Linux, there are two ways of using huge pages in an application: Explicit and Transparent Huge Pages. Windows support is not as rich as Linux and will be discussed later.

8.4.1 Explicit Huge Pages

Explicit Huge Pages (EHP) are available as part of the system memory, and are exposed as a huge page file system `hugetlbfs`. EHPs should be reserved either at

system boot time or before an application starts. See Appendix B for instructions on how to do that. Reserving EHPs at boot time increases the possibility of successful allocation because the memory has not yet been significantly fragmented. Explicitly preallocated pages reside in a reserved chunk of physical 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 require.

The simplest method of using EHP in a Linux application is to call `mmap` with `MAP_HUGETLB` as shown in Listing 8.7. In this code, the pointer `ptr` will point to a 2MB region of memory that was explicitly reserved for EHPs. Notice, that allocation may fail if the EHPs were not reserved in advance. Other less popular ways to use EHPs in user code are provided in Appendix B. Also, developers can write their own arena-based allocators that tap into EHPs.

Listing 8.7 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 `libhugetlbf`¹⁶⁹ library, which overrode `malloc` calls used in existing dynamically linked executables, to allocate memory in 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=libhugetlbf.so HUGETLB_MORECORE=yes <your app command line>` to make use of it. But luckily, other libraries enable the use of huge pages (not EHPs) with `malloc`, which we will discuss next.

8.4.2 Transparent Huge Pages

Linux also offers Transparent Huge Page 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 is enabled in the system with:

```
$ cat /sys/kernel/mm/transparent_hugepage/enabled
always [madvise] never
```

The value shown in brackets is the current setting. If this value is `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 documentation¹⁷⁰ regarding THP.

¹⁶⁹ libhugetlbf - <https://github.com/libhugetlbf/libhugetlbf>.

¹⁷⁰ Linux kernel THP documentation - <https://www.kernel.org/doc/Documentation/vm/transhuge.txt>

When THP is enabled system-wide, huge pages are used automatically for normal memory allocations, without an explicit request from applications. To observe the effect of huge pages on their application, a user just needs to enable system-wide THPs with `echo "always" | sudo tee /sys/kernel/mm/transparent_hugepage/enabled`. It will automatically launch a daemon process named `khugepaged` which starts scanning the application's memory space to promote regular pages to huge pages. Sometimes the kernel may fail to combine multiple regular pages into a huge page in case it cannot find a contiguous 2MB chunk of memory.

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 huge pages are enabled system-wide, applications may end up allocating more memory resources than needed. This is why the system-wide mode is disabled by default. Don't forget to disable system-wide THPs after you've finished your experiments as it may hurt overall system performance.

With the `madvise` (per-process) option, THP is enabled only inside memory regions attributed via the `madvise` system call with the `MADV_HUGEPAGE` flag. As shown in Listing 8.8, the pointer `ptr` will point to a 2MB region of the anonymous (transparent) memory region, which the kernel allocates dynamically. The `mmap` call will fail if the kernel cannot find a contiguous 2MB chunk of memory.

Listing 8.8 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 8.8. 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 using `jemalloc`, which is one of the most popular options. If you have access to the source code of the application, you can relink the binary with an 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 the `VirtualAlloc` system call. See details in Appendix B.

8.4.3 Explicit vs. Transparent Huge Pages

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 software 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, even when they are not used.

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 software 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 several kernel processes responsible for making space in virtual memory, which may include swapping memory to 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 disk, so they incur much less latency overhead.

All in all, THPs are easier to use, but incur bigger allocation latency overhead. That is 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.

8.5 Explicit Memory Prefetching

By now, you should know that memory accesses not resolved from caches are often very expensive. Modern CPUs try very hard to lower the penalty of cache misses by predicting which memory locations a program will access in the future and prefetch them ahead of time. If the requested memory location is not in the cache at the time a program demands it, then we will suffer the cache miss penalty as we have to go to the DRAM and fetch the data anyway. But if the CPU manages to bring that memory location into caches in time or if the request was predicted and data is underway, then the penalty of a cache miss will be much lower.

Modern CPUs have two automatic mechanisms for solving that problem: hardware prefetching and OOO execution. Hardware prefetchers help to hide the memory access latency by initiating prefetching requests on repetitive memory access patterns. An

OOO engine looks N instructions into the future and issues loads early to enable the smooth execution of future instructions that will demand this data.

Hardware prefetchers fail when data access patterns are too complicated to predict. And there is nothing software developers can do about it as we cannot control the behavior of this unit. On the other hand, an OOO engine does not try to predict memory locations that will be needed in the future as hardware 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 8.9, 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. A hardware prefetcher can't predict it since 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 8.4. The program waits for the value to come back (the hatched fill rectangle in the diagram) without making forward progress.

You're probably thinking: "But the next iteration of the loop should start executing speculatively in parallel". That's true, and indeed, it is reflected in Figure 8.4. The `doSomeExtensiveComputation` function requires a lot of work, and when execution gets closer to the finish of the first iteration, a CPU speculatively starts executing instructions from the next iteration. It creates a positive overlap in the execution between iterations. In fact, we presented an optimistic scenario where a processor was able to generate the next random number and issue a load in parallel with the previous iteration of the loop. However, a CPU wasn't able to fully hide the latency of the load, because it cannot look that far ahead of the current iteration to issue the load early enough. Maybe future processors will have more powerful OOO engines, but for now, there are cases where a programmer's intervention is needed.

Listing 8.9 A random number is an index for a subsequent load.

```
for (int i = 0; i < N; ++i) {
    size_t idx = random_distribution(generator);
    int x = arr[idx]; // cache miss
    doSomeExtensiveComputation(x);
}
```

Luckily, it's not a dead end as there is a way to speed up this code by fully overlapping the load with the execution of `doSomeExtensiveComputation`, which will hide the latency of a cache miss. We can achieve this with techniques called *software pipelining* and *explicit memory prefetching*. Implementation of this idea is shown in Listing 8.10. We pipeline generation of random numbers and start prefetching memory location for the next iteration in parallel with `doSomeExtensiveComputation`.

A graphical illustration of this transformation is shown in Figure 8.5. We utilized software pipelining to generate random numbers for the next iteration. In other words,

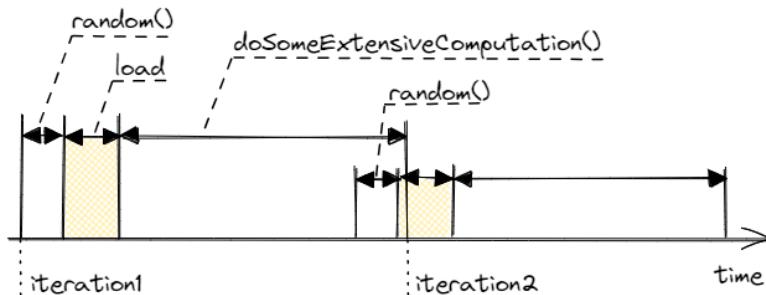


Figure 8.4: Execution timeline that shows the load latency standing on a critical path.

Listing 8.10 Utilizing Explicit 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);
}
```

on iteration M, we produce a random number that will be consumed on iteration M+1. This enables us to issue the memory request early since we already know the next index in the array. This transformation makes our prefetching window much larger and fully hides the latency of a cache miss. On the iteration M+1, the actual load has a very high chance to hit caches, because it was prefetched on iteration M.

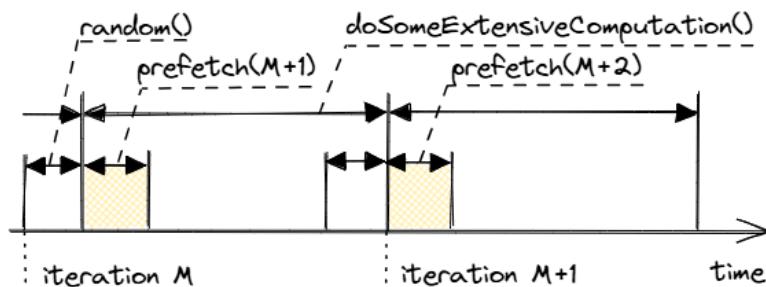


Figure 8.5: Hiding the cache miss latency by overlapping it with other execution.

Notice the usage of `__builtin_prefetch`,¹⁷¹ a special hint that developers can use to explicitly request a CPU to prefetch a certain memory location. Another option is to use compiler intrinsics. On x86 platforms there is `_mm_prefetch` intrinsic, on ARM platforms there is `__pld` intrinsic. Compilers will generate the PREFETCH instruction for x86 and the pld instruction for ARM.

¹⁷¹ GCC builtins - <https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html>.

There are situations when software memory prefetching is not possible. For example, when traversing a linked list, the prefetching window is tiny and it is not possible to hide the latency of pointer chasing.

In Listing 8.10 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 8.11 is one of those cases when it could be beneficial. It presents a typical code for populating a graph with edges. If the graph is very sparse and has a lot of vertices, it is very likely that accesses to `this->out_neighbors` and `this->in_neighbors` vectors will frequently miss in caches. This happens because every edge is likely to connect new vertices that are not currently in caches.

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 hardware 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 a prefetch hint to be effective, it must be inserted well ahead of time so that by the time the loaded value is 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 data that is not used for a long time. Notice, in Listing 8.11, `lookAhead` is a template parameter, which enables a programmer 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; an example of using this method can be found on the Easyperf blog.¹⁷²

Listing 8.11 Example of a software 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` ...
}
```

Explicit memory prefetching is most frequently used in loops, but also you can insert those hints into a parent function; again, it all depends on the available prefetching window.

¹⁷² “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 implementation-specific and platforms are not required to honor those hints. In such a case it will likely degrade performance. My recommendation 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 outweigh the potential maintenance burden.

For some complicated scenarios, make sure that the code 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 a `continue` statement or changing the next element to be processed is 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 Frontend. 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.

Questions and Exercises

1. Solve the `perf-ninja::data_packing` lab assignment, in which you need to make a data structure more compact.
2. Solve the `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.
3. Solve the `perf-ninja::swmem_prefetch_1` lab assignment by implementing explicit memory prefetching for future loop iterations.
4. Describe in general terms what it takes for a piece of code to be cache-friendly.
5. Run the application that you're working with daily. Measure its memory utilization and analyze heap allocations using memory profilers that we discussed in Section 7.8. Identify hot memory accesses using Linux perf, Intel VTune, or other profiler. Is there a way to improve those accesses?

Chapter Summary

- Most real-world applications experience memory-related performance bottlenecks. Emerging application domains, such as machine learning and big data, are particularly demanding in terms of memory bandwidth and latency.
- The performance of the memory subsystem is not growing as fast as the 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 frequently used recipes for developing cache-friendly data structures, explored data reorganization techniques, learned how to utilize huge memory pages to improve DTLB performance, and how to use explicit memory prefetching to reduce the number of cache misses.

9 Optimizing Computations

In the previous chapter, we discussed how to clear the path for efficient memory access. 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 applications involving complex graphics, artificial intelligence, cryptocurrency mining, and big data processing. In this chapter, we will focus on optimizing computations that can reduce the amount of work 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 the **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 instructions are limiting the performance. For example, a long sequence of dependent operations may lead to low Instruction Level Parallelism (ILP) and wasting many execution slots. The next section discusses data dependency chains in more detail.
- A shortage in hardware computing resources. This indicates that certain execution units are overloaded (also known as *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. However, there is a 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.

In this chapter, we will take a look at well-known techniques like function inlining, vectorization, and loop optimizations. Those code transformations aim to reduce the total amount of executed instructions or replace them with more efficient ones.

9.1 Data Dependencies

When a program statement refers to the output of a preceding statement, we say that there is a *data dependency* between the two statements. Sometimes people also use the terms *dependency chain* or *data flow dependencies*. The example we are most familiar with is traversing a linked list (see Figure 9.1). To access node $N+1$, we should first dereference the pointer $N \rightarrow \text{next}$. For the loop on the right, this is a *recurrent* data dependency, meaning it spans multiple iterations of the loop. Traversing a linked list is one very long dependency chain.

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.



Figure 9.1: Data dependency while traversing a linked list.

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 chasing 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, such as a loop or function.

You may wonder: “If you cannot get rid of dependency chains, what *can* you do?” Well, sometimes this will be a limiting factor for performance, and unfortunately, you will have to live with it. But there are cases where you can break unnecessary data dependency chains or overlap their execution. One such example is shown in Listing 9.1. Similar to a few other cases, we present the source code on the left along with the corresponding ARM assembly on the right. Also, this code example is included in the `dep_chains_2` lab assignment of the Performance Ninja online course, so you can try it yourself.¹⁷³

This small program simulates 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 a medium-quality `XorShift` RNG as it only has 3 shifts and 3 XORs inside. What can be simpler? Also, you searched the web and found algorithms for sine and cosine approximation using polynomials, which are accurate enough and very fast.

I compiled the code using the Clang-17 C++ compiler and ran it on a Mac mini

¹⁷³ Performance Ninja: Dependency Chains 2 - https://github.com/dendibakh/perf-ninja/tree/main/labs/core_bound/dep_chains_2

Listing 9.1 Random Particle Motion on a 2D Surface

```

1 struct Particle {
2     float x; float y; float velocity;
3 };
4
5 class XorShift32 {
6     uint32_t val;
7 public:
8     XorShift32 (uint32_t seed) : val(seed) {}
9     uint32_t gen() {
10         val ^= (val << 13);
11         val ^= (val >> 17);
12         val ^= (val << 5);
13         return val;
14     }
15 };
16
17 static float sine(float x) {
18     const float B = 4 / PI_F;
19     const float C = -4 / (PI_F * PI_F);
20     return B * x + C * x * std::abs(x);
21 }
22 static float cosine(float x) {
23     return sine(x + (PI_F / 2));
24 }
25
26 /* Map degrees [0;UINT32_MAX) to radians [0;2*pi)*/
27 float DEGREE_TO_RADIAN = (2 * PI_D) / UINT32_MAX;
28
29 void particleMotion(vector<Particle> &particles,
30                      uint32_t seed) {
31     XorShift32 rng(seed);
32     for (int i = 0; i < STEPS; i++)
33         for (auto &p : particles) {
34             uint32_t angle = rng.gen();
35             float angle_rad = angle * DEGREE_TO_RADIAN;
36             p.x += cosine(angle_rad) * p.velocity;
37             p.y += sine(angle_rad) * p.velocity;
38         }
39 }

```

<pre> .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 </pre>
--

(Apple M1, 2020). Let us examine the generated ARM assembly code:

- The first three `eor` (exclusive OR) instructions combined with `lsl` (shift left) or `lsr` (shift right) correspond to the `XorShift32::gen` function.
- Next `ucvtf` (convert unsigned integer to floating-point) and `fmul` (floating-point multiply) are used to convert the angle from degrees to radians (line 35 in the code).
- Sine and Cosine functions both have two `fmul` instructions and one `fmadd` (floating-point fused multiply-add) instruction. Cosine also has an additional `fadd` (floating-point add) instruction.
- Finally, we have one more pair of `fmadd` instructions to calculate x and y respectively, and an `stp` instruction to store the pair of coordinates.

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 on `XorShift32::val`. To generate the next random number, the generator has to produce the previous number first. The next call of method `XorShift32::gen` will generate the number based on the previous one. Figure 9.2 visualizes the problematic loop-carry dependency. Notice, that the code for calculating particle coordinates (convert the angle to radians, sine, cosine, multiple results by velocity) starts executing as soon as the corresponding random number is ready, but not sooner.

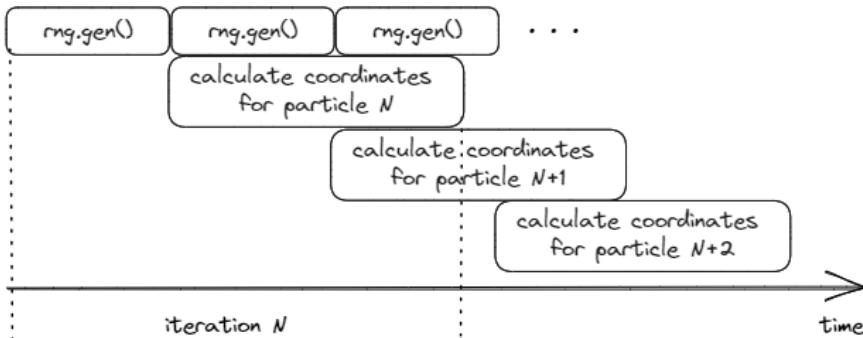


Figure 9.2: Visualization of dependent execution in Listing 9.1

The code that calculates the coordinates of particle N is not dependent on particle $N-1$, so it could be beneficial to pull them left to overlap their execution even more. You probably want to ask: “But how can those three (or six) instructions drag down the performance of the whole loop?”. 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 iterations 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.¹⁷⁴ Each `eor` and `lsl` instruction incurs 2 cycles of latency: one cycle for the shift and one for the 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 of latency to finish all the `fmul` and `fmadd` instructions. But it doesn't matter, because they are not on the critical path. The thing that matters is the throughput of these instructions. A useful rule of thumb: if an instruction is on a critical path, look at its latency, otherwise 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 the dependency chain), and the second is imposed by the hardware (2.25 cycles per iteration due to the throughput of the

¹⁷⁴ Apple published instruction latency and throughput data in [Apple, 2024, Appendix A].

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 an additional RNG object so that one of them feeds even iterations and another feeds odd iterations of the loop as shown in Listing 9.2. Notice, that we also manually unrolled the loop. Now we have two separate dependency chains, which can be executed in parallel. You 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. An alternative solution would be to pick a different RNG that has a less expensive internal dependency chain.

Listing 9.2 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)
    }
}
```

Once you do this transformation, the 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, I disabled compiler vectorization.

To measure the performance impact of the change, I ran “before” and “after” versions and observed the running time go 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 my due diligence, I also measured other metrics to make sure performance didn’t accidentally improve for other reasons. In the original code, the MPKI is 0.01, and **BranchMispredRate** is 0.2%, which means 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 Alder Lake 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 that instead of a simple RNG, you have a very complicated cryptographic algorithm that is 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 a nice 2x speedup, but see only 5% 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, that the Reservation Station (RS) capacity is not enough to see 10,000 instructions ahead as its number of entries is much smaller. So, the CPU will not be able to overlap the execution of two dependency chains. To fix this, 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 `XorShift32::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 you may hit is register pressure. Running multiple dependency chains in parallel requires keeping more state and thus more registers. If you run out of architectural registers, the compiler will start spilling them to the stack, which will slow down the program.

It is worth mentioning that data dependencies can also be created through memory. For example, if you write to memory location `M` on loop iteration `N` and read from this location on iteration `N+1`, there will effectively be a dependency chain. The stored value may be forwarded to a load, but these instructions cannot be reordered and executed in parallel.

As a closing thought, I 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 other block 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.

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, the 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 is called. Function inlining is a process of replacing a call to function `foo` with the code for `foo` 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 because 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 code size and compile time.¹⁷⁵

The primary mechanism for function inlining in many compilers relies on a cost model. For example, in the LLVM compiler, function inlining is based on computing a cost for each function *call site*. A call site is a place in the code where a function is called. 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 a fixed number; however, it can be varied under certain circumstances.¹⁷⁶ In addition to the generic cost model, many heuristics can overwrite cost model decisions in some cases. For instance:

- Tiny functions (wrappers) are almost always inlined.
- Functions with a single call site 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 unless it's a tail-recursive function (see next section). Also, if the depth of recursion is usually small, it's possible to partially inline a recursive function, i.e., inline a body of a recursive function to itself a couple of times, and then leave a recursive call as before. This may eliminate the overhead of a function call if the recursive call depth is usually small.
- A function that is referred to through a pointer can be inlined in place of a direct call but the function 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 I wrote earlier, compilers tend to use a cost model approach when deciding 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 adjusting 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. Listing 9.3 is an example of a function profile with a prologue and epilogue consuming ~50% of the function time:

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

¹⁷⁵ See the article: <https://aras-p.info/blog/2017/10/09/Forced-Inlining-Might-Be-Slow/>.

¹⁷⁶ 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`).

Listing 9.3 A profile of function `foo` which has a hot prologue and epilogue

Overhead (%)	Source code & Disassembly
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

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 a compiler to inline a function.

For the GCC and Clang compilers, you can make a hint for inlining `foo` with the help of a C++11 `[[gnu::always_inline]]` attribute as shown in the code example below. With earlier C++ standards you can use `__attribute__((always_inline))`. For the MSVC compiler, you can use the `__forceinline` keyword.

```
[[gnu::always_inline]] int foo() {
    // foo body
}
```

9.2.1 Tail Call Optimization

In a tail-recursive function, the recursive call is the last operation performed by the function before it returns its result. A simple example is demonstrated Listing 9.4. In the original code, the `sum` function recursively accumulates integer numbers from 0 to `n`, for example, a call of `sum(5,0)` will yield $5+4+3+2+1$, which gives 15.

If we compile the original code without optimizations (`-O0`), compilers will generate assembly code that has a recursive call. This is very inefficient due to the overhead of the function call. Moreover, if you call `sum` with a large `n`, the program will create a large number of stack frames on top of each other. There is a high chance that it will result in a stack overflow since the stack memory is limited.

When you apply optimizations, e.g., `-O2`, to the example in Listing 9.4, compilers

will recognize an opportunity for tail call optimization. The transformation will reuse the current stack frame instead of recursively creating new frames. To do so, the compiler flushes the current frame and replaces the `call` instruction with a `jmp` to the beginning of the function. Just like inlining, tail call optimization provides room for further optimizations. So, later, the compiler can apply more transformations to replace the original version with an iterative version shown on the right. For example, GCC 13.2 generates identical machine code for both versions.

Listing 9.4 Tail Call Compiler Optimization

```
// original code
int sum(int n, int acc) {
    if (n == 0) {
        return acc;
    } else {
        return sum(n - 1, acc + n);
    }
}

// compiler intermediate transformation
int sum(int n, int acc) {
    for (int i = n; i > 0; --i) {
        acc += i;
    }
    return acc;
}
```

Like with any compiler optimization, there are cases when it cannot perform the code transformation you want. If you are using the Clang compiler, and you want guaranteed tail call optimizations, you can mark a `return` statement with `__attribute__((musttail))`. This indicates that the compiler must generate a tail call for the program to be correct, even when optimizations are disabled. One example, where it is beneficial is language interpreter loops.¹⁷⁷ In case of doubt, it is better to use an iterative version instead of tail recursion and leave tail recursion to functional programming languages.

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 account for the majority of the execution time. Small changes in such a critical piece of code may have a large 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.

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, in Section 9.3.1, that only move code around in a single loop. Next, in Section 9.3.2, we will take a look at high-level optimizations that restructure loops, which often affect multiple loops. Note, that what I present here 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] and [Allen & Kennedy, 2001].

¹⁷⁷ Josh Haberman's blog: motivation for guaranteed tail calls - <https://blog.reverberate.org/2021/04/21/musttail-efficient-interpreters.html>.

9.3.1 Low-level Optimizations.

Let's start with simple loop optimizations that transform the code inside a single loop: Loop Invariant Code Motion, Loop Unrolling, Loop Strength Reduction, and Loop Unswitching. 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): an expression whose value never changes across iterations of a loop is called a *loop invariant*. Since its value doesn't change across loop iterations, we can move a loop invariant expression outside of the loop. We do so by storing the result in a temporary variable and using it inside the loop (see Listing 9.5).¹⁷⁸ All decent compilers nowadays successfully perform LICM in the majority of cases.

Listing 9.5 Loop Invariant Code Motion

<pre>for (int i = 0; i < N; ++i) for (int j = 0; j < N; ++j) a[j] = b[j] * c[i];</pre>	<pre>=></pre>	<pre>for (int i = 0; i < N; ++i) { auto temp = c[i]; for (int j = 0; j < N; ++j) a[j] = b[j] * temp; }</pre>
--	------------------	--

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. Instead of modifying an induction variable on each iteration, we can unroll a loop and perform multiple iterations for each increment of the induction variable (see Listing 9.6).

Listing 9.6 Loop Unrolling

<pre>for (int i = 0; i < N; ++i) a[i] = b[i] * c[i];</pre>	<pre>=></pre>	<pre>int i = 0; for (; i+1 < N; i+=2) { a[i] = b[i] * c[i]; a[i+1] = b[i+1] * c[i+1]; } // remainder (when N is odd) for (; i < N; ++i) a[i] = b[i] * c[i];</pre>
---	------------------	---

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 and tested, then we go back to the top of the loop if it has more iterations to process. This work is commonly referred to as "loop overhead" or "loop tax", and it can be reduced with loop unrolling. For example, by unrolling the loop in Listing 9.6 by a factor of 2, we reduce the number of executed compare and branch instructions by 2x.

I do not recommend unrolling loops manually except in cases when you need to break loop-carry dependencies as shown in Listing 9.1. First, because compilers are very good at doing this automatically and usually can choose the optimal unrolling factor. 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

¹⁷⁸ The compiler will perform the transformation only if it can prove that a and c don't alias.

for long-latency instructions from the first iteration to finish (e.g. loads, divisions, microcoded instructions, long dependency chains), it will speculatively start executing instructions from the second iteration and only wait on loop-carried dependencies. This spans multiple iterations ahead, effectively unrolling the loop in the instruction Reorder Buffer (ROB). The third reason is that unrolling too much could lead to negative consequences due to code bloat.

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 9.7, 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.

Listing 9.7 Loop Strength Reduction

```
for (int i = 0; i < N; ++i)
    a[i] = b[i * 10] * c[i];      =>      int j = 0;
                                                for (int i = 0; i < N; ++i) {
                                                    a[i] = b[j] * c[i];
                                                    j += 10;
                                                }
```

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 9.8). While loop unswitching may double the amount of code written, each of these new loops may now be optimized separately.

Listing 9.8 Loop Unswitching

```
for (i = 0; i < N; i++) {
    a[i] += b[i];
    if (c)
        b[i] = 0;
}
=>
if (c)
    for (i = 0; i < N; i++) {
        a[i] += b[i];
        b[i] = 0;
    }
else
    for (i = 0; i < N; i++) {
        a[i] += b[i];
    }
```

9.3.2 High-level Optimizations.

High-level loop transformations change the structure of loops and often affect multiple nested loops. In this section, we will take a look at Loop Interchange, Loop Blocking (Tiling), Loop Fusion and Distribution (Fission), and Loop Unroll and Jam. This set of transformations aims at improving memory access and eliminating memory bandwidth and memory latency bottlenecks. From a compiler perspective, it is very difficult to prove the 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. Unfortunately, this means 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 9.9 shows an example of interchanging nested loops for *i* and *j*. The main purpose of this 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.

Listing 9.9 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];
```

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 nests is harder to do but still possible; check an example in the Codee¹⁷⁹ 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 into 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 9.10, 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*.

Listing 9.10 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];
                                            // remainder (not shown)
```

Loop Blocking is a widely known method of optimizing GEMM algorithms. It enhances the cache reuse of memory accesses and improves the memory bandwidth utilization and memory latency.

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

¹⁷⁹ Codee: perfect loop nesting - <https://www.codee.com/catalog/glossary-perfect-loop-nesting/>

of private caches are changing from generation to generation, so hardcoding a block size presents its own set of challenges. As an alternative solution, you can use [cache-oblivious](#)¹⁸⁰ algorithms whose goal is to work reasonably well for any size of the cache.

Loop Fusion and Distribution (Fission): separate loops can be fused 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 9.11. The opposite procedure is called Loop Distribution (Fission) when the loop is split into separate loops.

Listing 9.11 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;
```

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 9.11, 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 and increase 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 Frontend 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, you need to unroll the outer loop first, then jam (fuse) multiple inner loops together as shown in Listing 9.12. 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, the inner loop is a reduction operation that accumulates the deltas between elements of arrays `a` and `b`. When we unroll and jam the loop nest by a factor of two, we effectively execute two iterations of the original outer loop simultaneously. This is emphasized by having two independent accumulators. This breaks the dependency chain 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 the inner loop has memory accesses that are strided on the outer loop index (`i` in this case), otherwise,

¹⁸⁰ Cache-oblivious algorithm - https://en.wikipedia.org/wiki/Cache-oblivious_algorithm

Listing 9.12 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++) {
    diff1 += a[i][j] - b[i][j];      diff1 += a[i][j] - b[i][j];
    diff2 += a[i+1][j] - b[i+1][j];  diff2 += a[i+1][j] - b[i+1][j];
  }                                }
diffs = diff1 + diff2;
// remainder (not shown)

```

other transformations likely apply better. The Unroll and Jam technique 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.

The Unroll and Jam transformation sometimes is very useful for outer loop vectorization, which, at the time of writing, compilers cannot do automatically. In a situation when the trip count of the inner loop is not visible to a compiler, the compiler 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. By performing Unroll and Jam, we enable the compiler to vectorize the code differently: now “gluing” the independent instructions in the inner loop together. This technique is also known as Superword-Level Parallelism (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 to make 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 version of a loop automatically, but other times we have to do some rewriting 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.

As a first step, you can check compiler optimization reports or examine the machine code¹⁸¹ of the loop to search for easy improvements. Sometimes, it’s possible to adjust compiler transformations using user directives. There are compiler pragmas that control certain transformations, like loop vectorization, loop unrolling, loop distribution, and others. For a complete list of user directives, check your compiler’s manual.

Next, you should identify the bottlenecks in the loop and assess performance against the hardware theoretical maximum. The Top-down Microarchitecture Analysis (TMA, see Section 6.1) methodology and the Roofline performance model (Section 5.5) both help with that. The performance of a loop is limited by one or many of the following factors: memory latency, memory bandwidth, or the computing capabilities of a machine. Once the performance bottlenecks in a loop have been identified, try applying the required transformations that we discussed in a few previous sections.

¹⁸¹ Sometimes difficult, but always a rewarding activity.

Even though there are well-known optimization techniques for a particular set of computational problems, loop optimizations remain a “black art” that comes with experience. I recommend that you rely on your compiler and complement it with manually transforming 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.

9.4 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 toward discovering vectorization opportunities. For a recap of the SIMD capabilities of modern CPUs, readers can take a look at Section 3.4.

Vectorization often happens automatically without any user intervention; this is called compiler *autovectorization*. In such a situation, a compiler automatically recognizes the opportunity to produce SIMD machine code from the source code.

Autovectorization is very convenient because modern compilers can automatically generate fast SIMD code for a wide variety of programs. However, in some cases, autovectorization does not succeed without intervention by a software engineer. 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. We will provide several examples of using compiler autovectorization hints.

In this section, we will discuss how to harness compiler autovectorization, especially inner loop vectorization because it is the most common type of autovectorization. The other two types (outer loop vectorization, and Superword-Level Parallelism vectorization) are not discussed in this book.

9.4.1 Compiler Autovectorization

Multiple hurdles can prevent autovectorization, some of which are inherent to the semantics of programming languages. For example, the compiler must assume that 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, predicated (bitmask-controlled) load and store operations are not available on most processors. Despite all of the challenges, you can work around many of them and enable autovectorization. Later in this 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 legality phase collects a list of requirements that need to happen for the vectorization of the loop to be legal. 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 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.
- **Profitability-check:** next, the vectorizer checks if a transformation is profitable. It compares different vectorization widths and figures out which one 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, and 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 (remainder), 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.

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 reports (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 use 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 a high-level language like C++. It may take some time to figure out the semantics of the instructions generated by the

compiler. However, this skill is highly rewarding and often provides 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 the XMM register being used, that it is vector code—not necessarily. For instance, the VMULSS XMM1, XMM2, XMM3 instruction will only multiply one single-precision floating-point value, not four.

Another indicator for potential vectorization opportunities is a high `Retiring` metric (above 80%). In Section 6.1, we said that the `Retiring` metric is a good indicator of 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. Perhaps a workload executes a lot of simple instructions that can be replaced by vector instructions. In such situations, high `Retiring` metric doesn't translate into high performance.

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.

9.4.2.1 Vectorization Is Illegal. In some cases, the code that iterates over elements of an array is simply not vectorizable. Optimization reports are very effective at explaining what went wrong and why the compiler can't vectorize the code. Listing 9.13 shows an example of dependence inside a loop that prevents vectorization.¹⁸²

Listing 9.13 Vectorization: read-after-write dependence.

```
void vectorDependence(int *A, int n) {
    for (int i = 1; i < n; i++)
        A[i] = A[i-1] * 2;
}
```

While some loops cannot be vectorized due to the hard limitations (such as read-after-write dependence), others could be vectorized when certain constraints are relaxed. For example, the code in Listing 9.14 cannot be autovectorized by the compiler, because it will change the order of floating-point operations and may lead to different rounding and a slightly different result. Floating-point addition is 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, it is not associative, because rounding happens at different times: $((a + b) + c) != (a + (b + c))$.

If you tell the compiler that you can tolerate a bit of variation in the final result, it will autovectorize the code for you. Clang and GCC compilers have a flag, `-ffast-math`,¹⁸³ that allows this kind of transformation even though the resulting program may give slightly different results:

¹⁸² 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.

¹⁸³ The compiler flag `-Ofast` enables `-ffast-math` as well as the `-O3` compilation mode.

Listing 9.14 Vectorization: floating-point arithmetic.

```

1 // a.cpp
2 float calcSum(float* a, unsigned N) {
3     float sum = 0.0f;
4     for (unsigned i = 0; i < N; i++) {
5         sum += a[i];
6     }
7     return sum;
8 }
```

```

$ clang++ -c a.cpp -O3 -march=core-avx2 -Rpass-analysis=.*
...
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 -march=core-avx2 -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. Since Clang 18, you can limit the scope of transformations by using dedicated pragmas, e.g., `#pragma clang fp reassociate(on)`.¹⁸⁴

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. Given the code in Listing 9.15, compilers should account for the situation when the memory regions of arrays `a`, `b`, and `c` overlap.

Listing 9.15 a.c

```

1 void foo(float* a, float* b, float* c, unsigned N) {
2     for (unsigned i = 1; i < N; i++) {
3         c[i] = b[i];
4         a[i] = c[i-1];
5     }
6 }
```

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 and created multiple

¹⁸⁴ LLVM extensions to specify floating-point flags - <https://clang.llvm.org/docs/LanguageExtensions.html#extensions-to-specify-floating-point-flags>

versions of the loop. The compiler inserted runtime checks¹⁸⁵ to detect if the memory regions overlap. Based on those checks, it dispatches between vectorized and scalar 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`¹⁸⁶ right before the loop or use the `__restrict__` keyword as shown in Section 5.7. Such compiler hints will eliminate the need for the GCC compiler to insert the runtime checks mentioned earlier.

Some dynamic tools, such as Intel Advisor, can detect if issues like cross-iteration dependence or access to arrays with overlapping memory regions occur in a 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 decide that doing so is not profitable. In the code presented in Listing 9.16, 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.

Listing 9.16 Vectorization: not beneficial.

```
1 // a.cpp
2 void stridedLoads(int *A, int *B, int n) {
3     for (int i = 0; i < n; i++)
4         A[i] += B[i * 3];
5 }
```

Here is the compiler optimization report for the code in Listing 9.16:

```
$ 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 9.17. However, keep in mind that whether vectorization is profitable largely depends on the runtime data, for example, the number of iterations of the loop. Compilers don't have this information available,¹⁸⁷ so they often tend to be conservative. Though you can use such hints for performance experiments.

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 for several microseconds. The vectorized portion of the code should be hot enough to justify using AVX-512.¹⁸⁸ For example, sorting 80 KiB was found to be sufficient to amortize

¹⁸⁵ See the example on the Easyperf blog: https://easyperf.net/blog/2017/11/03/Multiversioning_by_DD.html.

¹⁸⁶ It is a GCC-specific pragma. For other compilers, check the corresponding manuals.

¹⁸⁷ Besides Profile Guided Optimizations (see Section 11.7).

¹⁸⁸ For more details read this blog post: <https://travisdowns.github.io/blog/2020/01/17/avxfreq1.html>.

Listing 9.17 Vectorization: not beneficial.

```

1 // a.cpp
2 void stridedLoads(int *A, int *B, int n) {
3 #pragma clang loop vectorize(enable)
4   for (int i = 0; i < n; i++)
5     A[i] += B[i * 3];
6 }
```

this overhead and make vectorization worthwhile.¹⁸⁹

9.4.2.3 Loop Vectorized but Scalar Version Used. In some scenarios, the compiler successfully vectorizes the code, but it does not show up as being executed 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 vector registers, which are not commonly used in other parts of the program.

If the vector code is not executed, one possible reason for this is that the generated code assumes loop trip counts that are higher than what the program uses. For example, a compiler may decide to vectorize and unroll the loop in such a way as to process 64 elements per iteration. An input array may not have enough elements even for a single iteration of the loop. In this case, the scalar version (remainder) of the loop will be used instead. It is easy to detect these cases because the scalar 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 the loop processes. You can achieve that with the help of `#pragma` hints. For the Clang compiler, you can use `#pragma clang loop vectorize_width(N)` as shown in the article on the Easyperf blog.¹⁹⁰

9.4.2.4 Loop Vectorized in a Suboptimal Way. When you see a loop being autovectorized and executed at runtime, there is a high chance that this part of the program already performs well. However, there are exceptions. 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, inserting/extracting elements, data shuffling, etc., if the compiler is required to use them 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 to a particular loop, e.g., `#pragma clang loop vectorize(disable)`.

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 example can be found in [Mula & Lemire, 2019]. Another example is outer loop autovectorization, which is not currently attempted by compilers. Vectorizing floating-point code is problematic because reordering arithmetic floating-point operations

¹⁸⁹ Study of AVX-512 downclocking: in VQSort readme

¹⁹⁰ Using Clang's optimization pragmas - https://easyperf.net/blog/2017/11/09/Multiversioning_by_trip_counts

results in different rounding and slightly different values.

There is one more subtle problem with autovectorization. As compilers evolve, optimizations that they make are changing. The successful autovectorization of code that was done in the previous compiler version may stop working in the next version, or vice versa. Also, during code maintenance or refactoring, the structure of the code 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.

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](#)),¹⁹¹ which I will briefly cover in this section. The ISPC language is based on the C programming language and uses the LLVM compiler infrastructure to generate 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 into any native project since functions written with ISPC can be called as if it was C code.

[Listing 9.18](#) shows an ISPC version of a function that I presented earlier in [Listing 9.14](#). ISPC considers that the program will run in parallel instances, based on the target instruction set. For example, when using SSE with `f16`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.

¹⁹¹ ISPC compiler: <https://ispc.github.io/>.

Listing 9.18 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);
}
```

Since the function `calcSum` must return a single value (a `uniform` variable) and our `sum` variable is `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 the compiler doesn’t always vectorize critical parts of code. 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 more readable than intrinsics (see Section 9.5) as it allows you to focus on the algorithm rather than the low-level instructions. Also, it has reportedly matched [Pharr & Mark, 2012] or beaten¹⁹² 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, AVX2, and ARM NEON.

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. Sometimes human experts can come up with code that outperforms the one generated by the compiler. It usually involves some tricky or specialized algorithms, that can be very hard or even impossible for the compiler to figure out. In such cases, there could be no way to make the compiler generate the desired assembly code using standard constructs of the C and C++ languages.

When it is absolutely necessary to generate specific assembly instructions, you should not rely on compiler autovectorization. In such cases, code can instead be written using *compiler intrinsics*. In most cases, compiler intrinsics provide a 1-to-1 mapping to assembly instructions. An example in Listing 9.19 shows how the C++ version of the horizontal sum of elements in an array (see Listing 9.14) can be coded with compiler intrinsics.

¹⁹² Some parts of the Unreal Engine that 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>.

Listing 9.19 Summing elements of an array with compiler intrinsics.

```

1 #include <immintrin.h>
2
3 float calcSum(float* a, unsigned N) {
4     __m128 sum = _mm_setzero_ps();           // init sum with zeros
5     unsigned i = 0;
6     for (; i + 3 < N; i += 4) {
7         __m128 vec = _mm_loadu_ps(a + i); // load 4 floats from array
8         sum = _mm_add_ps(sum, vec);      // accumulate vec into sum
9     }
10
11    // Horizontal sum of the 128-bit vector
12    __m128 shuf = _mm_movehdup_ps(sum); // broadcast elements 3,1 to 2,0
13    sum = _mm_add_ps(sum, shuf);        // partial sums [0+1] and [2+3]
14    shuf = _mm_movelh_ps(shuf, sum);   // high half -> low half
15    sum = _mm_add_ss(sum, shuf);       // result in the lower element
16    float result = _mm_cvtsd_f32(sum); // nop (compiler eliminates it)
17
18    // Process any remaining elements
19    for (; i < N; i++)
20        result += a[i];
21    return result;
22 }
```

When compiling Listing 9.14 for the SSE target, compilers would generate mostly identical assembly code as for Listing 9.19. I show this example just for illustration purposes. Obviously, there is no need to use intrinsics if the compiler can generate the same machine code. You should use intrinsics only when the compiler fails to generate the desired code.

When you leverage compiler auto-vectorization, it will insert all necessary runtime checks. For instance, it will ensure that there are enough elements to feed the vector execution units (see Listing 9.19, line 6). Also, the compiler will generate a scalar version of the loop to process the remainder (line 19). When you use intrinsics, you have to take care of safety aspects yourself.

Intrinsics are better to use than inline assembly because the compiler performs type checking, takes care of register allocation, and makes further optimizations, e.g., peephole transformations and instruction scheduling. However, they are still often verbose and difficult to read.

When you write code using non-portable platform-specific intrinsics, you should also provide a fallback option for other architectures. A list of all available intrinsics for the Intel platform can be found in this reference¹⁹³. For ARM, you can find such a list on the Arm's website.¹⁹⁴

9.5.1 Wrapper Libraries for Intrinsics

For a middle path between low-effort but unpredictable autovectorization, and verbose/unreadable but predictable intrinsics, you can use a wrapper library around intrinsics. These libraries tend to be more readable, provide portability, and still give

¹⁹³ Intel intrinsics guide - <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>.

¹⁹⁴ ARM intrinsics guide - <https://developer.arm.com/architectures/instruction-sets/intrinsics/>

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.

The write-once, target-many model of ISPC is appealing. However, you 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.

Wrapper libraries combine the advantages and avoid drawbacks of both 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’. 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 such a library is Highway,¹⁹⁵ which only requires the C++11 standard.

The Highway version of summing elements of an array is presented in Listing 9.20. The `ScalableTag<float>` `d` is a type descriptor that represents a “scalable” type, meaning that it can adjust to the available vector width on the target hardware (e.g., AVX2 or NEON). `Zero(d)` initializes `sum` to a vector filled with zeros. This variable will store the accumulated sum as the function iterates through the `array`. The for loop processes `Lanes(d)` elements at a time, where `Lanes(d)` represents the number of floats that can be loaded into a single SIMD vector. The `LoadU` operation loads `Lanes(d)` consecutive elements from `array`. The `Add` operation performs an element-wise addition of the loaded values with the current `sum`, accumulating the result in `sum`.

Listing 9.20 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);
}
```

Notice the explicit handling of remainders after the loop processes multiples of the vector size `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 the `count` is known to be a multiple of the vector size. Finally, the `ReduceSum` operation reduces all elements in the vector `sum` to a single scalar value by adding them together.

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),

¹⁹⁵ Highway library: <https://github.com/google/highway>

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.

When you use 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.

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.¹⁹⁶ Highway is not the only library of this kind. Other libraries include nsimd, SIMD, VCL, and xsimd. Note that a C++ standardization effort starting with the Vc library resulted in std::experimental::simd, however, it provides a very limited set of operations and as of this writing is not supported on all the major compilers.

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` and `2`
 - `perf-ninja::dep_chains_1` and `2`
 - `perf-ninja::compiler_intrinsics_1` and `2`
 - `perf-ninja::loop_interchange_1` and `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. 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 daily. Find the hottest loop. Is it vectorized? Is it possible to force compiler autovectorization? Is the loop bottlenecked by dependency chains or execution throughput?

¹⁹⁶ Highway Quick Reference - https://github.com/google/highway/blob/master/g3doc/quick_reference.md

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 to search performance headrooms in a program by forcing certain code optimizations. We discussed such popular transformations like function inlining, loop optimizations, and vectorization.

10 Optimizing Branch Prediction

So far we've been talking about optimizing memory accesses and computations. However, we haven't discussed another important category of performance bottlenecks 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 the next result will be. However, when the pattern becomes hard for the CPU branch predictor to follow, it may hurt performance.

Mispredicting a branch can add a significant penalty when it happens regularly. When such an event occurs, 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 a 10- to 25-cycle 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 a mispredict.

Perhaps the most frequent reason for a branch misprediction is simply because it has a complicated outcome pattern (e.g., exhibits pseudorandom behavior), which is unpredictable for a processor. For completeness, let's cover the other less frequent reasons behind branch mispredicts. Branch predictors use caches and history registers and therefore are susceptible to the issues related to caches, namely:

- **Cold misses:** mispredictions may happen on the first dynamic occurrence of the branch when no dynamic history is available and static prediction is employed.
- **Capacity misses:** mispredictions arising from the loss of dynamic history due to a 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 aliasing 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 experience a non-zero number of branch mispredictions. You can find out how much a program suffers from branch mispredictions by looking at the TMA Bad Speculation metric. It is normal for a general-purpose application to have a Bad Speculation metric in the range of 5–10%. My recommendation is to pay close attention once this metric goes higher than 10%.

In the past, developers had an option of providing a prediction hint to an x86 processor in the form of a prefix to the branch instruction (0x2E: Branch Not

Taken, 0x3E: Branch Taken). This could potentially improve performance on older microarchitectures, like Pentium 4. However, modern x86 processors used to ignore those hints until Intel’s RedwoodCove started using it again. Its branch predictor is still good at finding dynamic patterns, but now it will use the encoded prediction hint for branches that have never been seen before (i.e. when there is no stored information about a branch). [Intel, 2023, Section 2.1.1.1 Branch Hint]

There are indirect ways to reduce the branch misprediction rate by reducing the dynamic number of branch instructions. This approach helps because it alleviates the pressure on branch predictor structures. When a program executes fewer branch instructions, it may indirectly improve the prediction of branches that previously suffered from capacity and conflict misses. Compiler transformations such as loop unrolling and vectorization help reduce the dynamic branch count, though they don’t specifically aim to improve the prediction rate of any given conditional statement. Profile-Guided Optimizations (PGO) and post-link optimizers (e.g., BOLT) are also effective at reducing branch mispredictions thanks to improving the fallthrough rate (straightening the code). We will discuss those techniques in the next chapter.¹⁹⁷

The only direct way to get rid of branch mispredictions is to get rid of the branch instruction itself. In subsequent sections, we will take a look at both direct and indirect ways to improve branch prediction. In particular, we will explore the following techniques: replacing branches with lookup tables, arithmetic, selection, and SIMD instructions.

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 10.1. 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.

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. 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

¹⁹⁷ 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 a 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.

Listing 10.1 Replacing branches with lookup tables.

```

int8_t mapToBucket(unsigned v) {      int8_t buckets[50] = {
    if      (v < 10) return 0;          0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    else if (v < 20) return 1;          1, 1, 1, 1, 1, 1, 1, 1, 1,
    else if (v < 30) return 2;          => 2, 2, 2, 2, 2, 2, 2, 2, 2,
    else if (v < 40) return 3;          3, 3, 3, 3, 3, 3, 3, 3,
    else if (v < 50) return 4;          4, 4, 4, 4, 4, 4, 4, 4 };
    return -1;
}
int8_t mapToBucket(unsigned v) {
    if (v < (sizeof(buckets) / sizeof(int8_t)))
        return buckets[v];
    return -1;
}

```

not practical. In this case, we might use interval map data structures that accomplish that goal using much less memory but logarithmic lookups complexity. Readers can find existing implementations of interval map container in [Boost¹⁹⁸](#) and [LLVM¹⁹⁹](#).

10.2 Replace Branches with Arithmetic

In some scenarios, branches can be replaced with arithmetic. The code in Listing 10.1 can also be rewritten using a simple arithmetic formula, as shown in Listing 10.2. Notice, that for this code, the Clang-17 compiler replaced expensive division with much cheaper multiplication and right shift operations.

Listing 10.2 Replacing branches with arithmetic.

<pre> int8_t mapToBucket(unsigned v) { constexpr unsigned BucketRangeMax = 50; if (v < BucketRangeMax) return v / 10; return -1; } </pre>	<pre> mov al, -1 cmp edi, 49 ja .exit movzx eax, dil imul eax, eax, 205 shr eax, 11 .exit: ret </pre>
--	---

As of the year 2024, 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 frequently mispredicted branch with arithmetic, you will likely see a performance improvement.

10.3 Replace Branches with Selection

Some branches could be effectively eliminated by executing both parts of the branch and then selecting the right result. An example of code when such transformation might be profitable is shown in Listing 10.3. 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.

¹⁹⁸ C++ Boost `interval_map` - https://www.boost.org/doc/libs/1_65_0/libs/icl/doc/html/boost_icl/interval_map.html

¹⁹⁹ LLVM's `IntervalMap` - https://llvm.org/doxygen/IntervalMap_8h_source.html

Listing 10.3 Replacing Branches with Selection.

```

int a;
if (cond) { /* frequently mispredicted */ =>      int x = computeX();
    a = computeX();                                int y = computeY();
} else {                                         int a = cond ? x : y;
    a = computeY();                                foo(a);
}
foo(a);

```

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. A similar transformation can be done for floating-point numbers with FCMOVcc and VMAXSS/VMINSS instructions. In the ARM ISA, there is the CSEL (conditional selection) instruction, but also CSINC (select and increment), CSNEG (select and negate), and a few other conditional instructions.

Listing 10.4 Replacing Branches with Selection - x86 assembly code.

<i># original version</i>	<i># branchless version</i>
400504: test ebx,ebx	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	400552: mov edi,eax
400521: call <foo>	400554: call <foo>

Listing 10.4 shows assembly listings for the original and the branchless version. 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 doing more work than the original code.

We already know that the branch in the original version on the left is hard to predict. This is what motivated us to try a branchless version in the first place. In this example, the performance gain of this change depends on the characteristics of the `computeX` and `computeY` functions. If the functions are small²⁰⁰ and the compiler can inline them, then selection might bring noticeable performance benefits. If the functions are big²⁰¹, it might be cheaper to take the cost of a branch mispredict than to execute both `computeX` and `computeY` functions. Ultimately, performance measurements always decide which version is better.

Take a look at Listing 10.4 one more time. On the left, a processor can predict, for example, that the `je 400514` branch will be taken, speculatively call `computeY`, and start running code from the function `foo`. Remember, branch prediction happens

²⁰⁰ Just a handful of instructions that can be completed in a few cycles.

²⁰¹ More than twenty instructions that take more than twenty cycles.

many cycles before we know the actual outcome of the branch. By the time we start resolving the branch, we could be already halfway through the `foo` function, despite it is still speculative. If we are correct, we've saved a lot of cycles. If we are wrong, we have to take the penalty and start over from the correct path. In the latter case, we don't gain anything from the fact that we have already completed a portion of `foo`, it all must be thrown away. If the mispredictions occur too often, the recovering penalty outweighs the gains from speculative execution.

With conditional selection, it is different. There are no branches, so the processor doesn't have to speculate. It can execute `computeX` and `computeY` functions in parallel. However, it cannot start running the code from `foo` until it computes the result of the `CMOVNE` instruction since `foo` uses it as an argument (data dependency). When you use conditional select instructions, you convert a control flow dependency into a data flow dependency.

To sum it up, for small `if-else` statements that perform simple operations, conditional selects can be more efficient than branches, but only if the branch is hard to predict. So don't force the compiler to generate conditional selects for every conditional statement. For conditional statements that are always correctly predicted, having a branch instruction is likely an optimal choice, because you allow the processor to speculate (correctly) and run ahead of the actual execution. And don't forget to measure the impact of your changes.

Without profiling data, compilers don't have visibility into the misprediction rates. As a result, they usually prefer to generate branch instructions by default. Compilers are conservative at using selection 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.²⁰² Starting from Clang-17, the compiler now honors a `__builtin_unpredictable` hint for the x86 target, which indicates to the compiler that a branch condition is unpredictable. It can help influence the compiler's decision but does not guarantee that the `CMOV` instruction will be generated. Here's an example of how to use `__builtin_unpredictable`:

```
int a;
if (__builtin_unpredictable(cond)) {
    a = computeX();
} else {
    a = computeY();
}
```

10.4 Multiple Tests Single Branch

The last technique that we discuss in this chapter aims at minimizing the dynamic number of branch instructions by combining multiple tests. The main idea here is to avoid executing a branch for every element of a large array. Instead, the goal is to perform multiple tests simultaneously, which primarily involves using SIMD instructions. The result of multiple tests is a vector mask that can be converted into a byte mask, which often can be processed with a single branch instruction. This enables us to eliminate many branch instructions as you will see shortly. You may encounter this technique being used in SIMD implementations of various algorithms such as JSON/HTML parsing, media codecs, and others.

²⁰² Hardware-based PGO (see Section 11.7) will be a huge step forward here.

Listing 10.5 shows a function that finds the longest line in an input string by testing one character at a time. We go through the input string and search for end-of-line (`eol`) characters (`\n`, `0x0A` in ASCII). For every found `eol` character we check if the current line is the longest, and reset the length of the current line to zero. This code will execute one branch instruction for every character.²⁰³

Listing 10.5 Find the longest line (one character at a time).

```
uint32_t longestLine(const std::string &str) {
    uint32_t maxLen = 0;
    uint32_t curLen = 0;
    for (auto s : str) {
        if (s == '\n') {
            maxLen = std::max(curLen, maxLen);
            curLen = 0;
        } else {
            curLen++;
        }
    }
    // if no end-of-line in the end
    maxLen = std::max(curLen, maxLen);
    return maxLen;
}
```

Consider the alternative implementation shown in Listing 10.6 that tests eight characters at a time. You will typically see this idea implemented using compiler intrinsics (see Section 9.5), however, I decided to show a standard C++ code for clarity. This exact case is featured in one of Performance Ninja’s lab assignments,²⁰⁴ so you can try writing SIMD code yourself. Keep in mind, that the code I’m showing is incomplete as it misses a few corner cases; I provide it just to illustrate the idea.

We start by preparing an 8-byte mask filled with `eol` symbols. The inner loop loads eight characters of the input string and performs a byte-wise comparison of these characters with the `eol` mask. Vectors in modern processors contain 16/32/64 bytes, so we can process even more characters simultaneously. The result of the eight comparisons is an 8-bit mask with either 0 or 1 in the corresponding position (see `compareBytes`). For example, when comparing `0x0OFF0AO00AFFFF00` and `0x0AOAOAOAOAOAOAOA`, we will get `0b00101000` as a result. With x86 and ARM ISAs, the function `compareBytes` can be implemented using two vector instructions.²⁰⁵

If the mask is zero, that means there are no `eol` characters in the current chunk and we can skip it (see line 11). This is a critical optimization that provides large speedups for input strings with long lines. If a mask is not zero, that means there are `eol` characters and we need to find their positions. To do so, we use the `tzcnt` function, which counts the number of trailing zero bits in an 8-bit mask (the position of the rightmost set bit). For example, for the mask `0b00101000`, it will return 3. Most ISAs support implementing the `tzcnt` function with a single instruction.²⁰⁶ Line 14

²⁰³ Assuming that the compiler will avoid generating branch instructions for `std::max`.

²⁰⁴ Performance Ninja: compiler intrinsics 2 - https://github.com/dendibakh/perf-ninja/tree/main/labs/core_bound/compiler_intrinsics_2.

²⁰⁵ For example, with AVX2 (256-bit vectors), you can use `VPCMPEQB` and `VPMOVMSKB` instructions.

²⁰⁶ Although in x86, there is no version of the `TZCNT` instruction that supports 8-bit inputs.

Listing 10.6 Find the longest line (8 characters at a time).

```

1 uint32_t longestLine(const std::string &str) {
2     uint32_t maxLen = 0;
3     const uint64_t eol = 0x0a0a0a0a0a0a0a0a;
4     auto *buf = str.data();
5     uint32_t lineBeginPos = 0;
6     for (uint32_t pos = 0; pos + 7 < str.size(); pos += 8) {
7         // Load 8-byte chunk of the input string.
8         uint64_t vect = *((const uint64_t*)(buf + pos));
9         // Check all characters in this chunk.
10        uint8_t mask = compareBytes(vect, eol);
11        while (mask) {
12            uint16_t eolPos = tzcnt(mask);
13            // Compute the length of the current string.
14            uint32_t curLen = (pos - lineBeginPos) + eolPos;
15            // New line starts with the character after '\n'
16            lineBeginPos += curLen + 1;
17            // Is this line the longest?
18            maxLen = std::max(curLen, maxLen);
19            // Shift the mask to check if we have more '\n'
20            mask >>= eolPos + 1;
21        }
22    }
23    // process remainder (not shown)
24    return maxLen;
25 }
26
27 uint8_t compareBytes(uint64_t a, uint64_t b) {
28     // Perform a byte-wise comparison of a and b.
29     // Produce a bit mask with the result of comparisons:
30     // one if bytes are equal, zero if different.
31 }
32
33 uint8_t tzcnt(uint8_t mask) {
34     // Count the number of trailing zero bits in the mask.
35 }
```

calculates the length of the current line using the result of the `tzcnt` function. We shift right the mask and repeat until there are no set bits in the mask.

For an input string with a single very long line (best case scenario), the SIMD version will execute eight times fewer branch instructions. However, in the worst-case scenario with zero-length lines (i.e., only `eol` characters in the input string), the original approach is faster. I benchmarked this technique using AVX2 implementation (with chunks of 16 characters) on several different inputs, including textbooks, and source code files. The result was 5–6 times fewer branch instructions and more than 4x better performance when running on Intel Core i7-1260P (12th Gen, Alder Lake).

Questions and Exercises

1. Revisit the code example shown in Listing 10.1 on the right. Suppose we start frequently getting numbers outside of the [0–50) range. This will introduce many new mispredictions for the branch that guards against out-of-bounds access to the `buckets` array. How you will change the code to eliminate those newly

introduced mispredictions?

2. 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`

3. Run the application that you're working with daily. 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 the techniques we discussed in this chapter?

Coding exercise: write a microbenchmark that will experience a 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) the Google benchmark library similar to perf-ninja, 2) write a regular console program and collect CPU counters with Linux `perf`, or 3) integrate the libpfm library into the microbenchmark (see Section 5.3.2).
- 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 pre-generated 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, I recommend paying attention to branch mispredictions only when the TMA 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, I showed how branches could be replaced with lookup tables, arithmetic, and selection.
- Branchless algorithms are not universally beneficial. Always measure to find out what works better in your specific case.
- There are indirect ways to reduce the branch misprediction rate by reducing the dynamic number of branch instructions in a program. This approach helps because it alleviates the pressure on branch predictor structures. Examples of such techniques include loop unrolling/vectorization, replacing branches with bitwise operations, and using SIMD instructions.

11 Machine Code Layout Optimizations

The CPU Frontend (FE) is responsible for fetching and decoding instructions and delivering them to the out-of-order Backend (BE). As the newer processors get more execution “horsepower”, the 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 Backend is waiting for instructions to execute, but the Frontend 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 9-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 **Frontend 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 ‘Frontend 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 “Frontend Bound” metric being more than 20%, it’s worth spending 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 poor I-cache and ITLB utilization. Applications with a large codebase, e.g., millions of 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.

11.1 Machine Code Layout

When a compiler translates source code into machine code, it generates a linear byte sequence. Listing 11.1 shows an example of a binary layout for a small snippet of C++ code. Once the compiler finishes generating assembly instructions, it needs to encode them and lay them out in memory sequentially.

The way code is placed in an object file 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 11.1, a 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 executable image 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 program as you will see later. In the following sections of this chapter, we will take a look at some typical

Listing 11.1 Example of machine code layout

C++ Code	Assembly Listing	Disassembled Machine Code
..... if (<i>a</i> <= <i>b</i>) bar (); else baz (); ; <i>a</i> is in edi ; <i>b</i> is in esi cmp esi, edi jb .label1 call bar() jmp .label2 .label1: call baz() .label2: 401125 cmp esi, edi 401128 jb 401131 40112a call bar 40112f jmp 401136 401131 call baz 401136 ...

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 11.1 shows a simple example of a basic block, where the MOV instruction is an entry, and JA is an exit instruction. Basic block BB1 is a *predecessor* of basic block BB2 if the control flow can go from BB1 to BB2. Similarly, basic block BB3 is a *successor* of basic block BB2 if the control flow can go from BB2 to BB3. While a basic block can have one or many predecessors and successors, no instruction in the middle can enter or exit a basic block.

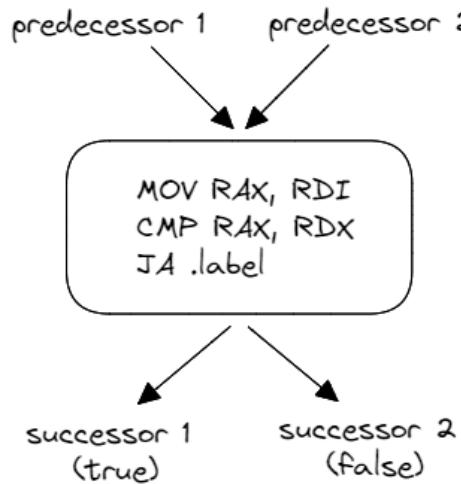


Figure 11.1: Basic Block of assembly instructions.

It is guaranteed that every instruction in the basic block will be executed only 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 classes 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 11.2 shows two possible physical layouts for this snippet of code. Figure 11.2a is the layout most compilers will emit by default, given no hints are provided. The layout that is shown in Figure 11.2b can be achieved if we invert the condition `cond` and place hot code as fall through.

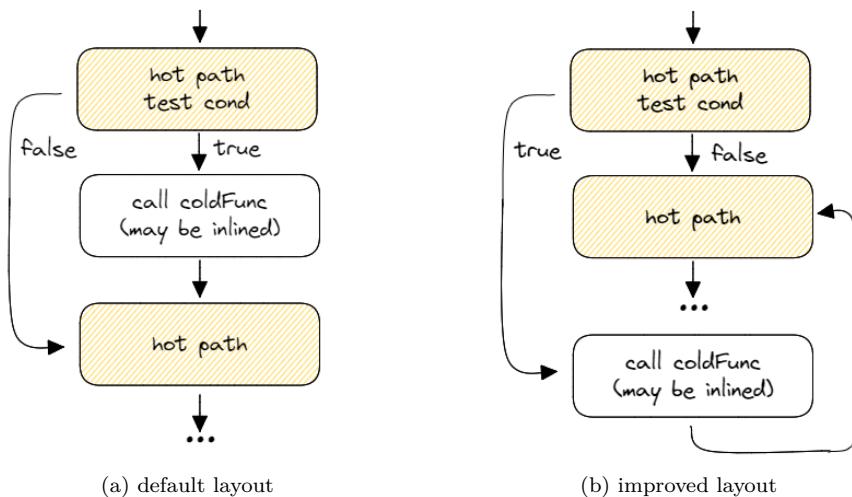


Figure 11.2: Two versions of machine code layout for the snippet of code above.

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, 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 11.2b, we maintain fall through between hot pieces of the code and convert the taken branch into not taken one.

There are a few reasons why the layout presented in Figure 11.2b performs better. First of all, the layout in Figure 11.2b makes better use of the instruction and μ op-cache (DSB, see Section 3.8.1). With all hot code contiguous, there is no cache line fragmentation: all the cache lines in the L1 I-cache are used by hot code. The same is true for the μ op-cache since it caches based on the underlying code layout as well. Secondly, taken branches are also more expensive for the fetch unit. The Frontend of a CPU fetches contiguous aligned blocks of bytes, usually 16, 32, or 64 bytes, depending on the architecture. For every taken branch, bytes in a fetch block after the jump instruction and before the branch target are unused. 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.²⁰⁷

To suggest a compiler to generate an improved version of the machine code layout, you can provide a hint using `[[unlikely]]` and `[[unlikely]]` attributes, which have been 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, the `[[unlikely]]` hint will instruct the compiler that `cond` is unlikely to be true, so the compiler should adjust the code layout accordingly. Prior to C++20, developers could have used the `__builtin_expect`²⁰⁸ construct. They usually created `LIKELY` wrapper hints 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 the `[[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 the `[[likely]]` attribute is also possible for a switch statement as presented in Listing 11.2. 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.

Listing 11.2 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
    }
}
```

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 11.3 along with the corresponding machine code when compiled with `-O3`

²⁰⁷ However, there is a special small loop optimization that allows very small loops to have one taken branch per cycle.

²⁰⁸ More about builtin-expect here: <https://llvm.org/docs/BranchWeightMetadata.html#builtin-expect>.

`-march=core-avx2 -fno-unroll-loops` (loop unrolling is disabled to illustrate the idea).

Listing 11.3 Basic block alignment

<pre>void benchmark_func(int* a) { for (int i = 0; i < 32; ++i) a[i] += 1; }</pre>	<pre>00000000004046a0 <_Z14benchmark_funcPi>: 4046a0: mov rax,0xfffffffffffff80 4046a7: vpcmpeqd ymm0,ymm0,ymm0 4046ab: nop DWORD [rax+rax+0x0] 4046b0: vmovdqu ymm1,[rdi+rax+0x80] # loop begins 4046b9: vpsubd ymm1,ymm1,ymm0 4046bd: vmovdqu [rdi+rax+0x80],ymm1 4046c6: add rax,0x20 4046ca: jne 4046b0 # loop ends 4046cc: vzeroupper 4046cf: ret</pre>
---	---

The code itself is reasonable, but its layout is not perfect (see Figure 11.3a). Instructions that correspond to the loop are highlighted in yellow. Thick boxes denote cache line borders. Cache lines are 64 bytes long.

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 sometimes cause performance problems for the CPU Frontend, especially for the small loops like those presented in Listing 11.3.

To fix this, we can shift the loop instructions forward by 16 bytes using a single NOP instruction so that the whole loop will reside in one cache line. Figure 11.3b shows the effect of doing this with the NOP instruction highlighted in blue.

Interestingly, the performance impact is visible even if 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 L1 I-cache size on any modern CPU. The reason for the better performance of the layout in Figure 11.3b 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 related article on the Easyperf blog.²⁰⁹

By default, the LLVM compiler recognizes loops and aligns them at 16B boundaries, as we saw in Figure 11.3a. To reach the desired code placement for our example, as shown in Figure 11.3b, you 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, I do not recommend using this and similar options as they affect the code layout of all the functions in the translation unit. There are other less intrusive options.

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. The following code shows how the new Clang attribute can be used to align a loop at a 64 bytes boundary:

²⁰⁹ “Code alignment issues” - https://easyperf.net/blog/2018/01/18/Code_alignment_issues

(a) default layout

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x80																
0x90																
0xA0	mov	vpcmp	nop
0xB0	vmov	vpsub	vmov
0xC0	add	jne	vzero	ret		
0xD0																
0xE0																
0xF0																

(b) improved layout

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x80																
0x90																
0xA0	mov	vpcmp	nop
0xB0	nop
0xC0	vmov	vpsub	vmov
0xD0	add	jne	nop
0xE0	vzero	ret												
0xF0																

Figure 11.3: Two different code layouts for the loop in Listing 11.3.

```
void benchmark_func(int* a) {
    [[clang::code_align(64)]]
    for (int i = 0; i < 32; ++i)
        a[i] += 1;
}
```

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.

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 the hot code from the cold. Such transformation is also often called *function outlining*. This optimization is beneficial for relatively big functions with a 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 11.4. To remove cold basic blocks from the hot path, we cut and paste them into a new function and create a call to it.

Listing 11.4 Function splitting: cold code outlined to the new functions.

<pre>void foo(bool cond1, bool cond2) { // hot path if (cond1) { /* cold code (1) */ } // hot path if (cond2) { /* cold code (2) */ } }</pre>	<pre>void foo(bool cond1, bool cond2) { // hot path if (cond1) { cold1(); } // hot path if (cond2) { cold2(); } } void cold1() __attribute__((noinline)) { /* cold code (1) */ } void cold2() __attribute__((noinline)) { /* cold code (2) */ }</pre>
---	---

Notice, that we disable the inlining of cold functions by using the `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.

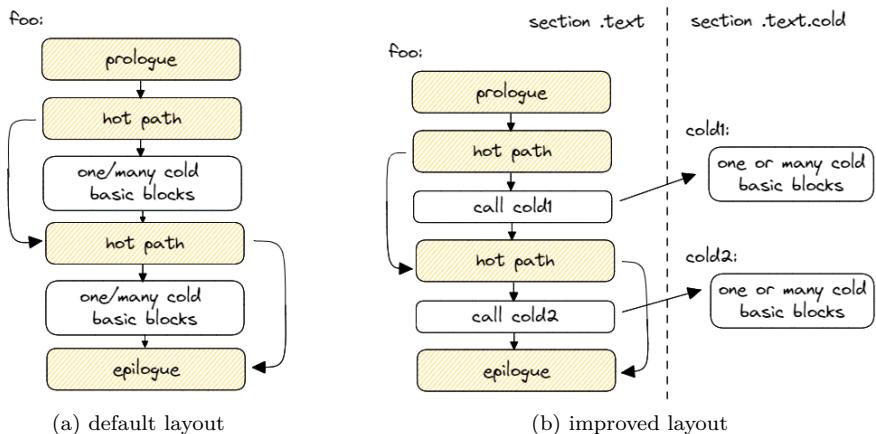


Figure 11.4: Splitting cold code into a separate function.

Figure 11.4 gives a graphical representation of this transformation. In the improved layout, we left just a `CALL` instruction inside the hot path, the next hot instruction will likely reside in the same cache line as the previous one. This improves the utilization of CPU Frontend data structures such as I-cache and *μop*-cache.

Outlined functions should be created outside of the `.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 Frontend. When hot functions are grouped, they start sharing cache lines, which reduces the *code footprint*, the total number of cache lines a CPU needs to fetch.

Figure 11.5 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 11.5a), 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.²¹⁰

We can rearrange the order of the functions such that hot functions are placed close to each other (see Figure 11.5b). In the improved version, the code of the `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.

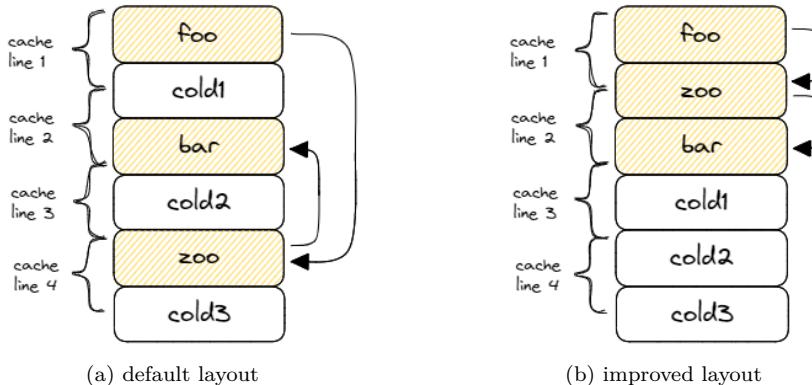


Figure 11.5: Reordering hot functions.

Similar to previous optimizations, function reordering improves the utilization of I-cache and μ op-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 of the desired physical layout. For decades people have been using linker scripts to achieve this goal. This is still 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, you can first compile the code with the `-ffunction-sections` flag, which will put each function

²¹⁰ Also, functions located in shared libraries do not participate in the careful layout of machine code.

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 the LLVM compiler infrastructure and is accessible via the `--symbol-ordering-file` option.

An interesting approach to solving the problem of grouping hot functions was introduced in 2017 by engineers from Meta. They implemented a tool called [HFSort²¹¹](#), 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 linker²¹². Since then, the algorithm has been superseded first by HFSort+, and most recently by Cache-Directed Sort (CDSort²¹³), with more improvements for workloads with a 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 the 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, a compiler could take into account the number of times this function will be called. The problem is that the 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 guess.

Here is when profiling information becomes handy. Given profiling information, the compiler can make better optimization decisions. There is a set of transformations in most compilers that can adjust their algorithms based on profiling data provided 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 11.6 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. The second step is to run the instrumented binary with 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 the 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

²¹¹ HFSort - <https://github.com/facebook/hhvm/tree/master/hphp/tools/hfsort>

²¹² HFSort in LLD - <https://github.com/llvm-project/lld/blob/master/ELF/CallGraphSort.cpp>

²¹³ Cache-Directed Sort in LLVM - <https://github.com/llvm/llvm-project/blob/main/llvm/lib/Transforms/Utils/CodeLayout.cpp>

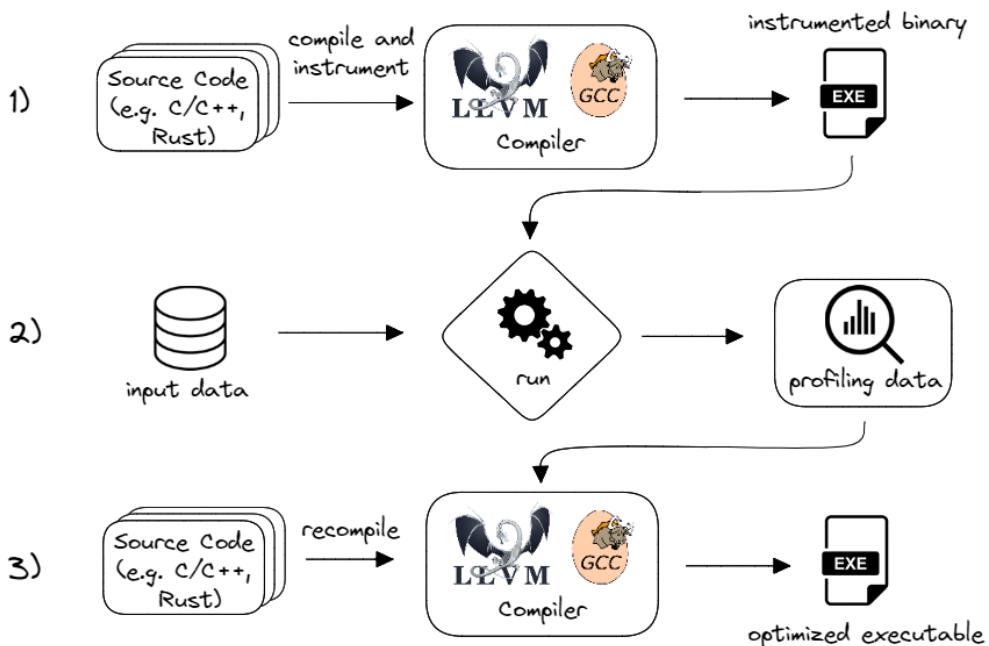


Figure 11.6: Instrumented PGO workflow.

the compiler to add instrumentation code to collect profiling information at runtime. Such information usually includes the number of times we enter every basic block in a program.

After that, the LLVM compiler can consume profiling data with the `-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](#).²¹⁴ GCC compiler uses a different set of options: `-fprofile-generate` and `-fprofile-use` as described in the [documentation](#).²¹⁵

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 of lines of code, it is the only practical way to improve machine code layout.

Not all workloads benefit from PGO. Workloads with severe Frontend bottlenecks may see speedups of up to 30% from using PGO. However, compute-bound workloads, such as scientific computing, may not see any benefit at all.

While some 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 the huge runtime overhead of instrumented executables. Running an instrumented binary and collecting profiling data frequently incurs a 5-10x slow-

²¹⁴ PGO in Clang - <https://clang.llvm.org/docs/UsersManual.html#profiling-with-instrumentation>

²¹⁵ PGO in GCC - <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options>

down, 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 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 the program’s performance. 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 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`²¹⁶ that converts sampling data generated by Linux `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 a production environment for a longer time. Since this approach is based on hardware 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+).

In mid-2018, Meta open-sourced its binary optimization tool called `BOLT`²¹⁷ that works on already compiled binaries. It first disassembles the code, then it uses the profile information collected by a 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 block reordering, function splitting and reordering, and others. Similar to traditional PGO, primary candidates for BOLT optimizations are programs that suffer from instruction cache and ITLB misses. Since January 2022, BOLT has been 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

²¹⁶ AutoFDO - <https://github.com/google/autofdo>

²¹⁷ BOLT - <https://code.fb.com/data-infrastructure/accelerate-large-scale-applications-with-bolt/>

for better scaling and less memory consumption. Post-link optimizers such as BOLT and Propeller can be used in combination with traditional PGO (and Link-Time Optimizations) and often provide an additional 5-10% performance speedup. Such techniques open up new kinds of binary rewriting optimizations that are based on hardware telemetry.

11.8 Reducing ITLB Misses

Another important area of tuning Frontend efficiency is the virtual-to-physical address translation of memory addresses. Primarily those translations are served by the TLB (see Section 3.7.1), which caches the 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 the machine code of your application fits in 1MB you should not be affected by ITLB misses. The problem starts to appear when frequently executed parts of an application are scattered around in 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 Coffee Lake processor is ~7%, which means that 7% of cycles are spent handling ITLB misses: doing demanded 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).

The general idea of reducing ITLB pressure is to map the portions of the performance-critical code of an application onto 2MB (huge) pages. Usually, the entire code section of an application gets remapped for simplicity. The key requirement for that transformation to happen is to have the code section aligned on a 2MB boundary. When on Linux, this can be achieved in two different ways: relinking the binary with an additional linker option or remapping the code sections at runtime. Both options are showcased on the Easyperf²¹⁸ blog. To the best of my knowledge, it is not possible on Windows, so I will only show how to do it on Linux.

The first option can be achieved by linking the binary with the following options: `-Wl,-zcommon-page-size=2097152 -Wl,-zmax-page-size=2097152`. 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 the linker will be forced to insert up to 2MB of padded (wasted) bytes, bloating the binary even more. In the example with the Clang compiler, it increased the size of the binary from 111 MB to 114 MB. After relinking the binary, we set a special bit

²¹⁸ “Performance Benefits of Using Huge Pages for Code” - <https://easyperf.net/blog/2022/09/01/Utilizing-Huge-Pages-For-Code>.

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`²¹⁹ 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 a 2MB boundary and 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 the code section there. The reference implementation of that approach is implemented in the `iodlr`²²⁰ library. One option would be to call that functionality from your `main` function. Another option, which is simpler, is 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 B.

Mapping code sections onto 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.

Besides employing huge pages, standard techniques for optimizing I-cache performance can be used to improve 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, `l1vm-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 Case Study: Measuring Code Footprint

As I mentioned a couple of times in this chapter, code layout optimizations are most impactful on applications with large amounts of code. The best way to clarify the uncertainty about the size of the hot code in your program is to measure its *code*

²¹⁹ libhugetlbfs - <https://github.com/libhugetlbfs/libhugetlbfs/blob/master/HOWTO>.

²²⁰ iodlr library, Linux-only - <https://github.com/intel/iodlr>.

footprint, which is defined as the number of bytes/cache lines/pages with machine instructions the program touches during its execution.

A large code footprint by itself doesn't necessarily negatively impact performance. Code footprint is not a decisive metric, and it doesn't immediately tell you if there is a problem. Nevertheless, it has proven to be useful as an additional data point in performance analysis. In conjunction with TMA's `Frontend_Bound`, L1-instruction cache miss rate, and other metrics, it may strengthen the argument for investing time in optimizing the machine code layout of your application.

Currently, there are very few tools available that can reliably measure code footprint. In this case study, I will demonstrate `perf-tools`,²²¹ an open-source collection of profiling tools built on top of Linux `perf`. To estimate²²² code footprint, `perf-tools` leverages Intel's LBR (see Section 6.2), so it currently doesn't work on AMD- or ARM-based systems. Below is a sample command to collect the code footprint data:

```
$ perf-tools/do.py profile --profile-mask 100 -a <your benchmark>
```

--profile-mask 100 initiates LBR sampling, and -a enables you to specify a program to run. This command will collect code footprint along with various other data. I don't show the output of the tool, curious readers are welcome to study documentation and experiment with the tool.

I took a set of four benchmarks: Clang C++ compilation, Blender ray tracing, Cloverleaf hydrodynamics, and Stockfish chess engine; these workloads should be already familiar to you from Section 4.11 where we analyzed their performance characteristics. I ran them on an Intel's Alder Lake-based processor.²²³

Before we start looking at the results, let's spend some time on terminology. Different parts of a program's code may be exercised with different frequencies, so some parts will be hotter than others. The `perf-tools` package doesn't make this distinction and uses the term "non-cold code" to refer to code that was executed at least once. This is called *two-way splitting* since it splits the code into cold and non-cold parts. Other tools (e.g., Meta's HHVM) use *three-way splitting* and distinguish between hot, warm, and cold code with an adjustable threshold between warm and hot. In this section, we use the term "hot code" to refer to the non-cold code.

Results for each of the four benchmarks are presented in Table 11.1. The binary and `.text` sizes were obtained with a standard Linux `readelf` utility, while other metrics were collected with `perf-tools`. The `non-cold code footprint [KB]` metric is the number of kilobytes with machine instructions that a program touched at least once. The metric `non-cold code [4KB-pages]` tells us the number of non-cold 4KB-pages with machine instructions that a program touched at least once. Together they help us to understand how dense or sparse those non-cold memory locations are. It will become clear once we dig into the numbers. Finally, we also present Frontend Bound

²²¹ `perf-tools` - <https://github.com/aayasin/perf-tools>

²²² The code footprint data collected by `perf-tools` is not exact since it is based on sampling LBR records. Other tools like Intel's `sde -footprint`, unfortunately, don't provide code footprint. However, it is not hard to write a PIN-based tool yourself that will measure the exact code footprint.

²²³ It doesn't matter which machine you use for collecting code footprint as it depends on the program and input data, and not on the characteristics of a particular machine. As a sanity check, I ran it on a Skylake-based machine and got very similar results.

percentages, a metric that should be already familiar to you from Section 6.1 about TMA.

Table 11.1: Code footprint of the benchmarks used in the case study.

Metric	Clang17 compilation	Blender	CloverLeaf	Stockfish
Binary size [KB]	113844	223914	672	39583
.text size [KB]	67309	133009	598	238
non-cold code footprint [KB]	5042	313	104	99
non-cold code [4KB-pages]	6614	546	104	61
Frontend Bound [%]	52.3	29.4	5.3	25.8

Let's first look at the binary and `.text` sizes. CloverLeaf is a tiny application compared to Clang17 and Blender; Stockfish embeds the neural network file which accounts for the largest part of the binary, but its code section is relatively small; Clang17 and Blender have gigantic code bases. The `.text size` metric is the upper bound for our applications, i.e. we assume²²⁴ the code footprint should not exceed the `.text` size.

A few interesting observations can be made by analyzing the code footprint data. First, even though the Blender `.text` section is very large, less than 1% of Blender's code is non-cold: 313 KB out of 133 MB. So, just because a binary size is large, doesn't mean the application suffers from CPU Frontend bottlenecks. It's the amount of hot code that matters. For other benchmarks this ratio is higher: Clang17 7.5%, CloverLeaf 17.4%, Stockfish 41.6%. In absolute numbers, the Clang17 compilation touches an order of magnitude more bytes with machine instructions than the other three applications combined.

Second, let's examine the `non-cold code [4KB-pages]` row in the table. For Clang17, non-cold 5042 KB are spread over 6614 4KB pages, which gives us $5042 / (6614 * 4) = 19\%$ page utilization. This metric tells us how dense/sparse the hot parts of the code are. The closer each hot cache line is located to another hot cache line, the fewer pages are required to store the hot code. The higher the page utilization the better. Basic block placement and function reordering that we discussed earlier in this chapter are perfect examples of a transformation that improves page utilization. For other benchmarks, the percentages are: Blender 14%, CloverLeaf 25%, and Stockfish 41%.

Now that we quantified the code footprints of the four applications, it's tempting to think about the size of L1-instruction and L2 caches and whether the hot code fits or not. On my Alder Lake-based machine, the L1 I-cache is only 32 KB, which is not enough to fully cover any of the benchmarks that we've analyzed. But remember, at the beginning of this section we said that a large code footprint doesn't immediately point to a problem. Yes, a large codebase puts more pressure on the CPU Frontend, but an instruction access pattern is also crucial for performance. The same locality principles as for data accesses apply. That's why we accompanied it with the Frontend Bound metric from Topdown analysis.

²²⁴ It is not always true: an application itself may be tiny, but call into multiple other dynamically linked libraries, or it may make heavy use of kernel code.

For Clang17, the 5 MB of non-cold code causes a huge 52.3% Frontend Bound performance bottleneck: more than half of the cycles are wasted waiting for instructions. From all the presented benchmarks, it benefits the most from PGO-type optimizations. CloverLeaf doesn't suffer from inefficient instruction fetch; 75% of its branches are backward jumps, which suggests that those could be relatively small loops executed over and over again. Stockfish, while having roughly the same non-cold code footprint as CloverLeaf, poses a far greater challenge for the CPU Frontend (25.8%). It has a lot more indirect jumps and function calls. Finally, Blender has even more indirect jumps and calls than Stockfish.

I stop my analysis at this point as further investigations are outside the scope of this case study. For readers who are interested in continuing the analysis, I suggest drilling down into the Frontend Bound category according to the TMA methodology and looking at metrics such as `ICache_Misses`, `ITLB_Misses`, `DSB coverage`, and others.

Another useful tool to study the code footprint is `llvm-bolt-heatmap`²²⁵, which is a part of LLVM's BOLT project. This tool can produce code heatmaps that give a fine-grained understanding of the code layout in your application. It is primarily used to evaluate the original layout of hot code and confirm that the optimized layout is more compact.

Questions and Exercises

1. Solve `perf-ninja:::pgo` and `perf-ninja:::lto` lab assignments.
2. Experiment with using Huge Pages for the 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`, and CPU performance counters that measure ITLB loads and misses.
3. Run the application that you're working with daily. Apply PGO, LLVM-bolt, or Propeller and check the result. Compare "before" and "after" profiles to understand where the speedups are coming from.

Chapter Summary

A summary of CPU Frontend optimizations is presented in Table 11.2.

Table 11.2: Summary of CPU Frontend 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

²²⁵ `llvm-bolt-heatmap` - <https://github.com/llvm/llvm-project/blob/main/bolt/docs/Heatmaps.md>

Transform	How transformed?	Why helps?	Works best for	Done by
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 overlooked. CPU Frontend 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 the best option to use LTO, PGO, BOLT, and similar tools to improve the code layout if you can come up with a set of typical use cases for your application. For large applications, it is the only practical option.

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, but still important enough to find their place in this book.

We will start by discussing how to introduce CPU-specific optimizations in your code. Then we will cover some of the corner case situations that can have a measurable impact on the performance of your application. Next, we move on to exploring low-latency tuning techniques, which are essential for applications that require fast response times. Finally, we will provide advice for tuning the system settings.

12.1 CPU-Specific Optimizations

Optimizing software for a specific CPU microarchitecture involves tailoring your code to leverage the strengths and mitigate the weaknesses of that microarchitecture. It is easier to do when you know the exact target CPU for your application. However, most applications run on a wide range of CPUs. Optimizing the performance of a cross-platform application with very high-speed requirements can be challenging since platforms from different vendors have different designs and implementations. Nevertheless, it is possible to write code that performs reasonably well on CPUs from different vendors, while providing a fine-tuned version for a specific microarchitecture.

The major differences between x86 (considered as CISC) and RISC ISAs, such as ARM and RISC-V, are summarized below:

- x86 instructions are variable-length, while ARM and RISC-V instructions are fixed-length. This makes decoding x86 instructions more complex.
- x86 ISA has many addressing modes, while ARM and RISC-V have few addressing modes. Operands in ARM and RISC-V instructions are either registers or immediate values, while x86 instruction inputs can also come from memory. This bloats the number of x86 instructions but also allows for more powerful single instructions. For instance, ARM requires loading a memory location first, then performing the operation; x86 can do both in one instruction.

In addition to this, there are a few other differences that you should consider when optimizing for a specific microarchitecture. As of 2024, the most recent x86-64 ISA has 16 architectural general-purpose registers, while the latest ARMv8 and RV64 require a CPU to provide 32 general-purpose registers. Extra architectural registers reduce register spilling and hence reduce the number of loads/stores. Intel has announced a new extension called APX²²⁶ that will increase the number of registers to 32.

There is also a difference in the memory page size between x86 and ARM. The default page size for x86 platforms is 4 KB, while most ARM systems (for example, Apple MacBooks) use a 16 KB page size, although both platforms support larger page sizes

²²⁶ Intel APX - <https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-performance-extensions-apx.html>

(see Section 3.7.2, and Section 8.4). All these differences can affect the performance of your application when they become a bottleneck.

Although ISA differences *may* have a tangible impact on the performance of a specific application, numerous studies show that on average, differences between the two most popular ISAs, namely x86 and ARM, don't have a measurable performance impact. Throughout this book, I carefully avoided advertisements of any products (e.g., Intel vs. AMD vs. Apple) and any religious ISA debates (x86 vs. ARM vs. RISC-V).²²⁷ Below are some references that I hope will close the debate:

- Performance or energy consumption differences are not generated by ISA differences, but rather by microarchitecture implementation. [Blem et al., 2013]
- ISA doesn't have a large effect on the number and type of executed instructions. [Weaver & McIntosh-Smith, 2023] [Blem et al., 2013]
- CISC code is not denser than RISC code. [Geelnard, 2022]
- ISA overheads can be effectively mitigated by microarchitecture implementation. For example, μ op cache minimizes decoding overheads; instruction cache minimizes code density impact. [Blem et al., 2013] [ChipsAndCheese, 2024]

Nevertheless, this doesn't remove the value of architecture-specific optimizations. In this section, we will discuss how to optimize for a particular platform. We will cover ISA extensions, the CPU dispatch technique, and discuss how to reason about instruction latencies and throughput.

12.1.1 ISA Extensions

ISA evolution has been continuous. It has focused on accelerating specialized workloads, such as cryptography, AI, multimedia, and others. Utilizing ISA extensions often results in substantial performance improvements. Developers keep finding smart ways to leverage these extensions in general-purpose applications. So, even if you're outside of one of these highly specialized domains, you might still benefit from using ISA extensions.

It's not possible to learn about all specific instructions. But I suggest you familiarize yourself with major ISA extensions available on your target platform. For example, if you are developing an AI application that uses `fp16` (16-bit half-precision floating-point) data types, and you target one of the modern ARM processors, make sure that your program's machine code contains corresponding `fp16` ISA extensions. If you're developing encryption/decryption software, check if it utilizes crypto extensions of your target ISA. And so on.

Here is a list of some notable x86 ISA extensions:

- SSE/AVX/AVX2: provide SIMD instructions for floating-point and integer operations.
- AVX512: extends AVX2 with 512-bit registers and many new instructions.
- AVX512_FP16/AVX512_BF16: add support for 16-bit half-precision and `Bfloat16` floating-point values.

²²⁷The debate also isn't interesting because after μ ops conversion, x86 becomes a RISC-style micro-architecture. Complex instructions get broken down into simpler instructions.

- AES/SHA: provide instructions for AES encryption, decryption, and SHA hashing.
- BMI/BMI2: provide instructions for bit manipulation.
- AVX_VNNI/AVX512_VNNI: Vector Neural Network Instructions for accelerating deep learning workloads.
- AMX: Advanced Matrix Extensions for accelerating matrix multiplication.

Here is a list of some notable ARM ISA extensions:

- Advanced SIMD: also known as NEON, provides arithmetic SIMD instructions.
- Cryptographic Instructions: provide instructions for encryption, hashing, and checksumming.
- FP16/BF16: provide 16-bit half-precision and Bfloat16 floating-point instructions.
- UDOT/SDOT: support for dot product instructions for accelerating machine learning workloads.
- SVE: enables scalable vector length instructions.
- SME: Scalable Matrix Extension for accelerating matrix multiplication.

When compiling your applications, make sure to enable the necessary compiler flags to activate required ISA extensions. On GCC and Clang compilers use the `-march` option. For example, `-march=native` will activate ISA features of your host system, i.e., on which you run the compilation. Or you can include a specific version of ISA, e.g., `-march=armv8.6-a`. On the MSVC compiler, use the `/arch` option, e.g., `/arch:AVX2`.

I do not recommend using `-march=native` for production builds, because code generation will depend on which machine you're building your code. Many CI/CD systems have old machines. Building software on one of these machines with `-march=native` may lead to suboptimal performance when the application is run on a newer machine. Instead, use `-march` with a specific microarchitecture that you want to target.

12.1.2 CPU Dispatch

When you want to provide a fast path for a specific microarchitecture while keeping a generic implementation for other platforms, you can use *CPU dispatching*. It is a technique that allows your program to detect which features your processor has, and based on that decide which version of the code to execute. It enables you to introduce platform-specific optimizations in a single codebase. As a rule of thumb, it is better to start with a generic implementation and then introduce microarchitecture-specific optimizations progressively, ensuring there is a fallback for architectures that do not have the required features. For example:

```
if (__builtin_cpu_supports ("avx512f")) {
    avx512_impl();
} else {
    generic_impl();
}
```

This demonstrates the use of built-in functions that are available in GCC and Clang compilers. Besides detecting supported ISA extensions, there is a `__builtin_cpu_is` function to detect an exact processor model. A compiler-agnostic way of writing CPU dispatch is to use the `CPUID` instruction (x86-only), `getauxval(AT_HWCAP)` Linux system call, or `sysctlbyname` on macOS.

You would typically see CPU dispatching constructs used to optimize only specific parts of the code, e.g., hot function or loop. Very often, these platform-specific implementations are written with compiler intrinsics (see Section 9.5) to generate desired instructions.

Even though CPU dispatching is a runtime check, its overhead is not high. You can identify hardware capabilities at startup once and save it in some variable, so at runtime, it becomes just a single branch, which is well-predicted. Perhaps a bigger concern about CPU dispatching is the maintenance cost. Every new specialized branch requires fine-tuning and validation.

12.1.3 Instruction Latencies and Throughput

Besides ISA extensions, it's worth learning about the number and type of execution units in your processor (e.g., the number of loads, stores, divisions, and multiplications a processor can issue every cycle). For most processors, this information is published by CPU vendors in corresponding technical manuals. However, information about latencies and throughput of specific instructions is not usually disclosed. Nevertheless, people have benchmarked individual instructions, which can be accessed online. For the latest Intel and AMD CPUs, latency, throughput, port usage, and the number of μ ops for an instruction can be found at the uops.info²²⁸ website. For Apple processors, similar data is accessible in [Apple, 2024, Appendix A].²²⁹ Along with instruction latencies and throughput, developers have reverse-engineered other aspects of a microarchitecture such as the size of branch prediction history buffers, reorder buffer capacity, size of load/store buffers, and others.

Be very careful about making conclusions just on the instruction latency and throughput numbers. In many cases, instruction latencies are hidden by the out-of-order execution engine, and it may not matter if an instruction has a latency of 4 or 8 cycles. If it doesn't block forward progress, such instruction will be handled "in the background" without harming performance. However, the latency of an instruction becomes important when it stands on a critical dependency chain because it delays the execution of dependent operations.

In contrast, if you have a loop that performs a lot of *independent* operations, you should focus on instruction throughput rather than latency. When operations are independent, they can be processed in parallel. In such a scenario, the critical factor is how many operations of a certain type can be executed per cycle, or *execution throughput*. There are also "in-between" scenarios, where both instruction latency and throughput may affect performance.

When you analyze machine code for one of your hot loops, you may find that multiple instructions are assigned to the same execution port. This situation is known as *execution port contention*. So the challenge is to find ways of substituting some of these instructions with the ones that are not assigned to the critical port. For example on Intel processors, if you're heavily bottlenecked on `port5`, then you may find that two instructions on `port0` are better than one instruction on `port5`. Often it is not

²²⁸ x86 instruction latency and throughput - <https://uops.info/table.html>

²²⁹ Also, there are instruction throughput and latency data collected via reverse-engineering experiments, such as in <https://dougallj.github.io/applecpu/firestorm-simd.html>. Since this is an unofficial source of data, you should take it with a grain of salt.

an easy task and it requires deep ISA and microarchitecture knowledge. When in doubt, seek help on specialized forums. Also, keep in mind that some of these things may change in future CPU generations, so consider using CPU dispatch to isolate the effect of your code changes.

Case Study: When FMA Instructions Hurt Performance

In Section 5.6.1, we looked at one example, of when the throughput of FMA instructions becomes critical. Now let's take a look at another example, involving FMA latency. In Listing 12.1 on the left, we have the `sqSum` function which computes a sum of every element squared. On the right, we present the corresponding machine code generated by Clang-18 when compiled with `-O3 -march=core-avx2`. Notice, that we didn't use `-ffast-math`, perhaps because we want to maintain bit-exact results over multiple platforms. That's why the code was not autovectorized by the compiler.

Listing 12.1 FMA latency

<pre> 1 float sqSum(float *a, int N) { 2 float sum = 0; 3 for (int i = 0; i < N; i++) { 4 sum += a[i] * a[i]; 5 } 6 }</pre>	<pre> .loop: vmovss xmm1, dword ptr [rcx + 4*rdx] vfmadd231ss xmm0, xmm1, xmm1 inc rdx cmp rax, rdx jne .loop</pre>
--	---

On each iteration of the loop, we have two operations: calculate the squared value of `a[i]` and accumulate the product in the `sum` variable. If you look closer, you may notice that multiplications are independent of each other, so they can be executed in parallel. The generated machine code (on the right) uses fused multiply-add (FMA) to perform both operations with a single instruction. The problem here is that by using FMAs, the compiler has included multiplication into the critical dependency chain of the loop.

The `vfmadd231ss` instruction computes the squared value of `a[i]` (in `xmm1`) and then accumulates the result in `xmm0`. There is a data dependency over `xmm0`: a processor cannot issue a new `vfmadd231ss` instruction until the previous one has finished since `xmm0` is both an input and an output of `vfmadd231ss`. Even though multiplication parts of FMA do not depend on each other, these instructions need to wait until all inputs become available. The performance of this loop is bound by FMA latency, which in Intel's Alder Lake is 4 cycles.

In this case, fusing multiplication and addition hurts performance. We would be better off with two separate instructions. The `nanobench` experiment below proves that:

<pre># ran on Intel Core i7-1260P (Alder Lake) \$ sudo ./kernel-nanoBench.sh -f -basic -loop 100 -unroll 1000 -warm_up_count 10 -asm " vmovss xmm1, dword ptr [R14]; vfmadd231ss xmm0, xmm1, xmm1;"</pre> <p>Instructions retired: 2.00 Core cycles: 4.00</p>	<pre>\$ sudo ./kernel-nanoBench.sh -f -basic -loop 100 -unroll 1000 -warm_up_count 10 -asm " vmovss xmm1, dword ptr [R14]; vmulss xmm1, xmm1, xmm1; vaddss xmm0, xmm0, xmm1;"</pre> <p>Instructions retired: 3.00 Core cycles: 2.00</p>
---	---

The version on the left runs in four cycles per iteration, which corresponds to the FMA latency. However, on the right-hand side, `vmulss` instructions do not depend on each other, so they can be run in parallel. Still, there is a loop carry dependency over `xmm0` in the `vaddss` instruction (FADD). But the latency of FADD is only two cycles, this is why the version on the right runs in just two cycles per iteration. The latency and throughput characteristics for other processors may vary.²³⁰

From this experiment, we know that if the compiler would not have decided to fuse multiplication and addition into a single instruction, it would result in two times better performance for this loop. This only became clear once we examined the loop dependencies and compared the latencies of FMA and FADD instructions. Since Clang 18, you can prevent generating FMA instructions within a scope by using `#pragma clang fp contract(off)`.²³¹

12.2 Microarchitecture-Specific Performance Issues

In this section, we will discuss some common microarchitecture-specific issues that affect the majority of modern processors. I call them microarchitecture-specific because they are caused by the way a particular microarchitecture feature is implemented. These issues are very specific and do not frequently appear as a major performance bottleneck. Typically, they are diluted among other more significant performance problems. Thus, these microarchitecture-specific performance issues are considered corner cases and are less known than the other issues that we already discussed in the book. Nevertheless, they can cause very undesirable performance penalties. Note that the impact of a particular problem can be more/less pronounced on one platform than another. Also, keep in mind that the list of microarchitecture-specific issues covered below is not exhaustive.

12.2.1 Memory Order Violations

I introduced the concept of memory ordering in Section 3.8.3. Memory reordering is a crucial aspect of modern CPUs, as it enables them to execute memory requests in parallel and out-of-order. The key element in load/store reordering is memory disambiguation, which predicts if it is safe to let loads go ahead of earlier stores. Since memory disambiguation is speculative, it can lead to performance issues if not handled properly.

Consider an example in Listing 12.2, on the left. This code snippet calculates a histogram of an 8-bit grayscale image, i.e., how many times a certain color appears in the image. Besides countless other places, this code can be found in Otsu's thresholding algorithm²³² which is used to convert a grayscale image to a binary image. Since the input image is 8-bit grayscale, there are only 256 different colors.

For each pixel on an image, you need to read the current histogram count of the color of the pixel, increment it, and store it back. This is a classic read-modify-write dependency through the memory. Imagine we have the following consecutive

²³⁰ The two versions will produce slightly different results due to the different rounding of floating-point values.

²³¹ LLVM extensions to specify floating-point flags - <https://clang.llvm.org/docs/LanguageExtensions.html#extensions-to-specify-floating-point-flags>

²³² Otsu's thresholding method - https://en.wikipedia.org/wiki/Otsu%27s_method

pixels in the image: `pixels = [0xFF, 0xFF, 0x00, 0xFF, ...]` and so on. The loaded value of the histogram count for `pixels[1]` comes from the result of the previous iteration (`pixels[0]`). The histogram count for `pixels[2]` comes from memory; it is independent and can be reordered. But then again, the histogram count for `pixels[3]` is dependent on the result of processing `pixels[1]`, and so on. Iterations 0, 1, and 3 are dependent and cannot be reordered.

Listing 12.2 Memory Order Violation Example.

```
std::array<uint32_t, 256> hist;
hist.fill(0);
int N = width * height;
for (int i = 0; i < N; ++i)      =>
    hist[image[i]]++;
                                         std::array<uint32_t, 256> hist1;
                                         std::array<uint32_t, 256> hist2;
                                         hist1.fill(0);
                                         hist2.fill(0);
                                         int N = width * height;
                                         int i = 0;
                                         for (; i + 1 < N; i += 2) {
                                             hist1[image[i+0]]++;
                                             hist2[image[i+1]]++;
                                         }
                                         // remainder
                                         for (; i < N; ++i)
                                             hist1[image[i]]++;
                                         // combine partial histograms
                                         for (int i = 0; i < hist1.size(); ++i)
                                             hist1[i] += hist2[i];
```

Recall from Section 3.8.3 that the processor doesn't necessarily know about a potential store-to-load forwarding, so it has to make a prediction. If it correctly predicts a memory order violation between two updates of color 0xFF, then these accesses will be serialized. The performance will not be great, but it is the best we could hope for with the initial code. On the contrary, if the processor predicts that there is no memory order violation, it will speculatively let the two updates run in parallel. Later it will recognize the mistake, flush the pipeline, and re-execute the youngest of the two updates. This is very hurtful for performance.

Performance will greatly depend on the color patterns of the input image. Images with long sequences of pixels with the same color will have worse performance than images where colors don't repeat often. The performance of the initial version will be good as long as the distance between two pixels of the same color is long enough. The phrase “long enough” in this context is determined by the size of the out-of-order instruction window. Repeating read-modify-writes of the same color may trigger ordering violations if they occur within a few loop iterations of each other, but not if they occur more than a hundred loop iterations apart.

A cure for the memory order violation problem is shown in Listing 12.2, on the right. As you can see, I duplicated the histogram, and now the processing of pixels alternates between two partial histograms. In the end, we combine the two partial histograms to get a final result. This new version with two partial histograms is still prone to potentially problematic patterns, such as 0xFF 0x00 0xFF 0x00 0xFF ... However, with this change, the original worst-case scenario (e.g., 0xFF 0xFF 0xFF ...) will run twice as fast as before. It may be beneficial to create four or eight partial histograms depending on the color pattern of input images. This exact code is featured in the

`mem_orderViolation_1`²³³ lab assignment of the Performance Ninja course, so feel free to experiment.

On a small set of input images, I observed from 10% to 50% speedup on various platforms. It is worth mentioning that the version on the right consumes 1 KB of additional memory, which may not be huge in this case but is something to watch out for.

12.2.2 Misaligned Memory Accesses

A variable is accessed most efficiently if it is stored at a memory address that is divisible by the size of the variable. For example, an `int` requires a 4-byte alignment, meaning its address should be a multiple of 4. In C++, it is called *natural alignment*, which occurs by default for fundamental data types, such as integer, float, or double. When you declare variables of these types, the compiler ensures that they are stored in memory at addresses that are multiples of their size. In contrast, arrays, structs, and classes may require special alignment as you'll learn in this section.

A typical case where data alignment is important is SIMD code, where loads and stores access large chunks of data with a single operation. In most processors, the L1 cache is designed to be able to read/write data at any alignment. Generally, even if a load/store is misaligned but does not cross the cache line boundary, it won't have any performance penalty.

However, when a load or store crosses the cache line boundary, such access requires two cache line reads (*split load/store*). It requires using a *split register*, which keeps the two parts and once both parts are fetched, they are combined into a single register. The number of split registers is limited. When executed sporadically, split accesses complete without any observable performance impact on overall execution. However, if that happens frequently, misaligned memory accesses will suffer delays.

A memory address is said to be *aligned* if it is a multiple of a specific size. For example, when a 16-byte object is aligned on the 64-byte boundary, the low 6 bits of its address are zero. Otherwise, when a 16-byte object crosses the 64-byte boundary, it is said to be *misaligned*. In the literature, you can also encounter the term *split load/store* to describe such a situation. Split loads/stores may incur performance penalties if many of them in a row consume all available split registers. Intel's TMA methodology tracks this with the `Memory_Bound` → `L1_Bound` → `Split Loads` metric.

For instance, AVX2 memory operations can access up to 32 bytes. If an array starts at offset `0x30` (48 bytes), the first AVX2 load will fetch data from `0x30` to `0x4F`, the second load will fetch data from `0x50` to `0x6F`, and so on. The first load crosses the cache line boundary (`0x40`). In fact, every second load will cross the cache line boundary which may slow down the execution. Figure 12.1 illustrates this. Pushing the data forward by 16 bytes would align the array to the cache line boundary and eliminate split loads. Listing 12.3 shows how to fix this example using the C++11 `alignas` keyword.

When it comes to dynamic allocations, C++17 made it much easier. Operator `new`

²³³ Performance Ninja lab assignment: Memory Order Violation - https://github.com/dendibakh/perf-ninja/tree/main/labs/memory_bound/mem_orderViolation_1

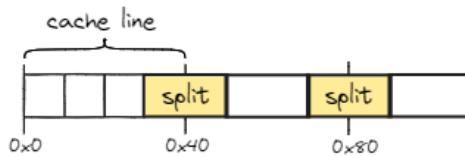


Figure 12.1: AVX2 loads in a misaligned array. Every second load crosses the cache line boundary.

Listing 12.3 Aligning data using the “alignas” keyword.

```
// Array of 16-bit integers aligned at a 64-byte boundary
#define CACHELINE_ALIGN alignas(64)
CACHELINE_ALIGN int16_t a[N];
```

now takes an additional argument, which you can use to control the alignment of dynamically allocated memory. When using standard containers, such as `std::vector`, you can define a custom allocator. Listing 12.4 shows a minimal example of a custom allocator that aligns the memory buffer at the cache line boundary.

Listing 12.4 Defining std::vector aligned at the cache line boundary.

```
// Returns aligned pointers when allocations are requested.
template <typename T>
class CacheLineAlignedAllocator {
public:
    using value_type = T;
    static std::align_val_t constexpr ALIGNMENT{64};
    [[nodiscard]] T* allocate(std::size_t N) {
        return reinterpret_cast<T*>(::operator new[](N * sizeof(T), ALIGNMENT));
    }
    void deallocate(T* allocPtr, [[maybe_unused]] std::size_t N) {
        ::operator delete[](allocPtr, ALIGNMENT);
    }
};
template<typename T>
using AlignedVector = std::vector<T, CacheLineAlignedAllocator<T> >;
```

To demonstrate the effect of misaligned memory accesses, I created the `mem_alignment_1`²³⁴ lab assignment in the Performance Ninja online course. It features a very simple matrix multiplication example, where the initial version doesn't take any care of the alignment of the matrices. The assignment asks to align the matrices to the cache line boundary and measure the performance difference. Feel free to experiment with the code and measure the effect on your platform.

The first step to mitigate split loads/stores in this assignment is to align the starting offset of a matrix. The operating system might allocate memory for the matrix such that it is already aligned to the cache line boundary. However, you should not rely on this behavior, as it is not guaranteed. A simple way to fix this is to use `AlignedVector` from Listing 12.4 to allocate memory for the matrices.

²³⁴ Performance Ninja lab assignment: Memory Alignment - https://github.com/dendibakh/perf-ninja/tree/main/labs/memory_bound/mem_alignment_1

However, it's not enough to only align the starting offset of a matrix. Consider an example of a 9x9 matrix of `float` values shown in Figure 12.2. If a cache line is 64 bytes, it can store 16 `float` values. When using AVX2 instructions, the program will load/store 8 elements (256 bits) at a time. In each row, the first eight elements will be processed in a SIMD way, while the last element will be processed in a scalar way by the loop remainder. The second vector load/store (elements 10-17) crosses the cache line boundary as many other subsequent vector loads/stores. The problem highlighted in Figure 12.2 affects any matrix with the number of columns that is not a multiple of 8 (for AVX2 vectorization). The SSE and ARM Neon vectorization requires 16-byte alignment; AVX-512 requires 64-byte alignment.

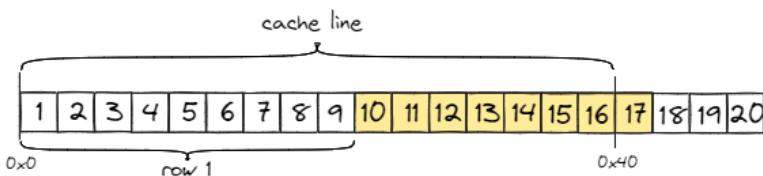


Figure 12.2: Split loads/stores inside a 9x9 matrix when using AVX2 vectorization. The split memory access is highlighted in yellow.

So, in addition to aligning the starting offset, each row of the matrix should be aligned as well. For example in Figure 12.2, it can be achieved by inserting seven dummy columns into the matrix, effectively making it a 9x16 matrix. This will align the second row (elements 10-18) at the offset 0x40. Similarly, all the other rows will be aligned as well. The dummy columns will not be processed by the algorithm, but they will ensure that the actual data is aligned at the cache line boundary. In my testing, the performance impact of this change was up to 30%, depending on the matrix size and platform configuration.

Alignment and padding cause holes with unused bytes, which potentially decreases memory bandwidth utilization. For small matrices, like our 9x9 matrix, padding will cause almost half of each row to be unused. However, for large matrices, like 1025x1025 the impact of padding is not that big. Nevertheless, for some algorithms, e.g., in AI, memory bandwidth can be a bigger concern. Use these techniques with care and always measure to see if the performance gain from alignment is worth the cost of unused bytes.

Accesses that cross a 4 KB boundary introduce more complications because virtual to physical address translations are usually handled in 4 KB pages. Handling such access would require accessing two TLB entries as well. Unless a TLB supports multiple lookups per cycle, such loads can cause a significant slowdown.

12.2.3 Cache Aliasing

There are specific data access patterns that may cause unpleasant performance issues. These corner cases are tightly connected with cache organization, e.g., the number of sets and ways in the cache. We discussed cache organization in Section 3.6.1, in case you want to revisit it. The placement of a memory location in the cache is determined

by its address. Based on the address bits, the cache controller does set selection, i.e., it determines the set to which a cache line with the fetched memory location will go.

If two memory locations map to the same set, they will compete for the limited number of available slots (ways) in a set. When a program repeatedly accesses memory locations that map to the same set, they will be constantly evicting each other. This may cause saturation of one set in the cache and underutilization of other sets. This is known as *cache aliasing*, though you may find people use the terms *cache contention*, *cache conflicts*, or *cache thrashing* to describe this effect.

A simple example of cache aliasing can be observed in matrix transposition as explained in detail in [Fog, 2023b, section 9.10 Cache contentions in large data structures]. I encourage readers to study this manual to learn more about why it happens. I repeated the experiment on a few modern processors and confirmed that it remains a relevant issue. Figure 12.3 shows the performance of transposing matrices of 32-bit floating-point values on Intel's 12th-gen core i7-1260P processor.

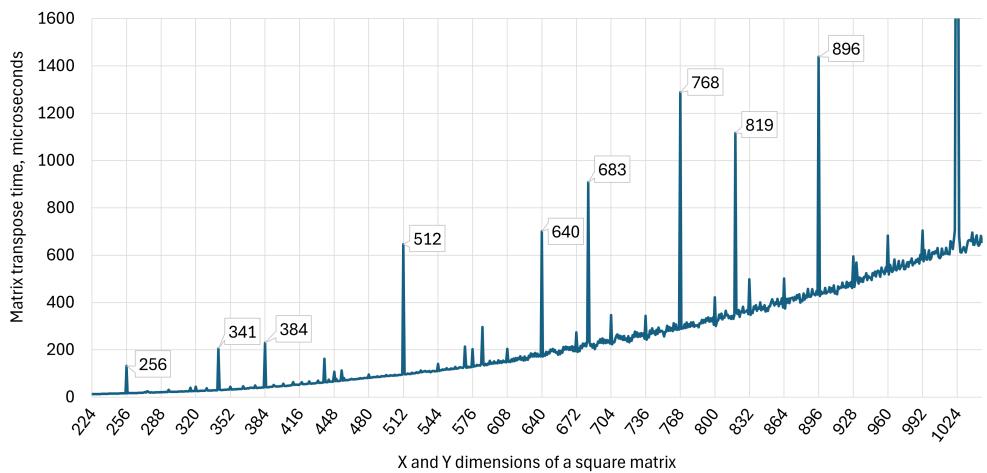


Figure 12.3: Cache aliasing effects observed in matrix transposition ran on Intel's 12th-gen processor. Matrix sizes that are powers of two or multiples of 128 cause more than 10x performance drop.

There are several spikes in the chart, which correspond to the matrix sizes that cause cache aliasing. Performance drops significantly when the matrix size is a power of two (e.g., 256, 512) or is a multiple of 128 (e.g., 384, 640, 768, 896).²³⁵ This happens because memory locations that belong to the same column are mapped to the same set in the L1D and L2 caches. These memory locations compete for the limited number of ways in the set which causes the same cache line to be reloaded many times before every element on this line is processed.

On Intel's processors, this issue can be diagnosed with the help of the `L1D.REPLACEMENT` performance event, which counts L1 cache line replacements. For instance, there are 17 times more cache line replacements for the matrix size 256×256 than for the size

²³⁵ Also, there are a few spikes at the sizes 341, 683, and 819. Supposedly, these sizes suffer from the same cache aliasing effect, but I haven't investigated them further.

255x255. I tested all sizes from 64x64 up to 10,000x10,000 and found that the pattern repeats very consistently. I also ran the same experiment on an Intel Skylake-based processor as well as an Apple M1 chip and confirmed that these chips are prone to cache aliasing effects.

To mitigate cache aliasing, you can use cache blocking as we discussed in Section 9.3.2. The idea is to process the matrix in smaller blocks that fit into the cache. That way you will avoid cache line eviction since there will be enough space in the cache. Another way to solve this is to pad the matrix with extra columns, e.g., instead of a 256x256 matrix, you would allocate a 256x264 matrix; in a similar way we did in the previous section. But be careful not to run into misaligned memory access issues.

12.2.4 Slow Floating-Point Arithmetic

Some applications that do extensive computations with floating-point (FP) values are prone to one very subtle issue that can cause performance slowdown. This issue arises when an application hits a *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 754,²³⁶ a subnormal is a non-zero number with an exponent smaller than the smallest normal number.²³⁷ Listing 12.5 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 others.

Listing 12.5 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 can come 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 slower than similar calculations on normal numbers and can run 10 times slower or more. For

²³⁶ IEEE Standard 754 - <https://ieeexplore.ieee.org/document/8766229>

²³⁷ Subnormal number - https://en.wikipedia.org/wiki/Subnormal_number

instance, Intel processors currently handle operations on subnormals with a microcode *assist*. When a processor recognizes a subnormal FP value, a Microcode Sequencer (MSROM) will provide the necessary microoperations (μ ops) to compute the result.

In many cases, subnormal values are generated naturally by the algorithm and thus are unavoidable. Most processors give the option to flush subnormal values to zero and not generate subnormals in the first place. Developers of performance-critical applications perhaps would prefer to have slightly less accurate results than slowing down the code.

Suppose your application doesn't need subnormal values, how do you detect and mitigate associated costs? While you can use runtime checks as shown in Listing 12.5, inserting them all over the codebase is not practical. There is a 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 or produced. The TMA methodology classifies such bottlenecks under the `Retiring` category, and yes, this is another situation when a high `Retiring` doesn't mean a good thing.

Once you confirm 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 costly handling of subnormal values 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 controlled 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 12.6 shows an 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 12.6 Enabling FTZ and DAZ modes manually

```
unsigned FTZ = 0x8000;
unsigned DAZ = 0x0040;
unsigned MXCSR = _mm_getcsr();
_mm_setcsr(MXCSR | FTZ | DAZ);
```

Keep in mind, that 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. I have observed a 3%-5% performance penalty on some production floating-point applications that were using subnormal values.

12.3 Low Latency Tuning Techniques

So far we have discussed a variety of software optimizations that aim at improving the 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 run as fast as possible. When you work in the HFT industry, every microsecond and nanosecond counts 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 the 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 costs 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 immediately.

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 is 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 commitment 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 operating system fulfills its duties. The very first write to a newly allocated page triggers a minor page fault, a hardware interrupt that is handled by the OS. The 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 12.7 shows a dump of the `top` command with the top-10 processes while compiling a large C++ project. The additional `vMn` column shows the number of minor page faults that occurred during the last 3 seconds.

Another way of detecting runtime minor page faults involves attaching to the running process with `perf stat -e page-faults`.

Listing 12.7 A dump of Linux top command with additional vMn field while compiling a 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

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 a `size` amount of memory on the heap as usual. However, immediately after that, it steps by and writes to the first byte of 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 12.8 with a more comprehensive approach to tuning the glibc allocator in conjunction with `mlock/mlockall` syscalls (taken from the “Real-time Linux Wiki”²³⁸).

The code in Listing 12.8 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.
- 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, that the latter hinders the glibc allocator’s multithreaded scalability feature.

²³⁸ The Linux Foundation Wiki: Memory for Real-time Applications - <https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/memory>

Listing 12.8 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);
```

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 that 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 the `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 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 an order to the exchange. In the aforementioned workload, the code paths involved with reading the market data are most commonly executed, while the code paths for executing an order are 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 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 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 to lose the race. This is where the technique of *cache warming* is 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 I will not provide an example implementation, you can get a taste of it in a [CppCon 2018 lightning talk²³⁹](#).

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. Unlike the case with MESI-based protocols and per-core CPU caches (i.e., L1, L2, and LLC), the hardware itself is not maintaining core-to-core TLB coherency. Therefore, this task must be performed in software by the operating system.

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` may invalidate TLB entries. These updates must be communicated among the constituent threads of a process. The kernel performs this job using 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.

Though a developer may avoid explicitly using these syscalls in his/her code, TLB shootdowns may still erupt from external sources – e.g., memory allocation 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

²³⁹ Cache Warming technique - <https://www.youtube.com/watch?v=XzRxikGgaHI>

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 12.9 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 behavior was a Linux kernel feature called Automatic NUMA Balancing, which can be easily disarmed with `sysctl -w numa_balancing=0`.

Listing 12.9 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 their duties, which require page table updates and, thus, TLB shootdowns.

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 that induce TLB shootdowns as a consequence of its function, such as Transparent Huge Pages and Automatic NUMA Balancing. For a more nuanced discussion on TLB shootdowns, along with their detection and prevention, read a related article²⁴⁰ 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 AVX512 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-premises and in the cloud) exhibit heavy core throttling/downclocking when executing heavy AVX512 instructions. If your compiler produces these instructions without your explicit knowledge or consent, you may experience unexplained latency anomalies during application runtime.

²⁴⁰ JabPerf blog: TLB Shootdowns - <https://www.jabperf.com/how-to-deter-or-disarm-tlb-shootdowns/>

For this specific case, if heavy AVX512 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 System Tuning

After 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 interrupt that halts the entire system. Such interrupt might run for up to tens to hundreds of milliseconds at a time.

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 Teams (Ops), which handle such sort of issues. It is in our interest to provide them with recommendations on how to set up the system to get the best performance out of our application.

There are many things to tune in a 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 guidelines²⁴¹. There, you will find tips for eliminating or significantly minimizing cache-disrupting interrupts from 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.

Most out-of-the-box platforms are configured for optimal throughput while saving power when 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 a “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 states²⁴² to keep it ready to react immediately. Another interesting technique is called Cache Locking,²⁴³ where portions of the CPU cache are reserved for a particular set of data. It helps to streamline the memory latencies within an application.

²⁴¹ Red Hat low latency tuning guidelines - <https://access.redhat.com/sites/default/files/attachments/201501-perf-brief-low-latency-tuning-rhel7-v2.1.pdf>

²⁴² Power Management States: P-States, C-States - <https://software.intel.com/content/www/us/en/develop/articles/power-management-states-p-states-c-states-and-package-c-states.html>

²⁴³ 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 mmaping: <https://events19.linuxfoundation.org/wp-content/uploads/2017/11/Introducing-g-Cache-Pseudo-Locking-to-Reduce-Memory-Access-Latency-Reinette-Chatre-Intel.pdf>

A more extreme shot at boosting performance is to overclock the CPU. Overclocking is a process of running the CPU at a higher frequency than it was designed for. It is a risky operation, as it can void the warranty and potentially damage the CPU. Overclocking is not suited for production environments and is usually done by enthusiasts who are willing to take the risk for the sake of performance. To overclock a CPU, you need to have the right parts, mainly a motherboard that supports overclocking, a CPU that has unlocked clock frequency, and a cooling system that can handle the increased heat output. At the beginning of 2024, overclocking experts crossed the 9 GHz barrier on a widely available CPU.²⁴⁴

It is helpful to understand performance bottlenecks in your application to tune the right settings. Scalability studies can help you to determine the sensitivity of your application to various system settings. For example, you may find out that your application doesn't scale with the number of cores (see Section 13.2.1), or that it is limited by the memory latency. Using this information, you can make educated decisions about tuning the system settings or perhaps buying new hardware components for your computing systems. In the next case study, we will show how you can determine whether an application is sensitive to the size of the last-level cache (LLC).

12.5 Case Study: Sensitivity to Last Level Cache Size

In this case study, we run the same set of applications multiple times with varying LLC sizes. Modern server processors let users control the allocation of LLC space to processor threads. In this way, a user can limit each thread to only use its allocated amount of shared resources. Such facilities are often called Quality of Service (QoS) extensions. They can be used to prioritize performance-critical applications and to reduce interference with other threads in the same system. Besides LLC allocation, QoS extensions support limiting memory read bandwidth.

Our analysis will help us identify applications whose performance drops significantly when decreasing the size of the LLC. We say that such applications are sensitive to the size of the LLC. Also, we identified applications that are not sensitive, i.e., LLC size doesn't have an impact on performance. This result can be applied to properly size the processor LLC, especially considering the wide range available on the market. For example, we can determine that an application benefits from a larger LLC. Then perhaps an investment in new hardware is justified. Conversely, if the performance of an application doesn't improve from having a large LLC, then we can probably buy a cheaper processor.

For this case study, we use an AMD Milan processor, but other server processors such as Intel Xeon [Herdich et al., 2016], and Arm ThunderX [Wang et al., 2017], also include hardware support for users to control the allocation of both LLC space and memory read bandwidth to processor threads. Based on our tests, the method described in this section works equally well on AMD Zen4-based desktop processors, such as 7950X and 7950X3D.

²⁴⁴ CPU overclocking records - <https://press.asus.com/news/press-releases/rog-maximus-z790-apex-encore-sets-3-overclocking-world-records/>

Target machine: AMD EPYC 7313P

We have used a server system with a 16-core AMD EPYC 7313P processor, code-named Milan, which AMD launched in 2021. The main characteristics of this system are specified in table 12.1.

Table 12.1: Main features of the server used in the experiments.

Feature	Value
Processor	AMD EPYC 7313P
Cores x threads	16 × 2
Configuration	4 CCX × 4 cores/CCX
Frequency	3.0/3.7 GHz, base/max
L1 cache (I, D)	8-ways, 32 KiB (per core), 64-byte lines
L2 cache	8-ways, 512 KiB (per core), 64-byte lines
LLC	16-ways, 32 MB, non-inclusive (per CCX), 64-byte lines
Main Memory	512 GiB DDR4, 8 channels, nominal peak BW: 204.8 GB/s
TurboBoost	Disabled
Hyperthreading	Disabled (1 thread/core)
OS	Ubuntu 22.04, kernel 5.15.0-76

Figure 12.4 shows the clustered memory hierarchy of an AMD Milan 7313P processor. It consists of four Core Complex Dies (CCDs) connected to each other and to off-chip memory via an I/O chiplet. Each CCD integrates a Core CompleX (CCX) and an I/O connection. In turn, each CCX has four Zen3 cores that share a 32 MB victim LLC.²⁴⁵

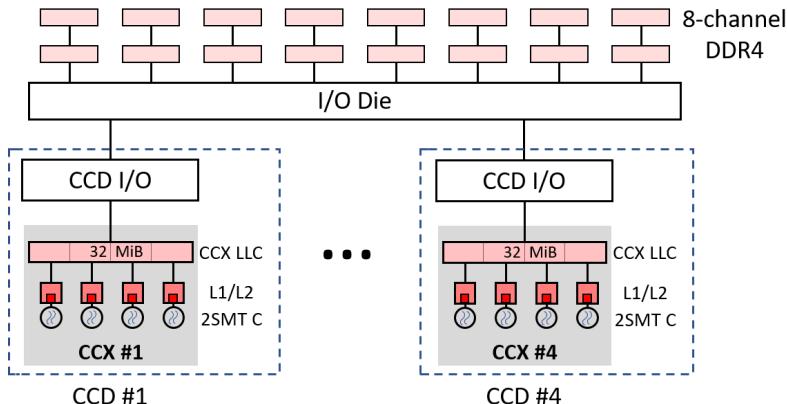


Figure 12.4: The clustered memory hierarchy of the AMD Milan 7313P processor.

Although there is a total of 128 MB of LLC (32 MB/CCX × 4 CCX), the four cores of a CCX cannot store cache lines in an LLC other than their own 32 MB LLC. Since we will be running single-threaded benchmarks, we can focus on a single CCX. The LLC size in our experiments will vary from 0 to 32 MB with steps of 2 MB. This is

²⁴⁵“Victim” means that the LLC is filled with the cache lines evicted from the four L2 caches of a CCX.

directly related to having a 16-way LLC: by disabling one of 16 ways, we reduce the LLC size by 2 MB.

Workload: SPEC CPU2017

We use a subset of applications from the SPEC CPU2017 suite²⁴⁶. SPEC CPU2017 contains a collection of industry-standardized workloads that benchmark the performance of the processor, memory subsystem, and compiler. It is widely used to compare the performance of high-performance systems and in computer architecture research.

Specifically, we selected 15 memory-intensive benchmarks from SPEC CPU2017 (6 INT and 9 FP) as suggested in [Navarro-Torres et al., 2019]. These applications have been compiled with GCC 6.3.1 and the following compiler options: `-O3 -march=native -fno-unsafe-math-optimizations`.

Controlling and Monitoring LLC allocation

To monitor and enforce limits on LLC allocation and memory *read* bandwidth, we will use *AMD64 Technology Platform Quality of Service Extensions* [Advanced Micro Devices, 2022]. Users can manage this QoS extension through the banks of model-specific registers (MSRs). First, a thread or a group of threads must be assigned a resource management identifier (RMID), and a class of service (COS) by writing to the PQR_ASSOC register (MSR 0xC8F). Here is a sample command for the hardware thread 1:

```
# write PQR_ASSOC (MSR 0xC8F): RMID=1, COS=2 -> (COS << 32) + RMID
$ wrmsr -p 1 0xC8F 0x200000001
```

where `-p 1` refers to the hardware thread 1. All `rdmsr` and `wrmsr` commands that we show require root access.

LLC space management is performed by writing to a 16-bit per-thread binary mask. Each bit of the mask allows a thread to use a given sixteenth fraction of the LLC ($1/16 = 2$ MB in the case of the AMD Milan 7313P). Multiple threads can use the same fraction(s), implying a competitive shared use of the same subset of LLC.

To set limits on the LLC usage by thread 1, we need to write to the L3_MASK_n register, where n is the COS, the cache partitions that can be used by the corresponding COS. For example, to limit thread 1 to use only half of the available space in the LLC, run the following command:

```
# write L3_MASK_2 (MSR 0xC92): 0x00FF (half of the LLC space)
$ wrmsr -p 1 0xC92 0x00FF
```

Similarly, the memory read bandwidth allocated to a thread can be limited. This is achieved by writing an unsigned integer to a specific MSR register, which sets a maximum read bandwidth in 1/8 GB/s increments. Interested readers are welcome to read [Advanced Micro Devices, 2022] for more details.

Metrics

The ultimate metric for quantifying the performance of an application is execution time. To analyze the impact of the memory hierarchy on system performance, we will

²⁴⁶ SPEC CPU® 2017 - <https://www.spec.org/cpu2017/>.

also use the following three metrics: 1) CPI, cycles per instruction, 2) DMPKI, demand misses in the LLC per thousand instructions, and 3) MPKI, total misses (demand + prefetch) in the LLC per thousand instructions. While CPI has a direct correlation with the performance of an application, DMPKI and MPKI do not necessarily impact performance. Table 12.2 shows the formulas used to calculate each metric from specific hardware counters. Detailed descriptions for each of the counters are available in AMD’s Processor Programming Reference [Advanced Micro Devices, 2021].

Table 12.2: Formulas for calculating metrics used in the case study.

Metric	Formula
CPI	Cycles not in Halt (PMCx076) / Retired Instructions (PMCx0C0)
DMPKI	Demand Data Cache Fills ²⁴⁷ (PMCx043) / (Retired Instr (PMCx0C0) / 1000)
MPKI	L3 Misses ²⁴⁸ (L3PMCx04) / (Retired Instructions (PMCx0C0) / 1000)

The methodology used in this case study is described in more detail in [Navarro-Torres et al., 2023]. It also explains how we configured and read hardware counters. The code and the information necessary to reproduce the experiments can be found in the following public repository: <https://github.com/agusnt/BALANCER>.

Results

We run a set of SPEC CPU2017 benchmarks *alone* in the system using only one instance and a single hardware thread. We repeat those runs while changing the available LLC size from 0 to 32 MB in 2 MB steps.

Figure 12.5 shows in graphs, from left to right, CPI, DMPKI, and MPKI for each assigned LLC size. We only show three workloads, namely `503.bwaves` (blue), `520.omnetpp` (green), and `554.roms` (red). They cover the three main trends observed in all other applications. Thus, we do not show the rest of the benchmarks.

For the CPI chart, a lower value on the Y-axis means better performance. Also, since the frequency on the system is fixed, the CPI chart is reflective of absolute scores. For example, `520.omnetpp` (green line) with 32 MB LLC is 2.5 times faster than with 0 MB LLC. For the DMPKI and MPKI charts, the lower the value on the Y-axis, the better.

Two different behaviors can be observed in the CPI and DMPKI graphs. On one hand, `520.omnetpp` takes advantage of its available space in the LLC: both CPI and DMPKI decrease significantly as the space allocated in the LLC increases. We can say that the behavior of `520.omnetpp` is sensitive to the size available in the LLC. Increasing the allocated LLC space improves performance because it avoids evicting cache lines that will be used in the future.

In contrast, `503.bwaves` and `554.roms` don’t make use of all available LLC space. For both benchmarks, CPI and DMPKI remain roughly constant as the allocation limit in the LLC grows. We can say that the performance of these two applications is

²⁴⁷ We used subevents `MemIoRemote` and `MemIoLocal`, that count demand data cache fills from DRAM or IO connected in remote/local NUMA node.

²⁴⁸ We used a mask to count only L3 Misses, specifically, `L3Event[0x0300C00000400104]`.

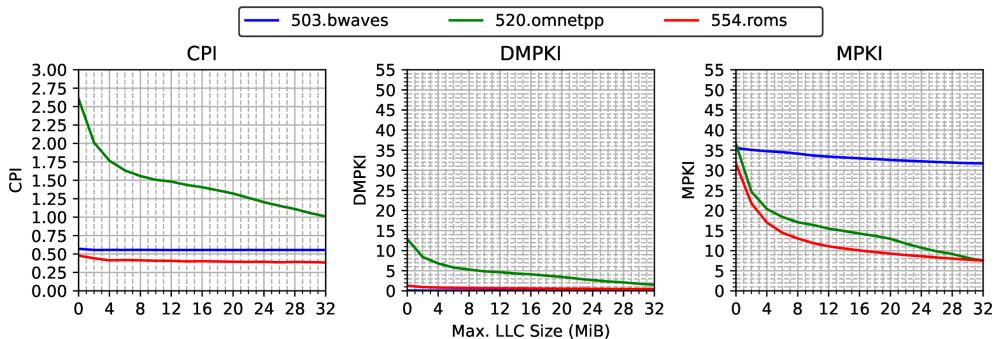


Figure 12.5: CPI, DMPKI, and MPKI for increasing LLC allocation limits (2 MB steps).

insensitive to their available space in the LLC.²⁴⁹ If your application shows similar behavior, you can buy a cheaper processor with a smaller LLC size without sacrificing performance.

Let's now analyze the MPKI graph, that combines both LLC demand misses and prefetch requests. First of all, we can see that the MPKI values are always much higher than the DMPKI values. That is, most of the blocks are loaded from memory into the on-chip hierarchy by the prefetcher. This behavior is due to the fact that the prefetcher is efficient in preloading the private caches with the data to be used, thus eliminating most of the demand misses.

For `503.bwaves`, we observe that MPKI remains roughly at the same level, similar to CPI and DMPKI charts. There is likely not much data reuse in the benchmark and/or the memory traffic is very low. The `520.omnetpp` workload behaves as we identified earlier: MPKI decreases as the available space increases.

However, for `554.roms`, the MPKI chart shows a large drop in total misses as the available space increases while CPI and DMPKI remain unchanged. In this case, there is data reuse in the benchmark, but it is not consequential to the performance. The prefetcher can bring required data ahead of time, eliminating demand misses, regardless of the available space in the LLC. However, as the available space decreases, the probability that the prefetcher will not find the blocks in the LLC and will have to load them from memory increases. So, giving more LLC capacity to `554.roms` does not directly benefit its performance, but it does benefit the system since it reduces memory traffic. So, it is better not to limit available LLC space for `554.roms` as it may negatively affect the performance of other applications running on the system. [Navarro-Torres et al., 2023]

Questions and Exercises

- Solve the following lab assignments from the Performance Ninja online course:
 - `perf-ninja::mem_orderViolation_1`

²⁴⁹ However, for `554.roms`, we can observe sensitivity to the available space in the LLC in the range 0-4 MB. Once the LLC size is 4 MB and above, the performance remains constant. That means that `554.roms` doesn't require more than 4 MB of LLC space to perform well.

- `perf-ninja::mem_alignment_1`
 - `perf-ninja::io_opt1`
2. Study the ISA extensions supported by the processor you're working with. Check if the application that you're working on uses these extensions. If not, can it benefit from them?
 3. Run the application that you're working with daily. Find the hotspot. Check if they suffer from any of the microarchitecture-specific issues that we discussed in this chapter.
 4. Describe how you can avoid page faults on a critical path in your application.

Chapter Summary

- Processors from different vendors are not created equal. They differ in terms of instruction set architecture (ISA) that they support and microarchitecture implementation. Reaching peak performance often requires leveraging the latest ISA extensions and tuning the application for a specific CPU microarchitecture.
- CPU dispatching is a technique that enables you to introduce platform-specific optimizations. Using it, you can provide a fast path for a specific microarchitecture while keeping a generic implementation for other platforms.
- We explored several performance corner cases that are caused by the interaction of the application with the CPU microarchitecture. These include memory ordering violations, misaligned memory accesses, cache aliasing, and denormal floating-point numbers.
- We also discussed a few low-latency tuning techniques that are essential for applications that require fast response times. We showed how to avoid page faults, cache misses, TLB shootdowns, and core throttling on a critical path.
- System tuning is the last piece of the puzzle. Some knobs and settings may affect the performance of your application. It is crucial to ensure that the system firmware, the OS, or the kernel does not destroy all the efforts put into tuning the application.

13 Optimizing Multithreaded Applications

Modern CPUs are getting more and more cores each year. As of 2024, you can buy a server processor which will have more than 200 cores! And even a laptop with 16 execution threads is a pretty usual setup nowadays. Since there is so much processing power in every CPU, effective utilization of all the hardware threads becomes more challenging. Preparing software to scale well with a growing amount of CPU cores is very important for the future success of your application.

There is a difference in how server and client products exploit parallelism. Most server platforms are designed to process requests from a large number of customers. Those requests are usually independent of each other, so the server can process them in parallel. If there is enough load on the system, applications themselves could be single-threaded, and still platform utilization will be high. However, when you use your server platform for HPC or AI computations; then you need all the computing power you have. On the other hand, client platforms, such as laptops and desktops, have all the resources to serve a single user. In this case, an application has to make use of all the available cores to provide the best user experience. In this chapter, we will focus on applications that can scale to a large number of cores.

From the software perspective, there are two primary ways to achieve parallelism: multiprocessing and multithreading. In a multiprocess application, multiple independent processes run concurrently. Each process has its own memory space and communicates with other processes through inter-process communication mechanisms such as pipes, sockets, or shared memory. In a multithreaded application, a single process contains multiple threads, which share the same memory space and resources of the process. Threads within the same process can communicate and share data more easily because they have direct access to the same memory space. However, synchronization between threads is usually more complex and is prone to issues like race conditions and deadlocks. In this chapter we will mostly focus on multithreaded applications, however, some techniques can be applied to multiprocess applications as well. We will show examples of both types of applications in this chapter.

When talking about throughput-oriented applications, we can distinguish the following two types of applications:

- **Massively parallel applications.** Such applications usually scale well with the number of cores. They are designed to process a large number of independent tasks. Massively parallel programs often use the divide-and-conquer technique to split the work into smaller tasks (also called *worker threads*) and process them in parallel. Examples of such applications are scientific computations, video rendering, data analytics, AI, and many others. The main obstacle for such applications is the saturation of a shared resource, such as memory bandwidth, that can effectively stall all the worker threads in the process.
- **Applications that require synchronization.** Such applications have workers share resources to complete their tasks. Worker threads depend on each other, which creates periods when some threads are stalled. Examples of such applications are databases, web servers, and other server applications. The main

challenge for such applications is to minimize required synchronization and to avoid contention on shared resources.

In this chapter, we will explore how to analyze the performance of both types of applications. Since this is a book about low-level performance, we will not discuss algorithm-level optimizations such as lock-free data structures, which are well covered in other books.

13.1 Parallel Efficiency Metrics

Let's start by introducing a few metrics that are important for analyzing the performance of multithreaded applications. When dealing with multithreaded applications, engineers should be careful in analyzing basic metrics, for example, CPU utilization. One of the threads might show high CPU utilization, but it could turn out that the thread was just spinning in a busy-wait loop while waiting for 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*.

Effective CPU Utilization

This metric represents how efficiently an application utilizes the available CPUs. It shows the percent of average CPU utilization by all logical CPUs on the system. It is based only on the *Effective time* and does not include the overhead introduced by the parallel runtime system²⁵⁰ and Spin time. An *Effective CPU utilization* of 100% means that your application keeps all the logical CPU cores busy for the entire time that it runs.

For a specified time interval T, *Effective CPU Utilization* can be calculated as

$$\text{Effective CPU Utilization} = \frac{\sum_{i=1}^{\text{ThreadCount}} \text{Effective CPU Time}(T,i)}{T \times \text{ThreadCount}}$$

$$\text{Effective CPU Time} = \text{CPU Time} - (\text{Overhead Time} + \text{Spin Time})$$

Measuring overhead and spin time can be challenging, and I recommend using a performance analysis tool like Intel VTune Profiler, which can provide these metrics.

Thread Count

Most parallel applications have a configurable number of threads, which allows them to run efficiently on platforms with a different number of cores. 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 *oversubscription*; some threads will be waiting for their turn to run.

Besides actual worker threads, multithreaded applications usually have other house-keeping threads: main thread, input/output threads, etc. If those threads consume significant time, they will take execution time away from worker threads, as they too

²⁵⁰ Threading libraries such as `pthread`, `OpenMP`, and `Intel TBB` incur additional overhead for creating and managing threads.

require CPU cores to run. 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 thread creation and destruction, engineers usually allocate a [pool of threads](#)²⁵¹ 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.

Wait Time

Wait Time occurs when software threads are waiting due to APIs that block or cause a context switch. Wait Time is per thread; therefore, the total *Wait Time* can exceed the application elapsed time.

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 problem either because of a large 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.

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. In reality, the implementation of kernel synchronization primitives spins on a lock for some time instead of immediately yielding to another thread. Too much Spin Time, however, can reflect the lost opportunity for productive work.

A list of other parallel efficiency metrics can be found on Intel's VTune [page](#).²⁵²

13.2 Performance Scaling in Multithreaded Programs

When dealing with a single-threaded application, optimizing one portion of a program usually yields positive results on performance. However, this is 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](#),²⁵³ which constitutes that the speedup of a parallel program is limited by its serial part. Figure 13.1a illustrates the theoretical speedup of the latency of the execution of a program as a function of the number of processors executing it. For a program, 75% of which is parallel, the speedup factor converges to 4.

²⁵¹ Thread pool - https://en.wikipedia.org/wiki/Thread_pool

²⁵² CPU metrics reference - <https://software.intel.com/en-us/vtune-help-cpu-metrics-reference>

²⁵³ Amdahl's law - https://en.wikipedia.org/wiki/Amdahl's_law.

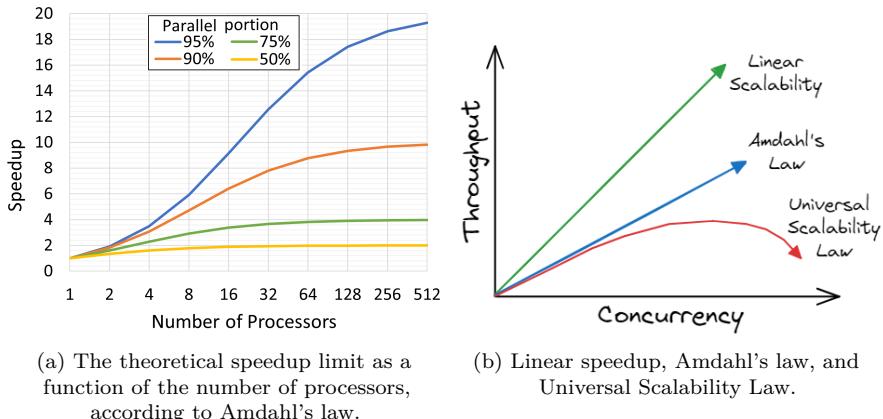


Figure 13.1: Amdahl's Law and Universal Scalability Law.

In reality, further adding computing nodes to the system may yield retrograde speed up. We will see examples of it in the next section. This effect is explained by Neil Gunther as the [Universal Scalability Law²⁵⁴](#) (USL), which is an extension of Amdahl's law. USL describes communication between computing nodes (threads) as yet another factor gating performance. As the system is scaled up, overheads start to neutralize the gains. Beyond a critical point, the capability of the system starts to decrease (see Figure 13.1b). USL is widely used for modeling the capacity and scalability of systems.

The 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 resources between many workers (coherence). For example, when multiple workers frequently change a globally visible object, those changes need to be broadcast to all nodes that use that object. Suddenly, usual operations start taking more time to finish due to the additional need to maintain coherence. 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.

13.2.1 Thread Count Scaling Case Study

Thread count scaling is perhaps the most valuable analysis you can perform on a multithreaded application. It shows how well the application can utilize modern multicore systems. As you will see, there is a ton of information you can learn along the way. Without further introduction, let's get started. In this case study, we will analyze the thread count scaling of the following benchmarks, some of which should be already familiar to you from the previous chapters:

1. Blender 3.4 - an open-source 3D creation and modeling software project.

²⁵⁴ USL law - http://www.perfdynamics.com/Manifesto/USLscalability.html#tth_sEc1.

This test is of Blender's Cycles performance with the BMW27 blend file. 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 -t N`, where N is the number of threads.

2. Clang 17 build - this test uses Clang 15 to build the Clang 17 compiler from sources. URL: <https://www.llvm.org>. Command line: `ninja -jN clang`, where N is the number of threads.
3. Zstandard v1.5.5, a fast lossless compression algorithm. URL: <https://github.com/facebook/zstd>. A dataset used for compression: <http://wanos.co/assets/silesia.tar>. Command line: `./zstd -TN -3 -f -- silesia.tar`, where N is the number of compression worker threads.
4. CloverLeaf 2018 - a Lagrangian-Eulerian hydrodynamics benchmark. All hardware threads are used. This test uses the input file `clover_bm.in` (Problem 5). URL: <http://uk-mac.github.io/CloverLeaf>. Command line: `export OMP_NUM_THREADS=N; ./clover_leaf`, where N is the number of threads.
5. CPython 3.12, a reference implementation of the Python programming language. URL: <https://github.com/python/cpython>. I ran a simple multithreaded binary search script written in Python, which searches 10,000 random numbers (needles) in a sorted list of 1,000,000 elements (haystack). Command line: `./python3 binary_search.py N`, where N is the number of threads. Needles are divided equally between threads.

The benchmarks were executed on a machine with the configuration shown below:

- 12th Gen Alder Lake Intel® Core™ i7-1260P CPU @ 2.10GHz (4.70GHz Turbo), 4P+8E cores, 18MB L3-cache.
- 16 GB RAM, DDR4 @ 2400 MT/s.
- Clang 15 compiler with the following options: `-O3 -march=core-avx2`.
- 256GB NVMe PCIe M.2 SSD.
- 64-bit Ubuntu 22.04.1 LTS (Jammy Jellyfish, Linux kernel 6.5).

This is clearly not the top-of-the-line hardware setup, but rather a mainstream computer, not necessarily designed to handle media, developer, or HPC workloads. However, it is an excellent platform for our case study as it demonstrates various effects of thread count scaling. Because of the limited resources, applications start to hit performance roadblocks even with a small number of threads. Keep in mind, that on better hardware, the scaling results will be different.

My processor has four P-cores and eight E-cores. P-cores are SMT-enabled, which means the total number of threads on this platform is sixteen. By default, the Linux scheduler will first try to use idle physical P-cores. The first four threads will utilize four threads on four idle P-cores. When they are fully utilized, it will start to schedule threads on E-cores. So, the next eight threads will be scheduled on eight E-cores. Finally, the remaining four threads will be scheduled on the 4 sibling SMT threads of P-cores.

I ran the benchmarks while affinitizing threads using the aforementioned scheme, except `Zstd` and `CPython`. Running without affinity does a better job of representing real-world scenarios, however, thread affinity makes thread count scaling analysis cleaner. Since performance numbers were very similar, in this case study I present the results when thread affinity is used.

The benchmarks do a fixed amount of work. The number of retired instructions is almost identical regardless of the thread count. In all of them, the largest portion of an algorithm is implemented using a divide-and-conquer paradigm, where work is split into equal parts, and each part can be processed independently. In theory, this allows applications to scale well with the number of cores. However, in practice, the scaling is often far from optimal.

Figure 13.2 shows the thread count scalability of the selected benchmarks. The x-axis represents the number of threads, and the y-axis shows the speedup relative to the single-threaded execution. The speedup is calculated as the execution time of the single-threaded execution divided by the execution time of the multi-threaded execution. The higher the speedup, the better the application scales with the number of threads.

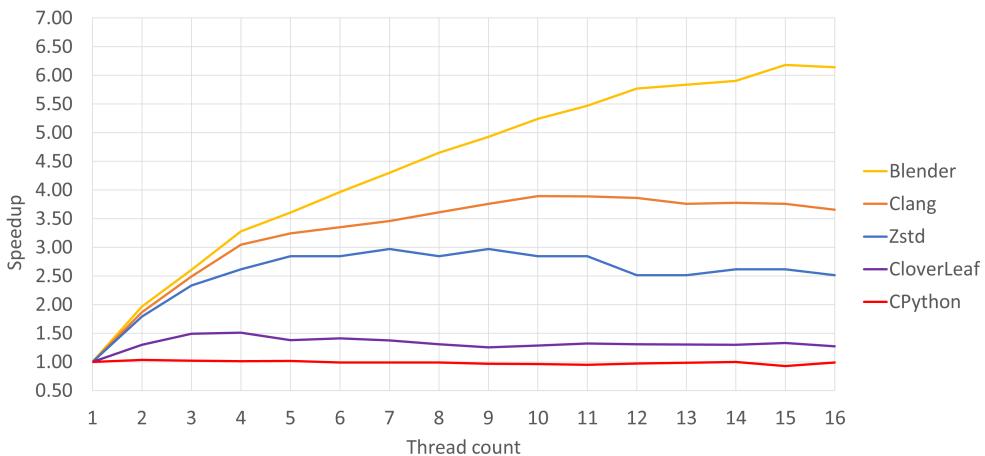


Figure 13.2: Thread Count Scalability chart for five selected benchmarks.

As you can see, most of them are very far from the linear scaling, which is quite disappointing. The benchmark with the best scaling in this case study, Blender, achieves only 6x speedup while using 16x threads. CPython, for example, enjoys no thread count scaling at all. The performance of Clang and Zstd degrades when the number of threads goes beyond 10. To understand why that happens, let's dive into the details of each benchmark.

Blender

Blender is the only benchmark in our suite that continues to scale up to all 16 threads in the system. The reason for this is that the workload is highly parallelizable. The rendering process is divided into small tiles, and each tile can be rendered independently. However, even with this high level of parallelism, the scaling is only $6.1 \times \text{speedup} / 16 \text{ threads} = 38\%$. What are the reasons for this suboptimal scaling?

From Section 4.11, we know that Blender's performance is bounded by floating-point computations. It has a relatively high percentage of SIMD instructions as well. P-cores are much better at handling such instructions than E-cores. This is why we see the slope of the speedup curve decrease after 4 threads as E-cores start getting used.

Performance scaling continues at the same pace up until 12 threads, where it starts to degrade again. This is the effect of using SMT sibling threads. Two active sibling SMT threads compete for the limited number of FP/SIMD execution units. To measure SMT scaling, we need to divide the performance of two SMT threads (2T1C - two threads one core) by the performance of a single P-core (1T1C).²⁵⁵ For Blender, SMT scaling is around 1.3x.

There is another aspect of scaling degradation that also applies to Blender that we will talk about while discussing Clang's thread count scaling.

Clang

While Blender uses multithreading to exploit parallelism, concurrency in C++ compilation is usually achieved with multiprocessing. Clang 17 has more than 2,500 translation units, and to compile each of them, a new process is spawned. Similar to Blender, we classify Clang compilation as massively parallel, yet they scale differently. We recommend you revisit Section 4.11 for an overview of Clang compiler performance bottlenecks. In short, it has a large codebase, flat profile, many small functions, and “branchy” code. Its performance is affected by D-Cache, I-Cache, and TLB misses, and branch mispredictions. Clang's thread count scaling is affected by the same scaling issue as Blender: P-cores are more effective than E-cores, and P-core SMT scaling is about 1.1x. However, there is more. Notice that scaling stops at around 10 threads, and starts to degrade. Let's understand why that happens.

The problem is related to the frequency throttling. When multiple cores are utilized simultaneously, the processor generates more heat due to the increased workload on each core. To prevent overheating and maintain stability, CPUs often throttle down their clock speeds depending on how many cores are in use. Additionally, boosting all cores to their maximum turbo frequency simultaneously would require significantly more power, which might exceed the power delivery capabilities of the CPU. My system doesn't have an advanced liquid cooling solution and only has a single processor fan. That's why it cannot sustain high frequencies when many cores are utilized.

Figure 13.3 shows the performance scaling of the Clang workload overlaid with the CPU frequency on our platform while running with different thread counts. Notice that sustained frequency drops when we start using just two P-cores simultaneously. By the time you start using all 16 threads, the frequency of P-cores is throttled down to 3.2GHz, while E-cores operate at 2.6GHz. I used Intel VTune's platform view to capture CPU frequencies as shown in Section 7.1.

The tipping point of performance scaling for the Clang workload is around 10 threads. This is the point where the frequency throttling starts to have a significant impact on performance, and the benefit of adding additional threads is smaller than the penalty of running at a lower frequency.

Keep in mind that this frequency chart cannot be automatically applied to all other workloads. Applications that heavily use SIMD instructions typically operate on lower frequencies, so Blender, for example, may see slightly more frequency throttling than Clang. However, such a chart can give you a good intuition about the frequency throttling issues that occur on your platform.

²⁵⁵ Also, you can measure SMT scaling as 4T2C/2T2C, 6T3C/3T3C, and so on.

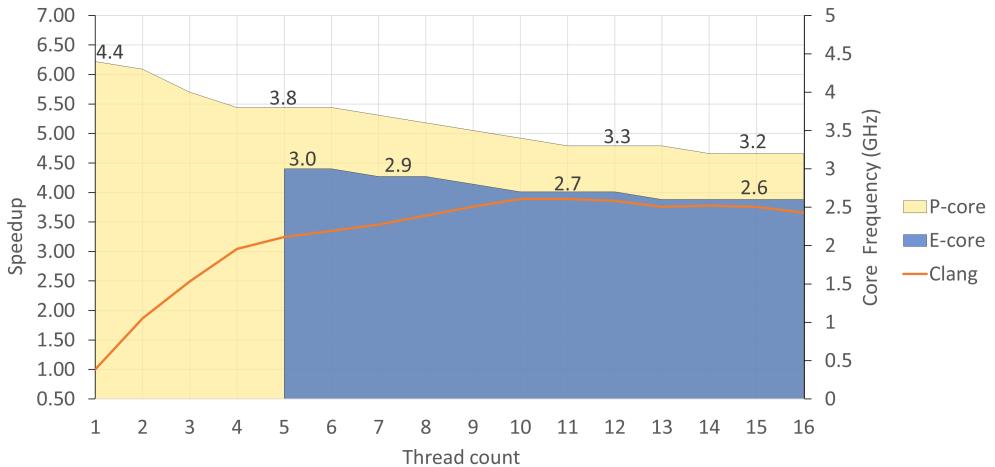


Figure 13.3: Frequency throttling while running Clang compilation on Intel® Core™ i7-1260P. E-cores become active only after using four threads on P-cores.

To confirm that frequency throttling is one of the main reasons for performance degradation, I temporarily disabled Turbo Boost on my platform and repeated the scaling study for Blender and Clang. When Turbo Boost is disabled, all cores operate on their base frequencies, which are 2.1 Ghz for P-cores and 1.5 Ghz for E-cores. The results are shown in Figure 13.4. As you can see, thread count scaling almost doubles when all 16 threads are used and TurboBoost is disabled, for both Blender ($38\% \rightarrow 69\%$) and Clang ($21\% \rightarrow 41\%$). It gives us an intuition of what the thread count scaling would look like if frequency throttling had not happened. In fact, frequency throttling accounts for a large portion of unrealized performance scaling in modern systems.

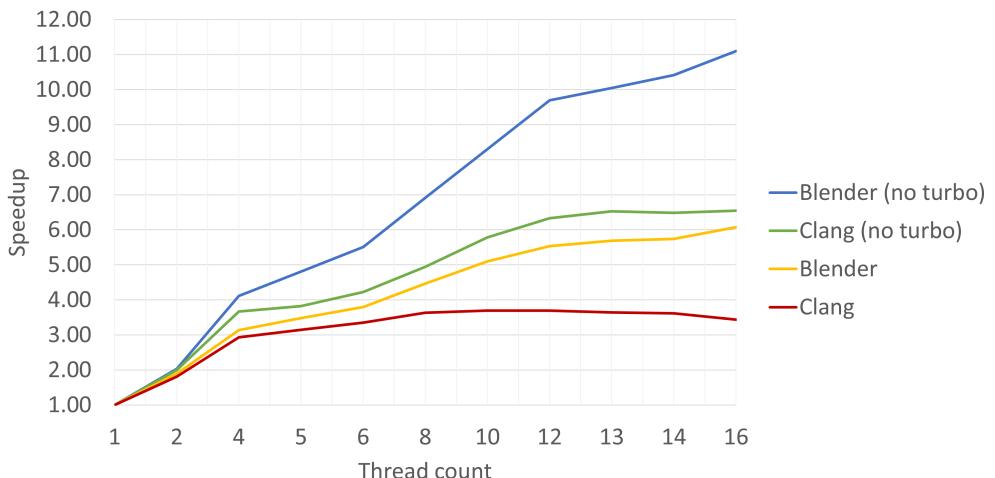


Figure 13.4: Thread Count Scalability chart for Blender and Clang with disabled Turbo Boost. Frequency throttling is a major roadblock to achieving good thread count scaling.

Zstandard

Next on our list is the Zstandard compression algorithm, or Zstd for short. When compressing data, Zstd divides the input into blocks, and each block can be compressed independently. This means that multiple threads can work on compressing different blocks simultaneously. Although it seems that Zstd should scale well with the number of threads, it doesn't. Performance scaling stops at around 5 threads, sooner than in the previous two benchmarks. As you will see, the dynamic interaction between Zstd worker threads is quite complicated.

First of all, the performance of Zstd depends on the compression level. The higher the compression level, the more compact the result. Lower compression levels provide faster compression, while higher levels yield better compression ratios. In the case study, I used compression level 3 (which is also the default level) since it provides a good trade-off between speed and compression ratio.

Here is the high-level algorithm of Zstd compression:

- The input file is divided into blocks, whose size depends on the compression level. Each job is responsible for compressing a block of data. When Zstd receives some data to compress, the main thread copies a small chunk into one of its internal buffers and posts a new compression job, which is picked up by one of the worker threads. In a similar manner, the main thread fills all input buffers for all its workers and sends them to work in order. Worker threads share a common queue from which they concurrently pick up jobs.
- Jobs are always started in order, but they can be finished in any order. Compression speed can be variable and depends on the data to compress. Some blocks are easier to compress than others.
- After a worker finishes compressing a block, it signals to the main thread that the compressed data is ready to be flushed to the output file. The main thread is responsible for flushing the compressed data to the output file. Note that flushing must be done in order, which means that the second job is allowed to be flushed only after the first one is entirely flushed. The main thread can “partially flush” an ongoing job, i.e., it doesn't have to wait for a job to be completely finished to start flushing it.

To visualize the work of the Zstd algorithm on a timeline, I instrumented the Zstd source code with VTune's Instrumentation and Tracing Technology (ITT) markers.²⁵⁶ They enable us to visualize the duration of instrumented code regions and events on the timeline and control the collection of trace data during execution.

The timeline of compressing the Silesia corpus using 8 threads is shown in Figure 13.5. Using 8 worker threads is enough to observe thread interaction in Zstd while keeping the image less noisy than when all 16 threads are active. The second half of the timeline was cut to make the image fit on the page.

On the image, we have the main thread at the bottom (TID 913273), and eight worker threads at the top. The worker threads are created at the beginning of the compression process and are reused for multiple compressing jobs.

²⁵⁶ VTune ITT instrumentation - <https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2023-1/instrumenting-your-application.html>

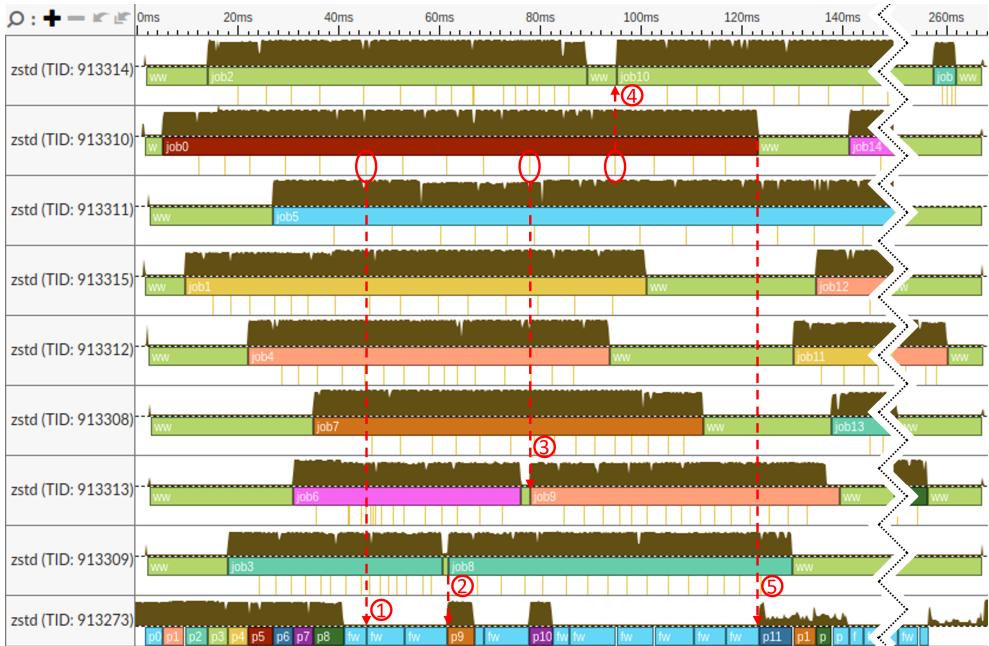


Figure 13.5: Timeline view of compressing Silesia corpus with Zstandard using 8 threads.

On the worker thread timeline (top 8 rows) we have the following markers:

- job0–job25 bars indicate the start and end of a job.
- ww (short for “worker wait”, green bars) bars indicate a period when a worker thread is waiting for a new job.
- Notches below job periods indicate that a thread has just finished compressing a portion of the input block and is signaling to the main thread that there is data available to be flushed (partial flushing).

On the main thread timeline (the bottom row, TID 913273) we have the following markers:

- p0–p25 boxes indicate periods of preparing a new job. It starts when the main thread starts filling up the input buffer until it is full (but this new job is not necessarily posted on the worker queue immediately).
- fw (short for “flush wait”, blue bars) markers indicate a period when the main thread waits for the produced data to start flushing it. This happens when the main thread has prepared the next job and has nothing else to do. During this time, the main thread is blocked.

With a quick glance at the image, we can tell that there are many **ww** periods when worker threads are waiting. This negatively affects the performance of Zstandard compression. Let’s progress through the timeline and try to understand what’s going on.

1. First, when worker threads are created, there is no work to do, so they are waiting for the main thread to post a new job.

2. Then the main thread starts to fill up the input buffers for the worker threads. It has prepared jobs 0 to 7 (see bars p0–p7), which were picked up by worker threads immediately. Notice, that the main thread also prepared job8 (p8), but it hasn't posted it in the worker queue yet. This is because all workers are still busy.
3. After the main thread has finished p8, it flushed the data already produced by job0. Notice, that by this time, job0 has already delivered five portions of compressed data (first five notches below the job0 bar). Now, the main thread enters its first fw period and starts to wait for more data from job0.
4. At the timestamp 45ms, one more chunk of compressed data is produced by job0, and the main thread briefly wakes up to flush it, see (1). After that, it goes to sleep (fw) again.
5. Job3 is the first to finish, but there is a couple of milliseconds delay before TID 913309 picks up the new job, see (2). This happens because job8 was not posted in the queue by the main thread. At this time, the main thread is waiting for a new chunk of compressed data from job0. When it arrives a couple of milliseconds later, the main thread wakes up, flushes the data, and notices that there is an idle worker thread (TID 913309, the one that has just finished job3). So, the main thread posts job8 to the worker queue and starts preparing the next job (p9).
6. The same thing happens with TID 913313 (see (3)) and TID 913314 (see (4)). But this time the delay is bigger. Interestingly, job10 could have been picked up by either TID 913314 or TID 913312 since they were both idle at the time job10 was pushed to the job queue.
7. We should have expected that the main thread would start preparing job11 immediately after job10 was posted in the queue as it did before. But it didn't. This happens because there are no available input buffers. We will discuss it in more detail shortly.
8. Only when job0 finishes, the main thread was able to acquire a new input buffer and start preparing job11 (see (5)).

As we just said, the reason for the 20–40ms delays between jobs (e.g., between job4 and job11) is the lack of input buffers, which are required to start preparing a new job. Zstd maintains a single memory pool, which allocates space for both input and output buffers. This memory pool is prone to fragmentation issues, as it has to provide contiguous blocks of memory. When a worker finishes a job, the output buffer is waiting to be flushed, but it still occupies memory. To start working on another job, it will require another pair of buffers (one input and one output buffer).

Limiting the capacity of the memory pool is a design decision to reduce memory consumption. In the worst case, there could be many “run-away” buffers, left by workers that have completed their jobs very fast, and moved on to process the next job; meanwhile, the flush queue is still blocked by one slow job and the buffers cannot be released. In such a scenario, the memory consumption would be high, which is undesirable. However, the downside of the current implementation is increased wait time between the jobs.

The Zstd compression algorithm is a good example of a complex interaction between threads. Visualizing worker threads on a timeline is extremely helpful in understanding how threads communicate and synchronize, and can be useful for identifying bottlenecks.

It is also a good reminder that even if you have a parallelizable workload, the performance of your application can be limited by the synchronization between threads and resource availability.

CloverLeaf

CloverLeaf is a hydrodynamics workload. We will not dig deeply into the details of the underlying algorithm as it is not relevant to this case study. CloverLeaf uses OpenMP to parallelize the workload. HPC workloads usually scale well, so we expect CloverLeaf to scale well too. However, on my platform performance stops growing after using 3 threads. What's going on?

To determine the root cause of poor scaling, I collected TMA metrics (see Section 6.1) in four data points: running CloverLeaf with one, two, three, and four threads. Once we compare the performance characteristics of these profiles, one thing becomes clear immediately: CloverLeaf's performance is bound by memory bandwidth. Table 13.1 shows the relevant metrics from these profiles that highlight increasing memory bandwidth demand when using multiple threads.

Table 13.1: Performance metrics for CloverLeaf workload.

Metric	1 thread	2 threads	3 threads	4 threads
TMA::Memory Bound (% of pipeline slots)	34.6	53.7	59.0	65.4
TMA::DRAM Memory Bandwidth (% of cycles)	71.7	83.9	87.0	91.3
Memory Bandwidth Utilization (range, GB/s)	20-22	25-28	27-30	27-30

As you can see from those numbers, the pressure on the memory subsystem keeps increasing as we add more threads. An increase in the *TMA::Memory Bound* metric indicates that threads increasingly spend more time waiting for data and do less useful work. An increase in the *DRAM Memory Bandwidth* metric further highlights that performance is hurt due to approaching bandwidth limits. The *Memory Bandwidth Utilization* metric indicates the range of total memory bandwidth utilization while CloverLeaf was running. I captured these numbers by looking at the memory bandwidth utilization chart in VTune's platform view as shown in Figure 13.6.

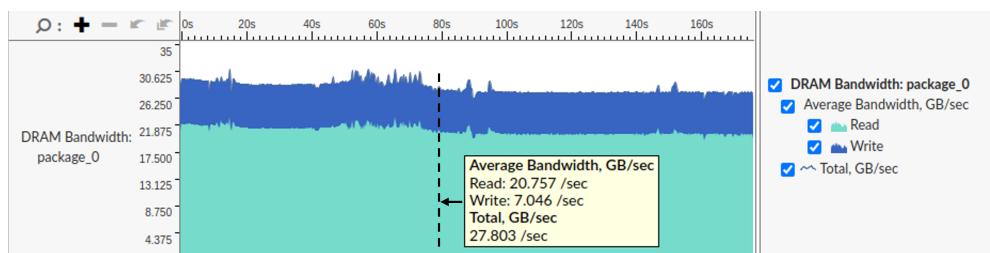


Figure 13.6: VTune's platform view of running CloverLeaf with 3 threads.

Let's put those numbers into perspective. The maximum theoretical memory bandwidth of my platform is 38.4 GB/s. However, as I measured in Section 4.10, the

maximum memory bandwidth that can be achieved in practice is 35 GB/s. With just a single thread, the memory bandwidth utilization reaches 2/3 of the practical limit. CloverLeaf fully saturates the memory bandwidth with three threads. Even when all 16 threads are active, *Memory Bandwidth Utilization* doesn't go above 30 GB/s, which is 86% of the practical limit.

To confirm my hypothesis, I swapped two 8 GB DDR4 2400 MT/s memory modules with two DDR4 modules of the same capacity, but faster speed: 3200 MT/s. This brings the theoretical memory bandwidth of the system to 51.2 GB/s and the practical maximum to 45 GB/s. The resulting performance boost grows with an increasing number of threads used and is in the range of 10% to 33%. When running CloverLeaf with 16 threads, faster memory modules provide the expected 33% performance as a ratio of the memory bandwidth increase ($3200 / 2400 = 1.33$). But even with a single thread, there is a 10% performance improvement. This means that there were moments when CloverLeaf fully saturated the memory bandwidth with a single thread in the original configuration.

Interestingly, for CloverLeaf, TurboBoost doesn't provide any performance benefit when all 16 threads are used, i.e., performance is the same regardless of whether you enable Turbo or let the cores run on their base frequency. How is that possible? The answer is: that having 16 active threads is enough to saturate two memory controllers even if CPU cores run at half the frequency. Since most of the time threads are just waiting for data, when you disable Turbo, they simply start to wait "slower".

C~~P~~ython

The final benchmark in the case study is CPython. I wrote a simple multithreaded Python script that uses binary search to find numbers (needles) in a sorted list (haystack). Needles are divided equally between worker threads. The script that I wrote doesn't scale at all. Can you guess why?

To solve this puzzle, I built CPython 3.12 from sources with debug information and ran Intel VTune's *Threading Analysis* collection while using two threads. Figure 13.7 visualizes a small portion of the timeline of the Python script execution. As you can see, the CPU time alternates between two threads. They work for 5 ms, then yield to another thread. In fact, if you were to scroll left or right, you would see that they never run simultaneously.

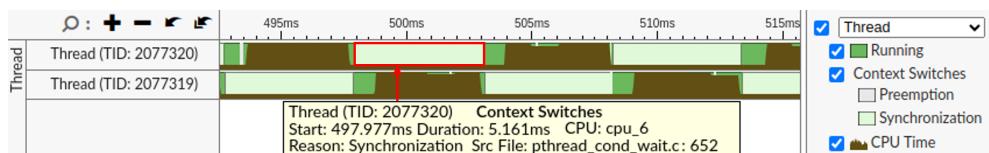


Figure 13.7: VTune's timeline view when running our Python script with two worker threads (other threads are filtered out).

Let's try to understand why two worker threads take turns instead of running together. Once a thread finishes its turn, the Linux kernel scheduler switches to another thread as highlighted in Figure 13.7. It also gives the reason for a context switch. If we take

a look at the `pthread_cond_wait.c` source code²⁵⁷ at line 652, we would land on the function `__pthread_cond_timedwait64`, which waits for a condition variable to be signaled. Many other inactive wait periods wait for the same reason.

On the *Bottom-up* page (see the left panel of Figure 13.8), VTune reports that the `__pthread_cond_timedwait64` function is responsible for the majority of *Inactive Sync Wait Time*. On the right panel, you can see the corresponding call stack. Using this call stack we can tell what is the most frequently used code path that led to the `__pthread_cond_timedwait64` function and subsequent context switch.

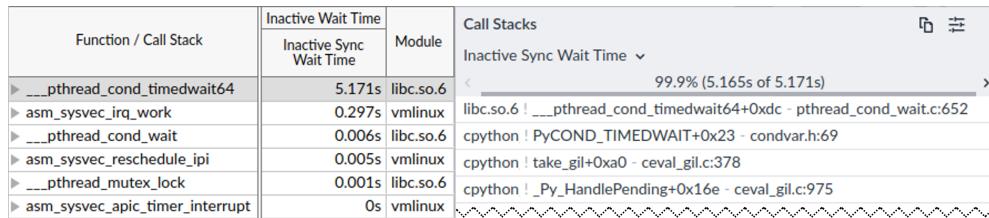


Figure 13.8: VTune’s timeline view when running our Python script with two threads (other threads are filtered out).

This call stack leads us to the `take_gil` function, which is responsible for acquiring the Global Interpreter Lock (GIL). The GIL is preventing our attempts at running worker threads in parallel by allowing only one thread to run at any given time, effectively turning our multithreaded program into a single-threaded one. If you take a look at the implementation of the `take_gil` function, you will find out that it uses a version of wait on a conditional variable with a timeout of 5 ms. Once the timeout is reached, the waiting thread asks the GIL-holding thread to drop it. Once another thread complies with the request, the waiting thread acquires the GIL and starts running. They keep switching roles until the very end of the execution.

Experienced Python programmers would immediately understand the problem, but in this example, I demonstrated how to find contested locks even without an extensive knowledge of CPython internals. CPython is the default and by far the most widely used Python interpreter. Unfortunately, it comes with GIL, which destroys the performance of compute-bound multithreaded Python programs.

Nevertheless, there are ways to bypass GIL, for example, by using GIL-immune libraries such as NumPy, writing performance-critical parts of the code as a C extension module, or using alternative runtime environments, such as `nogil`.²⁵⁸ Also, in Python 3.13 there is experimental support for running with the global interpreter lock disabled.²⁵⁹

Summary

In the case study, we have analyzed several throughput-oriented applications with varying thread count scaling characteristics. Here is a quick summary of our findings:

²⁵⁷ Glibc source code - <https://sourceware.org/git/?p=glibc.git;a=tree>

²⁵⁸ Nogil - <https://github.com/colesbury/nogil>

²⁵⁹ Python 3.13 release notes - <https://docs.python.org/3/whatsnew/3.13.html>

- Frequency throttling is a major roadblock to achieving good thread count scaling. Any application that makes use of multiple hardware threads suffers from frequency drop due to thermal limits. Platforms that have processors with higher TDP (Thermal Design Power) and advanced liquid cooling solutions are less prone to frequency throttling.
- Thread count scaling on hybrid processors (with performant and energy-efficient cores) is penalized because E-cores are less performant than P-cores. Once E-cores start being used, performance scaling is slowing down. Sibling SMT threads also don't provide good performance scaling.
- Worker threads in a throughput-oriented workload share a common set of resources, which may become a bottleneck. As we saw in the CloverLeaf example, performance doesn't scale because of the memory bandwidth limitation. This is a common problem for many HPC and AI workloads. Once you hit that limitation, everything else becomes less important, including code optimizations and even CPU frequency. The L3 cache and I/O are other examples of shared resources that often become a bottleneck.
- Finally, the performance of a concurrent application may be limited by the synchronization between threads as we saw in Zstd and CPython examples. Some programs have very complex interactions between threads, so it is very useful to visualize worker threads on a timeline. Also, you should know how to find contested locks using performance profiling tools.

To confirm that suboptimal scaling is a common case, rather than an exception, let's look at the SPEC CPU 2017 suite of benchmarks. In the *rate* part of the suite, each hardware thread runs its own single-threaded workload, so there are no slowdowns caused by thread synchronization. According to [Marr, 2023], benchmarks that have integer code (regular general-purpose programs) have a thread count scaling in the range of 40% - 70%, while benchmarks that have floating-point code (scientific, media, and engineering programs) have a scaling in the range of 20% - 65%. Those numbers represent inefficiencies caused just by the hardware platform. Inefficiencies caused by thread synchronization in multithreaded programs further degrade performance scaling.

In a latency-oriented application, you typically have a few performance-critical threads and the rest do background work that doesn't necessarily have to be fast. Many issues that we've discussed apply to latency-oriented applications as well. We covered some low-latency tuning techniques in Section 12.3.

13.3 Task Scheduling

With the emergence of hybrid processors, task scheduling becomes very challenging. For example, recent Intel's Meteor Lake chips have three types of cores; all with different performance characteristics. As you will see in this section, it is very easy to pessimize the performance of a multithreaded application by scheduling tasks suboptimally. Implementing a generic task scheduling policy is tricky because it greatly depends on the nature of the running tasks. Here are some examples:

- Compute-intensive lightly-threaded workloads (e.g., data compression) must be served only on P-cores.
- Background tasks (e.g., video calls) could be run on E-cores to save power.

- For bursty applications that demand high responsiveness (e.g., productivity software), a system should only use P-cores.
- Multithreaded programs with sustained performance demand (e.g., video rendering) should utilize both P- and E-cores.

For the most part, task schedulers in modern operating systems take care of these and many other corner cases. For example, Intel's Thread Director helps monitor and analyze performance data in real-time to seamlessly place the right application thread on the right core. My general recommendation here is to let the operating system do its job and not restrict it too much. The operating system knows how to schedule tasks to minimize contention, maximize reuse of data in caches, and ultimately maximize performance. This will play a big role if you are developing cross-platform software that is intended to run on different hardware configurations.

Below I show a few typical pitfalls of task scheduling in asymmetric systems. I took the same system I used in the previous case study: 12th Gen Alder Lake Intel® Core™ i7-1260P CPU, which has four P-cores and eight E-cores. For simplicity, I only enabled two P-cores and two E-cores; the rest of the cores were temporarily disabled. I also disabled SMT sibling threads on the two active P-cores. I wrote a simple OpenMP application, where each worker thread runs several bit manipulation operations on every 32-bit integer element of a large array. After a worker thread has finished processing, it hits a barrier and is forced to wait for other threads to finish their parts. After that, the main thread cleans up the array and the processing repeats. The program was compiled with GCC 13.2 and `-O3 -march=core-avx2`, which enables vectorization.

Figure 13.9 shows three strategies, which highlight common problems that I regularly see in practice. These screenshots were captured with Intel VTune. The bars on the timeline indicate CPU time, i.e., periods when a thread was running. For each software thread, there is one or two corresponding CPU cores. Using this view, we can see on which core each thread was running at any given moment.

Our first example uses static partitioning, which divides the processing of our large array into four equal chunks (since I have four cores enabled). For each chunk, the OpenMP runtime spawns a new thread. Also, I used `OMP_PROC_BIND=true`, which instructs OpenMP runtime to pin spawned threads to the CPU cores. Figure 13.9a demonstrates the effect: P-cores are much better at handling SIMD instructions than E-cores and they finish their jobs two times faster (see *Thread 1* and *Thread 2*). However, thread affinity does not allow *Thread 3* and *Thread 4* to migrate to P-cores, which are waiting at the barrier. That results in a high latency, which is limited by the speed of E-cores.

My recommendation is to avoid pinning threads to cores. With unbalanced work, pinning might restrict the work stealing, leaving the long execution tail for E-cores. On macOS, it is not possible to pin threads to cores since the operating system does not provide an API for that.

In the second example, I don't pin threads anymore, but the partitioning scheme remains the same (four equal chunks). Figure 13.9b illustrates the effect of this change. As in the previous scenario, *Thread 1* and *Thread 4* finished their jobs early, because they were using P-cores. *Thread 2* and *Thread 3* started running on E-cores, but once

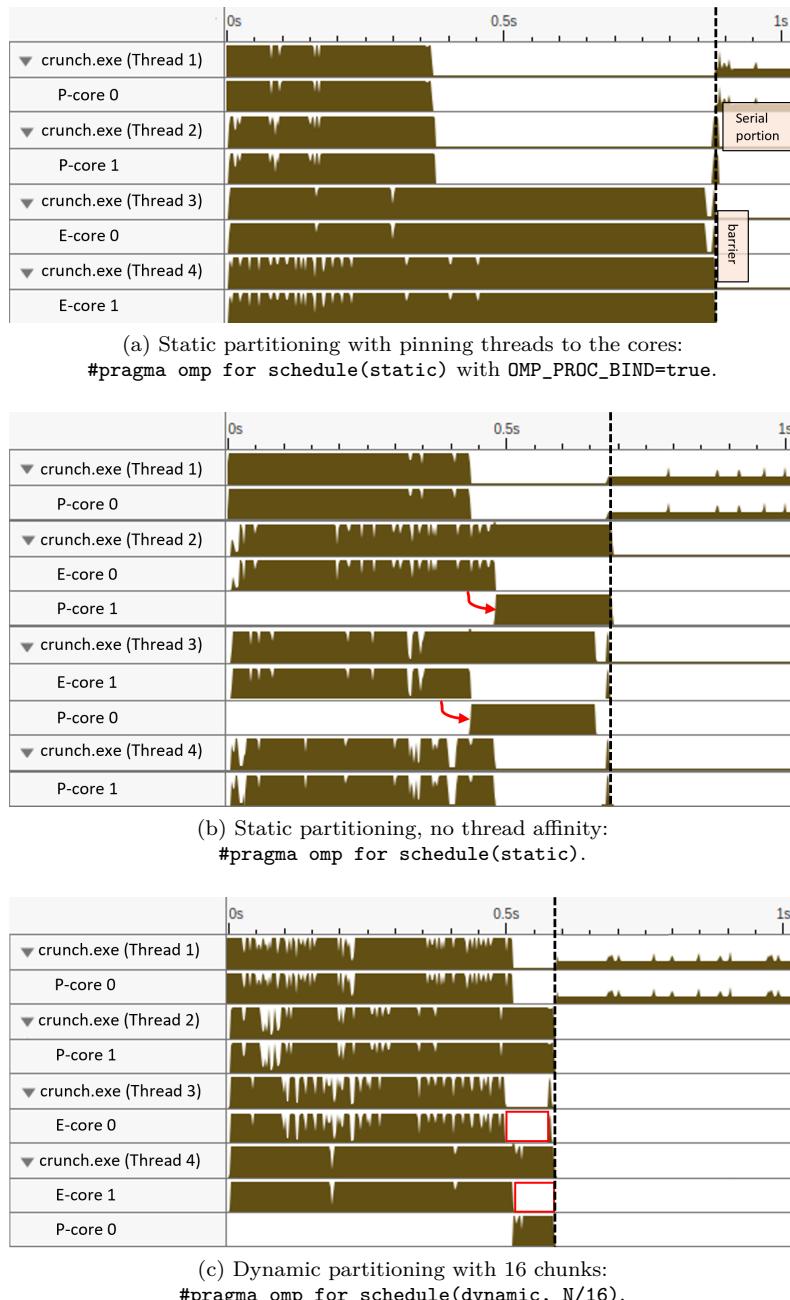


Figure 13.9: Typical task scheduling pitfalls: core affinity blocks thread migration, partitioning jobs with large granularity fails to maximize CPU utilization.

P-cores became available, they migrated. It solved the problem we had before, but now E-cores remain idle until the end of the processing.

My second piece of advice is to avoid static partitioning on systems with asymmetric cores. Equal-sized chunks will likely be processed faster on P-cores than on E-cores which will introduce dynamic load imbalance.

In the final example, I switch to using dynamic partitioning. With dynamic partitioning, chunks are distributed to threads dynamically. Each thread processes a chunk of elements, then requests another chunk, until no chunks remain to be distributed. Figure 13.9c shows the result of using dynamic partitioning by dividing the array into 16 chunks. With this scheme, each task becomes more granular, which enables OpenMP runtime to balance the work even when P-cores run two times faster than E-cores. However, notice that there is still some idle time on E-cores.

Performance can be slightly improved if we partition the work into 128 chunks instead of 16. But don't make the jobs too small, otherwise it will result in increased management overhead. The result summary of my experiments is shown in Table 13.2. Partitioning the work into 128 chunks turns out to be the sweet spot for our example. Even though this example is very simple, lessons from it can be applied to production-grade multithreaded software.

Table 13.2: Results of the task scheduling experiments.

	Affinity	Static	Dynamic, 4 chunks	Dynamic, 16 chunks	Dynamic, 128 chunks	Dynamic, 1024 chunks
Avg latency of 10 runs, ms		864	567	570	541	517

13.4 Cache Coherence

Multiprocessor systems incorporate means to ensure data coherence during shared usage of memory by each core containing its own, separate cache entity. Without such a protocol, if both CPU A and B read memory location L into their individual caches, and CPU B subsequently modifies its cached value for L, then the CPUs would have incoherent values of the same memory location L. Cache Coherency Protocols ensure that any updates to cached entries are dutifully updated or invalidated in any other cached entry of the same location.

13.4.1 Cache Coherency Protocols

One of the most well-known cache coherency protocols is MESI (Modified Exclusive Shared Invalid), 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 13.10):

- **Modified:** a cache line is present only in the current cache and has been modified from its value in RAM

- **Exclusive:** a cache line is present only in the current cache and matches its value in RAM
- **Shared:** a cache line is present here and in other cache lines and matches its value in RAM
- **Invalid:** a cache line is unused (i.e., does not contain any RAM location)

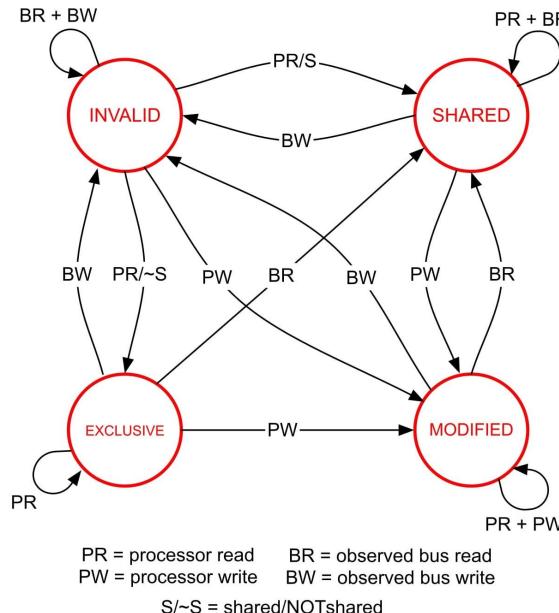


Figure 13.10: MESI States Diagram. © Source: University of Washington via courses.cs.washington.edu.

When fetched from memory, each cache line has one of the states encoded into its tag. Then the cache line state keeps transitioning from one state to another.²⁶⁰ In reality, CPU vendors usually implement slightly improved variants of MESI. For example, Intel uses **MESIF**,²⁶¹ which adds a Forwarding (F) state, while AMD employs **MOESI**,²⁶² which adds the Owning (O) state. However, these protocols still maintain the essence of the base MESI protocol.

Lack of cache coherency can cause sequentially inconsistent programs. This problem can be mitigated by having *snoop* caches watch all memory transactions and cooperate with each other to maintain memory consistency. Unfortunately, it comes with a cost since modification done by one core invalidates the corresponding cache line in another core'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 13.2. Two widely known types of coherency problems are *true sharing* and

²⁶⁰ There is an animated demonstration of the MESI protocol - <https://www.scss.tcd.ie/Jeremy.Jones/vivio/caches/MESI.htm>.

²⁶¹ MESIF - https://en.wikipedia.org/wiki/MESIF_protocol

²⁶² MOESI - https://en.wikipedia.org/wiki/MOESI_protocol

false sharing, which we will explore next.

13.4.2 True Sharing

True sharing occurs when two different cores access the same variable (see Listing 13.1).

Listing 13.1 True Sharing Example.

```
unsigned int sum; // shared between all threads
{ // code executed by thread A
    for (int i = 0; i < N; i++)
        sum += a[i];
}
{ // code executed by thread B
    for (int i = 0; i < N; i++)
        sum += b[i];
}
```

First of all, we have a bigger problem here besides true sharing. We actually have a *data race*, which sometimes can be quite tricky to detect. Notice, that we don't have a proper synchronization mechanism in place, which can lead to unpredictable or incorrect program behavior, because the operations on the shared data might interfere with one another. Fortunately, there are tools that can help identify such issues. [Thread sanitizer²⁶³](#) from Clang and [helgrind²⁶⁴](#) are among such tools. To prevent the data race in Listing 13.1, you should declare the `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. A better way of solving our true sharing issue 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 13.1 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.

13.4.3 False Sharing

If not careful, you may attempt to solve the true sharing issue as shown in Listing 13.2. This solution introduces another problem: *false sharing*. It occurs when two different cores modify different variables that happen to reside on the same cache line. In the code sample shown in Listing 13.2, even though threads A and B update different fields of struct `S`, they are very likely to reside on the same cache line, which will trigger a false sharing issue. Figure 13.11 illustrates this problem.

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. For applications that experience true/false sharing, TMA will likely show a high [Memory Bound → L3 Bound → Contested Accesses](#) metric.²⁶⁵

When using Intel VTune Profiler, I recommend running two types of analysis to find and eliminate false sharing issues. First, run a *Microarchitecture Exploration* analysis

²⁶³ Clang's thread sanitizer tool: <https://clang.llvm.org/docs/ThreadSanitizer.html>.

²⁶⁴ Helgrind, a thread error detector tool: <https://www.valgrind.org/docs/manual/hg-manual.html>.

²⁶⁵ See the Intel VTune user guide for a description of the *Contested Accesses* metric.

Listing 13.2 False Sharing Example.

```

struct S {
    int sumA; // sumA and sumB are likely to
    int sumB; // reside in the same cache line
};

S s;

{ // code executed by thread A
    for (int i = 0; i < N; i++)
        s.sumA += a[i];
}

{ // code executed by thread B
    for (int i = 0; i < N; i++)
        s.sumB += b[i];
}

```

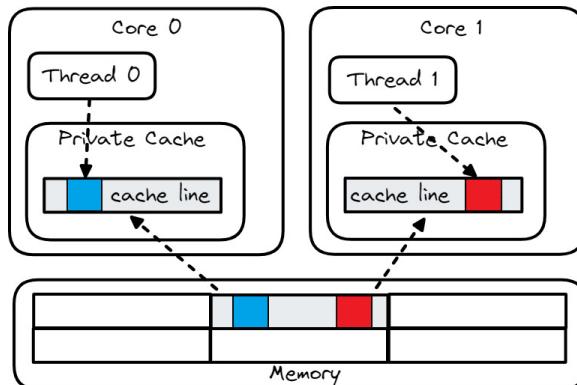


Figure 13.11: False Sharing: two threads access the same cache line.

that implements TMA methodology to detect the presence of false sharing in an 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* checkbox enabled. This analysis helps in finding out memory 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 in Intel Developer Zone.²⁶⁶

Linux `perf` has support for finding false sharing as well. As with the Intel VTune profiler, run TMA first (see Section 6.1.1) to find out if the program experiences false/true sharing issues. If that's the case, use the `perf c2c` tool to detect memory accesses with high cache coherency costs. `perf c2c` matches store/load addresses for different threads and checks 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.²⁶⁷

It is possible to eliminate false sharing with the help of aligning/padding memory objects. Example in Section 13.4.2 can be fixed by ensuring `sumA` and `sumB` do not share the same cache line as shown in Listing 13.3.²⁶⁸

²⁶⁶ VTune cookbook: false-sharing - <https://software.intel.com/en-us/vtune-cookbook-false-sharing>.

²⁶⁷ An article on `perf c2c` - <https://joemario.github.io/blog/2016/09/01/c2c-blog/>.

²⁶⁸ Do not take the size of a cache line as a constant value. For example, in Apple processors such as M1, M2, and later, the L2 cache operates on 128B cache lines.

Listing 13.3 Data padding to avoid false sharing.

```

constexpr int CacheLineAlign = 64;
struct S {
    int sumA;           => int sumA;
    int sumB;           alignas(CacheLineAlign) int sumB;
};                                };

```

False sharing can not only be observed in native languages, like C and C++, but also in managed ones, like Java and C#. 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".²⁶⁹

13.5 Advanced Analysis Tools

Many tools have been developed to address specific use cases where traditional profilers don't provide enough visibility. In this section, we will introduce Coz and eBPF tools. We encourage you to do further research on these and other tools.

13.5.1 Coz

In Section 13.2, we defined the challenge of identifying parts of code that affect the overall performance of a multithreaded program. Due to various reasons, optimizing one part of a multithreaded program might not always give visible results. Traditional sampling-based profilers only show code places where most of the time is spent. However, it does not necessarily correspond to where programmers should focus their optimization efforts.

Coz²⁷⁰ is a profiler that addresses this problem. 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. Also, Coz quantifies the potential impact of an optimization. [Curtsinger & Berger, 2018]

An example of applying the Coz profiler to the C-Ray²⁷¹ benchmark is shown in Figure 13.12. 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 the C-Ray benchmark overall of about 17%. Once we reach ~45% improvement on

²⁶⁹ Blog post "Diving Deeper into Cache Coherency" - <http://psy-lab-saw.blogspot.com/2013/09/diving-deeper-into-cache-coherency.html>

²⁷⁰ COZ source code - <https://github.com/plasma-umass/coz>.

²⁷¹ C-Ray benchmark - <https://github.com/jtsiomb/c-ray>.

that line, the impact on the application begins to level off by Coz's estimation. For more details on this example, see the article²⁷² on the Easyperf blog.

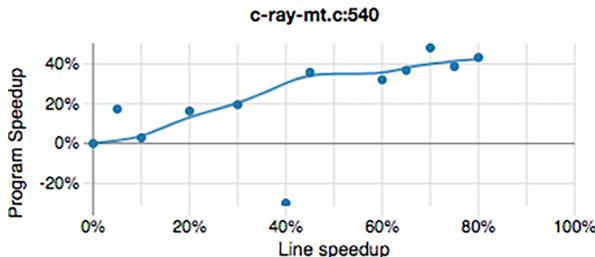


Figure 13.12: Coz profile for the C-Ray benchmark.

13.5.2 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 and profiling tools that make this possible, none more powerful than Extended Berkeley Packet Filter²⁷³ (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 compiler²⁷⁴ in preparation for loading into the kernel virtual machine. These BPF programs can 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 open-source community has provided many eBPF programs for general use. One such tool is the Generic Automatic Parallel Profiler (GAPP),²⁷⁵ which helps to track multithreaded contention issues. GAPP uses eBPF to track the contention overhead of a multithreaded application by ranking the criticality of identified serialization bottlenecks and it 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 ones in Parsec 3.0 Benchmark Suite²⁷⁶ and some large open-source projects. [Nair & Field, 2020]

As a closing thought, I would like to reemphasize the importance of optimizing multithreaded applications. From everything we have discussed in this book, advice from this chapter may bring the most significant performance improvements. In

²⁷² Blog article “COZ vs Sampling Profilers” - <https://easyperf.net/blog/2020/02/26/coz-vs-sampling-profilers>.

²⁷³ eBPF docs - <https://prototype-kernel.readthedocs.io/en/latest/bpf/>

²⁷⁴ BCC compiler - <https://github.com/iovisor/bcc>

²⁷⁵ GAPP - <https://github.com/RN-dev-repo/GAPP/>

²⁷⁶ Parsec 3.0 Benchmark Suite - <https://parsec.cs.princeton.edu/index.htm>

multithreaded applications, the devil is in the details. A subtle synchronization issue or a small inefficiency in data sharing can lead to significant performance degradation. As we look to the future, the trend toward many-core processors and parallel workloads will only accelerate. The complexity of multithreaded optimization will grow, but so will the opportunities for those who master it.

Questions and Exercises

1. Solve the `perf-ninja::false_sharing` lab assignment.
2. What are the benefits of multithreaded vs. multiprocessed applications?
3. What is the essence of Amdahl's Law and Universal Scalability Law?
4. Run the application that you're working with daily. Is it multithreaded? If not, pick one of the popular multithreaded benchmarks. Run the thread count scaling study and analyze the results. Generate a diagram that visualizes worker threads on a timeline. Can you identify any scheduling issues? Identify hot locks and which code paths lead to those locks. Can you improve locking?

Chapter Summary

- Applications that do not take 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 your application.
- When dealing with a single-threaded application, optimizing one portion of the program usually yields positive results on performance. However, this is 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.
- Thread communication can yield retrograde speedup as explained by Universal Scalability Law. This poses additional challenges for tuning multithreaded programs.
- As we saw in our thread count case study, frequency throttling, memory bandwidth saturation, and other issues can lead to poor performance scaling.
- Task scheduling on hybrid processors is challenging. Watch out for suboptimal job scheduling and do not restrict the OS scheduler when it is not necessary.
- Optimizing the performance of multithreaded applications also involves detecting and mitigating the effects of cache coherence, such as true sharing and false sharing.
- During the past years, new tools emerged that cover gaps in analyzing performance of multithreaded applications, that traditional profilers cannot cover. We introduced Coz and GAPP which have a unique set of features.

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 would 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:

- Modern software is massively inefficient. There are significant optimization opportunities to reduce carbon emissions and make a better user experience. People hate using slow software, especially when their productivity goes down because of it. Not all fast software is world-class, but all world-class software is fast. Performance is *the* killer feature.
- Single-threaded CPU performance is not increasing as rapidly as it used to a few decades ago. When it's no longer the case that each hardware generation provides a significant performance boost, developers should start optimizing the code of their software.
- For many years performance engineering was a nerdy niche. But now it is mainstream as software vendors have realized the impact that their poorly optimized software has on their bottom line. Performance tuning is 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.
- The importance of low-level performance tuning should not be underestimated, even if it's just a 1% improvement. The cumulative effect of these small improvements is what makes a difference.
- There is a famous quote by Donald Knuth: "Premature optimization is the root of all evil".[Knuth, 1974] 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 products. Save your project by integrating automated performance benchmarking into your CI/CD pipeline. *Measure early, measure often*.
- Knowledge of the CPU microarchitecture is required to reach peak performance. However, your mental model can never be as accurate as the actual microarchitecture design of a CPU. So don't solely rely on your intuition when you make a specific change in your code. Predicting the performance of a particular piece of code is nearly impossible. *Always measure!*
- When measuring performance, understand the underlying technical reasons for the performance results you observe. Always measure one level deeper and collect as many metrics as possible to support your conclusions.
- Performance engineering 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 hardware and software) available to you. I strongly suggest embracing the Top-down Microarchitecture Analysis (TMA) methodology. It will help you steer your work in the right direction.
- When you identify the performance-limiting factor of your application, you are

more than halfway through. Based on my experience, the fix is often easier than finding the root cause of the problem. In Part 2 we covered some essential optimizations for every type of CPU performance bottleneck: how to optimize memory accesses and computations, how to get rid of branch mispredictions, how to improve machine code layout, and several others. Use chapters from Part 2 as a reference to see what options are available when your application has one of these problems.

- Processors from different vendors are not created equal. They differ in terms of instruction set architecture (ISA) supported and microarchitectural implementation. Reaching peak performance on a given platform requires utilizing the latest ISA extensions, avoiding common microarchitecture-specific issues, and tuning your code according to the strengths of a particular CPU microarchitecture.
- Multithreaded programs add one more dimension of complexity to performance tuning. They introduce new types of bottlenecks and require additional tools and methods to analyze and optimize. Examining how an application scales with the number of threads is an effective way to identify bottlenecks in multithreaded programs.

I hope you now have a better understanding of low-level performance optimizations. Of course, this book doesn't cover every possible scenario you may encounter in your daily job. 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. I wish you experience the joy of discovering performance bottlenecks in your application and the satisfaction of fixing them.

Happy performance tuning!

I will post errata and other information about the book on my blog at the following URL: <https://easypf.net/blog/2024/11/11/Book-Updates-Errata>.

If you haven't solved the `perf-ninja` exercises yet, I encourage you to take the time to do so. They will help you to solidify your knowledge and prepare you for real-world performance engineering challenges.

P.S. If you enjoyed reading this book, make sure to pass it on to your friends and colleagues. I would appreciate your help in spreading the word about the book by endorsing it on social media platforms.

Acknowledgments

I write this section with profound gratitude to all the people mentioned below. I feel very humble to deliver to you all the knowledge these people have.

Mark E. Dawson, Jr. authored Section 12.3 “Low Latency Tuning Techniques” and Section 7.9 “Continuous Profiling”. He has also contributed a lot to the first edition of the book. Mark is a recognized expert in the High-Frequency Trading industry, currently working at WH Trading. Mark also has a blog (<https://www.jabperf.com>) where he writes about low latency and other performance optimizations.



Agustín Navarro-Torres, Jesús Alatruey-Benedé, Pablo Ibáñez-Marín, Víctor Viñals-Yúfera from the University of Zaragoza authored Section 12.5 “Case Study: Sensitivity to Last Level Cache Size”. They are researchers in the field of computer science and have published multiple papers on topics related to performance engineering.



Jan Wassenberg authored Section 9.5.1 “Wrapper Libraries for Intrinsics” and also proposed many improvements to Section 9.4 “Vectorization” and Section 3.4 “SIMD Multiprocessors”. Jan is a software engineer at Google DeepMind, where he leads the development of Gemma.cpp.²⁷⁷ He also has authored multiple research papers. His personal webpage is <https://wassenberg.dreamhosters.com>.



Swarup Sahoo helped me with writing about AMD PMU features in Chapter 6, and authored Section 7.1 about AMD uProf. Swarup is a senior developer at AMD, where he works on the uProf performance analysis tool for HPC (OpenMP, MPI) applications. Swarup’s LinkedIn page can be found at <https://www.linkedin.com/in/swarupsahoo>.



Alois Kraus authored Section 7.6 “Event Tracing for Windows”, and Appendix D. He developed **ETWAnalyzer**, a command-line tool for analyzing ETW files with simple queries. He is employed by Siemens Healthineers where he studies the performance of large software systems. Alois’ personal webpage and blog: <https://alokraus.wordpress.com>.



²⁷⁷ Gemma.cpp, LLM inference on CPU - <https://github.com/google/gemma.cpp>

Marco Castorina authored Section 7.7 “Specialized and Hybrid Profilers” which showcases performance profiling with Tracy. Marco currently works on the games graphics performance team at AMD, focusing on DX12. Also, he is the co-author of a book titled “Mastering Graphics Programming with Vulkan”. Marco’s personal web page is <https://marcocastorina.com>.



Lally Singh has authored Section 5.3.2 about Marker APIs. Lally is currently at Tesla, his prior work includes Datadog’s performance team, Google’s Search performance team, low-latency trading systems, and embedded real-time control systems. Lally has a PhD in CS from Virginia Tech, focusing on scalability in distributed VR.



Richard L. Sites provided a technical review of the book. He is a veteran of the semiconductor industry and has spent most of his career at the boundary between hardware and software, particularly in CPU/software performance interactions. Richard invented the performance counters found in most modern processors. He had worked at DEC, Adobe, Google, and Tesla. His personal page is <https://sites.google.com/site/dicksites>.



Matt Godbolt provided a technical review of the book. Matt is a creator of Compiler Explorer, an extremely popular tool among software developers. He is a C++ developer passionate about high-performance code. Matt has over twenty years of professional experience in computer game programming, systems design, real-time embedded systems, and high-frequency trading. He is also a speaker, a blogger, and a podcaster. His personal blog is <https://xania.org>.



Also, I would like to thank the following people. Jumana Mundichipparakkal from Arm, for helping me write about Arm PMU features in Chapter 6. Yann Collet, the author of Zstandard, for providing me with the information about the internal workings of Zstd for Section 13.2.1. Ciaran McHale, for finding tons of grammar mistakes in my initial draft. Nick Black for proofreading and editing the final version of the book. Peter Veentjer, Amir Aupov, and Charles-Francois Natali for various edits and suggestions.

I’m also thankful to 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.

A special “thank you” goes 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.

Images were created with excalidraw.com. Cover design by Darya Antonova. The fonts used on the cover of this book are Bebas Neue and Raleway, both provided under the Open Font License. Bebas Neue was designed by Ryoichi Tsunekawa. Raleway was designed by Matt McInerney, Pablo Impallari, and Rodrigo Fuenzalida.

I should also mention contributors to the first edition of this book. Below I only list names and section titles. More detailed acknowledgments are available in the first edition.

- Mark E. Dawson, Jr. wrote sections Section 8.4 “Optimizing For DTLB”, Section 11.8 “Optimizing for ITLB”, Section 12.3.2 “Cache Warming”, Section 12.4 “System Tuning”, and a few other.
- Sridhar Lakshmanamurthy, authored a large part of Chapter 3 about CPU microarchitecture.
- Nadav Rotem helped write Section 9.4 about vectorization.
- Clément Grégoire authored Section 9.4.2.5 about ISPC compiler.
- 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.

Support This Book

If you enjoyed this book and would like to support it, there are a few options.

Paypal (one-time)



Buy Me a Coffee (one-time)



GitHub Sponsors
(one-time or monthly)



Patreon (monthly)



Glossary

AOS Array Of Structures	LBR Last Branch Record
BB Basic Block	LLC Last Level Cache
BIOS Basic Input Output System	LSD Loop Stream Detector
CI/CD Contiguous Integration/ Contiguous Development	LTO Link-Time Optimizations
CFG Control Flow Graph	MPI Message Passing Interface
CP Continuous Profiling	MSR Model Specific Register
CPI Clocks Per Instruction	MS-ROM Microcode Sequencer Read-Only Memory
CPU Central Processing Unit	NUMA Non-Uniform Memory Access
DDD Data-Driven Development	OOO Execution Out-of-Order Execution
DSB Decoded Stream Buffer	OS Operating System
DRAM Dynamic Random-Access Memory	PEBS Processor Event-Based Sampling
DTLB Data Translation Lookaside Buffer	PDB files Program-Debug Data Base files
EBS Event-Based Sampling	PGO Profile-Guided Optimizations
EHP Explicit Huge Pages	PMC Performance Monitoring Counter
FLOPS Floating-point Operations Per Second	PMI Performance Monitoring Interrupt
FMA Fused Multiply Add	PMU Performance Monitoring Unit
FPGA Field-Programmable Gate Array	PT Processor Traces
GPU Graphics processing unit	RAT Register Alias Table
HFT High-Frequency Trading	RNG Random Number Generator
HPC High Performance Computing	ROB ReOrder Buffer
I/O Input/Output	SIMD Single Instruction Multiple Data
IDE Integrated Development Environment	SMT Simultaneous MultiThreading
ILP Instruction-Level Parallelism	SOA Structure Of Arrays
IPC Instructions Per Clock cycle	TLB Translation Lookaside Buffer
IPO Inter-Procedural Optimizations	TMA Top-down Microarchitecture Analysis
ITLB Instruction Translation Lookaside Buffer	TSC Time Stamp Counter
	μop 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 13.3: List of the 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
Lion Cove	LNC	2024	AVX2

Table 13.4: List of the recent AMD microarchitectures.

Name	Year released	Supported ISA
Steamroller	2014	AVX
Excavator	2015	AVX2
Zen	2017	AVX2
Zen2	2019	AVX2
Zen3	2020	AVX2
Zen4	2022	AVX512
Zen5	2024	AVX512

Table 13.5: List of recent ARM ISAs along with their own and third-party implementations.

ISA	Year of ISA release	Arm uarchs (latest)	Third-party uarchs
ARMv8-A	2011	Cortex-A73	Apple A7-A10; Qualcomm Kryo; Samsung M1/M2/M3
ARMv8.2-A	2016	Neoverse N1; Cortex-X1	Apple A11; Samsung M4; Ampere Altra
ARMv8.4-A	2017	Neoverse V1	AWS Graviton3; Apple A13, M1
ARMv9.0-A (64bit-only)	2018	Neoverse N2; Neoverse V2; Cortex X3	Microsoft Cobalt 100; NVIDIA Grace; AWS Graviton4;
ARMv8.6-A (64bit-only)	2019	—	Apple A15, A16, M2, M3
ARMv9.2-A	2020	Cortex X4	Apple M4

References

- [Advanced Micro Devices, 2021] Advanced Micro Devices (2021). *Processor Programming Reference (PPR) for AMD family 19h model 01h (55898)*. B1 Rev 0.50. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/programmer-references/55898_B1_pub_0_50.zip
- [Advanced Micro Devices, 2022] Advanced Micro Devices (2022). *AMD64 technology platform quality of service extensions*. Pub. 56375, rev 1.01. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/other/56375_1_03_PUB.pdf
- [Advanced Micro Devices, 2023] Advanced Micro Devices (2023). *Software Optimization Guide for the AMD Zen4 Microarchitecture*. Rev 1.00. <https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/software-optimization-guides/57647.zip>
- [Akinshin, 2019] Akinshin, A. (2019). *Pro .NET Benchmarking* (1 ed.). Apress. <https://doi.org/10.1007/978-1-4842-4941-3>
- [Allen & Kennedy, 2001] Allen, R. & Kennedy, K. (2001). *Optimizing Compilers for Modern Architectures: A Dependence-based Approach* (1 ed.). Morgan-Kaufmann.
- [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>
- [AMD, 2024] AMD (2024). *AMD uProf User Guide, Revision 4.2*. Advanced Micro Devices, Inc. <https://www.amd.com/content/dam/amd/en/documents/developer/version-4-2-documents/uprof/uprof-user-guide-v4.2.pdf>
- [Apple, 2024] Apple (2024). *Apple Silicon CPU Optimization Guide: 3.0*. Apple® Inc. <https://developer.apple.com/documentation/apple-silicon/cpu-optimization-guide>
- [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 Limited. <https://developer.arm.com/documentation/PJDOC-1063724031-605393/2-0/?lang=en>
- [Arm, 2023a] Arm (2023a). *Arm Neoverse V1 Core: Performance Analysis Methodology*. Arm Limited. <https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/neoverse-v1-core-performance-analysis.pdf>
- [Arm, 2023b] Arm (2023b). *Arm Statistical Profiling Extension: Performance Analysis Methodology*. Arm Limited. <https://developer.arm.com/documentation/109429/latest/>

- [Blem et al., 2013] Blem, E., Menon, J., & Sankaralingam, K. (2013). Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures. *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 1–12. <https://doi.org/10.1109/HPCA.2013.6522302>
- [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>
- [ChipsAndCheese, 2024] ChipsAndCheese (2024). *Why x86 doesn't need to die*. <https://chipsandcheese.com/2024/03/27/why-x86-doesnt-need-to-die/>
- [Cooper & Torczon, 2012] Cooper, K. & Torczon, L. (2012). *Engineering a Compiler*. Morgan Kaufmann. Morgan Kaufmann. <https://books.google.co.in/books?id=CGT0IAEACAAJ>
- [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>
- [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, 2023a] Fog, A. (2023a). 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>
- [Fog, 2023b] Fog, A. (2023b). *Optimizing software in c++: An optimization guide for windows, linux and mac platforms*. https://www.agner.org/optimize/optimizing_cpp.pdf
- [Geelnard, 2022] Geelnard, M. (2022). *Debunking cisc vs risc code density*. <https://www.bitsnbites.eu/cisc-vs-risc-code-density/>
- [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.

- [Herdrich et al., 2016] Herdrich, A. et al. (2016). Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family. *HPCA*, 657–668. <https://ieeexplore.ieee.org/document/7446102>
- [Intel, 2023] Intel (2023). *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>
- [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/10.1145/287288.7.2750392>
- [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>
- [Knuth, 1974] Knuth, D. E. (1974). Structured programming with go to statements. *ACM Comput. Surv.*, 6, 261–301. <https://api.semanticscholar.org/CorpusID:207630080>
- [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>
- [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>
- [Marr, 2023] Marr, D. (2023). *Keynote: Architecting for power-efficiency in general-purpose computing*. <https://youtu.be/IktNjMxJYPE?t=2599>
- [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! *SIGPLAN Not.*, 44(3), 265–276. <https://doi.org/10.1145/1508284.1508275>
- [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>
- [Navarro-Torres et al., 2023] Navarro-Torres, A., Alastruey-Benedé, J., Ibáñez, P., & Viñals-Yúfera, V. (2023). Balancer: bandwidth allocation and cache partitioning for multicore processors. *The Journal of Supercomputing*, 79(9), 10252–10276. <https://doi.org/10.1007/s11227-023-05070-0>
- [Navarro-Torres et al., 2019] Navarro-Torres, A. et al. (2019). Memory hierarchy characterization of SPEC CPU2006 and SPEC CPU2017 on the Intel Xeon Skylake-SP. *PLOS ONE*, 1–24. <https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0220135>
- [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>
- [Ousterhout, 2018] Ousterhout, J. (2018). Always measure one level deeper. *Commun. ACM*, 61(7), 74–83. <https://doi.org/10.1145/3213770>
- [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>
- [Shen & Lipasti, 2013] Shen, J. & Lipasti, M. (2013). *Modern Processor Design: Fundamentals of Superscalar Processors*. Waveland Press, Inc.
- [Sites, 2022] Sites, R. (2022). *Understanding Software Dynamics*. Addison-Wesley professional computing series. Addison-Wesley. <https://books.google.com/books?id=TklozgEACAAJ>
- [statista.com, 2024] statista.com (2024). *Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2020, with forecasts from 2021 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>
- [Wang et al., 2017] Wang, X. et al. (2017). SWAP: effective fine-grain management of shared last-level caches with minimum hardware support. *HPCA*, 121–132. <https://ieeexplore.ieee.org/document/7920819>
- [Weaver & McIntosh-Smith, 2023] Weaver, D. & McIntosh-Smith, S. (2023). An empirical comparison of the risc-v and aarch64 instruction sets. *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, 1557–1565. <https://doi.org/10.1145/3624062.3624233>
- [Williams et al., 2009] Williams, S., Waterman, A., & Patterson, D. (2009). Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4), 65–76. <https://doi.org/10.1145/1498765.1498785>
- [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 and a few techniques to reduce noise. I provided an introduction to the topic in Section 2.1.

This section is mostly specific to the Linux operating system. Readers are encouraged to search the web for instructions on how to configure other operating systems.

Dynamic Frequency Scaling

Dynamic Frequency Scaling²⁷⁸ (DFS) is a technique to increase the performance of a 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, and AMD CPUs employ Turbo Core functionality.

Here is an example of the 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 higher when Turbo Boost is on (2.7 GHz vs. 2.3 GHz).

DFS can be permanently disabled in BIOS. 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

Many modern CPU cores support simultaneous multithreading (see Section 3.5.2). SMT can be permanently disabled in BIOS. To programmatically disable SMT on Linux systems, you need root access. The sibling pairs of CPU threads can be found in the following files:

```
/sys/devices/system/cpu/cpuN/topology/thread_siblings_list
```

²⁷⁸ Dynamic frequency scaling - https://en.wikipedia.org/wiki/Dynamic_frequency_scaling.

Here is how you can disable a sibling thread of core 0 on Intel® Core™ i5-8259U, which has 4 cores and 8 threads:

```
# all 8 hardware 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
```

Also, the `lscpu --all --extended` command can be very helpful to see the sibling threads.

Scaling Governor

Linux kernel can control CPU frequency for different purposes. One such purpose is to save power. In this case, the scaling governor may decide to decrease the CPU 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:

```
echo performance | sudo tee /sys/devices/system/cpu/cpufreq/policy*/scaling_governor
```

CPU Affinity

Processor affinity²⁷⁹ enables the binding of a process to a certain CPU core(s). In Linux, you can do this with `taskset`²⁸⁰ tool.

```
# no affinity
$ perf stat -e context-switches,cpu-migrations -r 10 -- a.exe
      151      context-switches
      10      cpu-migrations

# 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
```

²⁷⁹ Processor affinity - https://en.wikipedia.org/wiki/Processor_affinity.

²⁸⁰ `taskset` manual - <https://linux.die.net/man/1/taskset>.

Notice the number of `cpu-migrations` gets down to 0, i.e., the process never leaves `core0`.

Alternatively, you can use `cset`²⁸¹ 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>
```

On Windows, a program can be pinned to a specific core using the following command:

```
$ start /wait /b /affinity 0xC0 myapp.exe
```

where the `/wait` option waits for the application to terminate, `/b` starts the application without opening a new command window, and `/affinity` specifies the CPU affinity mask. In this case, the mask `0xC0` means that the application will run on cores 6 and 7.

On macOS, it is not possible to pin threads to cores since the operating system does not provide an API for that.

Process Priority

In Linux, you can increase process priority using the `nice` tool. By increasing priority, the process gets more CPU time, and the 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, that the execution of the benchmarked process was interrupted by the OS more than 100 times. If we increase process priority by running 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.

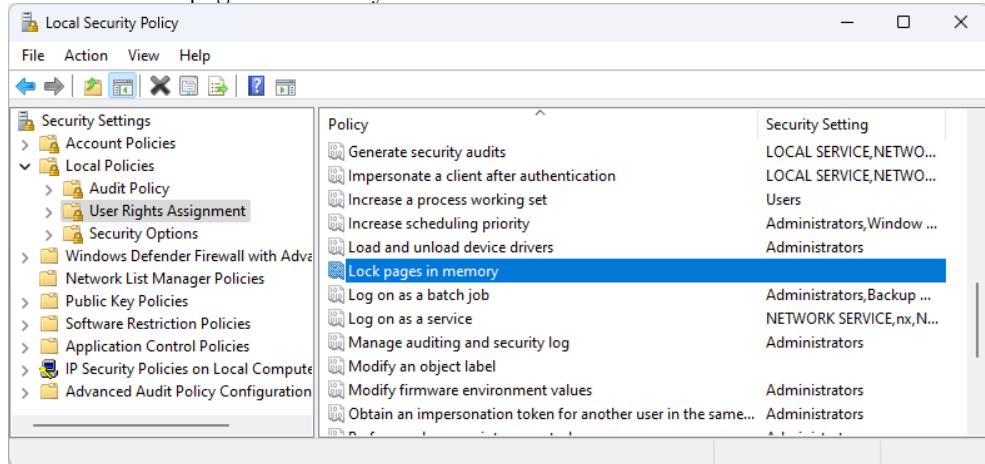
²⁸¹ `cpuset` manual - <https://github.com/lpechacek/cpuset>.

Appendix B. Enable Huge Pages

Windows

To utilize huge pages on Windows, you need to enable [SeLockMemoryPrivilege security policy](#). This can be done programmatically via the Windows API, or alternatively via the security policy GUI.

1. Hit start → search “secpol.msc”, and 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);
```

Linux

On Linux OS, there are two ways of using huge pages in an application: Explicit and Transparent Huge Pages.

Explicit Huge Pages

Explicit huge pages can be reserved at system boot time or before an application starts. 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 128 huge pages after the system has booted, you can use the following command:

```
$ echo 128 > /proc/sys/vm/nr_hugepages
```

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
```

You can utilize explicit huge pages in the code by calling `mmap` with the `MAP_HUGETLB` flag ([full example²⁸²](#)):

```
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 ([example code²⁸³](#)).
- `shmget` using the `SHM_HUGETLB` flag ([example code²⁸⁴](#)).

Transparent Huge Pages

To allow applications to use Transparent Huge Pages (THP) on Linux you should ensure that `/sys/kernel/mm/transparent_hugepage/enabled` is `always` or `madvise`. The former enables system-wide usage of THPs, while the latter gives control to the user code on which memory regions should use THPs, thus avoiding the risk of consuming more memory resources. Below is an 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_HUGE PAGE);
...
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
```

²⁸² `MAP_HUGETLB` example - https://github.com/torvalds/linux/blob/master/tools/testing/selftests/vm/map_hugetlb.c.

²⁸³ Mounted `hugetlbfs` filesystem - <https://github.com/torvalds/linux/blob/master/tools/testing/selftests/vm/hugepage-mmap.c>.

²⁸⁴ `SHM_HUGETLB` example - <https://github.com/torvalds/linux/blob/master/tools/testing/selftests/vm/hugepage-shm.c>.

Also, you can observe how your application utilizes EHPs and/or THPs by looking at the `smaps` file specific to your process:

```
$ watch -n1 "cat /proc/<PID_OF_PROCESS>/smaps"
```

Appendix C. 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,²⁸⁵ 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 PT 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 13.13. Instructions like PUSH, MOV, ADD, and CMP are ignored because they don't change the control flow. However, the JE instruction may jump to .label, so its result needs to be recorded. Later there is an indirect call for which the destination address is saved.

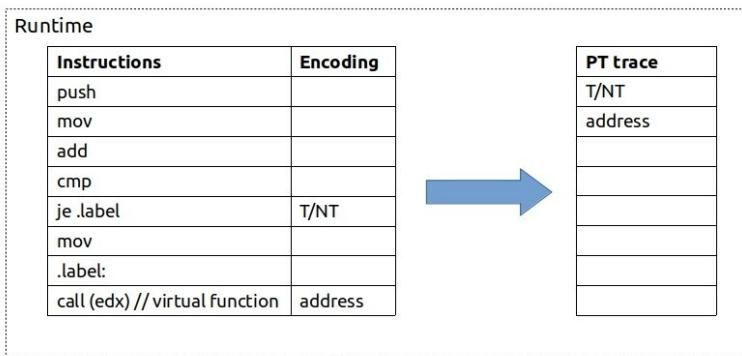


Figure 13.13: Intel Processor Traces encoding

At the time of analysis, we bring together the application binary and the collected PT trace. A software 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 13.14 shows an example of decoding Intel Processor Traces. Suppose that

²⁸⁵ See more information about Intel PT overhead in [Sharma, 2016].

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 software 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 13.14, 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.

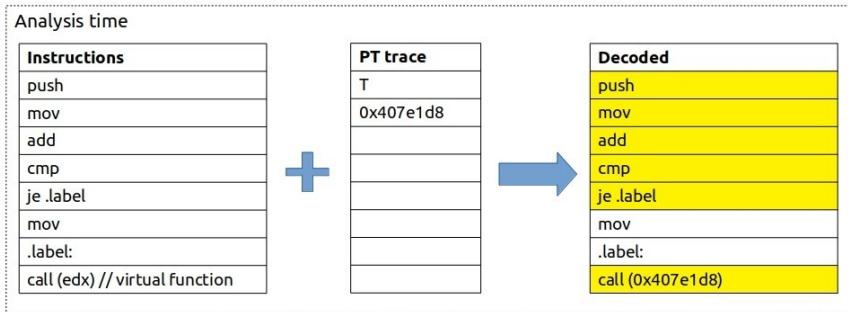


Figure 13.14: Intel Processor Traces decoding

Instructions highlighted in yellow were executed when our program was running. Note that this is an *exact* reconstruction of program execution; we did not skip any instructions. 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.

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 13.15 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 (NT), 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 (CALL(edx)), 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.

In the example shown in Figure 13.15, 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, 2023, Volume 3C, Chapter 36]).

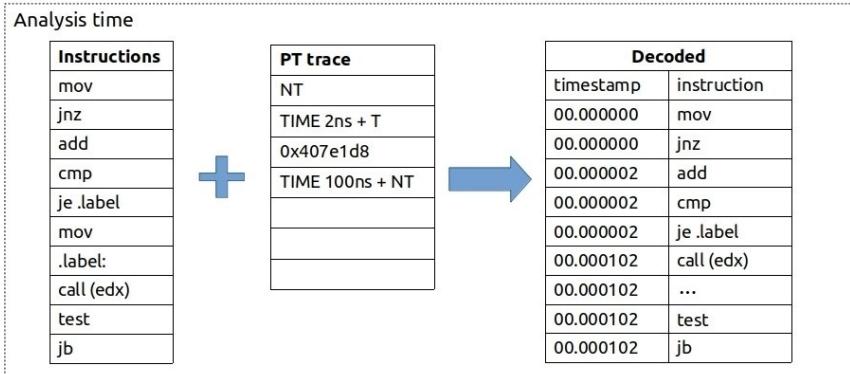


Figure 13.15: Intel Processor Traces timings

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, I 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)
```

The raw PT packets shown above are not very useful for performance analysis. To decode processor traces to human-readable form, you can execute:

```
$ perf script --ns --itrace=iit -F time,srcline,insn,srccode
```

Below is an example of decoded traces:

```
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
...
```

I only show 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 article²⁸⁶ 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.²⁸⁷ 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 patterns.

Disk Space and Decoding Time

Even taking into account the compressed format of the trace, 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 the workload, it's very common for the 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 use on long-running workloads. But it is affordable to run it for a short time, even on a big workload. In this case, the user can attach to the running process just for the 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 because it has to follow along with disassembled instructions from the binary and reconstruct the flow. 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 -F time,ip,sym,symoff,insn` takes ~20 seconds²⁸⁸ and the output consumes ~1.3GB of disk space.

²⁸⁶ Analyze performance glitches with Intel PT - <https://easyperf.net/blog/2019/09/06/Intel-PT-part3>

²⁸⁷ Postmortem debugging with Intel PT - <https://easyperf.net/blog/2019/08/30/Intel-PT-part2>

²⁸⁸ When you decode traces with `perf script -F` with `+srcline` or `+srccode` to emit source code, it gets even slower.

Tools

Besides Linux perf, several other tools support Intel PT. First, Intel VTune Profiler has *Anomaly Detection* analysis type that uses Intel PT. Another popular tool worth mentioning is magic-trace²⁸⁹, which collects and displays high-resolution traces of a process.

Intel PT References and links

- Intel® 64 and IA-32 Architectures Software Developer Manuals [Intel, 2023, 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/intelptmicrotutori al/>.
- 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>.

²⁸⁹ magic-trace - <https://github.com/janestreet/magic-trace>

Appendix D. Event Tracing for Windows Analysis

We provided an introduction to ETW in Section 7.6. In this section, we will continue where we left off, and explore tools to record and analyze ETW. To demonstrate the look-and-feel of these tools, we present a case study of debugging a slow start of a program.

Tools to Record ETW traces

Here is the list of tools you 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**²⁹⁰: a graphical recording and analysis tool with the main focus on .NET Applications. This is an open-source application developed by Microsoft.
- **Performance HUD**²⁹¹: a little-known but very powerful GUI tool to track UI delays, and user/handle leaks via live ETW recording of all unbalanced resource allocations with a live display of leaking/blocking call stack traces.
- **ETWController**²⁹²: a recording tool with the ability to record keyboard input and screenshots along with ETW data. This open-source application, developed by Alois Kraus, also supports distributed profiling on two machines simultaneously.
- **UIforETW**²⁹³: this open-source application, developed by Bruce Dawson, is a wrapper around xperf with special options to record data for Google Chrome issues. It can also record keyboard and mouse input.

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 of how your system behaves and what it is 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/Android²⁹⁴ profiling data that was generated by tools like Linux perf, LTTNG, Perfetto, and several log file formats: dmesg, Cloud-Init, WaLinuxAgent and AndroidLogcat.

²⁹⁰ PerfView - <https://github.com/microsoft/perfview>

²⁹¹ Performance HUD - <https://www.microsoft.com/en-us/download/100813>

²⁹² ETWController - <https://github.com/alois-xx/etwcontroller>

²⁹³ UIforETW - <https://github.com/google/UIforETW>

²⁹⁴ Microsoft Performance Tools Linux / Android - <https://github.com/microsoft/Microsoft-Performance-Tools-Linux-Android>

- **ETWAnalyzer:**²⁹⁵ reads ETW data and generates aggregate summary JSON files that can be queried, filtered, and sorted at the 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.

Case Study - Slow Program Start

Now we will take a look at an 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 the 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 Toolkit²⁹⁶ 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:

```
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. Ensure the check boxes *Record mouse clicks* and *Take cyclic screenshots* are ticked (see Figure 13.16) so later you will be able to navigate to the slow spots with the help of the screenshots.
- Download an application from the internet, and unpack it if needed. It doesn't matter what program you use, the goal is to see the delay when starting it.
- Start profiling by pressing the *Start Recording* button.
- Double-click the executable to start it.
- Once a program has started, stop profiling by pressing the *Stop Recording* button.

Stopping profiling the first time takes a bit longer because Program-Debug Data Base files (PDBs) are generated for all managed code, 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 that is stored in a 4 GB ring buffer, which allows you to record 1-2 minutes before the oldest events are

²⁹⁵ ETWAnalyzer - <https://github.com/Siemens-Healthineers/ETWAnalyzer>

²⁹⁶ Windows SDK Downloads - <https://developer.microsoft.com/en-us/windows/downloads/sdk-archive/>

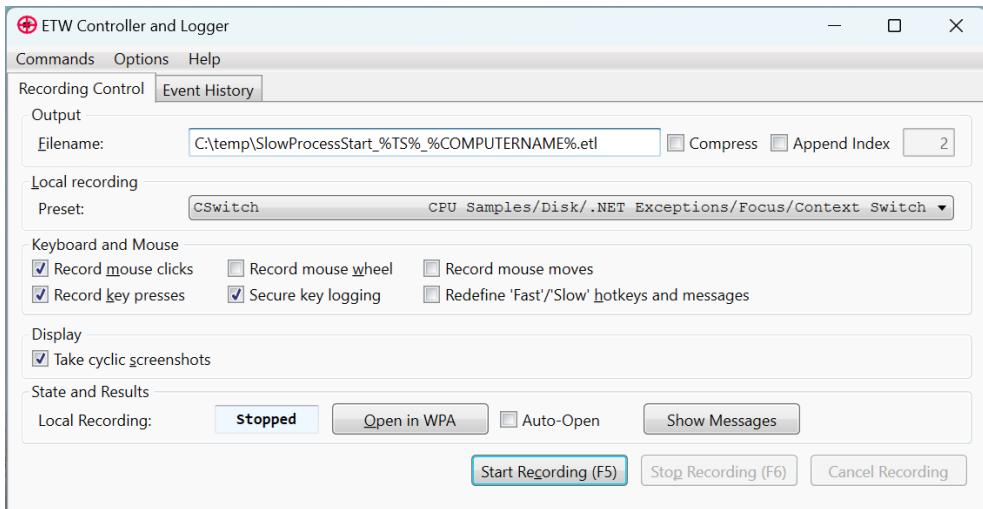


Figure 13.16: Starting ETW collection with ETWController UI.

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.

Windows supports Event Log and Performance Counter triggers that can 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 triggers, you should take a look at PerfView; this enables you to define a Performance Counter threshold that must be reached and stay there for N seconds before profiling is stopped. This way, random spikes will not trigger false positives.

Analysis in WPA

Figure 13.17 shows the recorded ETW data opened in Windows Performance Analyzer (WPA). The WPA view is divided into three vertically layered 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. This *CPU Usage (Sampled)* graph is very similar to the *Hotspots* view in other profiling tools.

Next comes the *Generic Events* view, which displays events such as 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 a different source of data than the *Sampled* view. While sampling data only captures running threads, CSwitch collection takes into account time intervals during which a process was not running. The data for the *CPU Usage (Precise)* view comes from the Windows Thread Scheduler. This

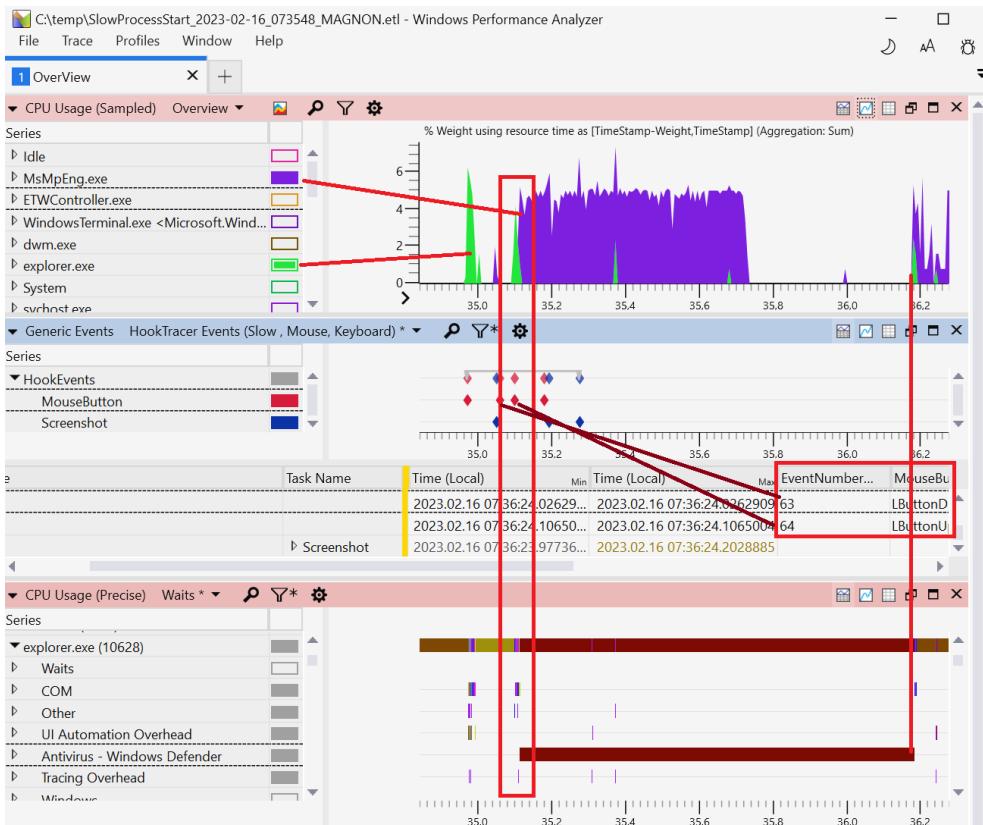


Figure 13.17: Windows Performance Analyzer: root causing a slow start of an application.

graph traces how long, and on which CPU, a thread was running (CPU Usage), how long it was blocked in a kernel call (Waits), in which priority, and how long the thread had been waiting for a CPU to become free (Ready Time), etc. Consequently, the *CPU Usage (Precise)* view doesn't show the top CPU consumers, but this view is very helpful for understanding how long and *why* a certain process was blocked.

Now that we have 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 that you can view in a web browser. Every recorded keyboard/mouse interaction is saved in a file with the event number in its name, e.g., `Screenshot_63.jpg`. Figure 13.18 (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.

The double click marks the beginning of a 1.2-second delay when our application was waiting for something. At timestamp 35.1, `explorer.exe` is active as it attempts to

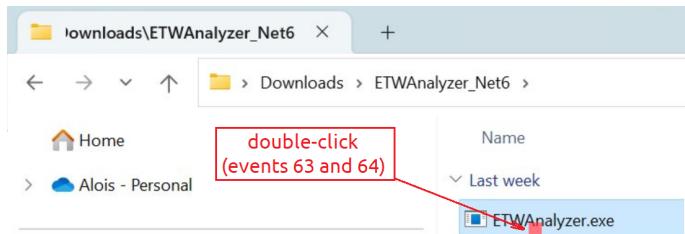


Figure 13.18: A mouse click screenshot captured with ETWController.

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. These are available on the *CPU Usage (Precise)* panel with *Waits* selected in the dropdown menu. 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 corresponding 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 the `CreateProcess` system call is delayed in the kernel by `WDFilter.sys`, the Windows Defender Filter Driver. It blocks the process from starting until the potentially malicious file contents are 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 until 35.7, but the application didn't start immediately after that. There is an additional delay of 0.5 sec from time 35.7 until 36.2, during which the CPU was mostly idle, not doing anything. To find the root cause of this, you would need 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 TCP/IP or socket ETW traces you can also find out with which remote endpoint Microsoft Defender was communicating. 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 issues you can effectively analyze with WPA, but there are many others. The WPA interface is very rich and highly customizable. It supports custom profiles to configure the graphs and tables for visualizing event data in the way you like best. Originally, WPA was developed for

device driver developers and there are built-in profiles that do not focus on application development. ETWController brings its own profile (*Overview.wpaprofile*) that you can set as the default profile under *Profiles* → *Save Startup Profile* to always use the performance overview profile.