

# Performance Analysis and Tuning on Modern CPU

Denis Bakhvalov

---

# Preface

## From The Author

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

I remember the days when I was just starting out with performance analysis. I was staring at unfamiliar metrics trying to the match the data that didn't match. And I was totally confused. I took me years until it finally "clicked" and all pieces of the puzzle came together. At the time, the only good source of information were software developer manuals which are not what mainstream developers like to read. So, I decided to write the book about performance analysis which, hopefully, allow developers to learn it much easier and faster.

Developers who consider themselves as beginners in performance analysis can start from the beginning of the book and read sequentially, chapter by chapter. Chapters 2-4 give developers minimal set of knowledge required by later chapters. Experienced developers can skip those chapters if this is not new to them. Additionally, this book can be used as a reference or a checklist for optimizing SW applications. Developers can use chapters 7,8 as a source of ideas for tuning their code.

## Contributors

Huge thanks to Mark E. Dawson, Jr. for his help writing several sections of this book including sec. 7.3.1.3, sec. 7.5.3, sec. 8.6, sec. 8.7, sec. 7.5.5.

Also, I would like to thank the whole performance community for countless blog articles and papers. I was able to learn a lot from reading blogs by Travis Downs, Daniel Lemire, Andi Kleen, Bruce Dawson, Brendan Gregg and many others. This book is a way to thank and give back to the whole community.

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

## Disclaimers

**Responsibility.** Information in this book should be used in educational purposes only and does not guarantee achieving business goals. Authors of the book do not take any responsibility for the actions taken by the readers as a result of reading this book.

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

**Advertisement.** This book does not advertise any SW, HW or any other product.

# Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
1.1	What Is Performance Analysis?	7
1.2	Why Do We Still Need Performance Tuning?	8
1.3	Who Needs Performance Tuning?	9
1.4	Chapter Summary	10
<b>2</b>	<b>Measuring Performance</b>	<b>11</b>
2.1	Conducting Performance Experiments	11
2.2	Comparing Performance Measurements	13
2.3	Interestingness threshold	15
2.4	Microbenchmarks	17
2.5	Chapter Summary	18
<b>3</b>	<b>CPU microarchitecture 101</b>	<b>19</b>
3.1	Classic ideas in CPU design	19
3.1.1	Pipelining	19
3.1.2	OOO execution	19
3.1.3	Speculation	19
3.1.4	SIMD	19
3.1.5	Simultaneous multithreading (SMT)	19
3.2	Modern CPU design	19
3.2.1	CPU Front-End	19
3.2.2	CPU Back-End	19
3.2.3	CPU Branch Predictor	19
3.2.4	Memory hierarchy	19
3.2.4.1	Hardware memory prefetchers.	20
3.2.5	Performance Monitoring Unit	20
3.2.5.1	Fixed and programmable counters.	21
<b>4</b>	<b>Terminology and metrics in performance analysis</b>	<b>23</b>
4.1	Retired vs. Executed Instruction	23
4.2	CPU Utilization	23
4.3	CPI & IPC	24
4.4	UOP (micro-op)	24
4.5	Pipeline Slot	25
4.6	Core vs. Reference Cycles	25
4.7	Cache miss	26
4.8	Mispredicted branch	27
4.9	Basic Block	27
<b>5</b>	<b>Performance Analysis Approaches</b>	<b>29</b>
5.1	Code Instrumentation	29
5.2	Tracing	30
5.3	Workload Characterization	31
5.3.1	Counting Performance Events	31
5.3.2	Manual performance counters collection	32
5.3.3	Multiplexing and scaling events	33
5.4	Sampling	34

5.4.1	User-Mode And Hardware Event-based Sampling . . . . .	35
5.4.2	Sampling process . . . . .	35
5.4.3	Collecting Call Stacks . . . . .	37
5.5	Static Performance Analysis . . . . .	37
5.5.1	Static vs. Dynamic Analyzers . . . . .	38
5.6	Compiler optimization reports . . . . .	39
5.7	Chapter Summary . . . . .	40
<b>6</b>	<b>CPU features for performance analysis</b>	<b>42</b>
6.1	Top-Down Microarchitecture Analysis . . . . .	42
6.1.1	TMAM in Intel® VTune™ Profiler . . . . .	44
6.1.2	TMAM in Linux Perf . . . . .	45
6.1.3	Step1: Identify the bottleneck. . . . .	46
6.1.4	Step2: Locate the place in the code. . . . .	48
6.1.5	Step3: Fix the issue. . . . .	49
6.1.6	Summary . . . . .	50
6.2	Last Branch Record . . . . .	51
6.2.1	Collecting LBR stacks . . . . .	53
6.2.2	Capture call graph. . . . .	54
6.2.3	Identify hot branches. . . . .	54
6.2.4	Analyze branch misprediction rate. . . . .	55
6.2.5	Precise timing of machine code. . . . .	56
6.2.6	Estimating branch outcome probability. . . . .	58
6.2.7	Other use cases . . . . .	58
6.3	Processor event-based sampling . . . . .	59
6.3.1	Precise events . . . . .	60
6.3.2	Lower sampling overhead . . . . .	61
6.3.3	Analyzing memory accesses . . . . .	61
6.4	Intel Processor Traces . . . . .	62
6.4.1	Workflow . . . . .	62
6.4.2	Timing Packets . . . . .	63
6.4.3	Collecting and Decoding Traces . . . . .	63
6.4.4	Usages . . . . .	65
6.4.5	Disk Space and Decoding Time . . . . .	65
6.5	Chapter Summary . . . . .	66
<b>7</b>	<b>Source Code Tuning For CPU</b>	<b>67</b>
7.1	Data-Driven Optimizations . . . . .	67
7.2	CPU Front-End Optimizations . . . . .	69
7.2.1	Machine code layout . . . . .	69
7.2.2	Basic block placement . . . . .	70
7.2.3	Basic block alignment . . . . .	72
7.2.4	Function splitting . . . . .	73
7.2.5	Function grouping . . . . .	74
7.2.6	Profile Guided Optimizations . . . . .	76
7.2.7	Optimizing for ITLB . . . . .	77
7.2.8	Summary . . . . .	77
7.3	CPU Back-End Optimizations . . . . .	78
7.3.1	Memory Bound . . . . .	78
7.3.1.1	Cache-Friendly Data Structures. . . . .	79

7.3.1.2	Memory Prefetching. . . . .	83
7.3.1.3	Optimizing For DTLB. . . . .	84
7.3.2	Core Bound . . . . .	86
7.3.2.1	Inlining Functions. . . . .	86
7.3.2.2	Loop Optimizations . . . . .	87
7.3.2.3	Vectorization . . . . .	87
7.4	Optimizing Bad Speculation . . . . .	88
7.4.1	Replace branches with lookup . . . . .	88
7.4.2	Replace branches with predication . . . . .	89
7.5	Other Tuning Areas . . . . .	90
7.5.1	Compile-Time computations . . . . .	91
7.5.2	Compiler intrinsics . . . . .	91
7.5.3	Cache Warming . . . . .	92
7.5.4	Detecting slow FP arithmetic . . . . .	93
7.5.5	System Tuning . . . . .	93
7.6	Chapter Summary . . . . .	94
<b>8</b>	<b>Optimizing multithreaded applications</b>	<b>96</b>
8.1	Performance scaling and overhead. . . . .	96
8.2	Parallel Efficiency Metrics. . . . .	98
8.2.1	Effective CPU Utilization. . . . .	98
8.2.2	Thread Count. . . . .	98
8.2.3	Wait Time. . . . .	98
8.2.4	Spin Time. . . . .	99
8.3	Analysis With Intel VTune Profiler . . . . .	99
8.3.1	Find Expensive Locks. . . . .	100
8.3.2	Platform View. . . . .	100
8.4	Analysis with Linux Perf . . . . .	101
8.4.1	Find Expensive Locks. . . . .	102
8.5	Analysis with Coz . . . . .	104
8.6	Analysis with eBPF and GAPP. . . . .	104
8.7	Detecting Coherence Issues . . . . .	105
8.7.1	Cache Coherency Protocols . . . . .	105
8.7.2	True Sharing . . . . .	106
8.7.3	False Sharing . . . . .	106
8.7.4	Detecting Sharing Issues . . . . .	107
8.7.5	Avoiding Sharing Issues . . . . .	107
8.8	Chapter Summary . . . . .	108
	<b>References</b>	<b>109</b>

# 1 Introduction

They say, “performance is king”. It was true a decade ago and it certainly is now. According to [Dom, 2017], in 2017 the world has been creating 2.5 quintillion<sup>1</sup> bytes of data every day, and as predicted in [Sta, 2018] this number is growing 25% per year. In our increasingly data-centric world the growth of information exchange fuels the need to both faster SW and faster HW. Fair to say, data growth puts demand not only on compute power, but also on storage and network systems.

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

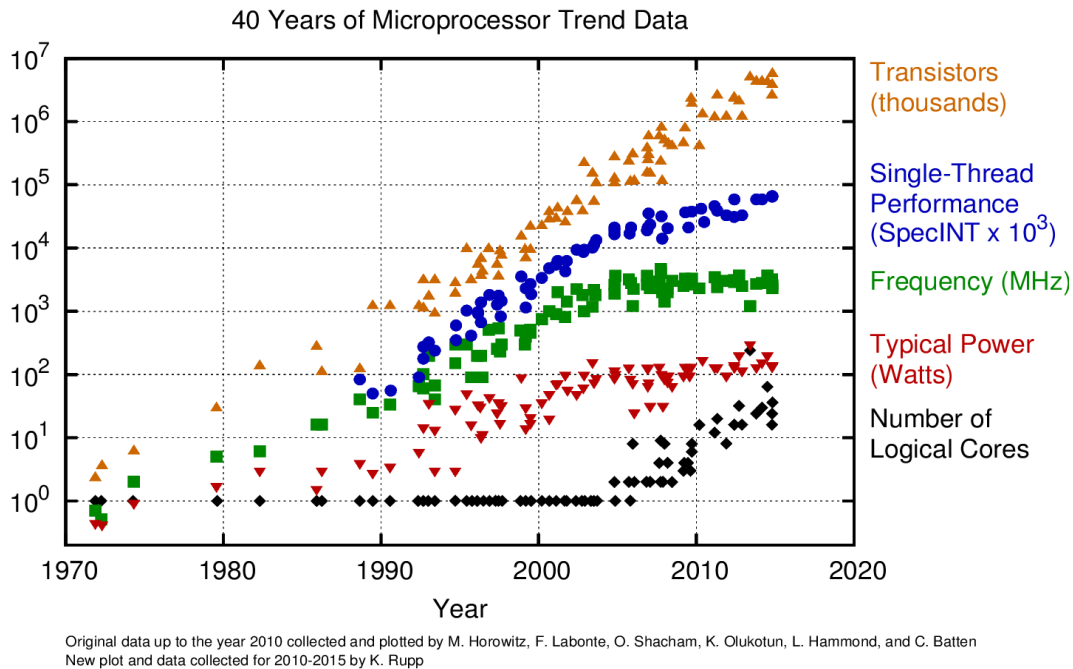


Figure 1: 40 Years of Microprocessor Trend Data. Credit: <https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/>

When it’s no longer the case that each HW generation provides significant performance boost, we must start paying more attention to how fast our code actually runs. When seeking ways to improve performance developers should not rely on HW and start optimizing 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

**Personal Experience:** While working at Intel I hear the same story from time to time: when Intel clients experience slowness in their application, they immediately and unconsciously start blaming Intel for having slow CPUs. But when Intel sends one of our performance ninjas to work with them and

<sup>1</sup>a thousand raised to the power of six ( $10^{18}$ )

help them improve their application, it is not unusual that they help speed it up the by a factor of 5x, sometimes even 10x.

Reaching ninja-level performance is hard, usually requires going several additional miles, but hopefully, in this book you will find the tools that will help you in it.

Performance of the system depends on different components: CPU, OS, memory, I/O devices, etc. Applications could benefit from tuning various components of the system. Even though in general it's wrong to focus our efforts only on CPU, the biggest factor in systems performance is its heart, the CPU. This book primarily focuses on performance analysis from CPU perspective occasionally touching on OS and memory subsystems.

This chapter explains what performance analysis is, why and who needs it. Chapter 2 discusses how to conduct fair performance experiments and analyze their results. Chapter 3 and 4 provide basics of CPU microarchitecture and terminology in performance analysis, feel free to skip if you know this already. Several most popular approaches for doing performance analysis are explored in Chapter 5. Chapter 6 provides information on features provided by HW to support and enhance performance analysis. Chapter 7 contains recipes for typical performance problems and is organized in a most convenient way to be used with Top-Down Microarchitecture Analysis (see sec. 6.1) which is one of the most important concepts of the book. Some important techniques for analyzing multithreaded applications are discussed in Chapter 8.

This book is written with the goal of helping developers better understand the performance of their application, learn to find inefficiencies and eliminate them. *Why my hand-written archiver performs\* 2 times slower than the conventional one? Why did my change in the function caused 2x performance drop? Customers are complaining about slowness of my application and I don't know where to start? Have I optimized the program to its full potential? What do I do with all that cache misses and branch mispredictions?\** Hopefully, by the end of this book you will have the answers to those questions.

All the examples in this book will be written in C, C++ or x86 assembly languages. But to a large degree this book is language agnostic, meaning that you can apply the same principles in other native languages like (Rust, Go and even Fortran).

## 1.1 What Is Performance Analysis?

Ever found yourself debating with a coworker about 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 the modern processors even a small tweak to the code can trigger significant performance change. That's why the first advice in this book is: Always Measure.

**Personal Experience:** I see many people rely on intuition when they try to optimize their application. And usually it ends up with random fixes here and there without making any real impact on performance of the application.

One simple example when inexperienced developers hope the change will improve performance is replacing `i++` with `++i` all over the code base, assuming that the previous value of `i` is not used. In the general case this change will make no difference to the generated code, because every decent optimizing compiler will recognize that the previous value of `i` is not used and will eliminate redundant copies anyway. Some people tend to overuse bit-twiddling tricks that

used to work well in the past. One such example is using [XOR-based swap idiom](#)<sup>2</sup> without testing, while in reality, simple `std::swap` produces faster code. Such accidental changes likely won't improve performance of the application. Finding the right place to fix should be a result of careful performance analysis, not intuition.

There are many performance analysis methodologies<sup>3</sup> that may or may not lead you to a discovery. CPU-specific approaches to performance analysis in this book (see sec. 5) have one thing in common: they are based on collecting certain information about how the program executes. Any change that ends up being made in the source code of the program is driven by analyzing and interpreting collected data.

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

## 1.2 Why Do We Still Need Performance Tuning?

Modern CPUs are getting more and more cores each year. As of the end of 2019, you can buy top bin server processor which will have more than 100 logical cores. This is very impressive, but that doesn't mean we don't have to care about performance anymore.

Very often application performance might not get better if you assign more cores to it. Later in sec. 8 we will discuss challenges of optimizing multithreaded applications. Performance of typical general-purpose multithread application doesn't always scale linearly with the number of cores we assign to the task. Understanding why that happens and possible ways to fix it, is critical for future growth of the product. Not being able to do proper performance analysis and tuning leaves lots of performance and money on the table and can kill the product.

This book focuses on squeezing out the last bit of performance from your application. Thus, the types of improvements that will be discussed are usually not big and often times do not exceed 10%. However, do not underestimate the importance of 10% speedup. It is especially important for large distributed applications being ran in the cloud:

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

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

1. **CPU limitations.** It's so tempting to ask: “*Why HW does not solve all our problems?*”. Modern CPUs execute instructions at incredible speed and are getting better with every generation. But still they cannot do much if the instructions that are used to perform the job are not optimal or even redundant. Processors cannot magically transform the bad code into something that performs better. For example, if we implement sorting routine using [BubbleSort](#)<sup>4</sup> algorithm, CPU will not make any attempts to recognize it and use the better alternatives, for example, [QuickSort](#)<sup>5</sup>. It will silently execute whatever it was told to do.

---

<sup>2</sup>[https://en.wikipedia.org/wiki/XOR\\_swap\\_algorithm](https://en.wikipedia.org/wiki/XOR_swap_algorithm)

<sup>3</sup><http://www.brendangregg.com/methodology.html>

<sup>4</sup>[https://en.wikipedia.org/wiki/Bubble\\_sort](https://en.wikipedia.org/wiki/Bubble_sort)

<sup>5</sup><https://en.wikipedia.org/wiki/Quicksort>



2. **Compilers limitations.** *“But isn’t it what compilers are supposed to do? Why compilers do not solve all our problems?”* Indeed, compilers are amazingly smart nowadays, however they are also subject to generating suboptimal code. Compilers are great at eliminating redundant work, but when it comes to making more complex decisions like function inlining, loop unrolling, etc. they may not generate best possible code. For example, there is no binary “yes” or “no” answer to the question if compiler should always inline a function into the place where it’s called. It usually depends on many factors which compiler should take into account. Often times, compilers rely on complex cost models and heuristics which may not work for every possible scenario.
3. **Complexity Analysis Limitations.** Developers are often times overly obsessed with complexity analysis of the algorithms, which leads them to choosing the popular algorithm with the optimal algorithmic complexity, even though it may not be the most efficient for a given problem. Considering two sorting algorithms [InsertionSort](#)<sup>6</sup> and [QuickSort](#)<sup>7</sup>, the latter clearly wins in terms of Big O notation for the average case: InsertionSort is  $O(N^2)$  while QuickSort is only  $O(N \log N)$ . Yet for relatively small sizes of  $N$ <sup>8</sup>, InsertionSort outperforms QuickSort. Complexity analysis cannot account for all the branch prediction and caching effects of various algorithms, so they just encapsulate them in an implicit  $C$  constant. Blindly trusting Big O notation without testing on the target workload could lead developers down an incorrect path.

This leaves the room for tuning performance of our applications to reach their full potential. Since HW is literally “set in stone”, we cannot change it<sup>9</sup>. So, the only influence point for most of us is SW. Broadly speaking, this includes all SW layers, including firmware, BIOS, OS and the source code of our application. Since for most of the readers lower SW layers are out of control, major focus will be made on the source code of user’s application. However, general tips for system SW tuning provided as well in sec. 7 and sec. 8. Another important piece of SW that we will touch a lot is compiler. A lion share of all performance gains usually comes from making compiler generate desired machine code through various hints. You will find many such examples throughout the book.

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

### 1.3 Who Needs Performance Tuning?

In the game of big numbers small improvements can make significant impact. Google reported that 2% slower search caused 2% less searches<sup>10</sup> per user. For Yahoo! 400 milliseconds faster page load caused 5-9% more traffic<sup>11</sup>. Those examples prove that the slower the application works the less people will use it. By putting emphasis on performance, you can give your product competitive advantage.

<sup>6</sup>[https://en.wikipedia.org/wiki/Insertion\\_sort](https://en.wikipedia.org/wiki/Insertion_sort)

<sup>7</sup><https://en.wikipedia.org/wiki/Quicksort>

<sup>8</sup>typically between 7 and 50 elements

<sup>9</sup>Not taking into account FPGAs.

<sup>10</sup>[https://assets.en.oreilly.com/1/event/29/Keynote Presentation 2.pdf](https://assets.en.oreilly.com/1/event/29/Keynote%20Presentation%20.pdf)

<sup>11</sup><https://www.slideshare.net/stoyan/dont-make-me-wait-or-building-highperformance-web-applications>

“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](#)<sup>12</sup>.

Performance engineering is an important and rewarding work, but it may be very time consuming. In fact, performance optimization is a never-ending game, there will be always something to optimize. Inevitably, developer will reach the point of diminishing returns at which further improvement will come at a very high engineering cost and likely will not worth the efforts.

So, before embarking on this road make sure you have a strong reason to do so. Optimization just for optimization sake is useless if it doesn’t add value to your product. Mindful performance engineering starts with clearly defined performance goals, stating what you are trying to achieve and why you are doing it. Also, you should pick the metrics you will use to measure if you reach the goal. More on the topic of setting performance goals you can read in [Gregg, 2013] and [Akinshin, 2019].

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

## 1.4 Chapter Summary

- Software Performance Analysis and Tuning is relevant and will remain so at least for the next decade.
- HW is not getting that much performance boosts in single-threaded performance as it used to in the past years.
- Predicting performance of a certain piece of code is nearly impossible, since modern systems are very non-deterministic. When doing performance optimizations of the SW, developers should not rely on intuition, but use careful analysis instead.
- For large distributed applications every small performance improvement result in immense cost savings. Importance of performance tuning should not be underestimated even if the relative speedup is not big. For some types of applications performance is not just a feature, it enables users to solve new kinds of problems in a new way.
- Certain limitations exist that prevent applications to reach their full performance potential. Both HW and SW environments have such limitations. CPUs cannot magically speedup slow algorithms, compilers are far from generating optimal code for every program. All this leaves room for tuning performance of our applications.
- SW optimizations should have quantifiable goals and metrics which should be used to measure progress.

---

<sup>12</sup><https://blog.nelhage.com/post/reflections-on-performance/>

---

## 2 Measuring Performance

Some people attribute performance as one of the features of the application<sup>13</sup>. But unlike other features, performance is not a Boolean property: applications always have some level of performance. This is why it's impossible to answer “yes” or “no” to the question whether application has performance. Additionally, “debugging” performance issues is usually harder than debugging functional issues, because every run of the benchmark is different from each other. For example, when unpacking a zip-file we get the same result over and over again, which means this operation is reproducible<sup>14</sup>. However, it's impossible to reproduce exactly the same performance profile of this operation.

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

### 2.1 Conducting Performance Experiments

There are many features in HW and SW that are intended to increase performance. But not all of them have deterministic behavior. Let's consider [Dynamic Frequency Scaling](#)<sup>15</sup> (DFS): this is a feature that allows CPU to increase its frequency for a short time intervals making it run significantly faster. However, CPU can't stay in “overclocked” mode for a long time, so later it decreases its frequency back to the base value. DFS usually depends a lot on a core temperature which makes it hard to predict impact on our experiments.

If we start two runs of the benchmark, one right after another on a “cold” processor, first run will possibly work for some time in “overclocked” mode. Only then CPU will decrease its frequency back to the base level. However, the second run will not have this advantage and will operate on base frequency without entering the “turbo” mode. Even though we run exact same version of the program two times, environment in which they run is not the same. Figure 2 shows situation where dynamic frequency scaling can cause variance in measurements.

Frequency Scaling is a HW feature, but variations in measurements might come also from SW features. Let's consider example of filesystem cache. If we benchmark an application that does lots of file manipulation, filesystem can play a big role in performance. When we start first iteration of the benchmark, required entries in the filesystem cache are missing. That may significantly increase the running time of the application. However, when running the same benchmark second time, filesystem cache will be already warmed up and this run will be much faster.

You can find more examples of features that can bring noise in performance measurements

---

<sup>13</sup>Blog post by Nelson Elhage “Reflections on software performance”: <https://blog.nelhage.com/post/reflect-ions-on-performance/>

<sup>14</sup>assuming no data races

<sup>15</sup>[https://en.wikipedia.org/wiki/Dynamic\\_frequency\\_scaling](https://en.wikipedia.org/wiki/Dynamic_frequency_scaling)

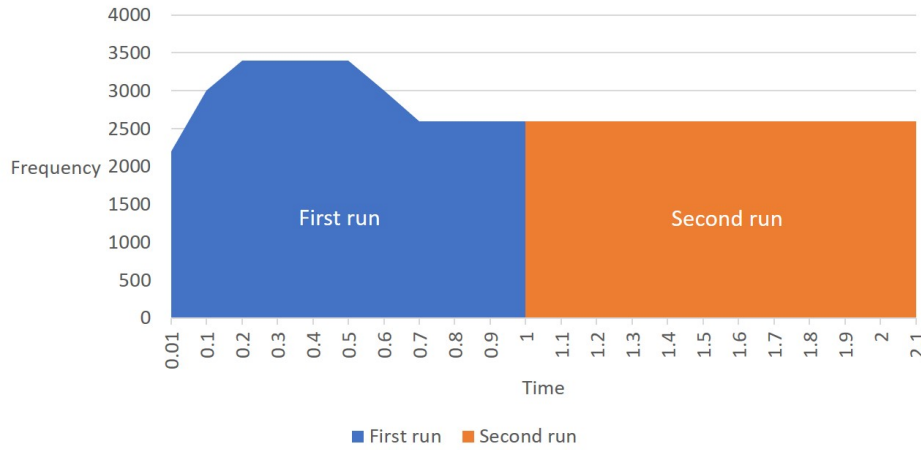


Figure 2: Variance in measurements caused by frequency scaling

and how to disable them in the [article<sup>16</sup>](#) on easyperf blog. Also, there are tools that can set up the environment to ensure benchmarking results with a low variance, one of them is [temci<sup>17</sup>](#).

Having consistent measurements, requires having all iterations of the benchmark to run in the same conditions. However, it is not possible to replicate the exact same environment and eliminate bias completely: there could be different temperature conditions, power delivery spikes, etc. Chasing all potential sources of noise and variation in the system can be a never-ending story. At some point trying to reduce the variance of the measurements can be very expensive. It is more of a business decision when and where exactly you should stop. This is a tradeoff between the time you spent on configuring your systems and accuracy of your measurements. Key indicator that you reached desired accuracy of the measurements can be [standard deviation<sup>18</sup>](#) of the measurement set.

It's important to keep in mind that even if particular HW or SW feature has non-deterministic behavior, that doesn't mean it is considered harmful. It gives inconsistent result, but overall it improves performance of the system. Disabling such a feature might give more accurate results but will likely decrease overall performance. So, overall the benchmark suite will run longer. This might be especially important for nightly performance testing when there are time limits for how long it should take to run the whole benchmark suite. Consequently, it is important to understand that it is not how your application will run in practice. Users of your application will likely have standard settings with maximum performance features enabled.

So, eliminating non-determinism in a system should only serve the goal of getting more consistent measurements, for example when you implement some code change and want to know the relative speedup ratio by benchmarking two different versions of the same program. If you just want to get absolute scores to understand how your app will behave in the field, you should try to replicate client configuration and should not make any artificial tuning to the system. Any performance analysis work, including profiling (see sec. 5.4), should be done on a target machine for which you're trying to optimize your application with corresponding configuration.

Usually big companies have a dedicated pool of machines which are properly configured for performance measurements. No one is allowed to log into those machines besides automatic

<sup>16</sup><https://easyperf.net/blog/2019/08/02/Perf-measurement-environment-on-Linux>

<sup>17</sup><https://github.com/partimenerd/temci>

<sup>18</sup>[https://en.wikipedia.org/wiki/Standard\\_deviation](https://en.wikipedia.org/wiki/Standard_deviation)

system that submits tasks. Those machines run minimal set of kernel and user background processes required to run performance tests. Finally, those machines are usually configured differently, depending on the type of experiments you want to make. They could have non-deterministic features disabled to have accurate measurements, or they could be tuned for maximum performance.

**Personal Experience:** Remember, even running ‘top’ on a machine can affect measurements, since some core will be activated and assigned to it. This might affect the frequency of the core that is running the actual benchmark.

More and more often datacenter application providers choose to profile and monitor performance on production systems [Ren et al., 2010], since it’s not possible to synthesize exact behavior for “in-house” performance testing. While it’s easier and more representative, it doesn’t eliminate the need of contiguous testing to catch performance problems early.

Unfortunately, measurement bias does not only come from environment configuration. [Mytkowicz et al., 2009] paper demonstrates that UNIX environment size (i.e., total number of bytes required to store the environment variables) and link order (the order of object files that are given to the linker) can affect performance in unpredictable ways. Moreover, there are numerous other ways of affecting memory layout and potentially affecting performance measurements.

One approach to enable statistically sound performance analysis of software on modern architectures was presented in [Curtsinger and Berger, 2013]. This work shows that it’s possible to eliminate measurement bias that comes from memory layout by efficiently and repeatedly randomizing the placement of code, stack, and heap objects at runtime.

## 2.2 Comparing Performance Measurements

When doing performance improvements in our code, we need a way to prove that we actually made it better. Also, when we commit a regular code change, we want to make sure performance did not regress. Typically, we do this by 1) measure the baseline performance, 2) measure performance of the modified program and 3) compare them with each other.

The goal in such scenario is to compare performance of two different versions of the same **functional** program. For example, we have a program that recursively calculates Fibonacci numbers and we decided to rewrite it in an iterative fashion. Both are functionally correct and yield the same numbers. Now we need to compare performance of two programs.

It is very much recommended to get not just a single measurement, but to run the benchmark multiple times. So, we have  $N$  measurements for the baseline and  $N$  measurements for the modified version of the program. Now we need a way to compare those 2 sets of measurements to decide which one is faster. This task is intractable by itself and there are many ways to be fooled by the measurements and potentially drive wrong conclusions from them.

If you ask any data scientist, they will tell you that you should **not** rely on a single metric (min/mean/median, etc.). By looking at the Figure 3, it’s tempting to say that **A** is faster than **B**, however, it is true only with some probability  $P$ . This is because there are some measurements of **B** that are faster than **A**. Even in the situation when all the measurements of **B** are slower than every measurement of **A** probability  $P$  is not equal to 100%. This is because we can always produce one additional sample for **B** which may be faster than some samples of **A**.

So, to be on a safe side, scientists usually present measurements by plotting the distributions

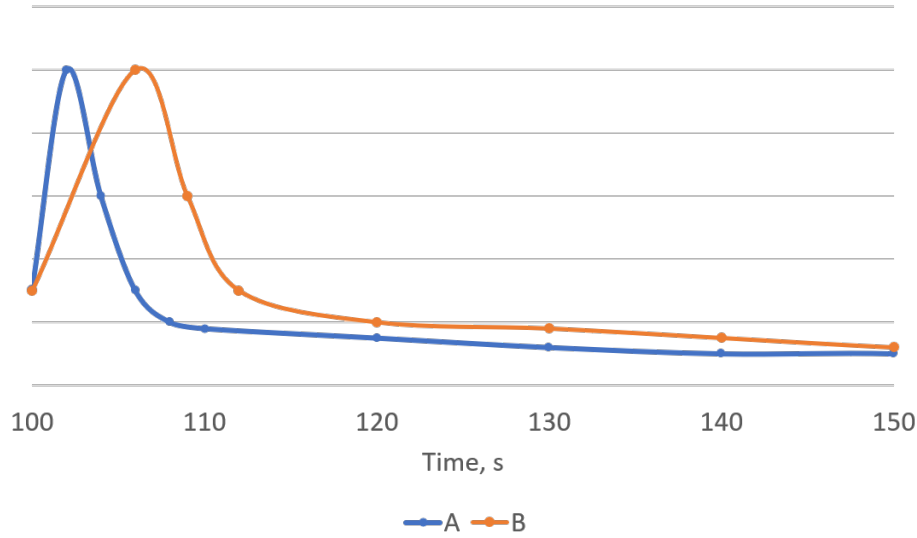


Figure 3: Comparing 2 performance measurement distributions

and let users drive their own conclusions from those plots. Another reason for looking at the distribution plots is that it allows to spot unwanted behavior of the benchmark<sup>19</sup>. If the distribution is bimodal, the benchmark likely experiences two different types of behavior. To “fix” this, different functional patterns should be isolated and benchmarked separately.

One of the popular ways to plot distribution is box plot (see 4). The main advantage of box plots is that they allow comparisons of multiple distributions on the same chart.

However, the approach with plots doesn’t work for automated benchmarking systems. Usually we want a clear answer, for example: “version A is faster than version B by X%”. To have this comparison we need to take a ratio between two values taken from each measurement set.

Since we know that aggregating performance distribution with a single value is not statistically correct and manual analysis of distributions is not an option either, this becomes more of a business decision. Saving developers time by automating benchmarking comes with a cost of comparison accuracy. Next we will discuss what usually works in practice.

The most important thing in achieving accurate performance conclusions is collecting rich collection of samples, i.e. run the benchmark a big number of times. This may sound obvious, but it is not always achievable. For example, some of the [SPEC benchmarks](http://spec.org/cpu2017/Docs/overview.html#benchmarks)<sup>20</sup> run for more than 10 minutes on a modern machine. That means it would take 1 hour to produce just 3 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.

How to know how many samples is required to reach statistically sufficient distribution? The answer to this question is again depends on how much accurate comparison you want to have. The lower the variance between the samples in the distribution, the lower number of samples you need. [Standard deviation](https://en.wikipedia.org/wiki/Standard_deviation)<sup>21</sup> is the metric that tells how consistent are the measurements in the distribution. One can implement adaptive strategy by dynamically limiting the number

<sup>19</sup> Another way to check this is to run normality test: [https://en.wikipedia.org/wiki/Normality\\_test](https://en.wikipedia.org/wiki/Normality_test).

<sup>20</sup> <http://spec.org/cpu2017/Docs/overview.html#benchmarks>

<sup>21</sup> [https://en.wikipedia.org/wiki/Standard\\_deviation](https://en.wikipedia.org/wiki/Standard_deviation)

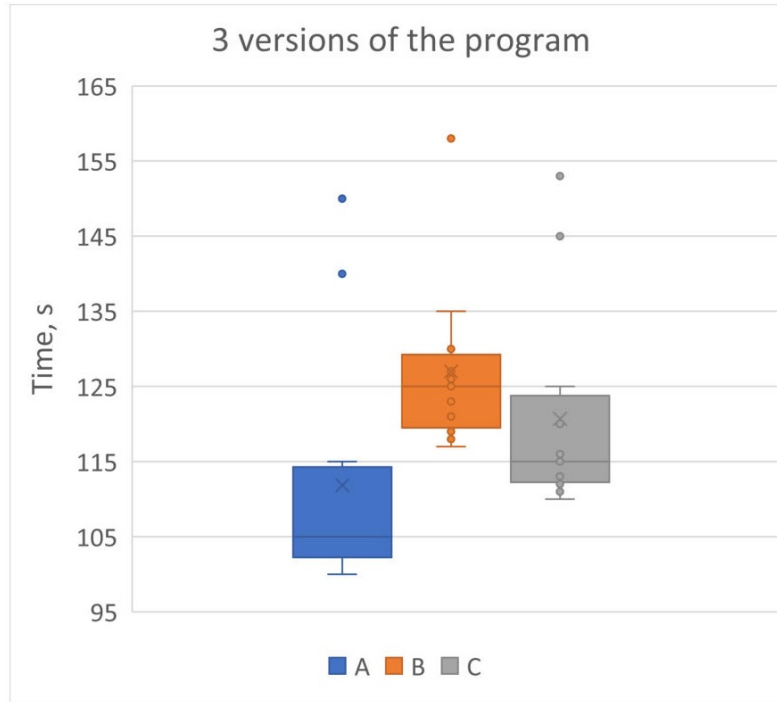


Figure 4: Box plots

of benchmark iterations based on standard deviation. I.e. you collect samples until you get standard deviation lie in a certain range. Once you have standard deviation lower than some threshold you could stop collecting measurements. For example, if you have a set of 5 measurements with the sample mean of 100s and standard deviation of 20s you could proceed collecting measurements until you will have standard deviation be lower than 5s. Again, there is no predefined value for standard deviation to control the distribution. Likely, the key factor here is how cheap it is to produce additional sample, i.e. rerun the benchmark.

Once you have statistically sufficient distributions it is usually acceptable to take mean (average) or geomean. But, on a small collection of samples mean and geomean can be affected by outliers. Unless the distribution has low variance, do not consider averages alone. If the variance in the measurements is in the order of magnitude with the mean, average is not a representative metric. Figure 5 shows example of 2 version of the program. By looking at averages (5a), it's tempting to say that A is faster than B by 20%. However, taking into account variance of the measurements (5b), we can clearly see that it is not always the case, and it's possible that sometimes B will be 20% faster than A.

Another important thing to watch out for is the presence of outliers. It is ok to discard some samples (for example cold runs) as outliers by using confidence intervals but do never blindly discard samples from the measurements. For some types of benchmarks outliers can bring a lot of value. When dealing with SW that has real-time constraints, 99-percentile could be the most important metric. There is a series of talks about measuring latency by Gil Tene on [YouTube](#) that covers this topic well.

## 2.3 Interestingness threshold

When comparing two versions of the same program, some fluctuation in performance is inevitable. While having a nightly performance testing in place, quite often you will see



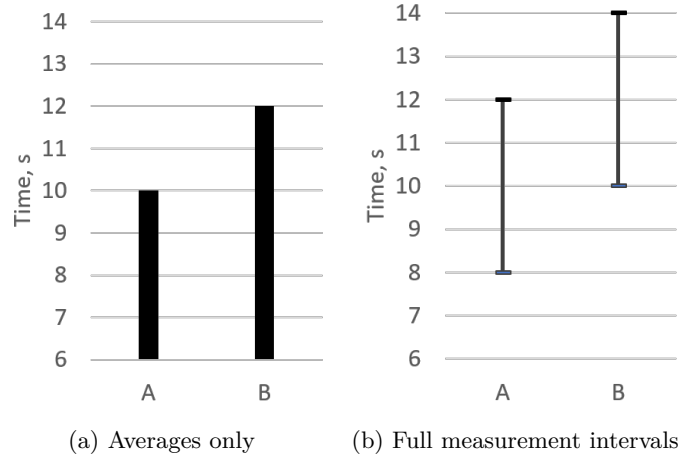


Figure 5: Two histograms showing how averages could be misleading.

performance variations as developers commit their changes. Small performance degradations can arise even without code changes due to HW instability, I/O, environment, etc. (see sec. 2.1). Sometimes even a harmless code change can trigger performance variation in your benchmark.<sup>22</sup>

Since usually we cannot do much about instability described above, we need a way to filter the noise from the real regressions resulted from a bad code change. The last thing we want is to be bothered with everyday performance changes of 0.5% up or down. Understanding whether regression comes from a bad code change or from other sources can be quite time consuming, especially if it a small regression ( $<2\%$ ).

To deal with such kind of issues it's recommended to define some threshold to filter small performance variations. For example, if we set the interestingness threshold to 2%, it means that every performance less than that will be considered as noise and can be ignored. Keep in mind, that by using such a criterion, sometimes you can ignore the real performance regressions. This is a business decision to make and again is a trade-off. On the one hand you want to save your time and avoid analyzing small regressions that often are just noise. On the other hand, you might miss the bad code change.

**Personal Experience:** For compute bound workloads I see people set their threshold in the range from 2% to 5%. The actual value of this threshold usually depends on 1) how flaky your benchmark is 2) how much you care about small performance changes.

Be sure to track absolute numbers as well, not just ratios. E.g. if you have 4 consecutive 1.5% regressions, they will all be filtered by the interestingness threshold, but they will sum up to 6% over 4 days. You don't want to skip such long-term degradation, so make sure you track the absolute numbers from time to time to see the trend in which performance of your benchmark is going.

<sup>22</sup>[https://easyperf.net/blog/2018/01/18/Code\\_alignment\\_issues](https://easyperf.net/blog/2018/01/18/Code_alignment_issues).



## 2.4 Microbenchmarks

To benchmark execution time engineers usually use 2 different timers which all the modern platforms provide:

- **System-wide high-resolution timer.** This is system timer which is typically implemented as a simple count of the number of ticks that have transpired since some arbitrary starting date, called the [epoch](#)<sup>23</sup>. This clock is monotonic, has nano-seconds resolution and is consistent between all the processes. System time can be retrieved from the OS with the system call<sup>24</sup>.
- **Time Stamp Counter (TSC).** This is a HW timer which is implemented as a HW register. TSC is also monotonic, meaning it doesn't account for frequency changes. Every thread has it's own TSC, which is simply the number of reference cycles (see sec. 4.6) elapsed. The value of TSC can be retrieved with `RDTSC` assembly instruction, or through higher-level tools like Linux `perf`.

Choosing which timer to use is very simple and depends on how long is the thing that you measure. If you are measuring something very small, TSC will give you better accuracy. Conversely, it's pointless to measure with TSC program that runs for hours. Unless you really need cycle accuracy, system timer should be enough for large portion of cases. The *de facto* standard of accessing system timer in C++ is using `std::chrono`:

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

Nearly all modern languages have benchmarking frameworks, for C++ use Google [benchmark](#)<sup>25</sup> library, C# has [BenchmarkDotNet](#)<sup>26</sup> library, Julia has [BenchmarkTools](#)<sup>27</sup> package, etc. It's possible to write self-contained microbenchmark for quickly testing some hypothesis. Even though it is ok to use microbenchmarks for conducting performance experiments, we recommend to always verify your ideas on a real application in practical conditions.

There is one more important consideration for microbenchmarks: optimizing compilers can eliminate important code which could make the experiment useless, or even worse, drive you to the wrong conclusions. 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");
}
```

<sup>23</sup>Unix epoch starts at 1 January 1970 00:00:00 UT: [https://en.wikipedia.org/wiki/Unix\\_epoch](https://en.wikipedia.org/wiki/Unix_epoch).

<sup>24</sup>[https://en.wikipedia.org/wiki/System\\_time#Retrieving\\_system\\_time](https://en.wikipedia.org/wiki/System_time#Retrieving_system_time)

<sup>25</sup><https://github.com/google/benchmark>

<sup>26</sup><https://github.com/dotnet/BenchmarkDotNet>

<sup>27</sup><https://github.com/JuliaCI/BenchmarkTools.jl>

Simple way to test this is to check the profile of the benchmark and see if the intended code stands out as the hotspot. Sometimes abnormal timings could be spotted instantly, so use common sense while analyzing and comparing benchmark runs. One of the popular ways to keep compiler from optimizing away important code is to use `DoNotOptimize`-like helper functions, which do 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);
    }
}
```

Microbenchmarks are good for proving something quickly, but you should always verify your ideas on a real application in practical conditions. For the same reason, performance of a single function should not be tested in unit tests, since in a real-world scenario resources are usually much more limited. I.e. when running in unit test, function has all resources available to it, including caches. When it is inserted into the final program, it must share resources with other functions which might become critical (see more in [Fog, 2004, chapter 16.2]).

## 2.5 Chapter Summary

- Debugging performance issues is usually harder than debugging functional bugs due to measurements instability.
- You could never stop optimizing unless you set a particular goal. To know if you reached the desired goal you need to come up with meaningful definitions and metrics for how you will measure that. Depending on what you care about it could be: throughput, latency, operations per second (roofline performance), etc.
- Due to random errors we never have 100% confidence we achieved results that we wanted. Performance analysis must account for errors, which means we have to use statistic methods.
- Collect statistically significant measurements with low variance between samples.
- It's OK to discard cold runs in order to ensure that everything is running hot, but do not blindly discard your data. If you choose to discard some samples, do it uniformly for all distributions.
- Do not try to mix different types of behavior in one benchmark.
- To filter uninteresting performance changes that were potentially caused by instability, we recommend defining interestingness threshold.
- Microbenchmarks are good for proving something quickly, but you should always verify your ideas on a real application in practical conditions. Make sure that you are benchmarking the meaningful code by checking performance profiles.

While we presented straight-forward approaches for performance measurements, one can implement more complicated techniques like the one presented in [Chen and Revels, 2016].

---

## 3 CPU microarchitecture 101

[THIS SECTION IS NOT FINISHED]

### Message to reviewers:

Let me know if you wish to help me write this chapter or some sections of it. If you do, let me know first so we can coordinate our efforts. Obviously, I will mention you in the contributors section. My vision for this chapter is to have it compact enough, at the same time provide all the basic information. I know that the entire book can be written on this subject. I think 10 pages should be enough to cover fundamentals.

### 3.1 Classic ideas in CPU design

#### 3.1.1 Pipelining

#### 3.1.2 OOO execution

#### 3.1.3 Speculation

#### 3.1.4 SIMD

#### 3.1.5 Simultaneous multithreading (SMT)

### 3.2 Modern CPU design

[TODO Write here general information about number of cores, frequency, physical/architectural registers, pipeline stages, in-order front-end + OOO back-end, etc.]

#### 3.2.1 CPU Front-End

CPU Front-End is responsible for fetching assembly instructions from memory, decoding them and feeding them to the execution units. This is the part of the CPU core which has the greatest number of complex data structures and buffers, like MITE, DSB, LSD, Decoders, ROB, RAT, etc. [TODO: Write about those structures]

#### 3.2.2 CPU Back-End

[TODO: Write about the structures below]

- ROB
- Register Renamer
- Load/Store buffers
- Scheduler
- CPU execution ports
- TLBs.
- Retirement

#### 3.2.3 CPU Branch Predictor

#### 3.2.4 Memory hierarchy

[TODO: Write about the structures below]

- registers

- caches

-cache line

-temporal/spatial locality

1. **Temporal locality:** when a given memory location was accessed, it is likely that the same location is accessed again in the near future. Ideally, this information will still be cached at that point.
2. **Spatial locality:** this refers to placing related data close to each other. Caching happens on many levels, not just in the CPU. For example, when you read from RAM, typically a larger chunk of memory is fetched than what was specifically asked for because very often the program will require that data soon. HDD caches follow the same line of thought. Specifically, for CPU caches, the notion of “*cache lines*” is important.

-coherency

-[MAYBE] eviction strategies

-[MAYBE] writeback strategies

-[MAYBE] handling miss-under-miss

- DRAM
- NUMA [maybe a few sentences]

**3.2.4.1 Hardware memory prefetchers.** Hardware prefetchers observe the behavior of a running application and initiate prefetching on repetitive patterns of cache misses. In contrast to software prefetching, hardware prefetching does not require support from an optimizing compiler or profiling support. Furthermore, hardware prefetching can automatically adapt to the dynamic behavior of the application, such as varying data sets, or the hardware, such as systems with various cache sizes. Also, the hardware prefetches are generated without the overhead of additional address-generation and prefetch instructions. However, hardware prefetching is limited to learning and prefetching for a limited set of cache-miss patterns that are implementable in hardware. If accesses to consecutive cache line addresses miss, the hardware prefetches for the third cache line and continues to prefetch successive cache lines as the prefetched cache lines are accessed by the processor.[Iacobovici et al., 2004]

### 3.2.5 Performance Monitoring Unit

If we imagine a simplified view of the processor it may look something like what is shown on Figure 6.

Modern CPUs have caches, branch predictor, execution pipeline and a clock generator which sends pulses and makes everything move to the next stage. If we add just a little bit of silicon and connect it to the clock, we can count cycles.

By connecting it to all the units inside CPU we can build a performance monitoring counter (PMC) and start collecting interesting performance events. For example, we can count cache misses, branch mispredictions, executed instructions, etc. Notice, we also have configuration register, because now we need a way to tell which event we want to count.

Conceptually, PMC is a HW register, we can read and write to it. We can find out how many events of particular type happened by reading the value of the counter. We can also restart the counting by setting the counter to 0.

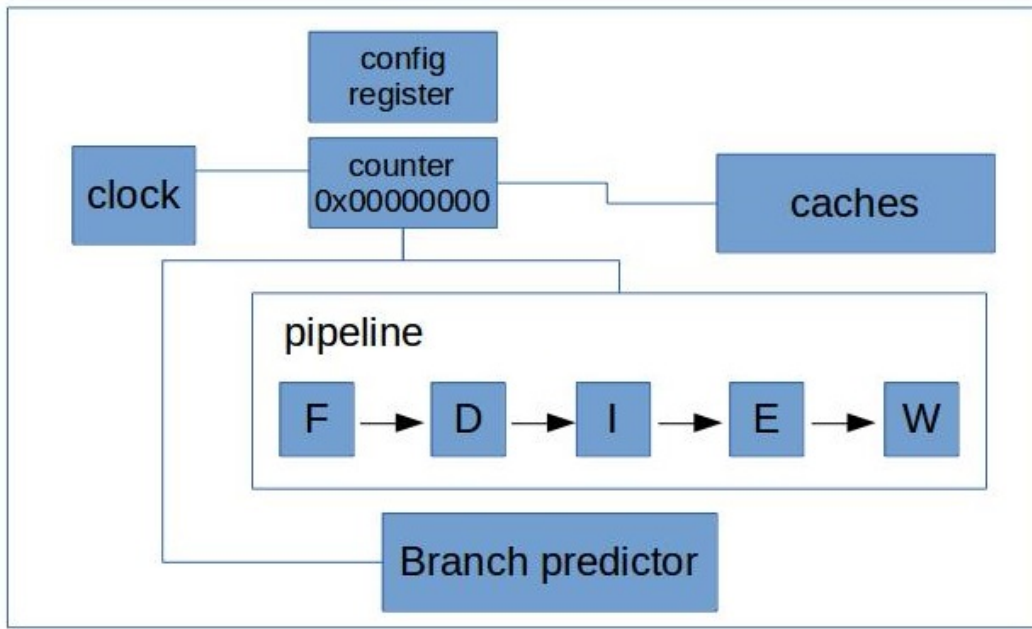


Figure 6: Simplified view of the processor

**3.2.5.1 Fixed and programmable counters.** It is so common that engineers count the number of executed instructions and elapsed cycles, that there are dedicated PMCs to count those. In practice CPUs have Performance Monitoring Unit (PMU) with fixed and programmable counters. Fixed PMCs always measures the same thing inside the core. With programmable counter it's up to user to choose what they want to measure.

Typically, there are 4 fully programmable counters and 3 fixed function counters per logical core. Fixed counters usually are set to count core clocks, reference clocks, instructions retired (see sec. 4 for terminology). [Int, 2020]

For Intel® Core™ i3-3220T processor here is the output of `cpuid` command:

```
$ cpuid
...
Architecture Performance Monitoring Features (0xa/eax):
  version ID                      = 0x3 (3)
  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)
...
```

Similar information can be grepped out from the kernel message buffer:

```
$ dmesg
...
[ 0.061530] Performance Events: PEBS fmt1+, IvyBridge events, 16-deep
LBR, full-width counters, Intel PMU driver.
[ 0.061550] ... version: 3
```

```
[ 0.061550] ... bit width:      48
[ 0.061551] ... generic registers: 4
[ 0.061551] ... value mask:      0000ffffffffffff
[ 0.061552] ... max period:      00007fffffffffff
[ 0.061552] ... fixed-purpose events: 3
[ 0.061553] ... event mask:      000000070000000f
...
```

PMU counters and configuration registers are implemented as Model Specific Registers (MSR). Number of counters and their width can vary from model to model and you can't rely on the same number of counters in your CPU, you should always query that first, using tools like `cpuid`, for example. MSRs are accessed via the `RDMSR` and `WRMSR` instructions. Certain counter registers can be accessed via the `RDPMC` instruction. TODO: More information and details are available in Volume 2B of the Programmer's Reference Manual.

---

## 4 Terminology and metrics in performance analysis

For a beginner it can be a very hard time looking into a profile generated by analysis tool like Linux `perf` and Intel VTune Profiler. Those profiles have lots of complex terms and metrics. This chapter is a gentle introduction into basic terminology and metrics used in performance analysis.

### 4.1 Retired vs. Executed Instruction

Modern processors execute much more instructions that the program flow needs. Speculative execution was discussed in sec. 3.1.3. Instructions that were proven as indeed needed by the program execution flow are “retired”.

So, an instruction processed by the CPU can be executed but not necessarily retired. Retired instruction is usually executed, except those times when it does not require an execution unit<sup>28</sup>. Taking this into account we can usually expect the number of executed instructions to be higher than the number of retired instructions.

There is a fixed performance counter (PMC) that is collecting the number of retired instructions. It can be easily collected with Linux `perf` by running:

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

### 4.2 CPU Utilization

CPU utilization is measured by the time a CPU is busy doing work. It is expressed as a percentage of the time CPU was busy during some time period. Technically, 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` PMC counts the number of reference cycles when the core is not in a halt state, `TSC` stands for timestamp counter.

If CPU utilization is low it usually means no good for performance of the application, since some portion of time was wasted by the CPU. However, high CPU utilization is not always good either. It is a sign that system is doing some work but not saying what exactly it is doing: CPU might be highly utilized even though it is stalled waiting on memory accesses. In multithreaded context a thread can also spin while waiting for resources to proceed, so there is Effective CPU utilization that filters spinning time (see sec. 8.2).

Linux `perf` automatically calculates CPU utilization:

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

---

<sup>28</sup>An example of it can be MOV elimination and zero idiom, read more on easyperf blog: <https://easyperf.net/blog/2018/04/22/What-optimizations-you-can-expect-from-CPU>.

### 4.3 CPI & IPC

Those are two very important metrics that stand for:

- Cycles Per Instruction (CPI) - how much cycles it took to retire one instruction on average.
- Instructions Per Cycle (IPC) - how much instructions were retired per one cycle on average.

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

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

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

There are many types of analysis that can be done based on those metrics. It is useful for both evaluating HW and SW efficiency. HW engineers use this metric to compare CPU generations and CPUs from different vendors. SW engineers look at IPC and CPI when they optimize their application. Universally, we want to have low CPI and high IPC. Linux `perf` users can get to know IPC for their workload by running:

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

### 4.4 UOP (micro-op)

The microprocessors with out-of-order execution are translating all instructions into micro-operations - abbreviated pops or uops. A simple instruction such as `ADD EAX,EBX` generates only one pop, while an instruction like `ADD EAX,[MEM1]` may generate two: one for reading from memory into a temporary (unnamed) register, and one for adding the contents of the temporary register to `EAX`. The instruction `ADD [MEM1],EAX` may generate three pops: one for reading from memory, one for adding, and one for writing the result back to memory. The advantage of this is that the pops can be executed out of order. [Fog, 2012, chapter 2.1]

Linux `perf` users can collect the number of issued, executed and retired uops for their workload by running<sup>29</sup>:

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

Uops also can be MicroFused and MacroFused. I encourage interested readers to visit easyperf blog<sup>30 31</sup> to find more details about it.

<sup>29</sup>`UOPS_RETIRED.ALL` event is not available since Skylake. Use `UOPS_RETIRED.RETIRE_SLOTS`.

<sup>30</sup><https://easyperf.net/blog/2018/02/15/MicroFusion-in-Intel-CPU>

<sup>31</sup><https://easyperf.net/blog/2018/02/23/MacroFusion-in-Intel-CPU>



## 4.5 Pipeline Slot

A pipeline slot represents hardware resources needed to process one uop. Figure 7 demonstrates execution pipeline of a CPU that can handle 4 uops every cycle. Nearly all modern x86 CPUs are made with a pipeline width of 4 (4-wide). During 6 consecutive cycles on the diagram only half of available slots were utilized. From microarchitecture perspective, efficiency executing such a code is only 50%.

Cycle	1	2	3	4	5	6
Slot1	X	X	X	X		X
Slot2	X	X	X			X
Slot3	X	X	X			
Slot4	X					

Figure 7: Pipeline diagram of a 4-wide CPU.

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

## 4.6 Core vs. Reference Cycles

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

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

Majority of modern CPUs including Intel's and AMD's ones don't have fixed frequency on which they operate. Instead, they implement [dynamic frequency scaling](#)<sup>32</sup>. In Intel's CPUs this technology is called [Turbo Boost](#)<sup>33</sup>, in AMD's processors it's called [Turbo Core](#)<sup>34</sup>. It allows the CPU to increase and decrease its frequency dynamically.

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

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

Metric `ref-cycles` counts cycles as if there were no frequency scaling. External clock on the setup has frequency of 100 MHz, and if we scale it by [clock multiplier](#) we will get the base

<sup>32</sup>[https://en.wikipedia.org/wiki/Dynamic\\_frequency\\_scaling](https://en.wikipedia.org/wiki/Dynamic_frequency_scaling)

<sup>33</sup>[https://en.wikipedia.org/wiki/Intel\\_Turbo\\_Boost](https://en.wikipedia.org/wiki/Intel_Turbo_Boost)

<sup>34</sup>[https://en.wikipedia.org/wiki/AMD\\_Turbo\\_Core](https://en.wikipedia.org/wiki/AMD_Turbo_Core)

frequency of the processor. Clock multiplier for Skylake i7-6000 processor equals to 34: it means that for every external pulse, CPU executes 34 internal cycles when it's running on the base frequency.

Metric "cycles" counts real CPU cycles, i.e. taking into account frequency scaling. We can also calculate how well dynamic frequency scaling feature was utilized as:

$$Turbo\ Utilization = \frac{Core\ Cycles}{Reference\ Cycles},$$

The core clock cycle counter is very useful when testing which version of a piece of code is fastest because you can avoid the problem that the clock frequency goes up and down.[\[Fog, 2004\]](#)

## 4.7 Cache miss

As discussed in sec. 3.2.4, any memory request missing in particular level of cache must be serviced by higher level caches or DRAM. This implies significant increase in latency of such memory access. Typical latency of memory subsystem components is shown in the table 1.

TODO: fix the numbers

Table 1: Typical latency of a memory subsystem.

Memory Hierarchy Component	Latency (cycle/time)
L1 Cache	4 cycles (1 ns)
L2 Cache	10-25 cycles (4 ns)
L3 Cache	cycles ( ns)
Main Memory	cycles (100 ns)

Cache miss might happen both for instruction and the data. According to Top-Down Microarchitecture Analysis (see sec. 6.1), instruction cache miss is characterized as Front-End stall, while data cache miss is characterized as Back-End stall. When I-Cache miss happens during instruction fetch, it is attributed as Front-End issue. Consequently, when the data that this load requests is not found in D-Cache, this will be attributed as Back-End issue.

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 L1 data cache. We can see that only 3.5% of all loads miss in L1 cache. 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 misses in L1D cache also missed L2 cache. In a similar way, breakdown for L3 cache can be made.

## 4.8 Mispredicted branch

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

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

Instruction `jz` is a branch instruction and in order to increase performance modern CPU architectures try to predict the result of such branch. This is also called “Speculative Execution”. Processor will speculate that, for example, branch will not be taken and will execute the code that corresponds to the situation when `eax` is not 0. However, if the guess was wrong, this is called “branch misprediction” and CPU is required to undo all the speculative work that it has done lately. This typically involves penalty between 10 and 20 clock cycles.

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

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

## 4.9 Basic Block

Basic block is a sequence of instructions with single entry and single exit. Figure 8 shows simple example of a basic block, where `MOV` instruction is an entry and `JA` is an exit instruction. While basic block can have one or many predecessors and successors, no instruction in the middle can exit the block.

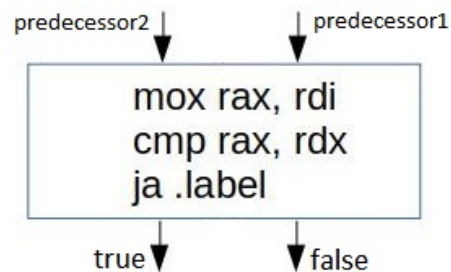


Figure 8: Basic Block of assembly instructions.

It is guaranteed that every instruction in the basic block will be executed exactly once. This is a very important property that is utilized by many compiler transformations. For example,

it greatly reduces the problem of control flow graph analysis and transformations, since for some class of problems we can treat all instructions in the basic block as one entity.

---

## 5 Performance Analysis Approaches

When doing high-level optimization, it is usually easy to tell whether performance was improved or not. When you write a better version of an algorithm, you expect to see visible difference in running time of the program. But also, there are situations when you see change in execution time, but you have no clue where it's coming from. Time alone does not provide any insight about why that happens. In this case we need more information about how our program executes. That's the situation when we need to do performance analysis to understand the underlying nature of the slowdown or speedup that we observe.

Both HW and SW track performance data while our program is running. In this context by HW we mean CPU which executes the program and by SW we mean OS and all the tools enabled for analysis. Typically, SW stack provides high-level metrics like time, number of context switches and page-faults, while CPU is capable of observing cache-misses, branch mispredictions, etc. Depending on the problem we are trying to solve, some metrics would be more useful than others. So, it doesn't mean that HW metrics will always give us more precise overview of the program execution. Some metrics, like the number of context-switches for instance, cannot be provided by CPU. Performance analysis tools, like `Linux perf`, can consume data both from OS and CPU.

In this section we will briefly discuss some of the most popular performance analysis techniques: Code Instrumentation, Tracing, Characterization and Sampling. In the end of this chapter we will touch on static performance analysis techniques and compiler optimization reports which do not involve running the actual application.

### 5.1 Code Instrumentation

Probably the first approach for doing performance analysis ever invented is Code Instrumentation. The simplest example of this method is inserting `printf` statements in the beginning of the function to count the number of times this function was called (see Listing 1). I think every programmer in the world did it at some point at least once. This technique provides very detailed information when you need specific knowledge about the execution of the program. You're pretty much capable to track any information about every variable in the program. Code instrumentation is highly useful and generally provides more detailed data than other techniques.

---

#### Listing 1 Code Instrumentation

---

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

---

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

But more importantly, the downside of this technique is that every time something new needs to be instrumented, say another variable, recompilation is required. This can become

a burden to an engineer and increase analysis time. It is not all downsides, unfortunately. Code instrumentation also adds significant amount of overhead. Since usually you care about hot paths in the application, you’re instrumenting the things that reside in the performance critical part of the code. Inserting instrumentation code in a hot piece of code might easily result in a 2x slowdown of the overall benchmark<sup>35</sup>. Finally, by instrumenting the code, you change the behavior of the program, so you might not see the same effects you saw earlier.

All of the above increases time between experiments and consumes more development time, which is why engineers don’t manually instrument their code very often these days. However, code instrumentation is still widely used by the tools. The most widely known use case for this is compiler Profile Guided Optimizations (see sec. 7.2.6). Compilers are capable of automatically instrumenting the whole program and collect some statistics about the execution. After that compiler can use this data to reoptimize the executable file based on the knowledge of the runtime behavior of the program.

When talking about instrumentation it’s important to mention binary instrumentation techniques like `Pin`<sup>36</sup> tool. This is extremely powerful technique, although is usually very expensive in terms of required time for analysis.

## 5.2 Tracing

Tracing is conceptually very similar to instrumentation. With the difference that something other than our code gets instrumented. And usually it is done in advance. For example, `strace` tool can be considered as instrumentation of Linux kernel. Intel Processor Traces (see sec. 6.4) can be viewed as instrumentation of the CPU. The difference here is that we can use only those tracing capabilities that are already available in the system.

Example of tracing system calls with Linux `strace` tool is demonstrated in Listing 2. This listing shows first several lines of output when running `git status` command. By tracing system calls with `strace` it’s possible to know the timestamp for each system call (the leftmost column), it’s exit status and the duration of each system call (in the angle brackets).

The overhead of tracing very much depends on what exactly we try to trace. For example, if we trace the program that almost never does system calls, the overhead of running it under `strace` will be close to zero. On the opposite, if we trace the program that heavily relies on system calls, the overhead could be very large like 100x<sup>37</sup>. Also, tracing can generate massive amount of data, since it doesn’t skip any sample. To compensate this, tracing tools provide the means to filter collection only for specific time slice or piece of code.

Usually, tracing alike instrumentation is used for exploring anomalies in the system. For example, you may want to find what was going on in the application during 10s period of unresponsiveness. Profiling is not designed for this, but with tracing you can see what lead to the program being unresponsive. For example, with Intel PT (see sec. 6.4) we 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. Since most of the tracing tools capable of decorating events with timestamps, this allows to have correlation with external events that were happening during that time.

<sup>35</sup>Remember not to benchmark instrumented code, i.e. do not measure score and do analysis in the same run.

<sup>36</sup><https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool>

<sup>37</sup><http://www.brendangregg.com/blog/2014-05-11/strace-wow-much-syscall.html>

**Listing 2** Tracing system calls with strace.

```

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

```

One important difference from profiling (see sec. 5.4) is that tracing approaches operate without interrupts which might be attractive for real time systems. I.e. profiling tools interrupt execution of analyzed program in order to capture a sample, while tracing tools avoid that.

### 5.3 Workload Characterization

Workload characterization is a process of describing a workload by means of quantitative parameters and functions. Its goal is to define the behavior of the workload and its most important features. On a high level, application can belong to one or many of the following types: interactive, database, network-based, parallel, etc. Different workloads can be characterized using different metrics and parameters to address particular application domain.

In sec. 6.1 we will closely look at Top-Down Microarchitecture Analysis (TMAM) methodology which attempts to characterize an application by putting it into one of 4 buckets: Front End Bound, Back End Bound, Retiring and Bad Speculation. TMAM uses Performance Monitoring Counters (PMC, see sec. 3.2.5.1) to collect needed information and identify the inefficient use of CPU microarchitecture.

#### 5.3.1 Counting Performance Events

PMCs are very important instrument of low-level performance analysis. They can provide unique information about execution of our program. PMCs are generally used in two modes: “Counting” and “Sampling”. Counting mode is used for workload characterization, while Sampling mode is used for finding hotspots which we will discuss in sec. 5.4. The idea behind Counting is very simple: we want to count the number of certain performance events during the time our program was running. Figure 9 illustrates the process of counting performance events in the time perspective.

The steps outlined in figure 9 roughly represent what typical analysis tool will do to count performance events. This process is implemented in `perf stat` tool which can be used to count various HW events, like the number of instructions, cycles, cache-misses, etc. Below is the example of output from `perf stat`:

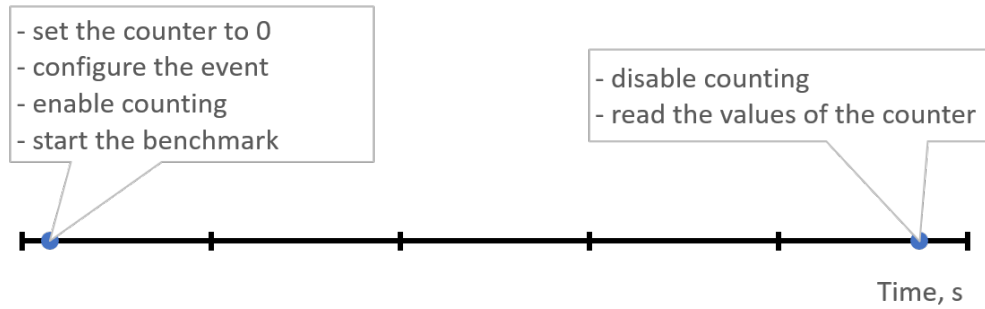


Figure 9: Counting performance events.

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

It is very informative to know this data. First of all, it allows to quickly spot some anomalies like high cache miss rate or poor IPC. But also, it might come handy when you’ve just made code improvement and you want to validate performance gain. Looking at absolute numbers might help you justify or reject the code change.

**Personal Experience:** I use ‘perf stat’ as a simple benchmark wrapper. Since the overhead of counting events is minimal, I run almost all benchmarks automatically under ‘perf stat’. It serves me as a first step of performance investigation. Sometimes the anomalies can be spotted right away which can save you some analysis time.

### 5.3.2 Manual performance counters collection

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

Complete list of performance events for all Intel CPU generations can be found in [Int, 2020, Volume 3B, Chapter 19].<sup>38</sup> Every event is encoded with **Event** and **Umask** hexadecimal values. Sometimes performance events can also be encoded with additional parameters, like **Cmask** and **Inv** and others. Example of encoding 2 performance events for Intel Skylake microarchitecture is shown in the table 2.

<sup>38</sup>They also available here: <https://download.01.org/perfmon/index/>.



Table 2: Example of encoding Skylake performance events.

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

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

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

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

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

Also there are wrappers around Linux `perf` that can do the mapping, for example, `oprofile`<sup>39</sup> and `ocperf.py`<sup>40</sup>. Below is the example of their usage:

```
$ ocperf -e uops_retired ./a.exe
$ ocperf.py stat -e uops_retired.retire_slots -- ./a.exe
```

**Personal Experience:** I usually use `perf` with raw encoding to collect performance counters, since most of the time I use performance events that have default name mapping. In case I need to use something different, I have a ready-to-use Linux `perf` command line fragments for most of the events.

### 5.3.3 Multiplexing and scaling events

There are situations when we want to count many different events at the same time. But with only one counter it's possible to count only one thing at a time. That's why PMUs have multiple counters in it. Even then, the number of fixed and programmable counter is not always sufficient. Top-Down Analysis Methodology (TMAM) requires collecting near 100

<sup>39</sup><https://oprofile.sourceforge.io/about/>

<sup>40</sup><https://github.com/andikleen/pmu-tools/blob/master/ocperf.py>

different performance events in a single execution of a program. Obviously, CPUs don't have that many counters, and here is when multiplexing comes into play.

If there are more events than counters, analysis tool uses time multiplexing to give each event a chance to access the monitoring hardware. With multiplexing, an event is not measured all the time. At the end of the run, the tool scales the count based on total time enabled vs time running. The actual formula is:

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

For example, say during profiling we were able to measure some counter 5 times, each measurement interval lasted 100ms (**time enabled**). The program executed time is 1s (**time running**). Total number of events for this counter is measured as 10000 (**raw count**). So, the **final count** will be equal to 20000:

$$\text{final count} = 10000 * (1000\text{ms} / 500\text{ms}) = 20000$$

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 an estimate not an actual count. Multiplexing and scaling can be used safely on steady workloads which execute the same code during long time intervals. On the opposite, if the program regularly jumps between different hotspots, there will be blind spots which can introduce errors during scaling. To avoid scaling, one can try to reduce the number of events to be not bigger than the number of physical PMCs available. However, this will require running the benchmark multiple times to measure all the counters one is interested in.

## 5.4 Sampling

Sampling is the most frequently used approach for doing performance analysis. People usually associate it with finding hotspots in the program. In general terms, sampling gives the answer to the question: which place in the code contributes to the greatest number of certain performance events. If we consider finding hotspots, the problem can be reformulated as: which place in the code consumes the biggest amount of CPU cycles. People often use the term “Profiling” for what is technically called Sampling. According to [Wikipedia](#)<sup>41</sup>, profiling is a much broader term and includes a wide variety of techniques to collect data including interrupts, code instrumentation and PMC.

It may come as a surprise, but the simplest sampling profiler one can imagine is a debugger. In fact, you can identify hotspots by a) run the program under 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 will build a collection of samples. The line of code where you stopped the most will be the hottest place in the program.<sup>42</sup> Of course, this is an oversimplified description of how real profiling tools work. Modern profilers are capable of collecting thousands of samples per second, which gives pretty accurate estimate about the hottest places in a benchmark.

As in example with debugger, execution of analyzed program is interrupted every time new sample is captured. At the time of interrupt, profiler collects the snapshot of the program state which constitutes one sample. Information collected for every sample may include instruction address that was executed at the time of interrupt, register state, call stack (see sec. 5.4.3), etc. Collected samples are stored in data collection files, which can be further used to display a call graph, the most time-consuming parts of the program (see 11) and control flow for statistically important code sections.

<sup>41</sup>[https://en.wikipedia.org/wiki/Profiling\\_\(computer\\_programming\)](https://en.wikipedia.org/wiki/Profiling_(computer_programming))

<sup>42</sup>This is an awkward way though, and we don't recommend doing this, it's just to illustrate the concept.

### 5.4.1 User-Mode And Hardware Event-based Sampling

Sampling can be performed in 2 different modes, using user-mode or HW event-based sampling (EBS). User-mode sampling is a pure SW approach which embeds an agent library into the profiled application. The agent sets up the OS timer for each thread in the application. Upon timer expiration, the application receives the `SIGPROF` signal that is handled by the collector. EBS uses hardware PMCs to trigger interrupts. In particular, the counter overflow feature of the PMU is used which we will discuss in next section.[\[Int, 2020\]](#)

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

User-mode sampling incurs more runtime overhead than EBS. Average overhead of the user mode sampling is about 5% when sampling is using the default interval of 10ms. The average overhead of event-based sampling is about 2% on a 1ms sampling interval. Typically, EBS is more accurate since it allows collecting samples with higher frequency. However, user-mode sampling generates much less data to analyze, and it takes less time to process it.

### 5.4.2 Sampling process

In this section we will discuss the scenario of using PMCs with EBS. Figure 10 illustrates counter overflow feature of the PMU which is used to trigger performance monitoring interrupt (PMI).

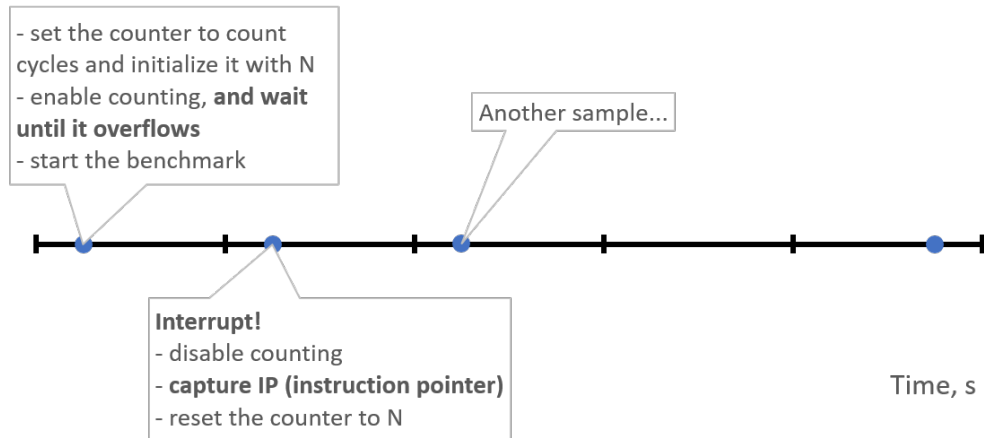


Figure 10: Using performance counter for sampling

In the beginning we configure the event that we want to sample on. Identifying hotspots means knowing where program spends time the most, so sampling on cycles is very natural. However, we can sample on any performance event we want. For example, if we would like to know the place where the program experiences the biggest number of L3-cache misses we would sample on the corresponding event, i.e. `MEM_LOAD_RETIRED.L3_MISS`. By default, most sampling profilers sample on cycles, but it's not necessary the strict rule.

After preparations are done, we enable counting and let the benchmark go. We configured PMC to count cycles, so it will be incremented every cycle. Eventually it will overflow. At the time the counter overflows HW will raise PMI. Profiling tool is configured to capture PMIs and has Interrupt Service Routine (ISR) for handling them. Inside this routine we do multiple steps. First of all, we disable counting. After that we record the instruction which

was executed by the CPU at the time the counter overflowed. After that 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 of instructions. To achieve this, we can set the counter to -1 million, so that it will overflow after every 1 million of instructions. This value is usually referred to as “sample after” value.

We repeat the process many times to build sufficient collection of samples. If we later aggregate those samples, we could build a histogram of the hottest places in our program like the one shown on fig. 11. This view gives us the breakdown of the CPU time by function sorted in descending order.

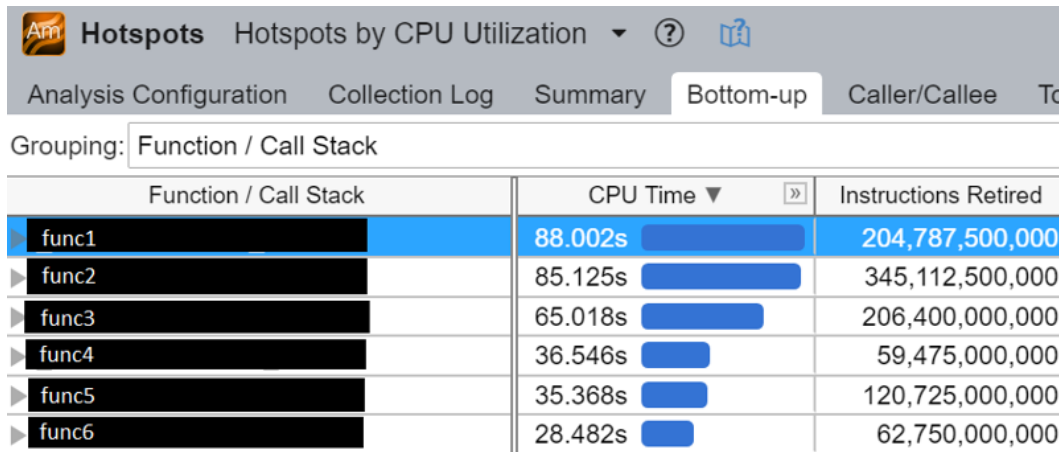


Figure 11: Intel® VTune™ Profiler Bottom up view.

**Personal Experience:** Profiling heavy-templated C++ code can be tricky, especially if the hotspot is in one of functions that is instantiated through a chain of templates. Mangled name of such a function can be long. I profiled applications with functions that have more than 300 characters in their mangled name. Hopefully, one can find the place in the code for such a hotspot with the help of debug information. If you simply double-click on a function, Intel® VTune™ Profiler will get you to the source code of the function.

If you further double-click on a function, it will get you to the source code level view like the one shown on fig. 12. Being able to see profiling data for functions that were inlined as well as source code associated with particular assembly instruction, requires the application being built with debug information (`-g`<sup>43</sup>) in the first place. Users can further limit the amount of debug information to just line numbers of the symbols as they appear in the source code (`-gline-tables-only`). If a user doesn’t need full debug experience, having line numbers is enough for profiling the application.<sup>44</sup>

Similar process is implemented in Linux `perf record/report` tool. Readers interested in more details about sampling with `perf record` can visit [easyp perf](#) blog. Below is the output of `perf record/report` tool:

<sup>43</sup>Equivalent to `-gdwarf` on Linux.

<sup>44</sup>There were cases when LLVM transformation passes incorrectly treated presence of debug intrinsics and made different optimizations when there was no debug information. Line numbers are stored as metadata, which reduces the probability of being hit by those kind of bugs.

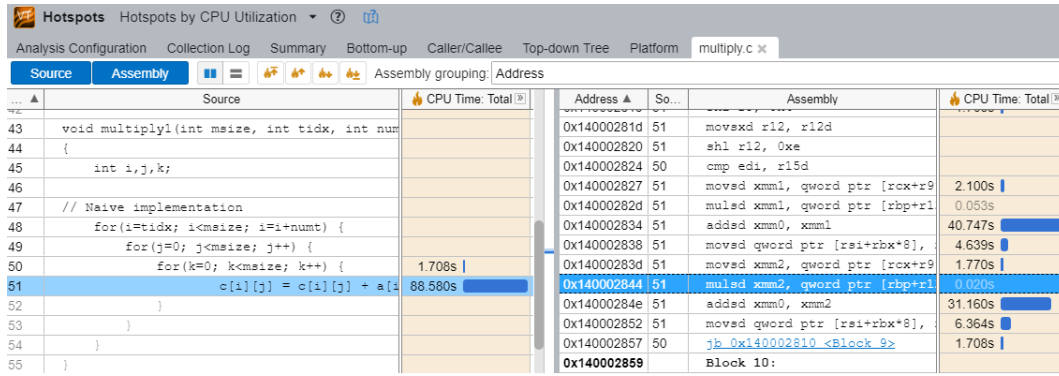


Figure 12: Intel® VTune™ Profiler source code and assembly view.

```
$ perf record -- ./a.exe
$ perf report -n --stdio
# Samples: 434K of event 'cycles'
# Overhead      Samples  Shared Object  Symbol
# .....
68.53%          297420    a.exe          [.] foo
15.76%           68398    a.exe          [.] bar
3.64%           15797     a.exe          [.] main
```

### 5.4.3 Collecting Call Stacks

Often when sampling, we want to know what path led to a particular performance event. Consider situation when the hot function `foo` has multiple callers. Analyzing all of its callers might be time consuming, as we want to focus only on those callers that contribute to `foo` consuming biggest amount of time. Knowing which edges of the call graph are hot can be done by collecting call stack along with every sample.

Getting call stacks in Linux `perf` is possible with 3 methods:

1. Frame pointers (`perf record --call-graph fp`). Requires binary being built with `--fnoomit-frame-pointer`. Historically, frame pointer (RBP) was used for debugging, since it allows to get the call stack without popping all the arguments from the stack. Frame pointer can tell the return address immediately. However, it consumes 1 register just for this purpose, so it was expensive. It is also used for profiling, since it enables cheap stack unwinding.
2. DWARF debug info (`perf record --call-graph dwarf`). Requires binary being built with DWARF debug information `-g` (`-gline-tables-only`).
3. Intel LBR Hardware feature `perf record --call-graph lbr`. Not as deep call graph as first 2 methods. See more information about LBR in the sec. 6.2.

## 5.5 Static Performance Analysis

Today we have extensive tooling for static code analysis. For C and C++ languages we have such well known tools like [Clang Static Analyzer](#), [Klocwork](#), [Cppcheck](#) and others<sup>45</sup>. They aim at checking correctness and semantics of the code. Likewise, there are tools that try to address performance aspect of the code. Static performance analyzers don't run the actual

<sup>45</sup>[https://en.wikipedia.org/wiki/List\\_of\\_tools\\_for\\_static\\_code\\_analysis#C,\\_C++](https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis#C,_C++)

code. Instead they simulate the code as if it is executed on a real HW. Statically predicting performance is almost impossible, so there are many limitations to this type of analysis.

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

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

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

### 5.5.1 Static vs. Dynamic Analyzers

**Static tools** don't run the actual code but try to simulate the execution keeping as much microarchitectural details as they can. They are not capable of doing real measurements (execution time, performance counters) because they don't run the code. The upside here is that you don't need to have the real HW and can simulate the code for different CPU generations. Another benefit is that you don't need to worry about consistency of the results: static analyzers will always give you stable output, because simulation (in comparison with execution on a real hardware) is not biased in any way. The downside of static tools is that usually they can't predict and simulate everything inside modern CPU: they are based on some model which may have bugs and limitations in it. Examples of static performance analyzers are [IACA](#)<sup>46</sup> and [llvm-mca](#)<sup>47</sup>.

**Dynamic tools** are based on running the code on the real HW and collecting all sorts of information about the execution. This is the only 100% reliable method of proving any performance hypothesis. As a downside, usually you are required to have privileged access rights to collect low-level performance data like PMCs. It's not always easy to write a good benchmark and measure what you want to measure. Finally, you need to filter the noise and different kinds of side effects. Examples of dynamic performance analyzers are Linux perf, [likwid](#)<sup>48</sup> and [uarch-bench](#)<sup>49</sup>. Examples of usage and output for the tools mentioned above can be found on [easyperf blog](#)<sup>50</sup>.

**Personal Experience:** I use those tools whenever I need to explore some interesting CPU microarchitecture effect. Static and low-level dynamic analyzers (like likwid and uarch-bench) allow to observe HW effects in practice while doing performance experiments. They are a great help for building up your mental model of how CPU works.

<sup>46</sup><https://software.intel.com/en-us/articles/intel-architecture-code-analyzer>

<sup>47</sup><https://llvm.org/docs/CommandGuide/llvm-mca.html>

<sup>48</sup><https://github.com/RRZE-HPC/likwid>

<sup>49</sup><https://github.com/travisdowns/uarch-bench>

<sup>50</sup><https://easyperf.net/blog/2018/04/03/Tools-for-microarchitectural-benchmarking>

## 5.6 Compiler optimization reports

Nowadays, software development very much relies on compiler to do performance optimizations. Compilers play very important role in speeding up our software. Usually, developers leave this job to compilers, interfering only when they see opportunity to improve something compilers cannot accomplish. Fair to say, this is a good default strategy. For better interaction, compilers provide optimization reports which developers can use for performance analysis.

Sometimes you want to know if some function was inlined, or loop was vectorized, unrolled, etc. If it was unrolled, what is the unroll factor? There is a hard way to know this: by studying generated assembly instructions. Unfortunately, not all people are comfortable at reading assembly language. This can be especially hard if the function is big, it calls other functions or has many loops that were also vectorized, or if compiler created multiple versions of the same loop. Hopefully, most compilers, including GCC, ICC and Clang provide optimization reports to check what optimizations were done for particular piece of code. Listing 3 shows example of the loop that was not vectorized by clang 6.0:

**Listing 3** 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 opt report in clang you need to pass `-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 4 with run-time trip count
    [-Rpass=loop-unroll]
    for (unsigned i = 1; i < N; i++) {
    ^
```

By checking 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 existence of vector dependency in the loop on line 5 in Listing 3. Value that is loaded by `c[i-1]` depends on the store from previous iteration. We can fix this by swapping lines 6 and 7 without changing semantics of the function as shown in Listing 4.

In opt report we can see that the loop was now vectorized:

```
$ clang -O3 -Rpass-analysis=.* -Rpass=.* -Rpass-missed=.* a.c -c
a.cpp:5:3: remark: vectorized loop (vectorization width: 4, interleaved
count: 2) [-Rpass=loop-vectorize]
    for (unsigned i = 1; i < N; i++) {
    ^
```



**Listing 4 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 }

```

Compiler optimization reports not only help in finding missed optimization opportunities and explain why that happened, but also are useful for testing hypothesis. Compiler often decides whether certain transformation will be beneficial based on its cost model analysis. But it doesn't always make the optimal choice which we can be tuned further. One can detect missing optimization in a report and provide a hint to a compiler by using `#pragma`, attributes, compiler built-ins, etc. See example of using such hints on [easyperf blog](#)<sup>51</sup>. As always, verify your hypothesis by measuring it in practical environment.

**Personal Experience:** Compiler optimization reports could be one of the key items in your toolbox. It is a fast way to check what optimizations were done for particular hotspot and see if some important ones failed. I have found many improvement opportunities by using opt reports.

## 5.7 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 of how application executes. Both HW and SW provide data that can be used for performance monitoring.
- Code instrumentation allows to track many things in the program but causes relatively large overhead both on the development and runtime side. While developers do not manually instrument their code these day very often, this approach is still relevant for automated processes, e.g. PGO.
- Tracing is conceptually similar to instrumentation and is useful for exploring anomalies in the system. Tracing allows us to catch entire sequence of events with timestamps attached to each event.
- Workload Characterization is a way to compare and group applications based on their runtime behavior. Once characterized, specific recipes could be followed to find optimization headrooms in the program.
- Sampling skips the large portion of the program execution and take just one sample that is supposed to represent the entire interval. Even though, sampling usually gives precise enough distributions. The most well-known use case of sampling is finding hotspots in the code. Sampling is the most popular analysis approach, since it doesn't require recompilation of the program and has very little runtime overhead.

<sup>51</sup>[https://easyperf.net/blog/2017/11/09/Multiversioning\\_by\\_trip\\_counts](https://easyperf.net/blog/2017/11/09/Multiversioning_by_trip_counts)



- Generally counting and sampling incur very low runtime overhead (usually below 2%). Counting gets more expensive once you start multiplexing between different events (5-15% overhead), sampling gets more expensive with increasing sampling frequency [Nowak and Bitzes, 2014]. Consider using user-mode sampling for analyzing long-running workloads or when you don't need very accurate data.
- There are tools that try to statically analyze performance of the code. Such tools simulate the piece of code instead of executing it. Many limitations and constraints apply to this approach, but you get very detailed and low-level report in return.
- Compiler Opt reports help finding missing compiler optimizations. It may also guide developers into composing new performance experiments.

---

## 6 CPU features for performance analysis

The ultimate goal of performance analysis is to identify the bottleneck and locate the place in the code that associates with it. It can be approached in many different ways because there's no fixed steps you follow.

Usually profiling the application can give quick insights about the hotspots of the application. Sometimes it is everything developers need to do to fix performance inefficiencies. Especially high-level performance problems can often be revealed by profiling. For example, consider a situation when you profile an application with an interest on a particular function. According to your mental model of the application, you expect that function to be cold. But when you open the profile, you see it consumes a lot of time and is called large number of times. Based on that information you can apply techniques like caching or memoization to reduce amount of calls to that function and expect to see significant performance gains.

However, when all the major performance inefficiencies are fixed, but you still need to squeeze more performance from your application, basic information like profiling might not be enough. Here is when you might need additional support from the CPU to understand where performance bottleneck is.

**Personal Experience:** When I was starting with performance optimization work, I usually just profiled the app and tried to grasp through the hotspots of the benchmark hoping to find something there. This often led me to random experiments with unrolling, vectorization, inlining, you name it. I'm not saying it's always a losing strategy. Sometimes you can be lucky to get big performance boost from random experiments. But usually you need to have very good intuition and luck.

Modern CPUs are getting new features that enhance performance analysis in different ways. Using those features greatly simplifies finding low-level issues like cache-misses, branch mis-predictions, etc. In this chapter we will take a look at few HW performance monitoring capabilities of modern Intel CPUs<sup>52</sup> that provide insights on how your code looks like from the CPU perspective and how to make it more CPU-friendly. Identifying the problem to fix on a CPU level can be a challenging task and involves at least basic understanding of CPU microarchitecture, so you might want to revisit sec. 3.

### 6.1 Top-Down Microarchitecture Analysis

Top-Down Microarchitecture Analysis Methodology (TMAM) is a very powerful technique for identifying CPU bottlenecks in the program. It is a robust and formal methodology which is easy to use even for unexperienced developers. The best part of this methodology is that it does not require a developer to have deep understanding of the microarchitecture and PMCs in the system, and still efficiently find CPU bottlenecks. However, it does not automatically fix problems, otherwise this book would not exist.

At a high-level, TMAM identifies what was stalling execution of every instruction in the program. The bottleneck can be related to one of the 4 components: Front End Bound, Back End Bound, Retiring, Bad Speculation. Figure 13 illustrates this concept. Here is a short guide how to read this diagram. As we know from sec. 3, there are internal buffers in the CPU that keep track of information about instructions that are being executed. Whenever

---

<sup>52</sup>Most of the features have their counterparts in CPUs from other vendors like AMD, ARM and others. Look for more details in corresponding sections.

new instruction gets fetched and decoded, new entries in those buffers are allocated. If uop for instruction was not allocated during particular cycle of execution it could be for 2 reasons: we were not able to fetch and decode it (Front End Bound) or Back End was overloaded with work and resources for new uop could not be allocated (Back End Bound). Uop that was allocated and scheduled for execution but not retired is related to Bad Speculation bucket. Example of such a uop can be some instruction that was executed speculatively, but later was proven to be on a wrong program path and was not retired. Finally, Retiring is the bucket where we want all our uops to be, although there are exceptions<sup>53</sup>.

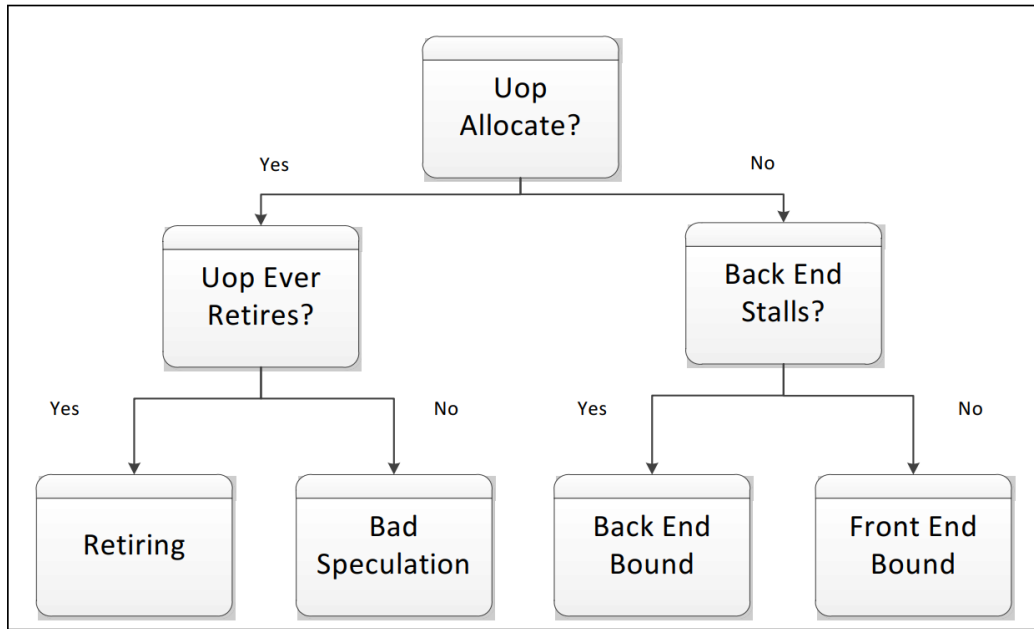


Figure 13: TMAM concept

Figure 13 gives a breakdown for every instruction in a program. However, analyzing every single instruction in the workload is definitely an overkill and of course TMAM doesn't do that. Instead we are usually interested in knowing what is stalling the program as a whole. To accomplish this goal TMAM observes execution of the program by collecting specific metrics (PMCs). Based on those metrics, it characterizes application by relating it to one of the 4 high-level buckets. To have better detalization of the CPU performance bottlenecks in the program, there are nested categories in each high-level bucket (see Figure 14). We run the workload several times<sup>54</sup> each time focusing on specific metrics and drilling down until we get to the more detailed classification of performance bottleneck. For example, initially we collect metrics for 4 main buckets: **Front End Bound**, **Back End Bound**, **Retiring**, **Bad Speculation**. Say, we found out that the big portion of the program execution was **Memory Bound**. Next step is to run the workload again and collect metrics specific for **Memory Bound** bucket only (drilling down). The process is repeated until we know the exact root cause.

In reality, performance could be limited by multiple factors. E.g. application can experience big number of branch mispredicts (**Bad Speculation**) and cache misses (**Back End Bound**) at the same time. In this case TMAM will drill down into multiple buckets simultaneously and will identify the impact that each type of bottleneck makes on performance of a program.

<sup>53</sup><https://easyperf.net/blog/2018/11/08/Using-denormal-floats-is-slow-how-to-detect-it>

<sup>54</sup>In reality, it is sufficient to run the workload once to collect all the metrics required for TMAM. Profiling tools achieve by multiplexing between different performance events during that single run.

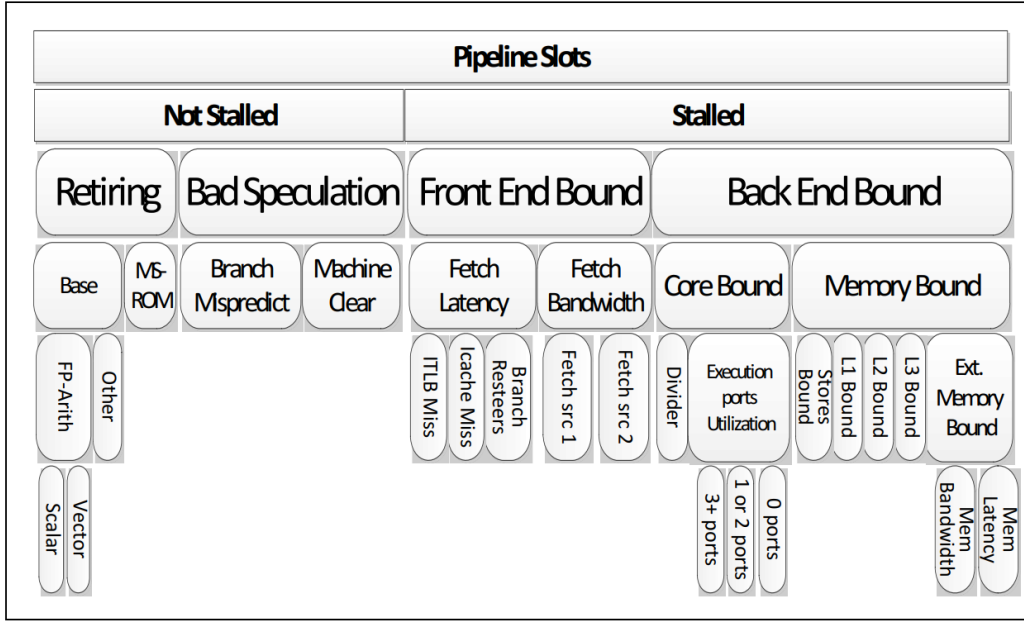


Figure 14: TMAM metrics

Analysis tools such as Intel VTune Profiler and Linux `perf` can calculate all the metrics with a single run of the benchmark. <sup>55</sup>

In later sections we will find out that many TMAM metrics are expressed in percentage of all pipeline slots (see sec. 4.5) that were available during execution of the program. It takes into account full bandwidth of the processor and gives us accurate representation of CPU microarchitecture utilization.

After we identified the type of performance bottleneck in the program, we would be interested to know where exactly in the code it is happening. Second stage of TMAM is locating the source of the problem down to exact line of code and assembly instruction. Analysis methodology provides exact PMC for each category of performance problem that one should use. Then developer can use this PMC to find what piece of code contributes to the biggest number of performance events that we are interested in. This correspondence can be found in [TMA metrics](#)<sup>56</sup> table in “Locate-with” column. For example, one should sample on `MEM_LOAD_RETIRED.L3_MISS_PS` performance counter on Skylake processor to locate the problem in application with high `DRAM_Bound` metric.

### 6.1.1 TMAM in Intel® VTune™ Profiler

Originally TMAM was developed in Intel VTune Profiler under “General Exploration” analysis<sup>57</sup>. Figure 15 shows analysis summary for `7-zip benchmark`<sup>58</sup>. On the diagram you can see that significant amount of execution time was wasted due to CPU `Bad Speculation` and in particular, due to mispredicted branches.

The beauty of the tool is that you can click on the metric you are interested in and the tool

<sup>55</sup>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.

<sup>56</sup>[https://download.01.org/perfmon/TMA\\_Metrics.xlsx](https://download.01.org/perfmon/TMA_Metrics.xlsx)

<sup>57</sup><https://software.intel.com/en-us/vtune-help-general-exploration-analysis>

<sup>58</sup><https://github.com/llvm-mirror/test-suite/tree/master/MultiSource/Benchmarks/7zip>

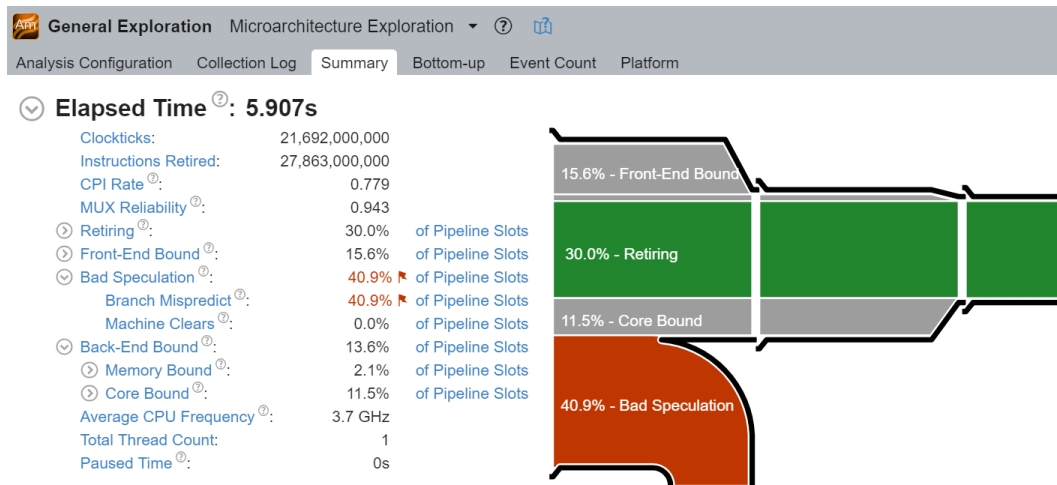


Figure 15: Intel VTune Profiler “Microarchitecture Exploration” analysis.

will get you to the page that shows top functions that contribute to that particular metric. For example, if you click on **Bad Speculation** metric, you will see something like what is shown on Figure 16.

Function / Call Stack	CPU Time	Clockticks	Instructions Retired	CPI Rate	Retiring	Front-End Bound	Bad Speculation	Back-End Bound
LzmaDec_DecodeReal2	2.883s	10,608,000,000	13,685,000,000	0.775	44.0%	0.0%	72.1%	0.0%
Bit4_MatchFinder_GetMatches	1.507s	5,304,000,000	3,196,000,000	1.660	7.1%	37.7%	25.9%	29.3%
LzmaEnc_CodeOneBlock	1.021s	3,995,000,000	8,279,000,000	0.483	31.3%	100.0%	15.6%	0.0%
CrcUpdateT4	0.130s	493,000,000	833,000,000	0.592	67.6%	0.0%	0.0%	32.4%
LenEnc_Encode2	0.080s	102,000,000	272,000,000	0.375	0.0%	100.0%	0.0%	0.0%
CBenchRandomGenerator::Generate	0.040s	170,000,000	357,000,000	0.476	0.0%	0.0%	0.0%	100.0%
FillDistancesPrices	0.040s	119,000,000	238,000,000	0.500	0.0%	100.0%	0.0%	0.0%
CBenchmarkInStream::Read	0.040s	170,000,000	374,000,000	0.455	100.0%	0.0%	0.0%	26.5%
clear_page_c_e	0.025s	34,000,000	17,000,000	2.000	0.0%	0.0%	0.0%	100.0%

Figure 16: “Microarchitecture Exploration” Bottom-up view.

From there if you double click on **LzmaDec\_DecodeReal2** function, Intel® VTune™ Profiler will get you to the source level view like the one that is shown on Figure 17.

Highlighted is the line that contributes to the biggest number of branch mispredicts in **LzmaDec\_DecodeReal2** function.

### 6.1.2 TMAM in Linux Perf

As of version 5.5 of Linux perf (end of 2019), there is `--topdown` option to `perf stat` command<sup>59</sup> that prints top down level 1 metrics, i.e. only 4 high level buckets:

```
$ perf stat --topdown -a -- taskset -c 0 ./7zip-benchmark b
      retiring  bad speculat  FE bound  BE bound
S0-C0    30.8%    41.8%      8.8%    18.6%  <==
S0-C1    17.4%     2.3%     12.0%    68.2%
S0-C2    10.1%     5.8%     32.5%    51.6%
S0-C3    47.3%     0.3%     2.9%    49.6%
...
```

<sup>59</sup>[http://man7.org/linux/man-pages/man1/perf-stat.1.html#STAT\\_REPORT](http://man7.org/linux/man-pages/man1/perf-stat.1.html#STAT_REPORT)

Analysis Configuration										Collection Log										Summary										Bottom-up										Event Count										Platform										LzmaDec.c																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																	
Source										Assembly																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																			

Figure 17: “General Exploration” source code and assembly view.

To get values for high-level TMAM metrics, Linux `perf` requires profiling the whole system (-a), this is why we see metrics for all cores. But since we pinned the benchmark to core0 with `taskset -c 0` we can only focus on the first row that corresponds to S0-C0.

To get access to Top-Down metrics level 2, 3, etc. one can use `toplev`<sup>60</sup> tool that is a part of `pmu-tools`<sup>61</sup> written by Andi Kleen. It is implemented in python and invokes Linux `perf` under the hood. You will see examples of using it in the next section. Specific Linux kernel setting must be enabled to be able to use `toplev`, check documentation for more details. To better present the workflow, next section provides step-by-step example of using TMAM to improve performance of memory bound application.

**Personal Experience:** Intel® VTune™ Profiler is an extremely powerful tool, no doubt about it. However, for quick experiments I often use Linux `perf` that is available on every Linux distribution I’m working on. Thus, the motivation for the example in the next section being explored with Linux `perf`.

### 6.1.3 Step1: Identify the bottleneck.

Suppose we have a tiny benchmark (`a.out`) that runs for 8.5 sec. Complete source code of the benchmark can be found on [github](#)<sup>62</sup>.

```
$ time -p ./a.out
real 8.53
```

As a first step we run our application and collect specific metrics that will help us to characterize it, i.e. we try to detect to which category our application belongs. Below are level-1 metrics for our benchmark:<sup>63</sup>

```
$ ~/pmu-tools/toplev.py --core S0-C0 -l1 -v --no-desc taskset -c 0 ./a.out
...
S0-C0 Frontend_Bound: 13.81 % Slots
S0-C0 Bad_Speculation: 0.22 % Slots
S0-C0 Backend_Bound: 53.43 % Slots <==
```

<sup>60</sup><https://github.com/andikleen/pmu-tools/wiki/toplev-manual>

<sup>61</sup><https://github.com/andikleen/pmu-tools>

<sup>62</sup>[https://github.com/dendibakh/dendibakh.github.io/tree/master/\\_posts/code/TMAM](https://github.com/dendibakh/dendibakh.github.io/tree/master/_posts/code/TMAM)

<sup>63</sup>Outputs in this section are trimmed to fit on the page. Do not rely on exact format that is presented.

```
S0-C0  Retiring:          32.53 % Slots
```

Notice, the process is pinned to CPU0 (using `taskset -c 0`) and output of `toplev` is limited to this core only (`--core S0-C0`). By looking at the output, we can tell that performance of the application is bound by the CPU backend. Without trying to analyze it right now, let us drill one level down: <sup>64</sup>

```
$ ~/pmu-tools/toplev.py --core S0-C0 -l2 -v --no-desc taskset -c 0 ./a.out
...
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
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
S0-C0  Retiring.Base:           24.83 % Slots
S0-C0  Retiring.Microcode_Sequencer: 7.65 % Slots
```

Now we see that application' performance is bound by memory accesses. Almost half of the CPU execution resources were wasted waiting for memory requests to complete. Now let us dig one level deeper: <sup>65</sup>

```
$ ~/pmu-tools/toplev.py --core S0-C0 -l3 -v --no-desc taskset -c 0 ./a.out
...
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
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
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 level of caches and go all the way down to the main memory. We can also confirm it if we collect absolute number of L3 cache misses (DRAM hit) for the program. For

<sup>64</sup>Alternatively, we could use `--nodes Backend_Bound` option instead of `-l2` to limit collection of all the metrics, since we know from the first level metrics that application is bound by CPU Backend.

<sup>65</sup>Alternatively, we could use `--nodes Memory_Bound` option instead of `-l3` to limit collection, since we knew from the second level metrics that application is bound by Memory.



Skylake architecture `DRAM_Bound` metric is calculated using `CYCLE_ACTIVITY.STALLS_L3_MISS` performance event. Let's collect it:

```
$ perf stat -e
  cycles,cpu/event=0xa3,umask=0x6,cmask=0x6,name=CYCLE_ACTIVITY.STALLS_L3_MISS/
  ./a.out
32226253316  cycles
19764641315  CYCLE_ACTIVITY.STALLS_L3_MISS
```

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

#### 6.1.4 Step2: Locate the place in the code.

As a second step in TMAM process we would be interested to locate the place in the code where the bottleneck is happening. To do so we could refer to the [TMA metrics](#)<sup>66</sup> table introduced earlier in the chapter. `Locate-with` column denotes performance event that should be used to locate exact place in the code where the issue occurs. For the purpose of our example, in order to find memory accesses that contribute to such high value of `DRAM_Bound` metric (miss in L3 cache) we should sample on `MEM_LOAD_RETIRE.L3_MISS_PS` precise event:<sup>67</sup>

```
$ perf record -e cpu/event=0xd1,umask=0x20,name=MEM_LOAD_RETIRE.L3_MISS/ppp
./a.out

$ 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   a.out          [.] foo
  0.03%      52   [kernel]      [k] get_page_from_freelist
  0.01%       3   [kernel]      [k] free_pages_prepare
  0.00%       1   [kernel]      [k] free_pcppages_bulk
```

Majority of L3 misses are caused by memory accesses in function `foo` inside executable `a.out`. In order to avoid compiler optimizations, function `foo` is implemented in assembly language, which is presented in Listing 5. The “driver” portion of the benchmark is implemented in `main` function as shown in Listing 6. We allocate big enough array `a` to make it not fit in the L3 cache<sup>68</sup>. The benchmark basically generates a random index to array `a` and passes it to `foo` function along with the address of array `a`. Later `foo` function reads this random memory location.<sup>69</sup>

By looking at Listing 5 we can see that all L3-Cache misses in function `foo` are tagged to a single instruction. Now that we know what instruction caused so many L3 misses let's fix it.

<sup>66</sup>[https://download.01.org/perfmon/TMA\\_Metrics.xlsx](https://download.01.org/perfmon/TMA_Metrics.xlsx)

<sup>67</sup>Alternatively, `toplev` tool has `--show-sample` option that will suggest us `perf record` command line that can be used to locate the issue.

<sup>68</sup>L3 cache on the machine I was using is 38,5 MB - Intel(R) Xeon(R) Platinum 8180 CPU.

<sup>69</sup>According to x86 calling conventions ([https://en.wikipedia.org/wiki/X86\\_calling\\_conventions](https://en.wikipedia.org/wiki/X86_calling_conventions)), first 2 arguments land in `rdi` and `rsi` registers respectively.



**Listing 5** Assembly code of function foo.

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

**Listing 6** Source code of function main.

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

### 6.1.5 Step3: Fix the issue.

Because there is a time window between the moment when we get the next address that will be accessed and actual load instruction, we can add a prefetch hint<sup>70</sup> as shown on Listing 7.

This hint improved execution time by 2 seconds which is 30% speedup:

```
$ perf stat -e
cycles,cpu/event=0xa3,umask=0x6,cmask=0x6,name=CYCLE_ACTIVITY.STALLS_L3_MISS/
./a.out
24621931288      cycles
2069238765      CYCLE_ACTIVITY.STALLS_L3_MISS
6,498080824 seconds time elapsed
```

Notice 10x less value for CYCLE\_ACTIVITY.STALLS\_L3\_MISS event. Also, the number for MEM\_LOAD\_RETIRED.L3\_MISS goes from 71M down to 9M.

TMAM is an iterative process, so we now need to repeat the process starting from the Step1.

<sup>70</sup>Documentation about `__builtin_prefetch` can be found at <https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html>.

**Listing 7** 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);
}

```

Likely it will move the bottleneck into some other bucket, in this case Retiring. This was an easy example demonstrating the workflow of TMAM methodology. Analyzing real-world application is unlikely to be that easy. The next entire chapter in this book is organized in a way to be conveniently used with TMAM process. E.g. its sections are broken down to reflect each high-level category of performance bottlenecks. The idea behind such structure is to provide some kind of checklist which developer can use to drive code changes after performance issue has been found. For instance, when developer see that the application they are working on is **Memory Bound**, they can look up sec. 7.3.1 for ideas.

### 6.1.6 Summary

TMAM is great for identifying CPU performance bottlenecks in the code. Ideally when we run it on some application, we would like to see Retiring metric at 100%. This would mean that this application fully saturates CPU. It is possible to achieve results close to this on a toy programs, however, real-world applications are far from getting there. And again, there are cases when high retiring doesn't necessarily mean a good thing<sup>71</sup>. Figure 18 shows expected ranges for TMAM metrics in a well-tuned application. Figure 19 shows top level TMAM metrics for **SPEC CPU2006**<sup>72</sup> benchmark for Skylake CPU generation. Keep in mind, that the numbers are likely to change for newer processors as architects constantly try to improve the CPU design.

	Expected Range of Pipeline Slots in This Category, for a Hotspot in a Well-Tuned:		
Category	Client/Desktop Application	Server/Database/Distributed application	High Performance Computing (HPC) application
Retiring	20-50%	10-30%	30-70%
Back-End Bound	20-40%	20-60%	20-40%
Front-End Bound	5-10%	10-25%	5-10%
Bad Speculation	5-10%	5-10%	1-5%

Figure 18: Expected ranges for TMAM metrics in a well-tuned application. Taken from: <https://software.intel.com/en-us/vtune-cookbook-top-down-microarchitecture-analysis-method>

<sup>71</sup><https://easyperf.net/blog/2018/11/08/Using-denormal-floats-is-slow-how-to-detect-it>

<sup>72</sup><http://spec.org/cpu2006/>

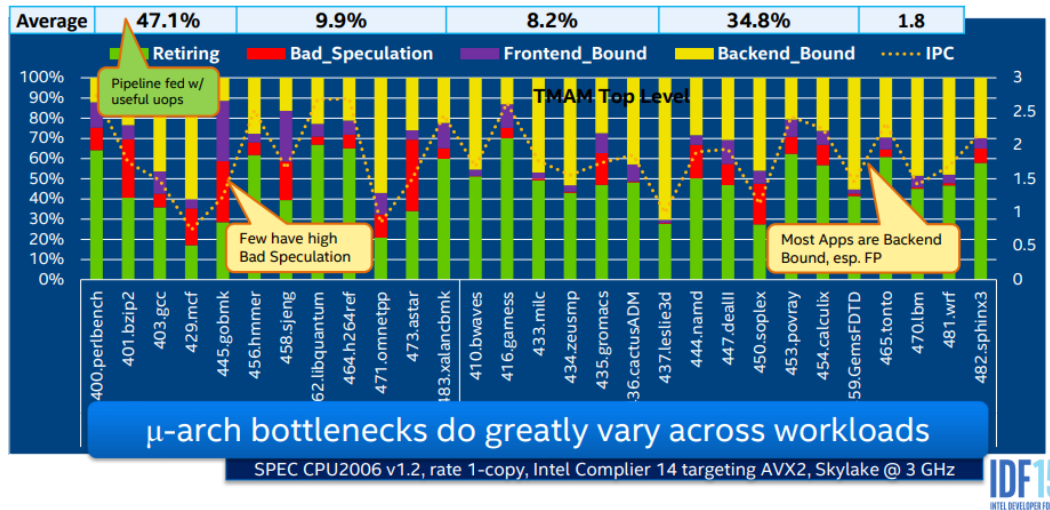


Figure 19: Top Level TMAM metrics for SPEC CPU2006. Taken from: <http://cs.haifa.ac.il/~yosi/PARC/yasin.pdf>, Ahmad Yasin

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

TMAM characterizes the workload which can increase the scope of potential optimizations beyond source code. For example, if the application is memory bound and all possible ways to speed it up on the software level are examined, it is possible to improve the memory subsystem by using faster memory. This enables educated experiments, since the money will be spent only once you found that the program is memory bound and it will benefit from faster memory.

#### Additional resources and links:

- Presentation on TMAM by Ahmad Yasin, URL: <http://intelstudios.edgesuite.net/idf/2015/sf/aep/ARCS002>
- Andi Kleen's blog - pmu-tools, part II: toplev, URL: <http://halobates.de/blog/p/262>.
- Toplev manual, URL: <https://github.com/andikleen/pmu-tools/wiki/toplev-manual>.
- Understanding How General Exploration Works in Intel® VTune™ Profiler, URL: <https://software.intel.com/en-us/articles/understanding-how-general-exploration-works-in-intel-vtune-amplifier-xe>.

## 6.2 Last Branch Record

Modern Intel CPUs have a feature called Last Branch Record (LBR) where CPU continuously logs a number of previously executed branches. But before going into the details, one might ask: *Why we are so interested in branches?* Well, because this is how we are able to determine the control flow of our program. We largely ignore other instructions in a basic block (see sec. 4.9), because branches are always the last instruction in a basic block. Since all instructions in the basic block are guaranteed to be executed once, we can only focus on branches which will “represent” the entire basic block. Thus, it's possible to reconstruct entire line-by-line execution path of the program if we track the outcome of every branch. In fact, this is what

Intel Processor Traces (PT) feature is capable of doing which will discuss in sec. 6.4. LBR feature predates PT and has different use cases and special features.

Thanks to LBR mechanism, CPU can continuously log branches to a set of model-specific registers (MSRs, see sec. 3.2.5.1) in parallel with executing the program, causing minimal slowdown<sup>73</sup>. Hardware logs the “from” and “to” address of each branch along with some additional metadata (see Figure 20). The registers act like a ring buffer that is continuously overwritten and provides only 32<sup>74</sup> most recent branch outcomes. If we collect long enough history of source-destination pairs, we will be able to unwind the control flow of our program, just like a call stack with limited depth.

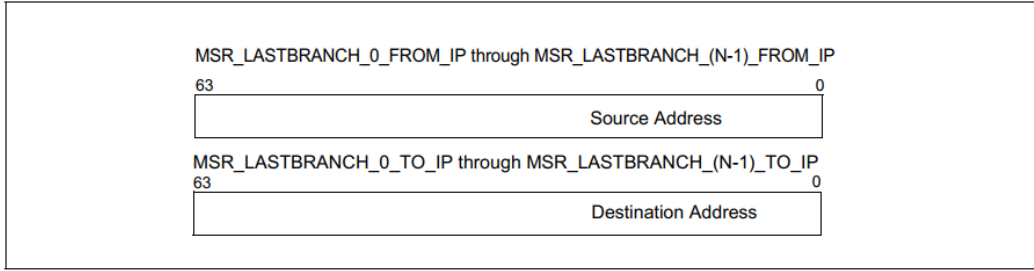


Figure 20: 64-bit Address Layout of LBR MSR. Image taken from [Int, 2020]

With LBRs we can sample branches, but during each sample look at the previous branches inside LBR stack that were executed. This gives reasonable coverage of the control flow in the hot code paths, but does not overwhelm us with too much information, as only a smaller number of the total branches are examined. It is important to keep in mind that this is still sampling, so not every executed branch can be examined. CPU generally executes too fast for that to be feasible.[Kleen, 2016]

- **Last Branch Record (LBR) Stack** — since Skylake provides 32 pairs of MSRs that store source and destination address of recent taken branches.
- **Last Branch Record Top-of-Stack (TOS) Pointer** — contains a pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded.

It is very important to keep in mind that only taken branches are being logged with LBR mechanism. Below is an example that shows how branch results are tracked in LBR stack.

```

----> 4eda10:  mov    edi,DWORD PTR [rbx]
|      4eda12:  test   edi,edi
| --- 4eda14:  jns     4eda1e          <== NOT taken
| |    4eda16:  mov     eax,edi
| |    4eda18:  shl     eax,0x7
| |    4eda1b:  lea     edi,[rax+rdi*8]
|  ↳ 4eda1e:  call    4edb26
|      4eda23:  add     rbx,0x4
|      4eda27:  mov     DWORD PTR [rbx-0x4],eax
|      4eda2a:  cmp     rbx,rbp
---- 4eda2d:  jne     4eda10          <== taken

```

<sup>73</sup>Runtime overhead for the most part of LBR use cases is below 1%. [Nowak and Bitzes, 2014]

<sup>74</sup>Only since Skylake microarchitecture. In Haswell and Broadwell architectures LBR stack is 16 entries deep. Check Intel manual for information about other architectures.

Because JNS branch was not taken, here is what we expect to see in the LBR stack at the moment we executed CALL instruction:

FROM_IP	TO_IP	
...	...	
4eda2d	4eda10	
4eda1e	4edb26	<== LBR TOS

Notice, JNS branch (4eda14 -> 4eda1e) is not logged, because it was not taken.

**Personal Experience:** Untaken branches not being logged might add some additional burden for analysis but usually doesn't complicate it too much. We can still unwind LBR stack since we know that the control flow was sequential from TO\_IP(N-1) to FROM\_IP(N).

Starting from Haswell LBR entry received additional component to detect branch misprediction. There is a dedicated bit for it in the LBR entry (see [Int, 2020, Volume 3B, Chapter 17]). Since Skylake additional LBR\_INFO component was added to LBR entry which received additional Cycle Count field which counts elapsed core clocks since last update to the LBR stack. There are important applications to those additions which we will discuss later. Exact format of LBR entry for specific processor can be found in [Int, 2020, Volume 3B, Chapters 17,18].

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

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

### 6.2.1 Collecting LBR stacks

On Linux perf one can collect LBR stacks using the command below:

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

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

Because each collected sample captures entire LBR stack (32 last branch records), size of collected data (`perf.data`) is significantly bigger than sampling without LBRs. Below is the Linux perf command one can use to dump the contents of collected branch stacks:

```
$ perf script -F brstack &> dump.txt
```

If we look inside the `dump.txt` (it might be big) we will see something like is shown below:

```
...
0x4edabd/0x4edad0/P/-/-/2 0x4edaf9/0x4edab0/P/-/-/29
0x4edabd/0x4edad0/P/-/-/2 0x4edb24/0x4edab0/P/-/-/23
0x4edadd/0x4edb00/M/-/-/4 0x4edabd/0x4edad0/P/-/-/2
0x4edb24/0x4edab0/P/-/-/24 0x4edadd/0x4edb00/M/-/-/4
0x4edabd/0x4edad0/P/-/-/2 0x4edb24/0x4edab0/P/-/-/23
```



```
$ 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 inside `bar` function, 22% of branches are calls from `foo` to `bar`, and so on. Notice, how `perf` switched from `cycles` events to analyzing LBR stacks: only 670 samples were collected, yet we have entire LBR stack captured with every sample. This gives us  $670 * 32 = 21440$  LBR entries (branch outcomes) for analysis.<sup>78</sup>

Most of the time it's possible to determine the location of the branch just from the line of code and target symbol. However, theoretically, one could write a code with two `if` statements written on a single line. Also, when expanding macro definition, all the expanded code gets the same source line which is another situation when this might happen. This issue does not totally block the analysis but only makes it a little more difficult. In order to disambiguate two branches you likely need to analyze raw LBR stacks yourself (see example on [easypf](#)<sup>79</sup> blog).

#### 6.2.4 Analyze branch misprediction rate.

It's also possible to know misprediction rate for hot branches<sup>80</sup>:

```
$ perf record -e cycles -b -- ./a.exe
$ perf report -n --sort symbol_from,symbol_to -F
+mispredict,srcline_from,srcline_to --stdio
# Samples: 657K of event 'cycles'
# Event count (approx.): 657888
# Overhead Samples Mis From Line To Line Source Sym Target Sym
# .....
46.12%    303391 N dec.c:36 dec.c:40 LzmaDec LzmaDec
22.33%    146900 N enc.c:25 enc.c:26 LzmaFind LzmaFind
6.70%     44074 N lz.c:13 lz.c:27 LzmaEnc LzmaEnc
6.33%     41665 Y dec.c:36 dec.c:40 LzmaDec LzmaDec
```

In this example<sup>81</sup>, lines that correspond to function `LzmaDec` are of particular interest to us.

<sup>78</sup>The report that `perf` generates when LBR entries are being analyzed instead of sampling events can be improved in order not to confuse users.

<sup>79</sup><https://easypf.net/blog/2019/05/06/Estimating-branch-probability>

<sup>80</sup>Adding `-F +srcline_from,srcline_to` slows down building report. Hopefully in newer versions of `perf` decoding time will be improved.

<sup>81</sup>This example is taken from the real world application, 7-zip benchmark: <https://github.com/llvm-mirror/test-suite/tree/master/MultiSource/Benchmarks/7zip>. Although the output of `perf` report is trimmed a little bit to fit nicely on a page.



Using the reasoning from sec. 6.2.3 we can conclude that the branch on source line `dec.c:36` is the most executed branch in the benchmark. In the output that `perf` provides we can spot two entries that correspond to `LzmaDec` function: one with Y and one with N letters. Analyzing those two entries together gives us misprediction rate of the branch. In this case we know that the branch on line `dec.c:36` was predicted 303391 times (corresponds to N) and was mispredicted 41665 times (corresponds to Y), which gives us 88% prediction rate.

Linux `perf` calculates misprediction rate by analyzing each LBR entry and extracting misprediction bits from it. So that for every branch we have a number of times it was predicted correctly and a number of mispredictions. Again, due to the nature of sampling, it is possible that some branches might have N entry but no corresponding Y entry. It could mean that there are no LBR entries for that branch being mispredicted, but it doesn't necessary mean that prediction rate equals to 100%.

### 6.2.5 Precise timing of machine code.

As was discussed in sec. 6.2, since Skylake architecture LBR entries have `Cycle Count` information. This additional field gives us number of cycles elapsed between two taken branches. If the target address in previous LBR entry is the beginning of some basic block (BB) and source address of current LBR entry is the last instruction of the same basic block, then the cycle count is the latency of this basic block. For example:

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

Suppose we have two entries in the LBR stack:

FROM_IP	TO_IP	Cycle Count	
...	...	...	
40060a	400618	10	
400628	400644	5	<== LBR TOS

Given that information, we know that there was one occurrence when the basic block that starts at offset 400618 was executed in 5 cycles. If we collect enough samples, we could plot a probability density function of the latency for that basic block (see figure 21). This chart was compiled by analyzing all LBR entries that satisfy the rule described above. For example, basic block was executed in ~75 cycles only 4% of time but more often it was executed between 260 and 314 cycles. This block has a random load inside a huge array which doesn't fit in CPU L3 cache, so the latency of the basic block largely depends on this load. There are 2 important spikes on the chart that is shown on Figure 21: first around 80 cycles corresponds to L3 cache hit and the second spike around 300 cycles corresponds to L3 cache miss where the load request goes all the way down to the main memory.

This information can be used for further fine-grained tuning of this basic block. This example might benefit from memory prefetching which we will discuss in sec. 7.3.1.2. Also, this cycle information can be used for timing loop iterations where every loop iteration ends with a taken branch (backedge).

Example of how one can build probability density function for latency of an arbitrary basic block can be found on [easyperf blog](https://easyperf.net/blog/2019/04/03/Precise-timing-of-machine-code-with-Linux-perf)<sup>82</sup>. However, in newer versions of Linux `perf`, getting this

<sup>82</sup><https://easyperf.net/blog/2019/04/03/Precise-timing-of-machine-code-with-Linux-perf>



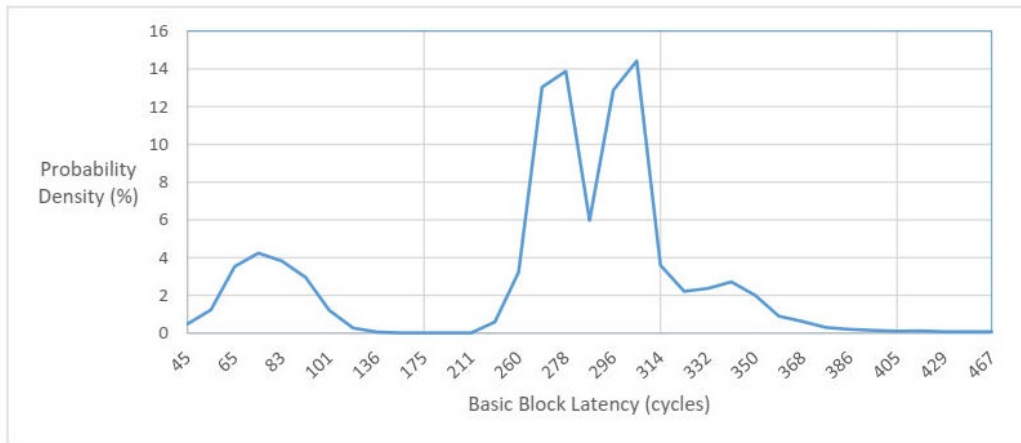


Figure 21: Probability density function for latency of the basic block that starts at address 0x400618.

information is much easier. For example<sup>83</sup>:

```
$ perf record -e cycles -b -- ./a.exe
$ perf report -n --sort symbol_from,symbol_to -F
+cycles,srcline_from,srcline_to --stdio
# Samples: 658K of event 'cycles'
# Event count (approx.): 658240
# Overhead Samples BBCycles FromSrcLine ToSrcLine
# .....
2.82% 18581 1 dec.c:325 dec.c:326
2.54% 16728 2 dec.c:174 dec.c:174
2.40% 15815 4 dec.c:174 dec.c:174
2.28% 15032 2 find.c:375 find.c:376
1.59% 10484 1 dec.c:174 dec.c:174
1.44% 9474 1 enc.c:1310 enc.c:1315
1.43% 9392 10 7zCrc.c:15 7zCrc.c:17
0.85% 5567 32 dec.c:174 dec.c:174
0.78% 5126 1 enc.c:820 find.c:540
0.77% 5066 1 enc.c:1335 enc.c:1325
0.76% 5014 6 dec.c:299 dec.c:299
0.72% 4770 6 dec.c:174 dec.c:174
0.71% 4681 2 dec.c:396 dec.c:395
0.69% 4563 3 dec.c:174 dec.c:174
0.58% 3804 24 dec.c:174 dec.c:174
```

Several not significant lines were removed from the output of `perf record` in order to make it fit on the page. If we now focus on branch which source and destination is `dec.c:174`<sup>84</sup>, we can find multiple lines associated with it. Linux `perf` sorts entries by overhead first, which requires us to manually filter entries for the branch which we are interested in. In fact, if we

<sup>83</sup>Adding `-F +srcline_from,srcline_to` slows down building report. Hopefully in newer versions of `perf` decoding time will be improved.

<sup>84</sup>In the source code line `dec.c:174` expands a macro which has self-contained branch, that's why source and destination happens to be on the same line.

filter them, we will get the latency distribution for the basic block that ends with this branch as shown in the table 3. Later user can plot this data and get chart similar to Figure 21.

Table 3: Probability density for basic block latency.

Cycles	Number of samples	Probability density
1	10484	17.0%
2	16728	27.1%
3	4563	7.4%
4	15815	25.6%
6	4770	7.7%
24	3804	6.2%
32	5567	9.0%

Currently timed LBR is the most precise cycle-accurate source of timing information in the system.

#### 6.2.6 Estimating branch outcome probability.

Later in sec. 7.2 we will discuss importance of code layout for performance. Going forward a little bit, having hot path in a fall through manner<sup>85</sup> generally improves performance of the program. Considering single branch, knowing that `condition` is 99% of the time false or true is essential for compiler to make better optimizing decisions.

Intel LBR feature allows to get this data without instrumenting the code. As the outcome from the analysis, user will get a ratio between true and false outcomes of the condition, i.e. how much time the branch was taken and how much not taken. This feature especially shines when analyzing indirect jumps (switch statement) and indirect calls (virtual calls). One can find examples of using it on a real world application on [easyperf blog](#)<sup>86</sup>.

#### 6.2.7 Other use cases

- **Profile guided optimizations.** LBR feature can provide profiling feedback data for compilers and other optimizers. LBR can be a better choice as opposed to static code instrumentation when runtime overhead is considered.
- **Capturing function arguments.** When LBR features is used together with PEBS (see sec. 6.3), it is possible to capture function arguments, since according to [x86 calling conventions](#)<sup>87</sup> first few arguments of a callee land in registers which are captured by PEBS record. [Int, 2020, Appendix B, Chapter B.3.3.4]
- **Basic Block Execution Counts.** Since all the basic blocks between a branch IP (source) and the previous target in the LBR stack are executed exactly once, it's possible to evaluate the execution rate of basic blocks inside a program. This process involves building a map of starting addresses of each basic block and then traversing collected LBR stacks backwards. [Int, 2020, Appendix B, Chapter B.3.3.4]

<sup>85</sup>I.e. when outcomes of branches are not taken.

<sup>86</sup><https://easyperf.net/blog/2019/05/06/Estimating-branch-probability>

<sup>87</sup>[https://en.wikipedia.org/wiki/X86\\_calling\\_conventions](https://en.wikipedia.org/wiki/X86_calling_conventions)

### 6.3 Processor event-based sampling

Processor Event-Based Sampling (PEBS) is another very useful feature in Intel CPUs that provides many different ways to enhance performance analysis. Similar to Last Branch Record (see sec. 6.2), PEBS is used while profiling the program to capture additional data with every collected sample. PEBS feature was introduced by processors based on Intel NetBurst microarchitecture.

Set of additional data has defined format which is called PEBS record. When a performance counter is configured for PEBS, processor saves the contents of PEBS buffer which is later stored in memory. This record contains the architectural state of the processor (state of the general-purpose registers, EIP register, and EFLAGS register) and more. The content layout of a PEBS record varies across different implementations that support PEBS. See [Int, 2020, Volume 3B, Chapter 18.6.2.4 Processor Event Based Sampling (PEBS)] for details of enumerating PEBS record format. PEBS Record Format for Intel Skylake CPU is shown on Figure 22.

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 22: PEBS Record Format for 6th Generation, 7th Generation and 8th Generation Intel Core Processor Families [Int, 2020, Volume 3B, Chapter 18]

Users can check if PEBS are enabled by executing `dmesg`:

```
$ dmesg | grep PEBS
[ 0.061116] Performance Events: PEBS fmt1+, IvyBridge events, 16-deep
LBR, full-width counters, Intel PMU driver.
```

Linux `perf` doesn't export the raw PEBS output as it does for LBR, instead it processes it and saves only subset of data depending on a particular need. So, it's not possible to access collection of raw PEBS records with Linux `perf`. Although, Linux `perf` provides some PEBS data processed from raw samples which can be accessed by `perf report -D`. To dump raw PEBS records one can use `pebs-grabber`<sup>88</sup> tool.

There is a number of benefits that PEBS mechanism brings to performance monitoring which we will discuss in the next section.

<sup>88</sup><https://github.com/andikleen/pmu-tools/tree/master/pebs-grabber>. Requires root access.

### 6.3.1 Precise events

One of the major problems in profiling is pinpointing exact instruction that caused particular performance event. As discussed in sec. 5.4, interrupt-based sampling is based on counting specific performance events and waiting until it overflows. When an overflow interrupt happens, it takes a processor some amount of time to stop the execution and tag instruction that caused the overflow. This is especially difficult on modern complex out-of-order CPU architectures.

It introduces the notion of skid, which is defined as the distance between the IP that caused the event to the IP where the event is tagged (in the IP field inside PEBS record). Skid makes it difficult to discover the instruction which is actually causing the performance issue. Consider application with big number of cache misses and the hot assembly code that look like this:

```
; load1
; load2
; load3
```

Profiler might attribute `load3` as the instruction that causes big number of cache misses while in reality `load1` is the instruction to blame. This usually causes a lot of confusion for beginners. Interested readers could learn more about underlying reasons for such issues on [Intel Developer Zone website](#)<sup>89</sup>.

The problem with the skid is mitigated by having the processor itself store the instruction pointer (along with other information) in a PEBS record. The `EventingIP` in the PEBS record will indicate the instruction that caused the event. This needs to be supported by the hardware and is typically available only for a subset of supported events, called “Precise Events”. Complete list of precise events for specific microarchitecture can be found in [Int, 2020, Volume 3B, Chapter 18]. Below listed precise events for the Skylake Microarchitecture:

```
INST_RETIRED.*
OTHER_ASSISTS.*
BR_INST_RETIRED.*
BR_MISP_RETIRED.*
FRONTEND_RETIRED.*
HLE_RETIRED.*
RTM_RETIRED.*
MEM_INST_RETIRED.*
MEM_LOAD_RETIRED.*
MEM_LOAD_L3_HIT_RETIRED.*
```

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

TMAM methodology (see sec. 6.1) heavily relies on precise events to locate the source of inefficient execution of the code. Example of using precise event to mitigate skid can be found on [easyperf blog](#)<sup>90</sup>. Users of Linux `perf` should add `ppp` suffix to the event to enable precise tagging:

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

<sup>89</sup><https://software.intel.com/en-us/vtune-help-hardware-event-skid>

<sup>90</sup><https://easyperf.net/blog/2018/08/29/Understanding-performance-events-skid>

### 6.3.2 Lower sampling overhead

Frequently generating interrupts and having analysis tool itself capture program state inside interrupt service routine is very costly since it involves OS interaction. This is why some hardware allows automatically sampling multiple times to a dedicated buffer without any interrupts. Only when the dedicated buffer is full, processor raises interrupt and the buffer gets flushed to memory. This has lower overhead than traditional interrupt-based sampling.

When a performance counter is configured for PEBS, an overflow condition in the counter will arm PEBS mechanism. On the subsequent event following overflow, the processor will generate a PEBS event. On a PEBS event, the processor stores the PEBS record in the PEBS buffer area, clear the counter overflow status, and reload the counter with the initial value. If the buffer is full, the CPU will raise an interrupt. [Int, 2020, Volume 3B, Chapter 18]

Note that PEBS buffer itself is located in memory and its size is configurable. Again, it is the job of performance analysis tool to allocate and configure the memory area for CPU to be able to dump PEBS records in it.

### 6.3.3 Analyzing memory accesses

Memory accesses are critical factor for performance of many applications. With PEBS it is possible to gather detailed information about memory accesses in the program. The feature that allows this is called “Data Linear Address” (DLA). To provide additional information about sampled loads and stores DLA leverages the following fields inside PEBS facility (see Figure 22):

- Data Linear Address (0x98)
- Data Source Encoding (0xA0)
- Latency value (0xA8)

If the performance event supports DLA and it is enabled, CPU will dump memory addresses and latency of the sampled memory access. Keep in mind, this feature does not trace all the stores and loads, otherwise, the overhead would be too big. Instead, it samples on memory accesses, i.e. analyzes only one from 1000 accesses or so. It is customizable how much samples per second you want.

One of the most important use cases for DLA feature is detecting [True/False sharing](#)<sup>91</sup> which we will discuss in sec. 8.7. Linux `perf c2c` tool heavily relies on DLA data to find contested memory accesses which could experience True/False sharing.

Also, with the help of DLA user can get general statistics about memory accesses in the program:

```
$ perf mem record -- ./a.exe
$ perf mem -t load report --sort=mem --stdio
# Samples: 656 of event 'cpu/mem-loads,ldlat=30/P'
# Total weight : 136578
# Overhead      Samples  Memory access
# .....
  44.23%         267  LFB or LFB hit
  18.87%         111  L3 or L3 hit
  15.19%          78  Local RAM or RAM hit
  13.38%          77  L2 or L2 hit
```

<sup>91</sup>[https://en.wikipedia.org/wiki/False\\_sharing](https://en.wikipedia.org/wiki/False_sharing)

8.34%	123	L1 or L1 hit
-------	-----	--------------

From this output we can see that 8% of the loads in the application were satisfied from L1 cache, 15% from DRAM and so on.

## 6.4 Intel Processor Traces

Intel Processor Traces (PT) is a CPU feature which records the program execution by encoding packets in a highly compressed binary format that can be used to reconstruct execution flow with timestamp on every instruction. PT has extensive coverage and relatively small ~5% overhead<sup>92</sup>. Its main usages are postmortem analysis and root causing performance glitches.

### 6.4.1 Workflow

Similar to sampling techniques PT do 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 are enabled and the benchmark launches, analysis tool starts writing tracing packets to DRAM.

Similar to LBR, Intel PT works by recording branches. At runtime, whenever CPU encounters any branch instruction, PT will record the outcome of this branch. For simple conditional jump instruction CPU will record whether it was taken (T) or not taken (NT) using just 1 bit. For indirect call PT will record the destination address. Note that unconditional branches are ignored, since we statically know their target.

Example of encoding for a small instruction sequence is shown on Figure 23. Instructions like PUSH, MOV, ADD and CMP are ignored, because they don't change the control flow. However, JE instruction may jump to `.label`, so its result needs to be recorder. Later there is an indirect call for which destination address is saved.

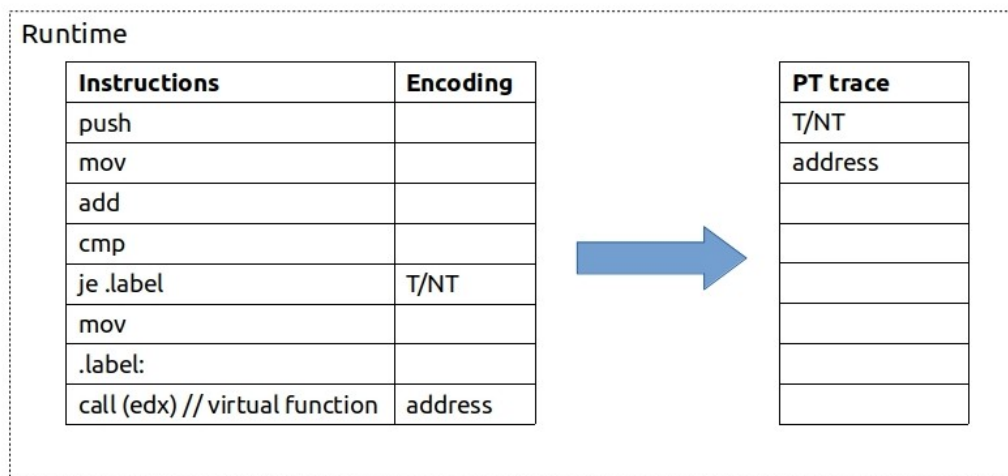


Figure 23: Intel Processor Traces encoding

At the time of analysis, we bring together the application binary and collected PT trace. SW decoder needs the application binary file, in order to reconstruct the execution flow of the

<sup>92</sup>Overhead for compute-bound application might come from lots of branches -> more data to log. Overhead for memory-bound application might come from the fact that PT pushes a lot of data to DRAM.

program. It starts from the entry point and then uses collected traces as a lookup reference to determine the control flow. Figure 24 shows example of decoding Intel Processor Traces. Suppose that 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 SW decoder encounters JE instruction which is a conditional branch and for which we need to look up the outcome. According to PT trace, JE was taken (T), so we skip next MOV instruction and go to CALL instruction. Again, CALL(edx) is an instruction that changes the control flow, so we lookup destination address in encoded traces, which is 0x407e1d8. Instructions highlighted in yellow were executed when our program was running. Note, that this is *exact* reconstruction of program execution, we did not skip any instruction. Later we can map assembly instructions back to the source code by using debug information and have a log of source code that was executed line by line.

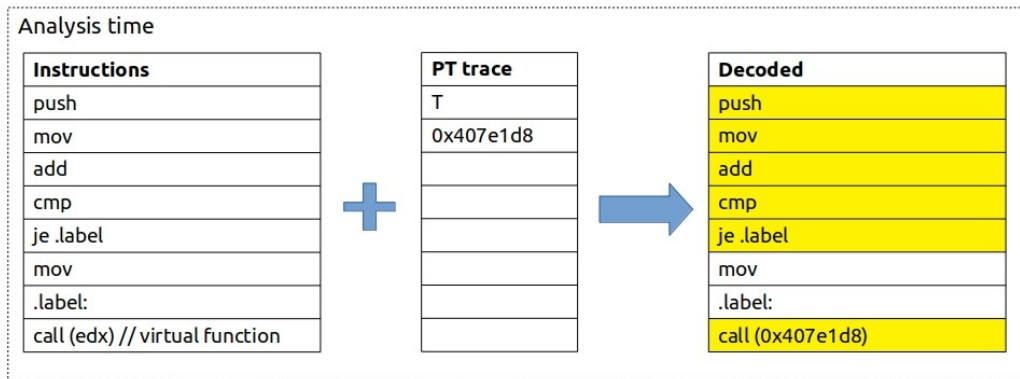


Figure 24: Intel Processor Traces decoding

### 6.4.2 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 25 provides visualization of how time packets can be used to restore timestamps for instructions. As in the previous example, we first see that JNZ was not taken, so we update it and all the instructions above with timestamp 0ns. Then we see a timing update of 2ns and JE being taken, so we update it and all the instructions above JE (and below JNZ) with timestamp 2ns. After that there is an indirect call, but no timing packet 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.

From this example we can see that **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 allows to align the time interval of our program with some other event in the system and it's easy to compare to wall clock time.

### 6.4.3 Collecting and Decoding Traces

Intel PT traces can be easily collected with Linux perf tool:

```
$ perf record -e intel_pt/cyc=1/u ./a.out
```



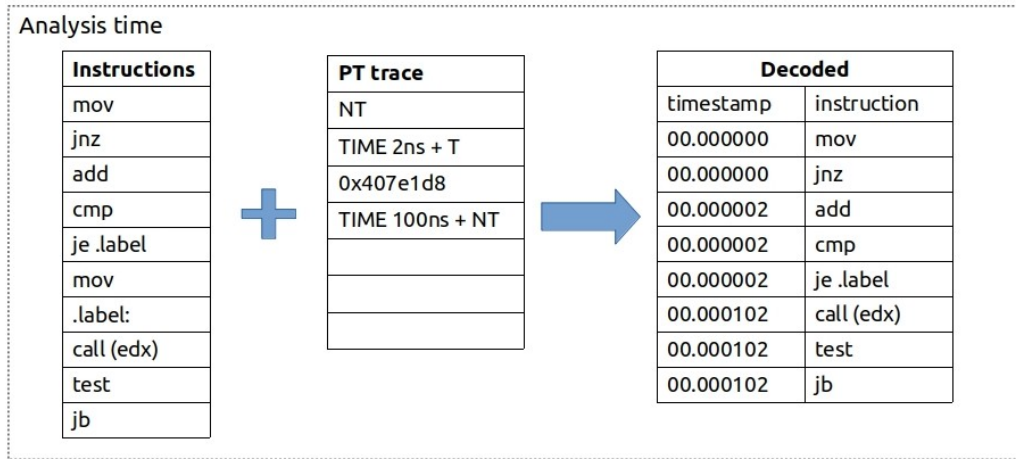


Figure 25: Intel Processor Traces timings

In the command line above we asked PT mechanism to update timing information every cycle. But likely it will not increase our accuracy greatly, since timing packets will be sent only when paired with some other control flow packet (see sec. 6.4.2).

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 Intel Skylake CPU generation, timing packets has cycle count elapsed from the previous packet. If we then look into the `trace.dump` we might see something like the following:

```
000073b3: 2d 98 8c TIP 0x8c98 // target address (IP)
000073b6: 13 CYC 0x2 // timing update
000073b7: c0 TNT TNNNNN (6) // 6 conditional branches
000073b8: 43 CYC 0x8 // 8 cycles passed
000073b9: b6 TNT NTTNTT (6)
```

Above we showed the raw PT packets which are not very useful for performance analysis. To decode processor traces to human readable form, one can execute:

```
$ perf script --ns --itrace=ilt -F time,srcline,insn,srccode
```

Below is the example of decoded traces one might get:

```
timestamp      srcline  instruction      srccode
...
253.555413143: a.cpp:24 call 0x35c      foo(arr, j);
253.555413143: b.cpp:7  test esi, esi    for (int i = 0; i <= n; i++)
253.555413508: b.cpp:7  js 0x1e
253.555413508: b.cpp:7  movsxd rsi, esi
...
```

Above is shown just a small snippet from the long execution log. In this log **we have traces of every instruction executed while out program was running**. We can literally observe



every step that was made by the program. It is a very strong foundation for further analysis. More detailed examples of traces can be found in a series of articles on [easyperf blog](#)<sup>93</sup>.

#### 6.4.4 Usages

Here are some of the cases when PT can be useful:

1. **Analyze performance glitches.** Because PT captures entire instruction stream, it is possible to analyze what was going on during the small time period when application was not responding.<sup>94</sup>
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<sup>95</sup>. PT traces could be collected on remote machine once and then analyzed offline. This is especially useful when issue is hard to reproduce or access to the system is limited.
3. **Introspect execution of the program.**
  - We can immediately tell if some code path was never executed.
  - Thanks to timestamps, it's possible to calculate how much time was spent waiting while spinning on a lock attempt, etc.
  - Security mitigation by detecting specific instruction pattern.

#### 6.4.5 Disk Space and Decoding Time

Even taking into account compressed format of the traces, encoded data can consume a lot of disk space. Typically, it's less than 1 byte per instruction, however taking into account speed at which CPU executes instructions, it is still a lot. Depending on a workload, it's very common for CPU to encode PT at a speed of 100 MB/s. Decoded traces might easily be 10 times more (~1GB/s). This makes PT not practical for using on long running workloads. But it is affordable to run it for a small period of time even on the big workload. In this case user can attach to the running process just for the period of time when the glitch happened. Or they can use a circular buffer, where new traces will overwrite old ones, i.e. always having traces for the last 10 seconds or so.

User can limit collection even further in several ways. They can limit collecting traces only on user/kernel space code. Also, there is address range filter, so it's possible to opt-in and opt-out of tracing dynamically to limit the memory bandwidth. This allows to trace just a single function or even a single loop.<sup>96</sup>

Decoding PT traces can take a long time. On Intel Core i5-8259U machine, for workload that runs for 7 milliseconds encoded PT trace takes around 1MB. Decoding this trace using `perf script` takes ~20 seconds. Decoded output from `perf script -F time,ip,sym,symoff,insn` takes ~1.3GB of disk space.

**Personal Experience:** Intel PT is supposed to be an end game for performance analysis. With its low runtime overhead, it is a very powerful analysis feature. However, right now (February 2020) decoding traces with '`perf script -F`' with '`+srcline`' or '`+srcode`' gets extremely slow and is not practical for daily usage. Implementation of Linux perf might be improved. Intel VTune

<sup>93</sup><https://easyperf.net/blog/2019/09/06/Intel-PT-part3>

<sup>94</sup><https://easyperf.net/blog/2019/09/06/Intel-PT-part3>

<sup>95</sup><https://easyperf.net/blog/2019/08/30/Intel-PT-part2>

<sup>96</sup><http://halobates.de/blog/p/410>

Profiler support for PT is still experimental.

## References and links

- Intel publication “Processor Tracing”, URL: <https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing>.
- Andi Kleen article on LWN, URL: <https://lwn.net/Articles/648154>.
- Intel PT Micro Tutorial, URL: <https://sites.google.com/site/intelptmicrotutorial/>.
- simple\_pt: Simple Intel CPU processor tracing on Linux, URL: <https://github.com/andikleen/simple-pt/>.
- Intel PT documentation in the Linux kernel, URL: <https://github.com/torvalds/linux/blob/master/tools/perf/Documentation/intel-pt.txt>.
- Cheat sheet for Intel Processor Trace, URL: <http://halobates.de/blog/p/410>.

## 6.5 Chapter Summary

- Utilizing HW features for low-level tuning is recommended only once all high-level performance issues are fixed. Tuning poorly designed algorithms is a bad investment of developer’s time. Once all the major performance problems eliminated, one can use CPU performance monitoring features to analyze and further tune their application.
- Intel Top-Down Microarchitecture Analysis Methodology (TMAM) is a very powerful technique for identifying ineffective usage of CPU microarchitecture by the program. It is a robust and formal methodology which is easy to use even for unexperienced developers. TMAM is an iterative process that consists of multiple steps including characterizing the workload and locating exact place in the source code where the bottleneck occurs. We advise that TMAM should be a starting point of analysis for every low-level tuning effort.
- Intel Last Branch Record (LBR) mechanism continuously logs most recent branch outcomes in parallel with executing the program, causing minimal slowdown. It allows to have deep enough call stack for every sample we collect while profiling. Also, LBR helps identify hot branches, misprediction rates and allows for precise timing of machine code.
- Intel Processor Event-Based Sampling (PEBS) feature is another enhancement for profiling. It lowers the sampling overhead by automatically sampling multiple times to a dedicated buffer without interrupts. However, PEBS are more widely known for introducing “Precise Events” which allow pinpointing exact instruction that caused particular performance event.
- Intel Processor Traces (PT) is a CPU feature which records the program execution by encoding packets in a highly compressed binary format that can be used to reconstruct execution flow with timestamp on every instruction. PT has extensive coverage and relatively small overhead. Its main usages are postmortem analysis and root causing performance glitches.

---

## 7 Source Code Tuning For CPU

In this chapter we will take a look at how to use CPU monitoring features (see sec. 6) to find the places in the code which can be tuned for execution on a CPU. For performance critical applications like large distributed cloud services, scientific HPC software, ‘AAA’ games, etc. it is very important to know how underlying HW works. It is a fail-from-the-start when the program is being developed without HW focus. This is why usually traditional algorithms and data structures don’t always work well for performance critical workloads. For example, linked list is pretty much deprecated in favor of ‘flat’ data structures. Traditionally every new node of the linked list is dynamically allocated which results in the situation when all the elements of the list are scattered in memory. Traversing such a data structure requires random memory access for every element. Even though algorithmic complexity is still  $O(N)$ , in practice the timings will be much worse than of a plain array. Some data structures, like binary trees, have natural linked-list-like representation, so it might be tempting to implement them in a pointer chasing manner. However, more efficient “flat” versions of those data structures exist, see `boost::flat_map`, `boost::flat_set`.

Even though the algorithm you choose is best known for particular problem, it might not work best for your particular case. For example, binary search is optimal for finding element in a sorted array. However, this algorithm usually suffers from branch mispredictions, since every test of the element value has 50% chance of being true. This is why on a small-sized array (<20 elements) linear search is usually better.

Performance engineering is an art. And like in any art the set of possible scenarios is endless. This chapter tries to address optimizations specific to CPU microarchitecture without trying to cover all existing optimization opportunities one can imagine. Still I think it is important to at least name some high-level ones:

- If a program is written using interpreted languages (python, javascript, etc.), rewrite its performance critical portion in a language with less overhead.
- Analyze the algorithms and data structures used in the program, see if you can find better ones.
- Tune compiler options. Check that you use at least those 3: `-O3/-Ofast`, `-march`, `-flto`.
- If a problem is a highly parallelizable computation, make it threaded, or consider running it on the GPU.
- Use async IO to avoid blocking while waiting for IO operations.
- Leverage using more RAM to reduce the amount of CPU and IO you have to use (memoization, look-up tables, caching of data, etc.)

### 7.1 Data-Driven Optimizations

One of the most important techniques for tuning is called “Data-Driven” optimization that is based on introspecting the data the program is working on. The approach is to focus on the layout of the data and how it is transformed throughout the program.<sup>97</sup> Classical example of such approach is Structure-Of-Array to Array-Of-Structures (SOA-to-AOS<sup>98</sup>) transformation which is shown on Listing 8.

The answer to the question which layout is better depends on how the code is accessing the data. If the program iterates over the data structure and only accesses field `b`, then SOA is better, because all memory accesses will be sequential (spatial locality). However, if the

---

<sup>97</sup>[https://en.wikipedia.org/wiki/Data-oriented\\_design](https://en.wikipedia.org/wiki/Data-oriented_design)

<sup>98</sup>[https://en.wikipedia.org/wiki/AoS\\_and\\_SoA](https://en.wikipedia.org/wiki/AoS_and_SoA)

**Listing 8** SOA to AOS transformation.

```

struct S {
    int a[N];
    int b[N];
    int c[N];
    // many other fields
};

<=>

struct S {
    int a;
    int b;
    int c;
    // many other fields
};
S s[N];

```

program iterates over the data structure and does excessive operations on all the fields of the same element of the structure (i.e. `a`, `b`, `c`), then AOS is better, because it's likely that all the members of the structure will reside in the same cache line in memory. It will additionally better utilize the memory bandwidth, since less cache line reads will be required.

This class of optimizations is based on knowing the data on which the program operates, how it is laid out and modifying the program accordingly.

**Personal Experience:** In fact, we can say that all optimizations are data-driven in some sense. Even the transformations that we will look at in next sections are based on some feedback we receive from the execution of the program: function call counts, profiling data, performance counters, etc.

Another very important example of data-driven optimization is “Small Size optimization”. Its idea is to preallocate some amount of memory for a container to avoid dynamic allocations. It is especially useful for small and medium sized containers when the upper limit of elements can be well-predicted. This approach was successfully deployed across whole LLVM infrastructure and provided significant performance benefits (search `SmallVector` for example). The same concept is implemented in `boost::static_vector`.

Obviously, it's not a complete list of data-driven optimizations, but as was written earlier, there was no attempt to list them all. Reader can find some more examples on [easypf blog](https://easypf.net/blog/2019/11/27/data-driven-tuning-specialize-indirect-call)<sup>99</sup>.

Modern CPU is a very complicated device and it's nearly impossible to predict how certain piece of code will perform. Instruction execution by the CPU depends on many factors and the number of moving parts is too big for a human mind to overlook. Hopefully, knowing how your code looks like from CPU perspective is possible thanks to all the performance monitoring capabilities we discussed in sec. 6.

Note, that optimization that you implement might not be beneficial for every platform. For

<sup>99</sup><https://easypf.net/blog/2019/11/27/data-driven-tuning-specialize-indirect-call> and <https://easypf.net/blog/2019/11/22/data-driven-tuning-specialize-switch>

example, [loop blocking](#)<sup>100</sup> very much depends on the characteristics of memory hierarchy in the system, especially L2 and L3 cache sizes. So, algorithm tuned for 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.

Further sections in this chapter are organized in a most convenient way to be used with TMAM (see sec. 6.1). The idea behind this classification is to offer some kind of checklist which engineers can use in order to effectively eliminate inefficiencies that TMAM reveals. Again, this is not supposed to be a complete list of transformations one can come up with, however, this is an attempt to describe the typical ones.

## 7.2 CPU Front-End Optimizations

CPU Front-End (FE) component is discussed in sec. 3.2.1. Most of the time inefficiencies in CPU FE can be described as situation when Back-End is waiting for instructions to execute, but FE is not able to provide them. As a result, CPU cycles are wasted without doing any actual useful work. Because modern processors are 4-wide (i.e. they can provide 4 uops every cycle), there can be situation when not all 4 available slots are filled. This can be a source of inefficient execution as well. In fact, `IDQ_UOPS_NOT_DELIVERED` performance event is counting how many available slots were not utilized due to a front-end stall. TMAM uses this performance counter value to calculate its “Front-End Bound” metric<sup>101</sup>.

The reasons for why FE could not deliver instructions to the execution units can be plenty. But usually it boils down to caches utilization, and inability to fetch instructions from memory. It's recommended to start looking into optimizing code for CPU FE only when TMAM points to a high “Front-End Bound” metric.

**Personal Experience:** Most of the real-world applications will experience non-zero "Front-End Bound" metric, meaning that some percentage of running time will be lost on suboptimal instruction fetching and decoding. Luckily it is usually below 10%. If you see the "Front-End Bound" metric being around 20%, it's definitely worth to spend time on it.

### 7.2.1 Machine code layout

When compiler translates your source code into machine code (binary encoding) it generates serial byte sequence. For example, for the following C code:

```
if (a <= b)
    c = 1;
```

compiler could generate assembly like this:

```
; a is in rax
; b is in rdx
; c is in rcx
cmp rax, rdx
jg .label
mov rcx, 1
.label:
```

<sup>100</sup>[https://en.wikipedia.org/wiki/Loop\\_nest\\_optimization](https://en.wikipedia.org/wiki/Loop_nest_optimization)

<sup>101</sup>[https://download.01.org/perfmon/TMA\\_Metrics.xlsx](https://download.01.org/perfmon/TMA_Metrics.xlsx).

Assembly instructions will be encoded and laid out in memory consequently:

```
400510  cmp rax, rdx
400512  jg 40051a
400514  mov rcx, 1
40051a  ...
```

This is what is called *machine code layout*. Note, that for the same program it's possible to layout the code in many different ways. For example, given 2 functions: `foo` and `bar`, we can place `bar` first in the binary and then `foo` or reverse the order. This might affect offsets at which instructions will be placed.

Next we will take a look at some typical optimizations for machine code layout.

### 7.2.2 Basic block placement

Suppose we have hot path in the program that has some error handling code (`coldFunc`) in between:

```
// hot path
if (cond)
    coldFunc();
// hot path again
```

Figure 26 shows two possible physical layouts for this snippet of code. Figure 26a is the layout most compiler will emit by default given no hints provided. Layout shown on figure 26b can be achieved if we invert the condition `cond` and place hot code as a fall through.

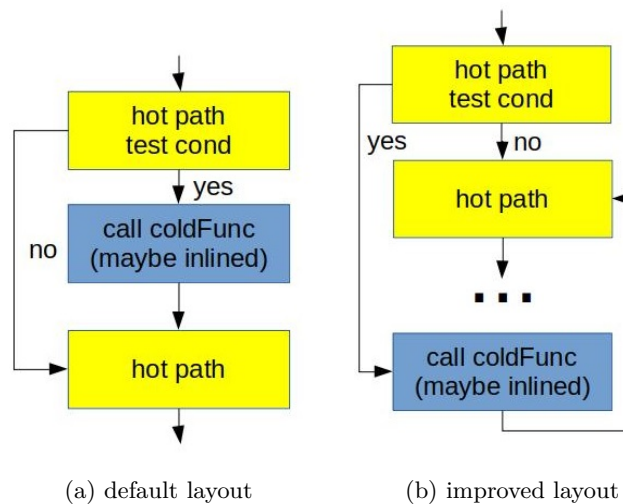


Figure 26: Two different machine code layouts.

Which layout is better in the common case, greatly depends on whether `cond` is usually true or not. If `cond` is usually true, then we would better choose default layout, because otherwise we would be doing 2 jumps instead of 1. Also, in the general case, we want to inline the function that is guarded under `cond`. However, in this particular example, we know that `coldFunc` is an error handling function and is likely not executed very often. By choosing layout 26b we maintain fall through between hot pieces of the code and convert taken branch into not taken one.

There are a few reasons why for the code presented earlier in this section layout 26b performs better. First of all, not taken branches are fundamentally cheaper than taken. In the general case, modern Intel CPUs can handle 2 untaken branches per cycle, but it can retire only 1 taken branches every 2 cycles.<sup>102</sup>

Secondly, layout 26b makes better use of the instruction and uop-cache (DSB). With all hot code contiguous, there is no cache line fragmentation: all the cache lines in the L1I-cache are used by hot code. The same is true for the uop-cache since it caches based on the underlying code layout as well.

Finally, taken branches are also more expensive for the fetch unit. It fetches contiguous chunks of 16 bytes, so every taken jump means the bytes after the jump are useless. This reduces the maximum effective fetch throughput.

In order to suggest the compiler to generate improved version of machine code layout, one can provide a hint using `__builtin_expect`<sup>103</sup> construct:

```
// hot path
if (__builtin_expect(cond, 0)) // NOT likely to be taken
    coldFunc();
// hot path again
```

Developers usually write LIKELY helper macros to make the code more readable, so more often you can find the code that looks like this:

```
#define LIKELY(EXPR)    __builtin_expect((bool)(EXPR), true)
#define UNLIKELY(EXPR) __builtin_expect((bool)(EXPR), false)

if (LIKELY(ptr != nullptr))
    // do something with ptr
```

Optimizing compilers will not only improve code layout when they encounter LIKELY/UNLIKELY hints. They will also leverage this information in other places. For example, when UNLIKELY hint is applied to our original example in this section, compiler will prevent inlining `coldFunc` as 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 `__builtin_expect` hint is also possible for a switch statement as presented in Listing 9.

---

#### Listing 9 Built-in expect hint for switch statement

---

```
for (;;) {
    switch (__builtin_expect(instruction, ADD)) {
        // handle different instructions
    }
}
```

---

Using this hint compiler will be able to reorder code a little bit differently and optimize the hot switch for faster processing `ADD` instructions. More details about this transformation is available on [easyperf](https://easyperf.net/blog/2019/11/22/data-driven-tuning-specialize-switch)<sup>104</sup> blog.

<sup>102</sup>There is a special small loop optimization that allows very small loops to have 1 taken branch per cycle.

<sup>103</sup>More about builtin-expect here: <https://llvm.org/docs/BranchWeightMetadata.html#builtin-expect>.

<sup>104</sup><https://easyperf.net/blog/2019/11/22/data-driven-tuning-specialize-switch>



### 7.2.3 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 10.

**Listing 10** Basic block alignment

```
void benchmark_func(int* a) {
    for (int i = 0; i < 32; ++i)
        a[i] += 1;
}
```

Decent optimizing compiler might come up with machine code for Skylake architecture that may look like below:

```
00000000004046a0 <_Z14benchmark_funcPi>:
4046a0: mov     rax,0xfffffffffffffff80
4046a7: vpcmpeqd ymm0,ymm0,ymm0
4046ab: nop     DWORD PTR [rax+rax*1+0x0]
4046b0: vmovdqu ymm1,YMMWORD PTR [rdi+rax*1+0x80] # loop begins
4046b9: vpsubd  ymm1,ymm1,ymm0
4046bd: vmovdqu YMMWORD PTR [rdi+rax*1+0x80],ymm1
4046c6: add     rax,0x20
4046ca: jne     4046b0 # loop ends
4046cc: vzeroupper
4046cf: ret
```

The code itself is pretty reasonable<sup>105</sup> for Skylake architecture, but its layout is not perfect (see Figure 27a). Instructions that correspond to the loop are highlighted in yellow. As well as for data caches, instruction cache lines are usually 64 bytes long. On Figure 27 thick boxes denote cache line borders. Notice that the loop spans multiple cache lines: it begins on the cache line 80-BF and ends in the cache-line C0-FF. Those kinds of situations are usually causing performance problems for the CPU Front-End, especially for the small loops like presented above.

To fix this, we can shift the loop instructions forward by 16 bytes using NOPs, so that the whole loop will reside in one cache line. Figure 27b shows the effect of doing this with NOP instructions highlighted in blue. Note that since the benchmark runs nothing but this hot loop, it is pretty much guaranteed that both cache lines will remain in L1I-cache. The reason for better performance of the layout 27b is not trivial to explain and will involve a fair amount of microarchitectural details, which we will avoid in this book. Interested readers can find more information in [Bakhvalov, 2018].

Even though CPU architects try hard to hide such kind of bottlenecks in their designs, still there are cases when code placement (alignment) can make a difference in performance.

By default, LLVM compiler recognizes loops and aligns them at 16B boundaries as we saw on Figure 27a. To reach the desired code placement for our example as shown on Figure 27b one should use option `-mllvm -align-all-blocks=5`. For other available options to control code placement one can take a look at [Bakhvalov, 2018]. However, be careful with using this option, as it can easily degrade performance. Inserting NOPs that will be executed can add

<sup>105</sup>Loop unrolling is disabled for illustrating the idea of the section



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80																
90																
a0	mov	mov	mov	mov	mov	mov	mov	vpcmp	vpcmp	vpcmp	vpcmp	nop	nop	nop	nop	nop
b0	vmov	vmov	vmov	vmov	vmov	vmov	vmov	vmov	vmov	vpsub	vpsub	vpsub	vpsub	vmov	vmov	vmov
c0	vmov	vmov	vmov	vmov	vmov	vmov	add	add	add	add	jne	jne	vzero	vzero	vzero	ret
d0																
e0																
f0																

(a) default layout

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80																
90																
a0	mov	mov	mov	mov	mov	mov	mov	vpcmp	vpcmp	vpcmp	vpcmp	nop	nop	nop	nop	nop
b0	nop	nop	nop	nop	nop	nop	nop	nop	nop	nop	nop	nop	nop	nop	nop	nop
c0	vmov	vmov	vmov	vmov	vmov	vmov	vmov	vmov	vmov	vpsub	vpsub	vpsub	vpsub	vmov	vmov	vmov
d0	vmov	vmov	vmov	vmov	vmov	vmov	add	add	add	add	jne	jne	nop	nop	nop	nop
e0	vzero	vzero	vzero	ret												
f0																

(b) improved layout

Figure 27: Two different alignment for the loop.

overhead to the program, especially if they stand on a critical path. NOPs do not require execution; however, they still require to be fetched from memory, decoded and retired. The latter additionally consumes space in FE data structures and buffers for bookkeeping, similar to all other instructions.

In order to have fine grained control over alignment it is also possible to use `ALIGN`<sup>106</sup> assembler directives. For experimental purposes, developer can emit assembly listing and then insert `ALIGN` directive:

```
; will place the .loop at the beginning of 256 bytes boundary
ALIGN 256
.loop
    dec rdi
    jnz rdi
```

### 7.2.4 Function splitting

The idea behind function splitting<sup>107</sup> is to separate hot code from the cold. This optimization is beneficial for relatively big functions with complex CFG and big pieces of cold code inside hot path. Example of code when such transformation might be profitable is shown on Listing 11. To remove cold basic blocks from the hot path we could cut and put them into its own new function and create a call to it (see Listing 12).

Figure 28 gives graphical representation of this transformation. Because we left just the `CALL` instruction inside the hot path it's likely that the next hot instruction will reside in the same

<sup>106</sup>[https://docs.oracle.com/cd/E26502\\_01/html/E28388/eoiyg.html](https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html). This example uses MASM. Otherwise you will see `.align` directive.

<sup>107</sup>Such transformation is also often called “outlining”. Readers can find LLVM implementation of this functionality in `lib/Transforms/IPO/HotColdSplitting.cpp`.

**Listing 11** Function splitting: baseline version.

---

```

void foo(bool cond1, bool cond2) {
    // hot path
    if (cond1) {
        // large amount of cold code (1)
    }
    // hot path
    if (cond2) {
        // large amount of cold code (2)
    }
}

```

---

**Listing 12** Function splitting: cold code outlined.

---

```

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) }

```

---

cache line. This improves utilization of CPU Front-End data structures like I-cache and DSB.

This transformation contains another important idea: disable inlining of cold functions. Even if we create a new function for the cold code, compiler may decide to inline it which will effectively undo our transformation. This is why we want to use `noinline` function attribute to prevent inlining. Additionally, we could use `LIKELY` macro (see sec. 7.2.2) to further improve layout of the hot path.

Finally, new functions should be created outside of `.text` segment, for example in `.text.cold`. This may improve memory footprint if the function is never called, since it won't be loaded into memory in the runtime.

### 7.2.5 Function grouping

Following the principles described in previous sections, hot functions can be grouped together to further improve utilization of caches in the CPU Front-End. When hot functions are grouped together, they might share the same cache line which reduces the number of cache lines CPU needs to fetch.

Figure 29 gives graphical representation of grouping `foo`, `bar` and `zoo`. Default layout (see fig. 29a) requires 3 cache line reads while in the improved version (see fig. 29b) code of `foo`, `bar` and `zoo` fits in only 3 cache lines. Additionally, when we call `zoo` from `foo`, beginning of `zoo` is already in the I-cache, since we fetched that cache line already.

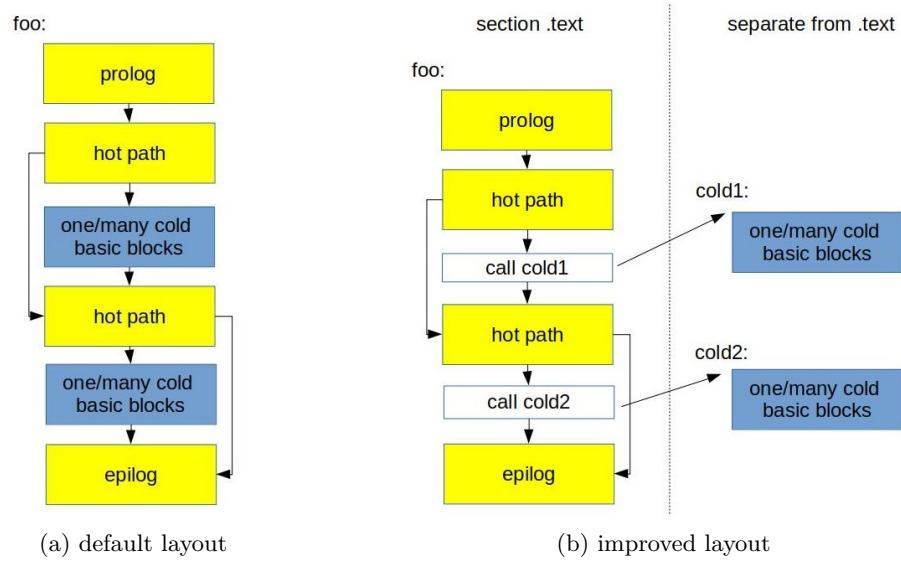


Figure 28: Splitting cold code into a separate function.

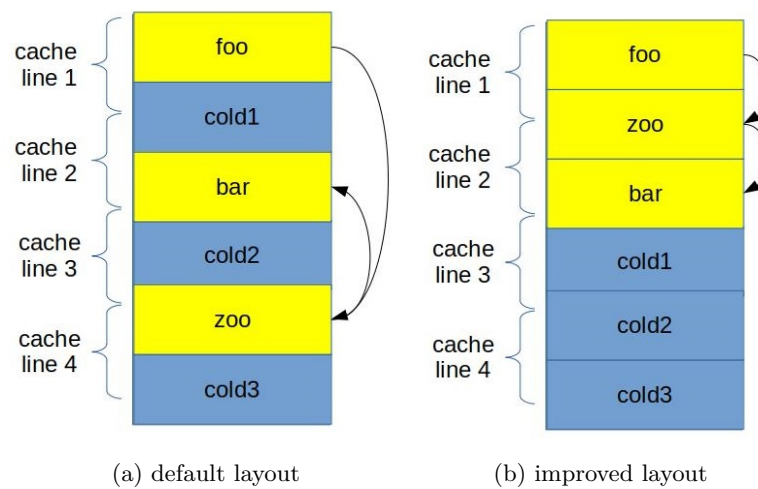


Figure 29: Grouping hot functions together.

Similar to previous optimizations, function grouping improves utilization of I-cache and DSB-cache. This optimization works best when there are many small hot functions.

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 the program themselves, there is no guarantee on the desired physical layout. For decades people have been using linker scripts to customize its behavior. Still this is the way to go if you are using GNU linker. The Gold linker (`ld.gold`) has easier approach to this problem. To get desired ordering of functions in the binary with Gold linker, one can first compile the code with `-ffunction-sections` flag, which will put each function into separate section. Then `--section-ordering-file=order.txt` option should be used to provide a file with a sorted list of function names that reflects desired final layout.

**HFSort** is a tool that generates the section ordering file automatically based on profiling data. Right now, HFSort is integrated into Facebook's HHVM project and is not available as a standalone tool. Using this tool Facebook engineers got 2% performance speedup of large distributed cloud applications like Facebook, Baidu and Wikipedia.[[Ottoni and Maher, 2017](#)]

### 7.2.6 Profile Guided Optimizations

Compiling a program and generating optimal assembly listing is all about heuristics. Code transformation algorithms have many corner cases that aim for the optimal performance in specific situations. For a lot of decisions 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, compiler could take into account the number of times this function will be called. The problem is that compiler doesn't know that beforehand.

Here is when profiling information becomes handy. Given profiling information compiler can make better optimization decisions. There is a set of transformations in most of compilers that can adjust their algorithms based on profiling data fed back to them. This set of transformation is called Profile Guided Optimizations (PGO). Often times compiler will rely on profiling data when it is available. Otherwise, it will fall back to using its standard algorithms and heuristics.

It is not uncommon to see real workloads performance increase by up to 15% from using Profile Guided Optimizations. PGO does not only improve inlining and code placement, but also improves register allocation<sup>108</sup> and more.

Profiling data can be generated based on two different ways: code instrumentation (see sec. 5.1) and sample-based profiling (see sec. 5.4). Both are relatively easy to use and have associated benefits and drawbacks discussed in sec. 5.7.

First method can be utilized in LLVM compiler by building the program with `-fprofile-instr-generate` option. This will instruct compiler to instrument the code which will collect profiling information at runtime. After that LLVM compiler can consume profiling data with `-fprofile-instr-use` option to recompile the program and output PGO-tuned binary. The guide for using PGO in clang is described in the [documentation](#)<sup>109</sup>. GCC compiler uses different set of options `-fprofile-generate` and `-fprofile-use` as described in GCC [documentation](#).

Second method, which is generating profiling data input for the compiler based on sampling, can be utilized thanks to [AutoFDO](#)<sup>110</sup> tool which converts sampling data generated by Linux

<sup>108</sup>because with PGO compiler can put all the hot variables into registers, etc.

<sup>109</sup><https://clang.llvm.org/docs/UsersManual.html#profiling-with-instrumentation>

<sup>110</sup><https://github.com/google/autofdo>

`perf` into format that compilers like GCC and LLVM can understand. [Chen et al., 2016]

Keep in mind that compiler “blindly” uses the profile data you provided. Compiler assumes that all the workloads will behave the same, so it optimizes your app just for that single workload. Users of PGO should be careful about choosing the workload to profile, because while improving one use case of the application, other might be pessimized. Luckily, it doesn’t have to be exactly single workload since profile data from different workloads can be merged together to represent a set of use case for the application.

In the mid 2018 Facebook open-sourced their binary relinker tool called **BOLT**. BOLT works on already compiled binary. It first disassembles the code, then it uses profile information to do various layout transformations (including basic blocks reordering, function splitting and grouping) and generates optimized binary [Panchenko et al., 2018]. Similar tool was developed at Google called **Propeller** which serves similar purpose as BOLT but claim certain advantages over it. It is possible to integrate optimizing relinker into the build system and enjoy extra 5-10% performance speedup from the optimized code layout. The only thing one need to worry about is to have representative and meaningful workload for collecting profiling information.

### 7.2.7 Optimizing for ITLB

Another important area of tuning FE efficiency is virtual-to-physical address translation of memory addresses. Primarily those translations are served by TLB (see sec. 3) which caches most recently used memory page translations in a dedicated entries. When TLB cannot serve 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. When TMAM points to a high **ITLB Overhead**<sup>111</sup>, advices in this section may become handy.

ITLB pressure can be reduced by mapping the portions of performance-critical code of an application onto large pages. This requires relinking the binary to align text segments at the proper page boundary in preparation for large page mapping (see [guide](#)<sup>112</sup> to `libhugetlbfs`). For general discussion on large pages see sec. 7.3.1.3.

Besides from employing large pages, standard techniques for optimizing I-cache performance can be used for improving ITLB performance; namely, reordering functions so that hot functions are more collocated, reducing the size of hot regions via **LTO/IPO**<sup>113</sup>, using profile-guided optimization, and less aggressive inlining.

### 7.2.8 Summary

Summary of CPU Front-End optimizations is presented in table 4.

Table 4: Summary of CPU Front-End optimizations.

Transform	How transformed?	Why helps?	Works best for	Done by
Basic block placement	maintain fall through hot code	not taken branches are cheaper; better cache utilization	any code, especially with a lot of branches	compiler

<sup>111</sup><https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/reference/cpu-metrics-reference/front-end-bound/itlb-overhead.html>

<sup>112</sup><https://github.com/libhugetlbfs/libhugetlbfs/blob/master/HOWTO>

<sup>113</sup>[https://en.wikipedia.org/wiki/Interprocedural\\_optimization](https://en.wikipedia.org/wiki/Interprocedural_optimization)

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 grouping	group hot functions together	better cache utilization	many small hot functions	linker

**Personal Experience:** I think code layout improvements are often underestimated and end up being omitted and forgotten. I agree that you might want to start with "fruits that hang lower" like loop unrolling and vectorization opportunities. But knowing that you might get extra 5-10% just from better laying out the machine code is still useful. It is usually the best option to use PGO if you can come up with a set of typical use cases for your application.

### 7.3 CPU Back-End Optimizations

CPU Back-End (BE) component is discussed in sec. 3.2.2. Most of the time inefficiencies in CPU BE can be described as situation when FE has fetched and decoded instructions, but BE is overloaded and can't handle new instructions. Technically speaking, it is a situation when FE cannot deliver uops due to a lack of required resources for accepting new uops in the Backend. Example of it may be stalls due to data-cache misses or stalls due to the divider unit being overloaded.

I want to emphasize to the reader that it's recommended to start looking into optimizing code for CPU BE only when TMAM points to a high "Back-End Bound" metric.

TMAM further divides **Backend Bound** metric into two main categories: **Memory Bound** and **Core Bound** which we will discuss next.

#### 7.3.1 Memory Bound

When an application executes big 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.

The statement that memory hierarchy performance is very important is backed by Figure 30. It shows the growth of the gap in performance between memory and processors. The vertical axis is on logarithmic scale and shows the growth of the CPU-DRAM performance gap. The memory baseline is latency of a memory access of 64 KB DRAM chip from 1980. Typical DRAM performance improvement is 7% per year, while CPUs enjoy 20-50% improvement per year. [Hennessy and Patterson, 2011]

In TMAM, **Memory Bound** estimates fraction of slots where CPU pipeline is likely stalled due to demand load or store instructions. The first step to solve such performance problem is to locate the memory accesses that contribute to the high **Memory Bound** metric (see sec. 6.1.4). Once guilty memory access identified, several optimization strategies could be applied. Below we will discuss few typical cases.

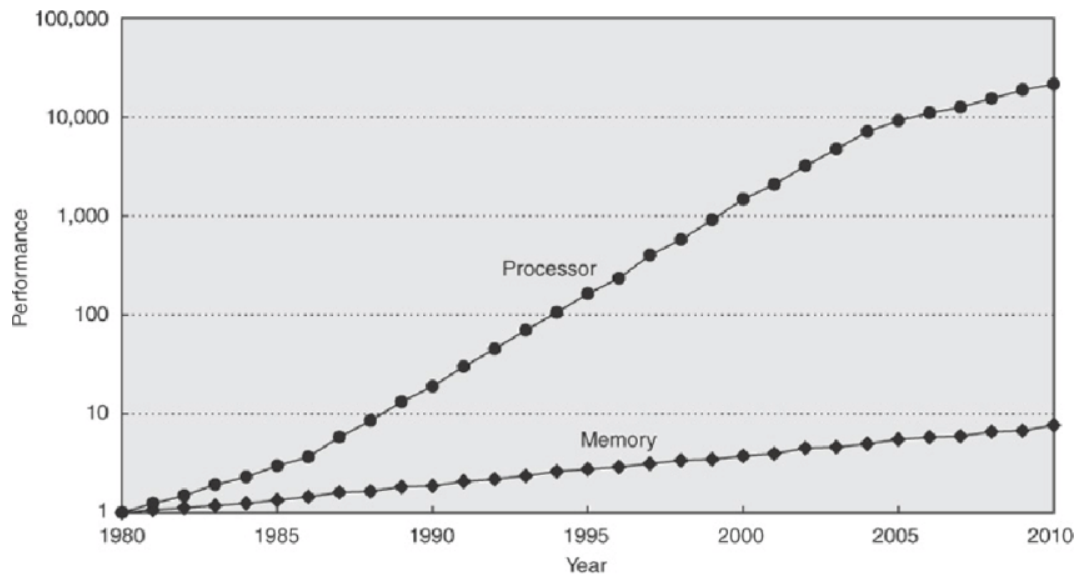


Figure 30: The gap in performance between memory and processors. [Hennessy and Patterson, 2011]

**7.3.1.1 Cache-Friendly Data Structures.** A variable can be fetched from the cache in just a few clock cycles, but it can take more than a hundred clock cycles to fetch the variable from RAM memory if it is not in the cache. There is a lot of information written on importance of writing cache-friendly algorithms and data structures as it is one of the key items in the recipe for a well-performing application. The key pillar of cache-friendly code is the principles of temporal and spatial locality (see sec. 3.2.4), which goal is to allow efficient fetching required data from caches. When designing cache-friendly code it's helpful to think in terms of cache lines, not only individual variables and their places in memory.

**7.3.1.1.1 Use appropriate containers.** There is a big variety of ready-to-use containers in almost any language. But it's important to know their underlying storage and performance implications. A good step-by-step guide of choosing appropriate C++ container can be found in [Fog, 2004, Chapter 9.7 Data structures and container classes].

Additionally, choose the data storage bearing in mind what the code will do with it. Consider situation when there is a need to choose between storing objects in the array and storing pointers to the objects, while the object size is big. Sorting array of pointers will be cheaper, since less amount of memory needs to be transferred. However, linear scan through an array of objects will be cheaper, since it is more cache friendly and does not require indirect memory accesses.<sup>114</sup>

**7.3.1.1.2 Access data sequentially.** The best way to exploit spatial locality of the caches is to make sequential memory accesses. Example of such cache-friendly access is shown on Listing 13. Swapping the order of indexes in the array (i.e. `matrix[column][row]`) will result in **column-major order**<sup>115</sup> traversal of the matrix which does not exploit spatial locality.

Example presented on Listing 13 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

<sup>114</sup><https://www.bfilipek.com/2014/05/vector-of-objects-vs-vector-of-pointers.html>

<sup>115</sup>[https://en.wikipedia.org/wiki/Row-\\_and\\_column-major\\_order](https://en.wikipedia.org/wiki/Row-_and_column-major_order)



---

**Listing 13** Cache-friendly memory accesses.

---

```
for (row = 0; row < NUMROWS; row++)
    for (column = 0; column < NUMCOLUMNS; column++)
        matrix[row][column] = row + column;
```

---

code. For instance, standard implementation of binary search in a sorted large array does not exploit spatial locality since it tests elements in different locations that are far away from each other and do not share the same cache line. The most famous way of solving this problem is storing elements of the array using Eytzinger layout [Khuong and Morin, 2015]. The idea of it is to maintain implicit binary search tree packed into an array using the BFS-like layout usually seen with binary heaps. If the code will perform a big number of binary searches in the array it may be beneficial to convert it to Eytzinger layout.

**7.3.1.1.3 Packing the data.** Memory hierarchy utilization can be improved by making the data more compact. There are many ways to pack the data, one of the classical examples is to use bitfields. Example of code when packing data might be profitable is shown on Listing 14. If we know that `a`, `b` and `c` represent enum values which take certain bit to encode, we can reduce their storage to just a handful of bits (see Listing 15).

---

**Listing 14** Packing Data: baseline struct.

---

```
struct S {
    unsigned a;
    unsigned b;
    unsigned c;
}; // S is sizeof(unsigned int) * 3
```

---



---

**Listing 15** Packing Data: packed struct.

---

```
struct S {
    unsigned a:4;
    unsigned b:2;
    unsigned c:2;
}; // S is only 1 byte
```

---

This greatly reduces the amount of memory transferred back and forth and saves cache space. Keep in mind, that this comes with a cost of accessing every packed element. Since the bits of `b` share the same machine word with `a` and `c` compiler need to perform a `>>` (shift right) and `&` (AND) operation to load it. Similarly, there `<<` (shift left) and `|` (OR) operations are needed to store the value back.

Also, a need to “pack” a data structure might arise in the following scenario. If the algorithm iterates over a collection of big objects

Packing the data is beneficial in places where additional computation is cheaper than the delay caused by inefficient memory transfers. See more examples of packed hybrid data structures in Chandler Carruth’ CppCon 2016 talk<sup>116</sup>.

<sup>116</sup><https://www.youtube.com/watch?v=vElZc6zSIXM>



**7.3.1.1.4 Aligning and padding.** Another technique to improve utilization of the memory subsystem is to align the data. There could be a situation when an object of size 16 bytes occupies 2 cache lines, i.e. it starts on one cache line and ends in the next cache line. Fetching such an object requires 2 cache line reads which could be avoided would the object be aligned properly. Listing 16 shows how memory objects can be aligned using C++11 `alignas` keyword.

---

**Listing 16** Aligning data using “alignas” keyword.

---

```
// Make aligned array
alignas(16) int16_t a[N];

// Objects of struct S are aligned at cache line boundaries
#define CACHELINE_ALIGN alignas(64)
struct CACHELINE_ALIGN S {
    //...
};
```

---

A variable is accessed most efficiently if it is stored at a memory address which is divisible by the size of the variable. For example, a double takes 8 bytes of storage space. It should therefore preferably be stored at an address divisible by 8. The size should always be a power of 2. Objects bigger than 16 bytes should be stored at an address divisible by 16. [Fog, 2004]

Alignment can cause holes of unused bytes which potentially decreases memory bandwidth utilization. If in the example above, struct S is only 40 bytes, next object of S starts at the beginning of next cache line, which leaves  $64 - 40 = 24$  unused bytes in every cache line which holds objects of struct S.

Sometimes padding data structure members is required to avoid edge cases like cache contentions [Fog, 2004, Chapter 9.10 Cache contentions] and false sharing (see sec. 8.7.3). For example, false sharing issue might occur in multithreaded applications when 2 threads A and B access different fields of the same structure. Example of code when such situation might happen is shown on Listing 17. Because a and b members of struct S could potentially occupy the same cache line, cache coherency issues might significantly slow down the program. In order to resolve the problem, one can pad S such that members a and b do not share the same cache line as shown in Listing 18.

---

**Listing 17** Padding data: baseline version.

---

```
struct S {
    int a; // written by thread A
    int b; // written by thread B
};
```

---

When it comes to dynamic allocations via `malloc`, it is guaranteed that the returned memory address satisfies the target platform’s minimum alignment requirements. Some applications might benefit from a stricter alignment. For example, dynamically allocating 16 bytes with a 64 bytes alignment instead of default 16 bytes alignment. In order to leverage this, users of POSIX systems can use `memalign`<sup>117</sup> API, other users can roll their own like described [here](#)<sup>118</sup>.

---

<sup>117</sup><https://linux.die.net/man/3/memalign>

<sup>118</sup><https://embeddedartistry.com/blog/2017/02/22/generating-aligned-memory/>

**Listing 18** Padding data: improved version.

```
#define CACHELINE_ALIGN alignas(64)
struct S {
    int a; // written by thread A
    CACHELINE_ALIGN int b; // written by thread B
};
```

One of the most important areas for alignment considerations is SIMD code. When relying on compiler auto-vectorization, developer doesn't have to do anything special. However, when you write the code using compiler vector intrinsics (see sec. 7.5.2), it's pretty common that they require addresses divisible by 16, 32, or 64. Vector types provided by the compiler intrinsic header files are already annotated to ensure the appropriate alignment. [Fog, 2004]

```
// ptr will be aligned by alignof(__m512) if using C++17
__m512 * ptr = new __m512[N];
```

**7.3.1.1.5 Dynamic memory allocation.** First of all, there are many drop-in replacements for `malloc`, which are faster, more scalable<sup>119</sup> and better address fragmentation problems. You can have few percent performance improvement just by using better memory allocation. Typical issue with dynamic memory allocation is when at startup threads race with each other trying to allocate their memory regions. One of the most popular memory allocation libraries are `jemalloc`<sup>120</sup> and `tcmalloc`<sup>121</sup>.

Second, it is possible to speed up allocations using custom allocators, for example, `arena allocators`<sup>122</sup>. One of the main advantages is their low overhead, since such allocators don't execute system calls for every memory allocation. Another advantage is its high flexibility. Developers can implement their own allocation strategies based on the memory region provided by the OS. One simple strategy could be to maintain two different allocators with their own arenas (memory regions): one for the hot data and one for the cold data. Keeping hot data together creates opportunities for it to share cache lines which improves memory bandwidth utilization and spatial locality. It also improves TLB utilization, since hot data occupies less amount of memory pages.

**7.3.1.1.6 Tune the code for memory hierarchy.** Performance of some applications depend on the size of the cache on particular level. The most famous example here is improving matrix multiplication with `loop blocking` (tiling). The idea is to break the working size of the matrix into a smaller pieces (tiles) such that each tile will fit in L2 cache<sup>123</sup>. Most of the architectures provide `CPUID`-like instruction<sup>124</sup> which allows to query the size of caches. Alternatively, one can use `cache-oblivious algorithms` which goal is to work reasonably well for any size of the cache.

Intel CPUs have Data Linear Address HW feature (see sec. 6.3.3) that supports cache blocking

<sup>119</sup>Typical `malloc` implementation involves synchronization in case multiple threads would try to dynamically allocate the memory

<sup>120</sup><http://jemalloc.net/>

<sup>121</sup><https://github.com/google/tcmalloc>

<sup>122</sup>[https://en.wikipedia.org/wiki/Region-based\\_memory\\_management](https://en.wikipedia.org/wiki/Region-based_memory_management)

<sup>123</sup>Usually people tune for the size of L2 cache since it is not shared between the cores.

<sup>124</sup>In Intel processors `CPUID` instruction is described in [Int, 2020, Volume 2]

as described on easyperf [blogpost](#)<sup>125</sup>.

**7.3.1.2 Memory Prefetching.** Often in general purpose workloads there are situations when data accesses have no clear pattern or are random, so hardware can't effectively prefetch the data ahead of time (see information about HW prefetchers in sec. 3.2.4.1). Those are cases when cache misses could not be eliminated by choosing better data structure either. Example of code when such transformation might be profitable is shown on Listing 19. If `calcNextIndex` returns values that significantly differ from what were previously returned, we might have a load `arr[j]` frequently missing in caches. When the `arr` array is big enough<sup>126</sup> HW prefetcher won't be able to catch the pattern and fail to pull the data on time. In this situation, developer can manually add explicit prefetching instructions with `__builtin_prefetch`<sup>127</sup> as shown on Listing 20.

---

**Listing 19** Memory Prefetching: baseline version.

---

```
for (int i = 0; i < N; ++i) {
    int j = calcNextIndex();
    // ...
    doSomeExtensiveComputation();
    // ...
    x = arr[j]; // this load misses in L3 cache a lot
}
```

---



---

**Listing 20** Memory Prefetching: using built-in prefetch hints.

---

```
for (int i = 0; i < N; ++i) {
    int j = calcNextIndex();
    __builtin_prefetch(a + j, 0, 1); // well before the load
    // ...
    doSomeExtensiveComputation();
    // ...
    x = arr[j];
}
```

---

For prefetch hints to make effect, be sure to insert it well ahead of time, so that by the time the loaded value will be used in other calculations, it will be already in the cache. Also, do not insert it too early, since it may pollute the cache with the data that is not used for some time. In order to estimate the prefetch window, use the method described in sec. 6.2.5. Readers can also find example of estimating prefetch window on [easyperf](#)<sup>128</sup> blog.

Prefetching data for next iteration of the loop is the most common scenario, however, linear function prefetching can also be very helpful. Software memory prefetching is not portable, meaning, that if it gives performance gains on one platform doesn't guarantee that it will give similar speedup on another platform.

---

<sup>125</sup><https://easyperf.net/blog/2019/12/17/Detecting-false-sharing-using-perf#2-tune-the-code-for-better-utilization-of-cache-hierarchy>

<sup>126</sup>For this example we define "big enough" to be more than this size of L3 cache inside typical desktop CPU, which is at the time of writing varies from 5 to 20 MB.

<sup>127</sup><https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html>

<sup>128</sup><https://easyperf.net/blog/2019/04/03/Precise-timing-of-machine-code-with-Linux-perf#application-estimating-prefetch-window>

While software prefetching gives programmer control and flexibility, it's not always easy to get it right. Consider situation when we want to insert prefetch instruction into the piece of code that has average IPC=2 and every DRAM access takes 100 cycle. To have the best effect we would need to insert prefetching instruction 200 instructions before the load. It is not always possible, especially if the load address is computed right before the load itself. Pointer chasing problem can be a good example when prefetching is helpless. [Nima Honarmand]

Finally, prefetch instruction increases code size and adds pressure on CPU Front-End. Remember, prefetch instruction is just like any other instruction, it consumes CPU resources. When using it wrong it can lead to pessimizing program performance.

**7.3.1.3 Optimizing For DTLB.** As described in sec. 3, the 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.

TLB hierarchy typically comprises from L1 ITLB (instructions), L1 DTLB (data) and L2 STLB (unified cache for instructions and data). A miss in the L1 (first level) ITLBs results in a very small penalty that can usually be hidden by the Out of Order (OOO) execution. A miss in the STLB results in the page walker being invoked. This penalty can be noticeable in the execution, because during this process, the CPU is stalled [Suresh Srinivas, 2019]. Assuming default page size in Linux kernel is 4KB, modern L1 level TLB caches can keep only up to a few hundred most recently used page-table entries which covers address space of ~1MB, while L2 STLB can hold up to a few thousands page-table entries. Exact numbers for specific processor can be found at <https://ark.intel.com>.

On Linux and Windows systems, applications are loaded into memory into 4KB pages, which is the default on most systems. Allocating many small pages is expensive. If an application actively references tens or hundreds of GBs of memory, that would require many 4KB-sized pages, each of which will contend for a limited set of TLB entries. Using large 2MB pages, 20MB of memory can be mapped with just 10 pages; whereas with 4KB pages, 5120 pages are required. This means fewer TLB entries are needed, in turn reducing the number of TLB misses. Both Windows and Linux allow applications to establish large-page memory regions. HugeTLB subsystem support depends on the architecture, while AMD64 and Intel 64 architecture support both 2 MB (huge) and 1 GB (gigantic) pages.

As we just learned, one way to reduce the number of ITLB misses is to use the larger page size. Thankfully, TLB is capable of caching entries for 2MB and 1GB pages as well. If the aforementioned application employed 2MB pages instead of the default 4KB pages, it would reduce TLB pressure by a factor of 512. Likewise, if it updated from using 2MB pages to 1GB pages, it would reduce TLB pressure by yet another factor of 512. That is quite an improvement! Using larger page size may be beneficial for some applications because less space is used in the cache for storing translations, allowing more space to be available for the application code. Huge pages typically lead 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.

Large pages can be used for code, data, or both. Large pages for data are good to try if your workload has large heap. Large memory applications such as relational database systems (e.g., MySQL, PostgreSQL, Oracle, etc.) and Java applications configured with large heap regions frequently benefit from using large pages. One example of using huge pages for optimizing runtimes is presented in [Suresh Srinivas, 2019] showing how this feature improves performance

and reduces ITLB misses (up to 50%) in three applications in three environments. However, as it is with many other features, large pages are not for every application. An application that wants to allocate only one byte of data would be better off using a 4k page rather than a huge one; that way, memory is used more efficiently.

On Linux OS, there are two ways of using large pages in an application: Explicit and Transparent Huge Pages.

**7.3.1.3.1 Explicit Hugepages.** Are available as a part of the system memory, exposed as a huge page file system (`hugetlbfs`), applications can access it using system calls, e.g. `mmap`. One can check Huge Pages appropriately configured on the system through `cat /proc/meminfo` and look at `HugePages_Total` entries. Huge pages can be reserved at boot time or at run time. Reserving at boot time increases the possibility of success because the memory has not yet been significantly fragmented. Exact instructions for reserving huge pages can be found in [Red Hat Performance Tuning Guide](#) <sup>129</sup>.

There is an option to dynamically allocate memory on top of large pages with `libhugetlbfs`<sup>130</sup> library that overrides `malloc` calls used in existing dynamically linked binary executables. It doesn't require to modify the code or even relink the binary; end users just need to configure several environment variables. It can use both explicitly reserved huge pages as well as transparent ones. See `libhugetlbfs` [how-to documentation](#)<sup>131</sup> for more details.

For more fine-grained control over accesses to large pages from the code (i.e. not affecting every memory allocation), developers have following alternatives:

- `mmap` using the `MAP_HUGETLB` flag ([example code](#)<sup>132</sup>).
- `mmap` using a file from a mounted `hugetlbfs` filesystem ([example code](#)<sup>133</sup>).
- `shmget` using the `SHM_HUGETLB` flag ([example code](#)<sup>134</sup>).

**7.3.1.3.2 Transparent Hugepages.** Linux also offers Transparent Hugepage Support (THP), which manages large pages<sup>135</sup> automatically and is transparent for applications. Under Linux, you can enable THP which dynamically switches to huge pages when large blocks of memory are needed. The THP feature has two modes of operation: system-wide and per-process. When THP is enabled system-wide, the kernel tries to assign huge pages to any process when 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 with the `madvise` system call. You can check if THP enabled in the system with:

```
$ cat /sys/kernel/mm/transparent_hugepage/enabled
always [madvise] never
```

If the values are `always` (system-wide) or `madvise` (per-process), then THP is available for your application. With `madvise` option, THP are enabled only inside memory regions attributed

<sup>129</sup>[https://access.redhat.com/documentation/en-us/red\\_hat\\_enterprise\\_linux/7/html/performance\\_tuning\\_guide/sect-red\\_hat\\_enterprise\\_linux-performance\\_tuning\\_guide-memory-configuring-huge-pages#sect-Red\\_Hat\\_Enterprise\\_Linux-Performance\\_tuning\\_guide-Memory-Configuring-huge-pages-at-run-tim](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/performance_tuning_guide/sect-red_hat_enterprise_linux-performance_tuning_guide-memory-configuring-huge-pages#sect-Red_Hat_Enterprise_Linux-Performance_tuning_guide-Memory-Configuring-huge-pages-at-run-tim)

<sup>130</sup><https://github.com/libhugetlbfs/libhugetlbfs>

<sup>131</sup><https://github.com/libhugetlbfs/libhugetlbfs/blob/master/HOWTO>

<sup>132</sup>[https://elixir.bootlin.com/linux/latest/source/tools/testing/selftests/vm/map\\_hugetlb.c](https://elixir.bootlin.com/linux/latest/source/tools/testing/selftests/vm/map_hugetlb.c)

<sup>133</sup><https://elixir.bootlin.com/linux/latest/source/tools/testing/selftests/vm/hugepage-mmap.c>

<sup>134</sup><https://elixir.bootlin.com/linux/latest/source/tools/testing/selftests/vm/hugepage-shm.c>

<sup>135</sup>Note that the THP feature only supports 2-MB pages.

with `MADV_HUGEPAGE` via `madvise` system call. Complete specification for every option can be found in Linux kernel [documentation](#)<sup>136</sup> regarding THP.

**7.3.1.3.3 Explicit vs. Transparent Hugepages.** Whilst Explicit Huge Pages (EHP) are reserved in virtual memory upfront, THPs are not. In the background, the kernel attempts to allocate a THP, and if it fails, will default to the standard 4k page. This all happens transparently to the user. The allocation process can potentially involve a number of kernel processes responsible for making space in the virtual memory for a future THP (which may include swapping memory to the disk, fragmentation or compacting pages<sup>137</sup>). Background maintenance of transparent huge pages incurs non-deterministic latency overhead from the kernel as it manages the inevitable fragmentation and swapping issues. EHP are not subject to memory fragmentation and cannot be swapped to the disk.

Secondly, EHP are available for use on all segments of an application, including text segments (i.e., benefits both DTLB *and* ITLB), while THP only available for dynamically allocated memory regions.

One advantage of THP is that less OS configuration effort required than with EHP, which enables faster experiments.

## 7.3.2 Core Bound

**7.3.2.1 Inlining Functions.** Inlining a function is one of the most important compiler optimizations. Not only because it eliminates the overhead of calling a function<sup>138</sup>, but also it enables other interprocedural optimizations. This happens because when the function is inlined, compiler widens the scope of transformation to a much larger chunk of code.

Compiler implementation usually approaches the problem of inlining with some form of cost model which generally does a good job. However, sometimes their cost model requires some hints from the source code. In general, it is a good strategy to rely on compiler on making all the inlining decisions and only tune those that can be improved.

One way to find potential candidates for inlining is by looking at how hot is [prologue and epilogue](#)<sup>139</sup> of the function. Below is an example<sup>140</sup> of a function profile with prologue and epilogue consuming 50% of the function time:

Percent		Source code & Disassembly of foo
-----		
3.77	:	418be0: push r15 <i># prologue</i>
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

<sup>136</sup><https://www.kernel.org/doc/Documentation/vm/transhuge.txt>

<sup>137</sup>E.g., compacting 4KB pages into 2MB, breaking 2MB pages back into 4KB, etc.

<sup>138</sup>Overhead of calling a function usually consists of executing `CALL`, `PUSH`, `POP` and `RET` instructions. Series of `PUSH` instructions is called “Prolog” and series of `POP` instructions is called “Epilog”.

<sup>139</sup>[https://en.wikipedia.org/wiki/Function\\_prologue](https://en.wikipedia.org/wiki/Function_prologue)

<sup>140</sup><https://easyperf.net/blog/2019/05/28/Performance-analysis-and-tuning-contest-3#inlining-functions-with-hot-prolog-and-epilog-265>



```

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

```

This might be a strong indicator that the time consumed by prologue and epilogue of the function might be saved if we inline the function.

One can make a hint to a compiler for inlining `foo` with a help of C++ attribute:

```

__attribute__((always_inline)) int foo() {
    // foo body
}

```

Note, that even if the prologue and epilogue are hot it doesn't necessary mean it will be profitable to inline the function. For example, inlining a big function is usually not profitable as it bloats the code. Inlining triggers a lot of different changes, so it's hard to predict the outcome. Always measure to confirm the need to force inlining.

### 7.3.2.2 Loop Optimizations [THIS SECTION IS NOT FINISHED]

#### Message to reviewers:

Let me know if you wish to help me write this chapter or some sections of it. If you do, let me know first so we can coordinate our efforts. Obviously, I will mention you in the contributors section. Even though I know a thing or two about loop optimization, I don't consider myself an expert in this topic.

My vision for this chapter is to answer the question what developer can do if the hotspot is inside the loop. Yes, loop unrolling is the first thing that comes to mind, but there are others, like blocking, unroll-and-jam, strip-mining, etc. This doesn't have to be theoretical, but rather practical advices and examples. We should give developers a general feeling how to find optimization opportunities in loops (of course with examples). We can give pointers to theory which can, again, take complete book just by itself. I expect this section to be ~5 pages.

### 7.3.2.3 Vectorization [THIS SECTION IS NOT FINISHED]

#### Message to reviewers:

Let me know if you wish to help me write this chapter or some sections of it. If you do, let me know first so we can coordinate our efforts. Obviously, I will mention you in the contributors section. Even though I know a thing or two about vectorization, I don't consider myself an expert in this topic.

My vision for this chapter is to answer the question how developer can use CPU SIMD capabilities to speedup the code (mostly loops, but also straight-line code). Having hotspot in the loop, how can developer make sure that the code is properly vectorized and it actually speeds up the program in comparison with scalar version. We should give developers a general feeling what to do in such situations (of course with examples). Compiler intrinsics partially covered in sec. 7.5.2. Compiler optimization reports are covered in sec. 5.6. Introduction to SIMD will be written in sec. 3.1.4. I expect this section to be ~5-7 pages.

## 7.4 Optimizing Bad Speculation

Speculation feature in modern CPUs is described in sec. 3.1.3. Mispredicting a branch can add significant speed penalty when it happens regularly. When such an event happens, 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 whole pipeline and start filling it with instructions from correct path. Typically, modern CPUs experience 15-20 cycles penalty as a result of branch misprediction.

I want to emphasize to the reader that it's recommended to start looking into optimizing branch mispredictions only when TMAM points to a high "Bad Speculation" metric.

**Personal Experience:** The program will always experience some number of branch mispredictions. It is normal for general purpose application to have "Bad Speculation" rate in the range of 5-10%. My recommendation is to pay attention at bad speculation, if the metric goes higher than 10%.

The way to get rid of branch mispredictions is to get rid of the branch itself.

### 7.4.1 Replace branches with lookup

Frequently branches can be avoided by using lookup tables. Example of code when such transformation might be profitable is shown in Listing 21. Function `mapToBucket` maps values into corresponding buckets. For uniformly distributed values of `v` we will have equal probability for `v` to fall into any of the buckets. In generated assembly for the baseline version we will likely see many branches, which could have high misprediction rates. Hopefully, it's possible to rewrite function `mapToBucket` using a single array lookup as shown in Listing 22.

---

**Listing 21** Replacing branches: baseline version.

---

```
int mapToBucket(unsigned v) {  
    if (v >= 0 && v < 10) return 0;  
    if (v >= 10 && v < 20) return 1;  
    if (v >= 20 && v < 30) return 2;  
    if (v >= 30 && v < 40) return 3;  
    if (v >= 40 && v < 50) return 4;  
    return -1;  
}
```

---

Assembly code of `mapToBucket` function from Listing 22 should be using only one branch instead of many. Typical hot path through this function will execute untaken branch and one load instruction. Since we expect most of the input values to fall into the range covered by `buckets` array, branch that guards out-of-bounds access will be well-predicted by CPU. Also,



**Listing 22** Replacing branches: lookup version.

```

int buckets[256] = {
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
    3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
    5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
    -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
    -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
    ... };

int mapToBucket(unsigned v) {
    if (v < (sizeof (buckets) / sizeof (int)))
        return buckets[v];
    return -1;
}

```

`buckets` array is relatively small, so we can expect it to reside in CPU caches, which should allow for fast accesses to it.[Lemire, 2020]

If we have a need to map bigger range of values, allocating very large array is not practical. In this case we might use interval map data structures, that accomplish that goal using much less memory, but logarithmic lookup complexity. Readers can find existing implementations of interval map container in [Boost](#)<sup>141</sup> and [LLVM](#)<sup>142</sup>.

#### 7.4.2 Replace branches with predication

Some branches could be effectively eliminated by executing both parts of the branch and then selecting the right result. Example<sup>143</sup> of code when such transformation might be profitable is shown on Listing 23. If TMAM suggest that `if (cond)` branch has a very high number of mispredictions, one can try to eliminate the branch by doing transformation shown on Listing 24.

**Listing 23** Predicating branches: baseline version.

```

int a;
if (cond) { // branch has high misprediction rate
    a = computeX();
} else {
    a = computeY();
}

```

For version of code in Listing 24, compiler can get rid of the branch and generate `CMOV` instruction instead. The `CMOVcc` instructions check the state of one or more of the status

<sup>141</sup>[https://www.boost.org/doc/libs/1\\_65\\_0/libs/icl/doc/html/boost/icl/interval\\_map.html](https://www.boost.org/doc/libs/1_65_0/libs/icl/doc/html/boost/icl/interval_map.html)

<sup>142</sup>[https://llvm.org/doxygen/IntervalMap\\_8h\\_source.html](https://llvm.org/doxygen/IntervalMap_8h_source.html)

<sup>143</sup><https://easyperf.net/blog/2019/04/10/Performance-analysis-and-tuning-contest-2#fighting-branch-mispredictions-9>

**Listing 24** Predicating branches: improved version.

```

int x = computeX();
int y = computeY();
int a = cond ? x : y;

```

flags in the EFLAGS register (CF, OF, PF, SF and ZF) and perform a move operation if the flags are in a specified state (or condition). [Int, 2020, Volume 2] Below are the two assembly listings for the baseline and improved version respectively:

```

# baseline version
400504:  test    edi,edi
400506:  je      400514      # branch on cond
400508:  mov     eax,0x0
40050d:  call    <computeX>
400512:  jmp     40051e
400514:  mov     eax,0x0
400519:  call    <computeY>
40051e:  mov     edi,eax

=>

# improved version
400537:  mov     eax,0x0
40053c:  call    <computeX>  # compute x
400541:  mov     ebp,eax     # assign x to a
400543:  mov     eax,0x0
400548:  call    <computeY>  # compute y
40054d:  test    ebx,ebx     # test cond
40054f:  cmovne  eax,ebp     # override a with y if needed

```

Modified assembly sequence doesn't have original branch instruction. However, in the second version both `x` and `y` are calculated independently and then only one of the values is selected. While this transformation eliminates penalty of branch mispredictions, it is potentially doing more work than the original code. Performance improvement in this case very much depends on the characteristics of `computeX` and `computeY` functions. If the functions are small and compiler is able to inline them, then it might bring noticeable performance benefits. If the functions are big, it might be cheaper to take the cost of branch mispredict, than to execute both functions.

Similar transformation can be done for floating-point numbers with `FCMOVcc`, `VMAXSS`/`VMINSS` instruction. In some situations, it is also possible to use compiler intrinsics to eliminate branches as shown in [Kapoor, 2009]. Generally, compilers are very smart at eliminating the branches especially when they can see the whole program. So, it is recommended to start the work of eliminating branch mispredictions only based on TMAM reports.

## 7.5 Other Tuning Areas

In this chapter we will take a look at some of optimization topics not specifically related to any of the categories covered in previous 3 sections of this chapter, still important enough to find their place in this book.

### 7.5.1 Compile-Time computations

If a portion of a program does some calculations that doesn't depend on the input, it can be precomputed ahead of time instead of doing it in the runtime. Modern optimizing compilers already move a lot of computations in compile-time, especially trivial cases like `int x = 2 * 10` into `int x = 20`. Although, they cannot handle more complicated calculations at compile time if they involve branches, loops, function calls. C++ language provides features that allow to make sure that certain calculations happen at compile time.

In C++ it's possible to move computations into compile-time with various metaprogramming techniques. Before C++11/14 developers were using templates to achieve this result. It is theoretically possible to express almost any algorithm with template metaprogramming; however, this method tends to be syntactically obtuse, and often compile quite slowly. Still it was a success that enabled new class of optimizations. Fortunately, metaprogramming gradually becomes a lot simpler with every new C++ standard. The C++14 standard allows having `constexpr` functions, and the C++17 standard provides compile-time branches with the `if constexpr` keyword. This new way of metaprogramming allows doing many computations in compile-time without sacrificing code readability. [Fog, 2004, Chapter 15 Metaprogramming]

Example of optimizing an application by moving computations into compile-time is shown in Listing 25. Suppose, a program involves a test for a number being prime. If we know that a large portion of tested numbers are less than 1024, we can precompute the results ahead of time and keep them in a `constexpr` array `primes`. At runtime, most of the calls of `isPrime` will involve just one load from `primes` array, which is much cheaper than computing it at runtime.

### 7.5.2 Compiler intrinsics

There are types of applications that have very few hotspots which calls for tune them heavily. However, compilers do not always do what we want in terms of generated code. For example, a program does some computation in a loop which compiler vectorizes in a suboptimal way. It usually involves some tricky or specialized algorithm, for which we can come up with a better sequence of instructions. It can be very hard or even impossible to make compiler generate desired assembly code using standard constructs of the C and C++ languages.

Hopefully, it's possible to force generating particular assembly instructions without writing in low-level assembly language. For that one can use compiler intrinsics which in turn are translated into specific assembly instructions. Intrinsics provide the same benefit as using inline assembly, but also, they improve code readability, allow compiler type checking, assist instruction scheduling, and help reduce debugging. Example in Listing 26 shows how the same loop in function `foo` can be coded via compiler intrinsics (function `bar`).

Both functions in Listing 26 generate similar assembly instructions, however there are several caveats. First, when relying on auto-vectorization, compiler will insert all necessary runtime checks. For instance, it will ensure that there are enough elements to feed the vector execution units. Secondly, function `foo` will have fallback scalar version of the loop for processing remainder of the loop. And finally, most vector intrinsics assume aligned data, so `movaps` (aligned load) is generated for `bar`, while `movups` (unaligned load) is generated for `foo`. Keeping that in mind, developers using compiler intrinsics have to take care about safety aspects themselves.

When writing code using non-portable platform-specific intrinsics, developers should also provide a fallback option for other architectures. List of all available intrinsics for Intel platform

**Listing 25** Precomputing prime numbers in compile-time

```
constexpr unsigned N = 1024;

// function pre-calculates first N primes in compile-time
constexpr std::array<bool, N> sieve() {
    std::array<bool, N> Nprimes{true};
    Nprimes[0] = Nprimes[1] = false;
    for(long i = 2; i < N; i++)
        Nprimes[i] = true;
    for(long i = 2; i < N; i++) {
        if (Nprimes[i])
            for(long k = i + i; k < N; k += i)
                Nprimes[k] = false;
    }
    return Nprimes;
}

constexpr std::array<bool, N> primes = sieve();

bool isPrime(unsigned value) {
    // primes is accessible both in compile-time and runtime
    static_assert(primes[97], "");
    static_assert(!primes[98], "");
    if (value < N)
        return primes[value];
    // fall back to computing in runtime
}
```

can be found in this [reference](#)<sup>144</sup>.

### 7.5.3 Cache Warming

Instruction and data caches, and the performance impact of each, were explained in sec. 7.2 and sec. 7.3.1.1, along with specific techniques to get the most benefit from each. However, in some application workloads, the portions of code that is most latency-sensitive are the least frequently executed. This results in the function blocks and associated data from aging out of the I-cache and D-cache after some time. Then, just when we need that critical piece of rarely executed code to execute, we take I-cache and D-cache miss penalties which may exceed our target performance budget.

An example of such a workload might be a high frequency trading application which continuously reads market data signals from the stock exchange and, once a favorable market signal is detected, sends a BUY order to the exchange. In the aforementioned workload, the code paths involved with reading the market data is most commonly executed, while the code paths for executing a BUY order is rarely executed. If we want our BUY order to reach the exchange as fast as possible and to take advantage of the favorable signal detected in the market data, then the last thing we want is to incur cache misses right at the moment we decide to execute that critical piece of code. This is where the technique of Cache Warming would be helpful.

<sup>144</sup><https://software.intel.com/sites/landingpage/IntrinsicsGuide/>

**Listing 26** Compiler Intrinsics

```

1 void foo(float *a, float *b, float *c, unsigned N) {
2     for (unsigned i = 0; i < N; i++)
3         c[i] = a[i] + b[i];
4 }
5
6 #include <xmmintrin.h>
7
8 void bar(float *a, float *b, float *c, unsigned N) {
9     __m128 rA, rB, rC;
10    for (int i = 0; i < N; i += 4){
11        rA = _mm_load_ps(&a[i]);
12        rB = _mm_load_ps(&b[i]);
13        rC = _mm_add_ps(rA,rB);
14        _mm_store_ps(&c[i], rC);
15    }
16 }

```

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 will also “warm-up” the D-cache by bringing latency-sensitive data in it. In fact, this technique is routinely employed for trading applications like the one described in [CppCon 2018 lightning talk](#)<sup>145</sup>.

#### 7.5.4 Detecting slow FP arithmetic

For applications, working with floating point values, there is some probability of hitting exceptional scenario when the FP values become [denormalized](#)<sup>146</sup>. Operations on denormal values could be easily 10 times slower or more. When CPU handles instruction that tries to do arithmetic operation on denormal FP values, it requires special treatment for such case. Since it is exceptional situation, CPU requests a microcode [assist](#)<sup>147</sup>. Microcode Sequencer (MSROM) then will feed the pipeline with lots of uops (see sec. 4.4) for handling such scenario.

TMAM methodology classifies such bottlenecks under **Retiring** category. This is one of the situations when high Retiring doesn’t mean a good thing. Since operations on denormal values likely represent unwanted behavior of the program, one can just collect `FP_ASSIST.ANY` performance counter. The value should be close to zero. Example of a program that does denormal FP arithmetics and thus experience many FP assists is presented on easyperf [blog](#)<sup>148</sup>. C++ developers can prevent their application fall into operations with subnormal values by using `std::isnormal()`<sup>149</sup> function.

#### 7.5.5 System Tuning

After successfully completing all the hard work of tuning an application to exploit all the intricate facilities of the CPU microarchitecture, the last thing we want is for the system

<sup>145</sup><https://www.youtube.com/watch?v=XzRxikGgaHI>

<sup>146</sup>[https://en.wikipedia.org/wiki/Denormal\\_number](https://en.wikipedia.org/wiki/Denormal_number)

<sup>147</sup><https://software.intel.com/en-us/vtune-help-assists>

<sup>148</sup><https://easyperf.net/blog/2018/11/08/Using-denormal-floats-is-slow-how-to-detect-it>

<sup>149</sup><https://en.cppreference.com/w/cpp/numeric/math/isnormal>

firmware, the OS, or the kernel to destroy all our efforts. The most highly tuned application will mean very little if it is intermittently disrupted by a System Management Interrupt (SMI), a BIOS interrupt that halts the entire OS in order to execute firmware code. Such interrupt might run for up to 10s to 100s of milliseconds at a time.

Fair to say, developers usually have little to no control over the environment in which the application is executed. When we ship the product, it's unrealistic to tune every setup customer might have. Usually, large-enough organizations have separate Operations (Ops) Teams, which handles such sort of issues. Nevertheless, when communicating with members of such teams, it's important to understand what else can limit the application to show its best performance.

As shown in sec. 2.1, there are many things to tune in the modern system and avoiding system-based interference is not an easy task. Example of performance tuning manual of x86-based server deployments is Red Hat [guidelines](#)<sup>150</sup>. 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.

When it comes to tuning specific system setting, it is not always easy 'yes' or 'no' answer. For example, it's not clear upfront whether your application will benefit from Simultaneous Multi-Threading (SMT) feature enabled in the environment in which your SW is running. The general guideline is to enable SMT only for heterogenous workloads<sup>151</sup> that exhibit a relatively low IPC. On the other hand, CPU manufacturers these days offer processors with such high core counts that SMT is far less necessary than it was in the past. However, this is just a general guideline, and as with everything stressed so far in this book, it is best to measure for yourself.

## 7.6 Chapter Summary

- This chapter is supposed to serve as a checklist for developers actively using TMAM methodology. The chapter section organization adheres to TMAM metrics structure. We recommend using particular section of this chapter once TMAM points to corresponding type of bottleneck.
- CPU Front-End (FE) bottlenecks can occur when Back-End is waiting for instructions to execute, but FE is not able to provide them (fetch, decode, etc.). The main rule of thumb for CPU Front-End efficiency is to group the hot code together and keep it away from the cold code. Attractive FE gains can be achieved with Profile Guided Optimizations (PGO), with little efforts.
- Large portion of all applications in the world are bound by CPU Back-End (BE). This category includes two subcategories: Memory Bound and Compute Bound. Since the gap in performance between memory and processors is growing every year, it becomes very important to design data structures and algorithms keeping in mind the memory hierarchy. Most of program time is spent in loops, so it's possible to get bit performance gains from loop optimizations and vectorization.
- When CPU mispredicts a branch outcome, this can add significant slowdown when it happens regularly. Typically, penalty for such occasion is between 10 and 20 clock

<sup>150</sup><https://access.redhat.com/sites/default/files/attachments/201501-perf-brief-low-latency-tuning-rhel7-v2.1.pdf>

<sup>151</sup>I.e., when sibling threads execute differing instruction patterns

cycles. The way to get rid of branch mispredictions is to get rid of the branch itself. We presented 2 ways of eliminating branch instructions by doing speculative work and lookup tables.

---

## 8 Optimizing multithreaded applications

Modern CPUs are getting more and more cores each year. As of 2020 you can buy x86 server processor which will have more than 50 cores! And even mid-range desktop with 8 execution threads is pretty usual setup. Since there is so much processing power in every CPU, the challenge is how to utilize all the HW threads efficiently. Preparing software to scale well with growing amount of CPU cores is very important for future success of the application.

Multithreaded applications have their own specifics. Certain assumptions of single-threaded execution get invalidated when we're dealing with multiple threads. For example, we can no longer identify hotspots by looking at the single thread, since each thread might have its own hotspot. In a popular [producer-consumer](#)<sup>152</sup> design, producer thread may sleep during most of the time. Profiling such a thread won't shed light on the reason why our multithreaded application is not scaling well.

### 8.1 Performance scaling and overhead.

When dealing with single-threaded application, optimizing one portion of the program usually yields positive results on performance. However, it's not necessary the case for multithreaded applications. There could be an application in which thread A does some very heavy 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 longer-running thread A.

This effect is widely known as [Amdahl's law](#)<sup>153</sup> which constitutes that the speedup of a parallel program is limited by its serial part. Figure 31 illustrates the theoretical speedup limit as a function of the number of processors. For a program, 75% of which is parallel, speedup factor converges to 4.

Figure 32a shows performance scaling of `h264dec` benchmark from [Starbench parallel benchmark suite](#). I tested it on Intel Core i5-8259U, which has 4 cores/8 threads. Notice, that after using 4 threads, performance doesn't scale much. Likely, getting a CPU with more cores won't improve performance.<sup>154</sup>

In reality, further adding computing nodes to the system may yield retrograde speed up. This effect is explained by Neil Gunther as [Universal Scalability Law](#)<sup>155</sup> (USL) which is an extension of Amdahl's law. USL describes communication between computing nodes (threads) as yet another gating factor against performance. As the system is scaled up, overheads start to hinder the gains. Beyond a critical point, the capability of the system starts to decrease (see fig. 33). USL is widely used for modeling capacity and scalability of the systems.

Communication overhead of `h264dec` benchmark on Intel Core i5-8259U can be observed on Figure 32b. Notice how the benchmark experience more overhead both in terms of executed instructions and elapsed core cycles as we assign more than 4 threads to the task.<sup>156</sup>

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

---

<sup>152</sup>[https://en.wikipedia.org/wiki/Producer-T1\textendashconsumer\\_problem](https://en.wikipedia.org/wiki/Producer-T1\textendashconsumer_problem)

<sup>153</sup>[https://en.wikipedia.org/wiki/Amdahl's\\_law](https://en.wikipedia.org/wiki/Amdahl's_law)

<sup>154</sup>However, it will benefit from a CPU with higher frequency.

<sup>155</sup>[http://www.perfdynamics.com/Manifesto/USLscalability.html#tth\\_sEc1](http://www.perfdynamics.com/Manifesto/USLscalability.html#tth_sEc1)

<sup>156</sup>There is interesting spike in the number of retired instruction when using 5 and 6 worker threads. This should be investigated by profiling the workload.



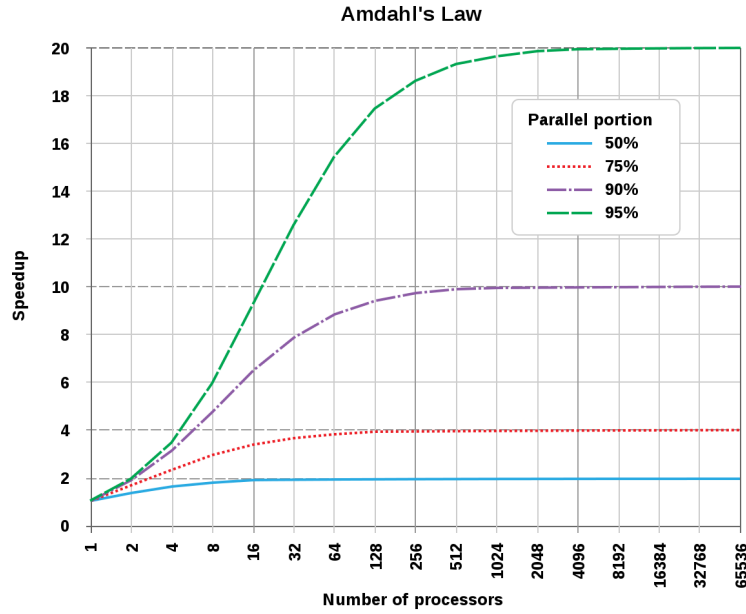
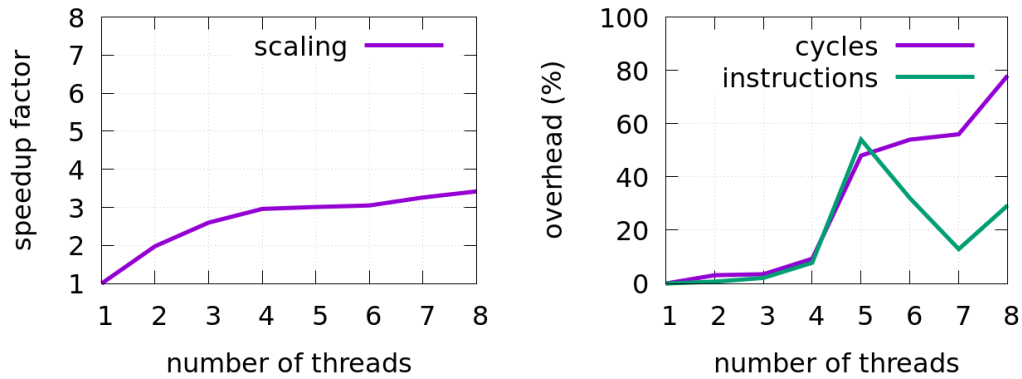


Figure 31: The theoretical speedup of the latency of the execution of a program as a function of the number of processors executing it, according to Amdahl's law. Taken from: [https://en.wikipedia.org/wiki/Amdahl's\\_law#/media/File:AmdahlsLaw.svg](https://en.wikipedia.org/wiki/Amdahl's_law#/media/File:AmdahlsLaw.svg)



(a) Performance scaling with different number of threads. (b) Overhead of using different number of threads.

Figure 32: Performance scaling and overhead of h264dec benchmark on Intel Core i5-8259U.

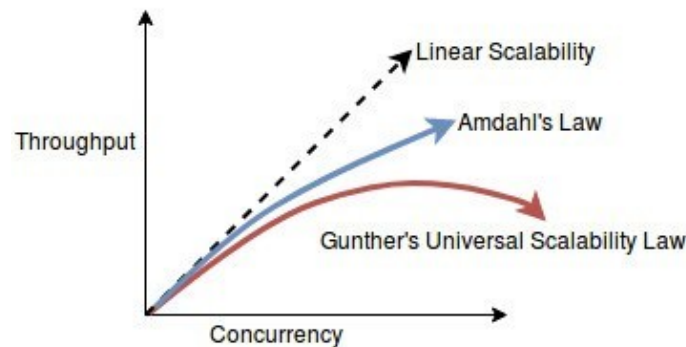


Figure 33: Universal Scalability Law and Amdahl's law. Taken from:

contention and coherence. The following subsections will describe techniques for addressing these additional challenges for tuning multithreaded programs.

## 8.2 Parallel Efficiency Metrics.

When dealing with multithreaded applications engineers should be careful with analyzing basic metrics like CPU utilization and IPC (see sec. 4). One of the threads can show high CPU utilization and high IPC, but it could turn out that all it was doing was just spinning on a lock. That's why when evaluating the parallel efficiency of the application, it's recommended to use Effective CPU Utilization, which is based only on the Effective time<sup>157</sup>.

### 8.2.1 Effective CPU Utilization.

Represents how efficiently the application utilized the CPUs available. It shows the percent of average CPU utilization by all logical CPUs on the system. CPU utilization metric is based only on the Effective time and does not include the overhead introduced by the parallel runtime system<sup>158</sup> and Spin time. A CPU utilization of 100% means that your application keeps all the logical CPU cores busy for the entire time that it runs. [Int, 2020]

For a specified time interval T, Effective CPU Utilization can be calculated as

$$Effective\ CPU\ Utilization = \frac{\sum_{i=1}^{ThreadsCount} Effective\ Cpu\ Time(T, i)}{T * ThreadsCount}$$

### 8.2.2 Thread Count.

Applications usually have configurable number of threads which allows them to run efficiently on platform with different number of cores. Obviously, running application using a lower number of threads than is available on the system underutilizes its resources. On the other hand, running an excessive number of threads can cause a higher CPU time because some of the threads may be waiting on others to complete or time may be wasted on context switches.

Besides actual worker threads, multithreaded applications usually have helper threads: main thread, input and output threads, etc. If those threads consume significant time, they require dedicated HW thread themselves. This is why it is important to know the total thread count and configure the number of worker threads properly.

To avoid a penalty for threads creation and destruction, engineers usually allocate a [pool of threads](#)<sup>159</sup> 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.

### 8.2.3 Wait Time.

Occurs when software threads are waiting due to APIs that block or cause synchronization. Wait Time is per-thread; therefore, the total Wait Time can exceed the application Elapsed Time. [Int, 2020]

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

<sup>157</sup>Performance analysis tools such as Intel VTune Profiler can distinguish a profiling samples that were taken while the thread was spinning. They do that with the help of call stacks for every sample (see sec. 5.4.3).

<sup>158</sup>Threading libraries and APIs like pthread, OpenMP and Intel TBB have their own overhead for creating and managing threads.

<sup>159</sup>[https://en.wikipedia.org/wiki/Thread\\_pool](https://en.wikipedia.org/wiki/Thread_pool)

Time. Big amount of Sync Wait Time likely indicates that 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](#)<sup>160</sup> problem either because of big number of application threads or a conflict with OS threads or other applications on the system. In this case developer should consider reducing the total number of threads or increasing task granularity for every worker thread.

#### 8.2.4 Spin Time.

Spin time is Wait Time during which the CPU is busy. This often occurs when a synchronization API causes the CPU to poll while the software thread is waiting. [Int, 2020]. In reality, implementation of kernel synchronization primitives prefers to spin on a lock for some time to the alternative of doing immediate thread context switch (which is expensive). Too much Spin Time, however, can reflect lost opportunity for productive work.

### 8.3 Analysis With Intel VTune Profiler

Intel VTune Profiler has dedicated type of analysis for multithreaded applications called [Threading Analysis](#). Its summary window (see fig. 34) displays statistics on the overall application execution, identifying all the metrics we described in sec. 8.2. From Effective CPU Utilization Histogram, we could learn several interesting facts about the captured applications behavior. First, on the average only 5 HW threads (logical cores on the diagram) were utilized at the same time. Second, almost never all 8 HW threads were active at the same time.

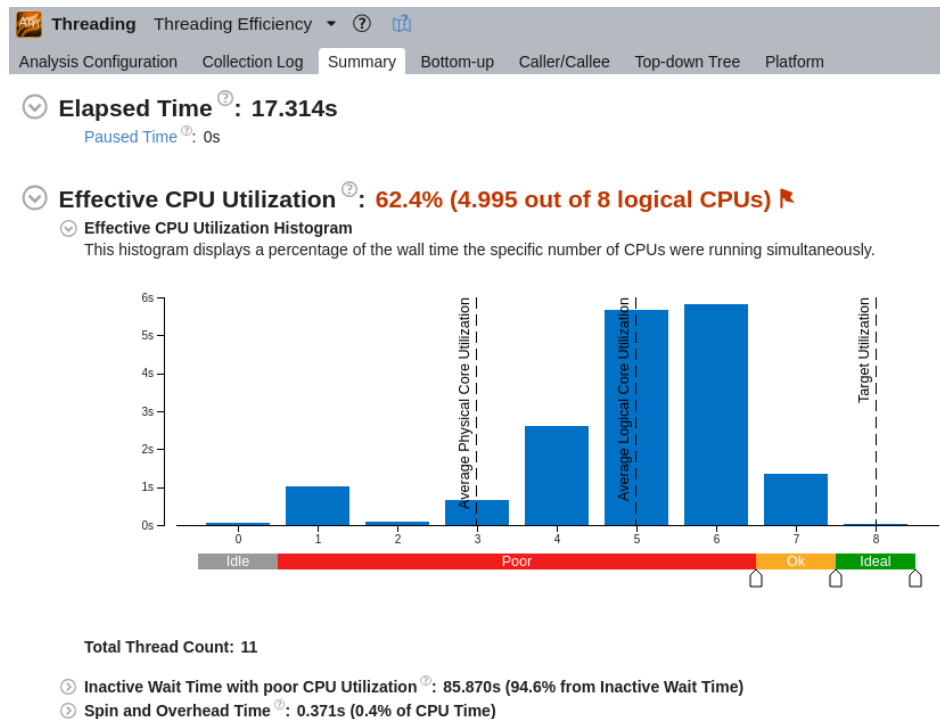


Figure 34: Intel VTune Profiler Threading Analysis summary for [x264](#) benchmark from [Phoronix test suite](#).

<sup>160</sup><https://software.intel.com/en-us/vtune-help-thread-oversubscription>

### 8.3.1 Find Expensive Locks.

Next, the workflow suggest that we identify the most contended synchronization objects. Figure 35 shows the list of such objects. We can see that `__pthread_cond_wait` definitely stands out, but since we might have dozens of conditional variables in the program, we need to know which one is the reason for poor CPU utilization.

☑ Inactive Wait Time with poor CPU Utilization <sup>Ⓜ</sup>: 85.870s (94.6% from Inactive Wait Time)

Inactive Sync Wait Time <sup>Ⓜ</sup>: 85.508s  
Preemption Wait Time <sup>Ⓜ</sup>: 0.361s

☑ Top functions by Inactive Wait Time with Poor CPU Utilization.  
This section lists the functions sorted by the time spent waiting on synchronization or thread preemption with poor CPU Utilization.

Function	Module	Inactive Wait Time <sup>Ⓜ</sup>	Inactive Sync Wait Time <sup>Ⓜ</sup>	Inactive Sync Wait Count <sup>Ⓜ</sup>
<code>__pthread_cond_wait</code>	libpthread-2.27.so	84.596s	84.594s	24,903
<code>_GI__pthread_mutex_lock</code>	libpthread-2.27.so	0.889s	0.889s	79
[vmlinux]	vmlinux	0.374s	0.016s	453
[vtsspp]	vtsspp	0.005s	0.005s	126
<code>_GI__pthread_mutex_unlock</code>	libpthread-2.27.so	0.002s	0s	0
[Others]		0.004s	0.003s	58

Figure 35: Intel VTune Profiler Threading Analysis showing the most contended synchronization objects for `x264` benchmark.

In order to know this, we can simply click on `__pthread_cond_wait` which will get us to the Bottom-Up view that is shown on fig. 36. We can see the most frequent path (47% of wait time) that lead to threads waiting on conditional variable: `__pthread_cond_wait <- x264_8_frame_cond_wait <- mb_analyse_init`.

**Threading** Threading Efficiency ▾ ? ⓘ

Analysis Configuration Collection Log Summary **Bottom-up** Caller/Callee Top-down Tree Platform

Grouping: Function / Call Stack

Function / Call Stack	CPU Time <sup>Ⓜ</sup>	Inactive Wait Time <sup>Ⓜ</sup>		Inactive Wait Count <sup>Ⓜ</sup>		Module
		Inactive Sync Wait Time <sup>Ⓜ</sup>	Preemption Wait Time <sup>Ⓜ</sup>	Inactive Sync Wait Count <sup>Ⓜ</sup>	Preemption Wait Count <sup>Ⓜ</sup>	
▼ <code>__pthread_cond_wait</code>	0.241s	89.428s	0.002s	26,375	4	libpthread-2.27.so
▶ <code>x264_8_frame_cond_wait &lt;- mb_analyse_init &lt;-</code>	0.216s	42.040s	0.002s	24,239	3	x264
▶ <code>threadpool_thread_internal &lt;- x264_stack_align</code>	0.015s	29.530s	0.000s	1,464	1	x264
▶ <code>x264_8_threadpool_wait &lt;- encoder_frame_end</code>	0.008s	14.377s	0s	491	0	x264
▶ <code>lookahead_thread_internal &lt;- x264_stack_align</code>	0.002s	2.562s	0s	139	0	x264
▶ <code>x264_8_lookahead_get_frames &lt;- x264_8_encod</code>	0s	0.918s	0s	42	0	x264
▶ <code>_GI__pthread_mutex_lock</code>	0.016s	0.895s	0.000s	87	1	libpthread-2.27.so
▶ [vmlinux]	0.520s	0.017s	0.391s	467	1,134	vmlinux
▶ [vtsspp]	0.002s	0.005s	0s	131	0	vtsspp
▶ <code>__pthread_cond_broadcast</code>	0.084s	0.001s	0.001s	4	5	libpthread-2.27.so

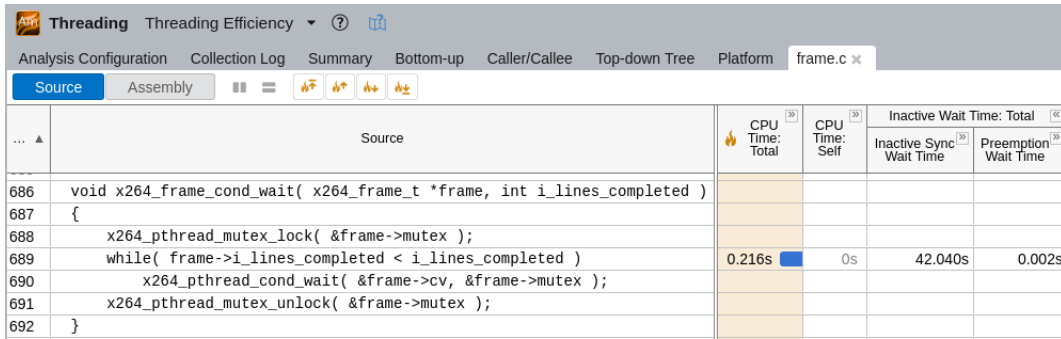
Figure 36: Intel VTune Profiler Threading Analysis showing the call stack for the most contended conditional variable in `x264` benchmark.

We can next jump into the source code of `x264_8_frame_cond_wait` by double-clicking on corresponding row in the analysis (see fig. 37). Next, we can study the reason behind the lock and possible ways to make thread communication in this place more efficient. <sup>161</sup>

### 8.3.2 Platform View.

Another very useful feature of Intel VTune Profiler is Platform View (see fig. 38) which allows to observe what each thread was doing in any given moment of program execution. This is very helpful for understanding the behavior of the application and finding potential performance

<sup>161</sup>I don't claim that it will be necessary an easy road and there is no guarantee that you will find the way to make it better.



Source	CPU Time: Total	CPU Time: Self	Inactive Wait Time: Total	Inactive Sync Wait Time	Preemption Wait Time
686 void x264_frame_cond_wait( x264_frame_t *frame, int i_lines_completed )					
687 {					
688 x264_thread_mutex_lock( &frame->mutex );					
689 while( frame->i_lines_completed < i_lines_completed )	0.216s	0s	42.040s	0.002s	
690 x264_thread_cond_wait( &frame->cv, &frame->mutex );					
691 x264_thread_mutex_unlock( &frame->mutex );					
692 }					

Figure 37: Source code view for x264\_8\_frame\_cond\_wait function in x264 benchmark.

headrooms. For example, we can see that during the time interval from 1s to 3s, only 2 threads were consistently utilizing ~100% of the corresponding CPU core (threads with TID 7675 and 7678). CPU utilization of other threads was bursty during that time interval.

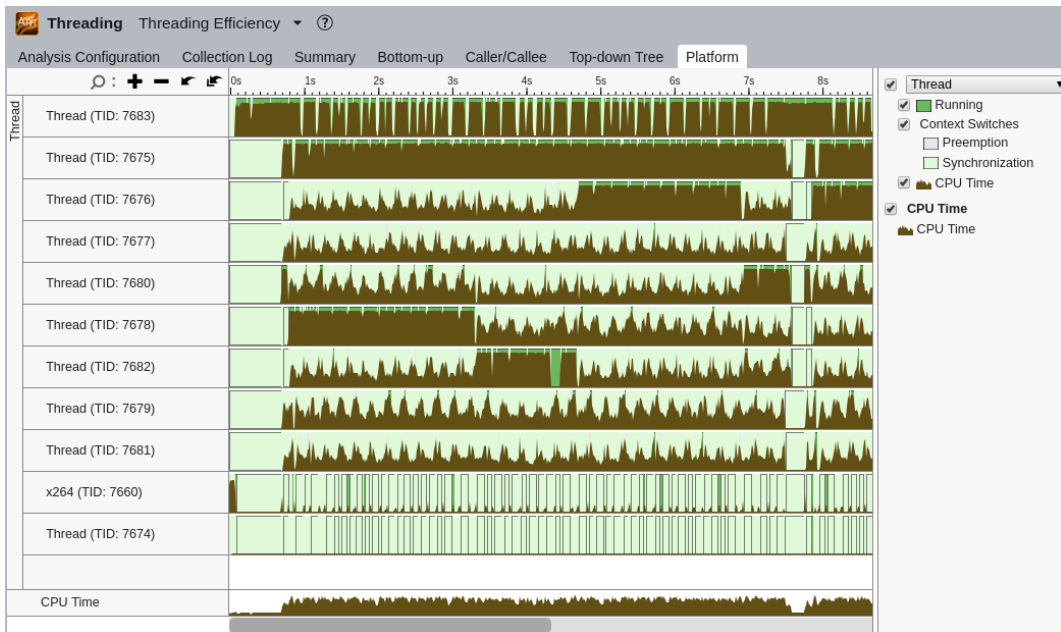


Figure 38: Vtune Platform view for x264 benchmark.

Platform View also has zooming and filtering capabilities. This allows to understand what each thread was executing during a specified time frame. To see this, select the range on the timeline, right-click and choose Zoom In and Filter In by Selection. Intel VTune Profiler will display functions or sync objects used during this time range.

**Personal Experience:** No other tool I worked with is anywhere near to Intel VTune Profiler for analyzing multithreaded applications. Performance analysis with Intel VTune Profiler is so convenient, especially thanks to its "zoom and filter" feature and platform view.

## 8.4 Analysis with Linux Perf

Linux `perf` tool profiles all the threads that the application might spawn. It has `-s` option which records per-thread event counts. Using this option, at the end of the report `perf` lists

all the thread IDs along with the number of samples collected for each of them:

```
$ perf record -s ./x264 -o /dev/null --slow --threads 8
  Bosphorus_1920x1080_120fps_420_8bit_YUV.y4m
$ perf report -n -T
...
#  PID    TID    cycles:ppp
6966 6976 41570283106
6966 6971 25991235047
6966 6969 20251062678
6966 6975 17598710694
6966 6970 27688808973
6966 6972 23739208014
6966 6973 20901059568
6966 6968 18508542853
6966 6967    48399587
6966 6966 2464885318
```

To filter samples for particular software thread, one can use `--tid` option:

```
$ perf report -T --tid 6976 -n
# Overhead Samples Shared Object Symbol
# .....
  7.17%    19877      x264      get_ref_avx2
  7.06%    19078      x264      x264_8_me_search_ref
  6.34%    18367      x264      refine_subpel
  5.34%    15690      x264      x264_8_pixel_satd_8x8_internal_avx2
  4.55%    11703      x264      x264_8_pixel_avg2_w16_sse2
  3.83%    11646      x264      x264_8_pixel_avg2_w8_mmx2
```

Linux `perf` also automatically provides some of the metrics we discussed in sec. 8.2:

```
$ perf stat ./x264 -o /dev/null --slow --threads 8
  Bosphorus_1920x1080_120fps_420_8bit_YUV.y4m
      86,720.71 msec task-clock      #    5.701 CPUs utilized
      28,386      context-switches  #    0.327 K/sec
      7,375      cpu-migrations    #    0.085 K/sec
      38,174      page-faults      #    0.440 K/sec
  299,884,445,581 cycles            #    3.458 GHz
  436,045,473,289 instructions      #    1.45 insn per cycle
  32,281,697,229 branches          #   372.249 M/sec
  971,433,345 branch-misses        #    3.01% of all branches
```

#### 8.4.1 Find Expensive Locks.

In order to find the most contended synchronization objects with Linux `perf` one needs to sample on scheduler context switches (`sched:sched_switch`), which is a kernel event and thus requires root access:

```
$ sudo perf record -s -e sched:sched_switch -g --call-graph dwarf -- ./x264
  -o /dev/null --slow --threads 8
  Bosphorus_1920x1080_120fps_420_8bit_YUV.y4m
```

```
$ sudo perf report -n --stdio -T --sort=overhead,prev_comm,prev_pid
--no-call-graph -F overhead,sample
# Samples: 27K of event 'sched:sched_switch'
# Event count (approx.): 27327
# Overhead      Samples      prev_comm      prev_pid
# .....
15.43%      4217      x264      2973
14.71%      4019      x264      2972
13.35%      3647      x264      2976
11.37%      3107      x264      2975
10.67%      2916      x264      2970
10.41%      2844      x264      2971
9.69%       2649      x264      2974
6.87%       1876      x264      2969
4.10%       1120      x264      2967
2.66%        727      x264      2968
0.75%        205      x264      2977
```

The output above shows which threads were switched out from the execution most frequently. Notice, we also collect call stacks (`--call-graph dwarf`, see sec. 5.4.3), since we need it for analyzing paths that lead to the expensive synchronization events:

```
$ sudo perf report -n --stdio -T --sort=overhead,symbol -F overhead,sample -G
# Overhead      Samples      Symbol
# .....
100.00%      27327      [k] __sched_text_start
|
|--95.25%--0xffffffffffffffff
| |
| |--86.23%--x264_8_macroblock_analyse
| | |
| | |--84.50%--mb_analyse_init (inlined)
| | |
| | |--84.39%--x264_8_frame_cond_wait
| | |
| | |--84.11%--__pthread_cond_wait (inlined)
| | |
| | |__pthread_cond_wait_common (inlined)
| | |
| | |--83.88%--futex_wait_cancelable (inlined)
| | |
| | |entry_SYSCALL_64
| | |do_syscall_64
| | |__x64_sys_futex
| | |do_futex
| | |futex_wait
| | |futex_wait_queue_me
| | |schedule
| | |__sched_text_start
...

```

On the listing above presented the most frequent path that lead to waiting on a conditional variable (`__pthread_cond_wait`) and later context switch. This path is



`x264_8_macroblock_analyse` -> `mb_analyse_init` -> `x264_8_frame_cond_wait`. From this output we can learn that 84% of all context switches were caused by threads waiting on a conditional variable inside `x264_8_frame_cond_wait`.

## 8.5 Analysis with Coz

In sec. 8.1 we stated the challenge of identifying parts of code that affects overall performance of a multithreaded program. Due to various reasons, optimizing one part of a multithreaded program might not always give visible results. [Coz<sup>162</sup>](#) is a new kind of profiler that addresses this problem and fills the gaps left behind by traditional software profilers. It uses a novel technique called “causal profiling”, whereby experiments are conducted during the runtime of an application by virtually speeding up segments of code to predict the overall effect of certain optimizations. It accomplishes these “virtual speedups” by inserting pauses that slow down all other concurrently running code. [Curtsinger and Berger, 2015]

Example of applying Coz profiler to [C-Ray](#) benchmark from [Phoronix test suite](#) is shown on 39. According to the chart, if we improve the performance of line 540 in `c-ray-mt.c` by 20%, Coz expects a corresponding increase in application performance of C-Ray benchmark overall of about 17%. Once we reach ~45% improvement on that line, the impact on the application begins to level off by COZ’s estimation. For more details on this example see the [article<sup>163</sup>](#) on [easyperf](#) blog.

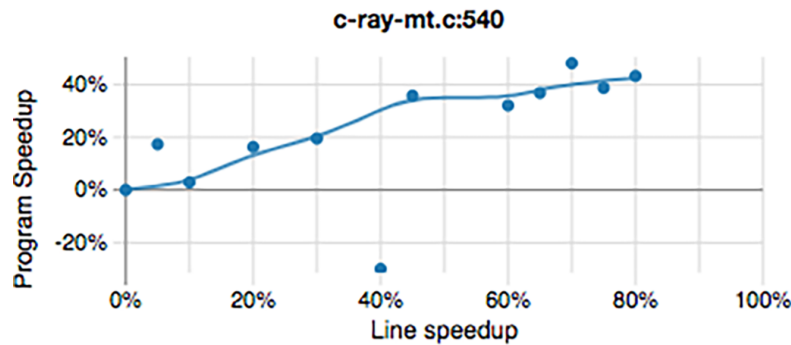


Figure 39: Coz profile for [C-Ray](#) benchmark.

## 8.6 Analysis with eBPF and GAPP.

Linux supports a variety of thread synchronization primitives – mutexes, semaphores, condition variables, etc. The kernel supports these thread primitives via the `futex` system call. Therefore, by tracing the execution of the `futex` system call in the kernel while gathering useful metadata from the threads involved, contention bottlenecks can be more readily identified. Linux provides kernel tracing/profiling tools that make this possible, none more powerful than [Extended Berkley Packet Filter<sup>164</sup>](#) (eBPF).

eBPF is based around a sandboxed virtual machine running in the kernel that allows execution of user-defined programs safely and efficiently inside the kernel. The user-defined programs can be written in C and compiled into BPF bytecode by the [BCC compiler<sup>165</sup>](#) in preparation

<sup>162</sup><https://github.com/plasma-umass/coz>

<sup>163</sup><https://easyperf.net/blog/2020/02/26/coz-vs-sampling-profilers>

<sup>164</sup><https://prototype-kernel.readthedocs.io/en/latest/bpf/>

<sup>165</sup><https://github.com/iovisor/bcc>



for loading into the kernel VM. These BPF programs can be written to launch upon the execution of certain kernel events and communicate raw or processed data back to user space via a variety of means.

The Opensource Community has provided many eBPF programs for general use, one such tool is the [Generic Automatic Parallel Profiler \(GAPP\)](#), that helps tracking multithreaded contention issues. GAPP uses eBPF to track contention overhead of a multithreaded application by ranking the criticality of identified serialization bottlenecks, collects stack traces of threads that were blocked and the one that caused the blocking. The best thing about GAPP is that it does not require code changes, expensive instrumentation, or recompilation. Creators of GAPP profiler were able to confirm known bottlenecks and also expose new, previously unreported bottlenecks on [Parsec 3.0 Benchmark Suite](#)<sup>166</sup> and some large open-source projects. [Nair and Field, 2020]

## 8.7 Detecting Coherence Issues

### 8.7.1 Cache Coherency Protocols

Multiprocessor systems incorporate Cache Coherency Protocols to ensure data consistency during shared usage of memory by each individual core containing its own, separate cache entity. Without such a protocol, if both CPU A and CPU B read memory location L into their individual caches, and processor B subsequently modified its cached value for L, then the CPUs would have inconsistent values of the same memory location L. Cache Coherency Protocols ensure that any updates to cached entries are dutifully updated in any other cached entry of the same location.

One of the most well-known cache coherency protocols is MESI (**M**odified **E**xclusive **S**hared **I**nvalid) which is used to support writeback caches like those used in modern CPUs. Its acronym denotes the 4 states with which a cache line can be marked (see fig. 40):

- **Modified** – cache line is present only in the current cache and has been modified from its value in RAM
- **Exclusive** – cache line is present only in the current cache and matches its value in RAM
- **Shared** – cache line is present here and in other cache lines and matches its value in RAM
- **Invalid** – cache line is unused (i.e., does not contain any RAM location)

When fetched from memory, each cache line has one of the states encoded into its tag. Then the cache line state keeps transiting from one state to another<sup>167</sup>. In reality, CPU vendors usually implement slightly improved variants of MESI. For example, Intel uses [MESIF](#)<sup>168</sup>, which adds a Forwarding (F) state, while AMD employs [MOESI](#)<sup>169</sup>, which adds the Owing (O) state. But these protocols still maintain the essence of the base MESI protocol.

As earlier example demonstrates, cache coherency problem can cause sequentially inconsistent programs. This problem can be mitigated by having snoopy caches to watch all memory transaction and cooperate with each other to maintain memory consistency. Unfortunately, it comes with a cost, since modification done by one processor invalidates the corresponding cache

<sup>166</sup><https://parsec.cs.princeton.edu/index.htm>

<sup>167</sup>Readers can watch and test animated MESI protocol here: <https://www.scss.tcd.ie/Jeremy.Jones/vivio/caches/MESI.htm>.

<sup>168</sup>[https://en.wikipedia.org/wiki/MESIF\\_protocol](https://en.wikipedia.org/wiki/MESIF_protocol)

<sup>169</sup>[https://en.wikipedia.org/wiki/MOESI\\_protocol](https://en.wikipedia.org/wiki/MOESI_protocol)

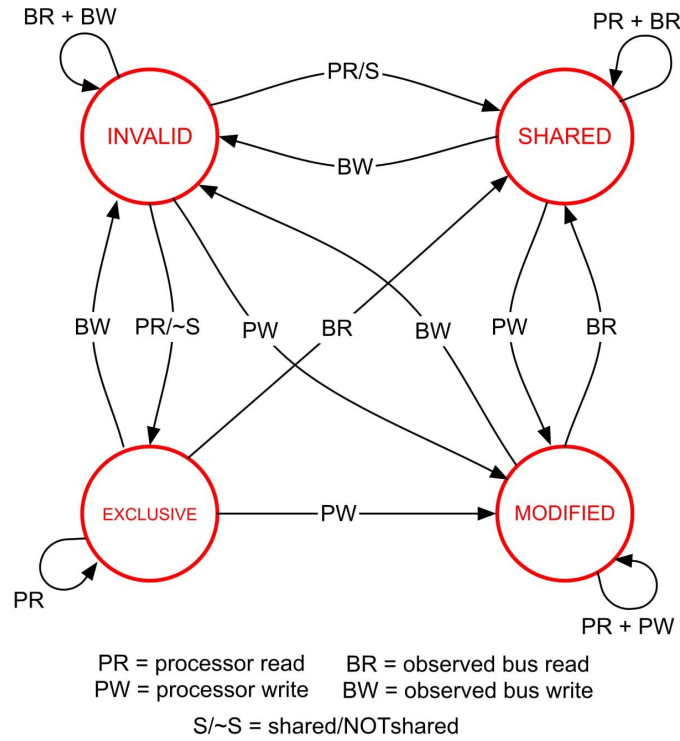


Figure 40: MESI States Diagram (from <https://courses.cs.washington.edu/>).

line in another processor’s cache. This causes memory stalls and wastes system bandwidth. In contrast to serialization and locking issues, which can only put a ceiling on performance of the application, coherence issues can cause retrograde effects as attributed by USL in sec. 8.1. Two widely known types of coherence problems are “True Sharing” and “False Sharing” which we will explore next.

### 8.7.2 True Sharing

True sharing occurs when two different processors access the same variable (see Listing 27).

---

#### Listing 27 True Sharing Example.

---

```
unsigned int sum;
{ // parallel section
  for (int i = 0; i < N; i++)
    sum = a[i]; // sum is shared between all threads
}
```

---

### 8.7.3 False Sharing

False Sharing occurs when two different processors modify different variables that happen to reside on the same cache line (see Listing 28). Figure 41 illustrates false sharing problem.

**Listing 28** False Sharing Example.

```

struct S {
    int sumA; // sumA and sumB are likely to
    int sumB; // reside in the same cache line
};
S s;

{ // section executed by thread A
    for (int i = 0; i < N; i++)
        s.sumA = a[i];
}

{ // section executed by thread B
    for (int i = 0; i < N; i++)
        s.sumB = b[i];
}

```

**8.7.4 Detecting Sharing Issues**

Because false sharing is a frequent source of performance issues for multithreaded applications, modern analysis tools have built-in support for detecting such cases. TMAM characterizes applications that experience true/false sharing as Memory Bound. Typically, in such cases you would see a high value for [Contested Accesses](#)<sup>170</sup> metric.

When using Intel VTune Profiler, user needs two types of analysis to find and eliminate false sharing issues. Firstly, run [Microarchitecture Exploration](#)<sup>171</sup> analysis that implements TMAM methodology to detect the presence of false sharing in the application. As noted before, high value for Contested Accesses metric prompts us to dig deeper and run the [Memory Access](#) analysis with the “Analyze dynamic memory objects” option enabled. This analysis helps in finding out accesses to the data structure that caused contention issues. Typically, such memory accesses have high latency which will be revealed by the analysis. See example of using Intel VTune Profiler for fixing false sharing issue on [Intel Developer Zone](#)<sup>172</sup>.

Linux `perf` has support for finding false sharing as well. As with Intel VTune Profiler, run TMAM first (see sec. 6.1.2) to find out that the program experience false/true sharing issues. If that’s the case, use `perf c2c` tool to detect memory accesses with high cache coherency cost. `perf c2c` matches store/load addresses for different threads and see if the hit in a modified cacheline occurred. Readers can find detailed explanation of the process and how to use the tool in dedicated [blog post](#)<sup>173</sup>.

**8.7.5 Avoiding Sharing Issues**

True sharing can usually be avoided with the help of Thread Local Storage (TLS). TLS is the method by which each thread in a given multithreaded process can allocate locations in which to store thread-specific data. By doing so, thread modify their local copies instead of contending for a globally available memory location. Example in sec. 8.7.2 can be fixed by

<sup>170</sup><https://software.intel.com/en-us/vtune-help-contested-accesses>

<sup>171</sup><https://software.intel.com/en-us/vtune-help-general-exploration-analysis>

<sup>172</sup><https://software.intel.com/en-us/vtune-cookbook-false-sharing>

<sup>173</sup><https://joemario.github.io/blog/2016/09/01/c2c-blog/>

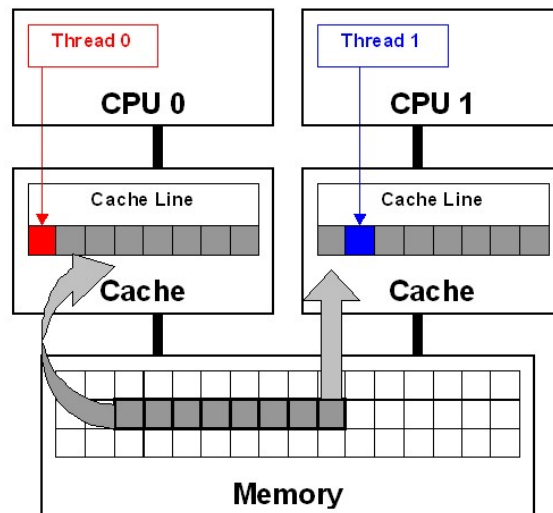


Figure 41: False Sharing: two threads access the same cache line (taken from [Intel Developer Zone](#)).

declaring `sum` with `thread_local` storage class specifiers (since C++11).

False sharing is possible to eliminate with the help of aligning/padding memory objects. Example in sec. 8.7.2 can be fixed by making sure `sumA` and `sumB` do not share the same cache line (see details in sec. 7.3.1.1.4).

From a general performance perspective, the most important thing to consider is the cost of the possible state transitions. Of all states, the only ones that do not involve a costly transition, involving cross-cache subsystem communication and data transfer, during CPU read/write operations are the Modified (M) and Exclusive (E) states. Thus, the longer the cacheline 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](#)"<sup>174</sup>.

## 8.8 Chapter Summary

- Applications not taking advantage of modern multicore CPUs are lagging behind their competitors. Preparing software to scale well with growing amount of CPU cores is very important for future success of the application.
- When dealing with single-threaded application, optimizing one portion of the program usually yields positive results on performance. However, it's not necessarily the case for multithreaded applications. This effect is widely known as Amdahl's law which constitutes that the speedup of a parallel program is limited by its serial part.
- Threads communication can yield retrograde speedup as explained by Universal Scalability Law. This poses additional challenges for tuning multithreaded programs. Optimizing performance of multithreaded applications also involves detecting and mitigating the effects of contention and coherence.
- Intel VTune Profiler is a "go-to" tool for analyzing multithreaded applications. But during the past years other tools emerged with unique set of features, e.g. Coz and GAPP.

<sup>174</sup><http://psy-lob-saw.blogspot.com/2013/09/diving-deeper-into-cache-coherency.html>

## References

- [1] Andrey Akinshin. *Pro .NET Benchmarking*. Apress, 1 edition, 2019. ISBN 978-1-4842-4940-6. doi: 10.1007/978-1-4842-4941-3.
- [2] Denis Bakhvalov. Code alignment issues. 2018. URL [https://easyperf.net/blog/2018/01/18/Code\\_alignment\\_issues](https://easyperf.net/blog/2018/01/18/Code_alignment_issues).
- [3] Denis Bakhvalov. Code alignment options in llvm. 2018. URL [https://easyperf.net/blog/2018/01/25/Code\\_alignment\\_options\\_in\\_llvm](https://easyperf.net/blog/2018/01/25/Code_alignment_options_in_llvm).
- [4] Dehao Chen, David Xinliang Li, and Tipp Moseley. Autofdo: Automatic feedback-directed optimization for warehouse-scale applications. In *CGO 2016 Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 12–23, New York, NY, USA, 2016.
- [5] Jiahao Chen and Jarrett Revels. Robust benchmarking in noisy environments, 2016.
- [6] Charlie Curtsinger and Emery Berger. Stabilizer: statistically sound performance evaluation. volume 48, pages 219–228, 03 2013. doi: 10.1145/2451116.2451141.
- [7] Charlie Curtsinger and Emery Berger. Coz: Finding code that counts with causal profiling. pages 184–197, 10 2015. doi: 10.1145/2815400.2815409.
- [8] *Data Never Sleeps 5.0*. Domo, Inc, 2017. URL [https://www.domo.com/learn/data-never-sleeps-5?aid=ogsm072517\\_1&sf100871281=1](https://www.domo.com/learn/data-never-sleeps-5?aid=ogsm072517_1&sf100871281=1).
- [9] Agner Fog. Optimizing software in c++: An optimization guide for windows, linux and mac platforms, 2004. URL [https://www.agner.org/optimize/optimizing\\_cpp.pdf](https://www.agner.org/optimize/optimizing_cpp.pdf).
- [10] Agner Fog. The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers. *Copenhagen University College of Engineering*, 2012. URL <https://www.agner.org/optimize/microarchitecture.pdf>.
- [11] Brendan Gregg. *Systems Performance: Enterprise and the Cloud*. Prentice Hall Press, USA, 1st edition, 2013. ISBN 0133390098.
- [12] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011. ISBN 012383872X.
- [13] Sorin Iacobovici, Sudarshan Kadambi, Yuan Chou, and Santosh Abraham. Effective stream-based and execution-based data prefetching. pages 1–11, 01 2004. doi: 10.1145/1006209.1006211.
- [14] *CPU Metrics Reference*. Intel® Corporation, 2020. URL <https://software.intel.com/en-us/vtune-help-cpu-metrics-reference>.
- [15] *Intel® 64 and IA-32 Architectures Software Developer Manuals*. Intel® Corporation, 2020. URL <https://software.intel.com/en-us/articles/intel-sdm>.
- [16] *Intel® VTune™ Profiler User Guide*. Intel® Corporation, 2020. URL <https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/analyze-performance/hardware-event-based-sampling-collection.html>.
- [17] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer.

- SIGARCH Comput. Archit. News*, 43(3S):158–169, June 2015. ISSN 0163-5964. doi: 10.1145/2872887.2750392. URL <https://doi.org/10.1145/2872887.2750392>.
- [18] Rajiv Kapoor. Avoiding the cost of branch misprediction. 2009. URL <https://software.intel.com/en-us/articles/avoiding-the-cost-of-branch-misprediction>.
  - [19] Paul-Virak Khuong and Pat Morin. Array layouts for comparison-based searching, 2015.
  - [20] Andi Kleen. An introduction to last branch records. 2016. URL <https://lwn.net/Articles/680985/>.
  - [21] Daniel Lemire. Making your code faster by taming branches. 2020. URL <https://www.infoq.com/articles/making-code-faster-taming-branches/>.
  - [22] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, page 265–276, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584065. doi: 10.1145/1508244.1508275. URL <https://doi.org/10.1145/1508244.1508275>.
  - [23] Reena Nair and Tony Field. Gapp: A fast profiler for detecting serialization bottlenecks in parallel linux applications. *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, Apr 2020. doi: 10.1145/3358960.3379136. URL <http://dx.doi.org/10.1145/3358960.3379136>.
  - [24] Nima Honarmand. Memory prefetching. URL <https://compas.cs.stonybrook.edu/~nhonarmand/courses/sp15/cse502/slides/13-prefetch.pdf>.
  - [25] Andrzej Nowak and Georgios Bitzes. The overhead of profiling using pmu hardware counters. 2014.
  - [26] Guilherme Ottoni and Bertrand Maher. Optimizing function placement for large-scale data-center applications. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, page 233–244. IEEE Press, 2017. ISBN 9781509049318.
  - [27] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. BOLT: A practical binary optimizer for data centers and beyond. *CoRR*, abs/1807.06735, 2018. URL <http://arxiv.org/abs/1807.06735>.
  - [28] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, pages 65–79, 2010. URL <http://www.computer.org/portal/web/csdl/doi/10.1109/MM.2010.68>.
  - [29] *Volume of data/information created worldwide from 2010 to 2025*. Statista, Inc, 2018. URL <https://www.statista.com/statistics/871513/worldwide-data-created/>.
  - [30] et al. Suresh Srinivas. Runtime performance optimization blueprint: Intel® architecture optimization with large code pages, 2019. URL <https://www.intel.com/content/www/us/en/develop/articles/runtime-performance-optimization-blueprint-intel-architecture-optimization-with-large-code.html>.