
通告

责任。工程和软件开发领域的知识和最佳实践不断变化。从业者和研究人员必须始终依靠自己的经验和知识来评估和使用此处描述的任何信息、方法、化合物或实验。在使用这些信息或方法时，他们应注意自己的安全和他人的安全，包括对其职业责任的各方。

在法律允许的最大范围内，作者、贡献者或编辑均不对任何因产品责任、疏忽或其他原因而导致的人身或财产伤害承担任何责任，也不对此处所述材料中的任何方法、产品、说明或观念的使用或操作承担责任。

商标。公司用来区分其产品的名称通常被视为商标或注册商标。Intel、Intel Core、Intel Xeon、Intel Pentium、Intel Vtune 和 Intel Advisor 是 Intel Corporation 在美国和/或其他国家的商标。AMD 是 Advanced Micro Devices Corporation 在美国和/或其他国家的商标。ARM 是 Arm Limited（或其子公司）在美国和/或其他地方的商标。然而，读者应联系相应的公司获取有关商标和注册的完整信息。

从属关系。在撰写本书时，主要作者（Denis Bakhvalov）是 Intel Corporation 的雇员。本书中提供的所有信息都不是上述公司的官方立场，而是作者个人的知识和观点。主要作者在撰写本书时没有收到 Intel Corporation 的任何财务赞助。

广告。本书不宣传任何软件、硬件或其他产品。

版权

版权所有 © 2020 Denis Bakhvalov，采用创作共用许可证（CC BY 4.0）。

前言

关于作者

Denis Bakhvalov 是 Intel 的高级开发人员，负责 C++ 编译器项目，旨在为各种不同架构生成优化代码。性能工程和编译器始终是他主要关注的领域之一。Denis 于 2008 年开始作为软件开发人员的职业生涯，并在多个领域工作，包括开发桌面应用程序、嵌入式系统、性能分析和编译器开发。2016 年，Denis 开始了他的 [easyperf.net](#) 博客，专注于性能分析和调优、C/C++ 编译器以及 CPU 微体系结构。Denis 是积极生活方式的支持者，在业余时间喜欢踢足球、打网球、跑步和下棋。此外，Denis 还是两个美丽女儿的父亲。

联系方式：

- 电子邮件：dendibakh@gmail.com
- Twitter：[@dendibakh](#)
- LinkedIn：[@dendibakh](#)

作者的话

我开始撰写这本书的初衷很简单：教育软件开发人员更好地理解其应用程序在现代硬件上的性能。我知道这对于初学者甚至有经验的开发人员来说可能是一个令人困惑的话题。这种困惑主要发生在没有先前处理性能相关任务的开发人员身上。这没关系，因为每个专家都曾经是初学者。

我记得我开始进行性能分析的日子。我盯着陌生的度量标准，试图匹配不符合的数据。我感到困惑。直到最后一切“豁然开朗”，所有的拼图才拼在一起。那时候，唯一的良好信息来源是软件开发人员手册，而这并不是主流开发人员喜欢阅读的。因此，我决定写这本书，希望能让开发人员更容易学习性能分析的概念。

认为自己在性能分析方面是初学者的开发人员可以从本书的开头开始顺序阅读，逐章阅读。第 2 至 4 章为开发人员提供了后续章节所需的最低知识集。已经熟悉这些概念的读者可以选择跳过这些章节。此外，本书还可以用作优化软件应用程序的参考或检查清单。开发人员可以将第 7 至 11 章作为调整其代码思路的来源。

目标受众

本书将主要对那些从事性能关键应用程序和进行低级优化的软件开发人员有用。仅举几个领域：高性能计算 (HPC)、游戏开发、数据中心应用程序（如 Facebook、Google 等）、高频交易。但是，本书的范围并不仅限于上述行业。任何想更好地了解其应用程序性能并知道如何进行诊断和改进的开发人员都会从中受益。作者希望本书所呈现的材料能帮助读者培养新的技能，可以应用于他们的日常工作中。

读者预期具备 C/C++ 编程语言的最低背景，以理解本书的示例。能够阅读基本的 x86 汇编语言是理想的，但不是严格要求。作者还期望读者熟悉计算机体系结构和操作系统的基本概念，如中央处理器、内存、进程、线程、虚拟和物理内存、上下文切换等。如果上述术语中有任何新的术语，建议先学习这些材料。

致谢

非常感谢 Mark E. Dawson，他在撰写本书的几个部分时提供了帮助：《为 DTLB 优化》(Section 7.4)、《为 ITLB 优化》(Section 10.6)、《缓存预热》(Section 10.10.2)、系统调优 (Section 10.12)、关于多线程应用程序性能扩展和开销的部分 (Section 11.1)、使用 COZ 性能分析器的部分 (Section 11.5)、关于 eBPF 的部分 (Section 11.6)、《检测一致性问题》(Section 11.7)。Mark 是高频交易行业的知名专家。在撰写本书的不同阶段，Mark 非常乐意分享他的专业知识和反馈意见。

接下来，我要感谢 Sridhar Lakshmanamurthy，在关于 CPU 微体系结构的部分（Chapter 3）中撰写了大部分内容。Sridhar 在 Intel 工作了几十年，是半导体行业的老手。

非常感谢 Nadav Rotem，他是 LLVM 编译器中矢量化框架的原始作者，他帮助我撰写了关于矢量化的部分（Section 8.4）。

Clément Grégoire 编写了关于 ISPC 编译器的部分（Section 8.4.2.5）。Clément 在游戏开发行业拥有丰富的背景。他的评论和反馈帮助解决了游戏开发行业中一些挑战。

如果没有以下评论员的帮助，本书不可能从草稿中走出来：Dick Sites、Wojciech Muła、Thomas Dullien、Matt Fleming、Daniel Lemire、Ahmad Yasin、Michele Adduci、Clément Grégoire、Arun S. Kumar、Surya Narayanan、Alex Blewitt、Nadav Rotem、Alexander Yermolovich、Suchakrapani Datt Sharma、Renat Idrisov、Sean Heelan、Jumana Mundichippakkal、Todd Lipcon、Rajiv Chauhan、Shay Morag 等等。

此外，我还要感谢整个性能社区为无数的博客文章和论文。我通过阅读 Travis Downs、Daniel Lemire、Andi Kleen、Agner Fog、Bruce Dawson、Brendan Gregg 等人的博客，学到了很多东西。我站在巨人的肩膀上，本书的成功不仅归功于我自己。本书是我向整个社区致敬和回馈的方式。

最后但并非最不重要的是，感谢我的家人，他们耐心地容忍我错过周末旅行和晚间散步。没有他们的支持，我不可能完成这本书。

Table Of Contents

Table Of Contents	4
1 介绍	9
1.1 为什么我们仍然需要性能调优?	10
1.2 谁需要性能调优?	11
1.3 什么是性能分析?	12
1.4 本书讨论了什么?	13
1.5 本书未涉及的内容	13
1.6 练习	14
第一部分：在现代 CPU 上的性能分析	15
2 性能测量	15
2.1 现代系统中的噪声	15
2.2 在生产环境中进行性能测量	17
2.3 自动检测性能回归	17
2.4 手动性能测试	19
2.5 假设检验方法	20
2.6 软件和硬件定时器	22
2.7 微基准测试	23
3 CPU 微体系结构	25
3.1 指令集架构	25
3.2 流水线技术	25
3.3 开发指令级并行性 (ILP)	27
3.3.1 乱序执行 (OOO Execution)	27
3.3.2 超标量引擎和 VLIW	28
3.3.3 推测执行	28
3.3.4 分支预测	29
3.4 SIMD 多处理器	30
3.5 开发线程级并行性	31
3.5.1 多核系统	32
3.5.2 同时多线程	32
3.5.3 混合架构	33
3.6 存储器层次结构	34
3.6.1 缓存层次结构	34
3.6.2 主存储器	36
3.7 虚拟内存	39
3.7.1 地址翻译缓冲区 (TLB)	41

3.7.2	大页	41
3.8	现代 CPU 设计	42
3.8.1	CPU 前端	43
3.8.2	CPU 后端	43
3.8.3	Load-Store Unit	44
3.8.4	TLB 层次结构	45
3.9	性能监控单元	46
3.9.1	性能监控计数器	46
4	性能分析中的术语和指标	49
4.1	已退役 vs. 已执行指令	49
4.2	CPU 利用率	49
4.3	CPI 和 IPC	50
4.4	微操作	51
4.5	管道槽	52
4.6	核心周期与参考周期	52
4.7	缓存失效	53
4.8	错误预测的分支	54
4.9	性能指标	55
4.10	内存延迟和带宽	57
4.11	案例研究：分析四个基准测试的性能指标	60
4.12	产能规划练习	64
5	性能分析方法	66
5.1	代码仪器化	66
5.2	跟踪	69
5.3	工作负载特征化	70
5.3.1	计数性能事件	71
5.3.2	手动收集性能计数器数据	71
5.3.3	多路复用和事件缩放	72
5.4	使用标记器 API	73
5.5	采样	76
5.5.1	用户模式和基于硬件事件的采样	77
5.5.2	寻找热点	77
5.5.3	收集调用堆栈	79
5.6	Roofline 性能模型	80
5.7	静态性能分析	84
5.7.1	案例研究：使用 UICA 优化 FMA 吞吐量	85
5.8	编译器优化报告	87
5.9	CPU 特性用于性能分析	91
5.10	自顶向下微架构分析 (TMA)	91
5.10.1	在英特尔平台上的 TMA	92
5.10.2	TMA 在 AMD 平台上	98
5.10.3	TMA 在 ARM 平台上	99
5.10.4	TMA 总结	101
5.11	分支记录机制 (Branch Recording Mechanisms)	103

5.12	未记录未执行的分支	104
5.12.1	英特尔平台上的 LBR	104
5.12.2	AMD 平台上的 LBR	105
5.13	ARM 平台上的 BRBE	105
5.13.1	捕获调用堆栈	106
5.13.2	识别热点分支 (#sec:lbr_hot_branch)	106
5.13.3	分析分支预测错误率 (#sec:secLBR_misp_rate)	107
5.13.4	机器码的精确计时 (#sec:timed_lbr)	108
5.13.5	估计分支结果概率	110
5.13.6	提供编译器反馈数据	110
5.14	基于硬件的采样功能	110
5.14.1	英特尔平台上的 PEBS	110
5.14.2	AMD 平台上的 IBS	111
5.14.3	ARM 平台上的 SPE	112
5.14.4	精确事件	113
5.15	分析内存访问	113
6	性能分析工具概述	115
6.1	Intel Vtune	115
6.2	AMD uProf	118
6.3	Apple Xcode Instruments	121
6.4	Linux Perf	124
6.5	火焰图	125
6.6	Windows 事件跟踪	125
6.6.1	如何配置它	126
6.6.2	你能用它做什么:	126
6.6.3	你不能用它做什么:	126
6.7	记录 ETW 跟踪的工具	126
6.7.1	查看和分析 ETW 跟踪的工具	127
6.8	专业和混合性能分析器	130
6.9	Tracy 的其他功能	134
6.10	持续性能分析	135
第二部分：源代码优化		138
7	优化内存访问	141
7.1	缓存友好数据结构	141
7.1.1	顺序访问数据	142
7.1.2	使用合适的容器	142
7.1.3	数据压缩	142
7.1.4	对齐和填充	143
7.1.5	动态内存分配	144
7.1.6	调整代码以适应内存层次结构	145
7.2	显式内存预取	145
7.3	内存分析	148
7.4	减少 DTLB 未命中	148

7.4.1 显式大页面 (EHP)	149
7.4.2 透明大页面 (THP)	150
7.4.3 显式大页面 (EHP) vs. 透明大页面 (THP)	151
7.5 问题与练习	151
7.6 章节总结	151
8 优化计算	152
8.1 数据流依赖	152
8.2 内联函数	156
8.3 循环优化	157
8.3.1 低级别优化	158
8.3.2 高级优化	159
8.3.3 发现循环优化机会	161
8.3.4 循环优化框架	162
8.4 向量化	162
8.4.1 编译器自动向量化	163
8.4.2 发现向量化机会	164
8.5 使用编译器内部函数	168
8.5.1 内部函数的包装库	169
9 优化分支预测	172
9.1 用查找表替换分支	172
9.2 用算术替换分支	173
9.3 用谓词替换分支	173
10 机器代码布局优化	176
10.1 基本块	177
10.2 基本块布局	177
10.3 函数拆分	181
10.4 函数重排序	182
10.5 使用配置分析文件引导的优化 (Profile Guided Optimizations)	183
10.6 减少 ITLB 未命中	185
10.7 案例研究：测量代码足迹	186
10.8 其他优化领域	191
10.9 优化输入输出	191
10.10 低延迟优化技术	191
10.10.1 避免轻微页面错误	192
10.10.2 高速缓存预热	193
10.10.3 避免 TLB 驱逐	194
10.10.4 防止意外内核节流	195
10.11 缓慢的浮点运算	195
10.12 系统调优	196
10.13 案例研究：对最后一级缓存大小的敏感性	197
10.14 AMD Milan 7313P 处理器的集群式内存层次结构	198
10.14.1 控制和监控 LLC 分配	199
11 优化多线程应用	202

11.1 性能扩展和开销	202
11.2 并行效率指标	204
11.2.1 有效 CPU 利用率	204
11.2.2 线程数	204
11.2.3 等待时间	205
11.2.4 自旋时间	205
11.3 使用 Intel VTune Profiler 进行分析	205
11.3.1 查找开销大的锁	206
11.3.2 平台视图	206
11.4 使用 Linux Perf 进行分析	208
11.4.1 查找开销大的锁	209
11.5 使用 Coz 进行分析	210
11.6 利用 eBPF 和 GAPP 进行分析	210
11.7 缓存一致性问题	211
11.7.1 缓存一致性协议	211
11.7.2 真共享	211
11.8 假共享	213
11.9 软件和硬件性能的当前和未来趋势	216
后记	216
 术语表	218
 主要 CPU 微架构列表	219
 References	220
 附录 A. 减少测量噪声 (Reducing Measurement Noise)	224
 附录 B. LLVM 向量化 (Vectorizer)	228
 附录 C. 启用大页面	232
11.10 Windows	232
11.11 Linux	232
11.11.1 显式大页面 (unnumbered)	232
 附录 D. Intel 处理器跟踪	234

1 介绍

常说，“性能为王”。十年前是真的，现在当然也是如此。根据 [domo.com, 2017] 的数据，在 2017 年，全球每天创造了 2.5 个万亿¹字节的数据，如 [statista.com, 2018] 所预测的那样，这个数字每年增长 25%。在我们日益数据中心化的世界中，信息交换的增长推动了对更快软件（SW）和更快硬件（HW）的需求。可以说，数据增长不仅对计算能力提出了需求，还对存储和网络系统提出了需求。

在个人电脑时代²，开发人员通常直接在操作系统之上进行编程，可能之间会有一些库。随着世界进入云时代，软件堆栈变得更加深奥和复杂。开发人员大多数时间都在工作的顶层堆栈已经远离了硬件。这些额外的层次抽象出实际的硬件，从而允许使用新型加速器来处理新兴工作负载。然而，这种演进的负面影响是，现代应用程序的开发人员对其软件运行的实际硬件的了解较少。

多亏了摩尔定律，软件程序员几十年来一直“轻松愉快”。过去，一些软件供应商更倾向于等待新一代硬件以加速其应用程序，而不是花费人力资源改进其代码。从图 @fig:50YearsProcessorTrend 可以看出，单线程性能增长正在放缓。单线程性能是指在隔离环境中测量时 CPU 核心内的单个硬件线程的性能。

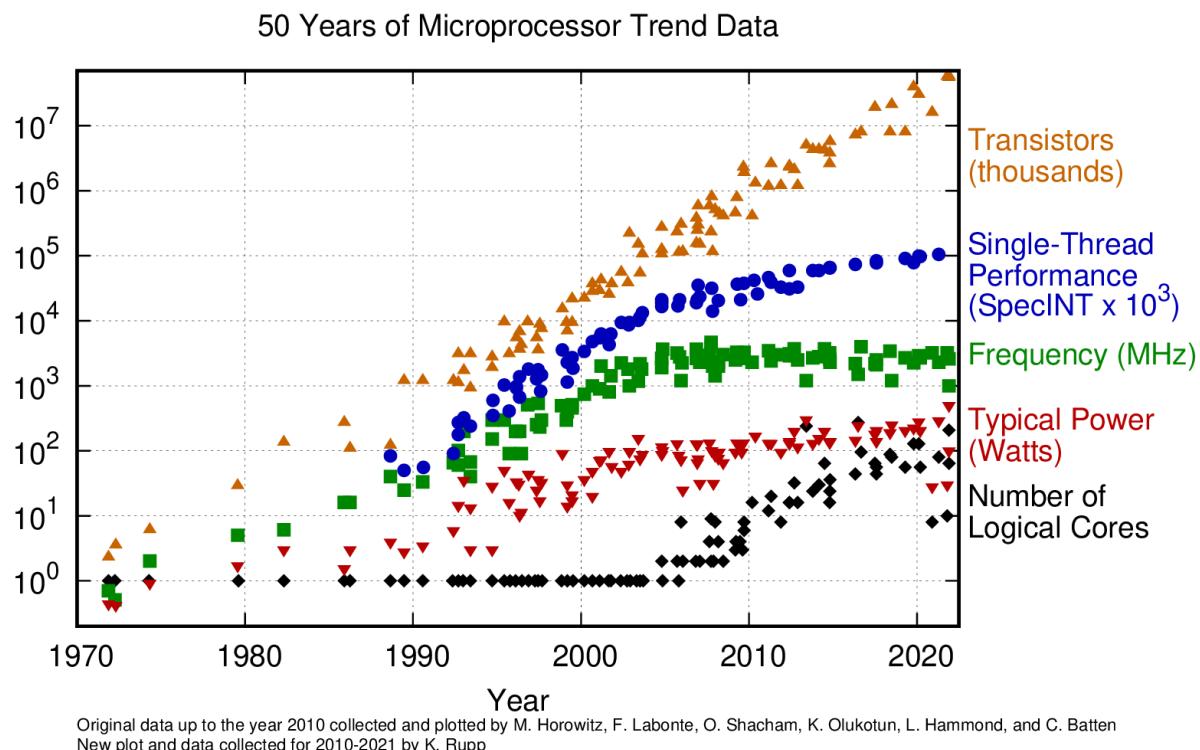


Figure 1: 50 Years of Microprocessor Trend Data. © Image by K. Rupp via karlrupp.net

当不再每个硬件世代都能提供显著的性能提升 [Leiserson et al., 2020] 时，我们必须开始更加关注代码运行的速度。在寻求提高性能的方法时，开发人员不应依赖硬件。相反，他们应该开始优化其应用程序的代码。

“当今软件非常低效；现在是软件程序员再次变得擅长优化的黄金时代。”- 美国企业家和投资者马克·安德森（Marc Andreessen），摘自 a16z Podcast，2020 年

¹ 万亿是千的六次方 (10^{18})。

² 从 1990 年代末到 2000 年代末，个人电脑占据了计算设备市场的主导地位。

Personal Experience: 在英特尔工作期间，我时常听到同样的故事：当英特尔的客户在其应用程序中遇到慢速时，他们会立即无意识地开始责怪英特尔的 CPU 速度慢。但是当英特尔派遣我们的性能高手与他们合作并帮助他们改进应用程序时，往往会将其加速 2 倍，有时甚至是 10 倍。

达到高水平的性能具有挑战性，并且通常需要大量的努力，但希望这本书能给你提供帮助的工具。

1.1 为什么我们仍然需要性能调优？

现代 CPU 每年都在增加越来越多的核心。截至 2019 年底，您可以购买到一款高端服务器处理器，其逻辑核心数量超过 100 个。这非常令人印象深刻，但这并不意味着我们不再需要关注性能。很多时候，随着 CPU 核心数量的增加，应用程序的性能可能并不会提升。典型的通用多线程应用程序的性能并不总是随着我们分配给任务的 CPU 核心数量线性增长的。了解为什么会发生这种情况以及可能的解决方法对于产品未来的增长至关重要。不能进行适当的性能分析和调优会导致性能和金钱的浪费，并可能毁掉产品。

根据 [Leiserson et al., 2020]，至少在短期内，大多数应用程序的性能收益的很大一部分将来自软件堆栈。可悲的是，应用程序默认情况下并不会获得最佳性能。该论文还提供了一个很好的例子，说明了在源代码级别上可以进行的性能改进潜力。在表 @tbl:PlentyOfRoom 中总结了对两个 4096×4096 矩阵相乘的程序进行性能工程的加速效果。通过应用多个优化，最终得到的程序运行速度提高了 60000 多倍。提供这个例子的原因不是挑剔 Python 或 Java（它们是很好的语言），而是打破了软件默认具有“足够好”的性能的观念。

Table 1: 在双插槽 Intel Xeon E5-2666 v3 系统上运行的一个程序，该程序执行两个 4096×4096 矩阵相乘，并具有总计 60GB 内存的加速效果。来源：[Leiserson et al., 2020]。

版本	实现	绝对加速	相对加速
1	Python	1	—
2	Java	11	10.8
3	C	47	4.4
4	并行循环	366	7.8
5	并行分治	6,727	18.4
6	加向量化	23,224	3.5
7	加 AVX 指令集	62,806	2.7

以下是阻止系统默认达到最佳性能的一些最重要因素：

1. **CPU 限制：**很容易问：“为什么硬件不能解决我们所有的问题？”现代 CPU 以令人难以置信的速度执行指令，并且每一代都在变得更好。但是，如果用于执行工作的指令不是最佳的，甚至是多余的，它们就无法做太多事情。处理器不能通过魔法将次优代码转换为性能更好的代码。例如，如果我们使用 BubbleSort 算法实现排序例程，CPU 将不会尝试识别并使用更好的替代方案，例如 QuickSort。它会盲目地执行被告知要执行的任何操作。
2. **编译器限制：**“但是编译器不是应该做这些吗？为什么编译器不能解决我们所有的问题？”的确，现在的编译器非常智能，但仍然可能生成次优代码。编译器擅长消除冗余工作，但是当涉及到更复杂的决策，如函数内联、循环展开等时，它们可能不会生成最佳的代码。例如，是否应该将一个函数始终内联到调用它的地方，这个问题没有二元的“是”或“否”答案。这通常取决于编译器应该考虑的许多因素。通常，编译器依赖于复杂的成本模型和启发式算法，这些算法可能不适用于每种可能的情况。此外，除非编译器确定这样做是安全的，并且不会影响生成的机器代码的正确性，否则它们不能执行优化。对于编译器开发人员来说，确保特定优化在所有可能的情况下生成正确的代码可能非常困难，因此他们通常必须保守行事，并避免进行一些优化。最后，编译器通常不会转换程序使用的数据结构，这在性能方面也是至关重要的。

3. 算法复杂度分析限制：开发人员经常过度关注算法的复杂度分析，这导致他们选择具有最优算法复杂度的流行算法，即使它对于给定问题可能不是最有效的。考虑两种排序算法，插入排序和快速排序，后者在平均情况的大O符号中显然更胜一筹：插入排序是 $O(N^2)$ ，而快速排序仅为 $O(N \log N)$ 。然而，对于相对较小的N值（最多 50 个元素），插入排序的性能优于快速排序。复杂度分析无法考虑各种算法的分支预测和缓存效果，因此人们只是将它们封装在隐含的常数C中，有时这可能会对性能产生重大影响。盲目地信任大O符号而没有在目标工作负载上进行测试可能会使开发人员走上错误的道路。因此，对于某个特定问题来说，最知名的算法并不一定是实践中最高效的。

以上所述的限制留下了调优我们的软件以发挥其全部潜力的余地。广义上来说，软件堆栈包括许多层，例如固件、BIOS、操作系统、库以及应用程序的源代码。但由于大多数较低的软件层不在我们的直接控制之下，因此重点将放在源代码上。我们将经常涉及的另一个重要的软件是编译器。通过让编译器生成所需的机器代码，可以获得令人满意的加速效果，通过各种提示方法。您将在本书中找到许多这样的例子。

\personal{要成功地在您的应用程序中实现所需的改进，您不必成为编译器专家。根据我的经验，至少 90% 的所有转换都可以在源代码级别完成，而无需深入研究编译器源代码。尽管如此，了解编译器的工作原理以及如何使其按照您的意愿进行操作在与性能相关的工作中始终是有利的。}

此外，如今，使应用程序能够通过将其分布在许多核心上扩展是至关重要的，因为单线程性能往往会上升到一个平台。这种启用需要应用程序各个线程之间的有效通信，消除资源的不必要消耗以及其他多线程程序常见的问题。

值得一提的是，性能收益不仅仅来自调整软件。根据 [Leiserson et al., 2020]，未来的两个主要潜在速度提升源是算法（特别是对于机器学习等新问题领域）和简化的硬件设计。算法显然在应用程序的性能中发挥了重要作用，但我们不会在本书中讨论这个主题。我们也不会讨论新硬件设计的主题，因为大多数情况下，软件开发人员必须处理现有的硬件。然而，了解现代 CPU 设计对于优化应用程序是重要的。

“在摩尔定律后的时代，使代码运行快速变得越来越重要，尤其是使其适合运行的硬件。”
[Leiserson et al., 2020]

本书的方法论侧重于从应用程序中挤出最后一点性能。这种转变可以归因于表 @tbl:PlentyOfRoom 中的第 6 行和第 7 行。将讨论的改进类型通常不大，并且通常不超过 10%。然而，不要低估 10% 的性能提升的重要性。这对于在云配置中运行的大型分布式应用程序尤为重要。根据 [Hennessy, 2018]，在 2018 年，谷歌在运行云的实际计算服务器上花费的资金大致与花费在电力和冷却基础设施上的资金相同。能源效率是一个非常重要的问题，可以通过优化软件来改善。

“在这种规模下，理解性能特征变得至关重要 - 即使是性能或利用率的小幅改进也可以转化为巨大的成本节省。” [Kanev et al., 2015]

1.2 谁需要性能调优？

在像高性能计算 (HPC)、云服务、高频交易 (HFT)、游戏开发和其他对性能要求极高的领域，性能工程无需多加辩解。例如，谷歌曾报告说，搜索慢 2% 导致每位用户搜索减少 2%。³ 对于 Yahoo! 来说，页面加载速度快了 400 毫秒导致流量增加了 5-9%。⁴ 在这个数字游戏中，微小的改进可能产生显著影响。这些例子证明，服务工作得越慢，使用的人就越少。

有一句著名的话：“过早优化是万恶之源”。但相反的情况也经常成立。推迟性能工程工作可能为时已晚，导致的问题可能和过早优化一样恶劣。对于从事性能关键项目的开发人员来说，了解底层硬件工作原理是至关重要的。在这些行业中，如果一个程序在开发过程中没有以硬件为焦点，那就是注定会失败的。软件的性能特性必须与正确性和

³ Marissa Mayer 的演讲幻灯片 - <https://assets.en.oreilly.com/1/event/29/Keynote%20Presentation%202.pdf>

⁴ Stoyan Stefanov 的演讲幻灯片 - <https://www.slideshare.net/stoyan/dont-make-me-wait-or-building-highperformance-web-applications>

安全性一样受到重视，从第一天开始。ClickHouse 数据库就是一个成功的软件产品，它是围绕一个小而非常高效的核心构建的。

有趣的是，性能工程不仅在上述领域需要。如今，在通用应用程序和服务领域也需要性能工程。我们每天使用的许多工具，如果不能满足其性能要求，就根本不会存在。例如，集成到 Microsoft Visual Studio IDE 中的 Visual C++ IntelliSense 功能具有非常严格的性能约束。为了使 IntelliSense 自动完成功能正常工作，它们必须在毫秒级别内解析整个源代码库。⁵ 如果源代码编辑器花费几秒钟才能建议自动完成选项，那么没有人会使用它。这样的功能必须非常响应迅速，并在用户输入新代码时提供有效的继续建议。只有通过注重性能并经过深思熟虑的性能工程，才能取得类似应用程序的成功。

有时，快速的工具会在它们最初设计的领域之外找到用途。例如，如今，游戏引擎如 Unreal 和 Unity 在建筑、3D 可视化、电影制作等领域中也得到了应用。由于游戏引擎非常高效，它们是需要 2D 和 3D 渲染、物理引擎、碰撞检测、声音、动画等功能的应用程序的自然选择。

“快速的工具不仅允许用户更快地完成任务，还允许用户以全新的方式完成完全新的任务。” - Nelson Elhage 在他的博客文章中写道（2020 年）⁶。

我希望不用说人们讨厌使用慢软件。应用程序的性能特性可能是用户切换到竞争对手产品的唯一因素。通过强调性能，您可以为您的产品赢得竞争优势。

性能工程是一项重要而有益的工作，但可能非常耗时。事实上，性能优化是一场永无止境的游戏。总会有一些地方需要优化。不可避免地，开发人员会达到边际收益的点，在这一点上，进一步的改进将以非常高的工程成本为代价，并且可能不值得努力。应用程序针对硬件理论极限的性能评估有助于了解优化的潜在空间。知道何时停止优化是性能工作的关键方面。有些组织通过将这些信息集成到代码审查流程中来实现：源代码行用相应的“成本”指标进行注释。使用这些数据，开发人员可以决定是否值得优化代码的特定部分。

在开始性能调优之前，请确保有足够的理由这样做。为了优化而优化是没有价值的，如果它不能为您的产品增加价值。慎重的性能工程始于明确定义的性能目标，说明您试图实现什么以及为什么这样做。此外，您还应该选择用于衡量是否达到目标的指标。您可以在 [Gregg, 2013] 和 [Akinshin, 2019] 中阅读有关设置性能目标的更多信息。

尽管如此，实践和掌握性能分析和调优的技能总是很好的。如果您因此原因拿起了这本书，那么非常欢迎您继续阅读。

1.3 什么是性能分析？

曾经和同事辩论过某段代码的性能吗？那么你可能知道要预测哪段代码会表现最佳有多么困难。在现代处理器内部有如此多的运作部件，即使对代码进行微小的调整也可能触发显著的性能变化。这就是为什么本书的第一个建议是：始终进行测试。许多人在尝试优化他们的应用程序时依赖直觉。而通常情况下，最终会在各处随意修复，而对应用程序的性能没有产生任何真正的影响。

经验不足的开发人员经常在源代码中进行更改，并希望它会改善程序的性能。一个这样的例子是在代码库中到处用 `i++` 替换为 `++i`，假设之前的 `i` 值未被使用。在一般情况下，这种更改对生成的代码不会有影响，因为任何像样的优化编译器都会认识到之前的 `i` 值未被使用，并会消除多余的副本。

许多围绕世界流传的微优化技巧在过去是有效的，但现在的编译器已经学会了它们。此外，一些人倾向于过度使用传统的位操作技巧。其中一个例子是使用 [XOR 交换算法⁷](#)，而事实上，简单的 `std::swap` 会产生更快的代码。这种意外的更改可能不会改善应用程序的性能。找到正确的修复位置应该是经过仔细的性能分析得出的结果，而不是靠直觉和猜测。

⁵ 事实上，不可能在毫秒级的时间内解析整个代码库。相反，IntelliSense 只重构 AST 中已更改的部分。请在视频中观看微软团队如何实现这一目标的更多细节：<https://channel9.msdn.com/Blogs/Seth-Juarez/Anders-Hejlsberg-on-Modern-Compiler-Construction>。

⁶ N. Elhage 的软件性能反思 - <https://blog.nelhage.com/post/reflections-on-performance/>

⁷ xor 交换算法 - https://en.wikipedia.org/wiki/XOR_swap_algorithm

有许多性能分析方法可能会或可能不会带领你发现问题。本书介绍的针对特定 CPU 的性能分析方法有一个共同点：它们都基于收集有关程序执行方式的特定信息。在程序源代码中做出的任何更改都是通过分析和解释收集的数据来驱动的。

找到性能瓶颈只是工程师工作的一半。第二部分是正确地修复它。有时，在程序源代码中更改一行代码可能会带来显著的性能提升。性能分析和调优就是要找到并修复这行代码。错过这样的机会可能会是一种巨大的浪费。

1.4 本书讨论了什么？

本书旨在帮助开发人员更好地理解其应用程序的性能，学会发现效率低下的地方，并加以消除。为什么我的手写压缩算法比传统的慢了两倍？为什么我的函数更改导致性能下降了一半？客户抱怨我的应用程序运行缓慢，我应该从哪里开始？我是否已经充分优化了程序？我的平台上有哪些性能分析工具？减少缓存未命中和分支预测失败的技术有哪些？希望通过本书的学习，你能找到这些问题的答案。

以下是本书的大纲：

- 第二章讨论如何进行公平的性能实验并分析其结果。它介绍了性能测试和比较结果的最佳实践。
- 第三章介绍了CPU微体系结构的基础知识，第四章介绍了性能分析中使用的术语和指标；我们建议即使你认为你已经了解这些内容，也不要跳过这些章节。
- 第五章探讨了几种最流行的性能分析方法。它解释了分析技术的工作原理，可以收集哪些运行时数据以及如何进行收集。
- 第六章审查了现代CPU提供的功能，以支持和增强性能分析。它展示了它们的工作原理以及它们可以解决的问题。
- 第七章概述了主要平台上最受欢迎的工具，包括基于x86和ARM的处理器上运行的Linux、Windows和MacOS。
- 第八至十一章包含了典型性能问题的解决方案。这些章节根据自顶向下微体系结构分析方法进行组织，这是本书最重要的概念之一。如果一些术语对您还不清楚，不要担心，我们将一步一步地覆盖所有内容。第八章（内存绑定）涉及优化内存访问、缓存友好的代码、内存分析、大页和其他一些技术。第九章（核心绑定）涉及优化计算，探讨了函数内联、循环优化和向量化。第十章（坏预测）涉及无分支编程，用于避免频繁的预测失败分支。第十一章（前端绑定）涉及机器代码布局优化，如基本块放置、函数拆分、基于配置文件的优化等。
- 第十三章包含了不特定于前四章所涵盖的任何类别的优化主题，但仍然足够重要，以至于在本书中找到了它们的位置。在这里，你会找到低延迟技术、调整系统以获得最佳性能的技巧、标准库函数的更快替代方法等。
- 第十四章讨论了分析多线程应用程序的技术。它概述了优化多线程应用程序性能的一些最重要挑战以及可用于分析它们的工具。这个主题本身很庞大，所以这一章节只侧重于硬件相关的问题，比如“伪共享”。
- 第十五章讨论了软件和硬件性能领域的当前和未来趋势。我们讨论了传统计算机系统设计的进步以及一些创新的想法。

本书提供的示例主要基于开源软件：Linux作为操作系统，基于LLVM的Clang编译器用于C和C++语言，以及Linux perf作为性能分析工具。选择这些技术的原因不仅是因为它们的流行程度，还因为它们的源代码是开放的，这使我们能够更好地理解它们的工作原理。这对于学习本书中提出的概念特别有用。我们有时也会展示一些专有工具，这些工具在其领域是“大玩家”，例如Intel® VTune™ Profiler。

1.5 本书未涉及的内容

系统性能取决于不同的组件：CPU、操作系统、内存、I/O设备等。应用程序可以通过调整系统的各个组件来获益。一般来说，工程师应该分析整个系统的性能。然而，系统性能的最大因素是其核心，即CPU。这就是为什么本书主要从CPU的角度进行性能分析，偶尔涉及操作系统和内存子系统。

本书的范围不超出单个CPU插槽，因此我们不会讨论分布式、NUMA和异构系统的优化技术。使用诸如OpenCL和

openMP 之类的解决方案将计算外包给加速器（GPU、FPGA 等）在本书中也没有讨论。

本书以 Intel x86-64 CPU 架构为中心，并不提供针对 AMD、ARM 或 RISC-V 芯片的具体调优方案。尽管如此，本书讨论的许多原则也适用于这些处理器。此外，Linux 是本书选择的操作系统，但对于提供的大多数示例来说，这并不重要，因为相同的技术也有益于运行在 Windows 和 macOS 操作系统上的应用程序。

本书中的所有代码片段均以 C、C++ 或 x86 汇编语言编写，但在很大程度上，本书中的思想也适用于其他编译为本机代码的语言，如 Rust、Go，甚至 Fortran。由于本书针对的是靠近硬件运行的用户模式应用程序，因此我们不会讨论托管环境，例如 Java。

最后，作者假设读者对他们开发的软件拥有完全控制权，包括他们使用的库和编译器的选择。因此，本书不涉及调整购买的商业软件包，例如调整 SQL 数据库查询。

1.6 练习

作为本书的补充材料，我们开发了一系列免费的实验任务，可以在<https://github.com/dendibakh/perf-ninja>上获取。性能忍者是一个在线课程，您可以在其中练习低级性能分析和调优。我们在整本书中都提供了该仓库中的实验任务。例如，当您看到perf-ninja::warmup时，这对应于位于上述仓库的labs/misc/warmup文件夹中的实验任务。

您可以在本地计算机上解决这些任务，也可以将您的代码提交到 Github 进行自动化基准测试。如果选择后者，请按照仓库的“入门”页面上的说明操作。如果您遇到问题，可以查看与实验任务相关的视频。这些视频解释了问题的可能解决方案。

章节总结

- CPU 的单线程性能增长速度不如几十年前那样迅猛。这就是为什么性能调优变得比过去 40 年更加重要的原因。计算行业正在变化，现在比 90 年代任何时候都更加剧烈。
- 根据 [Leiserson et al., 2020] 的观点，未来软件调优将成为性能提升的关键驱动因素之一。性能调优的重要性不容小觑。对于大型分布式应用程序，每一点性能改进都会带来巨大的成本节约。
- 软件默认情况下并不具有最佳性能。存在某些限制，阻止应用程序发挥其全部性能潜力。HW 和 SW 环境都有这样的限制。CPU 不能神奇地加速缓慢的算法。编译器远不能为每个程序生成最佳代码。由于硬件的特定性，某个特定问题的最佳算法并不总是性能最佳的。所有这些都为我们调优应用程序的性能留下了空间。
- 对于某些类型的应用程序，性能不仅仅是一个特性。它使用户能够以新的方式解决新的问题。
- 软件优化应该有强有力的业务需求支持。开发人员应该设定可量化的目标和度量标准，这些目标和度量标准必须用于衡量进展。
- 预测某段代码的性能几乎是不可能的，因为现代平台有很多因素影响性能。在实施软件优化时，开发人员不应依赖直觉，而是应该使用谨慎的性能分析。

第一部分：在现代 CPU 上的性能分析

2 性能测量

了解应用程序性能的第一步是知道如何进行测量。人们将性能视为应用程序的特性之一。⁸ 但与其他特性不同，性能不是一个布尔属性：一个应用程序可以非常慢，飞快，或者介于两者之间。这就是为什么对于是否具有性能的问题，不可能回答“是”或“否”的原因。

性能问题通常比大多数功能问题更难追踪和复现。尽管有时我们不得不处理非确定性的，难以复现的性能错误。通常，程序的每次运行在功能上都是相同的，但从性能的角度来看有些许差异。例如，解压缩 zip 文件时，我们一遍又一遍地获得相同的结果，这意味着这个操作是可重现的。然而，要精确复现此操作的每个 CPU 周期的性能特征是不可能的。

任何关心性能评估的人都知道有时进行公正的性能测量并从中得出准确的结论是多么困难。性能测量可能非常出乎意料和反直觉。更改源代码中看似无关的部分可能会让我们惊讶地发现对程序性能有重大影响。这种现象被称为测量偏差。由于测量中存在误差，性能分析需要使用统计方法来处理它们。这个主题值得一整本书来单独讨论。在这里，我们只关注高层次的思想和需要遵循的方向。

进行公平的性能实验是获得准确而有意义的结果的关键步骤。设计性能测试和配置环境都是评估性能过程中的重要组成部分。本章将简要介绍现代系统为何产生嘈杂的性能测量以及如何处理这种情况。我们将涉及在实际生产部署中测量性能的重要性。

没有一个长寿命的产品是不会发生性能退化的。这对于有许多贡献者的大型项目尤为重要，因为变化发生得非常迅速。本章将花几页讨论在持续集成和持续交付（CI/CD）系统中跟踪性能变化的自动化过程。我们还提供有关开发人员在源代码库中实施更改时如何正确收集和分析性能测量的一般指导。

本章的结尾描述了开发人员在基于时间的测量中可以使用的软件和硬件定时器，以及在设计和编写良好的微基准时可能遇到的常见问题。

2.1 现代系统中的噪声

硬件和软件中有许多旨在提高性能的特性。但并非所有特性都具有确定性行为。让我们考虑动态频率调整⁹（DFS）：这是一种允许 CPU 在短时间内增加频率的特性，从而使其运行速度显著提高。然而，CPU 不能长时间保持在“超频”模式下，因此稍后会将频率降回基础值。DFS 通常在很大程度上取决于核心温度，这使得很难预测对我们实验的影响。

如果我们在“冷”处理器上连续启动两次基准测试，¹⁰第一次运行可能会在一段时间内以“超频”模式运行，然后将频率降回基准水平。然而，第二次运行可能没有这种优势，将以基准频率运行而不进入“Turbo 模式”。尽管我们两次运行了完全相同版本的程序，但它们运行的环境并不相同。图 @fig:FreqScaling 显示了动态频率调整可能导致测量结果的差异。正如你所看到的，由于第一次运行在更高的频率下运行，因此比第二次运行快 1 秒。在笔记本电脑上进行基准测试时，这样的情况经常发生，因为它们的散热能力有限。

频率调整是一种硬件特性，但测量中的变化也可能来自软件特性。让我们考虑一个文件系统缓存的例子。如果我们

⁸ Nelson Elhage 的博客文章“Reflections on software performance”: <https://blog.nelhage.com/post/reflections-on-performance/>。

⁹ 动态频率缩放 - https://en.wikipedia.org/wiki/Dynamic_frequency_scaling。

¹⁰ 冷处理器是指处于空闲模式一段时间，使其冷却下来的 CPU。

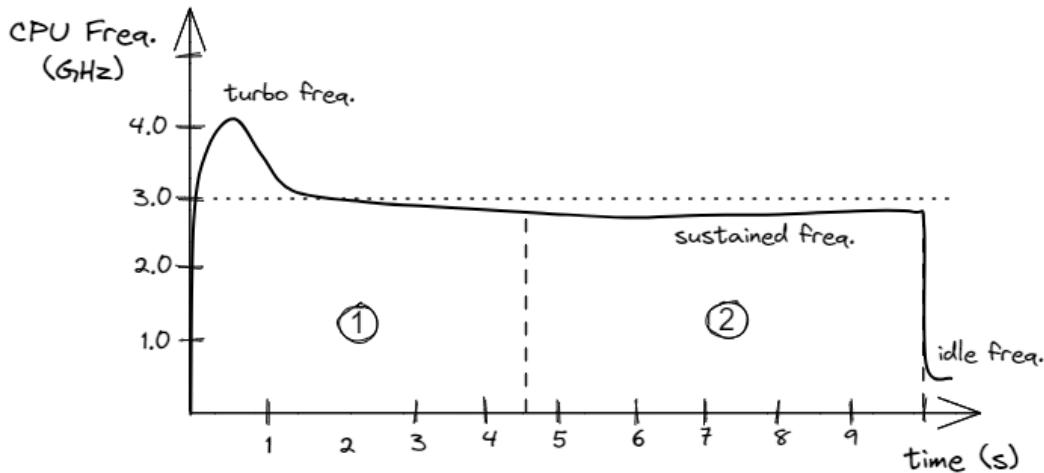


Figure 2: 频率缩放导致性能变化的差异：第一次运行比第二次运行快 1 秒。

对进行大量文件操作的应用程序进行基准测试，例如 `git status` 命令，文件系统可能在性能方面起到重要作用。当运行基准测试的第一次迭代时，文件系统缓存中可能缺少所需的条目。然而，当第二次运行相同的基准测试时，文件系统缓存将被热身，使其比第一次运行明显更快。

不幸的是，测量偏差不仅来自环境配置。[\[Mytkowicz et al., 2009\]](#) 的论文表明，UNIX 环境大小（即存储环境变量所需的总字节数）和链接顺序（给链接器提供的目标文件的顺序）可能以不可预测的方式影响性能。此外，还有许多其他影响内存布局并可能影响性能测量的方法。[\[Curtsinger & Berger, 2013\]](#) 提出了一种在现代架构上实现统计上可靠的方法。这项工作表明，通过在运行时高效且重复地随机化代码、堆栈和堆对象的放置，可以消除来自内存布局的测量偏差。不幸的是，这些想法并没有得到很大发展，现在这个项目几乎被放弃了。

请记住，即使运行类似 Linux `top` 这样的任务管理工具也会影响测量，因为 CPU 核心会被激活并分配给它。这可能会影响实际基准测试运行的核心频率。

要获得一致的测量结果，需要以相同的条件运行所有基准测试的所有迭代。然而，要复制完全相同的环境并完全消除偏差是不可能的：可能会存在不同的温度条件、电力传递峰值、运行的邻近进程等。追踪系统中所有潜在的噪音和变化可能是一个永无止境的故事。有时无法实现，例如在对大型分布式云服务进行基准测试时。

为了确保性能测试（例如微基准测试）具有明确且稳定的结果，消除系统中的非确定性行为是有益的。例如，当您实现代码更改并希望通过相同程序的两个不同版本进行基准测试来了解相对加速比时，您可以控制基准测试的大多数变量，包括其输入、环境配置等。在这种情况下，消除系统中的非确定性有助于获得更加一致和准确的比较。完成本地测试后，请记住验证投影性能改进是否反映在实际测量中。读者可以在附录 A 中找到一些示例，这些示例说明了哪些功能会使性能测量产生噪音以及如何禁用它们。此外，还有一些工具可以设置环境以确保基准测试结果具有低方差，其中一个这样的工具是 `temci`¹¹。

在评估实际性能提升时，不建议消除系统非确定性行为。工程师应该尝试复制他们正在优化的目标系统配置。对测试系统进行任何人工调整都会使结果与您的服务用户在实际中看到的结果相背离。此外，任何性能分析工作，包括分析（参见 Section 5.5），都应在与实际部署中类似的系统上进行。

最后，重要的是要记住，即使特定硬件或软件功能具有非确定性行为，也并不意味着它被认为是有害的。它可能会产生不一致的结果，但它旨在提高系统的整体性能。禁用此类功能可能会减少微基准测试中的噪音，但会使整个套件运行更长时间。这对于 CI/CD 性能测试尤其重要，因为整个基准测试套件的运行时间有限制。

¹¹ Temci - <https://github.com/parttimenerd/temci>。

2.2 在生产环境中进行性能测量

当一个应用程序在共享基础设施上运行时（在公共云中很常见），通常会有来自其他客户的其他工作负载在同一台服务器上运行。随着虚拟化和容器等技术变得越来越流行，公共云提供商试图充分利用他们服务器的容量。不幸的是，这为在这样的环境中进行性能测量带来了额外的障碍。与相邻进程共享资源可能以不可预测的方式影响性能测量。

通过在实验室中重现生产工作负载来分析生产工作负载可能会很棘手。有时，对于“内部”性能测试，可能无法合成精确的行为。这就是为什么越来越多的云提供商和超大规模云服务提供商选择直接在生产系统上进行性能分析和监控的原因。在“没有其他参与者”的情况下进行性能测量可能不反映真实的场景。在实验室环境中执行性能优化代码可能会得到良好的性能表现，但在生产环境中可能不尽如人意。话虽如此，这并不意味着不需要持续的“内部”测试来及早捕捉性能问题。并非所有性能回归都可以在实验室中捕捉到，但工程师应设计具有代表性的真场景性能基准测试。

大型服务提供商实施监控用户设备上性能的遥测系统正在成为一种趋势。Netflix Icarus¹²遥测服务就是一个例子，它在全球范围内的数千台不同设备上运行。这种遥测系统帮助 Netflix 了解真实用户如何感知 Netflix 应用的性能。它允许工程师分析从许多设备收集的数据，并找到否则无法发现的问题。这种数据可以帮助更明智地决定在哪里集中优化工作。

监控生产部署的一个重要注意事项是测量开销。因为任何类型的监控都会影响正在运行的服务的性能，因此建议仅使用轻量级的分析方法。根据 [Ren et al., 2010] 的说法：“对于提供实时流量的数据中心机器进行持续性能分析，极低的开销至关重要”。通常，可接受的聚合开销被认为低于 1%。性能监控开销可以通过限制受监控机器的集合以及使用更长的时间间隔来减少。

在这样的生产环境中进行性能测量意味着我们必须接受其嘈杂的特性，并使用统计方法来分析结果。[Liu et al., 2019] 提供了一个好例子，说明像 LinkedIn 这样的大公司如何在生产环境中使用统计方法来测量和比较基于分位数的指标（例如 90 分位数的页面加载时间）在其 A/B 测试中的表现。

2.3 自动检测性能回归

软件供应商试图增加部署频率正在成为一种趋势。公司不断寻求加速产品上市的方式。不幸的是，这并不意味着每次新发布的软件产品都会变得更好。特别是，软件性能缺陷往往以惊人的速度泄漏到生产软件中 [Jin et al., 2012]。在软件的演变过程中，大量的变化给分析所有这些结果和历史数据以检测性能回归带来了挑战。

软件性能回归是在软件从一个版本发展到另一个版本时错误地引入的缺陷。捕捉性能错误和改进意味着检测哪些提交改变了软件的性能（由性能测试测量），在测试基础设施的噪声存在的情况下。从数据库系统到搜索引擎再到编译器，几乎所有大规模软件系统在其持续演进和部署生命周期中都会经历性能回归。在软件开发过程中完全避免性能回归可能是不可能的，但通过适当的测试和诊断工具，可以显着减少这些缺陷悄然泄漏到生产代码中的可能性。

首先考虑的选择是：让人类查看图表并比较结果。毫不奇怪，我们很快就要摆脱这个选择。人们往往很快失去注意力，可能会错过回归，特别是在嘈杂的图表上，例如图 @fig:PerfRegress 所示的图表。人类可能会捕捉到发生在 8 月 5 日左右的性能回归，但人们可能不会发现后续的回归。除了容易出错外，让人类参与其中还是一项耗时且枯燥的工作，必须每天进行。

第二个选择是设置一个阈值，例如 2%：认为性能在阈值内的每个代码修改都是噪音，而超过阈值的则被认为是回归。这比第一个选择要好一些，但仍然有其自身的缺点。性能测试中的波动是不可避免的：有时，甚至一个无害的代码更改¹³也可能触发基准测试中的性能变化。选择正确的阈值极其困难，也不能保证低误报率和低漏报率。将阈值设置得太低可能会导致分析一堆并非由源代码更改引起而是由某些随机噪音引起的小型回归。将阈值设置得太高可能会过滤掉真正的性能回归。小的变化可能会逐渐积累成较大的回归，这可能会被忽视。例如，假设您设置了 2%

¹² 在 CMG 2019 年展示，https://www.youtube.com/watch?v=4RG2DUK03_0。

¹³ 下面的文章表明，改变函数的顺序或删除死函数可能会导致性能变化：https://easypf.net/blog/2018/01/18/Code_alignment_issues

的阈值。如果有两次连续的 1.5% 的回归，它们都将被过滤掉。但是在两天内，性能回归将累积达到 3%，这超过了阈值。通过观察图 @fig:PerfRegress，我们可以得出一个观察结果，即阈值需要进行每个测试的调整。对于绿色（上线）测试有效的阈值未必对于紫色（下线）测试同样有效，因为它们具有不同级别的噪声。每个测试都需要设置明确的阈值值以警报回归的 CI 系统示例是 LUCI¹⁴，它是 Chromium 项目的一部分。

第三个选择是使用统计分析来识别性能回归。一个简单的例子是使用学生 t 检验¹⁵来比较程序 A 的 100 次运行的算术平均值与程序 B 的 100 次运行的算术平均值。然而，这样的参数化测试假设了正态（即高斯）样本分布，而通常情况下系统性能运行时直方图通常是右偏、多峰的，因此在这些情况下误用统计工具可能会产生误导性的结果。幸运的是，对于非正态分布存在更合适的统计工具，称为“非参数”测试，其示例包括曼-惠特尼、安德森-达林和科尔莫戈洛夫-斯米尔诺夫（下一节将详细介绍）。对于那些希望自己搭建自动化性能回归测试框架的人，Python 和 R 提供了这些可下载的软件包，而像 stats-pal¹⁶这样的开源项目提供了现成的框架，可插入现有的 CI/CD 流水线中。

在 [Daly et al., 2020] 中采用了一种更复杂的统计方法来识别性能回归。MongoDB 开发人员实施了变点分析，以识别其数据库产品演变代码库中的性能变化。根据 [Matteson & James, 2014]，变点分析是在时间顺序观察中检测分布变化的过程。MongoDB 开发人员利用了一种称为“E-Div

isive means”的算法，该算法通过分层选择将时间序列划分为集群的分布变化点。他们的开源 CI 系统称为 Evergreen¹⁷，并将其纳入其中以在图表上显示变点并打开 Jira 票据。有关此自动化性能测试系统的更多详细信息可以在 [Ingo & Daly, 2020] 中找到。

[Alam et al., 2019] 中提出了另一种有趣的方法。本文作者提出了 AutoPerf，它使用硬件性能计数器（PMC，见 Section 3.9.1）来诊断修改程序中的性能回归。首先，它根据从原始程序收集的 PMC 配置文件数据来学习修改函数性能的分布。然后，它根据从修改后程序收集的 PMC 配置文件数据将性能的偏差检测为异常值。AutoPerf 表明，这种设计可以有效地诊断一些最复杂的软件性能错误，例如隐藏在并行程序中的错误。

无论性能回归检测的底层算法如何，一个典型的 CI 系统应该自动执行以下操作：

1. 设置测试系统。
2. 运行基准测试套件。
3. 报告结果。
4. 确定性能是否发生了变化。
5. 对性能的意外变化发出警报。
6. 为人类分析结果可视化。

CI 系统应支持自动化和手动基准测试，产生可重复的结果，并为发现的性能回归创建工单。及时检测回归非常重要。首先，因为自回归发生以来合并的变更较少。这使我们可以指定负责调查回归的人员在他们转移到其他任务之前解决问题。此外，对于开发人员来说，解决回归问题要容易得多，因为所有细节仍然新鲜在他们的脑海中，而不是在几周之后。

最后，CI 系统不应仅仅在软件性能回归上发出警报，还应在性能改进方面发出意外警报。例如，有人可能提交了一个看似无害的提交，然而，在自动性能回归测试中，它将延迟减少了惊人的 10%。您最初的直觉可能是庆祝这次幸运的性能提升并继续您的一天。然而，尽管此提交可能已通过 CI 流水线中的所有功能测试，但很有可能这个意外的延迟改进揭示了功能测试中的一个漏洞，这个漏洞只在性能回归结果中表现出来。这种情况经常发生，足以明确提及：将自动性能回归测试工具视为整体软件测试框架的一部分，而不是一个孤立的部分。

本书的作者强烈建议建立一个自动化的统计性能跟踪系统。尝试使用不同的算法，看看哪种对您的应用程序效果最好。这当然需要时间，但这将是项目未来性能健康的坚实投资。

¹⁴ LUCI - https://chromium.googlesource.com/chromium/src.git/+master/docs/tour_of_luci_ui.md

¹⁵ Student's t-test - https://en.wikipedia.org/wiki/Student%27s_t-test

¹⁶ Stats-pal - <https://github.com/JoeyHendricks/STATS-PAL>

¹⁷ Evergreen - <https://github.com/evergreen-ci/evergreen>

2.4 手动性能测试

当工程师能够在开发过程中利用现有的性能测试基础设施时，这是很棒的。在前一节中，我们讨论了 CI 系统的一个很好的功能是可以向其提交性能评估作业的可能性。如果支持这一点，那么系统将返回测试开发人员想要提交到代码库的补丁的结果。由于各种原因，这可能并不总是可能的，比如硬件不可用、设置对于测试基础设施来说过于复杂、需要收集额外的指标。在本节中，我们提供了进行本地性能评估的基本建议。

当我们在代码中进行性能改进时，我们需要一种方法来证明我们确实取得了改进。此外，当我们提交常规代码更改时，我们希望确保性能没有退化。通常，我们通过以下方式来做到这一点：1) 测量基准性能，2) 测量修改后程序的性能，3) 将它们进行比较。在这种情况下，我们的目标是比较同一个功能程序的两个不同版本的性能。例如，我们有一个以递归方式计算斐波那契数的程序，我们决定以迭代的方式重写它。两者在功能上是正确的，并且产生相同的结果。现在我们需要比较两个程序的性能。

强烈建议不仅获取单个测量值，而是多次运行基准测试。因此，我们有 N 次基准的测量结果和 N 次修改版本的程序的测量结果。现在我们需要一种方法来比较这两组测量结果，以确定哪个更快。这个任务本身是难以解决的，有很多方法可以被测量结果所欺骗，并且可能从中得出错误的结论。如果你问任何数据科学家，他们都会告诉你，你不应该依赖于单一的度量指标（最小值/平均值/中值等）。

考虑图 @fig:CompDist 中收集的两个程序版本的性能测量的分布。这个图显示了对于给定版本的程序，我们得到特定时间的概率。例如，版本A完成在大约 102 秒的概率约为 32%。诱人的是说A比B更快。然而，这只有一定的概率P才是真的。这是因为有一些B的测量比A快。即使在所有B的测量都比每个A的测量都慢的情况下，概率P也不等于100%。这是因为我们总是可以为B生成一个额外的样本，这个样本可能比一些A的样本更快。

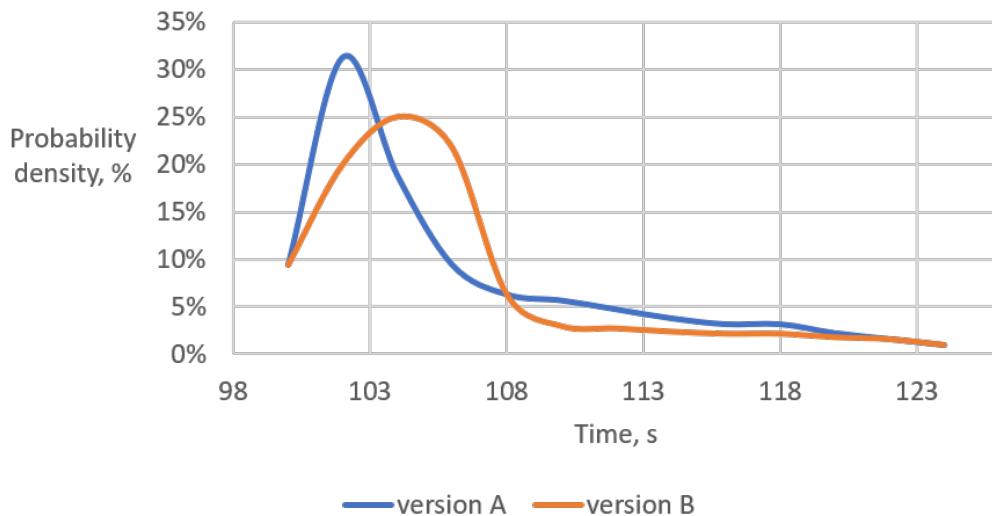


Figure 3: Comparing 2 performance measurement distributions.

使用分布图的一个有趣优势是它允许您发现基准测试的不良行为。¹⁸如果分布是双峰的，那么基准测试很可能经历了两种不同类型的行为。双峰分布测量的常见原因是代码具有快速路径和慢速路径，比如访问缓存（缓存命中 vs 缓存未命中）和获取锁（争用锁 vs 非争用锁）。要“修复”这个问题，应该将不同的功能模式分离出来并单独进行基准测试。

数据科学家通常通过绘制分布图来呈现测量结果，并避免计算加速比。这样做可以消除偏见的结论，并允许读者自行解释数据。绘制分布的一种流行方式是使用箱线图（参见图 @fig:BoxPlot），它允许在同一图表上比较多个分布。

¹⁸ 另一种检查方法是运行正态性检验: https://en.wikipedia.org/wiki/Normality_test.

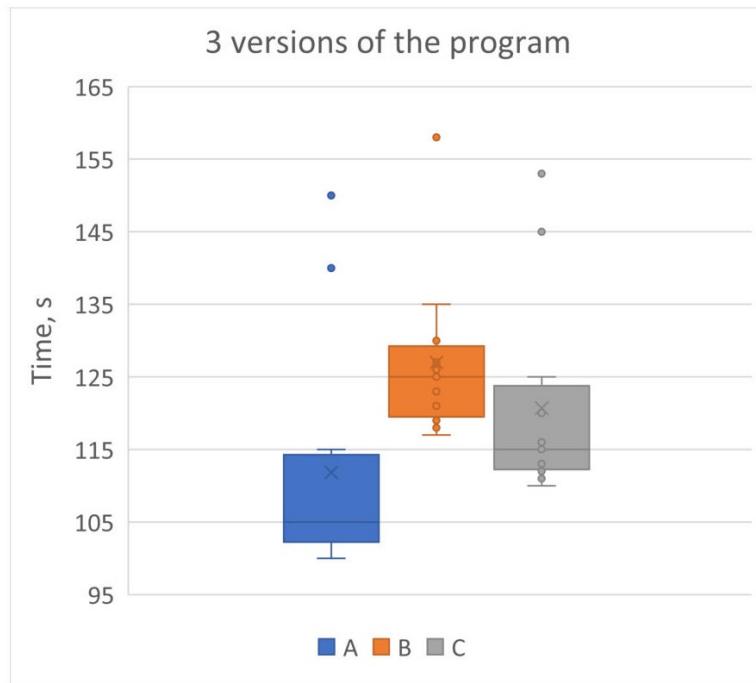


Figure 4: Box plots.

虽然可视化性能分布可能有助于发现某些异常，但开发人员不应该将它们用于计算加速比。一般来说，通过查看性能测量分布来估算加速比是困难的。此外，正如前一节讨论的那样，它在自动基准测试系统中不起作用。通常，我们希望获得一个标量值，该值代表两个程序版本的性能分布之间的加速比，例如，“版本A比版本B快X%”。

2.5 假设检验方法

假设检验方法是确定两个分布之间的统计关系的一种方法。如果根据阈值概率（显著水平），数据集之间的关系将拒绝零假设¹⁹，则将比较被认为是统计显著的。

- 如果分布是高斯分布（正态分布），那么使用参数假设检验（例如，学生T检验）来比较分布就足够了。尽管值得一提的是，性能数据中很少见到高斯分布。因此，在使用假设为高斯分布的公式时要谨慎。
- 如果要比较的分布不是高斯分布（例如，严重偏斜或多峰），那么可以使用非参数检验（例如，曼-惠特尼U检验²⁰、克鲁斯卡尔-沃利斯单因素方差分析²¹等）。

假设检验方法非常适用于确定速度提升（或减慢）是否是随机的。因此，最好在自动化测试框架中使用它来验证提交是否引入了性能回归。关于性能工程统计学的一个很好的参考资料是 Dror G. Feitelson 的书《计算机系统性能评估的工作负载建模》²²，该书有关于模态分布、偏度和其他相关主题的更多信息。

一旦通过假设检验确定了差异是统计显著的，那么速度提升可以计算为平均值或几何平均值之间的比率，但有一些注意事项。在少量样本中，均值和几何平均值可能会受到异常值的影响。除非分布的方差很小，否则不要仅考虑平均值。如果测量值的方差与均值相同数量级，那么平均值就不是一个代表性的指标。图 @fig:Averages 展示了程序的两个版本的示例。仅查看平均值（5），很容易认为版本A比版本B快 20%。然而，考虑到测量的方差（6），我们会发现情况并非总是如此。如果我们取版本A的最差分数和版本B的最佳分数，我们可以说版本B比版本A快 20%。对于正

¹⁹ Null hypothesis - https://en.wikipedia.org/wiki/Null_hypothesis.

²⁰ Mann-Whitney U test - https://en.wikipedia.org/wiki/Mann-Whitney_U_test.

²¹ Kruskal-Wallis analysis of variance - https://en.wikipedia.org/wiki/Kruskal-Wallis_one-way_analysis_of_variance.

²² 《计算机系统性能评估的工作量建模》一书 - <https://www.cs.huji.ac.il/~feit/wlmod/>.

态分布，可以使用均值、标准差和标准误差的组合来衡量程序两个版本之间的速度提升。否则，对于偏斜或多峰样本，必须使用更适合基准测试的百分位数，例如，最小值、中位数、90th、95th、99th、最大值或这些的某种组合。

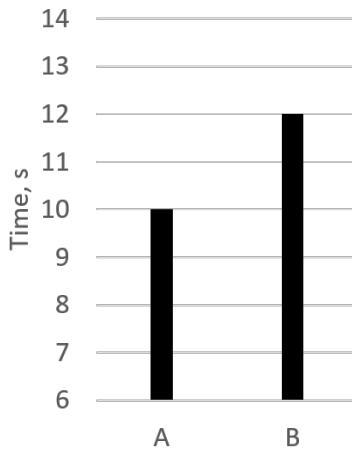


Figure 5: 仅平均值

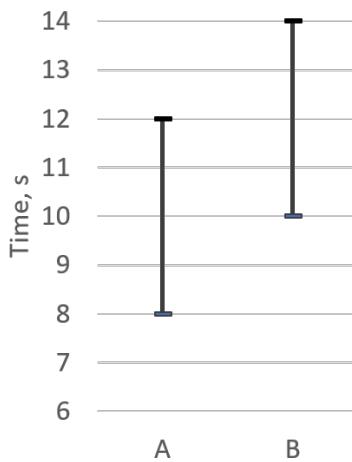


Figure 6: 完整测量间隔

Figure 7: 显示平均值可能具有误导性的两个直方图。

计算准确速度提升比率的一个最重要因素是收集丰富的样本集，即多次运行基准测试。这听起来可能很明显，但并不总是可行的。例如，一些SPEC CPU 2017 基准测试²³在现代机器上运行超过 10 分钟。这意味着仅生成三个样本就需要 1 小时：每个程序版本 30 分钟。想象一下，如果你的测试套件中不只是一个基准测试，而是有数百个。即使将工作分配到多台

机器上，收集足够统计数据也会变得非常昂贵。

如何知道需要多少样本才能达到统计上足够的分布？对这个问题的答案取决于你希望你的比较有多精确。样本之间的方差越低，你需要的样本数量就越少。标准差是一个告诉你分布中测量值的一致性的度量。可以通过动态限制基于标准差的基准迭代次数来实现自适应策略，也就是说，收集样本直到获得处于某个范围内的标准差。这种方法要求测量次数大于一次。否则，算法将在第一次运行基准测试之后停止，因为基准测试的单次运行具有`std.dev.`等于零。一旦标准差低于阈值，就可以停止收集测量。关于这种策略的更多细节可以在 [Akinshin, 2019, 第 4 章] 中找到。

²³ SPEC CPU 2017 benchmarks - <http://spec.org/cpu2017/Docs/overview.html#benchmarks>

另一个需要注意的重要事项是是否存在异常值。使用置信区间将一些样本（例如，冷启动）丢弃为异常值是可以接受的，但不要故意从测量集中丢弃不需要的样本。对于某些类型的基准测试，异常值可能是最重要的指标之一。例如，在对具有实时约束的软件进行基准测试时，99 百分位数可能非常有趣。Gil Tene 在YouTube上有一系列关于测量延迟的讲座，涵盖了这个话题。

2.6 软件和硬件定时器

为了对执行时间进行基准测试，工程师通常会使用现代平台提供的两种不同的定时器：

- 系统范围的高分辨率定时器：这是一个系统定时器，通常实现为自从一个任意的起始日期（称为纪元）²⁴以来经过的时钟周期数。这个时钟是单调的；即它总是递增的。可以通过系统调用从操作系统中获取系统时间。²⁵在 Linux 系统上，可以通过 `clock_gettime` 系统调用来访问系统定时器。系统定时器具有纳秒级分辨率，在所有 CPU 之间保持一致，并且与 CPU 频率无关。虽然系统定时器可以返回纳秒精度的时间戳，但由于通过 `clock_gettime` 系统调用获取时间戳需要很长时间，因此不适合测量短时间内发生的事件。但是对于持续时间超过一微秒的事件来说，这是可以接受的。在 C++ 中访问系统定时器的 de facto 标准是使用 `std::chrono`，如 Listing 25 所示。

代码清单：使用 C++ `std::chrono` 访问系统定时器

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

// 返回经过的纳秒数
uint64_t timeWithChrono() {
    using namespace std::chrono;
    auto start = steady_clock::now();
    // 运行一些代码
    auto end = steady_clock::now();
    uint64_t delta = duration_cast<nanoseconds>
        (end - start).count();
    return delta;
}
```

- 时间戳计数器 (TSC)：这是一种硬件定时器，实现为硬件寄存器。TSC 是单调的，并且具有恒定的速率，即不考虑频率变化。每个 CPU 都有自己的 TSC，它只是已经过去的参考周期数（参见 Section 4.6）。适用于持续时间从纳秒到一分钟的短事件的测量。可以使用编译器内置函数 `__rdtsc` 来获取 TSC 的值，如 Listing 25 所示，该函数在底层使用 RDTSC 汇编指令。有关使用 RDTSC 汇编指令对代码进行基准测试的更低级别的详细信息，可以参考白皮书 [Paoloni, 2010]。

代码清单：使用 `__rdtsc` 编译器内置函数访问 TSC

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

// 返回经过的参考时钟数
uint64_t timeWithTSC() {
    uint64_t start = __rdtsc();
```

²⁴ Unix 纪元从 1970 年 1 月 1 日 00:00:00 UT 开始：https://en.wikipedia.org/wiki/Unix_epoch。

²⁵ 检索系统时间 - https://en.wikipedia.org/wiki/System_time#Retrieving_system_time

```
// 运行一些代码
return __rdtsc() - start;
}
```

选择使用哪种定时器非常简单，取决于您要测量的事物持续的时间有多长。如果您要测量的时间很短，TSC 会给出更高的精度。相反，使用 TSC 来测量运行数小时的程序是没有意义的。除非您确实需要周期精度，否则系统定时器应该能够用于大多数情况。要记住的重要一点是，访问系统定时器通常的延迟比访问 TSC 要高。进行 `clock_gettime` 系统调用可能比执行 RDTSC 指令慢十倍以上，后者需要 20 多个 CPU 周期。这在尤其需要最小化测量开销时可能变得很重要，尤其是在生产环境中。有关在各种平台上访问定时器的不同 API 的性能比较可在 CppPerformanceBenchmarks 存储库的[wiki 页面²⁶](#)上找到。

2.7 微基准测试

微基准测试是人们编写的小型独立程序，用于快速测试假设。通常，微基准测试用于选择某个相对较小算法或功能的最佳实现。几乎所有现代语言都有基准测试框架。在 C++ 中，可以使用 Google 的[benchmark²⁷](#) 库，C# 有 [BenchmarkDotNet²⁸](#) 库，Julia 有 [BenchmarkTools²⁹](#) 包，Java 有 [JMH³⁰](#) (Java Microbenchmark Harness)，等等。

在编写微基准测试时，非常重要的一点是确保您要测试的场景在运行时确实被微基准测试执行。优化编译器可能会消除可能使实验无效的重要代码，甚至更糟的是，导致您得出错误的结论。在下面的示例中，现代编译器很可能会消除整个循环：

```
// foo 不会对字符串创建进行基准测试
void foo() {
    for (int i = 0; i < 1000; i++)
        std::string s("hi");
}
```

测试这一点的一种简单方法是检查基准测试的性能概要，并查看预期的代码是否突出显示为热点。有时可以立即发现异常的计时情况，因此在分析和比较基准测试运行时，请务必运用常识。防止编译器消除重要代码的一种流行方法是使用类似 [DoNotOptimize³¹](#) 的辅助函数，其在底层执行必要的内联汇编操作：

```
// foo 对字符串创建进行基准测试
void foo() {
    for (int i = 0; i < 1000; i++) {
        std::string s("hi");
        DoNotOptimize(s);
    }
}
```

如果编写得当，微基准测试可以成为性能数据的良好来源。它们经常用于比较关键函数的不同实现的性能。一个好的基准测试的定义在于它是否在功能实际使用的真实条件下测试性能。如果基准测试使用与实践中给定的输入不同的合成输入，则该基准测试很可能会误导您，并导致您得出错误的结论。此外，当基准测试在没有其他要求的进程的系统上运行时，它具有所有可用的资源，包括 DRAM 和缓存空间。这样的基准测试可能会成为更快版本功能的冠

²⁶ CppPerformanceBenchmarks wiki - <https://gitlab.com/chriscox/CppPerformanceBenchmarks/-/wikis/ClockTimeAnalysis>

²⁷ Google benchmark 库 - <https://github.com/google/benchmark>

²⁸ BenchmarkDotNet - <https://github.com/dotnet/BenchmarkDotNet>

²⁹ Julia BenchmarkTools - <https://github.com/JuliaCI/BenchmarkTools.jl>

³⁰ Java Microbenchmark Harness - <http://openjdk.java.net/projects/code-tools/jmh/etc>

³¹ 对于 JMH，这被称为 `Blackhole.consume()`。

军，即使它消耗的内存比其他版本多。但是，如果有占用 DRAM 的邻近进程，导致基准测试进程的内存区域被换出到磁盘，结果可能相反。

出于同样的原因，当从对函数进行单元测试中得出结果时，请谨慎。现代单元测试框架，例如 GoogleTest，提供每个测试的持续时间。但是，这些信息不能替代一个精心编写的基准测试，该测试使用真实的输入在实际条件下测试函数（有关更多信息，请参见 [Fog, 2004, 第 16.2 章]）。在实践中无法复制确切的输入和环境，但这是开发人员在编写良好基准测试时应考虑的事项。

问题和练习

1. 对一系列测量结果取平均时间安全吗？
2. 假设你已经发现了一个性能漏洞，你现在试图在你的开发环境中修复它。你将如何减少系统中的噪音，以获得更明显的基准测试结果？
3. 使用函数级别的单元测试跟踪程序的整体性能可以吗？
4. 您的组织是否有性能回归系统？如果有，可以改进吗？如果没有，请考虑安装一个的策略。请考虑以下因素：什么是变化的，什么是没有变化的（源代码、编译器、硬件配置等），更改发生的频率是多少，测量方差是多少，基准测试的运行时间是多少，您可以运行多少次迭代？

章节总结

- 调试性能问题通常比调试功能性 bug 更困难，这是因为测量的不稳定性。
- 除非设定特定的目标，否则不能停止优化。要知道是否达到了预期的目标，您需要制定有意义的定义和衡量标准。根据您关心的内容，可以是吞吐量、延迟、每秒操作数（性能屋顶）等。
- 现代系统具有非确定性性能。消除系统中的非确定性对于定义明确、稳定的性能测试（例如微基准测试）非常有帮助。在生产部署中测量性能需要使用统计方法来分析结果，以应对嘈杂的环境。
- 越来越多地，大型分布式软件的供应商选择在生产系统上直接对性能进行配置和监控，这需要使用轻量级的配置技术。
- 为了防止性能回归泄漏到生产软件中，采用自动化的统计性能跟踪系统非常有益。这种持续集成系统应该运行自动化性能测试，可视化结果，并在发现性能异常时发出警报。
- 可视化性能分布还可以帮助发现性能异常。这是向广大受众展示性能结果的安全方式。
- 使用假设检验方法确定性能分布之间的统计关系。一旦确定差异在统计上是显著的，就可以计算加速比，即均值或几何均值之比。
- 可以放弃冷运行以确保一切运行良好，但不要故意放弃不要的数据。如果决定放弃某些样本，应对所有分布均匀地进行。
- 为了测量执行时间，工程师可以使用现代平台提供的两种不同的定时器。系统范围内的高分辨率定时器适用于测量持续时间超过一微妙的事件。要以高精度测量短期事件，使用时间戳计数器。
- 微基准测试对于快速证明某些内容非常有用，但您应该始终在实际条件下的真实应用程序上验证自己的想法。确保您正在对有意义的代码进行基准测试，方法是检查性能分析。

3 CPU 微体系结构

本章简要概述了对软件性能产生直接影响的关键 CPU 微体系结构特性。本章的目标不是涵盖 CPU 架构的所有细节和权衡，这些内容已在文献中广泛讨论 [Hennessy & Patterson, 2017]。相反，本章快速回顾了现代处理器中存在的 CPU 硬件特性。

3.1 指令集架构

指令集是软件用来与硬件通信的词汇。指令集架构（ISA）定义了软件与硬件之间的约定。英特尔 x86、ARM v8 和 RISC-V 是当前广泛部署的 ISA 的示例。所有这些都是 64 位架构，即所有地址计算都使用 64 位。ISA 开发人员和 CPU 架构师通常确保符合规范的软件或固件将在使用规范构建的任何处理器上执行。广泛部署的 ISA 特许经营权通常还确保向后兼容性，使为处理器的 GenX 版本编写的代码将继续在 GenX+i 上执行。

大多数现代架构可以归类为通用寄存器为基础的，采用加载-存储架构，操作数明确指定，仅使用加载和存储指令访问内存。除了在 ISA 中提供基本功能（如加载、存储、控制和使用整数和浮点数进行标量算术运算）之外，广泛部署的架构继续增强其 ISA 以支持新的计算范式。这些包括增强的向量处理指令（例如，英特尔 AVX2、AVX512、ARM SVE）和矩阵/张量指令（英特尔 AMX）。映射到使用这些高级指令的软件通常在性能上提供数量级的改进。

现代 CPU 支持 32 位和 64 位精度的算术运算。随着机器学习和人工智能领域的快速发展，行业对于用于驱动显著性能改进的变量的替代数值格式重新产生了兴趣。研究表明，使用较少位数表示变量，既可以保持机器学习模型的性能，又可以节省计算和内存带宽。因此，几个 CPU 特许经营权最近已经在 ISA 中添加了对较低精度数据类型的支持，例如 8 位整数 (int8，例如，英特尔 VNNI)、16 位浮点数 (fp16, bf16)，除了传统的 32 位和 64 位格式进行算术运算。

3.2 流水线技术

流水线是使 CPU 快速运行的基础技术，其中多个指令在它们的执行过程中重叠。CPU 中的流水线技术从汽车组装线中汲取灵感。指令的处理被分为多个阶段。这些阶段并行运行，同时处理不同指令的不同部分。DLX 是由 John L. Hennessy 和 David A. Patterson 于 1994 年设计的一个相对简单的架构。正如 [Hennessy & Patterson, 2017] 中定义的那样，它具有 5 级流水线，包括：

1. 指令获取 (IF)
2. 指令解码 (ID)
3. 执行 (EXE)
4. 存储访问 (MEM)
5. 写回 (WB)

图 @fig:Pipelining 显示了 5 级流水线 CPU 的理想流水线视图。在周期 1 中，指令 x 进入流水线的 IF 阶段。在下一个周期中，随着指令 x 移动到 ID 阶段，程序中的下一条指令进入 IF 阶段，依此类推。一旦流水线满了，就像上面的周期 5 一样，CPU 的所有流水线阶段都在忙于处理不同的指令。没有流水线技术，指令 x+1 在指令 x 完成其工作之后才能开始执行。

现代高性能 CPU 具有多个流水线阶段，通常从 10 到 20 个或更多，具体取决于架构和设计目标。这涉及比之前介绍的简单 5 级流水线更复杂的设计。例如，解码阶段可能会分成几个新阶段，我们可能在执行阶段之前添加新阶段来缓冲解码指令，等等。

流水线 CPU 的吞吐量定义为单位时间内完成并退出流水线的指令数量。对于任何给定指令，其延迟是流水线各个阶段的总时间。由于流水线的所有阶段都彼此关联，因此每个阶段都必须准备好以锁步方式移动到下一条指令。将指令从一个阶段移动到下一个阶段所需的时间定义了 CPU 的基本机器周期或时钟。对于给定的流水线，所选择的时

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

Figure 8: 简单的 5 级流水线图。

钟值由流水线中最慢的阶段定义。CPU 硬件设计人员努力平衡每个阶段可以完成的工作量，因为这直接定义了 CPU 的操作频率。增加频率可以提高性能，通常涉及平衡和重新设计流水线以消除由最慢的流水线阶段引起的瓶颈。

在一个理想的、完全平衡且不产生任何停顿的流水线中，流水线机器中每个指令的时间由以下公式给出：

$$\text{流水线机器中每条指令的时间} = \frac{\text{非流水线机器中每条指令的时间}}{\text{流水线阶段数}}$$

在实际实现中，流水线技术引入了几个限制，限制了上述理想模型。流水线冲突阻止了理想的流水线行为，导致停顿。流水线冲突分为三类：结构冲突、数据冲突和控制冲突。幸运的是，对于程序员来说，在现代 CPU 中，所有类别的冲突都由硬件处理。

- **结构冲突：**由资源冲突引起。在很大程度上，它们可以通过复制硬件资源来消除，例如使用多端口寄存器或存储器。然而，消除所有这些冲突可能在硅面积和功耗方面变得非常昂贵。
- **数据冲突：**由程序中的数据依赖性引起，分为三种类型：
 - **写后读 (RAW)** 冲突需要依赖读取在写入之后执行。当指令 $x+1$ 在先前的指令 x 写入源之前读取源时，就会发生这种情况，导致读取到错误的值。CPU 实现了从流水线的后期阶段向前期阶段传递数据的数据转发（称为“绕道”），以减轻与 RAW 冲突相关的惩罚。其思想是在指令 x 完全完成之前，可以将指令 x 的结果转发到指令 $x+1$ 。如果我们看一个例子：

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

对寄存器 $R1$ 存在 RAW 依赖。如果我们直接在添加 $R0$ ADD 1 完成后（从 EXE 流水线阶段），取值，我们就不需要等到 WB 阶段完成，值就会被写入到寄存器文件。绕道有助于节省一些周期。流水线越长，绕道就越有效果。

- **读后写 (WAR)** 冲突需要依赖写入在读取之后执行。当指令 $x+1$ 在指令 x 读取源之前写入源时，就会发生这种情况，导致读取到错误的新值。WAR 冲突不是真正的依赖关系，可以通过一种称为 **寄存器重命名**³² 的技术来消除。这是一种从物理寄存器中抽象逻辑寄存器的技术。CPU 通过保留大量物理寄存器来支持寄存器重命名。逻辑寄存器（体系结构定义的寄存器）只是覆盖更广的寄存器文件上的别名。通过这种 **体系结构状态**³³ 的解耦，解决 WAR 冲突变得简单：我们只需要为写操作使用不同的物理寄存器。例如：

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

³² 寄存器重命名 - https://en.wikipedia.org/wiki/Register_renaming。

³³ 体系结构状态 - https://en.wikipedia.org/wiki/Architectural_state。

对寄存器 R0 存在 WAR 依赖。由于我们有大量的物理寄存器，我们可以简单地为从写操作开始的所有 R0 寄存器的出现重新命名。一旦通过重新命名寄存器 R0 消除了 WAR 冲突，我们就可以以任何顺序安全地执行这两个操作。

- 写后写 (WAW) 冲突需要依赖写入在写入之后执行。当指令 $x+1$ 在指令 x 写入源之前写入源时，就会发生这种情况，导致写入的顺序错误。通过寄存器重命名，可以消除 WAW 冲突，允许两个写入以任何顺序执行，同时保持正确的最终结果。
- 控制冲突：由程序流程的变化引起。它们起源于对分支和其他改变程序流程的指令进行流水线处理。决定分支方向（取或不取）的分支条件在执行流水线阶段解决。因此，除非消除控制冲突，否则无法对下一条指令的提取进行流水线处理。动态分支预测和在下一节中描述的推测执行等技术用于克服控制冲突。

3.3 开发指令级并行性 (ILP)

程序中的大多数指令都适合进行流水线处理并并行执行，因为它们是独立的。现代 CPU 实现了大量额外的硬件功能来利用这种指令级并行性 (ILP)。与先进的编译器技术配合使用，这些硬件功能可以提供显著的性能改进。

3.3.1 乱序执行 (OOO Execution)

图 @fig:Pipelining 中的流水线示例显示所有指令按顺序通过流水线的不同阶段移动，即按照它们在程序中出现的顺序。大多数现代 CPU 支持乱序执行 (OOO execution)，即顺序指令可以以任意顺序进入执行流水线阶段，仅受其依赖关系的限制。乱序执行的 CPU 仍然必须产生与所有指令按程序顺序执行相同的结果。当指令最终执行且其结果正确且可见于体系统结构状态时，该指令被称为 retired。为确保正确性，CPU 必须按照程序顺序使所有指令退役。乱序执行主要用于避免由于依赖关系引起的停顿而导致 CPU 资源的低利用率，特别是在下一节中描述的超标量引擎中。

这些指令的动态调度是由复杂的硬件结构（如分数板）和技术（如寄存器重命名）启用的，以减少数据冲突。在 1960 年代，一些支持动态调度和乱序执行的工作包括 [Tomasulo 算法](#)³⁴（在 IBM360 中实现）和 [Scoreboarding](#)³⁵（在 CDC6600 中实现）。这些开创性的工作影响了所有现代 CPU 架构。分数板硬件用于调度按顺序退役和所有机器状态更新。它跟踪每条指令的数据依赖关系以及流水线中数据的可用位置。大多数实现都致力于在硬件成本与潜在回报之间取得平衡。通常，分数板的大小决定了硬件可以查找的独立指令的距离有多远以进行调度。

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

Figure 9: 乱序执行的概念。

图 @fig:OOO 详细说明了乱序执行的概念，以一个示例为例。假设由于冲突，指令 $x+1$ 在周期 4 和 5 无法执行。顺序 CPU 将阻塞所有后续指令进入 EXE 流水线阶段。在具有乱序执行的 CPU 中，不具有任何冲突（例如，指令 $x+2$ ）的后续指令可以进入并完成其执行。所有指令仍然按顺序退役，即指令按程序顺序完成 WB 阶段。

³⁴ Tomasulo algorithm - https://en.wikipedia.org/wiki/Tomasulo_algorithm.

³⁵ Score boarding - <https://en.wikipedia.org/wiki/Scoreboarding>.

3.3.2 超标量引擎和 VLIW

大多数现代 CPU 都是超标量的，即它们可以在给定周期内发出多个指令。发出宽度是在同一周期内发出的指令的最大数量。当前一代 CPU 的典型发出宽度范围为 2 到 6。为了确保正确平衡，这种超标量引擎还具有多个执行单元和/或流水线执行单元。CPU 还将超标量功能与深度流水线和乱序执行结合起来，以提取给定软件的最大 ILP。

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

Figure 10: 简单 2 路超标量 CPU 的流水线图。

图 @fig:SuperScalar 显示了支持 2 路发出宽度的 CPU 的示例，即在每个周期内，每个流水线阶段处理两条指令。超标量 CPU 通常支持多个独立执行单元，以确保指令在流水线中无冲突地流动。除了流水线化之外，复制执行单元还进一步提高了机器的性能。

像英特尔 Itanium 这样的体系结构将调度超标量、多执行单元机器的负担从硬件转移到了编译器，使用了一种称为 VLIW (Very Long Instruction Word) 的技术。其理念是通过要求编译器选择正确的指令组合来使机器保持完全利用。编译器可以使用软件流水线和循环展开等技术，查看远远超出硬件结构合理支持范围的指令，以找到正确的 ILP。

3.3.3 推测执行

正如前一节所述，如果指令在分支条件解析之前被停顿，控制冲突可能会导致流水线的显著性能损失。为了避免这种性能损失，一种技术是使用硬件分支预测逻辑来预测分支的可能方向，并允许从预测路径执行指令（推测执行）。

让我们考虑 @lst:Speculative 中的短代码示例。为了让处理器了解它应该执行哪个函数，它应该知道条件 $a < b$ 是 false 还是 true。如果不知道，CPU 将等待分支指令的结果，如图 @fig>NoSpeculation 所示。

代码示例：推测执行

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

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

Figure 11: 无推测

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

Figure 12: 推测执行

使用推测执行，CPU 猜测分支的结果，并开始处理所选择路径的指令。假设处理器预测条件 $a < b$ 将被评估为 true。它继续执行而不等待分支结果，并推测性地调用函数 foo（见图 @fig:SpeculativeExec，推测工作用*标记）。直到条件解析为止，机器状态变化才能被提交，以确保机器的体系结构状态永远不会受到推测执行指令的影响。在上面的例子中，分支指令比较了两个标量值，这很快。但实际上，分支指令可能取决于从内存加载的值，这可能需要数百个周期。如果预测是正确的，它将节省大量周期。然而，有时预测是错误的，应该调用函数 bar。在这种情况下，推测执行的结果必须被取消并丢弃。这称为分支错误预测的惩罚，我们将在 Section 4.8 中讨论。

为了跟踪推测的进展，现代 CPU 支持一种称为重新排序缓冲区 (ROB) 的结构。ROB 维护所有指令执行的状态，并按顺序退役指令。推测执行的结果写入 ROB，并按程序流程的相同顺序提交给体系结构寄存器，仅在推测是正确的情况下。CPU 还可以将推测执行与乱序执行相结合，并使用 ROB 来跟踪推测和乱序执行。

3.3.4 分支预测

正如我们刚才看到的，正确的预测极大地提高了执行效率，因为它们允许 CPU 在没有前一条指令结果的情况下继续前进。然而，错误的推测通常会导致昂贵的性能惩罚。现代 CPU 采用了复杂的动态分支预测机制，提供非常高的准确性，并能够适应分支行为的动态变化。有三种类型的分支可能需要以特殊方式处理：

- 无条件跳转和直接调用：它们最容易预测，因为它们总是被执行并且每次都以相同的方向执行。
- 条件分支：它们有两个潜在的结果：被执行或不被执行。被执行的分支可以向前或向后跳转。前向条件分支通常用于生成 if-else 语句，其不被执行的可能性很高，因为通常表示错误检查代码。后向条件跳转经常出现在循环中，并用于转到循环的下一次迭代；此类分支通常被执行。
- 间接调用和跳转：它们具有许多目标。间接跳转或间接调用可以生成 switch 语句、函数指针或 virtual 函数。函数返回也值得关注，因为它也有许多潜在的目标。

大多数预测算法基于分支的先前结果。分支预测单元 (BPU) 的核心是分支目标缓冲区 (BTB)，它为每个分支缓存目标地址。预测算法每个周期都要查询 BTB，以生成下一个要提取指令的地址。CPU 使用该新地址提取下一个指令块。如果在当前提取块中没有识别出分支，则提取的下一个地址将是下一个顺序对齐的提取块（顺序提取）。

无条件分支不需要预测；我们只需要在 BTB 中查找目标地址。记住，每个周期 BPU 都需要生成下一个地址，以避免流水线停滞。我们可以仅从指令编码中提取地址，但这样我们必须等到解码阶段结束，这将在流水线中引入一个气泡并使事情变慢。因此，在提取分支时必须确定下一个提取地址。

对于条件分支，我们首先需要预测分支是否被执行。如果未执行，则我们会顺序执行，并且无需查找目标。否则，我们将在 BTB 中查找目标地址。条件分支通常占据总分支的最大部分，并且是生产软件中错误预测的主要来源。对于间接分支，我们需要选择可能的目标之一，但是预测算法可以与条件分支非常相似。

所有预测机制都试图利用两个重要原则，这与我们稍后将讨论的缓存类似：

- 时间相关性：分支的解决方式可能是下次执行时解决方式的很好的预测器。这也称为局部相关性。
- 空间相关性：几个相邻的分支可能以高度相关的方式解决（执行的首选路径）。这也称为全局相关性。

最佳的准确性通常通过同时利用局部和全局相关性来实现。因此，我们不仅查看当前分支的结果历史，还将其与其他分支的结果相关联。

另一种常见的技术是混合预测。其思想是一些分支具有偏向行为。例如，如果条件分支 99.9% 的时间都朝着一个方向走，就没有必要使用复杂的预测器并污染其数据结构。可以使用一个简单得多的机制。另一个示例是循环分支。如果分支具有循环行为，则可以使用专用的循环预测器进行预测，该预测器将记住循环通常执行的迭代次数。

如今，最先进的预测主要由类似 TAGE [Seznec & Michaud, 2006] 或感知器-based [Jimenez & Lin, 2001] 的预测器主导。冠军³⁶分支预测器在每 1000 条指令中不到 3 次错误预测。现代 CPU 在大多数工作负载上通常达到超过 95% 的预测率。

3.4 SIMD 多处理器

另一种被广泛用于许多工作负载的多处理器变体被称为单指令多数据 (SIMD)。顾名思义，在 SIMD 处理器中，单个指令在单个周期内使用多个独立的功能单元操作许多数据元素。向量和矩阵的操作很适合 SIMD 架构，因为向量或矩阵的每个元素都可以使用相同的指令进行处理。SIMD 架构可以更有效地处理大量数据，最适合涉及向量操作的数据并行应用程序。

图 @fig:SIMD 展示了代码 @lst:SIMD 中的标量和 SIMD 执行模式。在传统的 SISD (单指令，单数据) 模式中，加法操作分别应用于数组 **a** 和 **b** 的每个元素。然而，在 SIMD 模式中，加法同时应用于多个元素。如果我们针对具有能够对 256 位向量执行操作的执行单元的 CPU 架构进行优化，我们可以使用单个指令处理四个双精度元素。这导致发出的指令数量减少了 4 倍，并且可能比四个标量计算获得 4 倍的加速。但是在实践中，由于各种原因，性能优势并不那么直接。

代码清单：SIMD 执行

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

对于常规的 SISD 指令，处理器使用通用寄存器。同样，对于 SIMD 指令，CPU 有一组 SIMD 寄存器，用于保持从内存加载的数据和存储计算的中间结果。在我们的示例中，与数组 **a** 和 **b** 对应的两个连续的 256 位数据区域将从内存加载并存储在两个单独的向量寄存器中。接下来，将进行逐元素加法运算，并将结果存储在一个新的 256 位向量寄存器中。最后，将结果从向量寄存器写入对应于数组 **c** 的 256 位内存区域。请注意，数据元素可以是整数或浮点数。

大多数流行的 CPU 架构都具有矢量指令，包括 x86、PowerPC、Arm 和 RISC-V。1996 年，英特尔推出了 MMX，一个针对多媒体应用程序设计的 SIMD 指令集。随后，英特尔引入了具有增强功能和增加矢量大小的新指令集：SSE、AVX、AVX2、AVX-512。Arm 在其各个版本的架构中选择性地支持 128 位 NEON 指令集。在第 8 版 (aarch64) 中，这种支持变得是强制性的，并添加了新的指令。

随着新的指令集变得可用，开始着手使它们对软件工程师可用。利用 SIMD 指令所需的软件更改称为代码向量化。最初，SIMD 指令是用汇编语言编程的。后来，引入了特殊的编译器内置函数，它们是一对一映射到 SIMD 指令的小函数。今天，所有主要的编译器都支持针对流行处理器的自动矢量化，即它们可以直接从 C/C++、Java、Rust 等高级语言编写的代码生成 SIMD 指令。

[待办事项]：解释术语“循环余数” [待办事项]：或许解释/介绍掩码概念 [待办事项]：解释术语“SIMD 线”

为了使代码能够在支持不同矢量长度的系统上运行，Arm 引入了 SVE 指令集。其定义特征是可伸缩矢量的概念：它们的长度在编译时是未知的。使用 SVE，无需将软件移植到每种可能的矢量长度。用户不必重新编译其应用程序的

³⁶ 第五届冠军分支预测大赛 - <https://jilp.org/cbp2016>

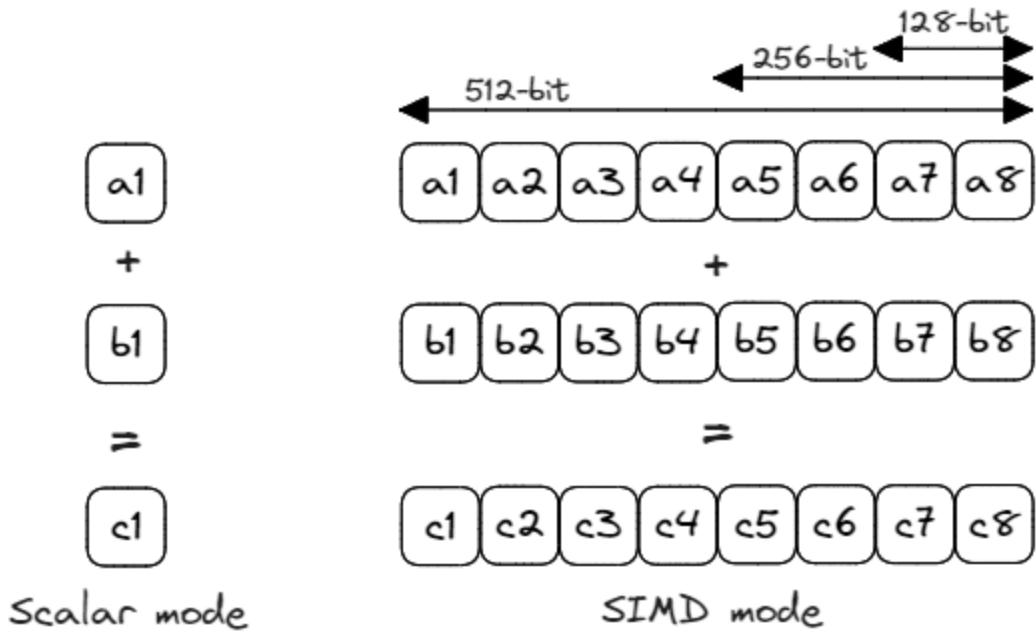


Figure 13: 标量和 SIMD 操作的示例。

源代码以利用在较新的 CPU 代中可用的更宽矢量。可伸缩矢量的另一个示例是 RISC-V V 扩展 (RVV)，该扩展于 2021 年底获得批准。一些实现支持相当宽 (2048 位) 的矢量，并且最多可以将八个矢量组合在一起，形成 16,384 位的矢量，这极大地减少了执行的指令数量。在每次循环迭代中，用户代码通常执行 `ptr += number_of_lanes`，其中 `number_of_lanes` 在编译时是未知的。ARM SVE 为这种长度相关操作提供了特殊指令，而 RVV 允许程序员查询/设置 `number_of_lanes`。

此外，CPU 越来越多地加速机器学习中经常使用的矩阵乘法。英特尔的 AMX 扩展，支持 Sapphire Rapids，将形状为 16x64 和 64x16 的 8 位矩阵相乘，并累积为 32 位的 16x16 矩阵。相比之下，苹果 CPU 中无关但同名的 AMX 扩展，以及 ARM 的 SME 扩展，计算存储在特殊的 512 位寄存器或可伸缩矢量中的一行和一列的外积。

最初，SIMD 是由多媒体应用程序和科学计算推动的，但后来在许多其他领域找到了用途。随着时间的推移，SIMD 指令集中支持的操作集稳步增加。除了图 @fig:SIMD 中显示的直接算术运算外，SIMD 的新用途还包括：

- 字符串处理：查找字符，验证 UTF-8，[1] 解析 JSON³⁷ 和 CSV；³⁸
- 哈希运算，³⁹ 随机生成，⁴⁰ 密码学 (AES)；
- 列式数据库（位打包、过滤、连接）；
- 对内置类型进行排序 (VQSort, ⁴¹ QuickSelect)；
- 机器学习和人工智能（加速 PyTorch、Tensorflow）。

3.5 开发线程级并行性

前面描述的技术依赖于程序中可用的并行性来加速执行。除此之外，CPU 还支持利用跨进程和/或线程的并行性的技术。接下来，我们将讨论三种利用线程级并行性 (TLP) 的技术：多核系统、同时多线程和混合架构。这些技术使

³⁷ 解析 JSON - <https://github.com/simdjson/simdjson>.

³⁸ 解析 CSV - <https://github.com/geofflangdale/simdcsv>

³⁹ SIMD 哈希运算 - <https://github.com/google/highwayhash>

⁴⁰ 随机生成 - [Abseil 库](#)

⁴¹ 排序 - [VQSort](#)

得能够充分利用可用的硬件资源，并提高系统的吞吐量。

3.5.1 多核系统

随着处理器架构师开始达到半导体设计和制造的实际限制，GHz 竞赛减缓，设计师不得不专注于其他创新来提高 CPU 性能。其中一个关键方向是多核设计，试图增加每代的核心数。其想法是在单个芯片上复制多个处理器核心，并让它们同时为不同的程序提供服务。例如，一个核心可以运行 Web 浏览器，另一个核心可以渲染视频，另一个可以播放音乐，所有这些都可以同时进行。对于服务器机器，来自不同客户端的请求可以在不同的核心上进行处理，这可以极大地增加系统的吞吐量。

第一个面向消费者的双核处理器是英特尔 Core 2 Duo，于 2005 年发布，随后是同年稍后发布的 AMD Athlon X2 架构。多核系统导致许多软件组件被重新设计，并影响了我们编写代码的方式。如今，几乎所有面向消费者的设备中的处理器都是多核 CPU。在撰写本书时，高端笔记本电脑包含超过十个物理核心，而服务器处理器包含的核心几乎达到 100 个。

这听起来可能非常令人印象深刻，但我们不能无限地增加核心。首先，每个核心在工作时会产生热量，并且安全地通过处理器封装从核心中散热仍然是一个挑战。这意味着当更多核心运行时，热量可能会迅速超过冷却能力。在这种情况下，多核处理器将降低时钟速度。这是您可以看到具有大量核心的服务器芯片频率明显低于进入笔记本电脑和台式机的处理器的原因之一。

多核系统中的核心彼此连接，也连接到共享资源，例如末级缓存和内存控制器。这样的通信通道称为互连，通常具有环形或网状拓扑结构。CPU 设计者面临的另一个挑战是随着核心数量的增加，保持机器的平衡。当您复制核心时，一些资源保持共享，例如内存总线和末级缓存。这导致随着核心的增加，性能回报递减，除非您还解决了其他共享资源的吞吐量，例如互连带宽、末级缓存大小和带宽，以及内存带宽。共享资源经常成为多核系统中性能问题的来源。

3.5.2 同时多线程

改进多线程性能的一种更复杂的方法是同时多线程 (SMT)。人们经常使用术语超线程来描述相同的事物。这种技术的目标是充分利用 CPU 管道的可用宽度。SMT 允许多个软件线程在同一物理核心上使用共享资源同时运行。更准确地说，来自多个软件线程的指令在同一周期内同时执行。这些线程不必来自同一个进程；它们可以是完全不同的程序，恰好被调度在同一个物理核心上。

图 @fig:SMT 展示了在非 SMT 和 SMT2 处理器上的执行示例。在两种情况下，处理器管道的宽度为四，每个插槽表示发出新指令的机会。100% 的机器利用率是指没有未使用的插槽，这在实际工作负载中从不会发生。很容易看出，在非 SMT 情况下，存在许多未使用的插槽，因此可用资源没有得到充分利用。这可能是由于多种原因引起的；一个常见的原因是缓存未命中。在周期 3 时，线程 1 由于等待数据到达而无法取得进展。SMT 处理器利用这个机会从另一个线程调度有用的工作。这里的目标是通过另一个线程占用未使用的插槽来隐藏内存延迟，提高硬件利用率和多线程性能。

在 SMT2 实现中，每个物理核心用两个逻辑核心表示，这些逻辑核心对操作系统显示为两个独立的处理器，可用于接受工作。考虑这样一种情况，我们有 16 个准备运行的软件线程，但只有 8 个物理核心。在非 SMT 系统中，只有 8 个线程将同时运行，而在 SMT2 中，我们可以同时执行所有 16 个线程。在另一种假设情况下，如果两个程序运行在一个启用了 SMT 的核心上，并且每个程序一直只利用了四个可用插槽中的两个，那么它们以与在该物理核心上独自运行时一样的速度运行的几率很高。

尽管两个程序运行在同一个处理器核心上，它们彼此完全分离。在启用了 SMT 的处理器中，即使指令混合在一起，它们也具有不同的上下文，有助于保持执行的正确性。为了支持 SMT，CPU 必须复制体系结构状态（程序计数器、寄存器）以保持线程上下文。其他 CPU 资源可以共享。在典型的实现中，缓存资源在硬件线程之间动态共享。用于跟踪乱序执行和推测执行的资源可以复制或分区。

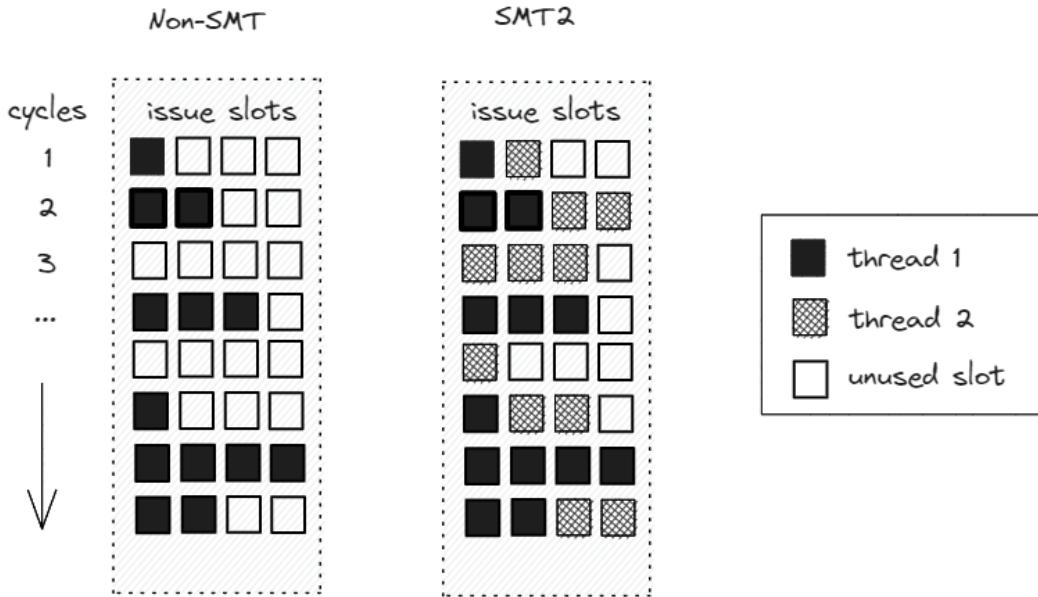


Figure 14: 在 4 宽度非 SMT 和 4 宽度 SMT2 处理器上的执行。

在 SMT2 核心中，两个逻辑核心确实同时运行。在 CPU 前端，它们以交替的顺序（每个周期或几个周期）获取指令。在后端，处理器每个周期从所有线程中选择要执行的指令。指令执行混合，因为处理器动态地将执行单元在两个线程之间调度。

因此，SMT 是一种非常灵活的设置，可以恢复未使用的 CPU 发射插槽。SMT 提供了相等的单线程性能，除了对多线程的好处外。现代多线程 CPU 支持两路（SMT2）或四路（SMT4）。

SMT 也有自己的缺点。由于某些资源在逻辑核心之间共享，它们最终可能会竞争使用这些资源。最有可能的 SMT 惩罚是由于对 L1 和 L2 缓存的竞争。由于它们在两个逻辑核心之间共享，它们可能在缓存中缺少空间，并迫使将来将由另一个线程使用的数据逐出。

SMT 也给软件开发人员带来了很大的负担，因为它使得更难以预测和衡量在 SMT 核心上运行的应用程序的性能。想象一下，您在 SMT 核心上运行性能关键代码，突然操作系统在同一处理器的兄弟逻辑核心上放置了另一个要求严格的作业。您的代码几乎耗尽了机器的资源，现在您需要与其他人共享。在云环境中，这个问题特别突出，因为您无法预测您的应用程序是否会有嘈杂的邻居。

某些同时多线程实现存在安全问题。研究人员表明，一些早期实现存在漏洞，通过这些漏洞可以使一个应用程序从同一处理器的兄弟逻辑核心中监视其缓存使用来窃取关键信息（如加密密钥）另一个应用程序。我们不会深入探讨这个话题，因为硬件安全不在本书的范围内。

3.5.3 混合架构

计算机架构师还开发了混合 CPU 设计，其中两种（或更多）类型的核心放置在同一处理器中。通常，更强大的核心与相对较慢的核心配对，以解决不同的目标。在这样的系统中，大核心用于延迟敏感的任务，而小核心则提供了较低的功耗。但是，两种类型的核心也可以同时使用，以提高多线程性能。所有核心都可以访问相同的内存，因此工作负载可以在大核心和小核心之间动态迁移。其目的是创建一个能够更好地适应动态计算需求并使用更少功耗的多核处理器。例如，视频游戏既有单核心突发性能的部分，也有可以扩展到多个核心的部分。

第一个主流的混合架构是 ARM 的 big.LITTLE，于 2011 年 10 月推出。其他供应商也采用了这种方法。苹果于 2020 年推出了其 M1 芯片，具有四个高性能的“Firestorm”核心和四个节能的“IceStorm”核心。英特尔于 2021 年推出了其

Alderlake 混合架构，顶级配置中配备了八个 P 核心和八个 E 核心。

混合架构结合了两种核心类型的优势，但它也带来了一系列挑战。首先，它要求核心完全兼容 ISA，即它们应该能够执行相同的指令集。否则，调度就会受到限制。例如，如果大核心具有一些小核心上不可用的高级指令，那么您只能将大核心分配给使用这些指令的工作负载。这就是为什么通常供应商在选择混合处理器的 ISA 时使用“最大公约数”的方法。

即使具有 ISA 兼容的核心，调度也变得具有挑战性。不同类型的工作负载需要特定的调度方案，例如，突发执行与稳定执行，低 IPC 与高 IPC，低重要性与高重要性等。这很快就变得不那么简单了。以下是一些优化调度的考虑因素：

- 利用小核心以节省功耗。不要为后台工作唤醒大核心。
- 识别适合转移到较小核心的候选项（低重要性，低 IPC）。类似地，将高重要性，高 IPC 任务提升到大核心。
- 在分配新任务时，首先使用空闲的大核心。在 SMT 的情况下，使用两个逻辑线程都空闲的大核心。之后，使用空闲的小核心。之后，使用大核心的兄弟逻辑线程。

从程序员的角度来看，不需要对代码进行任何更改就可以利用混合系统。这种方法在面向客户的设备中变得非常流行，特别是在智能手机中。

3.6 存储器层次结构

为了有效地利用 CPU 中提供的所有硬件资源，需要在正确的时间提供正确的数据。理解存储器层次结构对于充分发挥 CPU 性能至关重要。大多数程序都表现出局部性的属性：它们不会均匀地访问所有代码或数据。CPU 存储器层次结构建立在两个基本属性上：

- 时间局部性：当访问给定的内存位置时，很可能在不久的将来再次访问同一位置。理想情况下，我们希望在下一次需要时，该信息位于缓存中。
- 空间局部性：当访问给定的内存位置时，很可能在不久的将来访问附近的位置。这指的是将相关数据放在彼此附近。当程序从内存中读取一个字节时，通常会获取更大的内存块（缓存行），因为该程序很可能很快就会需要该数据。

本节概述了现代 CPU 支持的存储器层次结构系统的关键属性。

3.6.1 缓存层次结构

缓存是从 CPU 流水线发出的任何请求（用于代码或数据）的存储器层次结构的第一级。理想情况下，具有最小访问延迟的无限缓存是流水线的最佳选择。然而，在现实中，任何缓存的访问时间都会随着大小的增加而增加。因此，缓存被组织为一系列靠近执行单元的小型、快速的存储块，由较大、较慢的块支持。缓存层次结构的特定级别可以专门用于代码（指令缓存，i-cache）或数据（数据缓存，d-cache），或者在代码和数据之间共享（统一缓存）。此外，层次结构的某些级别可以是特定核心专用的，而其他级别可以在核心之间共享。

缓存被组织为具有定义的块大小（缓存行）。现代 CPU 中典型的缓存行大小为 64 字节。靠近执行流水线的缓存通常的大小范围从 8KiB 到 32KiB 不等。在层次结构中更远的缓存可以在 64KiB 到 16MiB 之间。任何级别的缓存的体系结构由以下四个属性定义。

3.6.1.1 数据在缓存中的放置。用于请求的地址用于访问缓存。在直接映射缓存中，给定块地址只能出现在缓存中的一个位置，并由下面的映射函数定义。

$$\text{缓存中的块数量} = \frac{\text{缓存大小}}{\text{缓存块大小}}$$

$$\text{直接映射位置} = (\text{块地址}) \bmod (\text{缓存中的块数量})$$

在完全关联的缓存中，给定块可以放置在缓存中的任何位置。

直接映射缓存和完全关联映射之间的中间选项是集合关联映射。在这样的缓存中，块被组织为集合，通常每个集合包含 2、4、8 或 16 个块。首先将给定地址映射到一个集合。在集合中，地址可以放置在该集合中的任何位置，即在该集合的块中。具有 m 个块的每个集合的缓存被描述为 m 路集合关联缓存。集合关联缓存的公式为：

$$\text{缓存中的集合数量} = \frac{\text{缓存中的块数量}}{\text{每组的块数量 (关联性)}}$$

$$\text{集合 (m 路) 关联位置} = (\text{块地址}) \bmod (\text{缓存中的集合数量})$$

3.6.1.2 在缓存中查找数据 集合关联缓存中的每个块都与一个地址标签相关联。此外，标签还包含状态位，例如有效位，用于指示数据是否有效。标签还可以包含其他位，用于指示访问信息、共享信息等，这将在本章的后续部分描述。

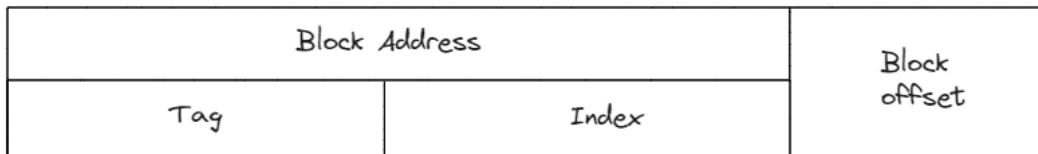


Figure 15: 用于缓存查找的地址组织。

图 15 显示了从流水线生成的地址如何用于检查缓存。最低位地址位定义了给定块内的偏移量；块偏移位（对于 32 字节缓存行为 5 位，对于 64 字节缓存行为 6 位）。使用上面描述的公式根据索引位选择集合。选择集合后，标签位用于与该集合中的所有标签进行比较。如果其中一个标签与传入请求的标签匹配并且有效位已设置，则会发生缓存命中。与该块条目相关联的数据（与标签查找并行读出缓存的数据数组）将提供给执行流水线。如果标签不匹配，则会发生缓存未命中。

3.6.1.3 处理未命中 当发生缓存未命中时，控制器必须选择要替换的缓存中的块，以分配导致未命中的地址。对于直接映射缓存，由于新地址只能分配到单个位置，因此取消分配映射到该位置的先前条目，并在其位置安装新条目。在集合关联缓存中，由于新的缓存块可以放置在集合的任何块中，因此需要替换算法。常用的替换算法是最近最少使用 (LRU) 策略，其中最近最少访问的块被淘汰以腾出空间以存放未命中地址。另一种选择是随机选择一个块作为受害块。大多数 CPU 在硬件中定义了这些功能，使执行软件更加容易。

3.6.1.4 处理写操作 相对于数据读取，对缓存的写入访问较少。在缓存中处理写入更加困难，CPU 实现使用各种技术来处理这种复杂性。软件开发人员应特别注意硬件支持的各种写缓存流程，以确保其代码的最佳性能。

CPU 设计使用两种基本机制来处理命中缓存的写操作：

- 在写透写缓存中，命中的数据被写入缓存中的块和层次结构中的下一级。
- 在写回缓存中，命中的数据仅被写入缓存中。随后，层次结构的较低级别包含陈旧的数据。修改行的状态通过标签中的脏位进行跟踪。当修改的缓存行最终从缓存中淘汰时，写回操作会强制将数据写回到下一级。

写操作导致的缓存未命中可以通过两种方式处理：

- 在写分配或写未命中获取缓存中，从层次结构的下一级加载未命中位置的数据到缓存中，随后处理写操作，就像写命中一样。
- 如果缓存使用非写分配策略，则将缓存未命中事务直接发送到层次结构的下一级，并且该块不加载到缓存中。

在这些选项中，大多数设计通常选择实现具有写分配策略的写回缓存，因为这两种技术都尝试将后续写事务转换为缓存命中，而无需向下一一级发送额外的流量。写透写缓存通常使用非写分配策略。

3.6.1.5 其他缓存优化技术 对于程序员来说，理解缓存层次结构的行为对于从任何应用程序中提取性能至关重要。从流水线的角度来看，访问任何请求的延迟由以下公式给出，该公式可以递归应用到缓存层次结构的所有级别，直到主存：

$$\text{平均访问延迟} = \text{命中时间} + \text{未命中率} \times \text{未命中惩罚}$$

硬件设计者面临的挑战是通过许多新颖的微体系结构技术来减少命中时间和未命中惩罚。从根本上说，缓存未命中会阻塞流水线并损害性能。任何缓存的未命中率高度依赖于缓存体系结构（块大小、关联性）和机器上运行的软件。因此，优化未命中率成为硬件-软件协同设计的工作。正如前面讨论的，CPU 为缓存提供了最佳的硬件组织。下面描述了可以在硬件和软件中实现的降低缓存未命中率的其他技术。

3.6.1.5.1 硬件和软件预取 避免缓存未命中和随后的停顿的一种方法是在流水线需求之前将指令和数据预取到缓存层次结构的不同级别。假设处理未命中的惩罚的时间大部分可以被隐藏，如果预取请求在流水线中足够提前发出。大多数 CPU 提供了隐式基于硬件的预取，由程序员控制的显式软件预取来补充。

硬件预取器观察正在运行的应用程序的行为，并在缓存未命中的重复模式上启动预取。硬件预取可以自动适应应用程序的动态行为，例如不同的数据集，并且不需要优化编译器或分析支持。此外，硬件预取工作而不需要额外的地址生成和预取指令的开销。然而，硬件预取仅限于学习和预取有限集的缓存未命中模式。

软件内存预取补充了硬件进行的预取。开发人员可以通过专用的硬件指令（见 Section ??）指定提前需要的内存位置。编译器还可以自动将预取指令添加到代码中，以请求使用之前的数据。预取技术需要在需求和预取请求之间进行平衡，以防止预取流量减慢需求流量。

3.6.2 主存储器

主存储器是缓存之后的下一个层次，由内存控制器单元（MCU）发起对数据的加载和存储请求。过去，这个电路位于主板上的北桥芯片中。但是现在，大多数处理器都将此组件嵌入其中，因此 CPU 与主存储器之间有一个专用的内存总线连接。

主存储器使用 DRAM（动态随机存取存储器）技术，支持以合理的成本点获取大容量。在比较 DRAM 模块时，人们通常会关注内存密度和内存速度，当然还有其价格。内存密度定义了模块拥有的内存量，以 GB 为单位。显然，可用内存越多越好，因为它是操作系统和应用程序使用的宝贵资源。

主存储器的性能由延迟和带宽描述。内存延迟是发出内存访问请求和 CPU 可用于使用数据之间经过的时间。内存带宽定义了在一定时间内可以获取多少字节，通常以每秒字节数（GB/s）表示。

3.6.2.1 DDR DDR（双倍数据速率）DRAM 技术是大多数 CPU 支持的主要 DRAM 技术。从历史上看，DRAM 的带宽每一代都在提高，而 DRAM 的延迟却保持不变甚至增加。表 @tbl:mem_rate 显示了过去三代 DDR 技术的最高数据速率、峰值带宽以及相应的读取延迟。数据速率以每秒传输的百万次（MT/s）为单位。此表中显示的延迟对应于 DRAM 器件本身的延迟。通常，由于缓存控制器、内存控制器和芯片内连接引起的额外延迟和排队延迟，从 CPU 流水线（在加载到使用的缓存未命中时）看到的延迟较高（在 50 纳秒至 150 纳秒范围内）。您可以在 Section 4.10 中看到测量观察到的内存延迟和带宽的示例。

Table 2: 过去三代 DDR 技术的性能特征。

DDR 年份	最高数据速率 (MT/s)	速率 (MT/s) 峰值	带宽 (GB/s)	设备	内读取延迟 (ns)
DDR3	2007	2133	17.1		10.3
DDR4	2014	3200	25.6		12.5
DDR5	2020	6400	51.2		14

值得一提的是，DRAM 芯片需要定期刷新其内存单元。这是因为位值被存储为微小电容器上的电荷存在，所以它可能会随着时间的推移失去电荷。为了防止这种情况发生，有特殊的电路读取每个单元并将其写回，有效地恢复电容器的电荷。当 DRAM 芯片处于刷新过程中时，它不会响应内存访问请求。

DRAM 模块组织为一组 DRAM 芯片。内存 rank 是一个术语，用于描述模块上存在多少组 DRAM 芯片。例如，单 rank (1R) 内存模块包含一组 DRAM 芯片。双 rank (2R) 内存模块有两组 DRAM 芯片，因此将单 rank 模块的容量加倍。同样，还可以购买四 rank (4R) 和八 rank (8R) 内存模块。

每个 rank 由多个 DRAM 芯片组成。内存 width 定义了每个 DRAM 芯片的总线宽度。由于每个 rank 的总线宽度为 64 位（或 ECC RAM 为 72 位），它还定义了 rank 内存在芯片中的数量。内存宽度可以是x4、x8或x16中的一个值，这定义了发送到每个芯片的总线宽度。例如，图 @fig:Dram_ranks 显示了一个 2Rx16 双 rank DRAM DDR4 模块的组织，总容量为 2GB。每个 rank 中有四个芯片，总线宽度为 16 位。四个芯片共同提供 64 位输出。两个 rank 通过一个 rank 选择信号逐个选择。

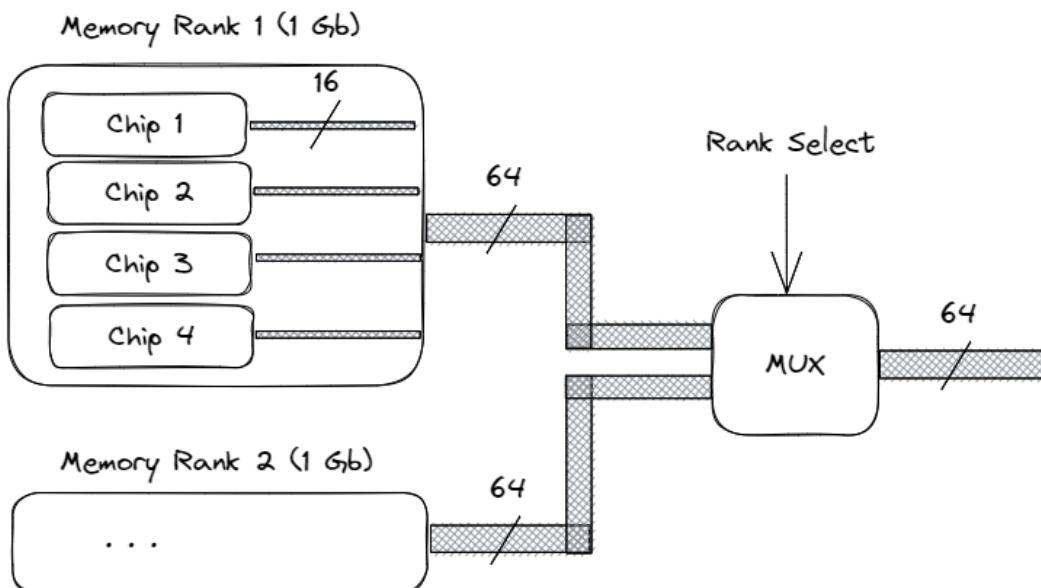


Figure 16: 2Rx16 双 rank DRAM DDR4 模块，总容量 2GB 的组织。

单 rank 或双 rank 的性能哪个更好并没有直接的答案，因为它取决于应用程序的类型。通过 rank 选择信号从一个 rank 切换到另一个 rank 需要额外的时钟周期，这可能会增加访问延迟。另一方面，如果一个 rank 没有被访问，它可以在其他 rank 忙碌时并行进行刷新周期。一旦上一个 rank 完成数据传输，下一个 rank 就可以立即开始传输。此外，单 rank 模块产生的热量更少，故障的可能性更低。

进一步说，我们可以在系统中安装多个 DRAM 模块，不仅增加内存容量，还增加内存带宽。多个内存通道的设置用于扩展内存控制器和 DRAM 之间的通信速度。

具有单个内存通道的系统在 DRAM 和内存控制器之间有一个 64 位宽的数据总线。多通道架构通过增加内存数据总线的宽度，允许同时访问 DRAM 模块。例如，双通道架构将内存数据总线的宽度从 64 位扩展到 128 位，将可用带宽加倍，参见图 @fig:Dram_channels。请注意，每个内存模块仍然是一个 64 位设备，但我们将它们连接方式有所不同。如今，服务器机器通常具有四个和八个内存通道。

或者，您也可能遇到具有复制内存控制器的设置。例如，处理器可能具有两个集成的内存控制器，每个内存控制器都可以支持多个内存通道。这两个控制器是独立的，只查看总物理内存地址空间的自己的部分。

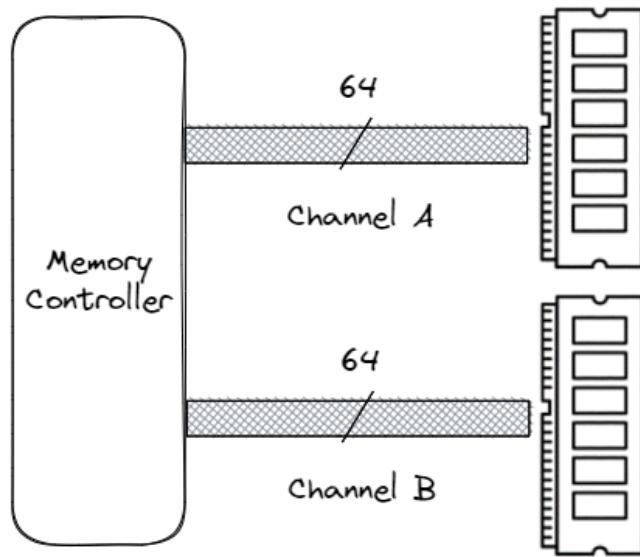


Figure 17: 双通道 DRAM 设置的组织。

我们可以通过以下简单的公式快速计算给定内存技术的最大内存带宽：

$$\text{最大内存带宽} = \text{数据速率} \times \text{每周期字节数}$$

例如，对于单通道 DDR4 配置，数据速率为 2400 MT/s，每个内存周期可以传输 64 位（8 字节），因此最大带宽等于 $2400 * 8 = 19.2 \text{ GB/s}$ 。双通道或双内存控制器设置将带宽加倍至 38.4 GB/s。但要记住，这些数字是假设数据传输将在每个内存时钟周期中发生的理论最大值，在实践中实际上是不会发生的。因此，当测量实际内存速度时，您将始终看到比最大理论传输带宽低的值。

要启用多通道配置，您需要具备支持这种架构的 CPU 和主板，并在主板上正确的内存插槽中安装相同数量的内存模块。在 Windows 上检查设置的最快方法是运行诸如 CPU-Z 或 HwInfo 之类的硬件识别实用程序；在 Linux 上，您可以使用 `dmidecode` 命令。或者，您可以运行内存带宽基准测试，例如 Intel 的 `m1c` 或 `Stream`。

要在系统中利用多个内存通道，有一种称为交错的技术。它在一个页面中将相邻地址在多个内存设备之间分布。图 @fig:Dram_channel_interleaving 显示了用于顺序内存访问的 2 路交错的示例。与以前一样，我们有双通道内存配置（通道 A 和 B），具有两个独立的内存控制器。现代处理器按每四个缓存行（256 字节）进行交错，即，前四个相邻缓存行发送到通道 A，然后下一组四个缓存行发送到通道 B。

如果不使用交错，连续的相邻访问将发送到同一个内存控制器，而不是利用第二个可用的控制器。相反，交错使硬件并行性更好地利用了可用的内存带宽。对于大多数工作负载，当所有通道都被填充时，性能最大化，因为它将单个内存区域扩展到尽可能多的 DRAM 模块中。

虽然增加内存带宽通常是有益的，但它并不总是转化为更好的系统性能，而且高度依赖于应用程序。另一方面，注意可用和已使用的内存带宽非常重要，因为一旦它成为主要瓶颈，应用程序就会停止扩展，即，添加更多核心并不能使其运行更快。

3.6.2.2 GDDR 和 HBM 除了多通道 DDR 外，还有其他技术针对需要更高内存带宽以实现更高性能的工作负载。GDDR（图形 DDR）和 HBM（高带宽内存）等技术是最显着的技术。它们在高端图形、高性能计算（如气候建模、分子动力学、物理模拟），但也包括自动驾驶和人工智能/机器学习等领域中得到应用。它们在这些领域中非常适用，因为这些应用需要非常快速地移动大量数据。

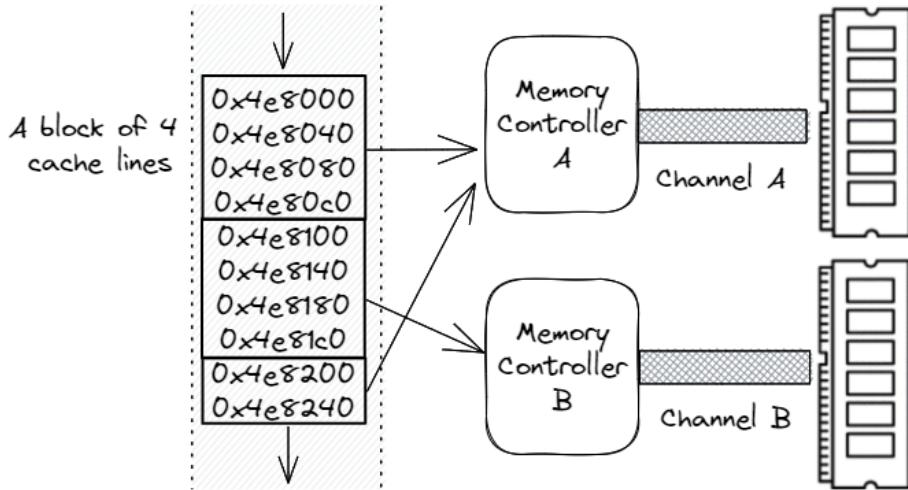


Figure 18: 顺序内存访问的 2 路交错。

GDDR 最初是为图形设计的，现在几乎每个高性能图形卡都在使用。虽然 GDDR 与 DDR 共享一些特征，但它也有很大的不同。虽然 DRAM DDR 设计用于更低的延迟，但 GDDR 则设计用于更高的带宽，因为它位于处理器芯片本身的一封装中。与 DDR 类似，GDDR 接口每个时钟周期传输两个 32 位字（总共 64 位）。最新的 GDDR6X 标准可以实现高达 168 GB/s 的带宽，以相对较低的 656 MHz 频率运行。

HBM 是一种新型的 CPU/GPU 内存，它垂直堆叠内存芯片，也称为 3D 堆叠。与 GDDR 类似，HBM 极大地缩短了数据到达处理器的距离。与 DDR 和 GDDR 的主要区别在于，HBM 内存总线非常宽：每个 HBM 堆栈为 1024 位。这使 HBM 能够实现超高带宽。最新的 HBM3 标准支持每个封装高达 665 GB/s 的带宽。它还以 500 MHz 的低频率运行，并具有每个封装高达 48 GB 的内存密度。

如果您想获得尽可能多的内存带宽，那么拥有 HBM 的系统将是一个不错的选择。但是，在撰写本文时，这项技术相当昂贵。由于 GDDR 主要用于图形卡，HBM 可能是加速在 CPU 上运行的某些工作负载的好选择。事实上，第一批具有集成 HBM 的 x86 通用服务器芯片现已上市。

3.7 虚拟内存

虚拟内存是一种机制，将连接到 CPU 的物理内存与在 CPU 上执行的所有进程共享。虚拟内存提供了一种保护机制，可以防止其他进程访问分配给特定进程的内存。虚拟内存还提供重定位，即能够在物理内存中的任何位置加载程序而不更改程序中的地址。

在支持虚拟内存的 CPU 中，程序使用虚拟地址进行访问。但是，虽然用户代码在虚拟地址上运行，但从内存中检索数据需要物理地址。此外，为了有效管理稀缺的物理内存，它被划分为页面。因此，应用程序在一组由操作系统提供的页面上运行。

访问数据和代码（指令）都需要地址转换。具有 4KB 页面大小的系统的机制如图 19 所示。虚拟地址分为两部分。虚拟页号（52 个最高有效位）用于索引页表，以生成虚拟页号和相应物理页之间的映射。对于 4KB 页面中的偏移量，我们需要 12 位；正如已经说过的那样，64 位指针的其他 52 位用于页本身的地址。请注意，页面内的偏移量（12 个最低有效位）不需要转换，并且“原样”用于访问物理内存位置。

页表可以是单层或嵌套的。图 20 显示了一个 2 级页表的示例。请注意地址如何分成更多部分。首先要提的是，没有使用 16 个最高有效位。这似乎浪费了位，但即使使用剩余的 48 位，我们也可以寻址 256 TB 的总内存 (2^{48})。一些应用程序使用这些未使用的位来保留元数据，也称为“指针标记”。

嵌套页表是一个 radix 树，它与一些元数据一起保存物理页地址。要找到这样一个 2 级页表的翻译，我们首先使用

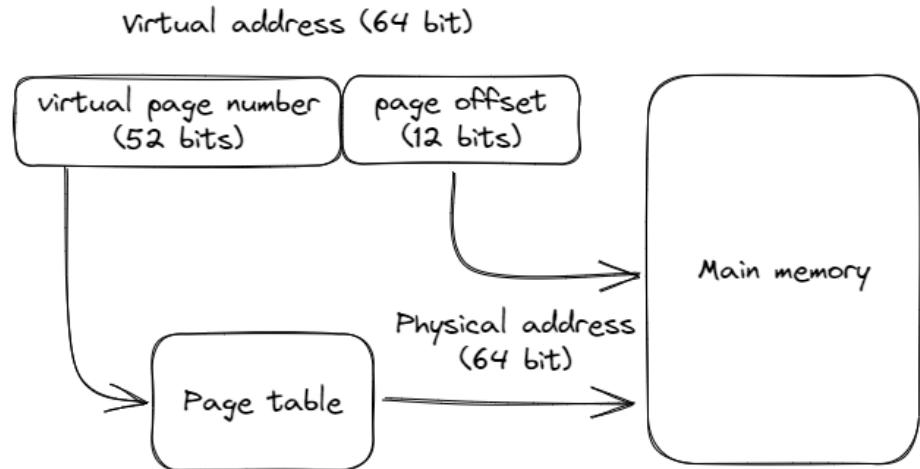


Figure 19: 4KB 页面的虚拟到物理地址转换

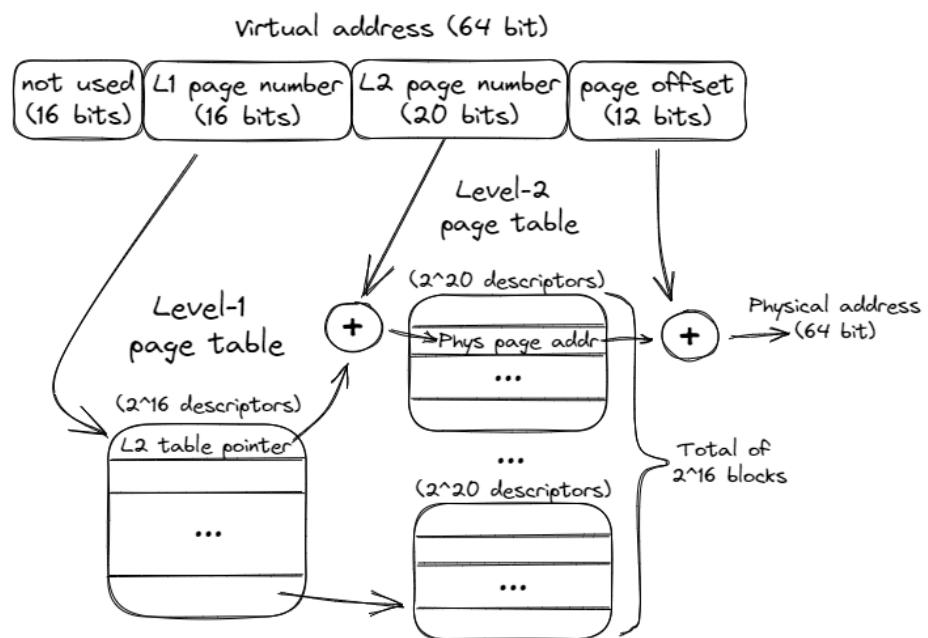


Figure 20: 2 级页表示例

位 32..47 作为索引到 1 级页表，也称为“页表目录”。目录中的每个描述符都指向 2^{16} 个 2 级表块之一。一旦找到合适的 L2 块，我们就使用位 12..31 来找到物理页地址。将其与页偏移量（位 0..11）连接起来，我们就得到了物理地址，可以用来从 DRAM 检索数据。

页表的确切格式由 CPU 决定，原因将在接下来的几个段落中讨论。因此，页表组织的变化仅限于 CPU 支持的内容。如今，通常可以看到 4 级和 5 级页表。现代 CPU 支持具有 48 位指针（256 TB 总内存）的 4 级页表和具有 57 位指针（128 PB 总内存）的 5 级页表。

将页表分为多个级别不会改变可寻址内存的总量。但是，嵌套方法不需要将整个页表存储为连续数组，也不分配没有描述符的块。这节省了内存空间，但增加了遍历页表的开销。

无法提供物理地址映射称为“页面错误”。如果请求的页面无效或当前不在主内存中，就会发生这种情况。两个最常见的原因是：1) 操作系统承诺分配一个页面，但尚未用物理页面支持它，以及 2) 访问的页面被换出到磁盘并且当前没有存储在 RAM 中。

3.7.1 地址翻译缓冲区 (TLB)

在分层页表中搜索可能代价高昂，需要遍历层次结构，可能进行多次间接访问。这种遍历通常称为“页行走”。为了减少地址翻译时间，CPU 支持一种称为地址翻译缓冲区 (TLB) 的硬件结构来缓存最近使用的翻译。类似于普通缓存，TLB 通常设计为 L1 ITLB（指令）、L1 DTLB（数据）的层次结构，然后是共享的（指令和数据）L2 STLB。为了降低内存访问延迟，TLB 和缓存查找并行发生，因为数据缓存使用虚拟地址操作，不需要预先进行地址翻译。

TLB 层次结构为相对较大的内存空间保留翻译。但是，TLB 未命中可能会非常昂贵。为了加快对 TLB 未命中处理，CPU 具有一个称为“硬件页行走器”的机制。这样的单元可以通过发出所需的指令来遍历页表，直接在硬件中执行页行走，而不会中断内核。这就是页表格式由 CPU 决定，操作系统必须遵守的原因。高端处理器有多个硬件页行走器，可以同时处理多个 TLB 未命中。然而，即使使用了现代 CPU 提供的所有加速，TLB 未命中仍然会为许多应用程序造成性能瓶颈。

3.7.2 大页

使用较小的页面大小可以更有效地管理可用内存并减少碎片化。然而，缺点是它需要更多的页表条目来覆盖相同的内存区域。考虑两种页面大小：4KB（x86 上的默认大小）和 2MB 的“大页”大小。对于处理 10MB 数据的应用程序，在第一种情况下需要 2560 个条目，而如果将地址空间映射到巨大页面，只需要 5 个条目。这些在 Linux 上称为“Huge Pages”，FreeBSD 上称为“Super Pages”，Windows 上称为“Large Pages”，但它们都表示同一个意思。在本书的其余部分，我们将它们称为 Huge Pages。

指向 Huge Page 中数据的地址示例如图 21 所示。与默认页面大小一样，使用 Huge Pages 时的确切地址格式由硬件决定，但幸运的是，我们作为程序员通常不必担心这一点。

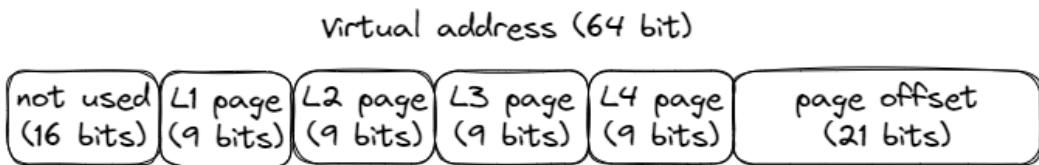


Figure 21: 指向 2MB 页面中数据的虚拟地址

使用 Huge Pages 可以大大减少对 TLB 层次结构的压力，因为需要的 TLB 条目更少。它大大增加了 TLB 命中率。我们将在 Section 7.4 和 Section 10.6 中讨论如何使用 Huge Pages 减少 TLB 未命中率。使用 Huge Pages 的缺点是内存碎片化，并且在某些情况下，由于操作系统更难管理大量内存块并确保有效利用可用内存，非确定性页面分配延迟会

增加。要在运行时满足 2MB Huge Page 分配请求，操作系统需要找到 2MB 的连续块。如果找不到，操作系统需要重组页面，从而导致更长的分配延迟。

3.8 现代 CPU 设计

为了看到我们在本章讨论的所有概念如何在实践中使用，让我们来看看英特尔第 12 代酷睿处理器 Goldencove 的实现，该处理器于 2021 年上市。该核心被用作 Alderlake 和 Sapphire Rapids 平台中的 P 核心。图 @fig:Goldencove_diag 显示了 Goldencove 核心的模块图。请注意，本节仅描述了单个核心，而不是整个处理器。因此，我们将跳过关于频率、核心数量、L3 缓存、核心互连、内存延迟和带宽以及其他内容的讨论。

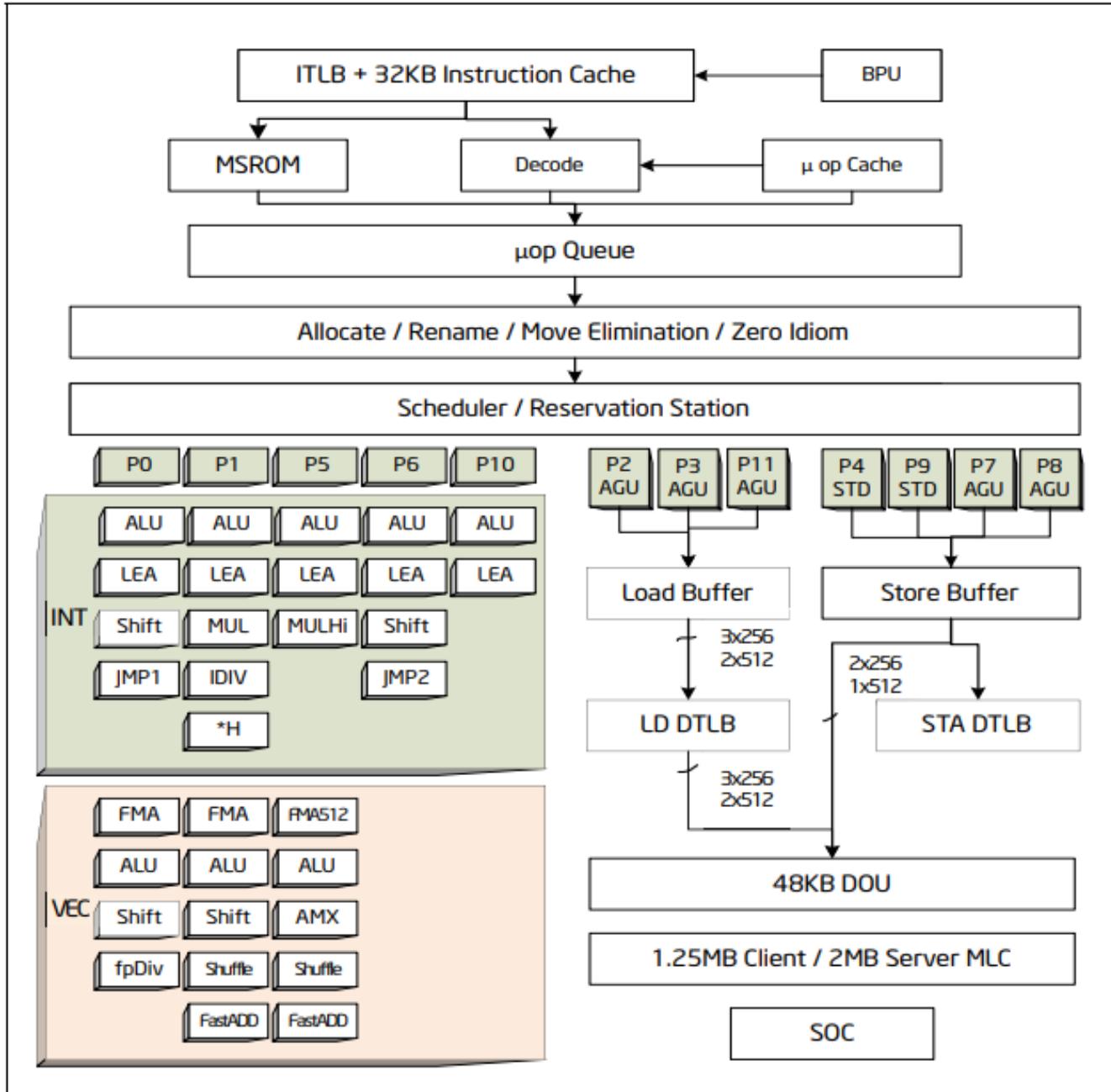


Figure 22: Intel GoldenCove 微架构 CPU 核心的模块图。© 图片来源 [Intel, 2023b]。

该核心分为一个按顺序执行的前端，负责从内存中提取和解码 x86 指令为 μ ops，以及一个 6 宽度的超标量、乱序执

行的后端。Goldencove 核心支持 2 路 SMT。它有一个 32KB 的一级指令缓存 (L1 I-cache)，和一个 48KB 的一级数据缓存 (L1 D-cache)。L1 缓存由统一的 1.25MB (服务器芯片中为 2MB) L2 缓存支持。L1 和 L2 缓存对每个核心是私有的。在本节末尾，我们还将查看 TLB 层次结构。

3.8.1 CPU 前端

CPU 前端由一些数据结构组成，用于从内存中提取和解码指令。其主要目的是向 CPU 后端提供准备好的指令，后者负责实际执行指令。

技术上讲，指令提取是执行指令的第一阶段。但一旦程序达到稳定状态，分支预测单元 (BPU) 就会引导 CPU 前端的工作。这就是从 BPU 到指令缓存的箭头的原因。BPU 预测所有分支指令的方向，并根据这个预测引导下一个指令提取。

BPU 的核心是一个包含 12K 条目的分支目标缓冲区 (BTB)，其中包含有关分支及其目标的信息。这些信息被预测算法使用。每个周期，BPU 生成下一个提取地址，并将其传递给 CPU 前端。

CPU 前端每个周期从 L1 I-cache 中提取 32 字节的 x86 指令。这在两个线程之间共享，因此每个线程每隔一个周期会获得 32 字节。这些是复杂的、可变长度的 x86 指令。首先，预解码阶段通过检查指令来确定和标记可变指令的边界。在 x86 中，指令长度可以从 1 字节到 15 字节不等。该阶段还识别分支指令。预解码阶段将多达 6 条指令（也称为宏指令）移动到指令队列（图表中未显示）中，该队列在两个线程之间划分。指令队列还支持一个宏指令融合单元，它检测到两个宏指令可以融合成一个单一的微操作 (μ op)。这种优化可以节省管道中的带宽。

稍后，多达六个预解码指令每个周期从指令队列发送到解码器单元。两个 SMT 线程每个周期交替访问此接口。6 路解码器将复杂的宏操作转换为固定长度的 μ ops。解码后的 μ ops 被排队到指令解码队列 (IDQ)，在图表上标记为“ μ op 队列”。

前端的一个主要性能提升特性是解码流缓冲器 (DSB) 或 μ op 缓存。其动机是在与 L1 I-cache 并行工作的单独结构中缓存宏操作到 μ ops 的转换。当 BPU 生成一个新地址进行提取时，也会检查 DSB，以查看 μ ops 的转换是否已经在 DSB 中可用。频繁发生的宏操作会命中 DSB，管道将避免为 32 字节的捆绑重复执行昂贵的预解码和解码操作。DSB 每个周期可以提供八个 μ ops，并且最多可以容纳 4K 个条目。

一些非常复杂的指令可能需要比解码器处理的 μ ops 更多。这些指令的 μ ops 来自微码顺序器 (MSROM)。这些指令的示例包括用于字符串操作、加密、同步等的 HW 操作支持。此外，MSROM 保留了处理异常情况的微码操作，例如分支预测失败（需要管道刷新）、浮点辅助（例如，当指令与非规范化的浮点值进行操作时）等。MSROM 每个周期可以向 IDQ 推送最多 4 个 μ ops。

指令解码队列 (IDQ) 提供了顺序 CPU 前端和乱序 CPU 后端之间的接口。IDQ 按顺序排列 μ ops，并且每个逻辑处理器在单线程模式下可以容纳 144 个 μ ops，在 SMT 活跃时每个线程可以容纳 72 个 μ ops。这是顺序 CPU 前端结束并且乱序 CPU 后端开始的地方。

3.8.2 CPU 后端

CPU 后端采用乱序执行引擎执行指令并存储结果。CPU 后端的核心是 512 条目的重排序缓冲区 (ROB)。该单元在图表中被称为“分配/重命名”。它有几个作用。首先，它提供寄存器重命名。只有 16 个通用整数寄存器和 32 个矢量/SIMD 体系结构寄存器，但是物理寄存器的数量要多得多。⁴²物理寄存器位于称为物理寄存器文件 (PRF) 的结构中。从体系结构可见寄存器到物理寄存器的映射保存在寄存器别名表 (RAT) 中。

其次，ROB 分配执行资源。当一条指令进入 ROB 时，将分配一个新条目，并为其分配资源，主要是一个执行单元和目标物理寄存器。ROB 每个周期可以分配多达 6 个 μ ops。

⁴² 大约有 300 个物理通用寄存器 (GPRs) 和类似数量的矢量寄存器。实际寄存器数量未公开。

第三，ROB 跟踪推测执行。当一条指令完成其执行时，其状态会更新，并且会保留在那里，直到前面的指令也完成。之所以这样做，是因为指令总是按程序顺序退役。一旦一条指令退役，其 ROB 条目将被释放，并且指令的结果变得可见。退役阶段比分配阶段更宽：ROB 每个周期可以退役 8 条指令。

处理器处理特定操作的方式有一定的规范，通常称为惯用法，这些操作不需要或执行成本较低。处理器识别这种情况并允许它们比常规指令运行得更快。以下是一些这种情况：

- 清零：为了将零赋给一个寄存器，编译器通常使用 XOR / PXOR / XORPS / XORPD 指令，例如 XOR RAX, RAX，而不是等效的 MOV RAX, 0x0 指令，因为 XOR 编码使用的编码字节更少。这种清零惯用法不像其他常规指令一样执行，并在 CPU 前端解决，这样可以节省执行资源。该指令后来会像通常一样退役。
- 移动消除：与前一个类似，寄存器到寄存器的 mov 操作，例如 MOV RAX, RBX，以零周期延迟执行。
- NOP 指令：NOP 通常用于填充或对齐目的。它只是被标记为已完成，而不会被分配到保留站。
- 其他旁路：CPU 架构师还优化了某些算术操作。例如，任何数乘以一将始终产生相同的数。除以一的任何数也是如此。任何数乘以零总是得到零，等等。某些 CPU 可以在运行时识别这种情况，并以比常规乘法或除法更短的延迟执行它们。

“调度器/保留站”（RS）是跟踪给定 μ op 的所有资源可用性并在准备就绪时将 μ op 分派到分配端口的结构。当一条指令进入 RS 时，调度器开始跟踪其数据依赖关系。一旦所有源操作数可用，RS 尝试将 μ op 分派到空闲的执行端口。RS 的条目比 ROB 少。它每个周期最多可以分派 6 个 μ ops。

如图 @fig:Goldencove_diag 所示，有 12 个执行端口：

- 端口 0、1、5、6 和 10 提供所有整数（INT）以及浮点和矢量（VEC/FP）操作。分派到这些端口的指令不需要内存操作。
- 端口 2、3 和 11 用于地址生成（AGU）和加载操作。
- 端口 4 和 9 用于存储操作（STD）。
- 端口 7 和 8 用于地址生成。

分派的算术操作可以进入 INT 或 VEC/FP 执行端口。整数和矢量/FP 寄存器堆栈位于不同位置。从 INT 堆栈到 VEC/FP 以及反之的操作（例如，转换、提取或插入）会带来额外的惩罚。

3.8.3 Load-Store Unit

Goldencove 核心每个周期最多可以执行三次加载和两次存储操作。一旦加载或存储操作离开调度器，加载-存储（LS）单元负责访问数据并将其保存在寄存器中。LS 单元有一个加载队列（LDQ，标记为“加载缓冲区”）和一个存储队列（STQ，标记为“存储缓冲区”），它们的大小未公开⁴³。LDQ 和 STQ 都在调度器分派时接收操作。

当有内存加载请求时，LS 使用虚拟地址查询 L1 缓存，并在 TLB 中查找物理地址转换。这两个操作同时启动。L1 D-cache 的大小为 48KB。如果两个操作都命中，则加载将数据传递给整数单元或浮点单元，并离开 LDQ。类似地，存储将数据写入数据缓存并退出 STQ。

如果发生 L1 未命中，硬件将启动对（私有）L2 缓存标签的查询。L2 缓存有两种变体：客户端为 1.25MB，服务器处理器为 2MB。在查询 L2 缓存时，将分配一个 64 字节宽的填充缓冲区条目（FB），该条目将保留一旦到达的缓存行。Goldencove 核心有 16 个填充缓冲区。为了降低延迟，与 L2 缓存查找并行进行 L3 缓存的推测性查询。

如果两个加载访问相同的缓存行，它们将命中相同的 FB。这样的两个加载将被“粘合”在一起，只会启动一个内存请求。LS 单元动态重新排序操作，支持旧加载操作绕过旧加载和旧非冲突存储操作绕过旧加载。此外，LS 单元在存在包含加载的所有字节的较旧存储并且存储的数据已生成并在存储队列中可用时，支持存储到加载的转发。

如果确认了 L2 未命中，加载将继续等待 L3 缓存的结果，这会产生更高的延迟。从那时起，请求离开核心并进入“不核心”，这是您在性能分析工具中经常看到的术语。来自核心的未完成的未命中在超级队列（SQ，图表上未显示）中

⁴³ LDQ 和 STQ 的大小未公开，但人们已经测量过分别为 192 和 114 条目。

被跟踪，该队列可以跟踪多达 48 个未核心请求。在 L3 未命中的情况下，处理器开始设置内存访问。进一步的细节超出了本章的范围。

当发生存储时，在一般情况下，要修改一个内存位置，处理器需要加载完整的缓存行，对其进行更改，然后将其写回内存。如果要写入的地址不在缓存中，则需要执行与加载类似的机制将数据带入。在将数据写入缓存层之前，存储不能完成。

当然，存储操作也有一些优化。首先，如果我们处理的是一个存储或多个相邻存储（也称为流存储），这些存储修改了整个缓存行，则无需首先读取数据，因为所有字节都将被覆盖。因此，处理器将尝试组合写入以填充整个缓存行。如果成功，根本不需要内存读取操作。

其次，写入组合使多个存储组装在一起，并作为一个单元写入缓存层次结构。因此，如果多个存储修改同一缓存行，则只会向内存子系统发出一个内存写入请求。现代处理器具有称为存储缓冲区的数据结构，该结构尝试合并存储。存储指令将数据从寄存器复制到存储缓冲区。从那里，它可以写入 L1 缓存，或者它可以与其他存储组合到同一缓存行。存储缓冲区的容量有限，因此它只能暂时保存对缓存行的部分写入的请求。然而，在数据坐在存储缓冲区等待写入时，其他加载指令可以直接从存储缓冲区读取数据（存储到加载的转发）。

最后，如果我们在覆盖数据之前读取数据，缓存行通常会保留在缓存中，替换其他行。通过使用非临时存储，可以改变这种行为，这是一种特殊的 CPU 指令，不会保留修改后的行在缓存中。在我们知道一旦更改数据就不再需要数据的情况下，非临时存储有助于更有效地利用缓存空间。非临时存储通过不驱逐其他可能很快需要的数据来帮助更有效地利用缓存空间。

3.8.4 TLB 层次结构

回想一下 Section 3.7.1，虚拟地址到物理地址的转换被缓存在 TLB 中。Golden Cove 的 TLB 层次结构如图 @fig:GLC_TLB 所示。与常规数据缓存类似，它有两个级别，其中级别 1 分

别为指令 (ITLB) 和数据 (DTLB) 有单独的实例。L1 ITLB 有 256 个条目，用于常规的 4KB 页面，覆盖 $256 * 4KB$ 等于 1MB 的内存空间，而 L1 DTLB 有 96 个条目，覆盖 384KB。

第二级别的层次结构 (STLB) 缓存了指令和数据的转换。这是一个更大的存储，用于在 L1 TLB 中未命中的请求提供服务。L2 STLB 可以容纳 2048 个最近的数据和指令页面地址转换，覆盖总共 8MB 的内存空间。对于 2MB 的大页面，可用的条目较少：L1 ITLB 有 32 个条目，L1 DTLB 有 32 个条目，而 L2 STLB 只能使用 1024 个条目，这些条目也是共享的常规 4KB 页面。

如果在 TLB 层次结构中找不到转换，则必须通过“行走”内核页表来检索。有一种机制可以加速这种情况，称为硬件页行走器。回想一下，页表是一个根据子表构建的基数树，其中子表的每个条目都包含指向树下一级的指针。

加速页行走过程的关键要素是一组页结构缓存⁴⁴，它缓存了页表结构中的热点条目。对于 4 级页表，我们有最低有效的十二位 (11:0) 用于页面偏移 (未转换)，并且页面编号的位 47:12。虽然 TLB 中的每个条目都是一个完整的单独转换，但页结构缓存仅覆盖地址的上 3 级 (位 47:21)。其思想是减少在 TLB 未命中的情况下所需的加载数量。例如，如果我们在地址的第 1 级和第 2 级找到了一个转换 (位 47:30)，则我们只需要执行剩下的 2 个加载。

Goldencove 微体系结构有四个专用页行走器，允许它同时处理 4 个页面行走。在 TLB 未命中的情况下，这些硬件单元将向内存子系统发出所需的加载，并使用新条目填充 TLB 层次结构。由页行走器生成的页表加载可以在 L1、L2 或 L3 缓存命中（详细信息未公开）。最后，页行走器可以预测未来的 TLB 未命中，并在未命中实际发生之前进行推测性页面行走以更新 TLB 条目。

Goldencove 的规格未公开两个 SMT 线程之间资源共享的方式。但是一般来说，为了提高这些资源的动态利用，缓存、TLB 和执行单元是完全共享的。另一方面，用于在主要管道阶段之间分段指令的缓冲区要么被复制，要么被分区。这些缓冲区包括 IDQ、ROB、RAT、RS、LDQ 和 STQ。PRF 也是复制的。

⁴⁴ AMD 的等效物称为 Page Walk Caches。

3.9 性能监控单元

每个现代 CPU 都提供了监控性能的设施，这些设施被合并到了性能监控单元（PMU）中。该单元集成了帮助开发人员分析其应用程序性能的功能。一个现代 Intel CPU 中的 PMU 示例如图 @fig:PMU 所示。大多数现代 PMU 都有一组性能监控计数器（PMC），可用于收集程序执行过程中发生的各种性能事件。稍后在 Section 5.3 中，我们将讨论如何使用 PMC 进行性能分析。此外，PMU 还具有其他增强性能分析的功能，如 LBR、PEBS 和 PT，Section ??专门讨论了这个话题。

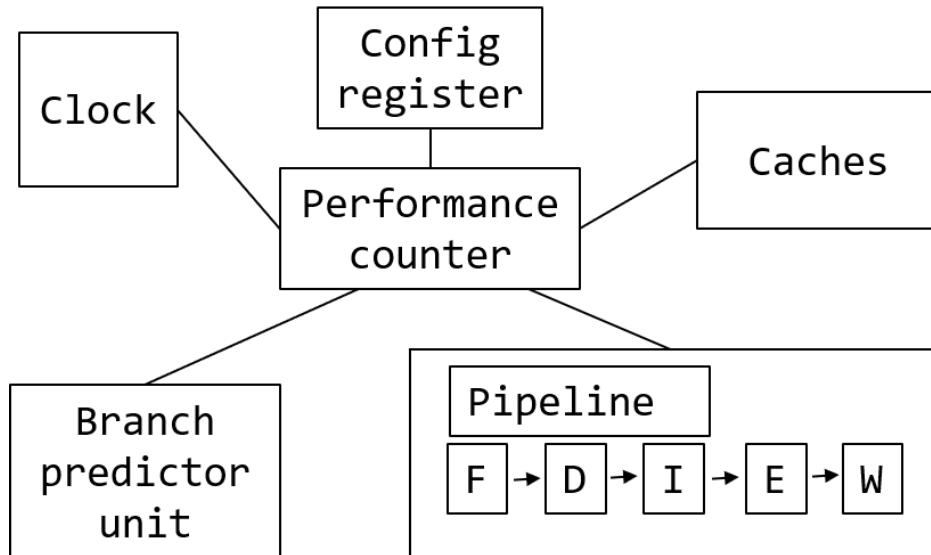


Figure 23: 现代 Intel CPU 的性能监控单元。

随着每一代 CPU 设计的演进，它们的 PMU 也在发展。在 Linux 上，可以使用 `cpuid` 命令确定 CPU 中 PMU 的版本，如 Listing ?? 所示。类似的信息可以通过检查 `dmesg` 命令的输出从内核消息缓冲区中提取。每个 Intel PMU 版本的特性，以及与上一个版本的变化，可以在 [Intel, 2023b, Volume 3B, Chapter 20] 中找到。

查询您的 PMU 的列表：

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

3.9.1 性能监控计数器

如果我们想象一下对处理器的简化视图，它可能看起来像图 @fig:PMC 所示的样子。正如我们在本章前面讨论过的，现代 CPU 具有缓存、分支预测器、执行流水线和其他单元。当连接到多个单元时，PMC 可以从中收集有趣的统计

信息。例如，它可以计算经过了多少个时钟周期，执行了多少条指令，在此期间发生了多少缓存失效或分支预测错误等性能事件。

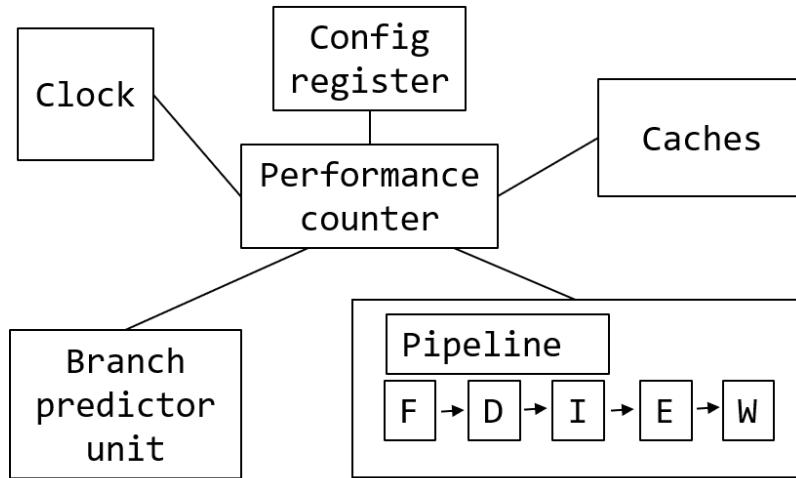


Figure 24: 带有性能监控计数器的 CPU 的简化视图。

通常，PMC 的宽度为 48 位，这使得分析工具能够在不中断程序执行的情况下运行很长时间⁴⁵。性能计数器是作为模型特定寄存器（MSR）实现的硬件寄存器。这意味着计数器的数量和它们的宽度会因型号而异，您不能依赖于在您的 CPU 中的相同数量的计数器。您应该始终首先查询它，例如使用 `cpuid` 等工具。PMC 可以通过 `RDMSR` 和 `WRMSR` 指令访问，这些指令只能在内核空间执行。幸运的是，只有当您是性能分析工具的开发人员时，才需要关注这一点，例如 Linux `perf` 或 Intel Vtune 分析器。这些工具处理了编程 PMC 的所有复杂性。

当工程师分析他们的应用程序时，他们通常会收集已执行的指令数和经过的周期数。这就是为什么一些 PMU 具有专用的 PMC 用于收集这些事件的原因。固定计数器始终在 CPU 核心内部测量相同的事物。对于可编程计数器，用户可以选择要测量的内容。

例如，在 Intel Skylake 架构（PMU 版本 4，参见 Listing ??），每个物理核心有三个固定计数器和八个可编程计数器。这三个固定计数器分别设置为计算核心时钟、参考时钟和已退休指令（有关这些指标的更多详细信息，请参见 Chapter 4）。AMD Zen4 和 ARM Neoverse V1 核心每个处理器核心支持 6 个可编程性能监控计数器，没有固定计数器。

PMU 提供了 100 多种可用于监控的事件并不罕见。图 @fig:PMU 仅显示了现代 Intel CPU 上供监控的性能事件的一小部分。不难注意到可用 PMC 的数量远远小于性能事件的数量。无法同时计算所有事件，但是分析工具通过在程序执行期间在性能事件组之间进行复用来解决此问题（参见 Section 5.3.3）。

- 对于 Intel CPU，可以在 [Intel, 2023b, Volume 3B, Chapter 20] 中找到性能事件的完整列表，或者在 perfmon-events.intel.com 上找到。
- AMD 并不为每个 AMD 处理器发布性能监控事件的列表。感兴趣的读者可以在 Linux `perf` 源代码中找到一些信息[代码⁴⁶](#)。此外，您可以使用 AMD uProf 命令行工具列出可用于监控的性能事件。有关 AMD 性能计数器的一般信息，请参见 [AMD, 2023, 13.2 Performance Monitoring Counters]。
- 对于 ARM 芯片，性能事件没有如此明确定义。供应商按照 ARM 架构实

现核心，但性能事件的含义和支持的事件在很大程度上变化。对于由 ARM 自己设计的 ARM Neoverse V1 处理器，性能事件的列表可以在 [Arm, 2022b] 中找到。

⁴⁵ 当 PMC 的值溢出时，必须中断程序的执行。然后，软件应该保存溢出的事实。我们稍后将详细讨论它。

⁴⁶ AMD 核心的 Linux 源代码 - <https://github.com/torvalds/linux/blob/master/arch/x86/events/amd/core.c>

问题和练习

1. 描述流水线处理、乱序执行和投机执行。
2. 寄存器重命名如何加速执行？
3. 描述空间局部性和时间局部性。
4. 在大多数现代处理器中，缓存行的大小是多少？
5. 构成 CPU 前端和后端的组件是什么？
6. 4 级页表的组织结构是什么？什么是页错误？
7. x86 和 ARM 架构中的默认页面大小是多少？
8. TLB（转换后备缓冲区）扮演了什么角色？

章节总结

- 指令集架构 (ISA) 是软件和硬件之间的基本契约。ISA 是计算机的抽象模型，定义了可用操作和数据类型、寄存器集、内存寻址等内容。你可以以许多不同的方式实现特定的 ISA。例如，你可以设计一个“小”核心，重视功耗效率，也可以设计一个“大”核心，追求高性能。
- 实现细节封装在术语“CPU 微体系结构”中。这个主题已经被成千上万的计算机科学家研究了很长时间。多年来，许多聪明的想法在大众市场的 CPU 中得到了实现。最值得注意的是流水线处理、乱序执行、超标量引擎、投机执行和 SIMD 处理器。所有这些技术都有助于利用指令级并行性 (ILP) 并提高单线程性能。
- 与单线程性能并行的是，硬件设计者开始推动多线程性能。绝大多数面向客户端的现代设备都有一个包含多个核心的处理器。一些处理器通过同时多线程 (SMT) 技术将可观察的 CPU 核心数量翻倍。SMT 允许多个软件线程同时在同一个物理核心上运行，共享资源。在这个方向上的一种更近期的技术被称为“混合”处理器，它将不同类型的核心组合在一个单一的封装中，以更好地支持各种工作负载。
- 现代计算机中的内存层次结构包括几个级别的缓存，反映了速度访问与大小之间的不同权衡。L1 缓存通常最近接近核心，速度快但容量小。L3/LLC 缓存速度较慢但容量较大。DDR 是大多数平台集成的主要 DRAM 技术。DRAM 模块在排数和存储器宽度方面有所不同，这可能会对系统性能产生轻微影响。处理器可能具有多个内存通道，以同时访问多个 DRAM 模块。
- 虚拟内存是与 CPU 上运行的所有进程共享物理内存的机制。程序在其访问中使用虚拟地址，这些地址被转换为物理地址。内存空间分割为页面。x86 的默认页面大小为 4KB，ARM 为 16KB。只有页面地址会被转换，页面内的偏移量保持原样。操作系统将转换保留在页表中，页表实现为基数树。有硬件功能可提高地址转换的性能：主要是转换后备缓冲区 (TLB) 和硬件页行走器。此外，开发人员可以利用巨大页面在某些情况下减轻地址转换的成本（见 Section 7.4）。
- 我们查看了英特尔最近的 GoldenCove 微体系结构的设计。逻辑上，核心分为前端和后端。前端包括分支预测单元 (BPU)、L1-I 缓存、指令提取和解码逻辑以及 IDQ，它将指令提供给 CPU 后端。后端包括乱序执行引擎、执行单元、加载存储单元、L1-D 缓存和 TLB 层次结构。
- 现代处理器具有性能监视功能，这些功能封装在性能监视单元 (PMU) 中。该单元围绕性能监视计数器 (PMC) 的概念构建，使得可以观察程序运行时发生的特定事件，例如缓存未命中和分支预测失败。

4 性能分析中的术语和指标

像许多工程学科一样，性能分析在使用特殊术语和指标方面非常重要。对于初学者来说，查看由分析工具如 Linux `perf` 或 Intel VTune Profiler 生成的性能分析文件可能会感到非常困难。这些工具涉及许多复杂的术语和指标，但如果您打算进行任何严肃的性能工程工作，这些内容是“必须了解”的。

既然我们提到了 Linux `perf`，让我们简要介绍一下这个工具，因为我们在本章和后续章节中有许多使用它的示例。Linux `perf` 是一个性能分析器，您可以使用它来查找程序中的热点，收集各种低级 CPU 性能事件，分析调用堆栈等等。我们将在整本书中广泛使用 Linux `perf`，因为它是最流行的性能分析工具之一。我们偏爱展示 Linux `perf` 的另一个原因是因为它是开源软件，这使得热衷的读者可以探索现代分析工具内部发生的机制。这对于学习本书中提出的概念特别有用，因为基于 GUI 的工具（如 Intel® VTune™ Profiler）往往隐藏所有复杂性。我们将在第 7 章中对 Linux `perf` 进行更详细的概述。

本章是对性能分析中使用的基本术语和指标的简要介绍。我们将首先定义诸如已退役/执行指令、IPC/CPI、 μ ops、核心/参考时钟、缓存失效和分支误预测等基本概念。然后，我们将看到如何测量系统的内存延迟和带宽，并介绍一些更高级的指标。最后，我们将对四种行业工作负载进行基准测试，并查看收集到的指标。

4.1 已退役 vs. 已执行指令

现代处理器通常执行的指令数量比程序流程所需的要多。这是因为一些指令是以推测性的方式执行的，如 Section 3.3.3 中所讨论的。对于大多数指令，CPU 在结果可用时提交，所有先前的指令都已退役。但对于以推测性方式执行的指令，CPU 会保留它们的结果，而不立即提交它们的结果。当推测结果被证明是正确时，CPU 会解除此类指令的阻塞并正常进行。但当推测结果被证明是错误时，CPU 会丢弃推测指令所做的所有更改，并不会退役它们。因此，CPU 处理的指令可以被执行但不一定被退役。考虑到这一点，我们通常可以预期执行的指令数量高于已退役的指令数量。

但有一个例外。某些指令被识别为惯用语，并且在没有实际执行的情况下被解析。其中一些示例是 NOP、移动消除和清零，如 Section 3.8.2 中所讨论的。这些指令不需要执行单元，但仍然被退役。因此，从理论上讲，可能存在已退役指令数量高于已执行指令数量的情况。

大多数现代处理器都有一个性能监视计数器 (PMC)，用于收集已退役指令的数量。虽然没有性能事件来收集已执行的指令，但有一种方法可以收集已执行和已退役的微操作，我们很快将会看到。可以通过运行以下命令轻松获取已退役指令的数量，使用 Linux `perf`：

```
$ perf stat -e instructions ./a.exe
 2173414  instructions # 0.80  insn per cycle
# 或者简单地执行:
$ perf stat ./a.exe
```

4.2 CPU 利用率

CPU 利用率是在一段时间内 CPU 处于忙碌状态的百分比。从技术上讲，当 CPU 不运行内核的 `idle` 线程时，CPU 被认为是被利用的。

$$CPU\ Utilization = \frac{CPU_CLK_UNHALTED.REF_TSC}{TSC},$$

其中，`CPU_CLK_UNHALTED.REF_TSC` 计算了核心处于非停顿状态时的参考周期数，`TSC` 代表时间戳计数器（在 Section 2.6 中讨论过），它始终在运行。

如果 CPU 利用率低，通常意味着应用程序性能较差，因为 CPU 浪费了一部分时间。然而，高 CPU 利用率并不总是好性能的指标。这仅仅是系统正在进行一些工作的迹象，但并不表示正在做什么：即使 CPU 由于等待内存访问而被阻塞，它仍然可能被高度利用。在多线程环境中，线程在等待资源继续进行时也可以自旋。稍后在 Section 11.2 中，我们将讨论并行效率指标，特别是“有效 CPU 利用率”，该指标过滤了自旋时间。

Linux perf 会自动计算系统上所有 CPU 的 CPU 利用率：

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

4.3 CPI 和 IPC

这两个是两个基本指标，分别代表：

- 每条指令周期数 (CPI) - 平均执行一条指令所需的周期数。

$$IPC = \frac{INST_RETIRED.ANY}{CPU_CLK_UNHALTED.THREAD},$$

其中，`INST_RETIRED.ANY` 计算已完成指令的数量，`CPU_CLK_UNHALTED.THREAD` 计算线程不在中止状态时的核心周期数。

- 每周期指令数 (IPC) - 平均每个周期完成的指令数。

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

使用哪一个取决于个人喜好。本书的主要作者更喜欢使用 IPC，因为它更容易比较。使用 IPC，我们希望每个周期尽可能多地执行指令，因此 IPC 越高越好。使用 CPI 则相反：我们希望每个指令的周期越少越好，所以 CPI 越低越好。使用“越高越好”的指标进行比较更简单，因为您不必每次都进行心理反转。在本书的其余部分，我们将主要使用 IPC，但再次申明，使用 CPI 也没有错。

IPC 和 CPU 时钟频率之间的关系非常有趣。从广义上讲，性能= 工作 / 时间，我们可以将工作表示为指令数，时间表示为秒。程序运行的秒数是 总周期/ 频率：

$$\text{性能} = \frac{\text{指令} \times \text{频率}}{\text{周期}} = IPC \times \text{频率}$$

正如我们看到的，性能与 IPC 和频率成正比。如果我们增加这两个指标中的任何一个，程序的性能就会提高。

从基准测试的角度来看，IPC 和频率是两个独立的指标。我们见过许多工程师错误地将它们混为一谈，认为如果增加频率，IPC 也会上升。但事实并非如此，IPC 将保持不变。如果您将处理器的时钟设置为 1 GHz 而不是 5Ghz，您仍然会拥有相同的 IPC。这非常令人困惑，尤其是因为 IPC 与 CPU 时钟密切相关。频率只告诉单个时钟的快慢，而 IPC 不考虑时钟变化的速度，它计算每个周期完成的工作量。因此，从基准测试的角度来看，IPC 完全取决于处理器的设计，与频率无关。乱序内核通常具有比顺序内核更高的 IPC。当您增加 CPU 缓存的大小或改进分支预测时，IPC 通常会上升。

现在，如果您问硬件架构师，他们肯定会告诉您 IPC 和频率之间存在依赖关系。从 CPU 设计的角度来看，您可以故意降低处理器时钟频率，这将使每个周期更长，并且可以在每个周期中塞入更多工作。最终，您将获得更高的 IPC

但更低的频率。硬件供应商以不同的方式处理性能公式。例如，英特尔和 AMD 芯片通常具有非常高的频率，最近的英特尔 13900KS 处理器开箱即用即可提供 6Ghz 的睿频频率，无需超频。另一方面，Apple M1/M2 芯片的频率较低，但通过更高的 IPC 进行补偿。较低的频率有助于降低功耗。另一方面，更高的 IPC 通常需要更复杂的设计、更多的晶体管和更大的芯片尺寸。我们这里不会讨论所有设计权衡，因为这涉及另一个主题。我们将在 Section 11.9 中讨论 IPC 和频率的未来发展。

IPC 对于评估硬件和软件效率都非常有用。硬件工程师使用此指标比较不同供应商的不同 CPU 代和 CPU。由于 IPC 是衡量微架构性能优劣的指标，因此工程师和媒体会使用它来表达最新 CPU 比上一代性能的提升。但要进行公平的比较，您需要在相同频率下运行这两个系统。

IPC 也是评估软件的有用指标。它可以让您直观地了解应用程序中的指令如何快速地穿过 CPU 管道。稍后，您将在本章中看到几个具有不同 IPC 的生产应用程序。内存密集型应用程序通常以低 IPC (0-1) 为特征，而计算密集型工作负载往往具有高 IPC (4-6)。

Linux perf 用户可以通过运行以下命令测量其工作负载的 IPC:

```
$ perf stat -e cycles,instructions -- a.exe
2369632 cycles
1725916 instructions # 0.73 insn per cycle
# 或更简单地:
$ perf stat ./a.exe
```

4.4 微操作

具有 x86 架构的微处理器将复杂的 CISC 类指令转换为简单的 RISC 类微操作，缩写为 μ ops 或 μ ops。例如，像 ADD rax , rbx 这样的简单加法指令只会生成一个 μ op，而更复杂的指令比如 ADD rax , $[mem]$ 可能生成两个：一个用于从 mem 内存位置读取到临时（未命名）寄存器，另一个用于将其添加到 rax 寄存器。指令 ADD $[mem]$, rax 会生成三个 μ ops：一个用于从内存读取，一个用于相加，一个用于将结果写回内存。

将指令分割成微操作的主要优点是 μ ops 可以执行：

- 乱序：考虑 PUSH rbx 指令，它将栈指针减少 8 字节，然后将源操作数存储在栈顶。假设在解码后 PUSH rbx 被“破解”成两个依赖的微操作：

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

通常，函数序言通过使用多个 PUSH 指令保存多个寄存器。在我们的例子中，下一个 PUSH 指令可以在前一个 PUSH 指令的 SUB μ op 完成后开始执行，而不必等待现在可以异步执行的 STORE μ op。

- 并行：考虑 HADDPD $xmm1$, $xmm2$ 指令，它将在 $xmm1$ 和 $xmm2$ 中对两个双精度浮点数进行求和（减少），并将两个结果存储在 $xmm1$ 中，如下所示：

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

微代码化此指令的一种方法是执行以下操作：1) 减少 $xmm2$ 并将结果存储在 $xmm_tmp1[63:0]$ 中，2) 减少 $xmm1$ 并将结果存储在 $xmm_tmp2[63:0]$ 中，3) 将 xmm_tmp1 和 xmm_tmp2 合并到 $xmm1$ 中。总共三个 μ ops。请注意，步骤 1) 和 2) 是独立的，因此可以并行完成。

尽管我们刚刚讨论了如何将指令分割成更小的部分，但有时 μ ops 也可以融合在一起。现代 CPU 中有两种类型的融合：

- 微融合: 融合来自同一机器指令的 μ ops。微融合只能应用于两种类型的组合: 内存写操作和读改操作。例如:

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

这条指令中有两个 μ ops: 1) 读取内存位置 mem, 2) 将其添加到 eax。使用微融合, 在解码步骤中将两个 μ ops 融合成一个。

- 宏融合: 融合来自不同机器指令的 μ ops。在某些情况下, 解码器可以将算术或逻辑指令与 subsequent 条件跳转指令融合成单个计算和分支 μ op。例如:

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

使用宏融合, 将来自 DEC 和 JNZ 指令的两个 μ ops 融合成一个。

微融合和宏融合都可以节省从解码到退休的所有管道阶段的带宽。融合操作在重新排序缓冲区 (ROB) 中共享单个条目。当一个融合的 μ op 只使用一个条目时, ROB 的容量得到更好的利用。这样的一个融合的 ROB 条目稍后会分派到两个不同的执行端口, 但作为单个单元再次退休。读者可以 [Fog, 2012] 中了解更多关于 μ op 融合的信息。

要收集应用程序发出的、执行的和退休的 μ ops 数量, 您可以使用 Linux perf, 如下所示:

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

指令被分解成微操作的方式可能会随着 CPU 世代的不同而有所差异。通常, 用于一条指令的 μ ops 数量越少, 意味着硬件对其支持越好, 并且可能具有更低的延迟和更高的吞吐量。对于最新的 Intel 和 AMD CPU, 绝大多数指令都会生成恰好一个 μ op。有关最近微架构中 x86 指令的延迟、吞吐量、端口使用情况和 μ ops 数量, 可以参考 uops.info: <https://uops.info/table.html>⁴⁷ 网站。

4.5 管道槽

另一个一些性能工具使用的重要指标是管道槽 (pipeline slot) 的概念。管道槽代表处理一个微操作所需的硬件资源。图 25 展示了一个每周期有 4 个分配槽的 CPU 的执行管道。这意味着核心可以在每个周期将执行资源 (重命名的源和目标寄存器、执行端口、ROB 条目等) 分配给 4 个新的微操作。这样的处理器通常被称为 4 宽机器。在图中连续的六个周期中, 只利用了一半可用槽位。从微架构的角度来看, 执行此类代码的效率只有 50%。

英特尔的 Skylake 和 AMD Zen3 内核具有 4 宽分配。英特尔的 SunnyCove 微架构采用 5 宽设计。截至 2023 年, 最新的 Goldencove 和 Zen4 架构都采用 6 宽分配。Apple M1 的设计没有官方披露, 但测得为 8 宽。⁴⁸

管道槽是自顶向下微架构分析 (见 Section ??) 的核心指标之一。例如, 前端受限和后端受限指标由于各种瓶颈而表示为未使用的管道槽的百分比。

4.6 核心周期与参考周期

大多数 CPU 都使用时钟信号来控制它们的顺序操作。时钟信号由外部发生器产生, 每秒提供一致数量的脉冲。时钟脉冲的频率决定了 CPU 执行指令的速率。因此, 时钟越快, CPU 每秒执行的指令就越多。

⁴⁷ 指令延迟和吞吐量 - <https://uops.info/table.html>

⁴⁸ Apple 微架构研究 - <https://dougallj.github.io/applecpu/firestorm.html>



Figure 25: 4 宽 CPU 的管道图

$$\text{频率} = \frac{\text{时钟周期数}}{\text{时间}}$$

大多数现代 CPU，包括英特尔和 AMD 的 CPU，没有固定的运行频率。相反，它们实现了动态频率缩放，在英特尔的 CPU 中称为 Turbo Boost，在 AMD 处理器中称为 Turbo Core。它使 CPU 能够动态增加和减少频率。降低频率可以减少功耗，但会牺牲性能，增加频率可以提高性能，但会牺牲节能。

核心时钟周期计数器计算的是 CPU 核心实际运行的频率下的时钟周期数，而不是外部时钟（参考周期）。让我们看一下在 Skylake i7-6000 处理器上运行单线程应用程序的实验，它的基础频率为 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
```

指标ref-cycles统计的周期数是如果没有频率缩放的情况下。设置的外部时钟频率为 100 MHz，如果我们乘以时钟倍频，我们将得到处理器的基础频率。Skylake i7-6000 处理器的时钟倍频为 34：这意味着对于每个外部脉冲，当 CPU 运行在基础频率上时，它执行 34 个内部周期。

指标cycles统计的是真实的 CPU 周期数，即考虑了频率缩放。使用上述公式，我们可以确认平均运行频率为 $43340884632 \text{ 个周期} / 10.899 \text{ 秒} = 3.97 \text{ GHz}$ 。当您比较两个版本的小段代码的性能时，以时钟周期计时比以纳秒计时更好，因为您避免了时钟频率上下波动的问题。

4.7 缓存失效

正如在 Section 3.6 中讨论的那样，任何在特定级别的缓存中缺失的内存请求都必须由更高级别的缓存或 DRAM 进行服务。这意味着这种内存访问的延迟会显著增加。内存子系统组件的典型延迟如表 3 所示。还有一个交互视图⁴⁹，可视化了现代系统中不同操作的延迟。性能会受到严重影响，特别是当内存请求在最后一级缓存（LLC）中丢失并一直到达主存时。英特尔 ® Memory Latency Checker⁵⁰ (MLC) 是用于测量内存延迟和带宽以及它们随系统负载增加而变化的工具。MLC 对于建立测试系统的基准和进行性能分析非常有用。当我们讨论 Section 4.10 中的内存延迟和带宽时，我们将使用这个工具。

⁴⁹ 交互延迟 - https://colin-scott.github.io/personal_website/research/interactive_latency.html

⁵⁰ Memory Latency Checker - <https://www.intel.com/software/mlc>

Table 3: x86 平台内存子系统的典型延迟。

内存层次结构组件 延迟	(周期/时间)
L1 缓存 4	个周期 (~ 1 纳秒)
L2 缓存 1	0-25 个周期 (5-10 纳秒)
L3 缓存 ~	40 个周期 (20 纳秒)
主内存 2	00 个周期以上 (100 纳秒)

缓存失效可能会发生在指令和数据上。根据 Top-down Microarchitecture Analysis (见 Section ??)，指令缓存 (I-cache) 失效被定义为前端停顿，而数据缓存 (D-cache) 失效被定义为后端停顿。指令缓存失效在 CPU 流水线的早期阶段 (指令获取阶段) 发生。数据缓存失效则发生在后期，即指令执行阶段。

Linux perf 用户可以通过运行以下命令来收集 L1 缓存失效的数量：

```
$ 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
```

以上是针对 L1 数据缓存和填充缓冲区的所有加载操作的细分。加载操作可能命中已分配的填充缓冲区 (fb_hit)，或者命中 L1 缓存 (l1_hit)，或者两者都未命中 (l1_miss)，因此 all_loads = fb_hit + l1_hit + l1_miss。我们可以看到，只有 3.5% 的所有加载操作在 L1 缓存中未命中，因此 L1 命中率为 96.5%。

我们可以进一步分析 L1 数据缺失并分析 L2 缓存行为，方法是运行：

```
$ 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
```

从这个例子中，我们可以看到，在 L1 D-cache 中缺失的加载操作中有 37% 也在 L2 缓存中缺失，因此 L2 命中率为 63%。以类似的方式，可以对 L3 缓存进行细分。

4.8 错误预测的分支

现代 CPU 尝试预测分支指令的结果 (是否被执行)。例如，当处理器看到这样的代码时：

```
dec eax
jz .zero
# eax 不为 0
...
zero:
# eax 为 0
```

在上面的例子中，`jz` 指令是一个分支。现代 CPU 架构试图预测每个分支的结果以提高性能。这被称为“推测执行”，我们在 Section 3.3.3 中讨论过。处理器会假设，例如，分支不会被执行，并执行相应于 `eax` 不为 0 的情况的代码。然而，如果猜测错误，这被称为“分支预测错误”，CPU 需要撤销最近所做的所有推测工作。

错误预测的分支通常会导致 10 到 20 个时钟周期的惩罚。首先，根据错误预测获取和执行的所有指令都需要从流水线中清除。之后，一些缓冲区可能需要清理，以恢复从坏的推测开始的状态。最后，流水线需要等待确定正确的分支目标地址，这会导致额外的执行延迟。

Linux perf 用户可以通过运行以下命令来检查分支预测错误的数量：

```
$ perf stat -e branches,branch-misses -- a.exe
 358209  branches
 14026  branch-misses # 3.92% 的分支错误预测率
# 或者简单地执行:
$ perf stat -- a.exe
```

4.9 性能指标

除了本章前面讨论的性能事件外，性能工程师经常使用基于原始事件的指标。表4显示了针对英特尔第 12 代 Goldencove 架构的一系列指标，包括描述和公式。该列表并非详尽无遗，但显示了最重要的指标。有关英特尔 CPU 及其公式的完整指标列表可在[TMA_metrics.xlsx](#)中找到。Section 4.11 展示了如何在实践中使用性能指标。

Table 4: 英特尔 Goldencove 架构的一系列次要指标及其描述和公式（非详尽）。

指标名称	描述	公式
L1MPKI	每千条已退休需求负载指令的 L1 缓存真未命中数量	$1000 * \text{MEM_LOAD_RETIRED.L1_MISS_PS} / \text{INST_RETIRED.ANY}$
L2MPKI	每千条已退休需求负载指令的 L2 缓存真未命中数量	$1000 * \text{MEM_LOAD_RETIRED.L2_MISS_PS} / \text{INST_RETIRED.ANY}$
L3MPKI	每千条已退休需求负载指令的 L3 缓存真未命中数量	$1000 * \text{MEM_LOAD_RETIRED.L3_MISS_PS} / \text{INST_RETIRED.ANY}$
BranchMispr.Ratio	所有分支的误判率	$\text{BR_MISP_RETIRED.ALL_BRANCHES} / \text{BR_INST_RETIRED.ALL_BRANCHES}$
CodeSTLBMPKIS	TLB(2 级 TLB) 代码推测性未命中每千条指令	$1000 * \text{ITLB_MISSES.WALK_COMPLETED} / \text{INST_RETIRED.ANY}$
LoadSTLBMPKIS	TLB 数据加载推测性未命中每千条指令	$1000 * \text{DTLB_LOAD_MISSES.WALK_COMPLETED} / \text{INST_RETIRED.ANY}$
StoreSTLBMPKI	STLB 数据存储推测性未命中每千条指令	$1000 * \text{DTLB_STORE_MISSES.WALK_COMPLETED} / \text{INST_RETIRED.ANY}$
LoadMissRealLatency	L1 数据缓存未命中需求负载操作的实际平均延迟（核心周期）	$\text{L1D_PEND_MISS.PENDING} / \text{MEM_LOAD_COMPLETED.L1_MISS_ANY}$
ILP	每内核指令级并行性（当存在执行时，平均执行的 μ ops 数量）	$\text{UOPS_EXECUTED.THREAD} / \text{UOPS_EXECUTED.CORE_CYCLES_GE_1}$ 如果启用 SMT，则除以 2

指标名称	描述	公式
MLP	每线程内存级并行性（当至少有一个这样的未命中时 L1 未命中需求负载的平均数量）	L1D_PEND_MISS.PENDING , /L1D_PEND_MISS.PENDING_CYCLES
DRAMBWUse	平均外部内存带宽使用（读写 GB/秒）	(64*(UNC_M_CAS_COUNT.RD + UNC_M_CAS_COUNT.WR) /1GB)/时间
IpCall	近调用每条指令	INST_RETIREANY /BR_INST_RETIRE.NEAR_CALL
IpBranch 每	分支指令 IN	ST_RETIREANY /BR_INST_RETIRE.ALL_BRANCHES
IpLoad	每加载指令	INST_RETIREANY /MEM_INST_RETIRE.ALL_LOADS_PS
IpStore	每存储指令	INST_RETIREANY /MEM_INST_RETIRE.ALL_STORES_PS
IpMisp	每非推测分支误判指令	INST_RETIREANY /BR_MISP_RETIRE.ALL_BRANCHES
IpFLOP	每浮点 (FP) 操作指令	请参阅 TMA_metrics.xlsx
IpArithScalarSP	每标量单精度 FP 算术指令	INST_RETIREANY /FP_ARITH_INST_RETIRE.SCALAR_SINGLE
IpArithScalarDP	每标量双精度 FP 算术指令	INST_RETIREANY /FP_ARITH_INST_RETIRE.SCALAR_DOUBLE
IpArithAVX128	每 FP 算术 AVX128128 位指令	INST_RETIREANY/(FP_ARITH_INST_RETIRE.128B_PACKED_DOUBLE + FP_ARITH_INST_RETIRE.128B_PACKED_SINGLE)
IpArithAVX256	每 FP 算术 AVX256256 位指令	INST_RETIREANY/(FP_ARITH_INST_RETIRE.256B_PACKED_DOUBLE + FP_ARITH_INST_RETIRE.256B_PACKED_SINGLE)
IpSWPF	每个软件预取指令 (of any type)	INST_RETIREANY /SW_PREFETCH_ACCESS.T0:u0xF

关于这些指标的一些说明。首先，ILP 和 MLP 指标并不代表应用程序的理论最大值；而是衡量在给定机器上应用程序的实际 ILP 和 MLP。在具有无限资源的理想机器上，这些数字会更高。其次，除了“DRAM BW Use”和“Load Miss Real Latency”之外的所有指标都是分数；我们可以对每个指标进行相当直接的推理，以确定特定指标是高还是低。但要理解“DRAM BW Use”和“Load Miss Real Latency”指标的意义，我们需要将其放在一个上下文中。对于前者，我们想知道一个程序是否饱和了内存带宽。后者为你提供了缓存失效的平均成本，但这本身是无用的，除非你知道缓存层次结构中每个组件的延迟。我们将在下一节讨论如何找出缓存延迟和峰值内存带宽。

一些工具可以自动报告性能指标。如果没有，你总可以手动计算这些指标，因为你知道公式和必须收集的相应性能事件。表4提供了针对英特尔 Goldencove 架构的公式，但只要底层的性能事件可用，你就可以在另一个平台上构建类似的指标。

4.10 内存延迟和带宽

在现代环境中，低效的内存访问通常是主要的性能瓶颈。因此，处理器从内存子系统中获取单个字节的速度（延迟）以及每秒可以获取多少字节（带宽）是决定应用程序性能的关键因素之一。这两个方面在各种场景中都很重要，我们稍后将看到一些示例。在本节中，我们将专注于测量内存子系统组件的峰值性能。

在 x86 平台上，可以成为有用工具之一的是英特尔内存延迟检查器（MLC），⁵¹它在 Windows 和 Linux 上都可以免费使用。MLC 可以使用不同的访问模式和负载来测量缓存和内存的延迟和带宽。在基于 ARM 的系统上没有类似的工具，但是用户可以从源代码中下载并构建内存延迟和带宽基准测试。这类项目的示例包括 lmbench⁵²，bandwidth⁵³ 和 Stream。⁵⁴

我们只关注一个子集指标，即空闲读取延迟和读取带宽。让我们从读取延迟开始。空闲表示在进行测量时，系统处于空闲状态。这将为我们提供从内存系统组件获取数据所需的最长时间，但是当系统被其他“内存消耗量大”的应用程序加载时，此延迟会增加，因为在各个点上可能会有更多的资源排队。MLC 通过进行相关加载（也称为指针追踪）来测量空闲延迟。一个测量线程分配一个非常大的缓冲区，并对其进行初始化，以便缓冲区内的每个（64 字节）缓存行包含指向该缓冲区内另一个非相邻缓存行的指针。通过适当调整缓冲区的大小，我们可以确保几乎所有的加载都命中某个级别的缓存或主存。

我们的测试系统是一台英特尔 Alderlake 主机，配备 Core i7-1260P CPU 和 16GB DDR4 @ 2400 MT/s 双通道内存。该处理器有 4 个 P（性能）超线程核心和 8 个 E（高效）核心。每个 P 核心有 48KB 的 L1 数据缓存和 1.25MB 的 L2 缓存。每个 E 核心有 32KB 的 L1 数据缓存，而四个 E 核心组成一个集群，可以访问共享的 2MB L2 缓存。系统中的所有核心都由 18MB 的 L3 缓存支持。如果我们使用一个 10MB 的缓冲区，我们可以确保对该缓冲区的重复访问会在 L2 中未命中，但在 L3 中命中。以下是示例 mlc 命令：

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

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

选项 --idle_latency 测量读取延迟而不加载系统。MLC 具有 --loaded_latency 选项，用于在由其他线程生成的内存流量存在时测量延迟。选项 -c0 将测量线程固定在逻辑 CPU 0 上，该 CPU 位于 P 核心上。选项 -L 启用大页以限制我们的测量中的 TLB 效应。选项 -b10m 告诉 MLC 使用 10MB 缓冲区，在我们的系统上可以放在 L3 缓存中。

图 26 显示了 L1、L2 和 L3 缓存的读取延迟。图中有四个不同的区域。从 1KB 到 48KB 缓冲区大小

的左侧的第一个区域对应于 L1d 缓存，该缓存是每个物理核心私有的。我们可以观察到 E 核心的延迟为 0.9ns，而 P 核心稍高为 1.1ns。此外，我们可以使用此图来确认缓存大小。请注意，当缓冲区大小超过 32KB 时，E 核心的延迟开始上升，但是在 48KB 之前 E 核心的延迟保持不变。这证实了 E 核心的 L1d 缓存大小为 32KB，而 P 核心的 L1d 缓存大小为 48KB。

第二个区域显示 L2 缓存延迟，E 核的延迟几乎是 P 核的两倍（5.9ns vs. 3.2ns）。对于 P 核，延迟在我们超过 1.25MB 缓冲区大小后会增加，这是预期的。但我们期望 E 核的延迟保持不变，直到 2MB，但在我们的测量中没有发生这种情况。

⁵¹ Intel MLC 工具 - <https://www.intel.com/content/www/us/en/download/736633/intel-memory-latency-checker-intel-mlc.html>

⁵² lmbench - <https://sourceforge.net/projects/lmbench>

⁵³ Zack Smith 的内存带宽基准测试 - <https://zsmith.co/bandwidth.php>

⁵⁴ Stream - <https://github.com/jeffhammond/STREAM>

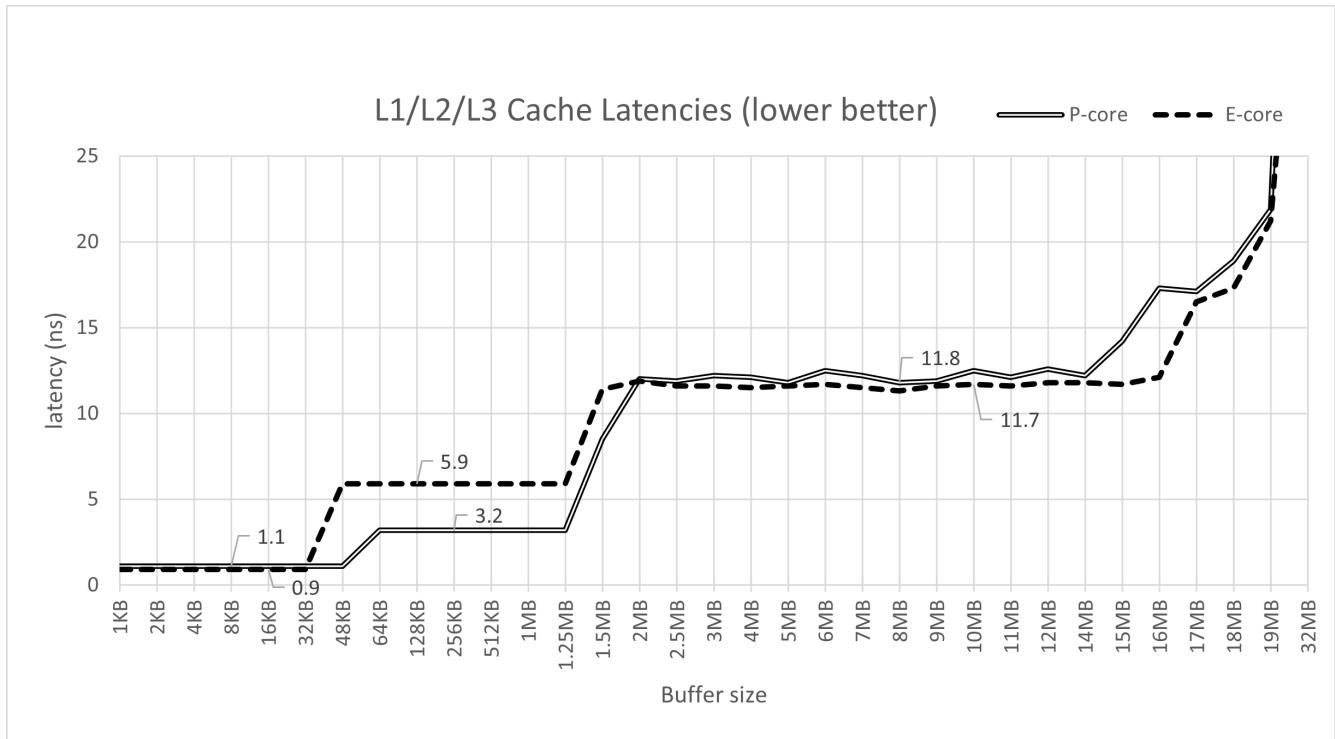


Figure 26: 在 Intel Core i7-1260P 上使用 mlc 工具测量的 L1/L2/L3 缓存读取延迟，启用了大页。

第三个区域从 2MB 到 14MB 对应于 L3 缓存延迟，对于两种类型的内核都大约为 12ns。系统中所有内核共享的 L3 缓存的总大小为 18MB。有趣的是，我们从 15MB 开始看到一些意想不到的动态，而不是 18MB。这很可能是因为一些访问错过了 L3，需要去主内存。

第四个区域对应于内存延迟，图表上只显示了其开始部分。当我们越过 18MB 的边界时，延迟会急剧上升，并在 E 核心的 24MB 和 P 核心的 64MB 处开始趋于稳定。使用更大的缓冲区大小为 500MB 时，E 核心的访问延迟为 45ns，P 核心为 90ns。这测量了内存延迟，因为几乎没有加载会命中 L3 缓存。

使用类似的技术，我们可以测量内存层次结构的各个组件的带宽。为了测量带宽，MLC 执行的加载请求不会被任何后续指令使用。这允许 MLC 生成可能的最大带宽。MLC 在每个配置的逻辑处理器上生成一个软件线程。每个线程访问的地址是独立的，线程之间没有数据共享。与延迟实验一样，线程使用的缓冲区大小确定了 MLC 是在测量 L1/L2/L3 缓存带宽还是内存带宽。

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

这里的新选项是 `-k`，它指定了用于测量的 CPU 编号列表。`-Y` 选项告诉 MLC 使用 AVX2 加载，即每次加载 32 字节。MLC 使用不同的读写比例来测量带宽，但在下图中，我们只显示了全部读取带宽，因为它可以让我们对内存带宽的

峰值有一个直观的了解。但其他比例也可能很重要。我们在使用 Intel MLC 测量的系统的组合延迟和带宽数字如图 27 所示。

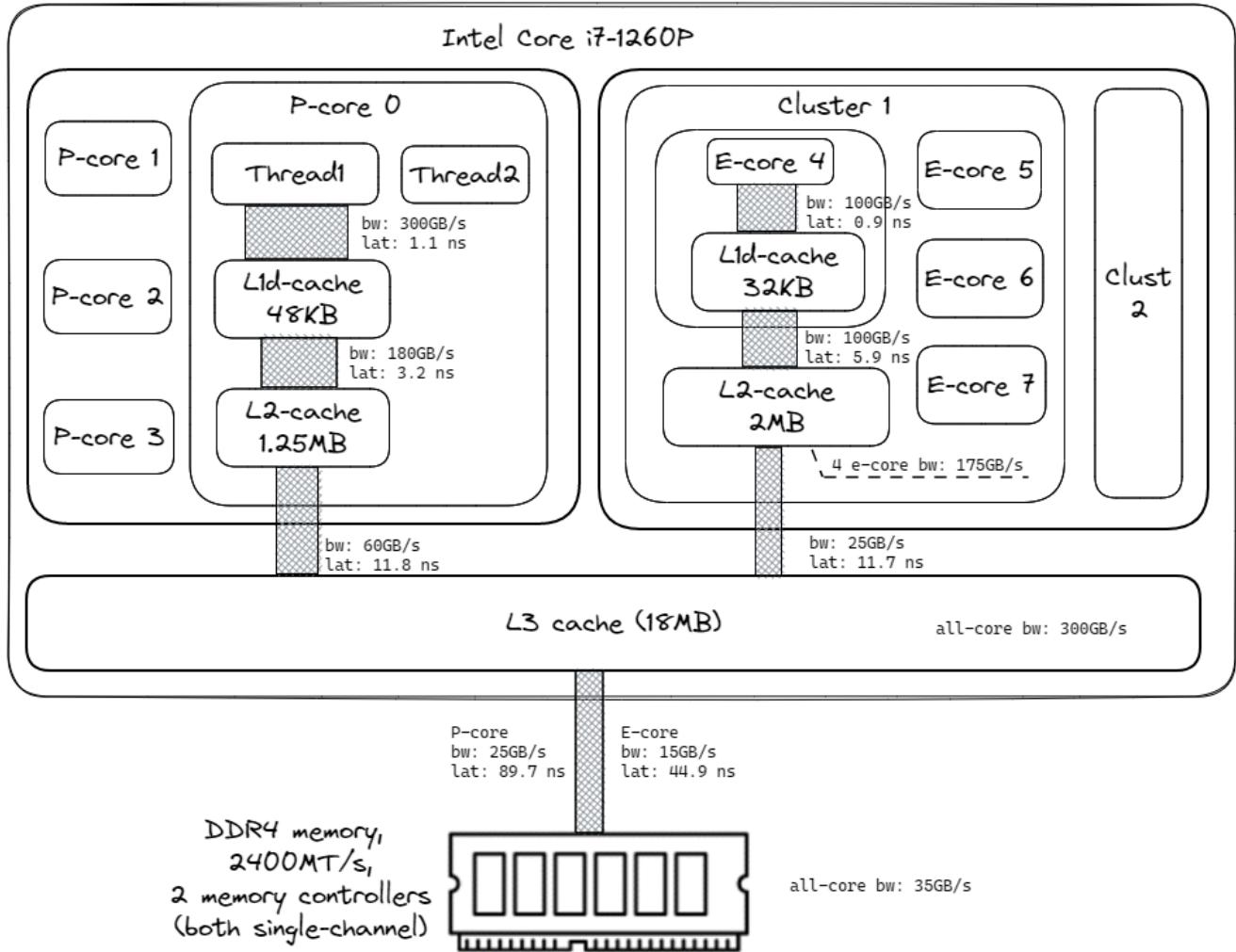


Figure 27: Intel Core i7-1260P 内存层次结构的块图和外部 DDR4 内存。

核心可以从较低级别的缓存（如 L1 和 L2）中获得比从共享的 L3 缓存或主内存中更高的带宽。共享缓存（如 L3 和 E 核心 L2）相当好地扩展，可以同时为多个核心提供请求。例如，单个 E 核心 L2 的带宽为 100GB/s。使用来自同一集群的两个 E 核心，我测量了 140GB/s，三个 E 核心为 165GB/s，而所有四个 E 核心可以从共享 L2 中获得 175GB/s。对于 L3 缓存也是如此，单个 P 核心的带宽为 60GB/s，而单个 E 核心只有 25GB/s。但是当所有核心都被使用时，L3 缓存可以维持 300GB/s 的带宽。

请注意，我们用纳秒测量延迟，用 GB/s 测量带宽，因此它们还取决于核心运行的频率。在各种情况下，观察到的数字可能不同。例如，假设当仅在系统上以最大睿频运行时，P 核心的 L1 延迟为 X，L1 带宽为 Y。当系统负载满时，我们可能观察到这些指标分别变为 $1.25X$ 和 $0.75Y$ 。为了减轻频率效应，与其使用纳秒，延迟和度量可以使用核心周期来表示，并归一化为一些样本频率，比如 3Ghz。

了解计算机的主要特征是评估程序如何利用可用资源的基本方法。当我们讨论 Roofline 性能模型时，我们将在 Section 5.6 返回到这个主题。如果您经常在单个平台上分析性能，最好记住内存层次结构的各个组件的延迟和带宽，或者将它们随时备查。这有助于建立对测试系统的心理模型，将有助于您进一步的性能分析，正如您将在接下来看到的那样。

4.11 案例研究：分析四个基准测试的性能指标

在本章中讨论的所有内容综合起来，我们运行了来自不同领域的四个基准测试，并计算了它们的性能指标。首先，让我们介绍这些基准测试。

1. Blender 3.4 - 一个开源的 3D 创建和建模软件项目。这个测试是使用 Blender 的 Cycles 性能进行的，使用了 BMW27 混合文件。使用了所有的硬件线程。URL: <https://download.blender.org/release>。命令行: `./blender -b bmw27_cpu.blend --noaudio --enable-autoexec -o output.test -x 1 -F JPEG -f 1`。
2. Stockfish 15 - 一个先进的开源国际象棋引擎。这个测试是一个内置的 stockfish 基准测试。只使用了一个硬件线程。URL: <https://stockfishchess.org>。命令行: `./stockfish bench 128 1 24 default depth`。
3. Clang 15 自我构建 - 这个测试使用 clang 15 从源代码构建 clang 15 编译器。使用了所有的硬件线程。URL: <https://www.llvm.org>。命令行: `ninja -j16 clang`。
4. CloverLeaf 2018 - 一个拉格朗日-欧拉流体动力学基准测试。使用了所有的硬件线程。这个测试使用了 `clover_bm.in` 输入文件（问题 5）。URL: <http://uk-mac.github.io/CloverLeaf>。命令行: `./clover_leaf`。

为了进行这个练习，我们在具有以下特征的机器上运行了所有四个基准测试：

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

为了收集性能指标，我们使用了 `toplev.py` 脚本，它是 `pmu-tools`⁵⁵ 的一部分，由 Andi Kleen 编写：

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

表 [5] 提供了我们四个基准测试的性能指标的并排比较。通过查看这些指标，我们可以了解这些工作负载的性质。在收集性能配置文件并深入研究这些应用程序的代码之前，我们可以对这些基准测试做出一些假设。

- **Blender**。工作在 P 核心和 E 核心之间相当均匀分配，两种核心类型的 IPC 都相当不错。每千条指令的缓存未命中次数相当低（见 L*MPKI）。分支误预测对性能有所影响：分支误预测比例指标为 2%；我们每 610 条指令就有一个误预测（见 IpMispredict 指标），这个数值并不糟糕，但也不完美。TLB 并不是瓶颈，因为我们在 STLB 中很少发生未命中。我们忽略加载未命中延迟指标，因为缓存未命中的数量非常低。ILP 相当高。Goldencove 是一个 6 宽度的体系结构；ILP 为 3.67 意味着算法几乎每个周期利用了核心资源的 2/3。内存带宽需求很低，只有 1.58 GB/s，远低于该机器的理论最大值。从 Ip* 指标来看，我们可以得知 Blender 是一个浮点算法（见 IpFLOP 指标），其中有很大一部分是矢量化的浮点运算（见 IpArith AVX128）。但是，算法的某些部分也是非矢量化的标量浮点单精度指令（IpArith Scal SP）。另外，请注意每 90 条指令就会有一个明确的软件内存预取（IpSWPF）；我们期望在 Blender 的源代码中看到这些提示。结论：Blender 的性能受到 FP 计算的限制，偶尔会出现分支误预测。
- **Stockfish**。我们只使用了一个硬件线程运行它，因此 E 核心上没有任何工作，这是预期的。L1 缓存未命中的数量相对较高，但大部分都包含在 L2 和 L3 缓存中。分支误预测比例很高；我们每 215 条指令就会付出一次误预测的代价。我们可以估计，我们每 215 (指令) / 1.80 (IPC) = 120 个周期就会发生一次误预测，这是非常频繁的。与 Blender 的推理类似，我们可以说 TLB 和 DRAM 带宽对 Stockfish 不构成问题。进一步分析，我们发现工作负载中几乎没有 FP 操作。结论：Stockfish 是一个整数计算工作负载，受分支误预测的影响很大。
- **Clang 15 自我构建**。C++ 代码编译是一项性能特性非常平坦的任务，即没有大的热点。通常，您会发现运行时间归因于许多不同的函数。我们首先注意到的是，P 核心比 E 核心多做了 68% 的工作，并且 IPC 要好 42%。但是 P 核心和 E 核心的 IPC 都很低。乍一看，L*MPKI 指标看起来并不令人担忧；然而，结合加载未命中实际

⁵⁵ pmu-tools - <https://github.com/andikleen/pmu-tools>

延迟 (LdMissLat, 以核心时钟表示), 我们可以看到缓存未命中的平均成本相当高 (~77 个周期)。现在, 当我们查看*STLB_MPKI 指标时, 我们注意到与我们测试的任何其他基准测试都存在实质性差异。这是由于 Clang 编译器 (以及其他编译器) 的另一个方面: 二进制文件的大小相对较大 (超过 100 MB)。代码不断跳转到远处的位置, 导致 TLB 子系统的压力很大。正如您所看到的, 该问题存在于指令 (请参阅Code stlb MPKI) 和数据 (请参阅Ld stlb MPKI) 之间。让我们继续进行分析。DRAM 带宽使用率高于前两个基准测试, 但仍然没有达到我们平台的最大内存带宽的一半 (约为 25 GB/s)。我们关注的另一个问题是每次调用的指令数量非常少 (IpCall): 每个函数调用只有约 41 条指令。不幸的是, 这是编译代码库的本质: 它有数千个小函数。编译器需要更积极地内联所有这些函数和包装器。然而, 我们怀疑与进行函数调用相关的性能开销仍然是 Clang 编译器的一个问题。此外, 人们可以注意到高 ipBranch 和 IpMispredict 指标。对于 Clang 编译, 每五条指令中就有一条分支, 大约每 35 条分支中就有一条误预测。几乎没有 FP 或矢量指令, 但这并不奇怪。结论: Clang 具有庞大的代码库, 平坦的性能配置文件, 许多小函数和“分支”代码; 性能受到数据缓存和 TLB 未命中以及分支误预测的影响。

- CloverLeaf。与之前一样, 我们从分析指令和核心周期开始。P 核心和 E 核心完成的工作量大致相同, 但 P 核心需要更长的时间来完成这项工作, 导致 P 核心上的一个逻辑线程的 IPC 比一个物理 E 核心上的 IPC 低。我们对此还没有一个很好的解释。L*MPKI 指标很高, 特别是每千条指令的 L3 未命中次数。加载未命中延迟 (LdMissLat) 超出了图表范围, 表明平均缓存未命中的价格非常高。接下来, 我们看一下 DRAM 带宽使用指标, 发现内存带宽完全饱和了。这就是问题所在: 系统中的所有核心共享同一个内存总线, 因此它们竞争访问主存, 有效地阻塞了执行。CPU 缺乏它们需要的数据。进一步说, 我们可以看到 CloverLeaf 几乎没有受到分支误预测或函数调用开销的影响。指令混合主要由 FP 双精度标量操作主导, 代码的某些部分被矢量化。结论: 多线程 CloverLeaf 受到内存带宽的限制。

Table 5: 四个基准测试的性能指标。

指标名称	核心类型	h Clang15			
		Blender	Stockfis	-selfbuild	CloverLeaf
指令数	P-核心	6.02E+12	6.59E+11	2.40E+13	1.06E+12
核心周期数	P-核心	4.31E+12	3.65E+11	3.78E+13	5.25E+12
IPC	P-核心	1.40	1.80	0.64	0.20
CPI	P-核心	0.72	0.55	1.57	4.96
指令数	E-核心	4.97E+12	0	1.43E+13	1.11E+12
核心周期数	E-核心	3.73E+12	0	3.19E+13	4.28E+12
IPC	E-核心	1.33	0	0.45	0.26
CPI	E-核心	0.75	0	2.23	3.85
L1MPKI	P-核心	3.88	21.38	6.01	13.44
L2MPKI	P-核心	0.15	1.67	1.09	3.58
L3MPKI	P-核心	0.04	0.14	0.56	3.43
分支错误率	E-核心	0.02	0.08	0.03	0.01
代码 STLB MPKI	P-核心	0	0.01	0.35	0.01
加载 STLB MPKI	P-核心	0.08	0.04	0.51	0.03
存储 STLB MPKI	P-核心	0	0.01	0.06	0.1
加载缺失延迟 (时钟)	P-核心	12.92	10.37	76.7	253.89
ILP	P-核心	3.67	3.65	2.93	2.53
MLP	P-核心	1.61	2.62	1.57	2.78

指标名称	核心类型	Blender	h Clang15		
			Stockfis	-selfbuild	CloverLeaf
DRAM 带宽 (GB/s)	全部	1.58	1.42	10.67	24.57
IpCall	全部	176.8	153.5	40.9	2,729
IpBranch	全部	9.8	10.1	5.1	18.8
IpLoad	全部	3.2	3.3	3.6	2.7
IpStore	全部	7.2	7.7	5.9	22.0
IpMispredict	全部	610.4	214.7	177.7	2,416
IpFLOP	全部	1.1	1.82E+06	286,348	1.8
IpArith	全部	4.5	7.96E+06	268,637	2.1
IpArith Scal SP	全部	22.9	4.07E+09	280,583	2.60E+09
IpArith Scal DP	全部	438.2	1.22E+07	4.65E+06	2.2
IpArith AVX128	全部	6.9	0.0	1.09E+10	1.62E+09
IpArith AVX256	全部	30.3	0.0	0.0	39.6
IpSWPF	全部	90.2	2,565	105,933	172,348

正如您从这项研究中看到的，仅仅通过查看指标就可以了解很多关于程序行为的信息。它回答了“是什么？”的问题，但没有告诉你“为什么？”。为此，您需要收集性能配置文件，我们将在以后的章节中介绍。本书的第二部分将讨论如何减轻我们分析的四个基准测试中可能出现的性能问题。

请记住，表 5 中的性能指标摘要只告诉您程序的平均行为。例如，我们可能看到 CloverLeaf 的 IPC 为 0.2，而实际上它可能永远不会以这样的 IPC 运行，相反它可能有两个持续时间相同的阶段，一个以 IPC 0.1 运行，另一个以 IPC 0.3 运行。性能工具通过为每个指标报告统计数据和平均值来解决这个问题。通常，最小值、最大值、第 95 个百分位数和方差 (stdev/avg) 足以了解分布。此外，一些工具允许绘制数据，因此您可以看到特定指标的值在程序运行期间如何变化。例如，图 28 显示了 CloverLeaf 基准测试中 IPC、L*MPKI、DRAM BW 和平均频率的动态变化。“pmu-tools”软件包可以在您添加 `--xlsx` 和 `--xchart` 选项后自动生成这些图表。

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

尽管与摘要中报告的值偏差不大，但我们可以看到工作负载并不总是稳定的。在查看 IPC 图表后，我们可以假设工作负载中没有不同的阶段，变化是由性能事件的多路复用引起的（在 Section 5.3 中讨论）。然而，这只是一个需要证实或否定的假设。可能的方法是通过以更高粒度（在本例中为 10 秒）运行收集来收集更多数据点并研究源代码。仅根据数字得出结论要小心；始终获取第二个数据源来确认您的假设。

总之，查看性能指标有助于构建关于程序中发生了什么和没有发生什么的正确思维模型。深入分析，这些数据将对您大有裨益。

问题和练习

- CPU 核心时钟和参考时钟的区别是什么？

CPU 核心时钟表示处理器内核运行的频率，单位为 GHz。它决定了处理器在每个时钟周期内可以执行的指令数量。参考时钟是系统中一个基准时钟信号，通常比核心时钟频率更低，用于同步系统中的各个组件。

- 已执行指令和已完成指令的区别是什么？

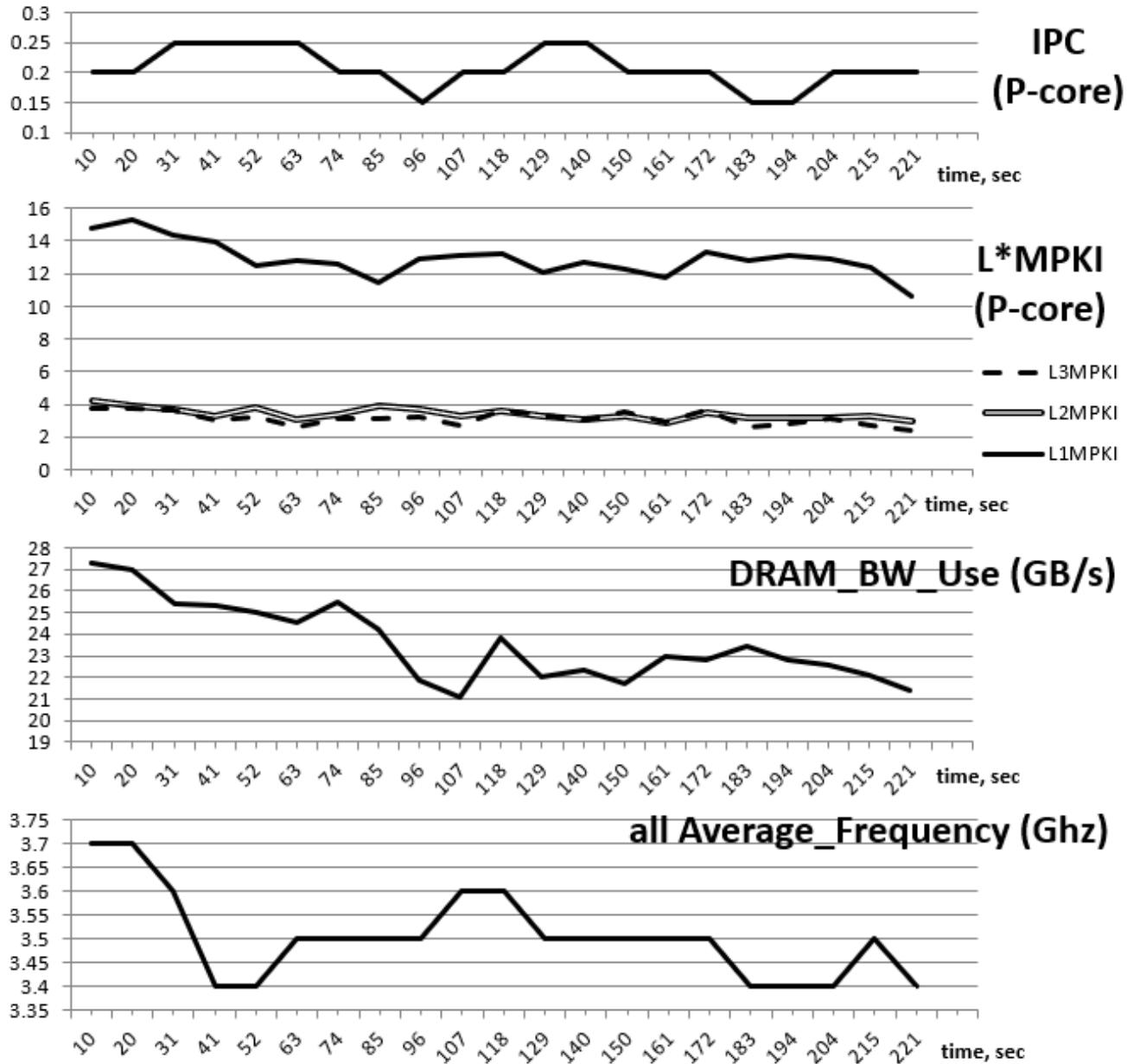


Figure 28: CloverLeaf 基准测试的一组指标图表

已执行指令是指已经进入流水线并且开始执行的指令。已完成指令是指已经成功完成所有执行阶段并写回结果的指令。已执行指令的数量可能大于已完成指令的数量，因为有些指令可能由于分支错误、数据依赖等原因无法完成。

3. 当您增加频率时，IPC 会上升、下降还是保持不变？

增加频率不一定意味着 IPC 会上升。IPC 还取决于其他因素，例如指令集架构、缓存命中率、内存带宽等。如果其他因素没有改善，增加频率只会提高时钟周期的执行速度，但并不会增加每个周期内完成的指令数量，因此 IPC 可能会保持不变甚至下降。

4. 请看表 4 中的 DRAM BW Use 公式，为什么会有个常数 64？

公式中的常数 64 可能代表了缓存行的大小，以字节为单位。DRAM 带宽使用率计算的是每秒从内存中读取或写入的数据量，除以缓存行大小是为了将单位转换为缓存行访问次数。

5. 使用 MLC、stream 或其他工具测量您开发/基准测试机器上的缓存层次结构和内存的带宽和延迟。

这道练习要求您使用性能分析工具测量计算机的缓存和内存性能。您可以使用不同的工具，例如 MLC (Memory Latency Checker)、stream 等，来测量内存带宽和延迟。

6. 运行您每天使用的应用程序。收集性能指标。有什么让你惊讶的吗？

这道练习鼓励您分析日常使用的应用程序的性能。您可以使用性能分析工具收集诸如 CPU 使用率、内存使用率、磁盘 I/O 等指标。通过分析这些指标，您可以发现应用程序性能瓶颈并进行优化。

4.12 产能规划练习

假设您是案例研究中基准测试的四个应用程序的所有者。您的公司管理层要求您为每个应用程序构建小型计算集群，主要目标是最大化性能（吞吐量）。给您的预算有限，但足以购买 1 台用于运行每个工作负载的中端服务器系统 (Mac Studio、Supermicro/Dell/HPE 服务器机架等) 或 1 台高端台式机（带超频 CPU、液体冷却、顶级 GPU、快速 DRAM），共计 4 台机器。它们可以是所有四种不同的系统。此外，您可以用这笔钱购买 3-4 台低端系统，选择权由您决定。管理层希望将每个应用程序的成本控制在 10,000 美元以下，但如果能证明开支合理，他们可以灵活调整 (10-20%)。假设 Stockfish 仍然是单线程的。再次查看四个应用程序的性能特征，并写下您将为每个工作负载购买哪些计算机部件 (CPU、内存、如有必要，还有独立 GPU)。您将优先考虑哪些规格参数？您将在哪里使用最昂贵的部件，您可以在哪里节省资金？请尽可能详细地描述，并在网上搜索确切的组件及其价格。考虑系统的所有组件：主板、磁盘驱动器、冷却解决方案、电源分配单元、机架/机箱/塔式机箱等。您将运行哪些额外的性能实验来指导您的决策？

这道练习模拟了为特定应用程序选择硬件的场景。您需要考虑应用程序的性能特征、预算限制以及可用的硬件选项。您需要权衡不同的因素，例如 CPU 速度、内存容量、存储速度等，以找到最适合应用程序需求的配置。

章节总结

- 在本章中，我们介绍了性能分析中的基本指标，例如已执行/已完成指令、CPU 利用率、IPC/CPI、微操作、流水线槽、核心/参考时钟、缓存未命中和分支预测错误。我们展示了如何使用 Linux perf 采集这些指标中的每一个。
- 对于更高级的性能分析，可以收集许多派生指标。例如，MPKI（每千条指令的未命中）、Ip*（每个函数调用、分支、加载等指令）、ILP、MLP 等。本章的案例研究展示了如何通过分析这些指标获得可操作的见解。但是，仅查看总数字就得出结论要谨慎。不要陷入“电子表格性能工程”的陷阱，即只收集性能指标而不查看代码。总是寻求第二个数据源（例如，稍后讨论的性能配置文件）来验证您的假设。
- 内存带宽和延迟是当今许多生产软件包性能的关键因素，包括人工智能、高性能计算、数据库和许多通用应用程序。内存带宽取决于 DRAM 速度 (MT/s) 和内存通道数。现代高端服务器平台拥有 8-12 个内存通道，整个系

统的带宽可达 500 GB/s，单线程模式下可达 50 GB/s。如今的内存延迟变化不大，事实上，随着新一代 DDR4 和 DDR5 的出现，它还在略微恶化。大多数现代系统的内存访问延迟都在 70-110 纳秒范围内。

5 性能分析方法

当你在进行高级优化工作时，比如将更好的算法集成到应用程序中，通常很容易判断性能是否有所提高，因为基准测试结果很明显。大幅提速，如 2 倍、3 倍等，从性能分析的角度来看相对容易。当你从程序中消除了大量计算时，你预期会在运行时间上看到明显的差异。

但也有一些情况下，你会看到执行时间的微小变化，比如 5%，而你不知道这个变化是从哪里来的。仅仅依靠计时或吞吐量测量并不能解释性能是为什么提高或下降的。在这种情况下，我们需要进行性能分析，以了解我们观察到的减速或加速的潜在原因。

性能分析类似于侦探工作。要解决性能谜团，你需要收集尽可能多的数据，然后尝试形成一个假设。一旦假设被提出，你就设计一个实验来证明或反驳它。这可能需要多次来回，直到你找到线索。就像一个优秀的侦探一样，你会尽力收集尽可能多的证据来证实或反驳你的假设。一旦你有足够的线索，你就可以对你观察到的行为做出令人信服的解释。

当你刚开始处理性能问题时，你可能只有测量数据，比如在代码变更之前和之后的数据。基于这些测量数据，你得出结论：程序变慢了 X 个百分点。如果你知道减速发生在某次提交之后，这可能已经为你提供了足够的信息来修复问题。但如果你没有良好的参考点，那么减速的可能原因就是无穷无尽的，你需要收集更多的数据。收集这些数据的最流行方法之一是对应用程序进行性能分析并查看热点。本章介绍了这个方法以及其他一些在性能工程中被证明有效的方法。

接下来的问题是：“可用的性能数据有哪些，如何收集？”堆栈的硬件和软件层都有跟踪性能事件并记录它们的设施。在这个背景下，所谓的硬件是指执行程序的 CPU，而软件是指操作系统、库、应用程序本身和用于分析的其他工具。通常，软件堆栈提供高级指标，如时间、上下文切换次数和页面错误，而 CPU 则监视缓存未命中、分支错误预测和其他与 CPU 相关的事件。根据你试图解决的问题，一些指标比其他指标更有用。因此，这并不意味着硬件指标总会给我们提供程序执行的更精确的概览。例如，一些指标，比如上下文切换次数，CPU 无法提供。性能分析工具，如 Linux Perf，可以使用来自操作系统和 CPU 的数据。

正如你可能猜到的那样，性能工程师可能会使用数百种数据源。由于本书是关于 CPU 底层性能的，我们将重点介绍收集硬件级别信息的技术。我们将介绍一些最流行的性能分析技术：代码仪器化、跟踪、特性化、采样和屋顶线模型。我们还将讨论静态性能分析技术和不涉及运行实际应用程序的编译器优化报告。

5.1 代码仪器化

也许有史以来进行性能分析的第一种方法就是代码 仪器化 (Instrumentation)⁵⁶。它是一种在程序中插入额外代码以收集特定运行时信息的技术。Listing ??展示了在函数开头插入printf语句的最简单示例，以指示该函数何时被调用。然后，运行程序并计算输出中看到“foo 被调用”的次数。也许，世界上每个程序员在其职业生涯中至少有一次这样做过。

代码仪器化

```
int foo(int x) {  
+ printf("foo被调用\n");  
// 函数体...  
}
```

⁵⁶ 测试领域内所用的 Instrumentation 术语翻译为“插桩”，这里翻译成仪器化，进行代码性能测量，书中有些地方也会翻译成“插桩”

行首的加号表示此行是添加的，不在原始代码中。通常，仪器化代码并不意味着将其推送到代码库中，而是用于收集所需的数据，然后可以丢弃。

稍微有趣一些的代码仪器化示例在 Listing ?? 中给出。在这个虚构的代码示例中，函数 `findObject` 在地图上搜索具有某些属性 `p` 的对象的坐标。函数 `findObj` 返回使用当前坐标 `c` 定位正确对象的置信度级别。如果是完全匹配，我们停止搜索循环并返回坐标。如果置信度高于 `threshold`，我们选择 `zoomIn` 以找到对象更精确的位置。否则，我们在 `searchRadius` 范围内获取新的坐标以便下次尝试搜索。

仪器化代码由两个类组成：`histogram` 和 `incrementor`。前者跟踪我们感兴趣的变量值及其出现频率，然后在程序完成后打印直方图。后者只是一个辅助类，用于将值推送到 `histogram` 对象中。它非常简单，可以快速调整以满足您的特定需求。我有一个稍微更高级的版本，通常会将其复制粘贴到我正在工作的任何项目中，然后将其删除。

代码仪器化

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

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

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

```
}
```

在这个假设情景中，我们添加了仪器化代码以了解在找到对象之前我们多频繁地`zoomIn`。变量`inc.tripCount`计算循环退出之前循环运行的次数，而变量`inc.zoomCount`计算我们减少搜索半径的次数。我们总是期望`inc.zoomCount`小于或等于`inc.tripCount`。下面是运行仪器化程序后可能观察到的输出：

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

在方括号中的第一个数字是循环的次数，第二个数字是在同一个循环中进行的`zoomIn`次数。冒号后面的数字是该特定组合的出现次数。例如，我们观察到 7 次循环迭代和 6 次`zoomIn`发生了两次，循环运行了 7 次迭代且没有`zoomIn`的情况发生了 251004 次，依此类推。然后，您可以绘制数据以进行更好的可视化，采用一些其他统计方法，但我们可以得出的主要观点是`zoomIn`并不频繁。在调用了 400k 次`findObject`的情况下，总共有 10k 次`zoomIn`调用。

本书的后续章节包含许多示例，说明了这类信息如何用于基于数据的优化。在我们的情况下，我们得出结论：`findObj`经常无法找到对象。这意味着循环的下一次迭代将尝试使用新坐标来找到对象，但搜索半径仍然相同。有了这个信息，我们可以尝试一些优化：1) 并行运行多个搜索，并在其中任何一个成功时同步；2) 为当前搜索区域预先计算某些内容，从而消除`findObj`内的重复工作；3) 编写一个软件管道，调用`getNewCoords`以生成下一组所需坐标，并从内存中预取相应的地图位置。本书的第二部分将更深入地探讨一些这样的技术。

代码仪器化在需要关于程序执行的特定知识时提供了非常详细的信息。它允许我们跟踪程序中每个变量的任何信息。在优化大型代码块时，使用这种方法通常会产生最好的见解，因为您可以使用自上而下的方法（仪器化主函数，然后逐步深入到其被调用的函数）来定位性能问题。虽然代码仪器化在小程序的情况下并不是很有帮助，但通过让开发人员观察应用程序的架构和流程，它提供了最大的价值和见解。对于与不熟悉的代码库一起工作的人来说，这种技术尤其有帮助。

值得一提的是，代码仪器化在具有许多不同组件的复杂系统中表现突出，这些组件根据输入或时间的不同而产生不同的反应。例如，在游戏中，通常有一个渲染线程、一个物理线程、一个动画线程等。对这样的大型模块进行仪器化有助于相对快速地理解哪个模块是问题的源头。因为有时，优化不仅仅是优化代码，还包括数据。例如，渲染可能太慢是因为网格未压缩，或者物理可能太慢是因为场景中的对象太多。

仪器化技术在实时场景的性能分析中被广泛使用，例如视频游戏和嵌入式开发。一些性能分析器将仪器化与其他技术（如跟踪和采样）混合在一起。我们将在 Section 6.8 中介绍其中一个混合性能分析器，名为 Tracy。

虽然在许多情况下代码仪器化是强大的，但它并不提供有关代码如何从操作系统或 CPU 的角度执行的任何信息。例如，它无法告诉您进程被调度到执行中和退出执行的频率（由操作系统知道），或者分支错误预测发生的次数（由 CPU 知道）。被仪器化的代码是应用程序的一部分，并具有与应用程序本身相同的特权。它在用户空间中运行，无法访问内核。

但更重要的是，这种技术的缺点是每次需要仪器化新内容，例如另一个变量时，都需要重新编译。这可能会给工程师带来负担，并增加分析时间。不幸的是，还有其他一些缺点。由于通常您关心的是应用程序中的热点路径，因此您正在为位于代码性能关键部分的内容进行仪器化。在热点路径中注入仪器化代码可能很容易导致整体基准测试减慢 2 倍。请记住不要对被仪器化的程序进行基准测试，即不要在同一运行中进行评分和分析。请记住，通过对代码进行仪器化，您会改变程序的行为，因此您可能看不到之前看到的相同效果。

上述所有内容增加了实验之间的时间，消耗了更多的开发时间，这就是为什么工程师如今很少手动仪器化他们的代码的原因。然而，自动化代码仪器化仍然被编译器广泛使用。编译器能够自动对整个程序进行仪器化，并收集有关执行的有趣统计信息。自动仪器化最广泛的用例是代码覆盖分析和基于性能指导的优化（参见 Section 10.5）。

在谈到仪器化时，重要的是要提到二进制仪器化技术。二进制仪器化的思想类似，但它是在已构建的可执行文件上完成的，而不是在源代码级别上。有两种类型的二进制仪器化：静态（在构建之前完成）和动态（在程序执行时根据需要插入仪器化代码）。动态二进制仪器化的主要优势在于它不需要重新编译和重新链接程序。此外，通过动态仪器化，可以将仪器化的量限制为仅限于感兴趣的代码区域，而不是整个程序。

二进制仪器化在性能分析和调试中非常有用。二进制仪器化最流行的工具之一是 Intel Pin⁵⁷ 工具。Pin 拦截程序在发生有趣事件时的执行，并生成从程序中的这一点开始的新仪器化代码。它允许收集各种运行时信息，例如：

- 指令计数和函数调用计数。
- 拦截函数调用和应用程序中任何指令的执行。
- 允许通过在区域开始时捕获内存和硬件寄存器状态来“记录和重放”程序区域。

与代码仪器化类似，二进制仪器化只允许对用户级代码进行仪器化，而且可能非常慢。

5.2 跟踪

跟踪在概念上与仪器化非常相似，但略有不同。代码仪器化假设用户可以编排他们应用程序的代码。另一方面，跟踪依赖于程序的外部依赖项的现有仪器化。例如，strace 工具使我们能够跟踪系统调用，并可以被视为对 Linux 内核的仪器化。英特尔处理器跟踪（见附录 D）使您能够记录程序执行的指令，并可以被视为对 CPU 的仪器化。跟踪可以从事先适当仪器化的组件中获得，并且不受更改的影响。跟踪通常被用作黑匣子方法，其中用户无法修改应用程序的代码，但他们希望了解程序在幕后执行的操作。

Listing 5.2 提供了使用 Linux strace 工具跟踪系统调用的示例，显示了运行 git status 命令时输出的前几行。通过使用 strace 跟踪系统调用，可以得知每个系统调用的时间戳（最左边的列），其退出状态以及每个系统调用的持续时间（在尖括号内）。

代码清单：使用 strace 跟踪系统调用。

⁵⁷ PIN - <https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool>

```
$ 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>
...
...
```

跟踪的开销非常取决于我们尝试跟踪的内容。例如，如果我们跟踪的程序几乎不进行系统调用，那么在strace下运行它的开销将接近零。另一方面，如果我们跟踪的程序严重依赖于系统调用，那么开销可能会非常大，例如，增加了100倍⁵⁸。此外，跟踪可能会生成大量数据，因为它不会跳过任何样本。为了补偿这一点，跟踪工具提供了过滤器，使您能够将数据收集限制为特定的时间片段或特定代码段。

通常，类似于仪器化的跟踪用于探查系统中的异常情况。例如，您可能想要确定在程序出现10秒不响应的情况下应用程序中发生了什么。正如您将在后面看到的，采样方法并不是为此设计的，但是通过跟踪，您可以看到是什么导致了程序不响应。例如，使用英特尔PT，您可以重构程序的控制流并确切地知道执行了哪些指令。

跟踪对调试也非常有用。其底层特性支持基于记录的跟踪的“记录和重放”用例。Mozilla的一个这样的工具是rr⁵⁹调试器，它执行进程的记录和重放，支持向后单步执行等等。大多数跟踪工具都能够为事件添加时间戳，这使我们能够与在那段时间内发生的外部事件进行相关。也就是说，当我们观察到程序中出现故障时，我们可以查看我们应用程序的跟踪，并将此故障与在该时间段内整个系统中发生的情况进行关联。

5.3 工作负载特征化

工作负载特征化是通过定量参数和函数描述工作负载的过程。简单来说，它意味着计算某些性能事件的绝对数量。特征化的目标是定义工作负载的行为并提取其最重要的特征。在高层次上，一个应用程序可以属于以下一种或多种类型：交互式、数据库、实时、基于网络的、大规模并行等。不同的工作负载可以使用不同的指标和参数来解决特定的应用程序领域。

这是一本关于低级性能的书，记住了吗？所以，我们将专注于提取与CPU和内存性能相关的特征。从CPU角度看，最好的工作负载特征化示例是顶层微体系结构分析（Top-down Microarchitecture Analysis，TMA）方法，我们将在Section ??中仔细研究。TMA试图通过将应用程序放入以下4个桶之一来表征其性能：CPU前端、CPU后端、退役和错误预测，具体取决于造成最多性能问题的原因。TMA使用性能监视计数器（PMCs）收集所需信息，并识别CPU微体系结构的低效使用。

但即使没有完全成熟的特征化方法，收集某些性能事件的绝对数量也可能会有所帮助。我们希望前一章中对四种不同基准测试的性能指标进行的案例研究证明了这一点。PMCs是低级性能分析的非常重要的工具。它们可以提供有

⁵⁸一篇有关strace的文章，作者是B. Gregg - <http://www.brendangregg.com/blog/2014-05-11/strace-wow-much-syscall.html>

⁵⁹Mozilla rr 调试器 - <https://rr-project.org/>

关程序执行的独特信息。PMCs 通常以两种模式使用：“计数”和“采样”。计数模式用于工作负载特征化，而采样模式用于查找热点，我们将很快讨论。

5.3.1 计数性能事件

计数背后的想法非常简单：我们希望在程序运行时计数某些性能事件的绝对数量。图 @fig:Counting 展示了从程序开始到结束计数性能事件的过程。

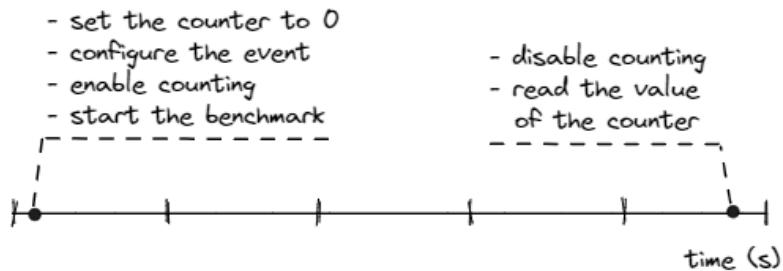


Figure 29: 计数性能事件。

图 @fig:Counting 中概述的步骤大致代表了典型分析工具会执行的操作来计数性能事件。这个过程是在perf stat工具中实现的，它可以用于计数各种硬件事件，比如指令数、周期数、缓存失效等。下面是perf stat的输出示例：

```
$ 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
```

知道这些数据非常有用。首先，它使我们能够快速发现一些异常，比如高的分支误预测率或低的 IPC。此外，当您进行了代码更改并希望验证更改是否改进了性能时，它可能会派上用场。查看绝对数值可能有助于您证明或拒绝代码更改。主要作者将perf stat用作简单的基准包装器。由于计数事件的开销很小，几乎所有基准测试都可以自动在perf stat下运行。它作为性能调查的第一步。有时，可以立即发现异常，这可以节省一些分析时间。

5.3.2 手动收集性能计数器数据

现代 CPU 拥有数百个可计数的性能事件。记住所有这些事件及其含义是非常困难的。更难的是理解何时使用特定的 PMC。这就是为什么通常我们不建议手动收集特定的 PMCs，除非您真的知道自己在做什么。相反，我们建议使用像 Intel Vtune Profiler 这样的工具来自动化这个过程。尽管如此，有时候您可能有兴趣收集特定的 PMCs。

您可以在 [Intel, 2023b, 第 3B 卷, 第 19 章] 中找到所有 Intel CPU 代数的性能事件的完整列表。PMCs 的描述也可以在 perfmon-events.intel.com 上找到。每个事件都使用Event和Umask十六进制值进行编码。有时性能事件还可以使用附加参数进行编码，例如Cmask、Inv等。表 [6] 显示了针对 Intel Skylake 微体系结构编码的两个性能事件的示例。

Table 6: Skylake 性能事件的编码示例。

事件编号	Umask 值	事件掩码助记符	描述
C0H	00H	INST_RETIRED. ANY_P	退役的指令数量。
C4H	00H	BR_INST_RETIRED. ALL_BRANCHESk	退役的分支指令。

Linux perf提供了常用性能计数器的映射。可以通过伪名称而不是指定Event和Umask十六进制值来访问它们。例如，branches只是BR_INST_RETIRE_ALL_BRANCHES的同义词，并且将测量相同的内容。可以使用perf list查看可用映射名称的列表：

```
$ 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]
```

但是，Linux perf并不为每个CPU架构的所有性能计数器提供映射。如果您要查找的PMC没有映射，则可以使用以下语法进行收集：

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

由于访问PMCs需要root访问权限，因此并非每个环境都可以使用性能计数器。在虚拟化环境中运行的应用程序通常没有root访问权限。对于在公共云中执行的程序，如果虚拟机(VM)管理器未正确向客户端公开PMU编程接口，则在客户端容器中直接运行基于PMU的分析器将不会产生有用的输出。因此，基于CPU性能计数器的分析器在虚拟化和云环境中效果不佳[Du et al., 2010]。尽管情况正在改善。VmWare®是第一个启用⁶⁰虚拟CPU性能计数器(vPMC)的VM管理器之一。AWS EC2云⁶¹为专用主机启用了PMCs。

5.3.3 多路复用和事件缩放

有些情况下，我们希望同时计数许多不同的事件。但是只有一个计数器，一次只能计数一件事情。这就是为什么PMU中有多个计数器的原因（在最近的英特尔Goldencove微体系结构中，每个硬件线程有12个可编程的PMC，每个线程有6个）。即使这样，固定和可编程计数器的数量并不总是足够的。Top-down微体系结构分析(TMA)方法要求在单个程序执行中收集多达100种不同的性能事件。现代CPU没有那么多的计数器，这就是多路复用发挥作用的时候。

如果事件比计数器多，分析工具使用时间多路复用为每个事件提供访问监视硬件的机会。图@fig:Multiplexing1显示了只有4个PMC可用时8个性能事件之间的多路复用示例。

通过多路复用，事件并不是一直被测量的，而只在一段时间内被测量。在运行结束时，性能分析工具需要根据总启用时间对原始计数进行缩放：

$$\text{最终计数} = \text{原始计数} \times (\text{运行时间}/\text{启用时间})$$

让我们以图@fig:Multiplexing2为例。假设在分析过程中，我们能够在三个时间间隔内测量来自第1组的一个事件。每个测量间隔持续100ms(启用时间)。程序运行时间为500ms(运行时间)。该计数器的总事件数为10'000(原始计数)。因此，最终计数需要按以下方式进行缩放：

$$\text{最终计数} = 10'000 \times (500ms / (100ms \times 3)) = 16'666$$

这提供了在整个运行过程中测量事件时计数的估计。非常重要的是要理解，这仍然是一个估计值，而不是实际计数。多路复用和缩放可以安全地用于执行长时间间隔内相同代码的稳定工作负载。然而，如果程序经常在不同的热点之

⁶⁰ VMWare PMCs - <https://www.vladan.fr/what-are-vmware-virtual-cpu-performance-monitoring-counters-vpmcs/>

⁶¹ Amazon EC2 PMCs - <http://www.brendangregg.com/blog/2017-05-04/the-pmc-of-ec2.html>

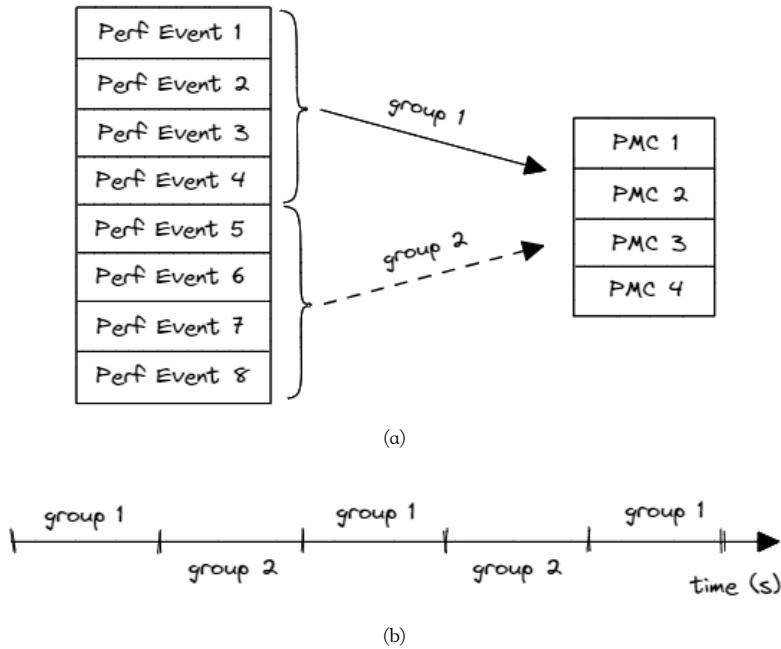


Figure 30: 8 个性能事件之间的多路复用示例，只有 4 个 PMC 可用。

间跳转，即有不同的阶段，那么就会产生盲点，这可能会在缩放过程中引入错误。为了避免缩放，可以尝试将事件的数量减少到不超过可用物理 PMC 的数量。然而，这将需要多次运行基准测试，以测量感兴趣的所有计数器。

5.4 使用标记器 API

在某些情况下，我们可能对分析特定代码区域的性能感兴趣，而不是整个应用程序。例如，当您开发一段新代码并只想关注该代码时，就会遇到这种情况。自然地，您会希望跟踪优化进度并捕获其他性能数据，以帮助您一路前进。大多数性能分析工具都提供特定的 标记器 API，可以让您做到这一点。这里有一些例子：

- Likwid 有 LIKWID_MARKER_START / LIKWID_MARKER_STOP 宏。
- Intel VTune 有 __itt_task_begin / __itt_task_end 函数。
- AMD uProf 有 amdProfileResume / amdProfilePause 函数。

这种混合方法结合了检测和性能事件计数的优点。标记器 API 允许我们将性能统计数据归因于代码区域（循环、函数）或功能片段（远程过程调用（RPC）、输入事件等），而不是测量整个程序。您获得的数据质量足以证明这种努力是值得的。例如，在追查仅针对特定类型 RPC 出现的性能漏洞时，您可以仅针对该类型的 RPC 启用监控。

下面我们提供了一个非常基本的示例，展示了如何使用 libpfm⁶²，这是一个流行的用于收集性能监控事件的 Linux 库。它构建在 Linux perf_events 子系统之上，该子系统允许您直接访问性能事件计数器。perf_events 子系统相当底层，因此 libpfm 在这里很有用，因为它增加了用于识别 CPU 上可用事件的发现工具以及围绕原始 perf_event_open 系统调用的包装库。Listing 63 展示了如何使用 libpfm 为 C-Ray⁶³ benchmark 的 render 函数进行检测。

清单：在 C-Ray benchmark 上使用 libpfm4 标记器 API

```
+#include <perfmon/pfmlib.h>
+#include <perfmon/pfmlib_perf_event.h>
```

⁶² libpfm4 - <https://sourceforge.net/p/perfmon2/libpfm4/ci/master/tree/>

⁶³ C-Ray 基准测试 - <https://openbenchmarking.org/test/pts/c-ray>

```

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

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

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

+         // save deltas in separate arrays
+         nanosecs[j * xsz + i] = after.time_running - before.time_running;
    }
}

```

```

+     instrs [j * xsz + i] = after.values[0] - before.values[0];
+     cycles [j * xsz + i] = after.values[1] - before.values[1];
+     branches[j * xsz + i] = after.values[2] - before.values[2];
+     br_misps[j * xsz + i] = after.values[3] - before.values[3];

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

```

在这个代码示例中，我们首先初始化libpfm库并配置性能事件以及我们将用于读取它们的格式。在 C-Ray 基准测试中，`render`函数只被调用一次。在您自己的代码中，务必小心不要多次进行libpfm初始化。然后，我们选择要分析的代码区域，在我们的案例中，它是一个带有`trace`函数调用的循环。我们用两个`read`系统调用包围这个代码区域，它们将在循环之前和之后捕获性能计数器的值。接下来，我们保存这些增量以供以后处理，例如，在这种情况下，我们通过计算平均值、90th 百分位数和最大值对其进行聚合（代码未显示）。在基于 Intel Alderlake 的机器上运行它，我们得到了下面显示的输出。不需要 root 权限，但`/proc/sys/kernel/perf_event_paranoid`应该设置为小于 1。当在一个线程内读取计数器时，这些值仅适用于该线程。它可以选择性地包括运行并归因于该线程的内核代码。

```
$ ./c-ray-f -s 1024x768 -r 2 -i sphfract -o output.ppm
```

Per-pixel ray tracing stats:

	avg	p90	max
<hr/>			
nanoseconds	4571	6139	25567
instructions	71927	96172	165608
cycles	20474	27837	118921
branches	5283	7061	12149
branch-misses	18	35	146

请记住，我们添加的仪器测量了每个像素的光线跟踪统计数据。将平均数乘以像素数（1024x768）应该给出大致的程序总统计数据。在这种情况下，一个很好的健全性检查是运行`perf stat`并比较我们收集的性能事件的整体 C-Ray 统计数据。

C-ray 基准测试主要强调 CPU 核心的浮点性能，通常不应该导致测量结果的高方差，换句话说，我们期望所有的测量结果都非常接近。然而，我们看到情况并非如此，因为 p90 值是平均值的 1.33 倍，而最大值有时比平均情况慢 5 倍。这里最可能的解释是对于一些像素，算法遇到了一个边界情况，执行了更多的指令，随后运行时间更长。但最好通过研究源代码或扩展仪器测量来捕获更多有关“慢”像素的数据，以确认假设。

Listing 63 中显示的附加仪器测量代码导致了 17% 的开销，这对于本地实验来说是可以接受的，但在生产环境中运行的开销相当高。大多数大型分布式系统的目地是小于 1% 的开销，对于某些系统来说，最多可接受 5% 的开销，但是 17% 的减速不太可能让用户满意。管理仪器测量的开销至关重要，特别是如果您选择在生产环境中启用它。

开销通常以时间单位或工作单位（RPC、数据库查询、循环迭代等）的发生率来计算。如果我们系统上的一个系统调用大约需要 1.6 微秒的 CPU 时间，并且我们每个像素都执行两次（外部循环的迭代），那么每个像素的开销就是 3.2 微秒的 CPU 时间。

降低开销的策略有很多。作为一个通用原则，您的仪器测量应该始终具有固定的成本，例如，确定性系统调用，但

不是列表遍历或动态内存分配，否则它会干扰测量。仪器测量代码有三个逻辑部分：收集信息、存储信息和报告信息。为了降低第一部分（收集）的开销，我们可以减少采样率，例如，每 10 个 RPC 采样一次，然后跳过其余的。对于长时间运行的应用程序，性能可以通过相对便宜的随机采样进行监视 - 随机选择要观察的事件。这些方法牺牲了收集的准确性，但仍然提供了对整体性能特征的良好估计，同时产生了非常低的开销。

对于第二部分和第三部分（存储和聚合），建议仅收集、处理和保留您需要了解系统性能的数据量。您可以通过使用“在线”算法来计算平均值、方差、最小值、最大值和其他指标来避免将每个样本存储在内存中。这将大大减少仪器测量的内存占用。例如，方差和标准差可以使用 Knuth 的在线方差算法来计算。一个良好的实现⁶⁴ 使用不到 50 字节的内存。

对于长时间运行的例程，您可以在开始、结束和一些中间部分收集计数器。在连续运行中，您可以二分搜索执行最差的例程部分并进行优化。重复此过程，直到所有性能差的地方都被消除。如果尾延迟是主要关注的问题，那么在特别慢的运行中发出日志消息可以提供有用的见解。

在 Listing 63 中，我们同时收集了 4 个事件，尽管 CPU 有 6 个可编程计数器。您可以打开具有不同事件集的其他组。内核将选择不同的组来运行。`time_enabled` 和 `time_running` 字段指示了多路复用。它们都是以纳秒为单位的持续时间。`time_enabled` 字段表示事件组已启用的纳秒数。`time_running` 表示实际收集事件的时间占已启用时间的多少。如果同时启用了两个无法放在 HW 计数器上的事件组，您可能会看到它们都收敛到 `time_running = 0.5 * time_enabled`。调度通常很复杂，因此在依赖于您的确切场景之前，请进行验证。

同时捕获多个事件允许计算我们在第 4 章中讨论的各种指标。例如，捕获 `INSTRUCTIONS_RETIRED` 和 `UNHALTED_CLOCK_CYCLES` 使我们能够测量 IPC。我们可以通过比较 CPU 周期 (`UNHALTED_CORE_CYCLES`) 和固定频率参考时钟 (`UNHALTED_REFERENCE_CYCLES`) 来观察频率缩放的影响。通过请求消耗的 CPU 周期 (`UNHALTED_CORE_CYCLES`, 仅在线程运行时计数) 并与墙钟时间进行比较，可以检测线程未运行的情况。此外，我们可以对数字进行归一化，以获得每秒/时钟/指令的事件速率。例如，通过测量 `MEM_LOAD_RETIRED.L3_MISS` 和 `INSTRUCTIONS_RETIRED`，我们可以获得 L3MPKI 指标。正如您所见，这种设置非常灵活。

事件分组的重要属性是计数器将原子地在同一次 `read` 系统调用下可用。这些原子束非常有用。首先，它允许我们在每个组内相关事件。例如，我们为代码区域测量 IPC，并发现它非常低。在这种情况下，我们可以将两个事件（指令和周期）与第三个事件配对，例如 L3 缓存丢失，以检查它是否对我们正在处理的低 IPC 有贡献。如果没有，我们将继续使用其他事件进行因子分析。其次，事件分组有助于减轻工作负载具有不同阶段的偏差。由于组内的所有事件同时测量，它们始终捕获相同的阶段。

在某些场景中，仪器测量可能成为功能或特性的一部分。例如，开发人员可以实现一个仪器测量逻辑，用于检测 IPC 的下降（例如，当有一个繁忙的兄弟硬件线程运行时）或 CPU 频率的下降（例如，由于负载过重而导致系统节流）。当发生这种事件时，应用程序会自动推迟低优先级的工作以补偿临时增加的负载。

5.5 采样

采样是最常用的性能分析方法。人们通常将其与程序中的热点识别联系起来。广义而言之，采样有助于找到代码中对特定性能事件贡献最多的位置。如果我们考虑发现热点，那么这个问题可以重新表述为程序中的哪个地方消耗了最多的 CPU 周期。人们通常将技术上称为采样的操作称为“性能分析”。根据维基百科 [https://en.wikipedia.org/wiki/Profiling_\(computer_programming\)](https://en.wikipedia.org/wiki/Profiling_(computer_programming))⁶⁵ 的说法，性能分析是一个更广泛的术语，包括各种收集数据的技术，例如中断、代码检测和 PMC。

令人惊讶的是，人们可以想象到的最简单的采样性能分析器就是调试器。事实上，您可以通过以下步骤识别热点：
a) 在调试器下运行程序，b) 每 10 秒暂停一次程序，c) 记录程序停止的位置。如果您多次重复 b) 和 c)，您就可以从这些样本构建一个直方图。您停止最多的代码行将是程序中最热的代码行。当然，这不是一种高效的发现热点的方法。

⁶⁴ 准确计算运行方差 - https://www.johndcook.com/blog/standard_deviation/

⁶⁵ Profiling(wikipedia) - [https://en.wikipedia.org/wiki/Profiling_\(computer_programming\)](https://en.wikipedia.org/wiki/Profiling_(computer_programming)).

法，我们也不推荐这样做。它只是为了说明这个概念。尽管如此，它是关于真实性能分析工具如何工作的简化描述。现代性能分析器每秒可以收集数千个样本，这为基准测试中的热点提供了相当准确的估计。

与调试器的例子一样，每次捕获新的样本时，被分析程序的执行都会中断。在中断时，性能分析器会收集程序状态的快照，构成一个样本。为每个样本收集的信息可能包括中断时执行的指令地址、寄存器状态、调用堆栈（见 Section 5.5.3），等等。收集到的样本存储在一个转储文件中，该文件可以进一步用于显示程序中耗时最多的部分、调用图等。

5.5.1 用户模式和基于硬件事件的采样

采样可以采用两种不同的模式进行，即用户模式采样或基于硬件事件的采样 (EBS)。用户模式采样是一种纯软件方法，将代理库嵌入到被分析的应用程序中。代理为应用程序中的每个线程设置操作统计时器。计时器到期后，应用程序会收到由收集器处理的 SIGPROF 信号。EBS 使用硬件 PMC 触发中断。特别是，它使用 PMU 的计数器溢出功能，我们将在稍后讨论。

用户模式采样只能用于识别热点，而 EBS 可用于涉及 PMC 的其他分析类型，例如，基于缓存未命中、TMA（见 Section ??）等进行采样。

与 EBS 相比，用户模式采样的运行时开销更大。当采样间隔为 10ms 时，用户模式采样的平均开销约为 5%，而 EBS 的开销不到 1%。由于开销更低，您可以使用更高的采样率使用 EBS，这将提供更准确的数据。然而，用户模式采样生成的数据更少，因此处理起来也更快。

5.5.2 寻找热点

在本节中，我们将讨论使用 PMC 和 EBS 的机制。图 31 说明了 PMU 的计数器溢出功能，该功能用于触发性能监控中断 (PMI)，也称为 SIGPROF。在基准测试开始时，我们会配置我们想要采样的事件。识别热点意味着知道程序花费大部分时间在哪里。因此，在周期上进行采样是非常自然的，这也是许多性能分析工具的默认设置。但这并不一定是严格的规则；我们可以对任何想要的性能事件进行采样。例如，如果我们想知道程序中 L3 缓存未命中最多的位置，我们将在相应的事件上进行采样，即 MEM_LOAD_RETIRED.L3_MISS。

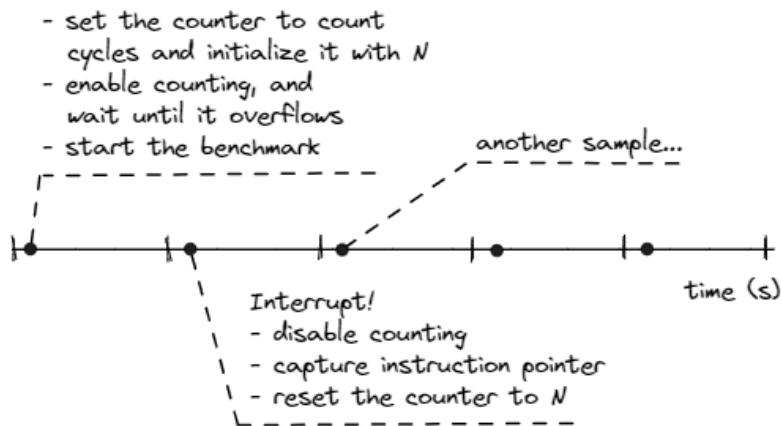


Figure 31: 使用性能计数器进行采样

初始化寄存器后，我们开始计数并让基准测试继续。我们将 PMU 配置为计数周期，因此它将在每个周期递增。最终，它会溢出。当寄存器溢出时，硬件将引发 PMI。性能分析工具被配置为捕获 PMI，并具有用于处理它们的中断服务程序 (ISR)。我们在 ISR 中执行多个步骤：首先，我们禁用计数；然后，我们记录 CPU 在计数器溢出时执行的指令；然后，我们将计数器重置为 N 并恢复基准测试。

现在，让我们回到值 N。使用这个值，我们可以控制我们想要多久获得一个新的中断。假设我们想要更细粒度的粒度，每 100 万条指令获得一个样本。为了实现这一点，我们可以将计数器设置为 (unsigned) -1'000'000，这样它将在每 100 万条指令后溢出。这个值也称为“采样后”值。

我们重复这个过程多次，以建立足够的样本集合。如果我们稍后聚合这些样本，就可以构建程序中最热位置的直方图，就像下面 Linux perf record/report 输出所示。这给了我们按降序排序的程序函数开销的细分（热点）。下面展示了一个来自 Phoronix 测试套件: <https://www.phoronix-test-suite.com/>⁶⁶ 的 x264: <https://openbenchmarking.org/test/pts/x264>⁶⁷ 基准测试的采样示例：

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

Linux perf 采集了 35'035 个样本，这意味着中断执行的过程发生了这么多次。我们还使用了 -F 1000，将采样率设置为每秒 1000 个样本。这大致与 36.2 秒的整体运行时间相匹配。请注意，Linux perf 提供了大约经过的总周期数。如果我们将它除以样本数，我们将得到 $156756064947 \text{ 个周期} / 35035 \text{ 个样本} = 450 \text{ 万个周期/样本}$ 。这意味着 Linux perf 将数字 N 设置为大约 4'500'000 以每秒收集 1000 个样本。数字 N 可以由工具根据实际 CPU 频率动态调整。

当然，对我们最有价值的是按每个函数分配的样本数量排序的热点列表。在知道最热门的函数之后，我们可能想要更深入地研究：每个函数内部代码的热门部分是什么。要查看内联函数的配置文件数据以及为特定源代码区域生成的汇编代码，我们需要使用调试信息 (-g 编译器标志) 构建应用程序。

调试信息有两个主要用例：调试功能问题（错误）和性能分析。对于功能调试，我们需要尽可能多的信息，这是您传递 -g 编译器标志时的默认设置。但是，如果用户不需要完整的调试体验，那么只需要行号就足以进行性能分析。您可以通过使用 -gline-tables-only 选项将生成的调试信息量减少到代码中符号的行号。⁶⁸

Linux perf 没有丰富的图形支持，因此查看源代码的热门部分非常不方便，但可以做到。Linux perf 将源代码与生

⁶⁶ Phoronix test suite - <https://www.phoronix-test-suite.com/>.

⁶⁷ x264 benchmark - <https://openbenchmarking.org/test/pts/x264>.

⁶⁸ 过去，使用调试信息 (-g) 编译时存在 LLVM 编译器错误。代码转换传递错误地处理了调试内部函数的存在，导致了不同的优化决策。它不会影响功能，只会影响性能。其中一些已经修复，但很难说是否存在其他问题。

成的汇编代码混合在一起，如下所示：

```
# snippet of annotating source code of 'x264_8_me_search_ref' function
$ perf annotate x264_8_me_search_ref --stdio
Percent | Source code & Disassembly of x264 for cycles:ppp
-----
...
:           bmx += square1[bcost&15] [0];    <== source code
1.43 : 4eb10d: movsx  ecx,BYTE PTR [r8+rdx*2]      <== corresponding machine code
:           bmy += square1[bcost&15] [1];
0.36 : 4eb112: movsx  r12d,BYTE PTR [r8+rdx*2+0x1]
:           bmx += square1[bcost&15] [0];
0.63 : 4eb118: add    DWORD PTR [rsp+0x38],ecx
:           bmy += square1[bcost&15] [1];
...
...
```

大多数带有图形用户界面 (GUI) 的性能分析器，例如 Intel VTune Profiler，都可以并排显示源代码和关联的汇编代码。此外，还有一些工具可以以类似于 Intel Vtune 和其他工具的丰富图形界面可视化 Linux perf 原始数据的输出。您将在第 7 章中更详细地看到所有这些内容。

5.5.3 收集调用堆栈

在采样时，我们经常会遇到程序中最热门的函数被多个函数调用的情况。图 32 显示了一个这样的场景示例。性能分析工具的输出可能显示 `foo` 是程序中最热门的函数之一，但如果它有多个调用者，我们想知道哪个调用者调用 `foo` 的次数最多。对于程序中出现诸如 `memcpy` 或 `sqrt` 之类的库函数的热点，这是典型情况。要了解特定的函数为什么成为热点，我们需要知道程序控制流图 (CFG) 中哪个路径导致了这种情况。

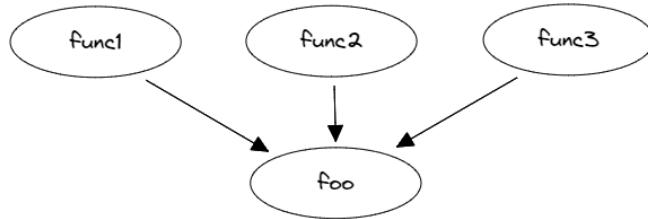


Figure 32: Control Flow Graph: hot function “foo” has multiple callers.

分析 `foo` 所有调用者的源代码可能非常耗时。我们只想关注那些导致 `foo` 成为热点的调用者。换句话说，我们想要找出程序 CFG 中最热门的路径。性能分析工具通过在收集性能样本时捕获进程的调用堆栈和其他信息来实现这一点。然后，对所有收集到的堆栈进行分组，使我们能够看到导致特定函数的最热门路径。

在 Linux perf 中，可以使用三种方法收集调用堆栈：

1. 帧指针 (`perf record --call-graph fp`)。要求使用 `--fnoomit-frame-pointer` 构建二进制文件。历史上，帧指针 (RBP 寄存器) 用于调试，因为它使我们能够在不弹出所有参数的情况下获取调用堆栈（也称为堆栈展开）。帧指针可以立即告诉返回地址。但是，它仅为这个目的占用了一个寄存器，所以开销很大。它也可用于性能分析，因为它可以进行廉价的堆栈展开。
2. DWARF 调试信息 (`perf record --call-graph dwarf`)。要求使用 DWARF 调试信息 `-g(-gline-tables-only)` 构建二进制文件。通过堆栈展开过程获取调用堆栈。

3. 英特尔最后分支记录 (LBR) 硬件功能 (`perf record --call-graph lbr`)。通过解析 LBR 堆栈 (一组硬件寄存器) 获取调用堆栈。调用图不像前两种方法那么深。有关 LBR 的更多信息, 请参见 Section 5.11。

下面是使用 LBR 在程序中收集调用堆栈的示例。通过查看输出, 我们知道 55% 的时间 `foo` 是由 `func1` 调用的, 33% 的时间是由 `func2` 调用的, 11% 的时间是由 `fun3` 调用的。我们可以清楚地看到 `foo` 的调用者之间的开销分布, 现在可以将注意力集中在程序 CFG 中最热的边 `func1 -> foo` 上, 但我们也应该关注边 `func2 -> foo`。

```
$ perf record --call-graph lbr -- ./a.out
$ perf report -n --stdio --no-children
# Samples: 65K of event 'cycles:ppp'
# Event count (approx.): 61363317007
# Overhead            Samples  Command  Shared Object      Symbol
# .....              .....
99.96%          65217  a.out    a.out        [.] foo
|
--99.96%--foo
|
|--55.52%--func1
|       main
|       __libc_start_main
|       _start
|
|--33.32%--func2
|       main
|       __libc_start_main
|       _start
|
|--11.12%--func3
       main
       __libc_start_main
       _start
```

当使用 Intel VTune Profiler 时, 可以在配置分析时勾选相应的“收集堆栈”框来收集调用堆栈数据。当使用命令行界面时, 指定 `-knob enable-stack-collection=true` 选项。

知道一种有效的收集调用堆栈的方法非常重要。不熟悉该概念的开发人员会尝试使用调试器来获取此信息。他们通过中断程序的执行并分析调用堆栈 (例如, `gdb` 调试器中的 `backtrace` 命令) 来做到这一点。不要这样做, 让性能分析工具来完成这项工作, 它更快、更准确。

5.6 Roofline 性能模型

Roofline 性能模型是一个以吞吐量为导向的性能模型, 在 HPC 领域广泛使用。它于 2009 年在加州大学伯克利分校开发。模型中的“roofline”表示应用程序的性能不能超过机器的能力。程序中的每个函数和每个循环都受到机器的计算或内存容量的限制。这个概念在图 33 中有所体现。应用程序的性能始终会受到某条“roofline”函数的限制。

硬件有两个主要限制: 计算速度 (峰值计算性能, FLOPS) 和数据移动速度 (峰值内存带宽, GB/s)。应用程序的最大性能受峰值 FLOPS (水平线) 和平台带宽乘以运算强度 (对角线) 两者之间的最小值限制。图 33 中的 roofline 图将两个应用程序 A 和 B 的性能与硬件限制进行了对比。应用程序 A 的运算强度较低, 其性能受内存带宽限制, 而应用

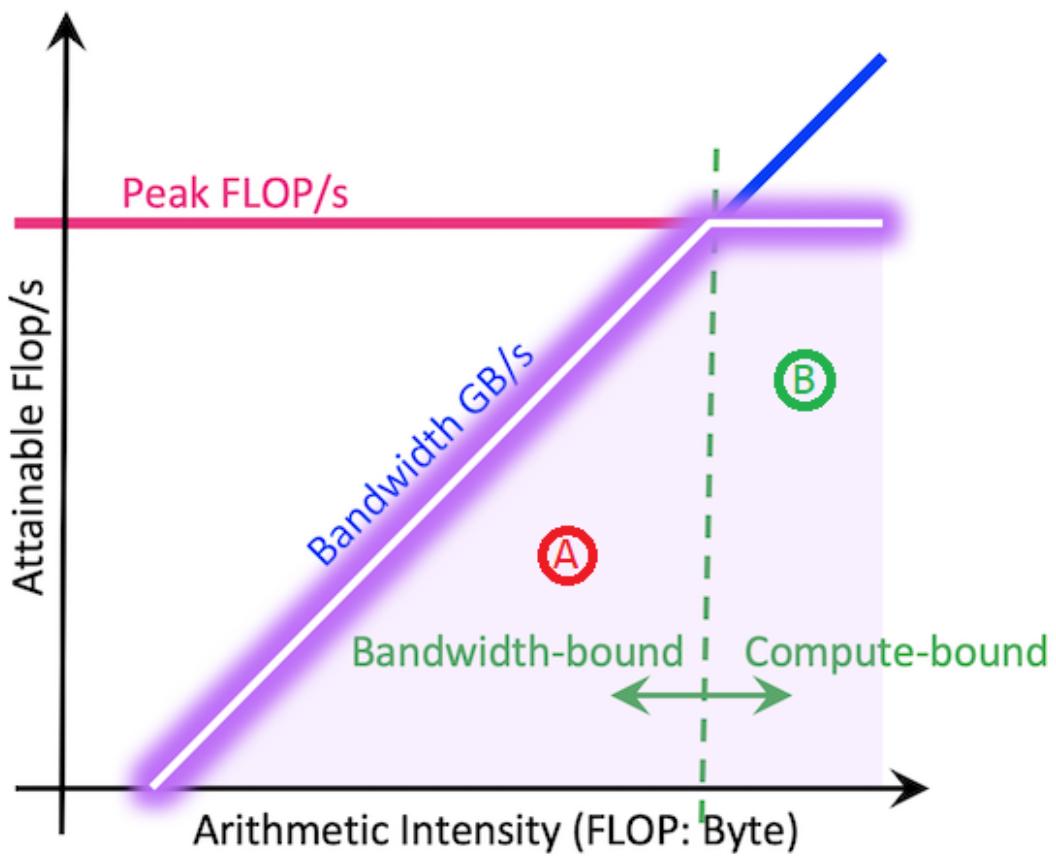


Figure 33: Roofline model. © Image taken from [NERSC Documentation](#).

程序 B 的计算密集型程度更高，因此不会受到内存瓶颈的太大影响。类似地，A 和 B 可以代表程序中的两个不同函数，并具有不同的性能特征。Roofline 性能模型会考虑到这一点，可以在同一个图表上显示应用程序的多个函数和循环。

算术强度 (AI) 是 FLOPS 和字节之间的比率，可以针对程序中的每个循环进行提取。让我们计算 Listing ?? 中代码的算术强度。在最内层的循环体中，我们有一个加法和一个乘法；因此，我们有 2 个 FLOP。此外，我们还有三个读取操作和一个写入操作；因此，我们传输了 $4 \text{ ops} * 4 \text{ bytes} = 16$ 个字节。该代码的算术强度为 $2 / 16 = 0.125$ 。AI 是给定性能点的 X 轴上的值。

清单：朴素并行矩阵乘法。
~~~~~ {#lst:BasicMatMul .cpp .numberLines}  
void matmul(int N, float a[][2048], float b[][2048],  
float c[][2048]) { #pragma omp parallel for for(int i = 0; i < N; i++) { for(int j = 0; j < N; j++) { for(int k = 0; k < N; k++) {  
c[i][j] = c[i][j] + a[i][k] \* b[k][j]; } } } } ~~~~

传统的应用程序性能提升方式是充分利用机器的 SIMD 和多核能力。通常情况下，我们需要优化多个方面：矢量化、内存、线程。Roofline 方法可以帮助评估应用程序的这些特性。在 roofline 图表上，我们可以绘制标量单核、SIMD 单核和 SIMD 多核性能的理论最大值（见图 34）。这将使我们了解改进应用程序性能的空间。如果我们发现我们的应用程序受计算绑定（即具有高算术强度）并且低于峰值标量单核性能，我们应该考虑强制矢量化（参见 Section 8.4）并将工作分发到多个线程上。相反，如果应用程序的算术强度低，我们应该寻求改善内存访问的方法（参见 Chapter 7）。使用 Roofline 模型优化性能的最终目标是向上移动这些点。矢量化和线程化向上移动点，而通过增加算术强度优化内存访问则会将点向右移动，并且可能也会提高性能。

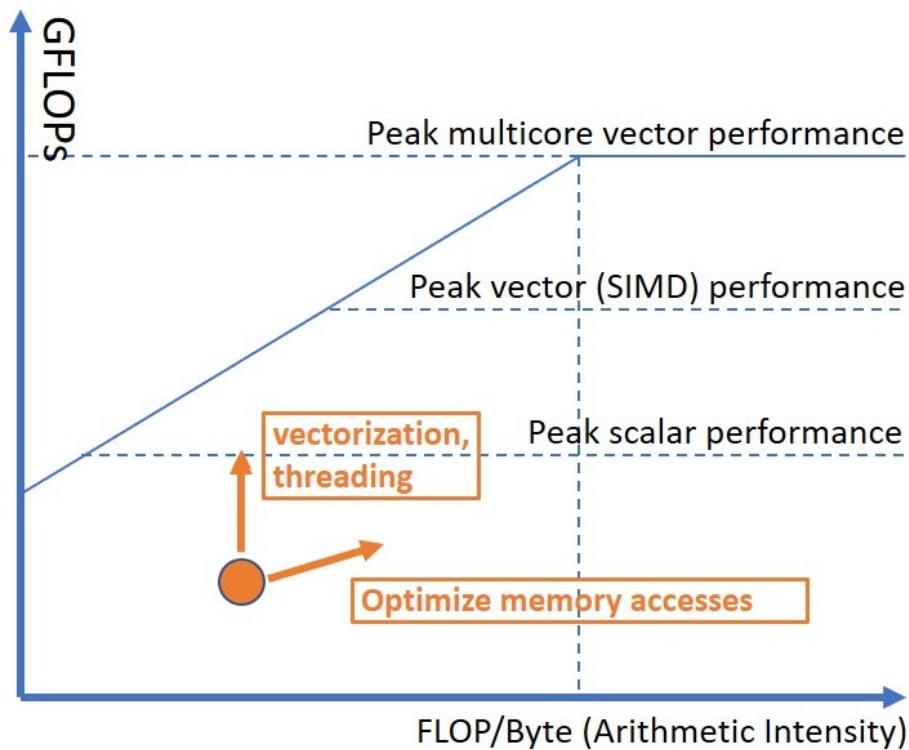


Figure 34: Roofline model.

理论最大值（roofline）通常在设备规范中给出，可以很容易地查阅。您也可以根据您使用的机器的特性计算理论最大值。一旦您知道机器的参数，这通常并不难做。对于 Intel Core i5-8259U 处理器，使用 AVX2 和 2 个 Fused Multiply Add (FMA) 单元的最大 FLOP 数（单精度浮点）可以计算如下：

$$\begin{aligned}
 \text{峰值 FLOPS} &= 8 \text{ (逻辑核心数量)} \times \frac{256 \text{ (AVX 位宽)}}{32 \text{ 位 (float 大小)}} \times \\
 &\quad 2 \text{ (FMA)} \times 3.8 \text{ GHz (最大睿频)} \\
 &= 486.4 \text{ GFLOPs}
 \end{aligned}$$

我用于实验的 Intel NUC Kit NUC8i5BEH 的最大内存带宽可以如下计算。请记住，DDR 技术允许每次内存访问传输 64 位或 8 个字节。

$$\begin{aligned}
 \text{峰值内存带宽} &= 2400 \text{ (DDR4 内存传输速率)} \times 2 \text{ (内存通道)} \times \\
 &\quad 8 \text{ (每次内存访问的字节数)} \times 1 \text{ (插槽)} = 38.4 \text{ GiB/s}
 \end{aligned}$$

像 Empirical Roofline Tool: <https://bitbucket.org/berkeleylab/cs-roofline-toolkit/src/master/><sup>69</sup> 和 Intel Advisor: <https://software.intel.com/content/www/us/en/develop/tools/advisor.html><sup>70</sup> 这样的自动化工具能够通过运行一组预先准备的基准测试来经验性地确定理论最大值。如果一个计算可以重用缓存中的数据，则可以实现更高的 FLOP 速度。Roofline 可以通过为每个内存层次引入专门的 roofline 来实现这一点（参见图 35）。

确定硬件限制后，我们可以开始评估应用程序相对于 roofline 的性能。用于自动收集 Roofline 数据的两种最常用方法是采样（由 likwid: <https://github.com/RRZE-HPC/likwid><sup>71</sup> 工具使用）和二进制插桩（由 Intel 软件开发仿真器 (SDE: <https://software.intel.com/content/www/us/en/develop/articles/intel-software-development-emulator.html><sup>72</sup> ) 使用）。采样在数据收集方面产生的开销较低，而二进制插桩则能提供更准确的结果。<sup>73</sup> Intel Advisor 自动构建 Roofline 图表，并为给定循环的性能优化提供提示。图 35 展示了 Intel Advisor 生成的 Roofline 图表示例。请注意，Roofline 图表使用的是对数刻度。

Roofline 方法可以通过在同一个图表上打印“之前”和“之后”的点来跟踪优化进度。因此，它是一个迭代的过程，指导开发人员帮助他们的应用程序充分利用硬件功能。图 35 显示了对之前在 Listing ?? 中显示的代码进行以下两个更改所带来的性能提升：

- 交换两个最内层的循环（交换第 4 和第 5 行）。这可以实现缓存友好的内存访问（参见 Chapter 7）。
- 使用 AVX2 指令启用最内层循环的自动矢量化。

总结来说，Roofline 性能模型可以帮助：

- 识别性能瓶颈。
- 指导软件优化。
- 确定优化何时结束。
- 相对于机器能力评估性能。

其他资源和链接：

- NERSC 文档，网址：<https://docs.nersc.gov/development/performance-debugging-tools/roofline/>。
- 劳伦斯伯克利国家实验室研究，网址：<https://crd.lbl.gov/departments/computer-science/par/research/roofline/>
- 关于 Roofline 模型和 Intel Advisor 的视频演示集合，网址：<https://techdecoded.intel.io/> (搜索“Roofline”)。
- Perfplot 是一个脚本和工具集合，允许用户在最近的 Intel 平台上测量性能计数器，并使用结果生成 roofline 和性能图。网址：<https://github.com/GeorgOfenbeck/perfplot>

<sup>69</sup> Empirical Roofline Tool - <https://bitbucket.org/berkeleylab/cs-roofline-toolkit/src/master/>.

<sup>70</sup> Intel Advisor - <https://software.intel.com/content/www/us/en/develop/tools/advisor.html>.

<sup>71</sup> Likwid - <https://github.com/RRZE-HPC/likwid>.

<sup>72</sup> Intel SDE - <https://software.intel.com/content/www/us/en/develop/articles/intel-software-development-emulator.html>.

<sup>73</sup> 在此演示文稿中，可以看到更详细的收集 roofline 数据的方法比较：<https://crd.lbl.gov/assets/Uploads/ECP20-Roofline-4-cpu.pdf>

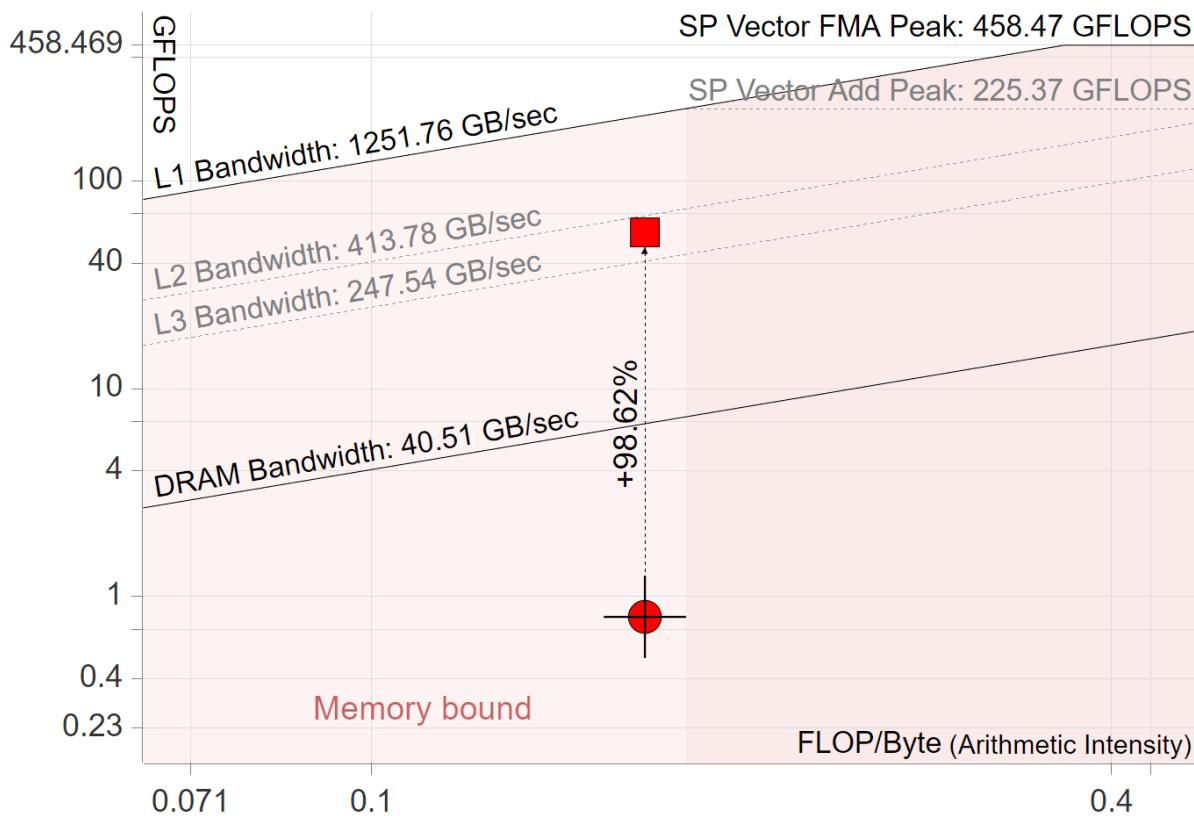


Figure 35: Roofline analysis for matrix multiplication on Intel NUC Kit NUC8i5BEH with 8GB RAM using clang 10 compiler.

## 5.7 静态性能分析

如今，我们拥有广泛的静态代码分析工具。对于 C 和 C++ 语言，我们有一些著名的工具，例如 Clang 静态分析器: <https://clang-analyzer.llvm.org/>、Klocwork: <https://www.perforce.com/products/klocwork>、Cppcheck: <http://cppcheck.sourceforge.net/> 等。它们旨在检查代码的正确性和语义。同样，也有一些工具试图解决代码的性能方面的问题。静态性能分析器不会执行或分析程序，而是模拟代码，就好像它在真实硬件上执行一样。静态预测性能几乎是不可能的，因此这种类型的分析有很多限制。

首先，由于我们不知道要编译成的机器代码，所以不可能静态分析 C/C++ 代码的性能。因此，静态性能分析针对的是汇编代码。

其次，静态分析工具模拟工作负载而不是执行它。这显然非常慢，因此不可能静态分析整个程序。相反，工具会取一小段汇编代码，并试图预测它在真实硬件上的行为。用户应该选择特定的汇编指令（通常是小型循环）进行分析。因此，静态性能分析的范围非常窄。

静态性能分析器的输出相当低级，有时会将执行分解到 CPU 周期。通常，开发人员将其用于关键代码区域的细粒度调整，其中每个 CPU 周期都很重要。

### 静态分析器 vs. 动态分析器

**静态工具:** 不运行实际代码，而是尝试模拟执行，尽可能保留微架构细节。它们无法进行实际测量（执行时间、性能计数器），因为它们不运行代码。优点是您不需要拥有真正的硬件，可以针对不同代的 CPU 模拟代码。另一个好处是您不必担心结果的一致性：静态分析器总是会给您确定性的输出，因为模拟（与实际硬件上的执行相比）不会出现任何偏差。静态工具的缺点是它们通常无法预测和模拟现代 CPU 中的所有内容：它们

基于一个可能存在错误和限制的模型。静态性能分析器的例子包括 UICA: <https://uica.uops.info/><sup>74</sup> 和 llvm-mca: <https://llvm.org/docs/CommandGuide/llvm-mca.html><sup>75</sup>。

**动态工具:** 基于在真实硬件上运行代码并收集有关执行的所有信息。这是证明任何性能假设的唯一 100% 可靠的方法。缺点是,通常您需要具有特权访问权限才能收集低级性能数据,例如 PMCs。编写一个好的基准测试并测量您想要测量的内容并不总是容易的。最后,您需要过滤噪音和不同类型的副作用。动态微架构性能分析器的例子包括 nanoBench: <https://github.com/andreas-abel/nanoBench>,<sup>76</sup> uarch-bench: <https://github.com/travisdowns/uarch-bench><sup>77</sup> 等。

一个更大的静态和动态微架构性能分析工具集合可以在 [这里](https://github.com/MattPD/cpplinks/blob/master/performance.tools.md#microarchitecture): <https://github.com/MattPD/cpplinks/blob/master/performance.tools.md#microarchitecture><sup>78</sup> 获得。

### 5.7.1 案例研究: 使用 UICA 优化 FMA 吞吐量

开发人员经常会问的一个问题是:“最新处理器拥有 10 多个执行单元;我该如何编写代码让它们一直保持繁忙?”这确实是一个最难解决的问题之一。有时它需要仔细观察程序如何运行。UICA 模拟器就是这样一个显微镜,可以让您深入了解您的代码如何流经现代处理器。

让我们看一下 Listing 5.7.1 中的代码。我们有意使示例尽可能简单。当然,现实世界中的代码通常比这更复杂。该代码将数组 `a` 的每个元素乘以常数 `B`,并将缩放后的值累积到 `sum` 中。在右侧,我们展示了使用 `-O3 -ffast-math -march=core-avx2` 编译时 Clang-16 生成的循环的机器代码。汇编代码看起来非常紧凑,让我们更好地理解它。

清单: FMA 吞吐量

```

1 float foo(float * a, float B, int N){ .loop:
2     float sum = 0;           vfmadd231ps ymm2, ymm1, ymmword [rdi + rsi]
3     for (int i = 0; i < N; i++) vfmadd231ps ymm3, ymm1, ymmword [rdi + rsi + 32]
4         sum += a[i] * B;    vfmadd231ps ymm4, ymm1, ymmword [rdi + rsi + 64]
5     return sum;            vfmadd231ps ymm5, ymm1, ymmword [rdi + rsi + 96]
6 }                         sub rsi, -128
7                           cmp rdx, rsi
8                           jne .loop

```

这段代码是一个归约循环,即我们需要对所有乘积求和,最终返回一个浮点数。按照目前代码的写法, `sum` 上存在循环传递依赖性。您无法覆盖 `sum`,直到累积上一个乘积。一种并行化的巧妙方法是使用多个累加器并在最后将它们汇总。因此,我们可以用多个累加器代替单个 `sum`,例如 `sum1` 用于累积偶数次迭代的结果, `sum2` 用于累积奇数次迭代的结果。这就是 Clang-16 所做的:它使用了 4 个向量寄存器 (`ymm2-ymm5`),每个都包含 8 个浮点累加器,并使用 FMA 将乘法和加法融合成单个指令。常量 `B` 被广播到 `ymm1` 寄存器中。`-ffast-math` 选项允许编译器重新关联浮点运算,我们将在 Section 8.4 中讨论这个选项如何帮助优化。顺便说一句,乘法在循环后只需要做一次。这肯定是程序员的疏忽,但希望编译器将来能够处理它。

代码看起来不错,但它真的是最优的吗?让我们找出答案。我们将 Listing 5.7.1 中的汇编代码片段带到 UICA 进行模拟。在撰写本文时, UICA 不支持 Alderlake (英特尔第 12 代, 基于 GoldenCove), 因此我们在最新可用的 RocketLake (英特尔第 11 代, 基于 SunnyCove) 上运行了它。虽然架构不同,但这次实验暴露的问题在两者上都同样明显。模拟结果如图 36 所示。这是一个类似于我们在第 3 章中展示的管道图。我们跳过了前两个迭代,只显示了第 2 和第 3 个迭代 (最左列 “It.”)。这时,执行已经达到稳定状态,所有后续迭代看起来都非常相似。

<sup>74</sup> UICA - <https://uica.uops.info/>

<sup>75</sup> LLVM MCA - <https://llvm.org/docs/CommandGuide/llvm-mca.html>

<sup>76</sup> nanoBench - <https://github.com/andreas-abel/nanoBench>

<sup>77</sup> uarch-bench - <https://github.com/travisdowns/uarch-bench>

<sup>78</sup> C++ 性能工具链接集合 - <https://github.com/MattPD/cpplinks/blob/master/performance.tools.md#microarchitecture>.

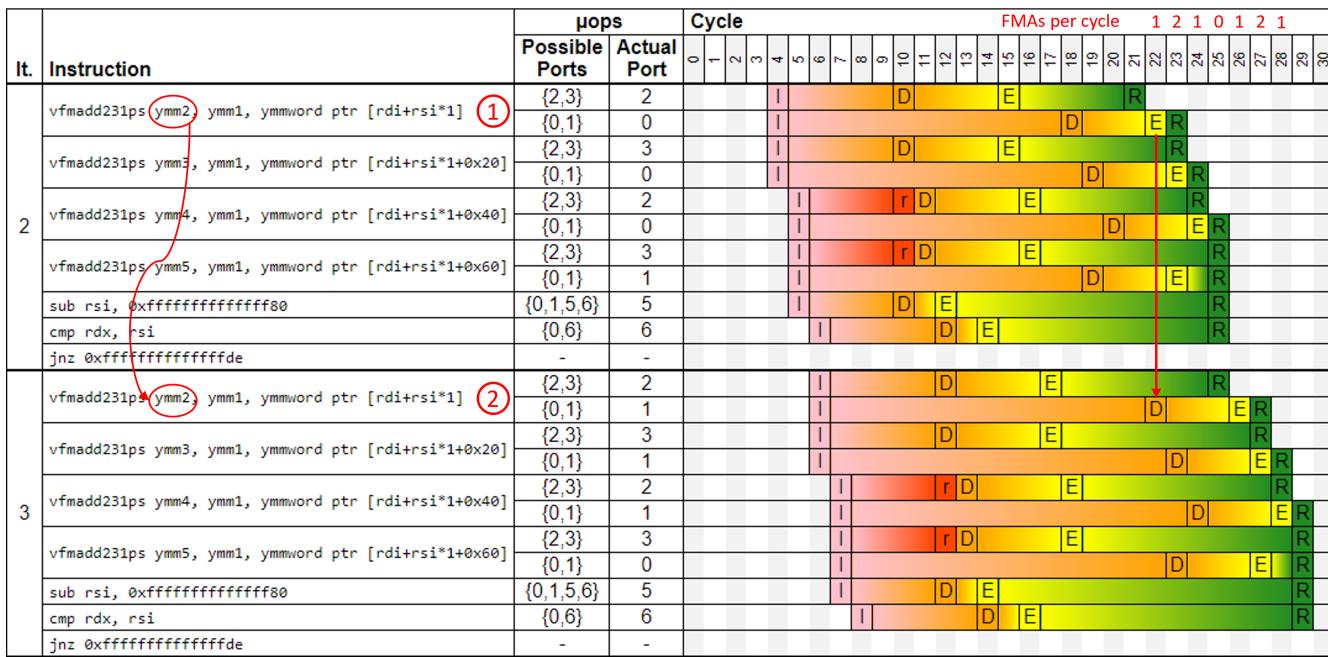


Figure 36: UICA pipeline diagram. I = issued, r = ready for dispatch, D = dispatched, E = executed, R = retired.

UICA 是一个非常简化了的实际 CPU 管道模型。例如，您可能会注意到指令提取和译码阶段丢失了。此外，UICA 不考虑缓存未命中和分支预测错误，因此它假设所有内存访问总是命中 L1 缓存并且分支总是预测正确。我们都知道这在现代处理器中并非如此。同样，这与我们的实验无关，因为我们仍然可以使用模拟结果来找到改进代码的方法。您能看到问题吗？

让我们仔细看看这个图表。首先，每个 FMA 指令都被分解成两个微操作（见 ①）：一个加载微操作，进入端口 {2,3}，一个 FMA 微操作，可以进入端口 {0,1}。加载微操作的延迟是 5 个周期：从第 11 个周期开始到第 15 个周期结束。FMA 微操作的延迟是 4 个周期：从第 19 个周期开始到第 22 个周期结束。所有 FMA 微操作都依赖于加载微操作，我们可以在图表上清楚地看到这一点：FMA 微操作总是对应加载微操作完成后才开始。现在找到第 10 个周期中的两个 r 单元格，它们已经准备调度，但是 RocketLake 只有两个加载端口，并且在同一个周期都被占用了。因此，这两个加载指令在下个周期发出。

该循环在 ymm2-ymm5 上具有四个跨迭代依赖性。来自指令 ② 的写入 ymm2 的 FMA 微操作无法在上一迭代的指令 ① 完成之前开始执行。请注意，来自指令 ② 的 FMA 微操作与指令 ① 完成执行的同一个周期 22 被调度。您也可以观察其他 FMA 指令的这种模式。

那么，您可能会问，“问题是什么？”请看图片的右上角。对于每个周期，我们都计算了已执行的 FMA 微操作的数量，这不是 UICA 打印的。它看起来像 1, 2, 1, 0, 1, 2, 1, …，或者平均每个周期 1 个 FMA 微操作。最近的英特尔处理器大多有两个 FMA 执行单元，因此每个周期可以发出两个 FMA 微操作。该图表清楚地显示了差距，因为每个第四个周期都没有执行 FMA 指令。正如我们之前发现的，由于它们的输入 (ymm2-ymm5) 没有准备好，因此无法调度任何 FMA 微操作。

为了将 FMA 执行单元的利用率从 50% 提高到 100%，我们需要将循环展开两倍。这将使累加器的数量从 4 个增加到 8 个。此外，我们将有 8 个独立的数据流链，而不是 4 个。我们这里不会展示展开版本的模拟，您可以自己尝试。相反，让我们通过在真实硬件上运行两个版本来确认假设。顺便说一句，这是一个好主意，因为 UICA 等静态性能分析器并不是准确的模型。下面，我们展示了我们在最近的 Alderlake 处理器上运行的两个 nanobench：<https://github.com/andreas-abel/nanoBench> 测试的输出。该工具采用提供的汇编指令 (-asm 选项) 并创建一个 benchmark 内核。读者可以查阅 nanobench 文档中其他参数的含义。左侧的原始代码在 4 个周期内执行 4 条指令，

而改进后的版本可以在 4 个周期内执行 8 条指令。现在我们可以确定我们最大化了 FMA 执行吞吐量，右侧的代码使 FMA 单元始终处于忙碌状态。

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

Instructions retired: 4.20
Core cycles: 4.02

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

Instructions retired: 8.20
Core cycles: 4.02
```

作为经验法则，在这种情况下，循环必须按  $T * L$  的倍数展开，其中  $T$  是指令的吞吐量， $L$  是其延迟。在我们的案例中，由于 Alderlake 上 FMA 的吞吐量为 2，延迟为 4 个周期，因此我们应该将其展开  $2 * 4 = 8$  倍以实现最大 FMA 端口利用率。这会创建 8 个可以独立执行的单独数据流链。

值得一提的是，在实践中您并不总是会看到 2 倍的加速。这只能在 UICA 或 nanobench 等理想化环境中实现。在实际应用程序中，即使您最大化了 FMA 的执行吞吐量，收益也可能会受到最终缓存未命中和其他流水线冲突的阻碍。发生这种情况时，缓存未命中的影响会超过 FMA 端口利用率不理想的影响。这很容易导致令人失望的 5% 速度提升。但别担心，你仍然做对了。

最后，让我们提醒您，UICA 或任何其他静态性能分析器都不适合分析大段代码。但它们非常适合探索微架构效应。此外，它们还可以帮助您建立 CPU 工作方式的心理模型。UICA 的另一个非常重要的用例是在循环中找到关键依赖性链，正如 easyperf 博客的文章：<https://easyperf.net/blog/2022/05/11/Visualizing-Performance-Critical-Dependency-Chains><sup>79</sup> 中所述。

## 5.8 编译器优化报告

如今，软件开发在很大程度上依赖编译器进行性能优化。编译器在加速软件方面扮演着关键角色。大多数开发人员将优化代码的工作留给编译器，只有当他们发现编译器无法完成的优化机会时才会干预。可以说，这是一个好的默认策略。但是，当您追求最佳性能时，它就不太管用了。如果编译器没有执行关键优化，例如矢量化循环，怎么办？您将如何知道这一点？幸运的是，所有主流编译器都提供优化报告，我们现在将讨论这些报告。

假设您想知道一个关键循环是否被展开。如果是，展开因子是多少？有一种艰苦的方法可以知道这一点：研究生成的汇编指令。不幸的是，并不是每个人都习惯于阅读汇编语言。如果函数很大，它调用其他函数或也有许多被矢量化的循环，或者如果编译器为同一个循环创建了多个版本，这可能会特别困难。大多数编译器，包括 GCC、Clang 和 Intel 编译器（但不包括 MSVC），都提供优化报告，用于检查特定代码段执行了哪些优化。

让我们看一下 Listing 5.8，它展示了一个由 clang 16.0 未矢量化的循环示例。

代码清单：a.c

<sup>79</sup> Easyperf 博客 - <https://easyperf.net/blog/2022/05/11/Visualizing-Performance-Critical-Dependency-Chains>

```

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 }
```

在 clang 中生成优化报告，您需要使用 `-Rpass*`: <https://llvm.org/docs/Vectorizers.html#diagnostics> 标志：

```

$ clang -O3 -Rpass-analysis=.* -Rpass=.* -Rpass-missed=.* a.c -c
a.c:5:3: remark: loop not vectorized [-Rpass-missed=loop-vectorize]
for (unsigned i = 1; i < N; i++) {
^

a.c:5:3: remark: unrolled loop by a factor of 8 with run-time trip count [-Rpass=loop-unroll]
for (unsigned i = 1; i < N; i++) {
^
```

检查上面的优化报告，我们可以看到循环没有被矢量化，而是被展开了。开发人员并不总是很容易识别 Listing 5.8 第 6 行循环中是否存在循环进位依赖。由 `c[i-1]` 加载的值取决于前一次迭代的存储（参见图 37 中的操作② 和 ③）。可以通过手动展开循环的前几个迭代来揭示依赖关系：

```

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

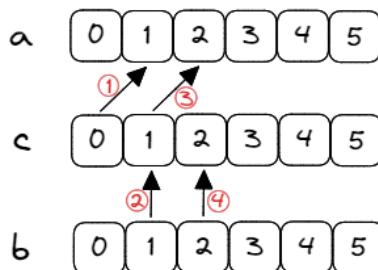


Figure 37: Visualizing the order of operations in Listing 5.8.

如果我们将 Listing 5.8 中的代码矢量化，它会导致在数组 `a` 中写入错误的值。假设 CPU SIMD 单元可以一次处理四个浮点数，我们可以得到可以用以下伪代码表示的代码：

```

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

Listing 5.8 中的代码无法矢量化，因为循环内部的操作顺序很重要。如 Listing 5.8 所示，通过交换第 6 行和第 7 行可以修复此示例。这不会改变代码的语义，所以这是一个完全合法的更改。另外，可以通过将循环拆分成两个单独的循环来改善代码。

代码清单: 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 }
```

在优化报告中，我们可以看到循环成功向量化了：

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

这只是使用优化报告的一个例子，我们将在本书的第二部分讨论发现矢量化机会时更详细地介绍。编译器优化报告可以帮助您找到错过的优化机会，并了解这些机会错过的原因。此外，编译器优化报告对于测试假设很有用。编译器通常会根据其成本模型分析来决定某个转换是否有益。但编译器并不总是做出最佳选择。一旦您在报告中发现缺少关键优化，您可以尝试通过更改源代码或向编译器提供提示（例如 `#pragma`、属性、编译器内置函数等）来纠正它。始终通过在实际环境中进行测量来验证您的假设。

编译器报告可能相当庞大，每个源代码文件都会生成单独的报告。有时，在输出文件中找到相关记录可能成为一项挑战。我们应该提到，最初这些报告的设计明确供编译器编写者用于改进优化过程。多年来，已经出现了一些工具，使它们更易于应用程序开发人员访问和操作。最值得注意的是 `opt-viewer`<sup>80</sup> 和 `optview2`<sup>81</sup>。此外，Compiler Explorer 网站还为基于 LLVM 的编译器提供了“优化输出”工具，当您将鼠标悬停在源代码相应行上时，它会报告执行的转换。所有这些工具都帮助可视化基于 LLVM 的编译器成功的和失败的代码转换。

在 LTO<sup>82</sup> 模式下，一些优化是在链接阶段进行的。为了同时从编译和链接阶段发出编译器报告，应该向编译器和链接器传递专用选项。有关更多信息，请参见 LLVM “remarks”<sup>83</sup> 指南。

Intel® ISPC<sup>84</sup> 编译器（已在 Section 8.4.2.5 中讨论）采用稍微不同的方式报告缺失的优化。它会针对编译为相对低效代码的代码结构发出警告。无论哪种方式，编译器优化报告都应该是您工具箱中的关键工具之一。它是一种快速的方法，可以检查对特定热点进行了哪些优化，以及是否失败了一些重要的优化。许多改进机会都是通过编译器优化报告发现的。

<sup>80</sup> opt-viewer - <https://github.com/llvm/llvm-project/tree/main/llvm/tools/opt-viewer>

<sup>81</sup> optview2 - <https://github.com/OfekShilon/optview2>

<sup>82</sup> 链接时间优化，也称为过程间优化 (IPO)。阅读更多: [https://en.wikipedia.org/wiki/Interprocedural\\_optimization](https://en.wikipedia.org/wiki/Interprocedural_optimization)

<sup>83</sup> LLVM compiler remarks - <https://llvm.org/docs/Remarks.html>

<sup>84</sup> ISPC - <https://ispc.github.io/ispc.html>

## 问题和练习

1. 在以下场景中你会使用哪些方法?
  - 场景 1：客户支持团队报告客户问题：升级到新版本应用程序后，特定操作的性能下降了 10%。
  - 场景 2：客户支持团队报告客户问题：某些交易完成时间比平时长 2 倍，没有特定模式。
  - 场景 3：您正在评估三种不同的压缩算法，想知道每种算法都存在哪些类型的性能瓶颈（内存延迟/带宽、分支预测错误等）。
  - 场景 4：有一个新的闪亮库声称比您目前项目中集成的库更快；您决定比较它们的性能。
  - 场景 5：您被要求分析不熟悉代码的性能；想知道某个分支被采取的频率以及循环执行的迭代次数。
2. 运行您每天使用的应用程序。使用本章讨论的方法进行性能分析练习。收集各种 CPU 性能事件的原始计数，找到热点，收集顶线数据，并生成和研究程序中热点函数的编译器优化报告。

## 章节总结

- 延迟和吞吐量通常是程序性能的最终指标。当寻求改善它们的方法时，我们需要获取更多关于应用程序如何执行的详细信息。硬件和软件都提供可用于性能监视的数据。
- 代码检测允许我们跟踪程序中的许多内容，但会在开发和运行时都造成相对较大的开销。虽然现在开发人员很少手动检测他们的代码，但这种方法仍然与自动化流程相关，例如 PGO。
- 跟踪在概念上类似于检测，可用于探索系统中的异常。跟踪允许我们捕获整个事件序列，并在每个事件上附加时间戳。
- 工作负载特征化是一种基于应用程序运行时行为进行比较和分组的方法。一旦进行特征化，就可以遵循特定的方法来找到程序中的优化空间。带有标记 API 的分析工具可用于分析特定代码区域的性能。
- 采样跳程序执行的大部分，只获取一个样本，该样本应该代表整个区间。尽管如此，采样通常会产生足够精确的分布。采样最著名的用例是找到代码中的热点。采样是最流行的分析方法，因为它不需要重新编译程序，并且运行时开销非常小。
- 通常，计数和采样会产生非常低的运行时开销（通常低于 2%）。一旦您开始在不同事件之间进行多路复用，计数就会变得更加昂贵（5-15% 的开销），采样会随着采样频率的增加而变得更加昂贵 [Nowak & Bitzes, 2014]。考虑使用用户模式采样来分析长时间运行的工作负载或您不需要非常准确的数据时。
- Roofline 性能模型是一个面向吞吐量的性能模型，在高性能计算 (HPC) 领域得到了广泛使用。它允许绘制应用程序性能与硬件限制之间的关系图。Roofline 模型有助于识别性能瓶颈，指导软件优化，并跟踪优化进度。
- 有些工具试图静态分析代码的性能。此类工具模拟一段代码而不是执行它。这种方法存在许多限制和约束，但您会得到非常详细和低级别的报告作为回报。
- 编译器优化报告有助于发现丢失的编译器优化。这些报告还指导开发人员设计新的性能实验。

## 5.9 CPU 特性用于性能分析

性能分析的最终目标是识别性能瓶颈并定位与之相关的代码部分。不幸的是，没有预定的步骤可供遵循，因此可以采用多种不同的方法。

通常，对应用程序进行性能分析可以快速了解应用程序的热点。有时，这是开发人员修复性能低效率所需要做的全部工作。特别是高级性能问题通常可以通过性能分析来揭示。例如，考虑这样一个情况：您刚刚对应用程序中的函数 `foo` 进行了一些更改，突然发现性能明显下降。因此，您决定对应用程序进行性能分析。根据您对应用程序的心理模型，您期望 `foo` 是一个冷函数，它不会出现在热门函数的前 10 名列表中。但是当您打开性能分析结果时，您会发现它消耗的时间比以前多得多。您很快意识到自己在代码中犯了错误并修复了它。如果性能工程中的所有问题都这么容易修复，这本书就不会存在了。

当您踏上从应用程序中榨取最后一点性能的旅程时，最基本的热点列表是不够的。除非你有一个水晶球或者你脑海中有一个完整处理器的准确模型，你需要额外的支持来理解性能瓶颈是什么。然而，在使用本章提供的信息之前，请确保您尝试优化的应用程序没有重大性能缺陷。因为如果确实存在，使用 CPU 性能监控功能进行低级别优化是没有意义的。它可能会让你走错方向，而不是修复真实的高级性能问题，你将只会优化糟糕的代码，这只是浪费时间。

一些开发人员依赖他们的直觉，并进行随机实验，试图强制各种编译器优化，如循环展开、矢量化、内联化等等。事实上，有时你可能会很幸运，并会享受来自同事的一部分赞扬，甚至可能在你的团队中获得非官方的性能大师称号。但通常，你需要有非常好的直觉和运气。在这本书中，我们不教你如何变得幸运。相反，我们展示了在实践中被证明行之有效的方法。

现代 CPU 不断增加新特性，以不同方式增强性能分析。使用这些特性可以大大简化查找缓存未命中、分支预测错误等低级问题。在本章中，我们将介绍现代 CPU 上的一些硬件性能监控功能。不同厂商的处理器不一定具有相同的特性集。在本章中，我们将重点关注英特尔、AMD 和 ARM 处理器中可用的性能监控功能。RISC-V 生态系统还没有成熟的性能监控基础设施，因此我们这里不会涉及。

- 自顶向下微架构分析 (TMA) 方法，详见 Section ??。这是一种强大的技术，用于识别程序对 CPU 微架构利用效率低下的情况。它描述了工作负载的瓶颈，并允许定位源代码中发生瓶颈的确切位置。它抽象了 CPU 微架构的复杂性，即使对于经验不足的开发人员来说也相对容易使用。
- 最近分支记录 (LBR)，详见 Section 5.11。这是一个机制，可以与执行程序同时连续记录最近的分支结果。它用于收集调用堆栈、识别热门分支、计算单个分支的错误预测率等等。
- 处理器事件采样 (PEBS)，详见 Section ??。这是一个增强采样的特性。其主要优点包括：降低采样的开销；并提供“精确事件”功能，使其能够精准定位导致特定性能事件的指令。
- 英特尔处理器跟踪 (PT)，详见附录 D。它是一个记录和重建程序执行的工具，每条指令都会记录时间戳。其主要用途是事后分析和根源分析性能问题。

英特尔 PT 特性在附录 D 中介绍。英特尔 PT 本来应该是性能分析的“终结手段”。由于其运行时开销低，它是一个非常强大的分析功能。但事实证明它在性能工程师中并不受欢迎。部分原因是工具中的支持还不成熟，部分原因是在很多情况下它都是一种杀鸡用牛刀，使用采样分析器更容易。此外，它会产生大量数据，这不适合长时间运行的工作负载。

上述特性从 CPU 的角度提供了程序效率的见解。在下一章中，我们将讨论性能分析工具如何利用它们提供多种不同类型的性能分析。

## 5.10 自顶向下微架构分析 (TMA)

自顶向下微架构分析 (TMA) 方法是一种非常强大的技术，用于识别程序中的 CPU 瓶颈。它是一种健壮、正式的方法，即使是经验不足的开发人员也易于使用。该方法最棒的一点是，它不需要开发人员深入了解系统中的微架构和 PMCs，即可高效找到 CPU 瓶颈。

从概念层面来看，TMA 识别导致程序执行停滞的原因。图 38 展示了 TMA 的核心思想。这不是实际分析的运作方式，因为分析每个微操作 ( $\mu$ op) 会非常慢。尽管如此，该图有助于理解该方法。

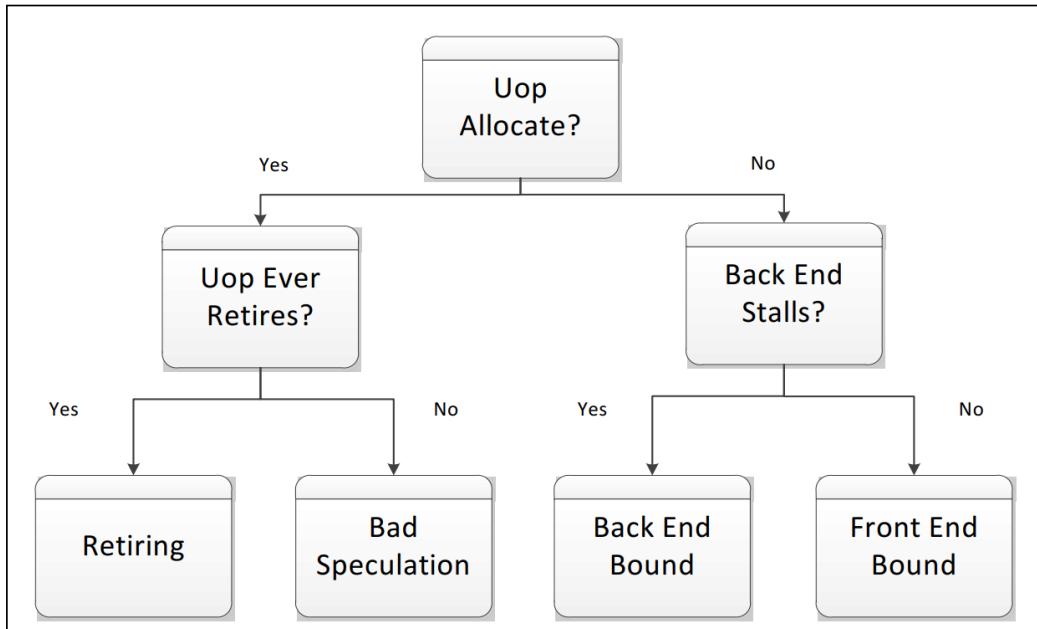


Figure 38: TMA 顶级细分的概念。(© 图像来自 [Yasin, 2014])

下面是如何阅读此图的简短指南。正如我们从 Chapter 3 中所知，CPU 内部有缓冲区来跟踪正在执行的  $\mu$ op 的信息。每当获取和解码新指令时，都会在这些缓冲区中分配新条目。如果指令的  $\mu$ op 在特定执行周期内未分配，可能是两个原因之一：我们无法获取和解码它（“前端受限”）；或者后端工作超载，无法为新的  $\mu$ op 分配资源（“后端受限”）。如果  $\mu$ op 已分配并安排执行但从未退休，则意味着它来自预测错误的路径（“错误猜测”）。最后，“退休”代表正常执行。它们是我们希望所有  $\mu$ op 都处的桶，尽管有一些例外情况，我们稍后会讨论。

为了实现其目标，TMA 通过监控特定的一组性能事件并根据预定义的公式计算指标来观察程序的执行。基于这些指标，TMA 通过将程序分配到四个高级桶之一来对其进行表征。这四个高级类别中的每一个都具有多个嵌套级别，CPU 供应商可以选择不同的实现方式。每代处理器计算这些指标的公式可能会有所不同，因此最好依赖工具进行分析，而不是自己尝试计算。

在接下来的部分中，我们将讨论 AMD、ARM 和 Intel 处理器中的 TMA 实现。

### 5.10.1 在英特尔平台上的 TMA

TMA 方法首次由英特尔于 2014 年提出，并从 SandyBridge 系列处理器开始提供支持。英特尔的实现支持每个高级桶的嵌套类别，从而更好地了解程序中的 CPU 性能瓶颈（参见图 39）。

该工作流程旨在“深入挖掘 (drill down)”TMA 层次结构的较低级别，直到我们达到对性能瓶颈的非常具体的分类为止。例如，首先，我们收集主要的四个桶的指标：Front End Bound、Back End Bound、Retiring、Bad Speculation。比如，我们发现程序执行的大部分时间被内存访问阻塞了（这是 Back End Bound 桶，参见图 39）。接下来的步骤是再次运行工作负载，并仅收集与 Memory Bound 桶有关的特定指标。这个过程重复进行，直到我们知道确切的根本原因，例如，L3 Bound。

多次运行工作负载并在每次运行时专注于特定指标是完全可以的。但通常，运行一次工作负载并收集所有 TMA 各级别所需的所有指标就足够了。性能分析工具通过在单次运行中在不同性能事件之间进行多路复用（参见 Section 5.3.3）来实现这一点。此外，在现实应用中，性能可能受到多个因素的限制。例如，它可以同时经历大量的分支错误预测

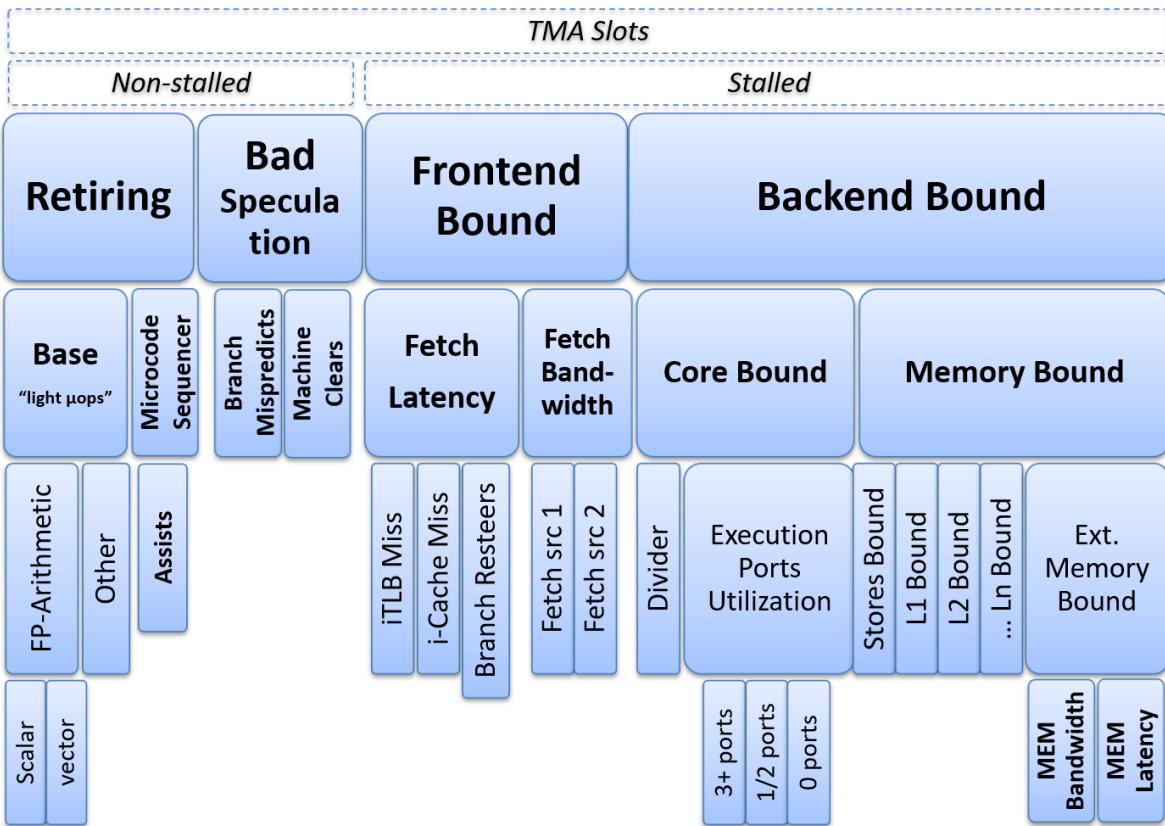


Figure 39: TMA 性能瓶颈的层次结构。© Image by Ahmad Yasin.

(Bad Speculation) 和缓存丢失 (Back End Bound)。在这种情况下, TMA 将同时深入挖掘多个桶, 并确定每种类型的瓶颈对程序性能的影响。像英特尔的 VTune Profiler、AMD 的 uProf 和 Linux 的perf等分析工具可以在单次基准测试运行中计算所有 TMA 指标。但是, 这仅在工作负载稳定时才可行。否则, 最好回到多次运行和每次运行都进行深入挖掘的原始策略。

TMA 的前两个级别的指标以所有流水线插槽的百分比表示 (参见 Section ??), 这些插槽在程序执行期间是可用的。这使得 TMA 能够准确表示 CPU 微体系结构的利用率, 考虑到处理器的全部带宽。到目前为止, 所有内容都应该很好地总结为 100%。然而, 从第 3 级开始, 桶可能以不同的计数域表示, 例如时钟和停顿。因此, 它们不一定与其他 TMA 桶直接可比。

TMA 的第一步是识别程序中的性能瓶颈。在完成这一步之后, 我们需要知道问题具体出现在代码的哪里。TMA 的第二步是将问题的源头定位到代码的确切行和汇编指令。该分析方法提供了应针对性能问题的每个类别使用的确切性能事件。然后, 您可以在此事件上进行采样, 以找到在第一阶段识别的性能瓶颈所在的源代码行。如果这个过程让您感到困惑, 不用担心, 阅读案例研究后一切都会变得清晰。

## 案例研究：使用 TMA 减少缓存未命中数量

作为本案例研究的示例, 我们采用了非常简单的基准测试, 使其易于理解和更改。它显然不能代表现实世界的应用程序, 但足以演示 TMA 的工作流程。本书的第二部分中有更多实用的例子。

本书的大部分读者可能会将 TMA 应用于他们熟悉自己的应用程序。但即使您是第一次看到该应用程序, TMA 也非常有效。因此, 我们不会首先向您展示基准测试的原始源代码。但这里有一个简短的描述: 基准测试在堆上分配了一个 200 MB 的数组, 然后进入一个 100M 次迭代的循环。在循环的每次迭代中, 它都会生成一个指向已分配数组的随机索引, 执行一些虚拟工作, 然后从该索引读取值。

我们使用配备有 Intel Core i5-8259U CPU (基于 Skylake) 和 16GB DRAM (DDR4 2400 MT/s) 的机器运行实验, 运行 64 位 Ubuntu 20.04 (内核版本 5.13.0-27)。

### 步骤 1: 识别瓶颈

作为第一步, 我们运行微基准测试并收集一组有限的事件, 这些事件将帮助我们计算第 1 级指标。在这里, 我们尝试通过将它们归因于四个 L1 桶 (“前端受限”、“后端受限”、“退休”、“错误猜测”) 来识别应用程序的高级性能瓶颈。可以使用 Linux perf 工具收集第 1 级指标。从 Linux 内核 4.8 开始, perf 在 perf stat 命令中有一个 --topdown 选项, 用于打印 TMA 第 1 级指标。以下是我们基准测试的细分。本部分的命令输出经过修剪以节省空间。

[TODO]: 在 AlderLake 上, perf stat --topdown 无法在内核 4.8 上工作, 需要更新版本。现在它可以打印 L1 和 L2 TMA 指标。(请参阅 <https://github.com/dendibakh/perf-book/issues/42>)

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

为了获得高级 TMA 指标的值, Linux perf 需要分析整个系统 (-a)。这就是为什么我们看到所有内核的指标。但是由于我们已经使用 taskset -c 0 将基准测试固定在核心 0 上, 因此我们只需要关注与 S0-C0 对应的行。我们可以丢弃其他行, 因为它们正在运行其他任务或处于空闲状态。通过查看输出, 我们可以判断应用程序的性能受 CPU 后端限制。现在先不进行分析, 让我们向下钻取一层。

Linux perf 只支持 1 级 TMA 指标，因此要访问 2、3 级及更高级别的 TMA 指标，我们将使用 toplev 工具，它是 Andi Kleen 编写的 pmu-tools: <https://github.com/andikleen/pmu-tools><sup>85</sup> 的一部分。它用 Python 实现，并在幕后调用 Linux perf。要使用 toplev，必须启用特定的 Linux 内核设置，有关详细信息，请查看文档。

```
$ ~/pmu-tools/toplev.py --core S0-C0 -12 -v --no-desc taskset -c 0 ./benchmark.exe
...
# Level 1
S0-C0 Frontend_Bound:           13.92 % Slots
S0-C0 Bad_Speculation:          0.23 % Slots
S0-C0 Backend_Bound:            53.39 % Slots
S0-C0 Retiring:                 32.49 % Slots
# Level 2
S0-C0 Frontend_Bound.FE_Latency: 12.11 % Slots
S0-C0 Frontend_Bound.FE_Bandwidth: 1.84 % Slots
S0-C0 Bad_Speculation.Branch_Mispred: 0.22 % Slots
S0-C0 Bad_Speculation.Machine_Clears: 0.01 % Slots
S0-C0 Backend_Bound.Memory_Bound: 44.59 % Slots <=
S0-C0 Backend_Bound.Core_Bound:   8.80 % Slots
S0-C0 Retiring.Base:             24.83 % Slots
S0-C0 Retiring.Microcode_Sequencer: 7.65 % Slots
```

在此命令中，我们还将进程固定到 CPU0（使用 taskset -c 0），并将 toplev 的输出仅限于此核心（--core S0-C0）。选项 -12 告诉工具收集 Level 2 指标。选项 --no-desc 禁用每个指标的描述。

我们可以看到，应用程序的性能受内存访问限制（Backend\_Bound.Memory\_Bound）。近一半的 CPU 执行资源都浪费在等待内存请求完成上。现在让我们更深入地挖掘一次：<sup>86</sup>

```
$ ~/pmu-tools/toplev.py --core S0-C0 -13 -v --no-desc taskset -c 0 ./benchmark.exe
...
# Level 1
S0-C0 Frontend_Bound:           13.91 % Slots
S0-C0 Bad_Speculation:          0.24 % Slots
S0-C0 Backend_Bound:            53.36 % Slots
S0-C0 Retiring:                 32.41 % Slots
# Level 2
S0-C0 FE_Bound.FE_Latency:      12.10 % Slots
S0-C0 FE_Bound.FE_Bandwidth:     1.85 % Slots
S0-C0 BE_Bound.Memory_Bound:    44.58 % Slots
S0-C0 BE_Bound.Core_Bound:       8.78 % Slots
# Level 3
S0-C0-T0 BE_Bound.Mem_Bound.L1_Bound: 4.39 % Stalls
S0-C0-T0 BE_Bound.Mem_Bound.L2_Bound: 2.42 % Stalls
S0-C0-T0 BE_Bound.Mem_Bound.L3_Bound: 5.75 % Stalls
S0-C0-T0 BE_Bound.Mem_Bound.DRAM_Bound: 47.11 % Stalls <=
S0-C0-T0 BE_Bound.Mem_Bound.Store_Bound: 0.69 % Stalls
```

<sup>85</sup> PMU 工具 - <https://github.com/andikleen/pmu-tools>.

<sup>86</sup> 由于我们知道应用程序受内存限制，因此我们可以改用 -12 --nodes L1\_Bound,L2\_Bound,L3\_Bound,DRAM\_Bound,Store\_Bound 选项而不是 -13 来限制收集。

```
S0-C0-T0 BE_Bound.Core_Bound.Divider: 8.56 % Clocks
S0-C0-T0 BE_Bound.Core_Bound.Ports_Util: 11.31 % Clocks
```

我们发现瓶颈在于 DRAM\_Bound。这告诉我们，许多内存访问都会错过所有级别的缓存，并一直到达主内存。如果我们收集程序的 L3 缓存未命中绝对数量，也可以确认这一点。对于 Skylake 架构，DRAM\_Bound 指标是使用 CYCLE\_ACTIVITY.STALLS\_L3\_MISS 性能事件计算的。让我们手动收集它：

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

CYCLE\_ACTIVITY.STALLS\_L3\_MISS 事件会计算执行停顿时的周期数，而 L3 缓存未命中需求加载尚未完成。我们可以看到大约有 60% 的此类周期，这非常糟糕。

## 步骤 2：定位代码中的位置

TMA 过程的第二个步骤是找到性能事件最频繁发生的代码位置。为此，应该使用与步骤 1 中确定的瓶颈类型相对应的事件对工作负载进行采样。

查找此类事件的推荐方法是使用 toplev 工具的 --show-sample 选项，该选项将建议可用于定位问题的 perf record 命令行。为了理解 TMA 的机制，我们还介绍了手动查找与特定性能瓶颈关联的事件的方法。性能瓶颈和用于确定瓶颈在源代码中的位置的性能事件之间的对应关系可以使用 TMA metrics: [https://github.com/intel/perfmon/blob/main/TMA\\_Metrics.xlsx](https://github.com/intel/perfmon/blob/main/TMA_Metrics.xlsx)<sup>87</sup> 表格来完成。Locate-with 列表示用于定位问题发生确切代码位置的性能事件。在我们的例子中，为了找到导致 DRAM\_Bound 指标如此高的内存访问（L3 缓存未命中），我们应该对 MEM\_LOAD\_RETIRE.L3\_MISS\_PS 精确事件进行采样。以下是示例命令：

```
$ perf record -e cpu/event=0xd1,umask=0x20,name=MEM_LOAD_RETIRE.L3_MISS/ps ./benchmark.exe
$ perf report -n --stdio
...
# Samples: 33K of event 'MEM_LOAD_RETIRE.L3_MISS'
# Event count (approx.): 71363893
# Overhead  Samples  Shared Object  Symbol
# .....  .....
#
99.95%    33811  benchmark.exe  [.] foo
  0.03%      52  [kernel]        [k] get_page_from_freelist
  0.01%       3  [kernel]        [k] free_pages_prepare
  0.00%       1  [kernel]        [k] free_pcpages_bulk
```

几乎所有 L3 未命中都是由可执行文件 benchmark.exe 中的函数 foo 中的内存访问引起的。现在是时候查看基准测试的源代码了，可以在 Github: [https://github.com/dendibakh/dendibakh.github.io/tree/master/\\_posts/code/TMAM](https://github.com/dendibakh/dendibakh.github.io/tree/master/_posts/code/TMAM) 上找到。<sup>88</sup>

为了避免编译器优化，函数 foo 是用汇编语言实现的，如 Listing 89 所示。基准测试的“驱动”部分在 main 函数中实现，如 Listing 1 所示。我们分配了一个足够大的数组 a 以使其不适合 6MB 的 L3 缓存。基准测试生成一个指向数组 a 的随机索引，并将此索引与数组 a 的地址一起传递给 foo 函数。稍后，foo 函数会读取此随机内存位置。<sup>89</sup>

<sup>87</sup> TMA 指标 - [https://github.com/intel/perfmon/blob/main/TMA\\_Metrics.xlsx](https://github.com/intel/perfmon/blob/main/TMA_Metrics.xlsx).

<sup>88</sup> 案例研究示例 - [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>89</sup> 根据 x86 调用约定 ([https://en.wikipedia.org/wiki/X86\\_calling\\_conventions](https://en.wikipedia.org/wiki/X86_calling_conventions))，前两个参数分别位于 rdi 和 rsi 寄存器中。

清单：函数 foo 的汇编代码。

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

Listing 1 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);
    }
    ...
}
```

通过查看 Listing 89，我们可以看到函数 foo 中的所有 L3 缓存未命中都被标记为单个指令。现在我们知道是哪条指令导致了这么多 L3 未命中，让我们来修复它。

### 步骤 3：修复问题

请记住，在 foo 函数的开头有用 NOP 模拟的虚拟工作。这会在我们获得将要访问的下一个地址的那一刻与实际加载指令之间创建一个时间窗口。这个时间窗口的存在使我们有机会在虚拟工作的同时开始预取内存位置。Listing 89 展示了这个想法。有关显式内存预取技术的更多信息，请参阅 Section ??。

清单：在 main 中插入内存预取。

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

通过这个显式的内存预取提示，执行时间从 8.5 秒减少到 6.5 秒。此外，CYCLE\_ACTIVITY.STALLS\_L3\_MISS 事件的数量几乎减少了十倍：从 19B 减少到 2B。

TMA 是一个迭代过程，因此一旦我们修复了一个问题，我们就需要从步骤 1 开始重复该过程。它可能会将瓶颈移到另一个桶中，在本例中是“退休”。这是一个演示 TMA 方法工作流程的简单示例。分析现实世界的应用程序不太可能那么容易。本书第二部分的章节组织得井井有条，以便与 TMA 流程一起使用。特别是，第 8 章涵盖“内存受限”类别，第 9 章涵盖“核心受限”，第 10 章涵盖“错误猜测”，第 11 章涵盖“前端受限”。这种结构的目的是形成一个清单，供您在遇到特定性能瓶颈时用于驱动代码更改。

## 其他资源和链接

- Ahmad Yasin 的论文“用于性能分析和计数器架构的顶向下方法”[Yasin, 2014]。
- Ahmad Yasin 在 IDF'15 上的演讲“使用英特尔 Skylake 的顶向下分析使软件优化变得简单”，网址：[https://youtu.be/kjufVhyuV\\_A](https://youtu.be/kjufVhyuV_A)。
- Andi Kleen 的博客：pmu-tools，第二部分：toplev，网址：<http://halobates.de/blog/p/262>。
- Toplev 手册，网址：<https://github.com/andikleen/pmu-tools/wiki/toplev-manual>。

### 5.10.2 TMA 在 AMD 平台上

从 Zen4 开始，AMD 处理器支持一级和二级 TMA 分析。根据 AMD 文档，它被称为“管道利用率”分析，但基本思想保持不变。L1 和 L2 桶也与 Intel 的非常相似。Linux 用户可以使用 perf 工具收集管道利用率数据。

接下来，我们将研究 Crypto++: <https://github.com/weidai11/cryptopp><sup>90</sup> 实现的 SHA-256 (安全散列算法 256)，它是比特币挖掘的基本密码算法。Crypto++ 是一个开源的 C++ 密码算法类库，包含许多算法的实现，不仅仅是 SHA-256。但是，对于我们的示例，我们通过注释掉 `bench1.cpp` 中 `BenchmarkUnkeyedAlgorithms` 函数中相应的行来禁用所有其他算法的基准测试。

我们在配备 Ubuntu 22.04、Linux 内核 6.5.0-15-generic 的 AMD Ryzen 9 7950X 机器上运行了测试。我们使用 GCC 12.3 C++ 编译器编译了 Crypto++ 版本 8.9。我们使用了默认的 -O3 优化选项，但由于代码是用 x86 内在函数编写的（请参阅 Section 8.5）并利用了 SHA x86 ISA 扩展，因此对性能影响不大。

下面是我们用来获取 L1 和 L2 管道利用率指标的命令。输出经过修剪，删除了一些统计数据以消除不必要的干扰。

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

在输出中，括号中的数字表示运行时持续时间的百分比，当时正在监控指标。正如我们看到的，由于多路复用，所有指标只被监控了 20% 的时间。在我们的案例中，这可能不是问题，因为 SHA256 具有一致的行为，但并非总是如此。为了最小化多路复用的影响，您可以在单个运行中收集一组有限的指标，例如 `perf stat -M frontend_bound,backend_bound`。

<sup>90</sup> Crypto++ - <https://github.com/weidai11/cryptopp>

上面显示的管道利用率指标的描述可以在 [AMD, 2024, 第 2.8 章 管道利用率] 中找到。通过查看这些指标，我们可以看到分支预测不会在 SHA256 中发生 (bad\_speculation 为 0%)。仅使用了可用调度槽位的 26.3% (retiring)，这意味着其余 73.7% 由于前端和后端停顿而浪费。

加密指令并非简单，因此在内部被分解成更小的片段 ( $\mu$ ops)。一旦处理器遇到这样的指令，它就会从微码中检索它的  $\mu$ ops。微操作是从微码排序器获取的，带宽低于常规指令解码器，使其成为性能瓶颈的潜在来源。Crypto++ SHA256 实现大量使用诸如 SHA256MSG2, SHA256RNDS2 等指令，这些指令根据 uops.info: <https://uops.info/table.html><sup>91</sup> 网站由多个  $\mu$ ops 组成。retiring\_microcode 指标表明 6.1% 的调度槽位被微码操作使用。由于前端的带宽瓶颈，相同数量的调度槽位未使用 (frontend\_bound\_bandwidth)。这两个指标共同表明，这 6.1% 的调度槽位被浪费，因为微码排序器没有提供  $\mu$ ops，而后端本可以消耗它们。

大多数周期都停滞在 CPU 后端 (backend\_bound)，但只有 1.7% 的周期由于等待内存访问而停滞 (backend\_bound\_memory)。因此，我们知道基准测试主要受机器的计算能力限制。正如您将在本书第二部分中了解到的，这可能与数据流依赖性或某些加密操作的执行吞吐量有关。它们比传统的 ADD, SUB, CMP 等指令不那么频繁，因此通常只能在单个执行单元上执行。大量这样的操作可能会使该特定单元的执行吞吐量饱和。进一步的分析应该更仔细地观察源代码和生成的汇编代码，检查执行端口利用率，查找数据依赖性等；我们将在此停止。

就 Windows 而言，在撰写本文时，TMA 方法仅在服务器平台（代号 Genoa）上受支持，而不支持客户端系统（代号 Raphael）。TMA 支持在 AMD uProf 版本 4.1 中添加，但仅在命令行工具 AMDuProfPcm 工具中，它是 AMD uProf 安装的一部分。您可以参考 [AMD, 2024, 第 2.8 章 管道利用率] 了解更多有关如何运行分析的详细信息。AMD uProf 的图形版本还没有 TMA 分析。

### 5.10.3 TMA 在 ARM 平台上

ARM CPU 架构师也为他们的处理器开发了一种 TMA 性能分析方法，我们接下来将讨论。ARM 在其文档中将其称为“Topdown”[Arm, 2023a]，因此我们将使用他们的命名。在撰写本章节时（2023 年底），Topdown 仅支持 ARM 设计的内核，例如 Neoverse N1 和 Neoverse V1 及其衍生产品，例如 Ampere Altra 和 AWS Graviton3。如果您需要刷新有关 ARM 芯片系列的记忆，请参考本书末尾的主要 CPU 微架构列表。Apple 设计的处理器目前还不支持 ARM Topdown 性能分析方法。

Neoverse V1 是 Neoverse 系列中第一个支持全套 1 级 Topdown 指标的 CPU：Bad Speculation、Frontend Bound、Backend Bound 和 Retiring。据说未来的 Neoverse 内核将支持更高级别的 TMA。在撰写本文时，没有针对 Neoverse N2 和 V2 内核的分析指南。在 V1 内核之前，Neoverse N1 只支持两个 L1 类别：Frontend Stalled Cycles 和 Backend Stalled Cycles。

为了演示基于 V1 处理器的 ARM Topdown 分析，我们启动了一个由 AWS Graviton3 驱动的 AWS EC2 m7g.meta 实例。请注意，由于虚拟化，Topdown 可能无法在其他非金属实例类型上运行。我们请求了由 AWS 管理的 64 位 ARM Ubuntu 22.04 LTS 以及 Linux kernel 6.2。提供的 m7g.meta 实例有 64 个 vCPU 和 256 GB 的内存。

我们将 Topdown 方法应用于 AI Benchmark Alpha: <https://ai-benchmark.com/alpha.html><sup>92</sup>，它是一个用于评估各种硬件平台（包括 CPU、GPU 和 TPU）的 AI 性能的开源 Python 库。该基准测试依赖 TensorFlow 机器学习库来测量关键深度学习模型的推理和训练速度。AI Benchmark Alpha 总共包含 42 个测试，包括分类、图像分割、文本翻译等等。

ARM 工程师开发了 topdown-tool: <https://learn.arm.com/install-guides/topdown-tool/><sup>93</sup>，我们将在下面使用它。该工具可以在 Linux 和 Windows 上的 ARM 上运行。在 Linux 上，它使用标准的 perf 工具，而在 Windows 上，它使用 WindowsPerf: <https://gitlab.com/Linaro/WindowsPerf/windowsperf><sup>94</sup>，这是一款 Windows on Arm 性能分析工具。类

<sup>91</sup> uops.info - <https://uops.info/table.html>

<sup>92</sup> AI Benchmark Alpha - <https://ai-benchmark.com/alpha.html>

<sup>93</sup> ARM topdown-tool - <https://learn.arm.com/install-guides/topdown-tool/>

<sup>94</sup> WindowsPerf - <https://gitlab.com/Linaro/WindowsPerf/windowsperf>

似于 Intel 的 TMA，ARM 的方法也采用了“向下钻取”的概念，即首先确定高级性能瓶颈，然后向下钻取更细致的根本原因分析。以下是我们使用的命令：

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

其中 `--all-cpus` 选项启用所有 CPU 的系统级收集，而 `-m Topdown_L1` 则收集 Topdown 1 级指标。`--` 后面的所有内容都是运行 AI Benchmark Alpha 套件的命令行。

从上面的输出中，我们可以得出结论，基准测试不会出现分支预测错误。此外，如果不深入了解所涉及的工作负载，就很难说 16.5% 的“前端绑定”是否值得关注，因此我们将注意力转移到“后端绑定”指标上，该指标显然是停滞周期的主要来源。基于 Neoverse V1 的芯片没有二级细分，相反，该方法建议通过收集一组相应的指标来进一步探索有问题的类别。以下是我们如何深入研究更详细的“后端绑定”分析：

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

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

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

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

[Last Level Cache Effectiveness]
LL Cache Read MPKI..... 2.505 misses per 1,000 instructions
```

```

LL Cache Read Miss Ratio..... 0.219 per cache access
LL Cache Read Hit Ratio..... 0.783 per cache access

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

```

Misc 类别包含不在主类别中的指令。例如，barriers

在上面的命令中，选项 `-n BackendBound` 收集与 Backend Bound 类别及其后代相关的所有指标。输出中每个指标的描述在 [Arm, 2023a] 中给出。请注意，它们与我们在 Section 4.11 中讨论的非常相似，因此您可能也想重新查看它。

我们的目标不是优化基准测试，而是要描述性能瓶颈。但是，如果有这样的任务，我们的分析可以继续进行。有大量的 L1 Data TLB 未命中 (3.8 MPKI)，但随后 90% 的未命中命中 L2 TLB (参见 L2 Unified TLB Miss Ratio)。总而言之，只有 0.1% 的 TLB 未命中导致页表遍历 (参见 DTLB Walk Ratio)，这表明这不是我们的主要关注点，尽管快速使用大页面的实验仍然值得。

查看 L1/L2/LL Cache Effectiveness 指标，我们可以发现数据缓存未命中的潜在问题。对 L1D 缓存的 ~22 次访问中就有一次会导致未命中 (参见 L1D Cache Miss Ratio)，这是可以容忍但仍然很昂贵的。对于 L2，这个数字是 37 分之一 (参见 L2 Cache Miss Ratio)，这要好得多。然而对于 LLC，LL Cache Read Miss Ratio 是不令人满意的：每 4 次访问就会导致一次失败。由于这是一个 AI 基准测试，其中大部分时间可能花在矩阵乘法上，因此循环阻塞等代码转换可能会有所帮助 (参见 Section ??)。

最后一类给出了操作组合，这在某些情况下很有用。在我们的例子中，我们应该关注 SIMD 操作的低百分比，特别是考虑到使用了高度优化的 Tensorflow 和 numpy 库。相比之下，整数运算和分支的百分比似乎太高了。分支可能来自 Python 解释器或过多的函数调用。而高百分比的整数操作可能是由于缺乏矢量化或线程同步造成的。[Arm, 2023a] 给出了一个使用来自 Speculative Operation Mix 类别的数据发现矢量化机会的示例。

在我们的案例研究中，我们运行了两次基准测试，但在实践中，一次运行通常就足够了。运行没有选项的 `topdown-tool` 将使用单次运行收集所有可用的指标。此外，`-s combined` 选项将按 L1 类别对指标进行分组，并以类似于 Intel Vtune、toplev 和其他工具的格式输出数据。进行多次运行的唯一实际原因是工作负载具有突发行，其非常短的阶段具有不同的性能特征。在这种情况下，您希望避免事件多路复用 (参见 Section 5.3.3) 并通过多次运行工作负载来提高收集准确性。

AI Benchmark Alpha 有各种可能表现出不同性能特征的测试。上面显示的输出汇总了所有基准并给出了总体细分。如果单个测试确实存在不同的性能瓶颈，这通常不是一个好主意。您需要对每个测试进行单独的 Topdown 分析。`topdown-tool` 可以提供帮助的一种方法是使用 `-i` 选项，该选项将根据可配置的时间间隔输出数据。然后，您可以比较间隔并决定下一步。

#### 5.10.4 TMA 总结

TMA 非常适合识别 CPU 性能瓶颈。理想情况下，当我们在应用程序上运行它时，我们希望看到“Retiring”指标达到 100%。尽管存在例外。“Retiring”指标达到 100% 意味着 CPU 已满负荷工作，并且以全速处理指令。但这并不能说明这些指令的质量。程序可以在紧密循环中等待锁，这将显示高“Retiring”指标，但不会做任何有用的工作。

另一个您可能看到高“Retiring”值但整体性能较慢的例子是程序存在未矢量化的热点。您通过让处理器运行简单非矢量化操作来让它“轻松”，但这真的是利用可用 CPU 资源的最佳方式吗？当然不是。如果 CPU 没有执行代码的问题，并不意味着性能无法提高。注意这种情况，并记住 TMA 识别 CPU 性能瓶颈，但不会将其与程序性能相关联。您一旦进行必要的实验就会发现这一点。

虽然在玩具程序上实现“Retiring”接近 100% 是可能的，但现实世界的应用程序远不能达到。图 40 展示了 Google 数据中心工作负载的顶级 TMA 指标以及在 Intel 的 IvyBridge 服务器处理器上运行的几个 SPEC CPU2006: <http://spec.org/cpu2006/><sup>95</sup> 基准测试。我们可以看到，大多数数据中心工作负载在“Retiring”桶中所占的比例非常小。这意味着大多数数据中心工作负载都会花时间停滞在各种瓶颈上。“BackendBound”是性能问题的主要来源。“FrontendBound”类别对于数据中心工作负载来说比 SPEC2006 更重要，因为这些应用程序通常具有庞大的代码库。最后，一些工作负载比其他工作负载更易受分支预测错误的影响，例如“search2”和“445.gobmk”。

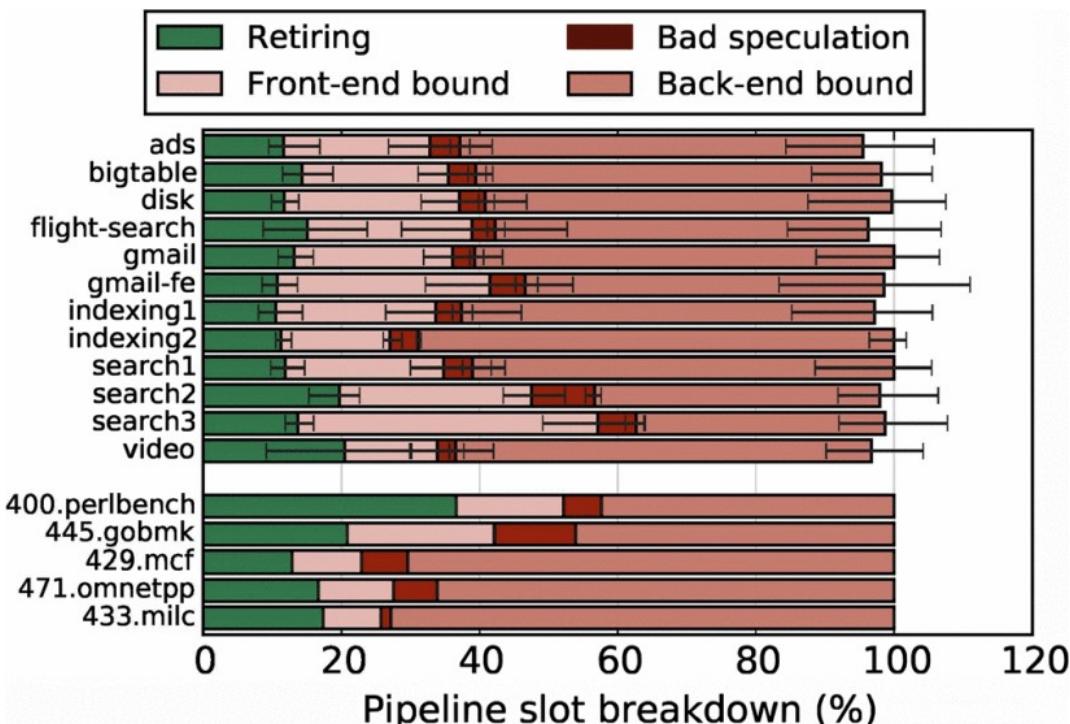


Figure 40: 谷歌数据中心工作负载的 TMA 分解以及几个 SPEC CPU2006 基准, © Image from [Kanev et al., 2015]

请记住，随着架构师不断尝试改进 CPU 设计，这些数字可能会随着其 CPU 而改变。这些数字也可能会随着其他指令集架构 (ISA) 和编译器版本的改变而改变。

在我们继续讨论之前，还有一些最后的思考……不建议在性能存在重大缺陷的代码上使用 TMA，因为它可能会将您引入错误的方向，并且您将修复真正的性能问题，而不是调整糟糕的代码，这只会浪费时间。同样，确保环境不会妨碍分析。例如，如果您丢弃文件系统缓存并在 TMA 下运行基准测试，它可能会显示您的应用程序受到内存限制，而实际上，当文件系统缓存预热时，这可能是错误的。

TMA 提供的工作负载特征描述可以将潜在优化的范围扩展到源代码之外。例如，如果应用程序受内存带宽限制，并且已经用尽所有可能的软件层面加速方法，那么可以通过升级内存子系统以使用更快的内存芯片来提高性能。这说明了如何使用 TMA 诊断性能瓶颈来支持您决定在新硬件上花钱。

<sup>95</sup> SPEC CPU 2006 - <http://spec.org/cpu2006/>.

## 5.11 分支记录机制 (Branch Recording Mechanisms)

现代高性能 CPU 提供分支记录机制，使处理器能够连续记录一组先前执行的分支。但在进入细节之前，你可能会问：为什么我们对分支如此感兴趣？嗯，因为这是我们如何确定程序控制流的方式。我们基本上忽略基本块（参见 Section 10.1）中的其他指令，因为分支总是基本块中的最后一个指令。由于基本块中的所有指令都保证执行一次，因此我们只能关注将“代表”整个基本块的分支。因此，如果我们跟踪每个分支的结果，就可以重建程序的整个逐行执行路径。事实上，这就是英特尔处理器跟踪 (PT) 功能可以做到的，它在附录 D 中讨论。我们将在这里讨论的分支记录机制基于采样而不是跟踪，因此具有不同的用例和功能。

由英特尔、AMD 和 ARM 设计的处理器都宣布了他们的分支记录扩展。确切的实现可能会有所不同，但基本思想是相同的。硬件并行记录每个分支的“来自”和“到”地址以及一些额外数据，同时执行程序。如果我们收集足够长的源目的地对历史记录，我们将能够像调用堆栈一样解开程序的控制流，但深度有限。此类扩展旨在使正在运行的程序的运行速度降低到最小，通常在 1% 以内。

如果使用分支记录机制，我们可以在分支（或周期，没关系）上进行采样，但在每个采样期间，查看先前执行的 N 个分支。这使我们在热门代码路径中合理地覆盖了控制流，但不会让我们因为只检查了总数较少的分支而获得过多信息。请务必记住，这仍然是采样，因此并不是每个执行的分支都可以被检查。CPU 通常执行得太快，无法做到这一点。

非常重要的一点是，只记录已采取的分支。Listing 5.11 显示了如何跟踪分支结果的示例。此代码表示一个循环，其中三个指令可能会改变程序的执行路径，即循环后缘 JNE (1)、条件分支 JNS (2)、函数 CALL (3) 和从此函数返回 (4，未显示)。

代码清单：记录分支的示例。

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

以下是使用分支记录机制可以记录的可能分支历史之一。它显示了执行 CALL 指令时最近的 7 个分支结果（未显示更多）。由于在循环的最新迭代中没有执行 JNS 分支 (4eda14 -> 4eda1e)，因此它没有被记录，因此不会出现在历史记录中。

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

## 5.12 未记录未执行的分支

未记录未执行的分支可能会增加分析负担，但通常不会使其过于复杂。由于我们知道控制流从条目 N-1 的目标地址到条目 N 的源地址是顺序的，因此我们仍然可以推断完整的执行路径。

接下来，我们将分别看一下每个供应商的分支记录机制，然后探讨如何在性能分析中使用它们。

### 5.12.1 英特尔平台上的 LBR

英特尔首次在其 Netburst 微架构中实现了其最后分支记录 (LBR) 功能。最初，它只能记录最近的 4 个分支结果。从 Nehalem 开始增加到 16，从 Skylake 开始增加到 32。在 Goldencove 微架构之前，LBR 被作为一组特定于模型的寄存器 (MSR) 实现，但现在它在架构寄存器内工作。其主要优点是 LBR 功能清晰可见，无需检查当前 CPU 的确切型号。这使操作系统和分析工具中的支持更加容易。此外，LBR 条目可以配置为包含在 PEBS 记录中（参见 Section ??）。

LBR 寄存器就像一个不断被覆盖的环形缓冲区，仅提供最近的 32 个分支结果。每个 LBR 条目由三个 64 位值组成：

- 分支的源地址（“来自 IP”）。
- 分支的目标地址（“到 IP”）。
- 操作的元数据，包括错误预测和经过周期时间信息。

除了源地址和目标地址之外，保存的其他信息还有一些重要的应用，我们将在稍后讨论。

当采样计数器溢出并触发性能监控中断 (PMI) 时，LBR 记录冻结，直到软件捕获 LBR 记录并恢复收集。

LBR 收集可以限制在一组特定的分支类型上，例如用户可以选择只记录函数调用和返回。将此类过滤器应用于 Listing 5.11 中的代码，我们只会看到历史记录中的分支 (3) 和 (4)。用户还可以过滤进/出条件跳转和无条件跳转、间接跳转和调用、系统调用、中断等。在 Linux perf 中，有一个 -j 选项可以启用/禁用记录各种分支类型。

默认情况下，LBR 数组作为一个环形缓冲区，捕获控制流转换。然而，LBR 数组的深度是有限的，这在分析某些应用程序时可能是一个限制因素，其中执行流的转换伴随着大量叶函数调用。这些对叶函数的调用及其返回很可能将主执行上下文从 LBR 中移除。再次考虑 Listing 5.11 中的示例。假设我们想要从 LBR 中的历史记录中解剖堆栈，因此我们将 LBR 配置为仅捕获函数调用和返回。如果循环运行数千次迭代，并且考虑到 LBR 数组只有 32 个条目，那么我们只看到 16 对条目 (3) 和 (4) 的可能性非常高。在这种情况下，LBR 数组充满了叶函数调用，这些调用对我们解开当前调用堆栈没有帮助。

这就是为什么 LBR 支持调用堆栈模式。启用此模式后，LBR 数组仍像以前一样捕获函数调用，但随着返回指令的执行，最后捕获的分支 (call) 记录将以后进先出 (LIFO) 方式从数组中刷新。因此，与已完成叶函数相关的分支信息将不会保留，同时保留主执行路径的调用堆栈信息。使用这种配置，LBR 数组模拟一个调用堆栈，其中 CALL 会将条目“压入”堆栈，而 RET 则会将条目“弹出”堆栈。如果你的应用程序中调用堆栈的深度永远不会超过 32 个嵌套框架，LBR 将为你提供非常准确的信息。[\[Intel, 2023b, 第 3B 卷, 第 19 章 最后分支记录\]](#)

可以使用以下命令确保你的系统上的 LBR 已启用：

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

使用 Linux 的 perf，可以使用以下命令收集 LBR 堆栈：

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

LBR 堆栈也可以使用 `perf record --call-graph lbr` 命令收集，但是收集的信息量少于使用 `perf record -b`。例如，在运行 `perf record --call-graph lbr` 时不会收集分支预测和周期数据。

因为每个收集的样本都捕获整个 LBR 堆栈（32 个最后的分支记录），所以收集的数据（`perf.data`）的大小比不使用 LBR 的采样要大得多。尽管如此，在大多数 LBR 使用案例中，运行时开销低于 1%。[Nowak & Bitzes, 2014]

用户可以导出原始 LBR 堆栈进行自定义分析。以下是可以用来转储收集的分支堆栈内容的 Linux perf 命令：

```
$ perf record -b -e cycles ./benchmark.exe
$ perf script -F brstack >> dump.txt
```

`dump.txt` 文件可能非常大，包含如下所示的行：

```
...
0x4edadf9/0x4edab0/P/-/-/29
0x4edabd/0x4edad0/P/-/-/2
0x4edaddd/0x4edb00/M/-/-/4
0x4edb24/0x4edab0/P/-/-/24
0x4edabd/0x4edad0/P/-/-/2
0x4edaddd/0x4edb00/M/-/-/1
0x4edb24/0x4edab0/P/-/-/3
0x4edabd/0x4edad0/P/-/-/1
...
...
```

上述输出展示了 LBR 堆栈中的 8 个条目，LBR 堆栈通常包含 32 个条目。每个条目都有 FROM 和 TO 地址（十六进制值）、预测标志（M - 预测错误，P - 预测正确）以及周期数（每个条目最后一个位置的数字）。用“-”标记的组件与事务内存扩展 (TSX) 相关，我们将在本文中不进行讨论。好奇的读者可以参考 `perf script` 规范：<http://man7.org/linux/man-pages/man1/perf-script.1.html><sup>96</sup> 中解码的 LBR 条目的格式。

### 5.12.2 AMD 平台上的 LBR

AMD 处理器也支持 AMD Zen4 处理器上的最后分支记录 (LBR)。Zen4 拥有 16 对“from”和“to”地址日志以及一些额外的元数据。类似于 Intel LBR，AMD 处理器能够记录各种类型的分支。与 Intel LBR 的主要区别在于 AMD 处理器目前还不支持调用堆栈模式，因此 LBR 功能无法用于调用堆栈收集。另一个明显的区别是 AMD LBR 记录中没有周期计数字段。有关更多详细信息，请参见 [AMD, 2023, 13.1.1.9 最后分支堆栈寄存器]。

从 Linux 内核 6.1 开始，Linux “perf”在 AMD Zen4 处理器上支持我们将在下面讨论的分支分析用例，除非另有明确说明。Linux perf 命令收集 AMD LBR 使用相同的 `-b` 和 `-j` 选项。

使用 AMD uProf CLI 工具也可以进行分支分析。以下示例命令将转储收集的原始 LBR 记录并生成 CSV 报告：

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

## 5.13 ARM 平台上的 BRBE

ARM 在 2020 年作为 ARMv9.2-A ISA 的一部分推出了其名为 BRBE 的分支记录扩展。ARM BRBE 与英特尔的 LBR 非常相似，提供了许多类似的功能。就像英特尔的 LBR 一样，BRBE 记录也包含源地址和目标地址、预测错误位和周期计数值。根据最新可用的 BRBE 规范，不支持调用堆栈模式。分支记录仅包含已在架构上执行的分支的信息，即不在预测错误路径上。用户还可以根据特定分支类型过滤记录。一个值得注意的区别是 BRBE 支持可配置的 BRBE

<sup>96</sup> Linux perf script 手册页 - <http://man7.org/linux/man-pages/man1/perf-script.1.html>.

缓冲区深度：处理器可以选择 BRBE 缓冲区的容量为 8、16、32 或 64 个记录。更多细节可在 [Arm, 2022a, 章节 F1 “Branch Record Buffer Extension”] 中找到。

在撰写本文时，还没有商用机器实现 ARMv9.2-A，因此无法测试此扩展的实际运行情况。

### 5.13.1 捕获调用堆栈

分支记录使许多重要用例成为可能。在本节和接下来的几节中，我们将介绍最重要的几个用例。

分支记录最流行的用例之一是捕获调用堆栈。我们已经在 Section 5.5.3 中介绍了为什么需要收集它们。即使你编译了一个没有帧指针或调试信息的程序，分支记录也可以用作收集调用图信息的轻量级替代方法。

在撰写本文时（2023 年），AMD 的 LBR 和 ARM 的 BRBE 不支持调用堆栈收集，但英特尔的 LBR 支持。以下是你可以使用英特尔 LBR 执行此操作的方法：

```
$ perf record --call-graph lbr -- ./a.exe
$ perf report -n --stdio
# Children  Self  Samples  Command  Object  Symbol
# .....  .....  .....  .....  .....  .....
99.96%  99.94%    65447     a.exe    a.exe  [.] bar
|
--99.94%--main
|
|--90.86%--foo
|   |
|   --90.86%--bar
|
--9.08%--zoo
      bar
```

正如你所见，我们已经确定了程序中最热的功能（即 `bar`）。我们还发现调用者对函数 `bar` 中花费的大部分时间做出了贡献：该工具捕获了 `main->foo->bar` 调用堆栈 91% 的时间，捕获了 `main->zoo->bar` 9% 的时间。换句话说，`bar` 中 91% 的样本都将 `foo` 作为其调用者函数。

值得一提的是，在这种情况下，我们不一定能得出关于函数调用次数的结论。例如，我们不能说 `foo` 调用 `bar` 的频率比 `zoo` 高 10 倍。可能的情况是，`foo` 调用 `bar` 一次，但在 `bar` 内部执行了昂贵的路径，而 `zoo` 调用 `bar` 多次，但很快就返回。

### 5.13.2 识别热点分支 (#sec:lbr\_hot\_branch)

分支记录还使我们能够知道哪些分支被采取的频率最高。它在 Intel 和 AMD 上都支持。根据 ARM 的 BRBE 规范，它可以支持，但由于缺乏实现此扩展的处理器，无法验证。这里是一个例子：

**TODO:** 检查：“添加 `-F +srcline_from,srcline_to` 会降低构建报告的速度。希望在更高版本的 perf 中，解码时间会得到改善”。

```
$ 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)  |  |

从这个例子中，我们可以看到超过 50% 的已采取分支位于 `bar` 函数内，22% 的分支是来自 `foo` 到 `bar` 的函数调用，等等。请注意，perf 如何从 `cycles` 事件切换到分析 LBR 堆栈：只收集了 670 个样本，但每个样本都捕获了整个 LBR 堆栈。这为我们提供了 21440 个 LBR 条目（分支结果）进行分析。<sup>97</sup>

大多数情况下，仅从代码行和目标符号就可以确定分支的位置。然而，理论上，可以编写代码，在单行上写两个 `if` 语句。此外，展开宏定义时，所有展开的代码都归于相同的源行，这也是可能发生此类情况的另一个场景。这个问题不会完全阻止分析，只会使其稍微困难一些。为了消除两个分支的歧义，您可能需要自己分析原始 LBR 堆栈（请参阅 easyperf 博客上的示例：<https://easyperf.net/blog/2019/05/06/Estimating-branch-probability><sup>98</sup>）。

使用分支记录，我们还可以找到一个“超块”（有时称为“超级块”），它是函数中一系列热门基本块的链，这些基本块不一定按照顺序排列，但它们是顺序执行的。因此，超块代表了函数、代码片段或程序的典型热路径。

### 5.13.3 分析分支预测错误率 (#sec:secLBR\_misp\_rate)

由于每个记录中保存的附加信息中包含预测错误位，因此还可以知道热门分支的预测错误率。在这个例子中，我们使用了 LLVM 测试套件中 7-zip 基准的纯 C 代码版本。<sup>99</sup> perf report 的输出经过稍微修剪，以便更好地适应页面。以下用例在 Intel 和 AMD 上都支持。根据 ARM 的 BRBE 规范，它可以支持，但由于缺乏实现此扩展的处理器，无法验证。

| # 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    |  |

在这个例子中，与函数 `LzmaDec` 相对应的行是我们特别关注的。按照上一节类似的分析，我们可以得出结论，`dec.c:36` 源行上的分支是基准测试中执行次数最多的分支。在 Linux perf 提供的输出中，我们可以看到两个与 `LzmaDec` 函数相对应的条目：一个带有 Y 字母，另一个带有 N 字母。将这两个条目一起分析，我们可以得到该分支的预测错误率。在这种情况下，我们知道 `dec.c:36` 行上的分支被正确预测了 303391 次（对应于 N），被错误预测了 41665 次（对应于 Y），因此预测率为 88%。

Linux perf 通过分析每个 LBR 条目并从中提取预测错误位来计算预测错误率。因此，对于每个分支，我们都知道它被正确预测的次数和错误预测的次数。同样，由于采样的性质，一些分支可能有一个 N 条目，但没有对应的 Y 条目。

<sup>97</sup> perf 生成的报告头可能仍然让人困惑，因为它说“21K of event cycles”。但这里有“21K”个 LBR 条目，而不是“cycles”。

<sup>98</sup> Easyperf: 估计分支概率 - <https://easyperf.net/blog/2019/05/06/Estimating-branch-probability>

<sup>99</sup> LLVM 测试套件 7zip 基准 - <https://github.com/llvm-mirror/test-suite/tree/master/MultiSource/Benchmarks/7zip>

这可能意味着没有该分支被错误预测的 LBR 条目，但这并不意味着预测率是 100%。

#### 5.13.4 机器码的精确计时 (#sec:timed\_lbr)

正如我们在英特尔 LBR 部分所展示的，从 Skylake 微架构开始，LBR 条目中有一个特殊的 周期计数 字段。这个附加字段指定了两个已采取分支之间经过的周期数。由于前一个 (N-1) LBR 条目中的目标地址是一个基本块 (BB) 的开始，而当前 (N) LBR 条目中的源地址是同一个基本块的最后一个指令，因此周期计数就是这个基本块的延迟。

这种类型的分析在 AMD 平台上不受支持，因为它们不会在 LBR 记录中记录周期计数。根据 ARM 的 BRBE 规范，它可以支持，但由于缺乏实现此扩展的处理器，无法验证。但是，英特尔支持它。这里是一个例子：

```
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
```

假设我们在 LBR 堆栈中有两个条目：

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

根据这些信息，我们知道从偏移量 400618 开始执行的基本块以 5 个周期执行了一次。如果我们收集足够多的样本，我们可以绘制该基本块延迟的概率密度图。

图 41 展示了这样的图表示例。它是通过分析所有满足上述规则的 LBR 条目编译而成的。读取图表的方法如下：它告诉我们给定延迟值出现的比率。例如，大约 2% 的时间测量到基本块延迟正好为 100 周期，14% 的时间测量到 280 周期，我们从未见过 150 到 200 周期之间的数值。另一种解读方式是：根据收集的数据，如果您要测量某个基本块的延迟，看到特定延迟的概率是多少？

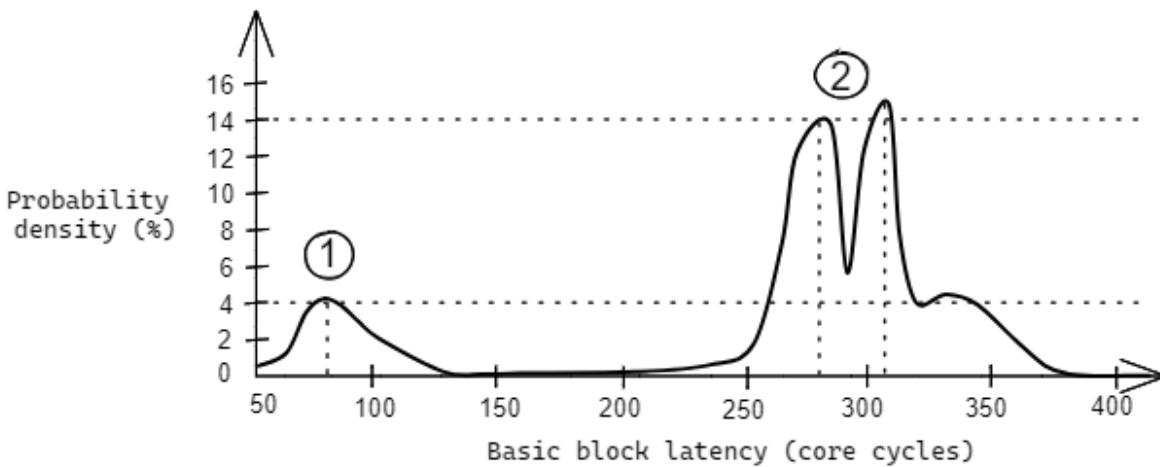


Figure 41: 基本块延迟的概率密度图，基本块起始地址为 0x400618

我们可以清楚地看到两个峰值：一个较小的峰值大约在 80 周期①，两个更大的峰值在 280 和 305 周期②。该块从一个不适合 CPU L3 缓存的大数组中进行非顺序加载，因此基本块的延迟很大程度上取决于此加载。基于图表，我们可以得出结论，第一个峰值①对应于 L3 缓存命中，第二个峰值②对应于 L3 缓存未命中，其中加载请求一直到主内存。

这些信息可以用于对该基本块进行细粒度调整。此示例可能受益于内存预取，我们将在 Section ?? 中讨论。此外，周期计数信息可用于循环迭代的计时，其中每个循环迭代都以一个已采取的分支（后向边缘）结束。

在适当的性能分析工具支持到位之前，构建类似于图 41 的概率密度图需要手动解析原始 LBR 转储。有关如何执行此操作的示例，请参见 easyperf 博客：<https://easyperf.net/blog/2019/04/03/Precise-timing-of-machine-code-with-Linux-perf<sup>100</sup>>。幸运的是，在较新的 Linux perf 版本中，获取这些信息要容易得多。以下示例直接使用 Linux perf 在我们之前介绍的 LLVM 测试套件中相同的 7-zip 基准测试上演示了这种方法：

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

请注意，我们添加了 `-F +cycles` 选项以在输出中显示周期计数（“BBCycles”列）。为适应页面大小，我们删除了几行无关紧要的 `perf report` 输出。让我们关注源代码和目标代码都是 `dec.c:174` 的行，输出中有七行这样的行。在源代码中，行 `dec.c:174` 展开了一个包含自包含分支的宏。这就是为什么源代码和目标代码恰好位于同一行的原因。

Linux perf 首先按开销对条目进行排序，因此我们需要手动过滤我们感兴趣的分支的条目。幸运的是，它们可以通过 `grep` 命令轻松过滤。事实上，如果我们过滤它们，我们将得到以这个分支结尾的基本块的延迟分布，如表 7 所示。这些数据可以绘制成一个类似于图 41 的图表。

Table 7: 基本块延迟的概率密度。

| 周期             | 样本数            | 概率密度        |
|----------------|----------------|-------------|
| 1 2 3 4 6 24   | 10484 16728    | 17.0% 27.1% |
| 32             | 4563 15815     | 7.4% 25.6%  |
| 4770 3804 5567 | 7.7% 6.2% 9.0% |             |

<sup>100</sup> Easyperf: 为任意基本块的延迟构建概率密度图 - <https://easyperf.net/blog/2019/04/03/Precise-timing-of-machine-code-with-Linux-perf>.

以下是我们如何解释这些数据：从所有收集的样本中，17% 的时间基本块的延迟为 1 个周期，27% 的时间为 2 个周期，等等。请注意，分布主要集中在 1 到 6 个周期，但也有第二个模式，延迟高得多，为 24 和 32 个周期，这可能对应于分支预测错误惩罚。分布中的第二个模式占所有样本的 15%。

这个例子表明，不仅可以绘制微型基准测试的基本块延迟，还可以绘制实际应用程序的基本块延迟。目前，LBR 是英特尔系统上最精确的周期级计时信息源。

### 5.13.5 估计分支结果概率

在后面的 Chapter 10 部分，我们将讨论代码布局对性能的重要性。进一步说，以透底方式放置热路径<sup>101</sup> 通常可以提高程序性能。知道某个分支最常见的执行结果可以让开发人员和编译器做出更好的优化决策。例如，如果一个分支有 99% 的时间被执行，我们可以尝试反转条件将其转换为未执行分支。

LBR 可以让我们在不检测代码的情况下收集这些数据。分析结果将为用户提供条件真假结果之间的比率，即分支被执行和未执行的次数。这一特性在分析间接跳转（switch 语句）和间接调用（虚函数调用）时尤其有用。您可以在 easyperf 博客上找到实际应用示例：[https://easyperf.net/blog/2019/05/06/Estimating-branch-probability<sup>102</sup>](https://easyperf.net/blog/2019/05/06/Estimating-branch-probability)。

### 5.13.6 提供编译器反馈数据

我们将在后面的 Section 10.5 部分讨论基于配置文件的优化 (PGO)，这里先简要提一下。分支记录机制可以为优化编译器提供配置文件反馈数据。想象一下，我们可以将我们在前面部分发现的所有数据反馈给编译器。在某些情况下，这些数据无法使用传统的静态代码检测工具获得，因此分支记录机制不仅因开销更低而成为更好的选择，而且还能提供更丰富的配置文件数据。依赖硬件 PMU 收集数据进行的 PGO 工作流程越来越流行，一旦 AMD 和 ARM 的支持成熟，可能会迅速发展。

## 5.14 基于硬件的采样功能

主要 CPU 供应商提供了一系列附加功能来增强性能分析。由于 CPU 供应商以不同的方式处理性能监控，因此这些功能不仅在调用方式上存在差异，而且在功能上也存在差异。在 Intel 处理器中，它被称为处理器事件采样 (PEBS)，首次引入于 NetBurst 微架构。AMD 处理器上类似的功能称为指令采样 (IBS)，从 AMD Opteron 家族 (10h 代) 核心开始可用。接下来，我们将更详细地讨论这些功能，包括它们的相似之处和不同之处。

### 5.14.1 英特尔平台上的 PEBS

与最后分支记录类似，PEBS 用于在分析程序时捕获每个收集到的样本的额外数据。当性能计数器配置为 PEBS 时，处理器会保存一组具有定义格式的额外数据，称为 PEBS 记录。英特尔 Skylake CPU 的 PEBS 记录格式如图 42 所示。记录包含通用寄存器状态 (EAX, EBX, ESP 等)、EventingIP, Data Linear Address 和稍后将讨论的 延迟值。PEBS 记录的内容布局因不同的微架构而异，请参阅 [Intel, 2023b, 第 3B 卷, 第 20 章 性能监控]。

从 Skylake 架构开始，PEBS 记录已经增强，可以收集 XMM 寄存器和 LBR 记录。格式已经重新组织，将字段分组为 基本组、内存组、GPR 组、XMM 组和 LBR 组。性能分析工具可以选择感兴趣的数据组，从而减小记录大小并降低记录生成延迟。默认情况下，PEBS 记录只包含 基本组。

使用 PEBS 的一个显著优点是与基于中断的常规采样相比，采样开销更低。回想一下，当计数器溢出时，CPU 会生成中断来收集一个样本。频繁地生成中断并让分析工具本身在中断服务例程中捕获程序状态是非常昂贵的，因为它涉及操作系统交互。

另一方面，PEBS 维护了一个缓冲区，用于临时存储多个 PEBS 记录。假设我们使用 PEBS 对加载事件进行采样。当性能计数器配置为 PEBS 时，计数器溢出条件不会触发中断，而是会激活 PEBS 机制。该机制将捕获下一个加载，捕

<sup>101</sup> 也就是说，当热分支没有被执行时。

<sup>102</sup> Easyperf: 估计分支概率 - <https://easyperf.net/blog/2019/05/06/Estimating-branch-probability>

| 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 42: PEBS 记录格式适用于第六代、第七代和第八代英特尔酷睿处理器家族。© Image from [Intel, 2023b, Volume 3B, Chapter 20].

获一个新的记录并将其存储在专用的 PEBS 缓冲区区域。该机制还会清除计数器溢出状态并重新加载计数器以初始值。只有当专用缓冲区已满时，处理器才会引发中断，缓冲区才会刷新到内存。这种机制通过减少中断触发次数来降低采样开销。

Linux 用户可以通过执行 `dmesg` 检查 PEBS 是否已启用：

```
$ dmesg | grep PEBS
[    0.113779] Performance Events: XSAVE Architectural LBR, PEBS fmt4+-baseline,
AnyThread deprecated, Alderlake Hybrid events, 32-deep LBR, full-width counters, Intel PMU driver.
```

对于 LBR，Linux perf 会在每个收集到的样本中转储整个 LBR 堆栈内容。因此，可以分析由 Linux perf 收集的原始 LBR 转储。但是，对于 PEBS，Linux perf 不会像 LBR 那样导出原始输出。相反，它会处理 PEBS 记录并仅提取根据特定需求的数据子集。因此，无法使用 Linux perf 访问原始 PEBS 记录的集合。但是，Linux perf 提供了一些从原始样本处理过的 PEBS 数据，可以通过 `perf report -D` 访问。要转储原始 PEBS 记录，可以使用 `pebs-grabber`: <https://github.com/andikleen/pmu-tools/tree/master/pebs-grabber><sup>103</sup> 工具。

#### 5.14.2 AMD 平台上的 IBS

指令采样 (IBS) 是 AMD64 处理器的一项功能，可用于收集与指令提取和指令执行相关的特定指标。AMD 处理器的处理器流水线由两个独立的阶段组成：一个前端阶段负责提取 AMD64 指令字节，一个后端阶段负责执行“ops”。由于这些阶段在逻辑上是分开的，因此存在两个独立的采样机制：IBS Fetch 和 IBS Execute。

- IBS Fetch 监控流水线的前端，并提供有关 ITLB (命中或未命中)、I-cache (命中或未命中)、获取地址、获取延迟等信息。
- IBS Execute 监控流水线的后端，通过跟踪单个 op 的执行来提供关于指令执行行为的信息。例如：分支 (是否被执行，是否被预测)、加载/存储 (D-cache 和 DTLB 中的命中或未命中，线性地址，加载延迟)。

PMC 和 IBS 在 AMD 处理器之间存在一些重要差异。PMC 计数器是可编程的，而 IBS 则类似于固定计数器。IBS 计数器只能用于监控启用或禁用，无法针对任何选择性事件进行编程。IBS Fetch 和 Execute 计数器可以独立启用/禁用。使用 PMC 时，用户必须提前决定要监控哪些事件。使用 IBS 时，会为每个采样的指令收集丰富的数据，然后由

<sup>103</sup> PEBS 抓取器工具 - <https://github.com/andikleen/pmu-tools/tree/master/pebs-grabber>。需要 root 权限。

用户分析他们感兴趣的数据部分。IBS 选择并标记要监控的指令，然后捕获该指令执行期间引起的微架构事件。有关 Intel PEBS 和 AMD IBS 的更多详细比较，请参见 [Sasongko et al., 2023]。

由于 IBS 被集成到处理器流水线中并作为固定事件计数器，因此样本收集开销最小。分析器需要处理 IBS 生成的数据，这些数据可能非常庞大，具体取决于采样间隔、配置的线程数、是否配置 Fetch/Execute 等。在 Linux 内核版本 6.1 之前，IBS 总是为所有内核收集样本。这个限制会导致巨大的数据收集和处理开销。从内核 6.2 开始，Linux perf 只支持为配置的内核收集 IBS 样本。

IBS 由 Linux perf 和 AMD uProf 分析器支持。以下是收集 IBS 执行和获取样本的示例命令：

```
$ perf record -a -e ibs_op/cnt_ctl=1,l3missonly=0/ -- benchmark.exe
$ perf record -a -e ibs_fetch/l3missonly=1/ -- benchmark.exe
$ perf report
```

在上一个命令中，`cnt_ctl=0` 表示计数时钟周期，`cnt_ctl=1` 表示在间隔期间计数已分配的 ops；`l3missonly=1` 只保留具有 L3 未命中的样本。请注意，在上述两个命令中，都使用了 `-a` 选项来为所有内核收集 IBS 样本，否则 perf 在 Linux 内核 6.1 上无法收集样本。从 6.2 版本开始，除非您想要为所有内核收集 IBS 样本，否则不再需要 `-a` 选项。`perf report` 命令将显示类似于常规 PMU 事件的与函数和源代码行关联的样本，但会提供我们稍后讨论的附加功能。

#### 5.14.3 ARM 平台上的 SPE

Arm Statistical Profiling Extension (SPE) 是一项架构功能，旨在增强 Arm CPU 内的指令执行性能分析。自 2019 年推出的 Neoverse N1 核心以来，这项功能就可用。SPE 功能扩展被指定为 Armv8-A 架构的一部分，从 Arm v8.2 起提供支持。与其他解决方案相比，SPE 更类似于 AMD IBS，而不是 Intel PEBS。类似于 IBS，SPE 与通用性能监测器计数器 (PMC) 分开，但它只有一个机制，而不是两种类型的 IBS（获取和执行）。

SPE 采样过程内置于指令执行流水线中。样本收集仍然基于可配置的间隔，但操作是根据统计信息选择的。每个采样的操作都会生成一个样本记录，其中包含有关此操作执行的各种数据。SPE 记录保存指令地址，负载和存储访问的数据的虚拟和物理地址，数据访问的来源（缓存或 DRAM）以及时间戳，以与系统中其他事件进行关联。此外，它还可以提供各种流水线阶段的延迟，例如发出延迟（从调度到执行）、转换延迟（虚拟到物理地址转换的周期数）和执行延迟（功能单元中负载/存储的延迟）。白皮书 [Arm, 2023b] 更详细地描述了 ARM SPE，并展示了一些使用它的优化示例。

类似于 Intel PEBS 和 AMD IBS，ARM SPE 有助于减少采样开销并支持更长的收集。除此之外，它还支持样本记录的后过滤，这有助于减少存储所需内存。

SPE 分析已在 Linux perf 工具中启用，可以使用以下方式：

```
$ perf record -e arm_spe_0/<controls>/ -- test_program
$ perf report --stdio
$ spe-parser perf.data -t csv
```

其中 `<controls>` 允许您可选地指定收集的各种控件和过滤器。`perf report` 将根据用户使用 `<controls>` 选项所要求的内容提供通常的输出。`spe-parser`<sup>104</sup> 是由 ARM 工程师开发的工具，用于解析捕获的 perf 记录数据并将所有 SPE 记录保存到 CSV 文件中。

<sup>104</sup> ARM SPE 解析器 - <https://gitlab.arm.com/telemetry-solution/telemetry-solution>

#### 5.14.4 精确事件

现在我们已经介绍了高级采样功能，让我们讨论一下如何使用它们来改善性能分析。我们将从精确事件的概念开始。

性能分析的一个主要问题是精确地定位导致特定性能事件的指令。正如 Section 5.5 中所讨论的，基于中断的采样基于计数特定性能事件并等待其溢出。当溢出中断发生时，处理器需要一段时间停止执行并标记导致溢出的指令。对于现代复杂的乱序 CPU 架构来说，这一点尤其困难。

它引入了滑动的概念，滑动定义为导致事件的 IP (指令地址) 与事件被标记的 IP 之间的距离。滑动使得难以发现导致性能问题的指令。考虑一个具有大量缓存未命中的应用程序，其热门汇编代码如下所示：

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

分析器可能会将 load3 标记为导致大量缓存未命中的指令，而实际上，真正的罪魁祸首是 load1。对于高性能处理器，这种滑动可能数百条处理器指令。这通常会让性能工程师感到非常困惑。有兴趣的读者可以访问 Intel 开发者专区网站: <https://software.intel.com/en-us/vtune-help-hardware-event-skid><sup>105</sup> 了解更多关于此类问题基础原因的信息。

通过让处理器本身存储指令指针 (以及其他信息) 可以缓解滑移问题。使用 Intel PEBS 时，PEBS 记录中的 EventingIP 字段指示导致事件的指令。这通常仅适用于受支持事件的一个子集，称为“精确事件”。可以在 [Intel, 2023b, 第 3B 卷, 第 20 章 性能监控] 中找到特定微架构的精确事件完整列表。有关使用 PEBS 精确事件缓解滑移的示例，请参见 easyperf 博客: <https://easyperf.net/blog/2018/08/29/Understanding-performance-events-skid><sup>106</sup>

以下是 Skylake 微架构的精确事件列表：

|                   |                          |                  |                   |
|-------------------|--------------------------|------------------|-------------------|
| INST_RETIRE.*     | OTHER_ASSISTS.*          | BR_INST_RETIRE.* | BR_MISP_RETIRE.*  |
| FRONTEND_RETIRE.* | HLE_RETIRE.*             | RTM_RETIRE.*     | MEM_INST_RETIRE.* |
| MEM_LOAD_RETIRE.* | MEM_LOAD_L3_HIT_RETIRE.* |                  |                   |

其中 \* 表示组内所有子事件都可以配置为精确事件。

使用 Intel 平台上 Linux perf 的用户应在上述列出的事件之一中添加 pp 后缀以启用精确标记：

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

对于 AMD IBS 和 ARM SPE，所有收集的样本在设计上都是精确的，因为硬件会捕获确切的指令地址。事实上，它们都以非常相似的方式工作。每当溢出发生时，该机制将导致溢出的指令保存到专用缓冲区中，然后由中断处理程序读取。由于地址被保留，因此 IBS 和 SPE 样本与指令的关联是精确的。

精确事件为性能工程师提供了便利，因为它们有助于避免误导性数据，这些数据经常让初学者甚至高级开发人员感到困惑。TMA 方法论依赖精确事件来定位低效执行发生的源代码确切行。

## 5.15 分析内存访问

内存访问是许多应用程序性能的关键因素。PEBS 和 IBS 都能够收集程序中内存访问的详细信息。例如，您不仅可以对加载进行采样，还可以收集它们的目标地址和访问延迟。请记住，这并不跟踪所有存储和加载。否则，开销会太大。相反，它只分析大约每 10 万次访问中的一个访问。您可以自定义每秒需要多少个样本。通过足够的样本收集，可以提供内存访问的准确统计图片。

<sup>105</sup> 硬件事件滑移 - <https://software.intel.com/en-us/vtune-help-hardware-event-skid>

<sup>106</sup> 性能滑移 - <https://easyperf.net/blog/2018/08/29/Understanding-performance-events-skid>

在 PEBS 中，允许实现此功能的功能称为数据地址分析 (DLA)。为了提供有关采样加载和存储的更多信息，它使用 PEBS 设施内的“Data Linear Address”和“Latency Value”字段（参见图 42）。如果性能事件支持 DLA 功能并且 DLA 已启用，处理器将转储所采样内存访问的内存地址和延迟。您还可以过滤延迟高于某个阈值的内存访问。这对于查找可能成为许多应用程序性能瓶颈的长延迟内存访问非常有用。

使用 IBS Execute 和 ARM SPE 采样，您还可以深入分析应用程序执行的内存访问。一种方法是转储收集的样本并手动处理它们。IBS 会保存确切的线性地址、其延迟、访问来自何处（缓存或 DRAM）、以及它是否在 DTLB 中命中或未命中。SPE 可用于估计内存子系统组件的延迟和带宽、估计单个加载/存储的内存延迟等等。

这些扩展最重要的用例之一是检测真实共享和虚假共享，我们将在 Section 11.7 中讨论。Linux `perf c2c` 工具大量依赖所有三种机制（PEBS、IBS 和 SPE）来查找可能遇到真实/虚假共享的争用内存访问：它匹配不同线程的加载/存储地址，并检查命中是否发生在由其他线程修改的缓存行中。

## 问题和练习

1. TMA 性能方法论的四个一级类别是什么？
2. HW 事件采样的优势是什么？
3. 什么是性能事件滑移？
4. 研究用于开发/基准测试的机器内部 CPU 上可用的性能分析功能。

## 章节总结

- 仅当所有高级性能问题都已修复后，才建议使用硬件功能进行低级微调。调整设计不良的算法是时间上的浪费。一旦消除所有主要性能问题，就可以使用 CPU 性能监控功能来分析和进一步调整应用程序。
- 自顶向下的微架构分析 (TMA) 方法是一种非常强大且易学的方法，即使是经验不足的开发人员也可以轻松使用它来识别程序对 CPU 微架构的无效使用。TMA 是一个迭代过程，包括多个步骤，其中包括描述工作负载和定位源代码中瓶颈发生的确切位置。我们建议 TMA 应该成为每次低级调整工作的起点之一。
- 分支记录机制（例如 Intel 的 LBR、AMD 的 LBR 和 ARM 的 BRBE）在执行程序的同时连续记录最近的分支结果，从而导致最小的减速。这些设施的主要用途之一是收集调用堆栈。此外，它们还有助于识别热点分支、预测错误率并实现机器代码的精确计时。
- 现代处理器通常提供基于硬件的采样功能，用于高级性能分析。此类功能通过将多个样本存储到专用缓冲区而不使用软件中断来降低采样开销。它们还引入了“精确事件”，能够精确定位导致特定性能事件的确切指令。此外，还有一些其他不太重要的用例。此类基于硬件的采样功能的示例实现包括 Intel 的 PEBS、AMD 的 IBS 和 ARM 的 SPE。
- Intel 处理器跟踪 (PT) 是一项 CPU 功能，通过以高度压缩的二进制格式编码数据包来记录程序执行，这些数据包可用于重建带有每个指令时间戳的执行流。PT 具有广泛的覆盖范围和相对较小的开销。其主要用途是事后分析和查找性能问题根本原因。Intel PT 功能在附录 D 中介绍。基于 ARM 架构的处理器也具有称为 ARM CoreSight 的跟踪功能<sup>107</sup>，但它主要用于调试而非性能分析。

性能分析器利用本章介绍的硬件功能来实现许多不同类型的分析。

<sup>107</sup> ARM CoreSight - <https://developer.arm.com/ip-products/system-ip/coresight-debug-and-trace>

---

## 6 性能分析工具概述

在上一章中，我们探讨了现代处理器中实现的用于辅助性能分析的功能。然而，如果你直接开始使用这些功能，很快就会变得非常微妙，因为需要大量的低级编程才能利用它们。幸运的是，性能分析工具处理了所有必要的复杂性，以有效地使用这些硬件性能监控功能。它使分析变得顺利，但了解工具如何获取和解释数据是至关重要的。这就是为什么我们在讨论 CPU 性能监控功能之后再讨论分析工具的原因。

本章简要介绍了主要平台上最流行的工具。选择将取决于你使用的操作系统和 CPU。一些工具是跨平台的，但大多数不是，因此了解哪些工具对你可用至关重要。这些分析工具通常由硬件供应商自己开发和维护，因为他们是唯一了解如何正确使用其 CPU 上可用性能监控功能的人。不幸的是，这造成了这样一种情况：如果你需要进行高级性能工程工作，你需要安装一个依赖于你使用的 CPU 的专门工具。

阅读完本章后，花时间练习使用你可能最终会使用的工具。熟悉这些工具的界面和工作流程。对你日常工作中使用的应用程序进行性能分析。即使你找不到任何可操作的见解，当真正需要时，你会更加做好准备。

### 6.1 Intel Vtune

Vtune Profiler（之前称为 VTune Amplifier）是一款适用于基于 x86 架构的机器的性能分析工具，具有丰富的图形用户界面。它可以在 Linux 或 Windows 操作系统上运行。我们跳过了关于 Vtune 对 MacOS 的支持的讨论，因为它不适用于苹果芯片（例如，M1 和 M2），而且基于英特尔的 MacBook 很快就会过时。

Vtune 可以在英特尔和 AMD 系统上使用，许多功能都可以工作。但是，高级基于硬件的采样需要英特尔制造的 CPU。例如，在 AMD 系统上，你将无法使用 Intel Vtune 收集硬件性能计数器。

截至 2023 年初，Vtune 作为一个独立工具或作为英特尔 oneAPI 基础工具包的一部分免费提供。

#### 如何配置

在 Linux 上，Vtune 可以使用两个数据收集器：Linux perf 和 Vtune 自己的驱动程序，称为 SEP。第一种类型用于用户模式采样，但如果你想进行高级分析，你需要构建并安装 SEP 驱动程序，这并不太难。

```
# 进入 vtune 安装的 sepdk 文件夹
$ cd ~/intel/oneapi/vtune/latest/sepdk/src

# 构建驱动程序
$ ./build-driver

# 添加 vtune 组并将你的用户添加到该组
# 创建一个新的 shell，或者重新启动系统
$ sudo groupadd vtune
$ sudo usermod -a -G vtune `whoami`

# 安装 sep 驱动程序
$ sudo ./insmod-sep -r -g vtune
```

完成上述步骤后，你应该能够使用像微架构探索和内存访问之类的高级分析类型。

在 Windows 上，在安装 Vtune 后不需要进行任何额外的配置。收集硬件性能事件需要管理员权限。

#### 它能做什么

- 找到热点：函数、循环、语句。

- 监控各种特定于 CPU 的性能事件，例如分支误判和 L3 缓存未命中。
- 定位这些事件发生的代码行。
- 使用 TMA 方法论对 CPU 性能瓶颈进行特征化。
- 为特定函数、进程、时间段或逻辑核心过滤数据。
- 随时间观察工作负载行为（包括 CPU 频率、内存带宽利用率等）。

Vtune 可以提供关于运行中进程的非常丰富的信息。如果你想要提高应用程序的整体性能，这是一个合适的工具。Vtune 总是提供一段时间内的聚合数据，因此它可用于找到“平均情况”下的优化机会。

### 它不能做什么

- 分析非常短的执行异常。
- 观察系统范围的复杂软件动态。

由于该工具的采样性质，它最终会错过持续时间非常短的事件（例如，亚微秒级）。

### 示例

以下是 VTune 最有趣的功能的一系列截图。为了这个示例，我们使用了 POV-Ray，一个用于创建 3D 图形的光线追踪器。图 43 显示了 povray 3.7 的内置基准测试的热点分析，该基准测试使用 clang14 编译器编译，并使用 -O3 -ffast-math -march=native -g 选项在英特尔 Alderlake 系统上运行（Core i7-1260P，4 个 P 核 + 8 个 E 核），并使用 4 个工作线程。

在图像的左侧部分，你可以看到工作负载中一系列热点函数，以及相应的 CPU 时间百分比和退休指令的数量。在右侧面板中，你可以看到导致调用函数 pov::Noise 的最频繁的调用栈之一。根据该截图，44.4% 的时间函数 pov::Noise 是从 pov::Evaluate\_TPat 被调用的，而 pov::Evaluate\_TPat 又是从 pov::Compute\_Pigment 被调用的。请注意，调用栈并没有一直指向 main 函数。这是因为使用基于硬件的收集时，VTune 使用 LBR 来采样调用栈，其深度有限。在这里很可能涉及递归函数，要进一步调查，用户必须深入代码。

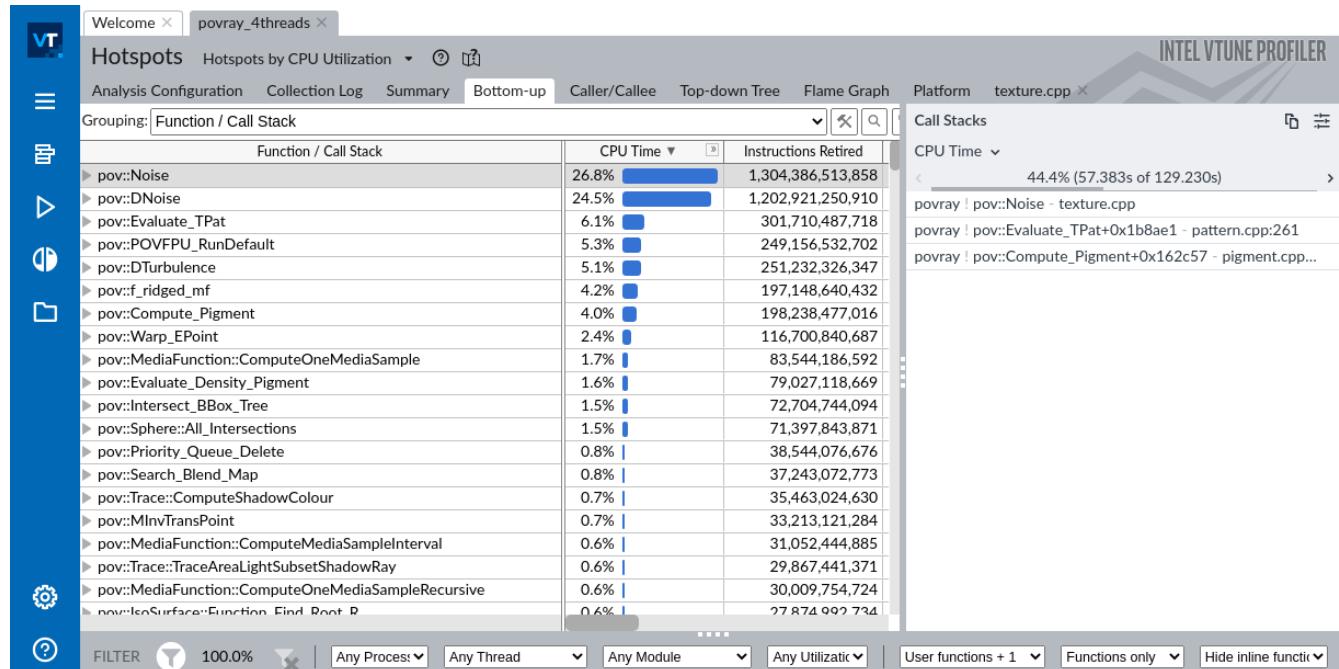


Figure 43: VTune 对 povray 内置基准测试的热点视图。

如果你双击 pov::Noise 函数，你会看到图 44 所示的图像。出于篇幅考虑，只显示了最重要的列。左侧面板显示了

源代码和每行代码对应的 CPU 时间。右侧，你可以看到一些汇编指令以及它们被归因的 CPU 时间。在左侧面板中高亮显示的机器指令对应于右侧面板中的第 476 行。两个面板中所有 CPU 时间百分比的总和等于归因给 pov::Noise 的总 CPU 时间，即 26.8%。

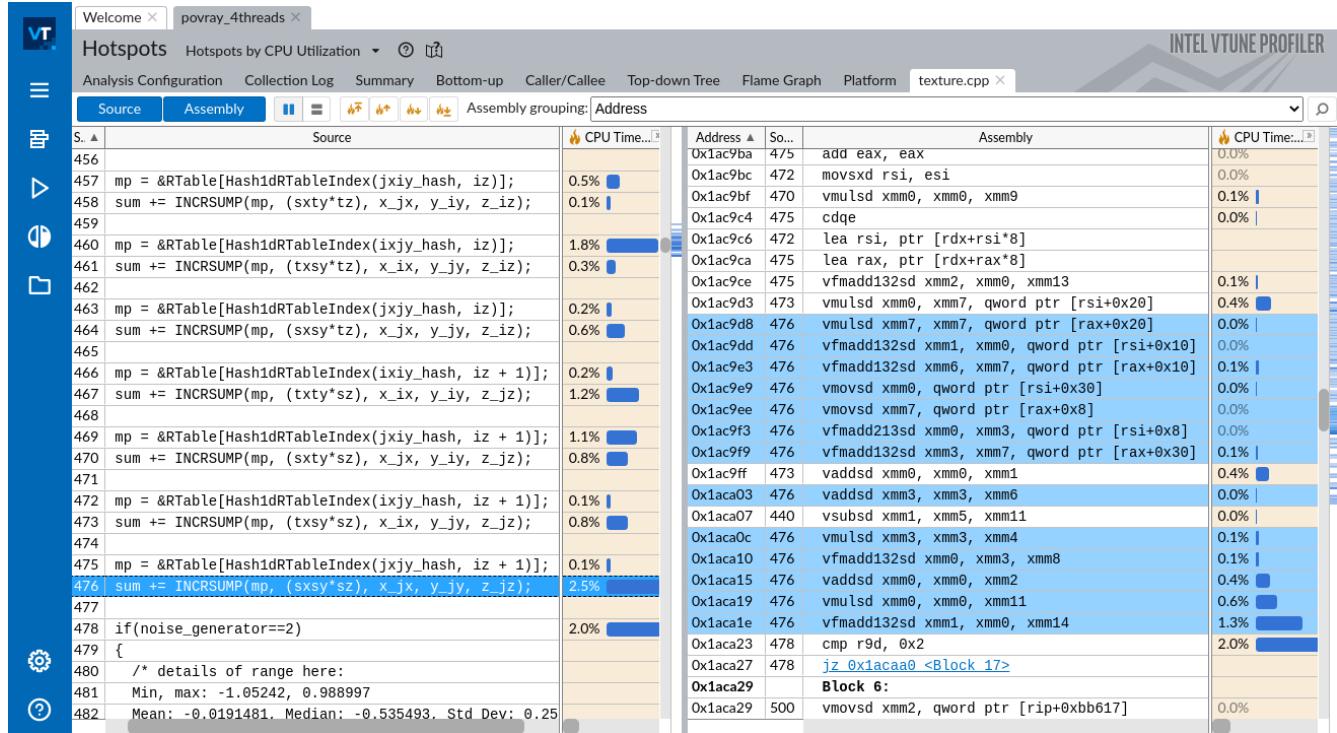


Figure 44: VTune 对 povray 内置基准测试的源代码视图。

当你使用 VTune 来分析运行在英特尔 CPU 上的应用程序时，它可以收集许多不同的性能事件。为了说明这一点，我们运行了不同的分析类型，微架构探索，我们已经在上一章中展示过。那时我们用它来进行自顶向下的微架构分析，而我们也可以用它来观察原始性能事件。要访问原始事件计数，可以将视图切换为硬件事件，如图 45 所示。要启用视图切换，你需要在选项 -> 通用 -> 显示所有适用的视角 中选中标记。还有两个有用的页面，没有显示在图像上：摘要页面提供了从 CPU 计数器收集的原始性能事件的绝对数目，事件计数页面提供了相同数据的函数级别细分。读者可以自行尝试并查看这些视图。

图 45 相当繁忙，需要一些解释。顶部面板（区域 1）是一个时间轴视图，显示了我们四个工作线程随时间的行为，关于 L1 缓存未命中，以及主线程（TID: 3102135）的一些微小活动，它生成所有工作线程。黑色条的高度越高，每时每刻发生的事件（在这种情况下是 L1 缓存未命中）就越多。注意到所有四个工作线程的 L1 未命中偶发性峰值。我们可以使用这个视图来观察工作负载的不同或可重复的阶段。然后为了弄清楚在那个时间执行了哪些函数，我们可以选择一个时间间隔，并点击“筛选进来”以只关注运行时间的那一部分。区域 2 就是这种过滤的一个示例。要查看更新后的函数列表，你可以转到自下而上或，在这种情况下，事件计数视图。这样的过滤和缩放功能在所有 Vtune 时间轴视图中都可用。

区域 3 显示了收集的性能事件及其随时间的分布。这次不是 perf-thread 视图，而是在所有线程上聚合的数据。除了观察执行阶段外，你还可以从中直观地提取一些有趣的信息。例如，我们可以看到执行的分支数很高 (BR\_INST\_RETIRED.ALL\_BRANCHES)，但误判率相当低 (BR\_MISP\_RETIRED.ALL\_BRANCHES)。这可能导致你得出结论，分支误判不是 POV-Ray 的瓶颈。如果你向下滚动，你会看到 L3 未命中的数量为零，而 L2 缓存未命中也非常罕见。这告诉我们，在 99% 的时间里，内存访问请求由 L1 处理，并且其余部分由 L2 处理。两个观察结合起来，我们可以得出结论，应用程序很可能受到计算的限制，即 CPU 忙于计算某些东西，而不是在等待内存或从误判中恢复。

最后，底部面板（区域 4）显示了四个硬件线程的 CPU 频率图表。悬停在不同的时间片上告诉我们，这些核心的频

率在 3.2GHz - 3.4GHz 区域波动。内存访问分析类型还显示了随时间变化的内存带宽（以 GB/s 为单位）。



Figure 45: VTune 对 povray 内置基准测试的性能事件时间轴视图。

### Intel® VTune™ Profiler 中的 TMA

TMA 通过最新的 Intel VTune Profiler 中的“微架构探索”<sup>108</sup> 分析进行展示。图 46 显示了 7-zip 基准测试<sup>109</sup> 的分析摘要。在图表中，你可以看到由于 CPU Bad Speculation（坏的猜测）以及特别是由于误判的分支，大量的执行时间被浪费了。

该工具的美妙之处在于，你可以点击你感兴趣的指标，工具会带你到显示对该特定指标做出贡献的顶级函数的页面。例如，如果你点击 Bad Speculation 指标，你会看到类似于图 47 所示的内容。<sup>110</sup>

从那里，如果你双击 LzmaDec\_DecodeReal2 函数，Intel® VTune™ Profiler 将带你到类似图 48 所示的源码级别视图。高亮显示的行在 LzmaDec\_DecodeReal2 函数中贡献了最大数量的分支误判。

## 6.2 AMD uProf

[AMD uProf](#) 是由 AMD 开发的一款用于监视在 AMD 处理器上运行的应用程序性能的工具。虽然 uProf 也可以用于 Intel 处理器，但你只能使用 CPU 无关的功能。该分析器可以免费下载，并可在 Windows、Linux 和 FreeBSD 上使用。AMD uProf 可用于在多个虚拟机（VM）上进行分析，包括 Microsoft Hyper-V、KVM、VMware ESXi、Citrix Xen，但并非所有 VM 上的所有功能都可用。此外，uProf 还支持分析使用各种语言编写的应用程序，包括 C、C++、Java、.NET/CLR。

<sup>108</sup> VTune 微架构分析 - <https://software.intel.com/en-us/vtune-help-general-exploration-analysis>。在 Intel® VTune Profiler 的 2019 年之前的版本中，它被称为“通用探索”分析。

<sup>109</sup> 7zip 基准测试 - <https://github.com/llvm-mirror/test-suite/tree/master/MultiSource/Benchmarks/7zip>。

<sup>110</sup> TMA 指标的每个函数视图是 Intel® VTune profiler 独有的功能。

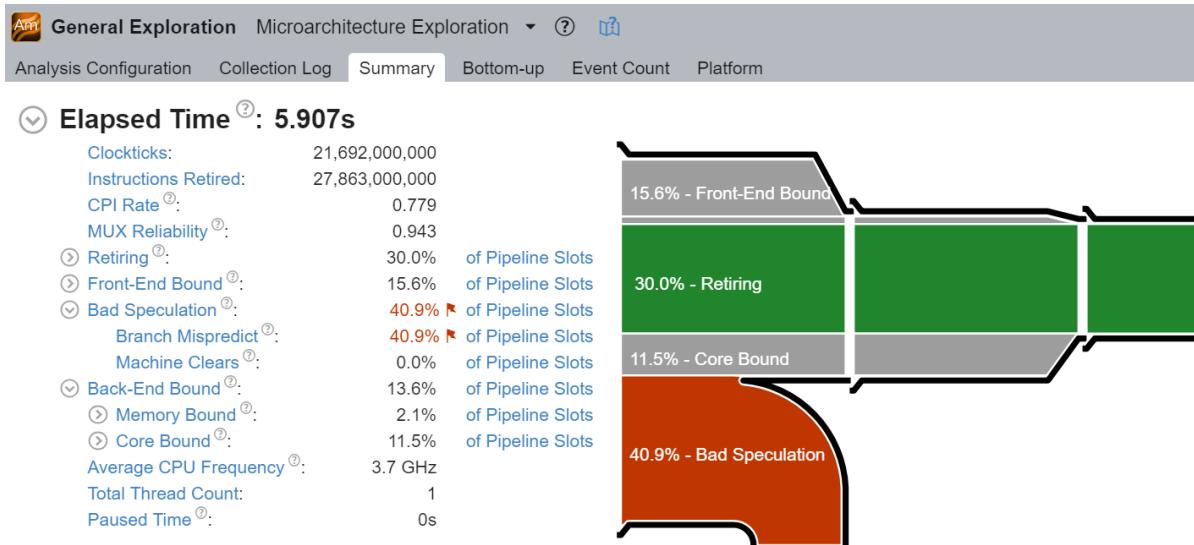


Figure 46: Intel VTune Profiler 中的“微架构探索”分析。

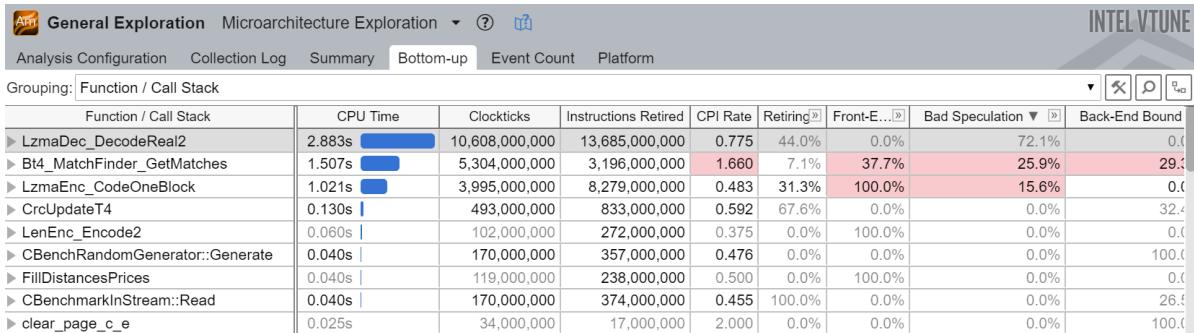


Figure 47: “微架构探索”自下而上视图。

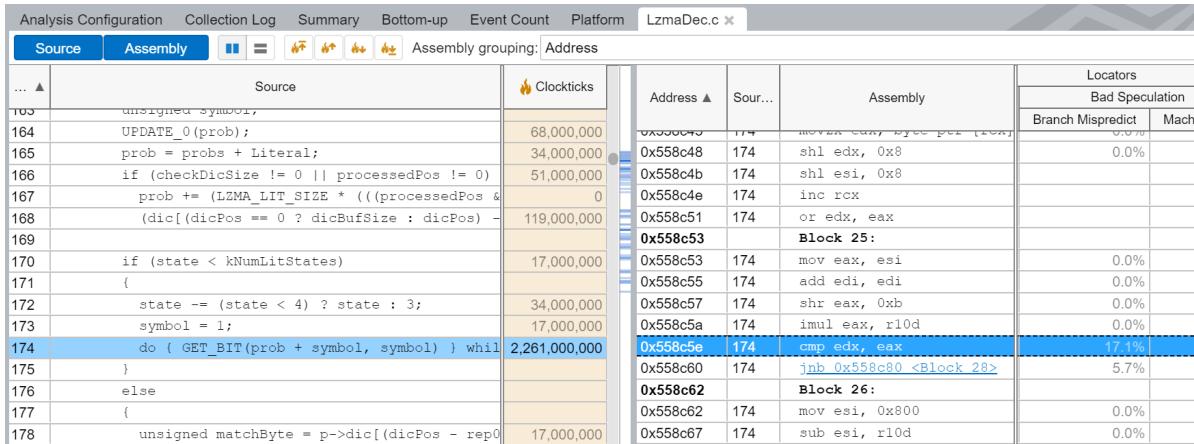


Figure 48: “微架构探索”源代码和汇编视图。

## 如何配置

在 Linux 上，uProf 使用 Linux perf 进行数据收集。在 Windows 上，uProf 使用自己的采样驱动程序，在安装 uProf 时会安装该驱动程序，无需额外配置。AMD uProf 支持命令行界面（CLI）和图形界面（GUI）。CLI 界面需要两个单独的步骤 - 收集和报告，类似于 Linux perf。

## 可以做什么

- 找到热点：函数、语句、指令。
- 监视各种硬件性能事件并定位发生这些事件的代码行。
- 为特定函数或线程过滤数据。
- 观察工作负载的行为随时间的变化：在时间轴图中查看各种性能事件。
- 分析热门调用路径：调用图、火焰图和自下而上图表。

此外，uProf 还可以监视 Linux 上的各种 OS 事件 - 线程状态、线程同步、系统调用、页面错误等。它还允许分析 OpenMP 应用程序以检测线程不平衡，并分析 MPI 应用程序以检测 MPI 群集节点之间的负载不平衡。关于 uProf 各种功能的更多详细信息可在[用户指南<sup>111</sup>](#)中找到。

## 不能做什么

由于工具的采样性质，它最终会错过持续时间非常短的事件。报告的样本是统计估算的数字，大多数情况下足以分析性能，但不是事件的精确计数。

## 示例

为了展示 AMD uProf 工具的外观和感觉，我们在 AMD Ryzen 9 7950X、Windows 11、64 GB RAM 上运行了 Scimark2<sup>112</sup> 基准测试中的密集 LU 矩阵分解组件。

图 49 显示了函数热点分析。在图像的顶部，你可以看到事件时间轴，显示了在应用程序执行的不同时间观察到的事件数量。在右侧，你可以选择要绘制的指标，我们选择了 RETIRED\_BR\_INST\_MISP。注意在时间范围从 20s 到 40s 的分支误判的峰值。你可以选择该区域以密切分析那里发生了什么。一旦你这样做了，它将更新底部面板，仅显示该时间间隔的统计信息。

在时间轴图下方，你可以看到一系列热点函数，以及相应的采样性能事件和计算的指标。事件计数可以显示为：样本计数、原始事件计数和百分比。有许多有趣的数字可以查看，然而，我们不会深入分析，但鼓励读者找出分支误判的性能影响并找到其源头。

在函数表下面，你可以看到所选函数在函数表中的自底向上调用栈视图。正如我们所见，所选的 LU\_factor 函数是从 kernel\_measureLU 被调用的，而 kernel\_measureLU 又是从 main 被调用的。在 Scimark2 基准测试中，这是 LU\_factor 的唯一调用栈，即使它显示了 Call Stacks [5]，这是可以忽略的采集工件。但在其他应用程序中，热点函数可以从许多不同的位置调用，因此你可能还想检查其他调用栈。

如果你双击任何函数，uProf 将打开该函数的源代码/汇编视图。为了简洁起见，我们不显示此视图。在左侧面板上，还有其他视图可用，如指标、火焰图、调用图

视图和线程并发。它们也对分析很有用，但我们决定跳过它们。读者可以自行尝试并查看这些视图。

<sup>111</sup> AMD uProf 用户指南 - <https://www.amd.com/en/developer/uprof.html#documentation>

<sup>112</sup> Scimark2 - <https://math.nist.gov/scimark2/index.html>

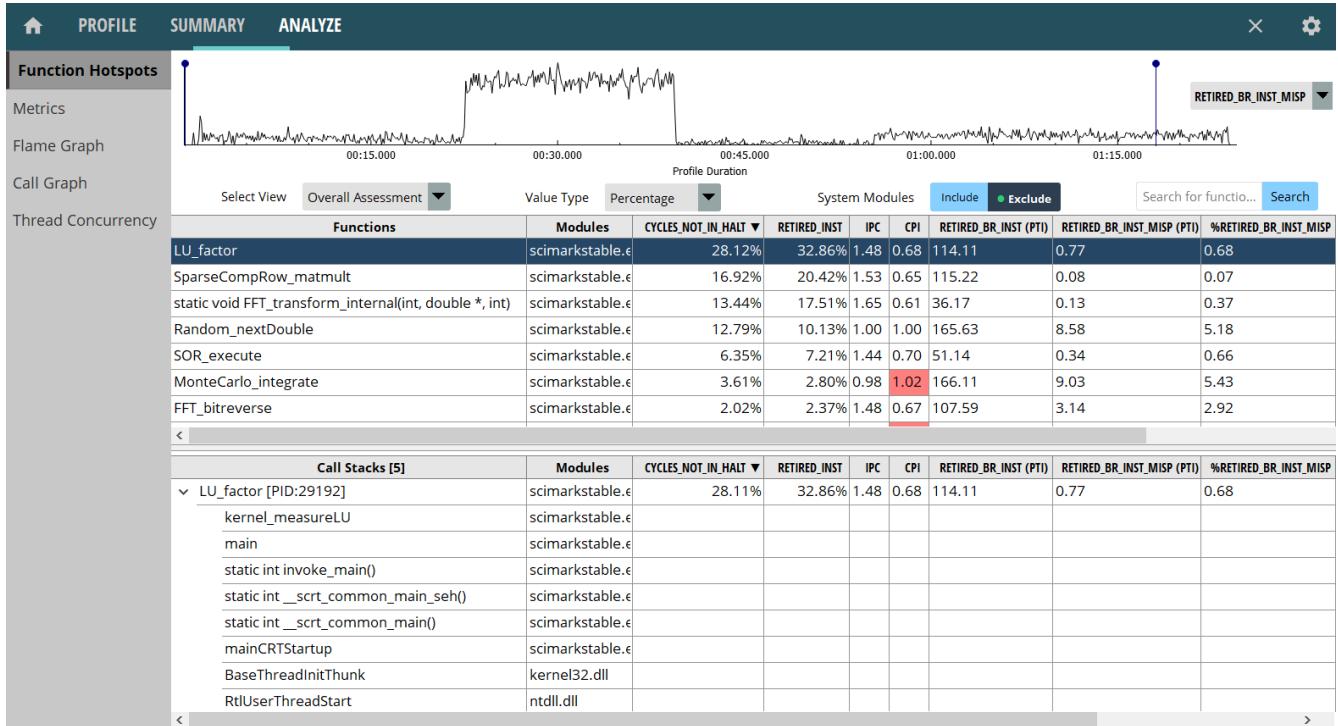


Figure 49: uProf 的函数热点视图。

### 6.3 Apple Xcode Instruments

在 MacOS 上进行初始性能分析最方便的方法是使用 Instruments。它是一个应用程序性能分析器和可视化工具，与 Xcode 一起免费提供。Instruments 建立在从 Solaris 移植到 MacOS 的 DTrace 追踪框架之上。Instruments 拥有许多工具，可用于检查应用程序的性能，并允许我们执行大多数其他性能分析器（如 Intel Vtune）可以执行的基本操作。获取分析器的最简单方法是从 Apple AppStore 安装 Xcode。该工具无需配置，安装后即可立即使用。

在 Instruments 中，你可以使用专门的工具（称为 instruments）来跟踪应用程序、进程和设备的不同方面的情况随时间的变化。Instruments 拥有强大的可视化机制。它在进行分析时收集数据，并实时向你展示结果。你可以收集不同类型的数据，并将它们并排显示，这使你可以看到执行中的模式，相关系统事件，并发现非常微妙的性能问题。

在本章中，我们只展示了“CPU 计数器”工具，这是本书最相关的工具之一。Instruments 还可以可视化 GPU、网络和磁盘活动，跟踪内存分配和释放，捕获用户事件（如鼠标点击），提供关于功耗效率的见解等。关于这些用例的更多信息可以在 Instruments 的文档<sup>113</sup>中找到。

#### 你可以用它做什么

- 访问 Apple M1 和 M2 处理器上的硬件性能计数器
- 找到程序中的热点以及调用栈
- 与源代码并排检查生成的 ARM 汇编代码
- 为时间轴上的选定区间筛选数据

<sup>113</sup> Instruments 文档 - <https://help.apple.com/instruments/mac/current>

你不能用它做什么

示例：编译 Clang

正如我们所宣传的那样，在此示例中，我们将展示如何在搭载 M1 处理器的 Apple Mac mini 上，macOS 13.5.1 Ventura，16 GB RAM 上收集 HW 性能计数器。我们选取了 LLVM 代码库中最大的文件之一，并使用 Clang C++ 编译器版本 15.0 对其进行编译。以下是我们将要进行分析的命令行：

```
$ clang++ -O3 -DNDEBUG -arch arm64 <other options ...> -c
    llvm/lib/Transforms/Vectorize/LoopVectorize.cpp
```

首先，打开 Instruments 并选择“CPU 计数器”分析类型。（这里我们需要稍微提前一点）。它将打开主时间轴视图，如图 51 所示，准备开始分析。但在开始之前，让我们配置收集。单击并按住红色目标图标①，然后选择“Recording Options...”菜单。它将显示如图 50 所示的对话框窗口。在这里，你可以为收集添加 HW 性能监视事件。

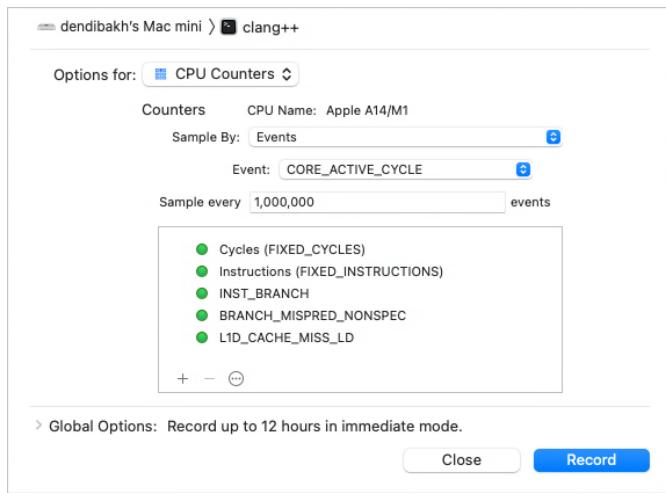


Figure 50: Xcode Instruments: CPU 计数器选项。

据我们所知，Apple 没有在线记录他们的 HW 性能监视事件，但他们在 /usr/share/kpep 中提供了一些带有最小描述的事件列表。有 plist 文件，你可以将其转换为 json。例如，对于 M1 处理器，可以运行：

```
$ plutil -convert json /usr/share/kpep/a14.plist -o a14.json
```

然后使用简单的文本编辑器打开 a14.json。

第二步是设置分析目标。要做到这一点，单击并按住应用程序的名称（在图 51 中标记为②），然后选择你感兴趣的应用程序，如果需要，设置参数和环境变量。现在，你已经准备好开始收集了，请按红色目标图标①。

Instruments 显示一个时间轴，并不断更新有关正在运行的应用程序的统计信息。一旦程序完成，Instruments 将显示类似于图 51 所示的图像。编译花费了 7.3 秒，我们可以看到事件量随时间的变化。例如，分支错误预测在运行时末尾变得更加明显。你可以放大时间轴上的该区间，以检查所涉及的函数。

底部面板显示了数值统计信息。要检查类似于 Intel Vtune 的自下而上视图的热点，选择菜单③中的“Profile”，然后单击菜单④中的“Call Tree”并选中“Invert Call Tree”复选框。这正是我们在图 51 中所做的。

Instruments 显示原始计数以及相对于总数的百分比，如果你想计算次要指标（如 IPC、MPKI 等），则非常有用。在右侧，我们有函数 llvm::FoldingSetBase::FindNodeOrInsertPos 的热调用栈。如果你双击一个函数，则可以查看为源代码生成的 ARM 汇编指令。

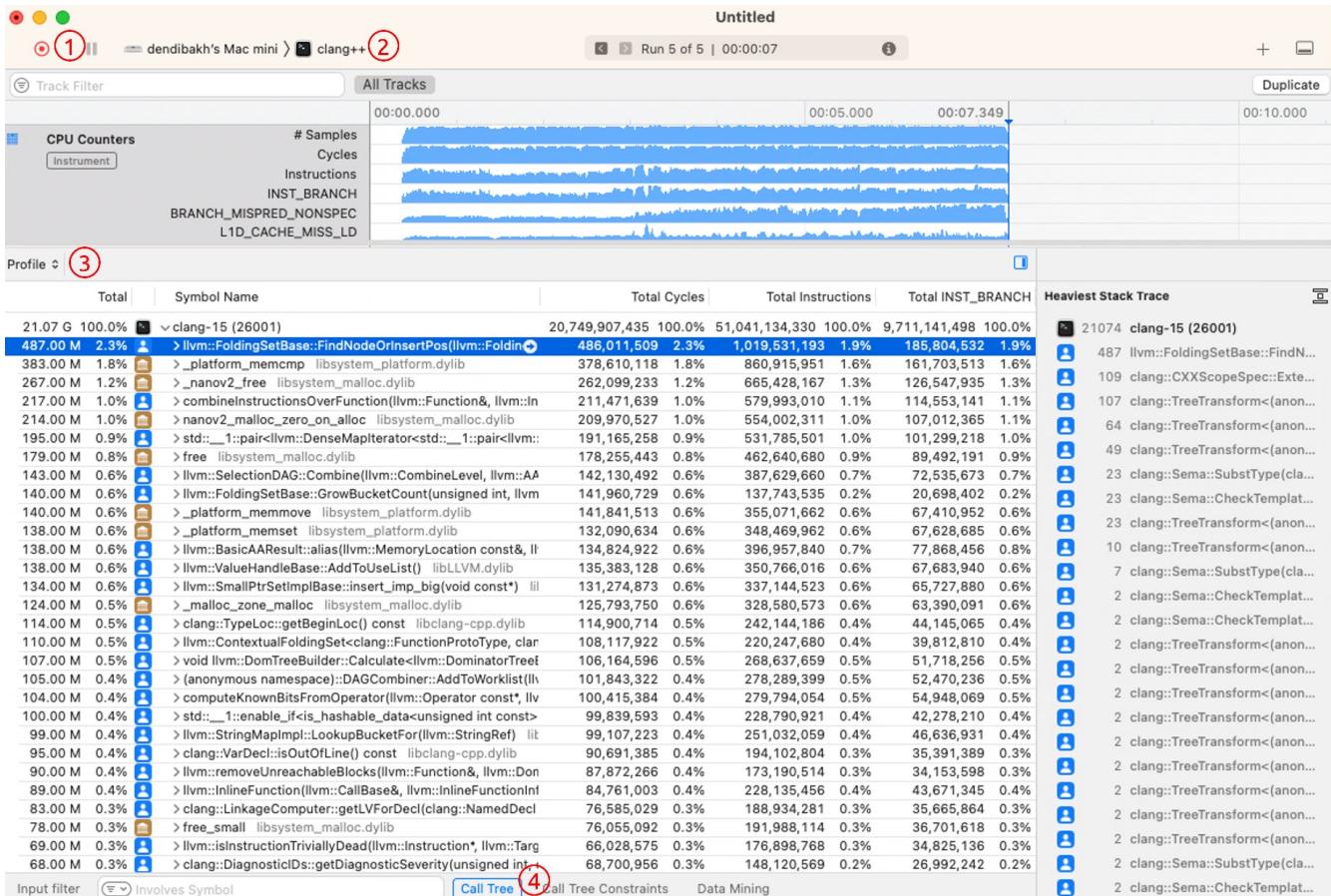


Figure 51: Xcode Instruments: 时间轴和统计面板。

据我们所知，MacOS 平台上没有类似质量的替代性分析工具。高级用户可以通过编写简短（或长）的命令行脚本来使用 dtrace 框架本身，但这超出了本书的范围。

## 6.4 Linux Perf

Linux Perf 是世界上使用最广泛的性能分析器之一，因为它可以在大多数 Linux 发行版上使用，这使其适用于广泛的用户。Perf 在许多流行的 Linux 发行版中都原生支持，包括 Ubuntu、Red Hat、Debian 等。它包含在内核中，因此您可以在任何运行 Linux 的系统上获取操作系统级别的统计信息（页面错误、cpu 迁移等）。截至 2023 年中期，该分析器支持 x86、ARM、PowerPC64、UltraSPARC 和其他一些架构。<sup>114</sup> 这允许访问硬件性能监控功能，例如性能计数器。有关 Linux perf 的更多信息，请访问其维基页面<sup>115</sup>。

### 如何配置它

安装 Linux perf 非常简单，只需一个命令即可完成：

```
$ sudo apt-get install linux-tools-common linux-tools-generic linux-tools-`uname -r`
```

另外，除非安全是您关注的问题，否则请考虑更改以下默认设置：

```
# 允许非特权用户进行内核分析和访问 CPU 事件
$ echo 0 | sudo tee /proc/sys/kernel/perf_event_paranoid
$ sudo sh -c 'echo kernel.perf_event_paranoid=0 >> /etc/sysctl.d/local.conf'
# 启用内核模块符号解析以供非特权用户使用
$ echo 0 | sudo tee /proc/sys/kernel/kptr_restrict
$ sudo sh -c 'echo kernel.kptr_restrict=0 >> /etc/sysctl.d/local.conf'
```

### 它能做什么？

通常，Linux perf 可以完成其他分析器所能做的大多数事情。硬件供应商优先在 Linux perf 中启用他们的功能。因此，当新的 CPU 上市时，perf 已经支持它了。大多数用户将使用两个主要命令：perf stat 和 perf record + perf report。第一个以计数模式收集性能事件的绝对数量，第二个以采样模式分析应用程序或系统。

perf record 命令的输出是原始的示例转储。许多工具构建在 Linux perf 之上，用于解析转储文件并提供新的分析类型。以下是其中最值得注意的：

- 火焰图，参见下一节。
- KDAB Hotspot: <https://github.com/KDAB/hotspot>，<sup>116</sup> 一个使用与 Intel Vtune 非常相似的界面可视化 Linux perf 数据的工具。如果您使用过 Intel Vtune，KDAB Hotspot 将会非常熟悉。有些人将其用作 Intel Vtune 的替代品。
- Netflix FlameScope: <https://github.com/Netflix/flamescope>，<sup>117</sup> 显示应用程序运行时采样事件的热图。您可以观察负载行为的不同阶段和模式。Netflix 工程师使用此工具发现了一些非常微妙的性能漏洞。此外，您可以在热图上选择一个时间范围并仅为该时间范围生成火焰图。

<sup>114</sup> RISCV 目前还不作为官方内核的一部分支持，尽管存在供应商的自定义工具。

<sup>115</sup> Linux perf wiki - [https://perf.wiki.kernel.org/index.php/Main\\_Page](https://perf.wiki.kernel.org/index.php/Main_Page).

<sup>116</sup> KDAB Hotspot - <https://github.com/KDAB/hotspot>.

<sup>117</sup> Netflix FlameScope - <https://github.com/Netflix/flamescope>.

它不能做什么？

Linux perf 是一个命令行工具，缺乏 GUI，这使得过滤数据、观察工作负载行为随时间如何变化、放大运行时的一部分等变得困难。通过 `perf report` 命令提供了有限的控制台输出，这对于快速分析来说已经足够，但不如其他 GUI 分析器方便。幸运的是，正如我们刚才提到的，有一些 GUI 工具可以对 Linux perf 的原始输出进行后处理和可视化。

## 6.5 火焰图

火焰图是一种流行的性能数据可视化方式，可以直观地呈现程序中最频繁的代码路径。它允许我们看到哪些函数调用占用了最大部分的执行时间。图 52 展示了使用 Brendan Gregg 开发的开源脚本<sup>118</sup> 生成的 x264：<https://openbenchmarking.org/test/pts/x264> 视频编码基准测试的火焰图示例。如今，几乎所有性能分析器都可以在配置文件会话期间收集调用堆栈的情况下自动生成火焰图。

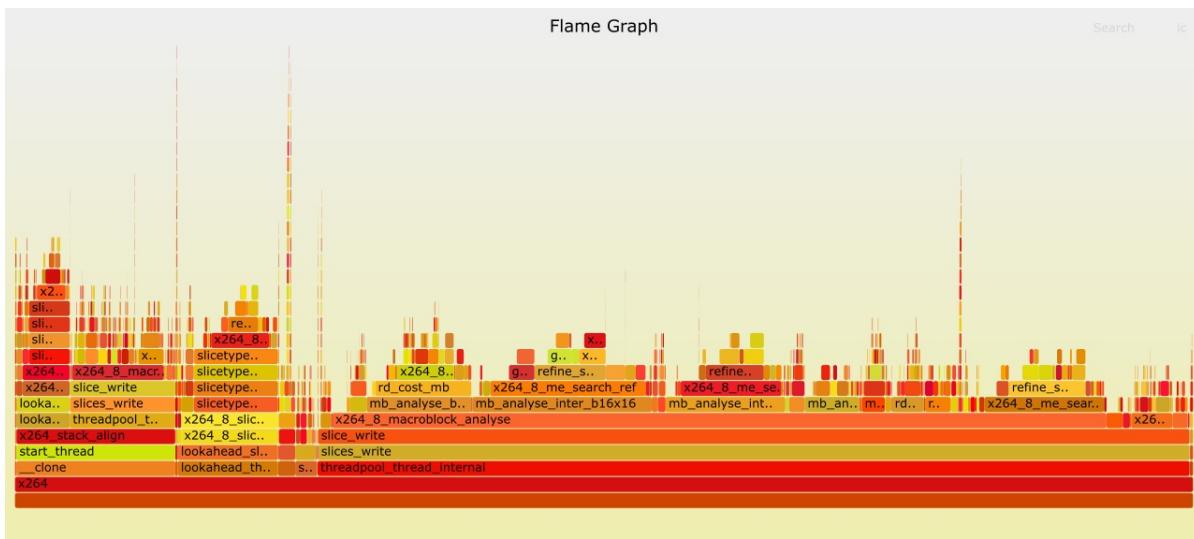


Figure 52: A Flame Graph for x264 benchmark.

在火焰图上，每个矩形（水平条）表示一个函数调用，矩形的宽度表示函数本身及其被调用者所花费的相对执行时间。函数调用是从下到上进行的，因此我们可以看到程序中最热的路径是 `x264 -> threadpool_thread_internal -> ... -> x264_8_macroblock_analyse`。函数 `threadpool_thread_internal` 及其被调用者占用了程序运行时间的 74%。但其自身时间，即函数本身花费的时间则相对较少。同样，我们可以对 `x264_8_macroblock_analyse` 进行相同的分析，它占用了 66% 的运行时间。这种可视化方式可以让您很好地直观地了解程序花费最多时间的地方。

火焰图是可交互的，您可以单击图像上的任何条形，它就会放大到特定的代码路径。您可以一直放大，直到找到与您的期望不符的地方，或者到达叶/尾函数——现在您就有了可以在分析中使用的可操作信息。另一种策略是找出程序中最热的函数（从这个火焰图中无法立即看出来），然后从下往上通过火焰图，试图理解这个最热的函数是从哪里被调用的。

## 6.6 Windows 事件跟踪

微软开发了一项名为 Windows 事件跟踪 (ETW) 的系统级跟踪功能。它最初旨在帮助设备驱动程序开发人员，但后来也发现它可以用于分析通用应用程序。ETW 在所有受支持的 Windows 平台 (x86、x64 和 ARM) 上可用，并提供相应的平台相关安装包。ETW 以结构化事件的形式记录用户和内核代码，并支持完整的调用堆栈跟踪，允许观察运行系统中的软件动态并解决许多棘手的性能问题。

<sup>118</sup> Brendan Gregg 的火焰图 - <https://github.com/brendangregg/FlameGraph>

### 6.6.1 如何配置它

从 Windows 10 开始，使用 `wpr.exe` 可以录制 ETW 数据，无需任何额外下载。但是要启用系统范围的分析，您必须是管理员并启用 `SeSystemProfilePrivilege`。Windows 性能记录器工具支持一组内置录制配置文件，适用于常见性能问题。您可以通过创作带有 `.wprp` 扩展名的自定义性能记录器配置文件 XML 文件来定制您的录制需求。

如果您不仅想要录制还想查看录制的 ETW 数据，则需要安装 Windows Performance Toolkit (WPT)。您可以从 Windows SDK<sup>119</sup> 或 ADK<sup>120</sup> 下载页面下载 Windows Performance Toolkit。Windows SDK 体积庞大，您不一定需要所有部分。在我们的例子中，我们只勾选了 Windows Performance Toolkit 的复选框。您可以将 WPT 作为您自己应用程序的一部分进行再分发。

### 6.6.2 你能用它做什么：

- 查看 CPU 热点，可配置的 CPU 采样率从 125 微秒到 10 秒。默认为 1 毫秒，运行时开销约为 5-10%。
- 哪个线程阻塞了某个线程以及阻塞了多长时间（例如，延迟事件信号、不必要的线程睡眠等）。
- 检查磁盘处理读/写请求的速度以及谁启动了这项工作。
- 检查文件访问性能和模式（包括导致没有磁盘 I/O 的缓存读/写）。
- 跟踪 TCP/IP 堆栈如何如何在网络接口和计算机之间传输数据包。

上述所有项目都在系统范围内记录所有进程，并具有可配置的调用堆栈跟踪（内核和用户模式调用堆栈结合）。您还可以添加自己的 ETW 提供程序，将系统范围的跟踪与您的应用程序行为关联起来。您可以通过检测您的代码来扩展收集的数据量。例如，您可以在源代码中的函数中添加注入进入/离开 ETW 跟踪钩子，以测量特定方法的执行频率。

### 6.6.3 你不能用它做什么：

- 检查 CPU 微架构瓶颈。为此，请使用特定于供应商的工具，例如 Intel VTune、AMD uProf、Apple Instruments 等。

ETW 跟踪捕获了所有进程在系统级别的动态，这很棒，但它可能会生成大量数据。例如，捕获线程上下文切换数据以观察各种等待和延迟很容易在每分钟生成 1-2 GB 的数据。这就是为什么不实际录制大量事件数小时而不覆盖之前存储的跟踪的原因。

## 6.7 记录 ETW 跟踪的工具

以下是一些可以用来捕获 ETW 跟踪的工具列表：

- `wpr.exe`：命令行录制工具，它是 Windows 10 和 Windows Performance Toolkit 的一部分。
- `WPRUI.exe`：一个用于录制 ETW 数据的简单 UI，它是 Windows Performance Toolkit 的一部分。
- `xperf`：`wpr` 的命令行前身，是 Windows Performance Toolkit 的一部分。
- `PerfView`<sup>121</sup>：一个图形化录制和分析工具，主要关注.NET 应用程序。由微软开源。
- `Performance HUD`<sup>122</sup>：一个鲜为人知但功能非常强大的 GUI 工具，用于跟踪 UI 延迟、用户/句柄泄漏，以及通过实时 ETW 录制所有不平衡的资源分配，并实时显示泄漏/阻塞调用堆栈跟踪。
- `ETWController`<sup>123</sup>：一个录制工具，能够录制键盘输入和屏幕截图以及 ETW 数据。还支持同时在两台机器上进行分布式分析。由 Alois Kraus 开源。

<sup>119</sup> Windows SDK 下载 <https://developer.microsoft.com/en-us/windows/downloads/sdk-archive/>

<sup>120</sup> Windows ADK 下载 <https://learn.microsoft.com/en-us/windows-hardware/get-started/adk-install#other-adk-downloads>

<sup>121</sup> PerfView <https://github.com/microsoft/perfview>

<sup>122</sup> Performance HUD <https://www.microsoft.com/en-us/download/100813>

<sup>123</sup> ETWController <https://github.com/alois-xx/etwcontroller>

- **UIForETW**:<sup>124</sup> 一个围绕 xperf 的包装器，具有记录 Google Chrome 问题数据的特殊选项。还可以录制键盘和鼠标输入。由 Bruce Dawson 开源。

### 6.7.1 查看和分析 ETW 跟踪的工具

- **Windows Performance Analyzer (WPA)**: 查看 ETW 数据最强大的 UI。WPA 可以可视化和叠加磁盘、CPU、GPU、网络、内存、进程等等更多的数据源，以便全面了解您的系统行为及其执行操作。虽然 UI 非常强大，但对于初学者来说也可能相当复杂。WPA 支持插件来处理来自其他来源的数据，而不仅仅是 ETW 跟踪。可以导入 Linux/Android<sup>125</sup> 分析数据，这些数据是由 Linux perf、LTTNG、Perfetto 和以下日志文件格式生成的：dmesg、Cloud-Init、WaLinuxAgent、AndroidLogcat。
- **ETWAnalyzer**:<sup>126</sup> 读取 ETW 数据并生成聚合摘要 JSON 文件，这些文件可以在命令行进行查询、过滤和排序，或者导出为 CSV 文件。
- **PerfView**: 主要用于故障排除.NET 应用程序。为垃圾回收和 JIT 编译触发的 ETW 事件被解析并作为报告或 CSV 数据轻松访问。

#### 案例研究 - 程序启动缓慢

接下来，我们将通过一个示例，使用 ETWController 捕获 ETW 跟踪并使用 WPA 进行可视化。

**问题描述:** 在 Windows 资源管理器中双击下载的可执行文件时，它的启动会明显延迟。似乎有些东西延迟了进程启动。可能是什么原因？磁盘速度慢？

#### 6.7.1.1 设置

- 下载 ETWController 以录制 ETW 数据和屏幕截图。
- 下载最新的 Windows 11 Performance Toolkit<sup>127</sup> 以便能够使用 WPA 查看数据。确保在系统环境对话框中将较新的 Win 11 wpr.exe 放置在您的路径之前，方法是将 WPT 的安装文件夹移动到 C:\\Windows\\System32 之前。它应该如下所示：

```
C> where wpr
C:\Program Files (x86)\Windows Kits\10\Windows Performance Toolkit\wpr.exe
C:\Windows\System32\wpr.exe
```

#### 6.7.1.2 捕获痕迹

- 启动 ETWController。
- 选择 CSwitch 配置文件以跟踪线程等待时间以及其他默认录制设置。保持复选框“记录鼠标点击”和“获取循环屏幕截图”启用，以便稍后借助屏幕截图导航到慢速点。参见图 53。
- 按“开始录制”。
- 从互联网下载一些可执行文件，解压缩它并双击可执行文件启动它。
- 之后，您可以通过按“停止录制”按钮停止分析。

第一次停止分析需要更长的时间，因为所有托管代码都会生成合成 pdb，这是一个一次性操作。分析达到已停止状态后，您可以按“在 WPA 中打开”按钮，将 ETL 文件加载到 Windows Performance Analyzer 中，并附带 ETWController 提供的配置文件。CSwitch 配置文件会生成大量数据，这些数据存储在 4 GB 的环形缓冲区中，允许您在最旧的事件

<sup>124</sup> UIforETW <https://github.com/google/UIforETW>

<sup>125</sup> Microsoft Performance Tools Linux / Android <https://github.com/microsoft/Microsoft-Performance-Tools-Linux-Android>

<sup>126</sup> ETWAnalyzer <https://github.com/Siemens-Healthineers/ETWAnalyzer>

<sup>127</sup> Windows SDK 下载 <https://developer.microsoft.com/en-us/windows/downloads/sdk-archive/>

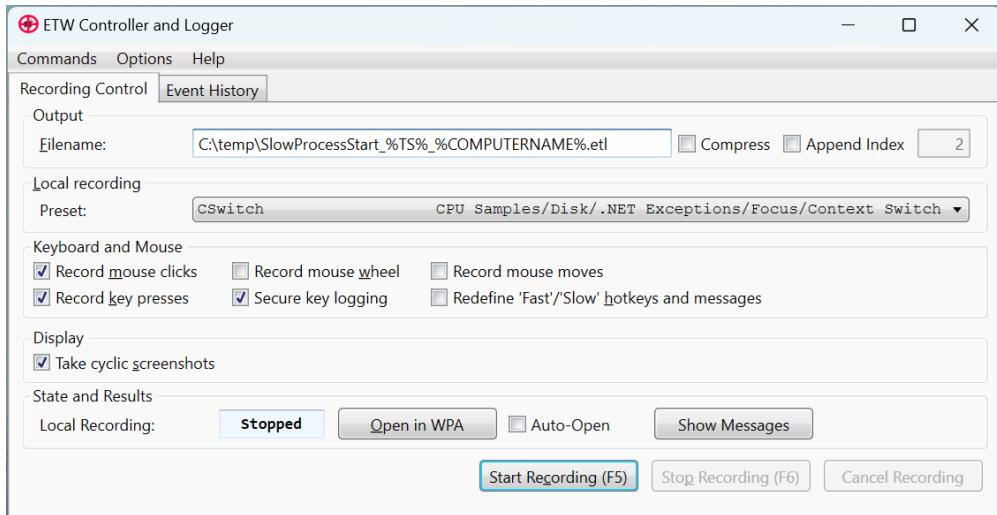


Figure 53: 使用 ETWController UI 启动 ETW 收集.

被覆盖之前录制 1-2 分钟。有时在正确的时间点停止分析有点艺术气息。如果您遇到偶发问题，可以将录制保持启用数小时，并在事件（例如文件中由轮询脚本检查的日志条目）出现时停止录制，以在问题发生时停止录制。

Windows 支持事件日志和性能计数器触发器，允许在性能计数器达到阈值或特定事件写入事件日志时启动脚本。如果您需要更复杂的停止触发器，应该看一下 PerfView，它允许定义一个性能计数器阈值，该阈值必须达到并在那里停留 x 秒，然后停止分析。这样，随机峰值就不会再触发误报。

**6.7.1.3 在 WPA 中分析** 图 54 显示了在 Windows Performance Analyzer (WPA) 中打开的已录制 ETW 数据。WPA 视图分为三个部分：CPU 使用率（采样）、通用事件 和 CPU 使用率（精确）。为了理解它们之间的区别，让我们更深入地研究一下。上面 CPU 使用率（采样）图表可用于识别 CPU 时间花在何处。数据是通过定期间隔对所有正在运行的线程进行采样收集的。与其他分析工具中的热点视图非常相似。

接下来是 通用事件 视图，其中显示鼠标点击和捕获的屏幕截图等事件。请记住，我们在 ETWController 窗口中启用了拦截这些事件的功能。因为事件放在时间线上，所以很容易将 UI 交互与系统如何响应它们相关联。

底部图表 CPU 使用率（精确）使用的数据源与 采样 视图不同。虽然采样数据只会捕获正在运行的线程，但 精确 收集会考虑进程未运行的时间间隔。精确视图的数据来自 Windows 线程调度程序。它跟踪线程运行的时间和所处 CPU (CPU 使用率)、它在内核调用中被阻塞了多长时间 (等待)、它的优先级以及线程等待 CPU 可用有多长时间 (准备时间) 等。因此，精确视图不会显示顶级 CPU 耗用者。但是，这个视图对于理解某个进程被阻塞了多长时间以及 为什么 被阻塞非常有用。

现在我们熟悉了 WPA 界面，让我们观察一下图表。首先，我们可以在时间线上找到 MouseButton 事件 63 和 64。ETWController 将收集期间拍摄的所有屏幕截图保存在一个新建的文件夹中。分析数据本身保存在名为 SlowProcessStart.etl 的文件中，还有一个名为 SlowProcessStart.etl.Screenshots 的新文件夹。该文件夹包含屏幕截图和一个可以在浏览器中查看的 Report.html 文件。每个记录的键盘/鼠标交互都保存在一个以其事件编号命名的文件中，例如 Screenshot\_63.jpg。图 55 (已裁剪) 显示鼠标双击 (事件 63 和 64)。鼠标指针位置标记为绿色方块，除非单击事件发生，则为红色。这使得很容易发现何时何地执行了鼠标单击。

双击标志着我们应用程序等待某事时 1.2 秒延迟的开始。在时间戳 35.1 时，explorer.exe 处于活动状态，因为它试图启动新的应用程序。但后来它没有做太多工作，应用程序也没有启动。相反，MsMpEng.exe 接管执行直到时间 35.7。到目前为止，它看起来像是在下载的可执行文件允许启动之前进行防病毒扫描。但我们不能 100% 确定 MsMpEng.exe 正在阻止新应用程序的启动。

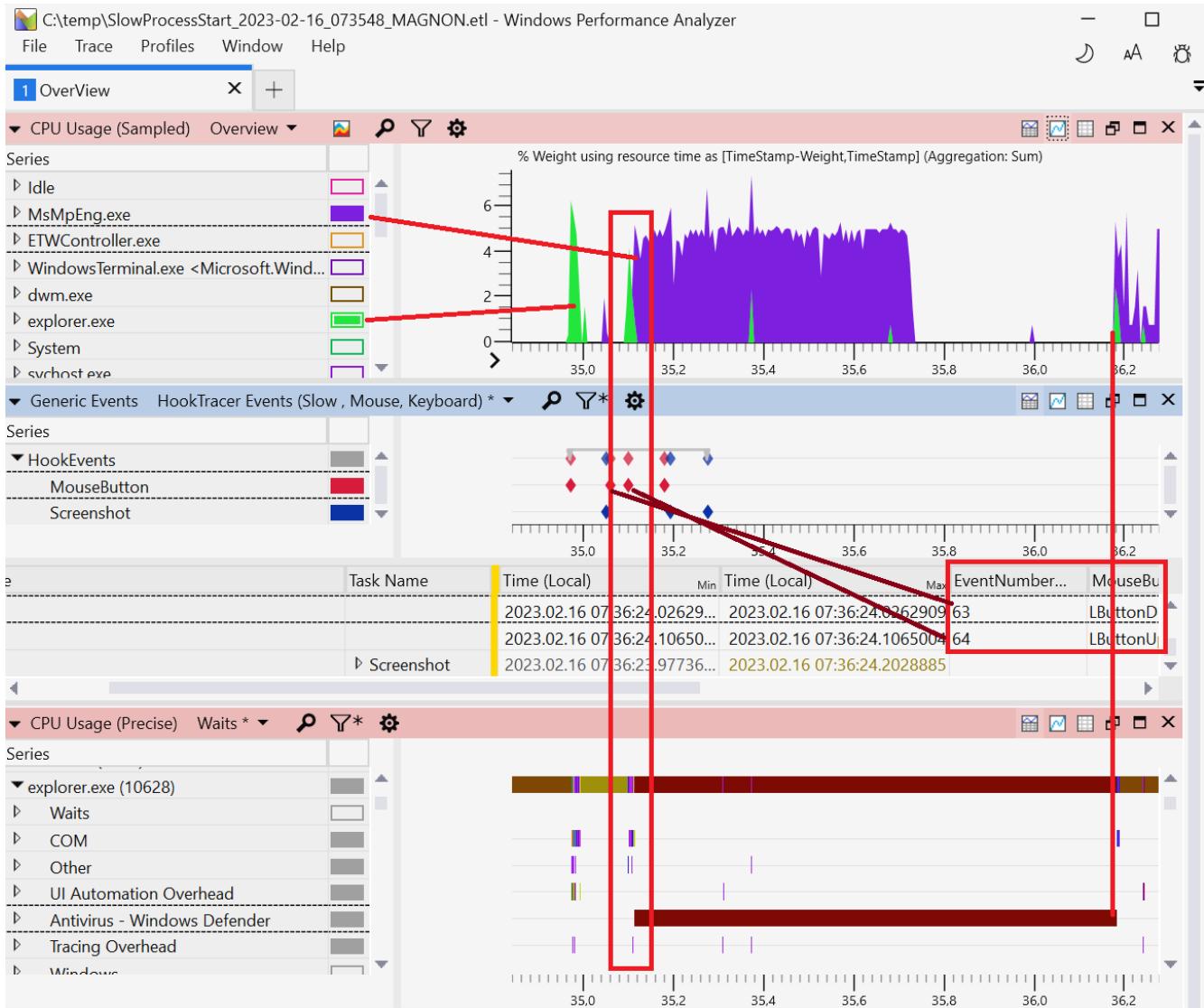


Figure 54: Windows Performance Analyzer 应用程序启动缓慢概览.

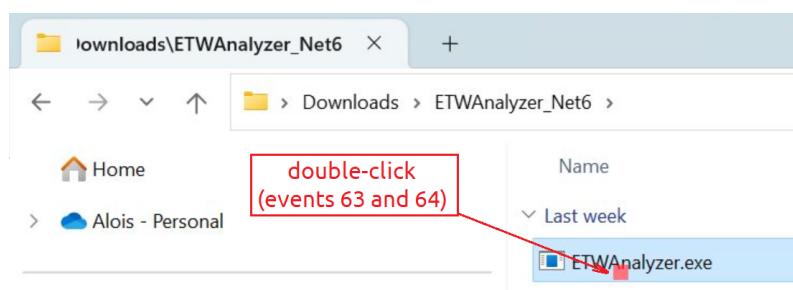


Figure 55: 使用 ETWController 捕获的鼠标点击屏幕截图.

由于我们正在处理延迟，因此我们对 CPU 使用率（精确）等待面板上可用的等待时间感兴趣。在那里，我们找到了我们的 `explorer.exe` 正在等待的进程列表，以条形图的形式可视化，与上部面板的时间线对齐。不难发现对应于 Antivirus - Windows Defender 的长条，其等待时间为 1.068 秒。因此，我们可以得出结论，启动我们应用程序的延迟是由 Defender 扫描活动引起的。如果您深入研究调用堆栈（未显示），您将看到系统调用 `CreateProcess` 在内核中被 `WDFilter.sys`（Windows Defender 筛选器驱动程序）延迟。它会阻止进程启动，直到扫描潜在恶意文件内容为止。防病毒软件可以拦截所有内容，从而导致难以诊断的性能问题，如果没有像 ETW 这样的全面内核视图。

谜题解开了吗？嗯，还不完全是。

知道 Defender 是问题只是第一步。如果您再看上面面板，您会看到延迟并不是完全由繁忙的防病毒扫描引起的。`MsMpEng.exe` 进程从时间 35.1 一直活跃到 35.7，但应用程序并没有立即启动之后。从时间 35.7 到 36.2 有额外的 0.5 秒延迟，在此期间 CPU 大部分处于空闲状态，什么都不做。要找到根本原因，需要跟踪跨进程的线程唤醒历史，我们将在此处不介绍。最后，您会发现一个阻止性的 Web 服务调用 `MpClient.dll!MpClient::CMpSpyNetContext::UpdateSpynetMetrics`，它确实等待某个 Microsoft Defender Web 服务做出响应。如果您还启用了 TCP/IP 或套接字 ETW 跟踪，您还可以找出 Microsoft Defender 与哪个远程端点通信。因此，延迟的第二部分是由 `MsMpEng.exe` 进程等待网络引起的，这也阻止了我们的应用程序运行。

这个案例研究只展示了一个使用 WPA 可以有效分析问题的例子，但还有其他类型的问题。WPA 界面非常丰富且高度可定制。它支持自定义配置文件，可以按照您喜欢的方式配置图表和表格以可视化 CPU、磁盘、文件等。最初，WPA 是为设备驱动程序开发人员开发的，并且内置了一些不专注于应用程序开发的配置文件。`ETWController` 带有自己的配置文件 (`Overview.wpaprofile`)，您可以将其设置为默认配置文件，位于 配置文件 -> 保存启动配置文件 下，以便始终使用性能概览配置文件。

## 6.8 专业和混合性能分析器

到目前为止，我们探索的大部分工具都属于采样性能分析器。当你想识别代码中的热点时，它们非常有用，但在某些情况下，它们可能无法提供足够的粒度进行分析。根据性能分析器的采样频率和程序的行为，大多数函数可能足够快，不会出现在性能分析器中。在某些情况下，您可能想要手动定义程序哪些部分需要始终测量。例如，视频游戏渲染帧（显示在屏幕上的最终图像）平均每秒 60 帧 (FPS)；一些显示器允许高达 144 FPS。在 60 FPS 时，每个帧只有不到 16 毫秒的时间完成工作，然后才能继续下一个帧。开发人员特别关注超过此阈值的帧，因为这会导致游戏中出现明显的卡顿，从而破坏玩家体验。这种情况很难用采样性能分析器捕捉，因为它们通常只提供给定函数所花费的总时间。

开发人员创建了性能分析器，这些性能分析器提供特定环境中的一些有用功能，通常带有您可以用于手动为代码插入标记的标记 API。这允许您观察特定函数或代码块（稍后称为 zone）的性能。继续游戏行业，这个领域有一些工具：一些直接集成到游戏引擎中，如 Unreal，而另一些则作为外部库和工具提供，可以集成到您的项目中。一些最常用的性能分析器是 Tracy、RAD Telemetry、Remotery: <https://github.com/Celtoys/Remotery> 和 Optick: <https://github.com/bombomby/optick>（仅限 Windows）。接下来，我们展示了 Tracy: <https://github.com/wolfpld/tracy>，<sup>128</sup> 因为这似乎是一个更受欢迎的项目，但是这些概念也适用于其他性能分析器。

你可以用 Tracy 做什么：

- 调试程序中的性能异常，例如慢帧。
- 将慢事件与系统中的其他事件关联起来。
- 找到慢事件的共同特征。
- 检查源代码和汇编代码。
- 在代码更改后进行“前后”比较。

<sup>128</sup> Tracy - <https://github.com/wolfpld/tracy>

你不能用 Tracy 做什么：

- 检查 CPU 微架构问题，例如收集各种性能计数器。

### 案例研究：使用 Tracy 分析慢帧

在下面的例子中，我们使用了 ToyPathTracer: <https://github.com/wolfpld/tracy/tree/master/examples/ToyPathTracer><sup>129</sup> 程序，这是一个简单的路径追踪器，类似于光线追踪技术，通过向场景中每个像素发射数千条射线来渲染逼真的图像。为了处理一帧，该实现将每个像素行的处理分配给一个单独的线程。

为了模拟 Tracy 可以帮助找到问题根源的典型场景，我们手动修改了代码，使某些帧比其他帧消耗更多时间。Listing ?? 显示了添加了 Tracy 仪器的代码草图。请注意，我们随机选择帧来减慢速度。此外，我们还包含了 Tracy 的头文件，并向我们想要跟踪的函数添加了 ZoneScoped 和 FrameMark 宏。FrameMark 宏可以插入到性能分析器中标识单个帧。每个帧的持续时间将在时间线上可见，这非常有用。

清单：Tracy 仪器 ~~~~ {#lst:TracyInstrumentation.cpp} #include "tracy/Tracy.hpp"

```
void TraceRowJob() { ZoneScoped;
if (frameCount == randomlySelected) DoExtraWork();
// ...
}
```

void RenderFrame() { ZoneScoped; for (...) { TraceRowJob(); } FrameMark; } ~~~~ 每个帧可以包含许多区域，由 ZoneScoped 宏指定。与帧类似，一个区域可以有许多实例。每次我们进入一个区域时，Tracy 都会捕获该区域新实例的统计数据。ZoneScoped 宏在堆栈上创建一个对象，该对象将记录对象范围内代码的运行时活动。Tracy 将此范围称为“区域”。在进入区域时，会捕获当前的时间戳。一旦函数退出，对象将记录一个新的 timestamp 并将此时间数据与函数名称一起存储。

Tracy 有两种操作模式：它可以存储所有时间数据，直到分析器连接到应用程序（默认模式），或者它只能在分析器连接时开始记录。后一个选项可以通过在编译应用程序时指定 TRACY\_ON\_DEMAND 预处理器宏来启用。如果要分发可以根据需要进行分析的应用程序，则应首选此模式。使用此选项，跟踪代码可以编译到应用程序中，并且除非附加了分析器，否则它对正在运行的程序几乎没有开销。分析器是一个单独的应用程序，它连接到正在运行的应用程序以捕获和显示实时分析数据，也称为“飞行记录器”模式。分析器可以在单独的机器上运行，这样它就不会干扰正在运行的应用程序。但是请注意，这并不意味着由插桩代码引起的运行时开销消失了——它仍然存在，但在这种情况下避免了可视化数据的开销。

我们使用 Tracy 调试程序并找出为什么某些帧比其他帧慢的原因。数据是在一台配备 Ryzen 7 5800X 处理器的 Windows 11 机器上捕获的。该程序是用 MSVC 19.36.32532 编译的。Tracy 图形界面非常丰富，不幸的是太难容纳在一张屏幕截图中，所以我们将其分解。在顶部，有一个时间线视图，如图 56 所示，已裁剪以适合页面。它仅显示第 76 帧的一部分，渲染该帧需要 44.1 毫秒。在该图上，我们看到了在该帧期间处于活动状态的 Main thread 和五个 WorkerThread。所有线程，包括主线程，都在执行工作以推进最终图像的渲染进度。正如我们之前所说，每个线程在 TraceRowJob 区域内处理一行像素。每个 TraceRowJob 区域实例包含许多较小的区域，这些区域不可见。Tracy 会折叠内部区域并仅显示折叠实例的数量——例如，主线程中第一个 TraceRowJob 下方的数字“4,109”表示的意思。请注意，DoExtraWork 区域的实例嵌套在 TraceRowJob 区域下。这种观察已经可以导致发现，但在实际应用中可能并不那么明显。现在让我们先放下这个。

在主面板的正上方，有一个直方图显示所有记录的帧的时间，请参见图 57。它可以更容易地发现可能导致卡顿的长时间运行的帧。它可以更容易地发现那些比平均时间更长才能完成的帧。在此示例中，大多数帧大约需要 33 毫秒（黄色条）。但是也有一些帧需要更长的时间，并用红色标记。如屏幕截图所示，将鼠标悬停在直方图中的条形图上

<sup>129</sup> ToyPathTracer - <https://github.com/wolfpld/tracy/tree/master/examples/ToyPathTracer>

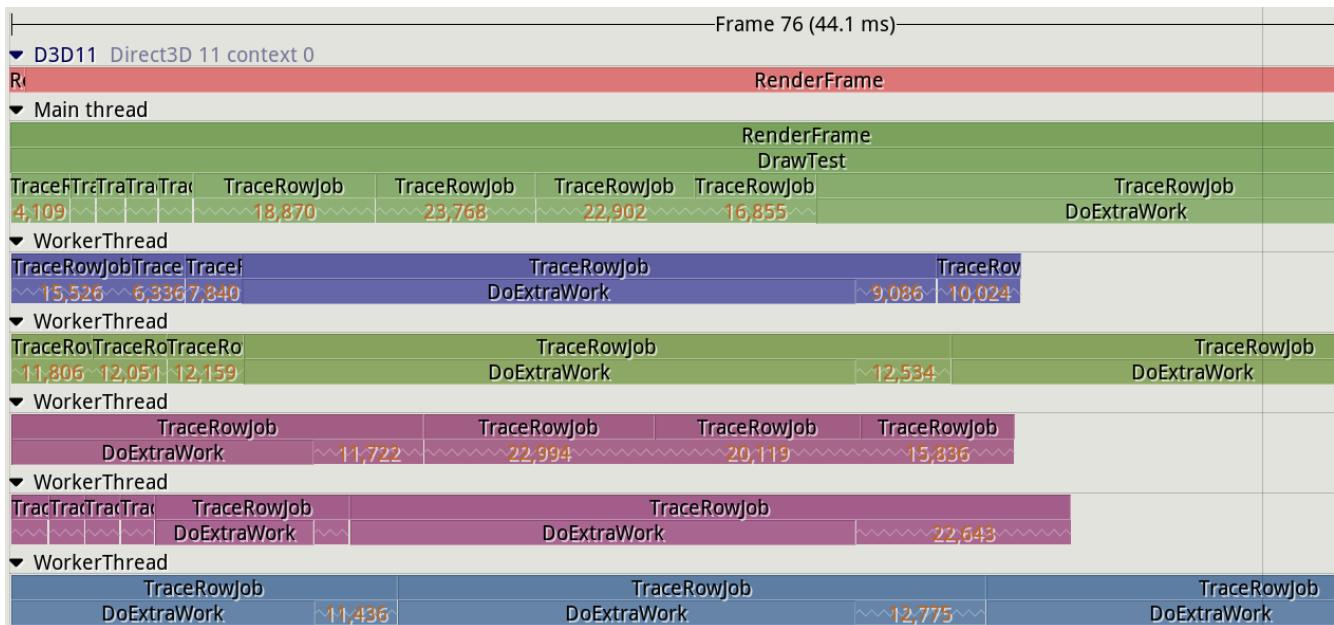


Figure 56: Tracy 主时间线视图。

时，会显示一个工具提示，显示给定帧的详细信息。在此示例中，我们显示了最后一个帧的详细信息，以绿色突出显示。



Figure 57: Tracy 帧时间。

Figure 58 展示了分析器的 CPU 数据部分。该区域显示给定线程正在哪个内核上执行，它还显示上下文切换。此部分还将显示其他正在 CPU 上运行的程序。如图像所示，将鼠标悬停在 CPU 数据视图中的给定部分上时，会显示给定线程的详细信息。详细信息包括线程运行所在的 CPU、父程序、单个线程和计时信息。我们可以看到 TestCpu.exe 线程在 CPU 1 上活动了 4.4 毫秒。

接下来是一个面板，提供有关程序花费时间的位置的信息，也称为热点。图 59 捕获了 Tracy 统计窗口的屏幕截图。我们可以检查记录的数据，包括给定函数处于活动状态的总时间、它被调用的次数等。还可以选择主视图中的时间范围，仅过滤与该时间间隔相对应的信息。

我们展示的最后一组面板允许我们更深入地分析单个区域实例。一旦您单击任何区域实例，例如，在主时间线视图或 CPU 数据视图上，Tracy 将打开一个区域信息窗口（参见图 60，左侧面板），其中包含此区域实例的详细信息。它告诉了区域本身或其子区域消耗了多少执行时间。在此示例中，TraceRowJob 函数的执行耗时 19.24 毫秒，但函数本身消耗的时间不包括其被调用者，仅为 1.36 毫秒，仅占 7%。其余时间由子区域占用。

很容易发现调用 DoExtraWork 占据了大部分时间，16.99 毫秒中的 19.24 毫秒。请注意，这个特定的 TraceRowJob 实例运行时间比平均情况长 4.4 倍（图像上找到“平均时间的 437.93%”）。Bingo! 我们发现了一个慢实例，其中 TraceRowJob 函数由于一些额外工作而变慢。一种方法是单击 DoExtraWork 行以检查此区域实例。这将使用 DoExtraWork 实例的详细信息更新区域信息视图，以便我们可以深入了解导致性能问题的原因。此视图还显示了区域启动的源文件和代码行。因此，另一个策略是检查源代码以了解为什么当前的 TraceRowJob 实例比平时花费更多

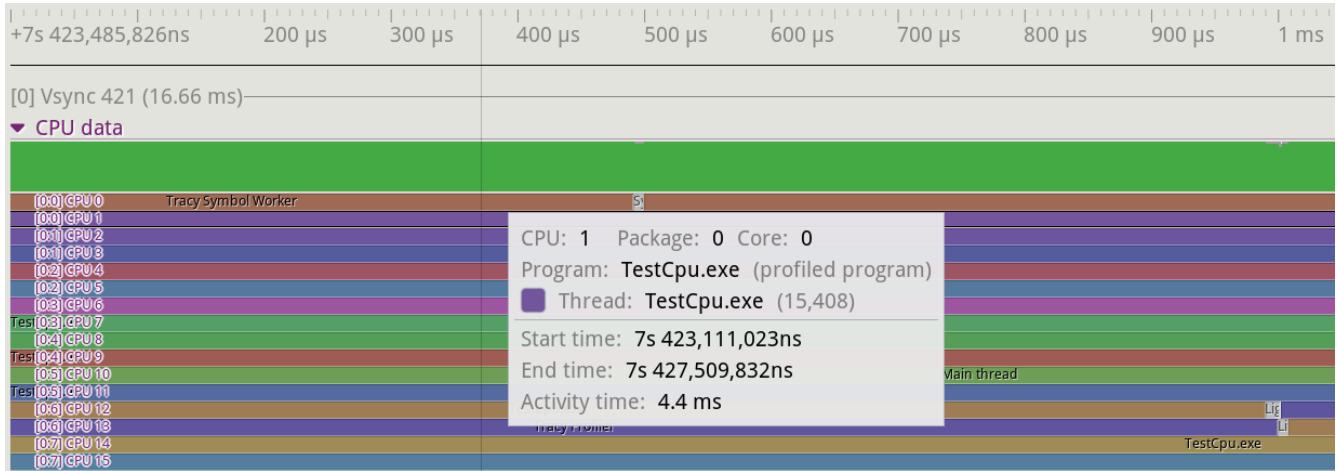


Figure 58: Tracy CPU 数据视图.

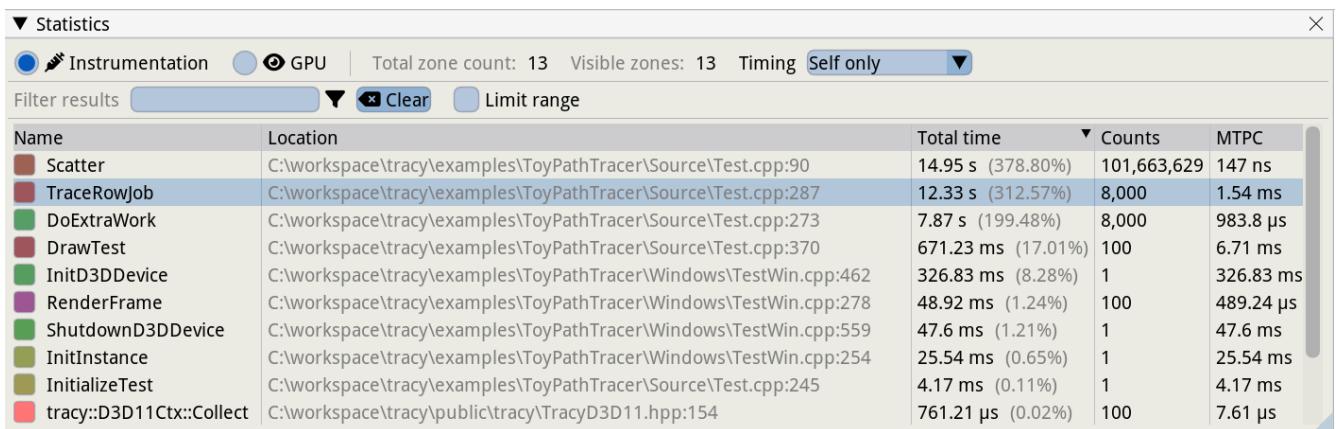


Figure 59: Tracy 函数统计数据.

时间。

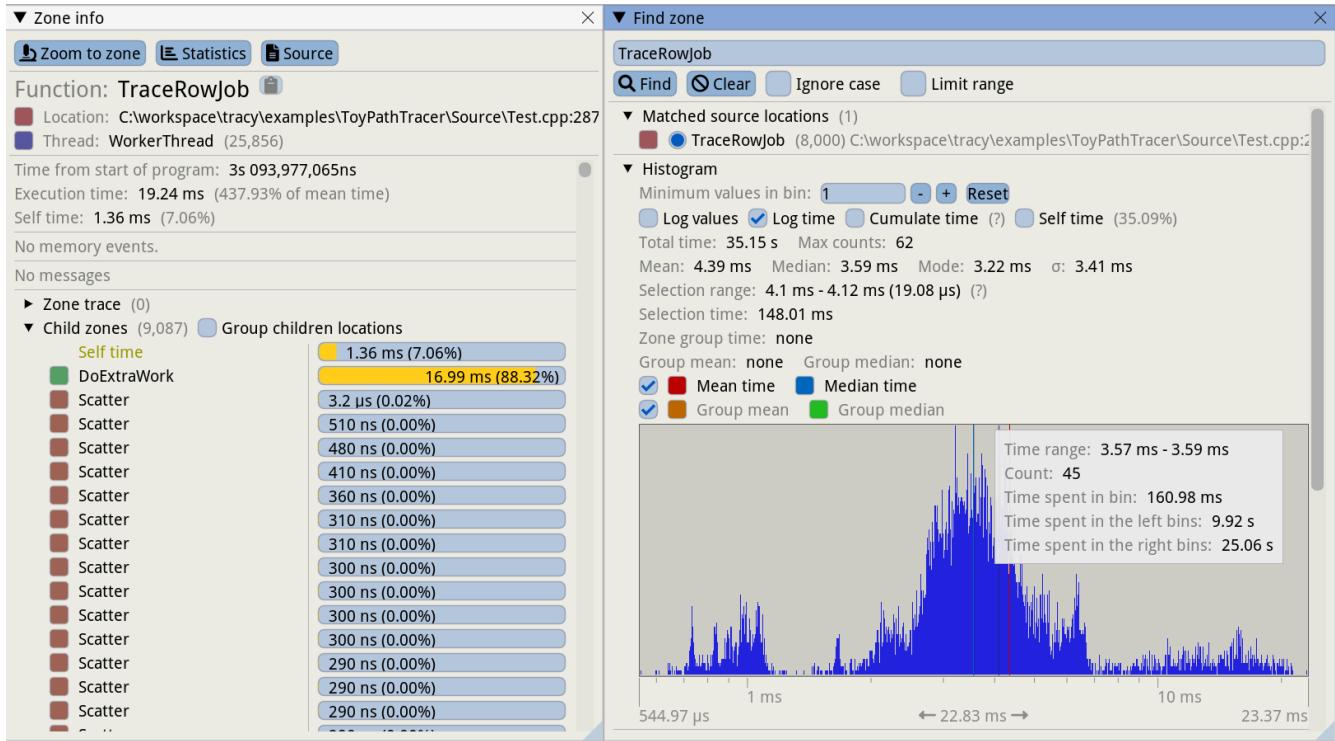


Figure 60: Tracy 区域详细信息窗口。

记得我们在图 57 上看到，还有其他慢帧。让我们看看这是否是所有慢帧的常见问题。如果我们点击“统计”按钮，它将显示“查找区域”面板（图 60，右侧）。在这里，我们可以看到聚合所有区域实例的时间直方图。这对于确定执行函数时有多少差异特别有用。查看右侧的直方图，我们看到 `TraceRowJob` 函数的中位数持续时间为 3.59 毫秒，大多数调用花费 1 到 7 毫秒之间。但是，有一些实例花费的时间超过 10 毫秒，峰值为 23 毫秒。请注意，时间轴是对数的。“查找区域”窗口还提供其他数据点，包括所检查区域的平均值、中位数和标准差。

现在我们可以检查其他慢实例，找到它们之间的共同点，这将帮助我们找到问题的根源。从这个视图中，您可以选择一个慢区域。这将使用该区域实例的详细信息更新区域信息窗口 (2)，然后单击“缩放到区域”按钮，主窗口将聚焦于此慢区域。从这里我们可以检查选定的 `TraceRowJob` 实例是否具有与我们刚刚分析的实例类似的特征。

## 6.9 Tracy 的其他功能

Tracy 不仅监控应用程序本身，还监控整个系统的性能。它还像传统采样分析器一样，报告与分析程序同时运行的应用程序的数据。该工具通过跟踪内核上下文切换（需要管理员权限）来监控线程迁移和空闲时间。区域统计数据（调用次数、时间、直方图）是准确的，因为 Tracy 会捕获每个区域的进入/退出，但系统级数据和源代码级数据是采样的。

在本节的示例中，我们使用了手动标记代码中感兴趣的区域。但这并不是开始使用 Tracy 的严格要求。您可以分析未修改的应用程序，稍后在需要时添加检测工具。Tracy 还提供许多其他功能，本文档无法全部涵盖。以下是一些值得注意的功能：

- 跟踪内存分配和锁。
- 会话比较。这对于确保更改提供了预期的益处至关重要。可以加载两个分析会话并比较更改前后区域数据。
- 源代码和汇编视图。如果可以使用调试符号，Tracy 还可以像 Intel Vtune 和其他分析器一样在源代码和相关汇编中显示热点。

与 Intel Vtune 和 AMD uProf 等其他工具相比，Tracy 无法获得相同级别的 CPU 微架构洞察力（例如，各种性能事件）。这是因为 Tracy 不利用特定于特定平台的硬件功能。

使用 Tracy 进行分析的开销取决于您激活的区域数量。Tracy 的作者提供了一些他在图像压缩程序上测量的数据点：使用两种不同的压缩方案，开销分别为 18% 和 34%。总共分析了 2 亿个区域，每个区域的平均开销为 2.25 纳秒。该测试检测了一个非常热的功能。在其他情况下，开销会低得多。虽然可以将开销保持在较低水平，但您需要谨慎选择要检测的代码部分，尤其是在决定将其用于生产环境时。

## 6.10 持续性能分析

在第 6 章中，我们介绍了进行性能分析的各种方法，包括但不限于代码注入、跟踪和采样。在这三种方法中，采样会带来相对较小的运行时开销，并且需要最少的预先工作，同时仍然可以提供宝贵的应用程序热点洞察。但这种洞察仅限于收集样本时的特定时间点 - 如果我们能为这种采样添加时间维度呢？与其只知道函数 A 在特定时间点消耗了 30% 的 CPU 周期，不如跟踪函数 A 在几天、几周甚至几个月内的 CPU 使用率变化？或者在同一时间段内检测其堆栈跟踪的变化，所有这些都在生产环境中进行？持续性能分析应运而生，将这些想法变成了现实。

### 什么是持续性能分析？

持续性能分析 (CP) 是一种始终处于开启状态的系统级、基于采样的性能分析器，但采样率较低，以尽量减少运行时影响。通过持续收集所有进程的数据，可以比较代码执行在不同时间段为什么会有不同，甚至可以在事件发生后进行调试。CP 工具提供了宝贵的见解，可以了解哪些代码使用了最多的资源，从而使工程师能够减少生产环境中的资源使用，从而节省成本。与 Linux perf 或 Intel VTune 等典型性能分析器不同，CP 可以从应用程序堆栈一直向下定位到内核堆栈，从任何指定的日期和时间找到性能问题，并支持比较任意两个日期/时间的调用堆栈，以突出性能差异。

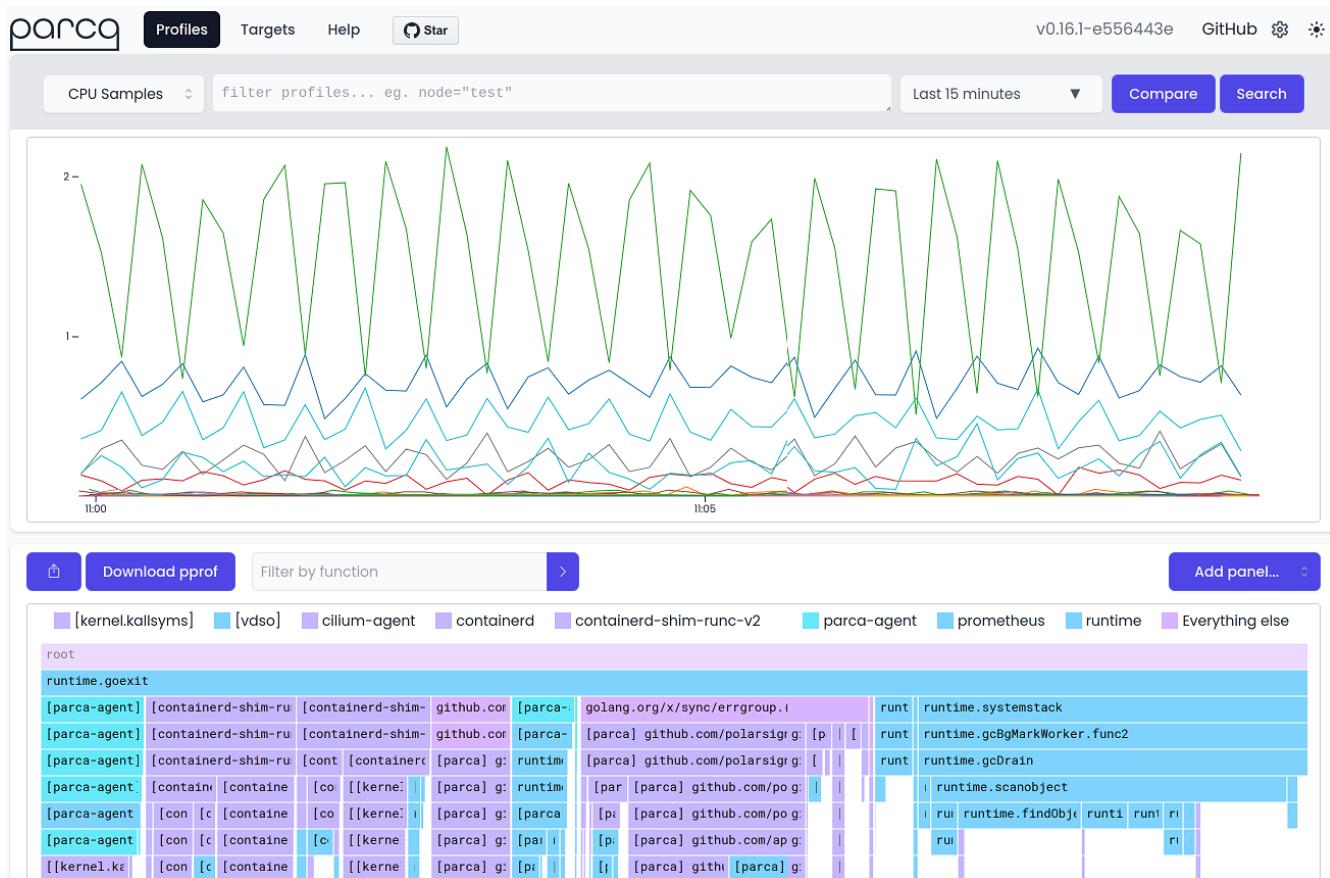


Figure 61: Parca 持续性能分析器 Web 界面截图。

为了展示典型 CP 的外观，让我们来看看开源 CP 之一 Parca: <https://github.com/parca-dev/parca><sup>130</sup> 的 Web 界面，如图 61 所示。顶部面板显示了一个时间序列图，其中包含在从时间窗口下拉列表中选择的期间（在本例中为“最近 15 分钟”）机器上各种进程收集的 CPU 样本数量，但是为了适应页面，图片被裁剪为仅显示最后 10 分钟。默认情况下，Parca 每秒收集 19 个样本。对于每个样本，它都会收集主机系统上所有进程的堆栈跟踪。分配给特定进程的样本越多，它在一段时间内的 CPU 活动就越多。在我们的例子中，您可以看到最繁忙的进程（顶部线条）表现出突发性行为，CPU 活动出现峰值和下降。如果您是此应用程序的主要开发人员，您可能会好奇为什么会发生这种情况。当您推出新版本的应用程序，突然看到分配给进程的 CPU 样本出现意外峰值，这表明出现了问题。

持续性能分析工具不仅可以更容易地发现性能变化点，还可以找到问题的根源。一旦您点击图表上的任何感兴趣点，工具就会在底部面板中显示与该周期相关的冰柱图。冰柱图是火焰图的倒置版本。使用它，您可以比较前后调用堆栈，并找到导致性能问题的确切原因。

想象一下，你将代码更改合并到了生产环境中，运行了一段时间后，收到关于间歇性响应时间峰值的报告 - 这些峰值可能与用户流量或一天中的任何特定时间相关，也可能不相关。这就是 CP 闪耀的地方。您可以拉起 CP Web UI 并搜索响应时间峰值日期和时间的堆栈跟踪，然后将它们与其他日期和时间的堆栈跟踪进行比较，以识别应用程序和/或内核堆栈级别的异常执行。这种类型的“视觉差异”直接在 UI 中支持，类似于图形“perf diff”或差异火焰图：<https://www.brendangregg.com/blog/2014-11-09/differential-flame-graphs.html>。

谷歌在 2010 年的论文“Google-Wide Profiling”中介绍了这个概念，倡导了在生产环境中始终启用性能分析的价值。然而，它花了近十年才在业界获得关注：

1. 2019 年 3 月，Google Cloud 发布了其持续性能分析器。
2. 2020 年 7 月，AWS 发布了 CodeGuru Profiler。
3. 2020 年 8 月，Datadog 发布了其持续性能分析器。
4. 2020 年 12 月，New Relic 收购了 Pixie 持续性能分析器。
5. 2021 年 1 月，Pyroscope 发布了其开源的持续性能分析器。
6. 2021 年 10 月，Elastic 收购了 Optimize 及其持续性能分析器 (Prodfiler)；Polar Signals 发布了其开源的 Parca 持续性能分析器。
7. 2021 年 12 月，Splunk 发布了其 AlwaysOn Profiler。
8. 2022 年 3 月，英特尔收购了 Granulate 及其持续性能分析器 (gProfiler)。

该领域的新的参与者继续出现，既有开源版本也有商业版本。其中一些产品比其他产品需要更多的手动操作。例如，有些产品需要对源代码或配置文件进行更改才能开始性能分析。其他产品则针对不同的语言运行时（例如 ruby、python、golang、C/C++/Rust）需要不同的代理。最好的产品围绕 eBPF 制作了秘密武器，因此只需要安装运行时代理就可以了。

它们还支持的语言运行时数量、获取可读堆栈跟踪所需的调试符号的工作量以及除了 CPU 之外可以分析的系统资源类型（例如内存、I/O、锁定等）方面有所不同。虽然持续性能分析器在上述方面存在差异，但它们都具有为各种语言运行时提供低开销、基于采样的性能分析的共同功能，以及用于基于 Web 的搜索和查询功能的远程堆栈跟踪存储。

持续性能分析器将走向何方？Optimyze 公司的联合创始人 Thomas Dullien 开发了创新的持续性能分析器 Prodfiler，他在 QCon London 2023 上发表了主题演讲，表达了他希望拥有一款集群级工具，可以回答“为什么这个请求慢？”或“为什么这个请求昂贵？”等问题。在一个多线程应用程序中，一个特定的函数可能在性能分析中显示为 CPU 和内存消耗最高的函数，但它的职责可能完全不在应用程序关键路径上，例如，一个维护线程。与此同时，另一个函数的 CPU 执行时间非常短，在性能分析中几乎没有记录，但却可能对整体应用程序延迟和/或吞吐量产生过大的影响。典型的性能分析器无法解决这个缺点。由于持续性能分析器基本上是始终运行的性能分析器，它们也继承了同样的盲点。

<sup>130</sup> Parca - <https://github.com/parca-dev/parca>

值得庆幸的是，新一代的持续性能分析器已经出现，它们利用人工智能和 LLM 启发架构来处理性能分析样本，分析函数之间关系，最终高精度地找出直接影响整体吞吐量和延迟的函数和库。Raven.io 就是今天提供这种功能的一家公司。随着该领域的竞争加剧，创新能力将继续增长，使持续性能分析工具变得像典型性能分析器一样强大和稳健。

## 问题和练习

1. 你会使用哪些工具？
  - 场景 1：客户支持团队报告了一个客户问题：升级到新版本应用程序后，特定操作的性能下降了 10%。
  - 场景 2：客户支持团队报告了一个客户问题：一些交易完成的时间比平时长 2 倍，没有特定模式。
  - 场景 3：你正在评估三种不同的压缩算法，想知道每个算法的性能瓶颈是什么？
  - 场景 4：有一个新的闪亮库，声称比你目前项目中集成的库更快；你决定比较它们的性能。
2. 运行你每天都在使用的应用程序。根据你想要做的改进，最适合分析应用程序性能的工具是什么？练习使用这个工具。
3. 假设你在一台机器上运行多个相同的程序副本，每个副本的输入都不同。只需要分析其中一个副本，还是需要分析整个系统？

## 章节总结

- 我们快速概述了三大平台（Linux、Windows 和 MacOS）上最流行的工具。根据 CPU 供应商的不同，性能分析工具的选择也会有所不同。对于使用英特尔处理器的系统，我们推荐使用 Vtune；对于使用 AMD 处理器的系统，我们推荐使用 uProf；对于苹果平台，我们推荐使用 Xcode Instruments。
- Linux perf 可能在 Linux 上使用最频繁的性能分析工具。它支持所有主要 CPU 供应商的处理器。但是它没有图形界面，不过有一些免费工具可以可视化 perf 的分析数据。
- 我们还讨论了 Windows 事件跟踪 (ETW)，它旨在观察运行系统中的软件动态。Linux 有一个类似的工具叫做 KUtrace: <https://github.com/dicksites/KUtrace>,<sup>131</sup> 我们这里不进行介绍。
- 此外，还有混合性能分析器，它结合了代码植入、采样和跟踪等技术。这结合了这些方法的优点，允许用户获得特定代码段非常详细的信息。本章中，我们介绍了 Tracy，它在游戏开发人员中非常流行。
- 连续性能分析器已经成为监控生产环境性能的重要工具。它们可以收集系统级的性能指标，包括调用堆栈，持续时间可达数天、数周甚至数月。这些工具可以更容易地发现性能变化的点和问题的根源。

<sup>131</sup> KUtrace - <https://github.com/dicksites/KUtrace>

## 第二部分：源代码优化

欢迎来到本书的第二部分，我们将讨论各种低级源代码优化技术，也称为 调优。在第一部分，我们学习了如何找到代码中的性能瓶颈，这只是开发人员工作的一半。另一半是解决问题。

现代 CPU 是一个非常复杂的设备，几乎不可能预测某些代码片段的运行速度。软件和硬件性能取决于许多因素，移动部件的数量太多，超出人类思维的范围。幸运的是，通过本书第一部分讨论的所有性能监控功能，我们可以观察代码从 CPU 的角度如何运行。我们将广泛利用本书前面学到的方法和工具来指导我们的性能工程过程。

在非常高的层面上，软件优化可以分为五个类别。

- **算法优化。** 理念：分析程序中使用的算法和数据结构，看看是否能找到更好的。示例：使用快速排序代替冒泡排序。
- **并行化计算。** 理念：如果一个算法高度可并行化，使程序多线程化，或者考虑在 GPU 上运行。目标是同时做多件事。并发性已经在硬件和软件堆栈的所有层中使用。示例：将工作分布到多个线程上，平衡数据中心多个服务器之间的负载，使用异步 IO 以避免在等待 IO 操作时阻塞，保持多个并发网络连接以重叠请求延迟。
- **消除冗余工作。** 理念：不要做你不需要的或已经完成的工作。示例：利用更多 RAM 来减少 CPU 和 IO 的使用量（缓存、记忆化、查找表、压缩），预算算已知编译时值，将循环不变计算移出循环，传引用 C++ 对象以避免传值引起的过度复制。
- **批量处理。** 理念：聚合多个类似的操作并一次性执行，从而减少重复操作的开销。示例：发送较大的 TCP 数据包而不是许多小的数据包，分配大块内存而不是为数百个微小对象分配空间。
- **排序。** 理念：重新排序算法中的操作序列。示例：更改数据布局以启用顺序内存访问，根据 C++ 多态对象的类型对数组进行排序以更好地预测虚函数调用，将热门函数分组并将其放置在二进制文件中更近的位置。

本书讨论的许多优化属于多个类别。例如，我们可以说矢量化是并行化和批量处理的结合；循环阻塞（tiling）是批量处理和消除冗余工作的体现。

为了使图片完整，让我们也列出其他一些也许明显但仍然相当合理的加速方法：

- **使用另一种语言重写代码：** 如果程序是用解释性语言（python、javascript 等）编写的，将其性能关键部分重写成开销更少的语言，例如 C++、Rust、Go 等。
- **调整编译器选项：** 检查您是否至少使用了以下三个编译器标志：-O3（启用与机器无关的优化）、-march（启用针对特定 CPU 架构的优化）、-floop（启用过程间优化）。但不要就此止步，还有许多其他选项会影响性能。我们将在以后的章节中研究一些。可以考虑挖掘最佳选项集应用程序，可用的商业产品可以自动完成此过程。
- **优化第三方软件包：** 绝大多数软件项目利用专有和开源代码层。这包括操作系统、库和框架。您还可以通过替换、修改或重新配置其中之一来寻求改进。
- **购买更快的硬件：** 显然，这是一个与成本相关的业务决策，但有时它是其他选项都已用尽时唯一提高性能的方法。当您识别出应用程序中的性能瓶颈并清楚地传达给上层管理人员时，购买硬件更容易获得批准。例如，一旦您发现内存带宽限制了您的多线程程序的性能，您可能会建议购买具有更多内存通道和 DIMM 插槽的服务器主板和处理器。

### 算法优化

标准算法和数据结构并不总是适用于性能关键型工作负载。例如，链接列表基本上已被淘汰，取而代之的是“扁平”数据结构。传统上，链接列表的每个新节点都是动态分配的。除了可能调用许多昂贵的内存分配之外，这可能导致列表的所有元素都分散在内存中。遍历这样的数据结构不利于缓存。尽管算法复杂度仍然是  $O(N)$ ，但实际上，其运行时间会比普通数组差得多。一些数据结构，比如二叉树，有自然链表式的表示，所以用指针追逐的方式实现它们可能很诱人。然而，存在更高效的“扁平”版本，参见 `boost::flat_map`、`boost::flat_set`。

在选择算法时，您可能会快速选择最流行的选项然后继续… 即使它可能不是您特定情况下的最佳选择。例如，您需要在已排序数组中找到一个元素。大多数开发人员考虑的第一个选项是二分搜索，对吧？它非常知名，并且在算法复杂度方面是最佳的， $O(\log N)$ 。如果我告诉你数组保存 32 位整数，并且数组的大小通常很小（小于 20 个元素），你会改变你的决定吗？最终，测量应该指导您的决策，但二分搜索会受到分支预测错误的影响，因为每个元素值的测试都有 50% 的可能性为真。这就是为什么在小型数组上，即使线性扫描具有更差的算法复杂度，它通常也更快的原因。

## 数据驱动优化

性能调优最重要的技术之一称为“数据驱动”优化，它基于程序处理的数据进行内省。该方法侧重于数据的布局及其在整个程序中的转换。这种方法的一个经典例子是数组结构到结构数组的转换，如 Listing 130 所示。

代码清单：SOA 到 AOS 转换。

```
struct S {
    int a;
    int b;
    int c;
    // other fields
};

S s[N];      // AOS

<=>

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

数据驱动开发 (DDD) 的主要思想是研究程序如何访问数据（数据在内存中的布局，访问模式），然后相应地修改程序（改变数据布局，改变访问模式）。

实际上，我们可以说所有的优化在某种程度上都是数据驱动的。即使是我们将在下一节看到的转换，也是基于我们从程序执行中获得的一些反馈：函数调用次数，分支是否被采取，性能计数器等。

DDD 的另一个广泛示例是“小尺寸优化”。它的想法是静态预分配一定量的内存来避免动态内存分配。对于元素上限可以很好预测的中小型容器来说，它尤其有用。现代 C++ STL 的 `std::string` 实现将前 15-20 个字符保存在堆栈上分配的缓冲区中，只有更长的字符串才会在堆上分配内存。LLVM 的 `SmallVector` 和 Boost 的 `static_vector` 也是这种方法的其他例子。

## 低级优化

性能工程是一种艺术。就像任何艺术一样，可能的情况是无穷无尽的。不可能涵盖所有可以想象的优化。接下来的几个章节主要讨论针对现代 CPU 架构的优化。

在我们跳入特定的源代码优化技术之前，需要做一些注意事项。首先，避免优化糟糕的代码。如果一段代码存在高级性能低下问题，你不应该对其应用机器特定的优化。始终先关注修复主要问题。只有当你确定算法和数据结构针对你要解决的问题是最佳的，然后尝试应用低级改进。

其次，请记住，您实施的优化可能并非在所有平台上都受益。例如，循环阻塞取决于系统内存层次结构的特征，尤其是 L2 和 L3 缓存的大小。因此，针对具有特定 L2 和 L3 缓存大小的 CPU 调整的算法可能无法很好地适用于缓存更小的 CPU。在您的应用程序将运行的平台上测试更改很重要。

接下来的四章按照 TMA 分类进行组织（参见 Section ??）：

- 第 8 章. 优化内存访问 - TMA:MemoryBound 类别
- 第 9 章. 优化计算 - TMA:CoreBound 类别
- 第 10 章. 优化分支预测 - TMA:BadSpeculation 类别
- 第 11 章. 机器代码布局优化 - TMA:FrontEndBound 类别

此分类背后的思想是为使用 TMA 方法进行性能工程工作的开发人员提供一个清单。每当 TMA 将性能瓶颈归因于上述类别之一时，您可以随时参考相应的章节以了解您的选项。

第 14 章涵盖了不属于上述任何类别的其他优化领域。第 15 章解决了一些优化多线程应用程序中常见的难题。

## 7 优化内存访问

现代计算机仍然基于经典的冯·诺伊曼体系结构构建，其中包括 CPU、内存和输入/输出单元。内存操作（加载和存储）占据了性能瓶颈和功耗的最大部分。毫无疑问，我们首先从这个类别开始。

关于内存层次结构性能非常重要的说法得到了图 62 的支持。它显示了内存和处理器之间性能差距的增长。垂直轴是对数刻度，显示了 CPU-DRAM 性能差距的增长。内存基线是来自 1980 年的 64 KB DRAM 芯片的内存访问延迟。典型的 DRAM 性能改进为每年 7%，而 CPU 每年享受 20-50% 的改进。[Hennessy & Patterson, 2017]

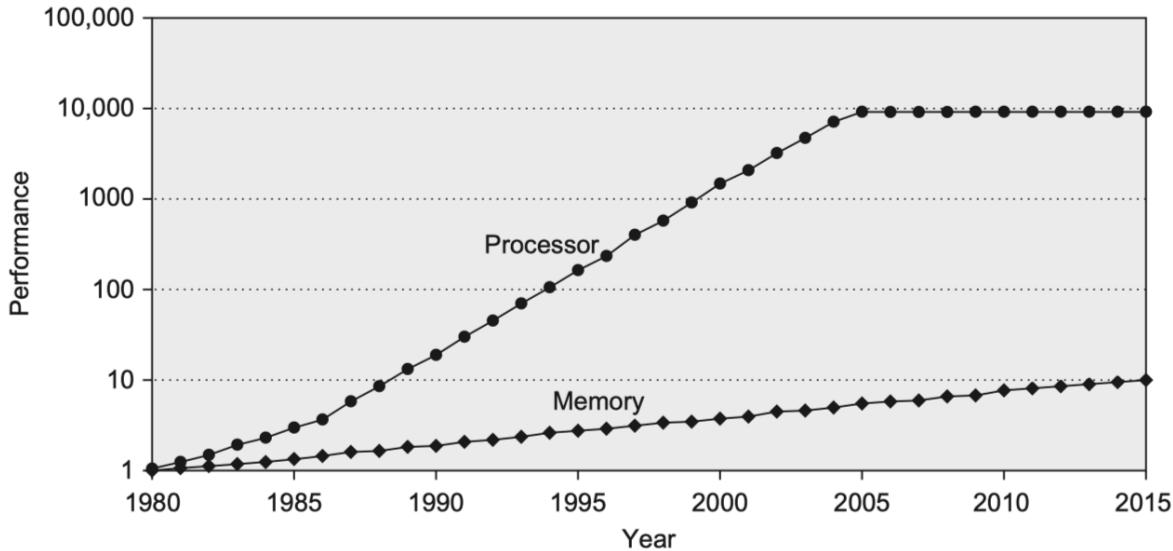


Figure 62: 内存和处理器之间性能差距。© 图片来自 [Hennessy & Patterson, 2017]。

确实，一个变量可以在最小的 L1 缓存中在几个时钟周期内获取，但如果不在 CPU 缓存中，则从 DRAM 获取该变量可能需要超过三百个时钟周期。从 CPU 的角度来看，最后一级缓存未命中感觉就像是一个非常长的时间，特别是如果处理器在此期间没有执行任何有用的工作。当系统高度加载线程，并且没有可用的内存带宽及时满足所有加载和存储时，执行线程也可能会饿死。

当应用程序执行大量内存访问并花费大量时间等待它们完成时，这样的应用程序被描述为受内存限制。这意味着为了进一步提高其性能，我们可能需要改进如何访问内存，减少此类访问的数量或升级内存子系统本身。

在 TMA 方法论中，Memory Bound 估算了由于对加载或存储指令的需求而导致 CPU 管道可能停滞的插槽的比例。解决这样的性能问题的第一步是找到导致高 Memory Bound 指标的内存访问（参见 Section 5.10.1）。一旦确定了有问题的内存访问，就可以应用几种优化策略。下面我们将讨论几种典型情况。

### 7.1 缓存友好数据结构

编写缓存友好的算法和数据结构是打造高性能应用程序的重要因素之一。缓存友好代码的关键支柱是我们之前在 Section 3.6 描述的局部性原理，包括时间局部性和空间局部性。这里的目标是允许高效地从缓存中获取所需数据。在设计缓存友好代码时，考虑缓存行而不是单个变量及其在内存中的位置会很有帮助。

### 7.1.1 顺序访问数据

利用缓存空间局部性的最佳方式是进行顺序内存访问。通过这样做，我们可以让硬件预取器（参见 Section ??）识别内存访问模式并提前引入下一块数据。Listing 132 中展示了 C 代码示例，它执行了此类缓存友好访问。该代码之所以“缓存友好”，是因为它按照矩阵在内存中的布局顺序访问矩阵元素（行优先遍历: [https://en.wikipedia.org/wiki/Row-and\\_column-major\\_order<sup>132</sup>](https://en.wikipedia.org/wiki/Row-and_column-major_order)）。交换数组中索引的顺序（即 `matrix[column][row]`）将导致以列为主的矩阵遍历，这不会利用空间局部性并会损害性能。

代码清单: 缓存友好的内存访问。

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

Listing 132 中展示的例子是一个经典例子，但现实世界中的应用程序通常要复杂得多。有时您需要付出更多努力才能编写缓存友好的代码。例如，在已排序的大数组中进行二分搜索的标准实现并没有利用空间局部性，因为它测试的是彼此相距甚远、不共享同一缓存行的不同位置的元素。解决这个问题最著名的方式是使用 Eytzinger 布局存储数组元素 [Khuong & Morin, 2015]。它的想法是使用类似 BFS 的布局（通常在二进制堆中看到）将一个隐式的二叉搜索树打包到数组中。如果代码在数组中执行大量二分搜索，将其转换为 Eytzinger 布局可能会有益处。

### 7.1.2 使用合适的容器

几乎任何语言都提供各种现成的容器。但了解它们的底层存储和性能影响很重要。[Fog, 2004, Chapter 9.7 Data structures, and container classes] 提供了一个很好的关于选择合适的 C++ 容器的逐步指南。

此外，选择数据存储时要考虑代码将如何使用它。考虑一种情况，需要在数组中存储对象与存储指向这些对象的指针之间进行选择，而对象的大小较大。指针数组占用更少的内存。这将使修改数组的操作受益，因为指针数组需要更少的内存传输。然而，当保留对象本身时，通过数组进行线性扫描会更快，因为它更符合缓存，不需要间接内存访问。<sup>133</sup>

### 7.1.3 数据压缩

通过使数据更紧凑可以改善内存层次结构的利用率。有许多方法可以压缩数据。经典示例之一是使用位字段。Listing 7.1.3 展示了在数据打包时代码可能获益的例子。如果我们知道 `a`, `b` 和 `c` 代表需要一定数量位才能编码的枚举值，我们就可以减少结构体 `S` 的存储空间（参见 Listing 7.1.3）。

代码清单: 打包数据: 基线结构体。

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

代码清单: 打包数据: 打包的结构体。

```
struct S {
    unsigned a:4;
    unsigned b:2;
```

<sup>132</sup> 行优先和列优先顺序 - [https://en.wikipedia.org/wiki/Row-and\\_column-major\\_order](https://en.wikipedia.org/wiki/Row-and_column-major_order)。

<sup>133</sup> 博客文章“对象向量 vs 指针向量”作者 B. Filipek - <https://www.bfilipek.com/2014/05/vector-of-objects-vs-vector-of-pointers.html>。

```
    unsigned c:2;
}; // S is only 1 byte
```

这大大减少了来回传输的内存量并节省了缓存空间。请记住，这会带来访问每个打包元素的成本。由于 b 的位与 a 和 c 共享同一个机器字，编译器需要执行 `>>` (右移) 和 `&` (AND) 操作来加载它。类似地，需要 `<<` (左移) 和 `|` (OR) 操作将值存储回去。在额外计算比低效内存传输引起的延迟更便宜的地方，数据打包是有益的。

此外，当避免编译器添加的填充（参见 Listing 7.1.3 中的示例）时，程序员可以通过重新排列结构或类中的字段来减少内存使用。编译器插入未使用的内存字节（填充）的原因是为了允许高效地存储和获取结构的单个成员。在该示例中，如果将 s1 的成员按其大小递减的顺序声明，则可以减小其大小。

代码清单：避免编译填充。

```
struct S1 {
    bool b;
    int i;
    short s;
}; // S1 is `sizeof(int) * 3` bytes

struct S2 {
    int i;
    short s;
    bool b;
}; // S2 is `sizeof(int) * 2` bytes
```

#### 7.1.4 对齐和填充

改善内存子系统利用率的另一个技术是对齐数据。可能出现这种情况，即一个大小为 16 字节的对象占用两个缓存行，即它从一个缓存行开始并结束于下一个缓存行。获取这样一个对象需要两个缓存行读取，如果对象正确对齐，可以避免这种情况。Listing 7.1.4 展示了如何使用 C++11 的 `alignas` 关键字对齐内存对象。

代码清单：使用“`alignas`”关键字对齐数据。

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

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

如果变量存储在可被其自身大小整除的内存地址上，则访问它的效率最高。例如，一个 `double` 类型变量占用 8 个字节的存储空间，因此最好将其存储在一个可被 8 整除的地址上。这个大小通常是 2 的幂次方。大于 16 个字节的对象应该存储在一个可被 16 整除的地址上。[\[Fog, 2004\]](#)

对齐可能会导致未使用字节的空洞，从而降低内存带宽利用率。在上面的例子中，如果结构体 S 只有 40 个字节，那么下一个 S 对象将会从下一个缓存行的开头开始，这会在每个保存 S 结构体的缓存行中留下  $64 - 40 = 24$  个未使用的字节。

有时需要填充数据结构成员以避免一些极端情况，例如缓存争用 [Fog, 2004, 第 9.10 章 缓存争用] 和伪共享（参见 Section 11.8）。例如，在多线程应用程序中，当两个线程 A 和 B 访问同一结构的不同字段时，可能会出现伪共享问题。Listing 7.1.4 展示了可能发生这种情况的代码示例。由于结构体 S 的成员 a 和 b 可能占据同一个缓存行，因此缓存一致性问题可能会显著减慢程序运行速度。为了解决这个问题，可以填充 S 使得成员 a 和 b 不共享同一个缓存行，如 Listing ?? 所示。

代码清单: 填充数据: 基线版本。

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

代码清单: 填充数据: 改进版本。~~~~~ {#lst:PadFalseSharing2 .cpp} #define CACHELINE\_ALIGN alignas(64) struct S { int a; // written by thread A CACHELINE\_ALIGN int b; // written by thread B };~~~~~

使用 `malloc` 进行动态分配时，保证返回的内存地址满足目标平台的最小对齐要求。一些应用程序可能受益于更严格的对齐。例如，以 64 字节对齐而不是默认的 16 字节对齐动态分配 16 字节。POSIX 系统的用户可以利用 `memalign`: <https://linux.die.net/man/3/memalign><sup>134</sup> API 来实现这一目的。其他人可以像这里: <https://embeddedartistry.com/blog/2017/02/22/generating-aligned-memory/><sup>135</sup> 所描述的那样自己实现。

对齐考虑最重要的领域之一是 SIMD 代码。当依赖于编译器自动矢量化时，开发人员无需做任何特殊操作。但是，当您使用编译器向量内联函数编写代码时（参见 Section 8.5），它们通常要求地址可被 16、32 或 64 整除。编译器内联头文件中提供的向量类型已经做了注释，以确保适当的对齐。[Fog, 2004]

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

### 7.1.5 动态内存分配

首先，有很多可以替代 `malloc` 的工具，它们更快、更可扩展，并且更好地解决了碎片化<sup>136</sup>问题。仅仅通过使用非标准内存分配器，你就可以获得 2% 左右的性能提升。动态内存分配的一个典型问题是，在启动时，线程会争相尝试同时分配内存区域。<sup>137</sup> 最受欢迎的内存分配库之一是 jemalloc: <http://jemalloc.net/><sup>138</sup> 和 tcmalloc: <https://github.com/google/tcmalloc><sup>139</sup>。

其次，可以使用自定义分配器来加速分配，例如 arena 分配器: [https://en.wikipedia.org/wiki/Region-based\\_memory\\_management](https://en.wikipedia.org/wiki/Region-based_memory_management)<sup>140</sup>。它们的主要优势之一是开销低，因为此类分配器不会为每次内存分配执行系统调用。另一个优点是它的高灵活性。开发人员可以根据操作系统提供的内存区域实现自己的分配策略。一个简单的方法是维护两个不同的分配器，每个分配器都有自己的 arena（内存区域）：一个用于热门数据，另一个用于冷数据。将热门数据放在一起可以使其共享缓存行，从而提高内存带宽利用率和空间局部性。它也改善了 TLB 利用率，因为热门数据占用的内存页面更少。此外，自定义内存分配器可以使用线程局部存储来实现每个线程的分配，并摆脱线程之间的任何同步。当应用程序基于线程池并且不会产生大量线程时，这变得很有用。

<sup>134</sup> Linux 手册页面，用于 `memalign` - <https://linux.die.net/man/3/memalign>。

<sup>135</sup> 生成对齐内存- <https://embeddedartistry.com/blog/2017/02/22/generating-aligned-memory/>。典型的 `malloc` 实现涉及同步，以防止多个线程试图动态分配内存。

<sup>136</sup> 碎片化- [https://en.wikipedia.org/wiki/Fragmentation\\_\(computing\)](https://en.wikipedia.org/wiki/Fragmentation_(computing))。

<sup>137</sup> 相同的情况也适用于内存释放。

<sup>138</sup> jemalloc - [<http://jemalloc.net/>] (<http://jemalloc.net/>)。

<sup>139</sup> tcmalloc - <https://github.com/google/tcmalloc>

<sup>140</sup> 基于区域的内存管理- [https://en.wikipedia.org/wiki/Region-based\\_memory\\_management](https://en.wikipedia.org/wiki/Region-based_memory_management)

### 7.1.6 调整代码以适应内存层次结构

某些应用程序的性能取决于特定级别的缓存大小。这里最著名的例子是用 [循环阻塞](https://en.wikipedia.org/wiki/Loop_nest_optimization)（平铺）改进矩阵乘法。这个想法是将矩阵的工作大小分解成更小的块（tiles），使每个块都适合 L2 缓存。<sup>141</sup> 大多数架构提供类似 CPUID 的指令，<sup>142</sup> 它允许我们查询缓存的大小。或者，可以使用 [缓存无关算法](https://en.wikipedia.org/wiki/Cache-oblivious_algorithm)<sup>143</sup>，其目标是针对任何大小的缓存都能合理地工作。

英特尔 CPU 具有一个数据线性地址硬件功能（见 Section 5.15），支持缓存阻塞，如 easyperf 博客文章所述 <https://easyperf.net/blog/2019/12/17/Detecting-false-sharing-using-perf#2-tune-the-code-for-better-utilization-of-cache-hierarchy><sup>144</sup>。

## 7.2 显式内存预取

到现在为止，您应该已经知道未能在缓存中解析的内存访问通常代价高昂。现代 CPU 非常努力地降低预取请求提前足够发出时的缓存未命中惩罚。如果请求的内存位置不在缓存中，我们将无论如何遭受缓存未命中，因为我们必须访问 DRAM 并提取数据。但是，如果我们在程序需要数据时将该内存位置引入缓存，那么我们实际上可以将缓存未命中惩罚降为零。

现代 CPU 有两种解决这个问题的机制：硬件预取和 OOO 执行。硬件预取器通过对重复内存访问模式发起预取请求来帮助隐藏内存访问延迟。而 OOO 引擎则向前看 N 条指令，并提前发出加载指令，以允许平滑执行未来将需要此数据的指令。

当数据访问模式太复杂无法预测时，硬件预取器就会失败。软件开发人员对此无能为力，因为我们无法控制该单元的行为。另一方面，OOO 引擎不像硬件预取器那样试图预测未来需要的内存位置。因此，它成功的唯一衡量标准是它通过提前调度加载隐藏了多少延迟。

考虑 Listing 7.2 中的一段代码片段，其中 arr 是一个包含一百万个整数的数组。索引 idx 被赋予一个随机值，然后立即用于访问 arr 中的一个位置，该位置几乎肯定会错过缓存，因为它是随机的。硬件预取器不可能预测，因为每次加载都进入内存中一个完全新的位置。从知道内存位置的地址（从函数 random\_distribution 返回）到需要该内存位置的值（调用 doSomeExtensiveComputation）的时间间隔称为 预取窗口。在这个例子中，由于预取窗口非常小，OOO 引擎没有机会提前发出加载指令。这导致内存访问 arr[idx] 的延迟成为执行循环时的关键路径，如图 63 所示。可以看到，程序在没有取得值的情况下等待（阴影填充矩形），无法向前推进。

代码清单：随机数为后续加载提供数据。

```
for (int i = 0; i < N; ++i) {
    size_t idx = random_distribution(generator);
    int x = arr[idx]; // cache miss
    doSomeExtensiveComputation(x);
}
```

这里还有另一个重要观察。当 CPU 接近完成第一次迭代时，它会推测性地开始执行来自第二次迭代的指令。这在迭代之间创建了一个积极的执行重叠。然而，即使在现代处理器中，也缺少足够的 OOO 功能，无法完全将缓存未命中延迟与来自迭代 1 的 doSomeExtensiveComputation 的执行重叠。换句话说，在我们的例子中，CPU 无法提前查看当前执行，以便足够早地发出加载指令。

幸运的是，这并不是死路一条，因为有一种方法可以加速这段代码。为了隐藏缓存未命中延迟，我们需要将其与 doSomeExtensiveComputation 的执行重叠。如果我们管道化随机数生成并在下一次迭代中开始预取内存位置，就

<sup>141</sup> 通常情况下，人们会针对 L2 缓存的大小进行调整，因为它是内核之间不共享的。

<sup>142</sup> 英特尔处理器的 CPUID 指令在 [Intel, 2023b, Volume 2] 中描述。

<sup>143</sup> 缓存无关算法：[https://en.wikipedia.org/wiki/Cache-oblivious\\_algorithm](https://en.wikipedia.org/wiki/Cache-oblivious_algorithm)。

<sup>144</sup> 博客文章“检测伪共享” - <https://easyperf.net/blog/2019/12/17/Detecting-false-sharing-using-perf#2-tune-the-code-for-better-utilization-of-cache-hierarchy>。

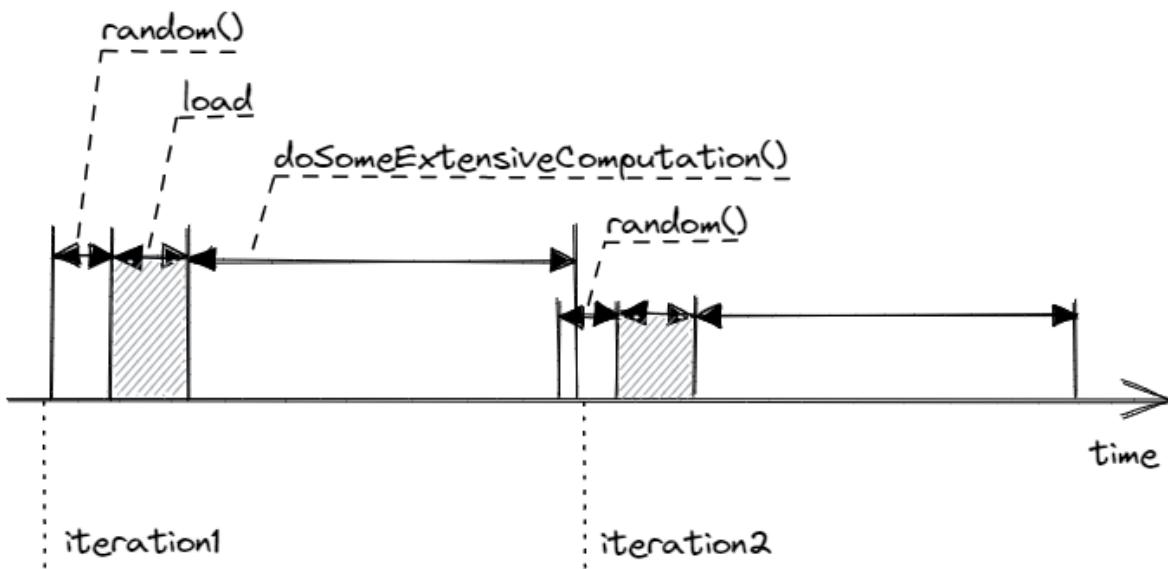


Figure 63: 显示关键路径上的负载延迟的执行时间线.

可以实现这一点，如 Listing 145 所示。请注意使用 `__builtin_prefetch`: <https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html>,<sup>145</sup> 开发人员可以使用的特殊提示，明确请求 CPU 预取特定内存位置。图 64 展示了这种转换的图形说明。

代码清单: 利用明确的软件内存预取提示。

```
size_t idx = random_distribution(generator);
for (int i = 0; i < N; ++i) {
    int x = arr[idx];
    idx = random_distribution(generator);
    // prefetch the element for the next iteration
    __builtin_prefetch(&arr[idx]);
    doSomeExtensiveComputation(x);
}
```

在 x86 平台上利用显式软件预取的另一种选择是使用编译器内部函数 `_mm_prefetch`。有关更多详细信息，请参见 Intel 内部函数指南。无论如何，编译器都会将其编译成机器指令：x86 的 PREFETCH 和 ARM 的 pld。对于某些平台，编译器可能会跳过插入指令，因此检查生成的机器代码是一个好主意。

存在一些情况下，软件内存预取是不可能的。例如，当遍历链表时，预取窗口非常小，无法隐藏指针追逐的延迟。

在 Listing 145 中，我们看到了针对下一次迭代进行预取的示例，但您也经常会遇到需要为 2、4、8 甚至更多次迭代进行预取的情况。Listing 146 中的代码就是这种情况之一，当图非常稀疏并且有很多顶点时，访问 `this->out_neighbors` 和 `this->in_neighbors` 向量很可能会错过缓存。

这段代码与前面的例子不同，因为每个迭代中都没有大量计算，所以缓存未命中的惩罚很可能主导了每个迭代的延迟。但是我们可以利用我们知道未来将访问的所有元素这一事实。向量 `edges` 的元素被顺序访问，因此很可能由硬件预取器及时地引入 L1 缓存。我们在这里的目标是将缓存未命中延迟与执行足够多的迭代重叠，以完全隐藏它。

一般来说，为了让预取提示有效，它们必须提前插入，以便在加载的值用于其他计算时，它已经存在于缓存中。但

<sup>145</sup> GCC 内置程序- <https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html>。

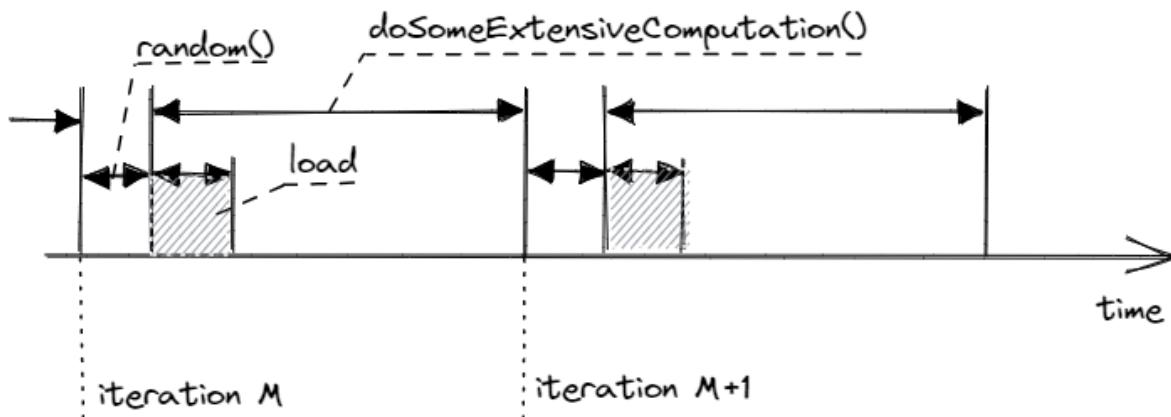


Figure 64: 通过与其他执行重叠来隐藏缓存未命中延迟.

是，也不应该插入得太早，因为它可能会污染缓存，使数据长时间未使用。请注意，在 Listing 146 中，`lookAhead` 是一个模板参数，它允许尝试不同的值并查看哪个值能提供最佳性能。更高级的用户可以尝试使用 Section ?? 中描述的方法估计预取窗口，在 easyperf 博客上可以找到使用这种方法的例子。<sup>146</sup>

代码清单: 接下来 8 次迭代的 SW 预获取示例。

```
template <int lookAhead = 8>
void Graph::update(const std::vector<Edge>& edges) {
    for(int i = 0; i + lookAhead < edges.size(); i++) {
        VertexID v = edges[i].from;
        VertexID u = edges[i].to;
        this->out_neighbors[u].push_back(v);
        this->in_neighbors[v].push_back(u);

        // prefetch elements for future iterations
        VertexID v_next = edges[i + lookAhead].from;
        VertexID u_next = edges[i + lookAhead].to;
        __builtin_prefetch(this->out_neighbors.data() + v_next);
        __builtin_prefetch(this->in_neighbors.data() + u_next);
    }
    // process the remainder of the vector `edges` ...
}
```

软件内存预取最常用于循环中，但也可以将这些提示插入到父函数中，这再次取决于可用的预取窗口。

这种技术是一个强大的武器，但应该非常谨慎地使用，因为它并不容易正确使用。首先，显式内存预取不可移植，这意味着如果它在一个平台上带来了性能提升，并不保证在另一个平台上也能获得类似的加速。它非常依赖于实现，并且平台不需要遵守这些提示。在这种情况下，它可能会降低性能。我的建议是使用所有可用工具验证影响是积极的。不仅要检查性能数字，还要确保缓存未命中数量（尤其是 L3）下降。一旦将更改提交到代码库中，请监控您运行应用程序的所有平台上的性能，因为它可能对周围代码的更改非常敏感。如果收益不超过潜在的维护负担，请考虑放弃这个想法。

<sup>146</sup> “精确计时的机器代码与 Linux perf” - <https://easyperf.net/blog/2019/04/03/Precise-timing-of-machine-code-with-Linux-perf#application-estimating-prefetch-window>。

对于一些复杂的场景，请确保代码实际预取了正确的内存位置。当循环的当前迭代依赖于前一个迭代时，事情可能会变得棘手，例如存在 `continue` 语句或通过 `if` 条件改变要处理的下一个元素。在这种情况下，我的建议是使用工具代码来测试预取提示的准确性。因为使用不当，它会通过驱逐其他有用数据来降低缓存的性能。

最后，显式预取会增加代码大小，并增加 CPU 前端压力。预取提示只是一个进入内存子系统的伪加载，但没有目标寄存器。就像任何其他指令一样，它会消耗 CPU 资源。请极其谨慎地应用它，因为使用错误时，它可能会降低程序的性能。

## 7.3 内存分析

内存分析是识别程序内存使用情况并优化其内存分配和释放行为的过程。它对于提高应用程序性能和稳定性至关重要，尤其是在资源受限的环境中。

本节将介绍用于内存分析的两种常用方法：

### 1. 测量内存占用:

- 查看应用程序使用的内存总量。
- 识别内存泄漏，即应用程序未能释放不再需要的内存。
- 确定内存分配高峰，以了解应用程序何处可能受益于优化。

### 2. 可视化内存访问热点图:

- 了解程序哪些部分最频繁地访问内存。
- 识别数据结构或算法中可能存在内存优化机会的地方。
- 帮助调试内存相关问题，例如缓存未命中或数据访问模式不佳。

注意：目前这些内容只是一个概要，需要进一步完善。

下面是一些可以用于内存分析的工具：

- Valgrind：一个开源的内存调试工具，可以检测内存泄漏、未初始化使用、无效指针访问等问题。
- gperftools：谷歌开发的一组性能分析工具，包括内存分析功能。
- jemalloc：一个高性能的内存分配器，提供内置的内存分析功能。
- Perf：Linux 内核提供的性能分析工具，可以测量内存访问次数等。

您还可以使用编程语言提供的内存分析库或函数，例如 Python 中的 `memory_profiler` 或 Java 中的 `AllocationSampler`。

选择合适的内存分析工具和方法取决于您的具体需求和应用程序的特性。

[TODO] 以下内容可以考虑移到第 6 章 - perf 方法：

- 使用 perf 工具进行内存分析
- 分析 perf 输出中的内存事件

## 7.4 减少 DTLB 未命中

正如本书前面所述，TLB 是一个快速但有限的每个内核缓存，用于将内存地址的虚拟到物理地址转换。如果没有它，应用程序每次内存访问都需要耗时的内核页表遍历来计算每个引用虚拟地址的正确物理地址。在具有 5 级页表的系统中，它将需要访问至少 5 个不同的内存位置才能获得地址转换。在 Section 10.6 部分，我们将讨论如何将大页面用于代码。在这里，我们将看到它们如何用于数据。

任何随机访问大内存区域的算法都可能遭受 DTLB 未命中之苦。这类应用程序的例子包括：在大数组中进行二进制搜索，访问大型哈希表，遍历图。使用大页面有可能加速这类应用程序。

在 x86 平台上，默认页面大小为 4KB。考虑一个应用程序主动引用数百 MB 的内存。首先，它需要分配许多小页面，这代价很高。其次，它将触及许多 4KB 大小的页面，每个页面都将在有限的一组 TLB 条目中竞争。例如，使用 2MB 的大页面，可以使用仅仅十个页面映射 20MB 的内存，而使用 4KB 的页面，您将需要 5120 个页面。这意味着需要的 TLB 条目更少，从而减少了 TLB 未命中次数。由于 2MB 条目的数量少得多，因此不会按 512 的比例减少。例如，在英特尔的 Skylake 内核系列中，L1 DTLB 为 4KB 页面提供 64 个条目，为 2MB 页面仅提供 32 个条目。除了 2MB 的大页面，AMD 和英特尔的 x86 架构芯片还支持 1GB 的超大页面，这些页面仅可用于数据，不能用于指令。使用 1GB 页面而不是 2MB 页面可以进一步减少 TLB 压力。

使用大页面通常会导致更少的页面遍历，并且在 TLB 未命中情况下遍历内核页表的惩罚也会减少，因为表本身更加紧凑。利用大页面的性能提升有时可以高达 30%，具体取决于应用程序遇到的 TLB 压力有多大。期望 2 倍的加速会要求太高，因为 TLB 未命中是主要瓶颈的情况相当罕见。论文 [Luo et al., 2015] 介绍了在 SPEC2006 benchmark 套件上使用大页面的评估。结果可以总结如下。在套件中的 29 个基准测试中，有 15 个的加速在 1% 以内，可以忽略不计。六个基准测试的加速范围在 1%-4% 之间。四个基准测试的加速范围在 4% 到 8% 之间。两个基准测试的加速分别为 10%，而获得最大收益的两个基准测试分别享受了 22% 和 27% 的加速。

许多现实世界的应用程序已经利用了大页面，例如 KVM、MySQL、PostgreSQL、Java JVM 等。通常，这些软件包提供了一个启用该功能的选项。每当您使用类似应用程序时，请查看其文档，了解是否可以启用大页面。

Windows 和 Linux 都允许应用程序建立大页面内存区域。有关如何在 Windows 和 Linux 上启用大页面的说明，请参见附录 C。在 Linux 上，应用程序中有两种使用大页面的方式：显式大页面和透明大页面。Windows 的支持不像 Linux 那么丰富，将在以后讨论。

#### 7.4.1 显式大页面 (EHP)。

显式大页面 (EHP) 是系统内存的一部分，作为大页面文件系统 hugetlbfs 暴露。顾名思义，EHP 应在启动时或运行时预留。有关如何操作的说明，请参见附录 C。在启动时预留 EHP 可以增加分配成功的可能性，因为内存尚未严重碎片化。显式预分配的页面驻留在预留的内存块中，并且在内存压力下无法换出。此外，该内存空间无法用于其他目的，因此用户应谨慎分配，仅预留他们需要的页面数量。

在应用程序中使用 EHP 的最简单方法是在 `mmap` 中调用 `MAP_HUGETLB`，如 Listing 7.4.1 所示。在此代码中，指针 `ptr` 将指向一个 2MB 的内存区域，该区域是显式预留给 EHP 的。请注意，由于 EHP 没有预先保留，分配可能会失败。应用程序中使用 EHP 的另一种不太流行的方法可以在附录 C 中找到。此外，开发人员可以编写自己的基于 `arena` 的分配器，利用 EHP 进行分配。

代码清单：从显式分配的巨大页面映射内存区域。

```
void ptr = mmap(nullptr, size, PROT_READ | PROT_WRITE,
                MAP_PRIVATE | MAP_ANONYMOUS | MAP_HUGETLB, -1, 0);
if (ptr == MAP_FAILED)
    throw std::bad_alloc{};
...
munmap(ptr, size);
```

过去，可以使用 libhugetlbfs：<https://github.com/libhugetlbfs/libhugetlbfs><sup>147</sup> 库，该库允许覆盖现有动态链接的可执行文件使用的 `malloc` 调用，从而在大页面 (EHP) 之上分配内存。不幸的是，此项目已不再维护。它不需要用户修改代码或重新链接二进制文件。他们只需用 `LD_PRELOAD=libhugetlbfs.so HUGETLB_MORECORE=yes <your app command line>` 预先填充命令行即可使用它。但幸运的是，还有其他库允许使用大页面（非 EHP）和 `malloc`，我们稍后会看到。

<sup>147</sup> libhugetlbfs - <https://github.com/libhugetlbfs/libhugetlbfs>.

#### 7.4.2 透明大页面 (THP)

Linux 还提供透明大页面支持 (THP)，它具有两种操作模式：系统范围和每个进程。启用系统范围的 THP 时，内核会自动管理大页面，这对应用程序是透明的。操作系统内核尝试在需要大量内存块时将大页面分配给任何进程，并且如果可以分配这样的页面，则无需手动保留大页面。如果启用每个进程的 THP，内核仅将大页面分配给单个进程的内存区域，这些区域归因于 `madvise` 系统调用。您可以使用以下命令检查系统中是否启用了 THP：

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

如果值为 `always` (系统范围) 或 `madvise` (每个进程)，则 THP 可供您的应用程序使用。有关每个选项的详细规范，请参见 Linux 内核文档<sup>148</sup> 中关于 THP 的内容。

启用系统范围的 THP 时，将自动将大页面用于常规内存分配，而无需应用程序明确请求。基本上，要观察大页面对应用程序的影响，用户只需启用系统范围的 THP，使用 `echo "always" | sudo tee /sys/kernel/mm/transparent_hugepage/enabled` 即可。它将自动启动名为 `khugepaged` 的守护进程，该进程开始扫描应用程序的内存空间，将普通页面提升为大页面。不过，有时内核可能无法将普通页面提升为大页面，因为找不到 2MB 的连续内存块。

系统范围的 THP 模式适用于快速实验，以检查大页面是否能提高性能。它可以自动工作，即使对于不知道 THP 的应用程序也是如此，因此开发人员不必更改代码即可看到大页面对他们应用程序的好处。

启用系统范围的大页面时，应用程序最终可能会分配更多的内存资源。应用程序可能会映射一个大区域，但只触碰其中 1 个字节，在这种情况下，可能会分配一个 2MB 的页面而不是 4k 的页面，这毫无意义。这就是为什么可以禁用系统范围的 THP，只让它们存在于 `MADV_HUGE PAGE` `madvise` 区域中，我们将在下面讨论这一点。完成实验后，请记得禁用系统范围的 THP，因为它可能不会使系统上运行的每个应用程序都受益。

使用 `madvise` (每个进程) 选项时，THP 仅在通过 `madvise` 系统调用并带有 `MADV_HUGE PAGE` 标志的内存区域内启用。如 Listing 148 所示，指针 `ptr` 将指向一个 2MB 的匿名 (透明) 内存区域，该区域由内核动态分配。如果内核找不到 2MB 的连续内存块，`mmap` 调用可能会失败。

代码清单: 将内存区域映射到一个透明的巨大页面。

```
void ptr = mmap(nullptr, size, PROT_READ | PROT_WRITE | PROT_EXEC,
               MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
if (ptr == MAP_FAILED)
    throw std::bad_alloc();
madvise(ptr, size, MADV_HUGE PAGE);
// use the memory region `ptr`
munmap(ptr, size);
```

开发人员可以根据 Listing 148 中的代码构建自定义 THP 分配器。但是，他们还可以将 THP 用于应用程序进行的 `malloc` 调用中。许多内存分配库通过覆盖 `libc` 的 `malloc` 实现来提供此功能。以下是最流行的此类库之一 `jemalloc` 的示例。

如果您拥有应用程序的源代码，则可以使用附加的 `-ljemalloc` 选项重新链接二进制文件。这将使您的应用程序与 `jemalloc` 库动态链接，该库将处理所有 `malloc` 调用。然后使用以下选项启用堆分配的 THP：

```
$ MALLOC_CONF="thp:always" <your app command line>
```

如果您没有源代码，仍然可以通过预加载动态库来利用 `jemalloc`：

<sup>148</sup> Linux kernel THP documentation - <https://www.kernel.org/doc/Documentation/vm/transhuge.txt>

```
$ LD_PRELOAD=/usr/local/libjemalloc.so.2 MALLOC_CONF="thp:always" <your app command line>
```

Windows 只提供类似于 Linux THP 每个进程模式的方式使用大页面，通过 WinAPI VirtualAlloc 系统调用。有关详细信息，请参见附录 C。

### 7.4.3 显式大页面 (EHP) vs. 透明大页面 (THP)

Linux 用户可以使用三种不同的模式使用大页面：

- 显式大页面 (EHP)
- 系统范围透明大页面 (THP)
- 每个进程透明大页面 (THP)

让我们比较一下这些选项。首先，EHP 预先保留在虚拟内存中，而 THP 则没有。这使得使用 EHP 的软件包更难以交付，因为它们依赖于机器管理员所做的特定配置设置。此外，EHP 静态驻留在内存中，占用宝贵的 DRAM 空间，即使它们未使用。

其次，系统范围的透明大页面非常适合快速实验。无需更改用户代码即可测试在您的应用程序中使用大页面的好处。但是，将软件包运送给客户并要求他们启用系统范围的 THP 是不明智的，因为这可能会对该系统上运行的其他程序产生负面影响。通常，开发人员会识别代码中可以从大页面中受益的分配，并在这些位置使用 madvise 提示（每个进程模式）。

每个进程的 THP 没有上述任何一个缺点，但它还有一个缺点。之前我们讨论过，内核分配 THP 对用户来说是透明的。分配过程可能涉及多个内核进程，这些进程负责在虚拟内存中腾出空间，这可能包括将内存换出到磁盘、碎片化或提升页面。透明大页面的后台维护会产生内核在管理不可避免的碎片和交换问题时产生的非确定性延迟开销。EHP 不受内存碎片化影响，也不能换出到磁盘，因此延迟开销要小得多。

总而言之，THP 更易于使用，但会产生更大的分配延迟开销。这正是 THP 在高频交易和其他超低延迟行业不受欢迎的原因，他们更喜欢使用 EHP。另一方面，虚拟机提供商和数据库往往倾向于使用每个进程的 THP，因为要求额外的系统配置会给他们的用户带来负担。

## 7.5 问题与练习

1. 完成 perf-ninja::data\_packing 实验作业，其中你需要使数据结构更紧凑。
2. 通过为未来循环迭代实现显式内存预取，解决 perf-ninja::swmem\_prefetch\_1 实验作业。
3. 使用我们在 Section 7.4 中讨论的方法解决 perf-ninja::huge\_pages\_1 实验作业。观察性能、/proc/meminfo 中的大页面分配以及测量 DTLB 加载和未命中次数的 CPU 性能计数器的任何变化。
4. 描述一段代码成为缓存友好的需要哪些条件？
5. 运行您每天使用的应用程序。测量其内存占用情况，分析并识别热点内存访问。它们是缓存友好的吗？有没有办法改进它们？

## 7.6 章节总结

- 大多数现实世界的应用程序都会遇到与 CPU 后端相关的性能瓶颈。这不足为奇，因为所有与内存相关的问题以及低效的计算都属于这一类别。
- 内存子系统的性能增长速度不及 CPU 性能增长速度。然而，内存访问是许多应用程序性能问题的常见来源。加速这类程序需要修改它们访问内存的方式。
- 在 第 x 节：内存限制 (MemBound): 你需要替换 x 为对应章节的编号，我们讨论了一些流行的配方，用于构建缓存友好的数据结构、内存预取以及利用大内存页来提高 DTLB 性能。

## 8 优化计算

在上章节中，我们讨论了如何清除路径以实现高效的内存访问。完成后，是时候研究 CPU 如何使用从内存中获取的数据。现代应用程序需要大量的 CPU 计算，尤其是涉及复杂图形、人工智能、加密货币挖掘和大数据处理的应用程序。在本章中，我们将重点关注优化计算，可以减少 CPU 需要做的工作量并提高程序的整体性能。

当应用 TMA 方法时，低效计算通常反映在“核心绑定”类别中，并在一定程度上反映在“退休”类别中。“核心绑定”类别代表 CPU 乱序执行引擎内部的所有停顿，这些停顿不是由内存问题引起的。主要有两大类：

- 软件指令之间的数据依赖性限制了性能。例如，一系列长的依赖操作可能导致指令级并行性 (ILP) 低，并浪费许多执行槽位。下一节将更详细地讨论数据依赖链。
- 硬件计算资源不足（也称为执行吞吐量不足）。它表明某些执行单元过载（也称为执行端口争用）。当工作负载频繁执行许多相同类型的指令时，可能会发生这种情况。例如，AI 算法通常执行大量乘法，科学应用程序可能运行许多除法和开方运算。但是任何给定的 CPU 核心中的乘法器和除法器数量都有限。因此，当端口争用发生时，指令会排队等待执行。这种类型的性能瓶颈与特定的 CPU 微架构非常特定，通常无法治愈。

在 Section ?? 中，我们说高“退休”指标是一个性能良好的代码的良好指标。其背后的原理是执行没有停顿，CPU 以高速度执行指令。然而，有时它可能会掩盖真正的性能问题，即低效的计算。工作负载可能执行许多指令，这些指令过于简单，没有做太多有用的工作。在这种情况下，高“退休”指标不会转化为高性能。

在本章中，我们将研究一些众所周知的技术，例如函数内联、矢量化和循环优化。这些代码转换旨在减少执行的指令总数，或者用更有效的指令替换它们。

### 8.1 数据流依赖

当程序语句引用前面语句的数据时，我们称这两个语句之间存在数据依赖性。有时人们也使用“依赖链”或“数据流依赖”等术语。我们最熟悉的例子如图 65 所示。要访问节点  $N+1$ ，我们应该首先取消引用指针  $n \rightarrow \text{next}$  的引用。对于右边的循环，这是一个递归的数据依赖性，这意味着它跨越了循环的多个迭代。基本上，遍历一个链表是一个非常长的依赖链。

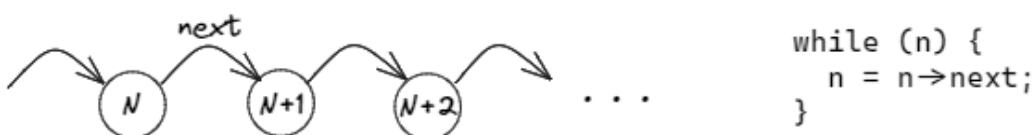


Figure 65: 遍历链表时的数据依赖性.

传统程序是在假设顺序执行模型的情况下编写的。在这个模型下，指令一个接一个地执行，原子地按照程序指定的顺序执行。然而，正如我们已经知道的，现代 CPU 不是这样构建的。它们被设计成乱序执行指令，并行执行，并以最大限度利用可用执行单元的方式执行。

当出现长数据依赖性时，处理器被迫以顺序执行代码，只利用了其全部能力的一部分。长依赖链阻碍了并行性，这违背了现代超标量 CPU 的主要优势。例如，指针追逐不能从 OOO 执行中获益，因此将以顺序 CPU 的速度运行。正如我们将在本节看到的那样，依赖链是性能瓶颈的主要来源。

您无法消除数据依赖性，它们是程序的基本属性。任何程序都接受输入来计算一些东西。事实上，人们已经开发了技术来发现语句之间的数据依赖性并构建数据流图。这称为依赖性分析，更适合编译器开发人员，而不是性能工程师。我们不希望为整个程序构建数据流图。相反，我们想在一段热代码（循环或函数）中找到一个关键的依赖链。

您可能会问：“如果不能摆脱依赖链，您可以做什么？”嗯，有时这会成为性能的限制因素，不幸的是您将不得不忍受它。在本书的最后一章，我们将讨论硬件中打破依赖链的一种可能解决方案，称为值预测。现在，您应该寻找打破不必要的数据依赖链或使其执行重叠的方法。一个这样的例子显示在 Listing 8.1 中。与其他一些情况类似，我们在左边展示了源代码，右边是相应的 ARM 汇编代码。此外，这个代码示例包含在 Performance Ninja 的 Github 存储库中，因此您可以自己尝试一下。

这个小程序模拟了随机粒子运动。我们有 1000 个粒子在 2D 表面上运动，没有约束，这意味着它们可以离它们的起始位置尽可能远。每个粒子由其在 2D 表面上的 x 和 y 坐标以及速度定义。初始 x 和 y 坐标在范围 [-1000,1000] 内，速度在范围 [0;1] 内，不会改变。程序模拟每个粒子 1000 个运动步长。对于每个步骤，我们使用随机数生成器 (RNG) 生成一个角度，该角度设置粒子的运动方向。然后，我们相应地调整粒子的坐标。

考虑到手头的任务，您决定自己编写 RNG、正弦和余弦函数，以牺牲一些精度使其尽可能快。毕竟，这是随机运动，所以这是一个不错的权衡。您选择中等质量的 XorShift RNG，因为它只有 3 个移位和 3 个异或操作。还有什么更简单的？另外，您很快搜索了网络，使用多项式找到了正弦和余弦近似值，既准确又快速。

让我们快速检查一下生成的 ARM 汇编代码：\* 前三个 eor 指令与 lsl 或 lsr 组合对应于 XorShift32::gen() 函数。  
 \* 接下来的 ucvtf 和 fmul 用于将角度从度转换为弧度（代码第 35 行）。\* 正弦和余弦函数都具有两个 fmul 和一个 fmadd 操作。余弦还具有附加的 fadd。\* 最后，我们再有一对 fmadd 来分别计算 x 和 y，以及 stp 指令将坐标对存储回原处。

使用 Clang-17 C++ 编译器编译了代码并在 Mac mini (Apple M1, 2020) 上运行。期望这段代码“飞起来”，但是有一个非常讨厌的性能问题使程序变慢。不提前阅读文本，你能在代码中找到一个递归依赖链吗？

代码清单: 二维表面上的随机粒子运动

```

1 struct Particle {
2     float x; float y; float velocity;
3 };
4
5 class XorShift32 {
6     uint32_t val;
7 public:
8     XorShift32 (uint32_t seed) : val(seed) {}
9     uint32_t gen() {
10         val ^= (val << 13);
11         val ^= (val >> 17);
12         val ^= (val << 5);
13         return val;
14     }
15 };
16
17 static float sine(float x) {
18     const float B = 4 / PI_F;
19     const float C = -4 / (PI_F * PI_F);
20     return B * x + C * x * std::abs(x);
21 }
22 static float cosine(float x) {
23     return sine(x + (PI_F / 2));
24 }
```

|        |                     |
|--------|---------------------|
| .loop: |                     |
| eor    | w0, w0, w0, lsl #13 |
| eor    | w0, w0, w0, lsr #17 |
| eor    | w0, w0, w0, lsl #5  |
| ucvtf  | s1, w0              |
| fmov   | s2, w9              |
| fmul   | s2, s1, s2          |
| fmov   | s3, w10             |
| fadd   | s3, s2, s3          |
| fmov   | s4, w11             |
| fmul   | s5, s3, s3          |
| fmov   | s6, w12             |

```

25
26 /* Map degrees [0;UINT32_MAX) to radians [0;2*pi)*/
27 float DEGREE_TO_RADIAN = (2 * PI_D) / UINT32_MAX;
28
29 void particleMotion(vector<Particle> &particles,
30                     uint32_t seed) {
31     XorShift32 rng(seed);
32     for (int i = 0; i < STEPS; i++) {
33         for (auto &p : particles) {
34             uint32_t angle = rng.gen();
35             float angle_rad = angle * DEGREE_TO_RADIAN;
36             p.x += cosine(angle_rad) * p.velocity;
37             p.y += sine(angle_rad) * p.velocity;
38         }
39     }

```

|                        |
|------------------------|
| fmul s5, s5, s6        |
| fmadd s3, s3, s4, s5   |
| ldp s6, s4, [x1, #0x4] |
| ldr s5, [x1]           |
| fmadd s3, s3, s4, s5   |
| fmov s5, w13           |
| fmul s5, s1, s5        |
| fmul s2, s5, s2        |
| fmadd s1, s1, s0, s2   |
| fmadd s1, s1, s4, s6   |
| stp s3, s1, [x1], #0xc |
| cmp x1, x16            |
| b.ne .loop             |

恭喜您找到了它！存在一个关于 `XorShift32::val` 的循环依赖。要生成下一个随机数，生成器必须首先生成前一个数字。`gen()` 方法的下一个调用将基于前一个数字生成数字。图 66 直观地显示了有问题的循环进位依赖。请注意，计算粒子坐标的代码（将角度转换为弧度，正弦，余弦，乘以速度）会在相应的随机数准备好后立即开始执行，但不会更早。

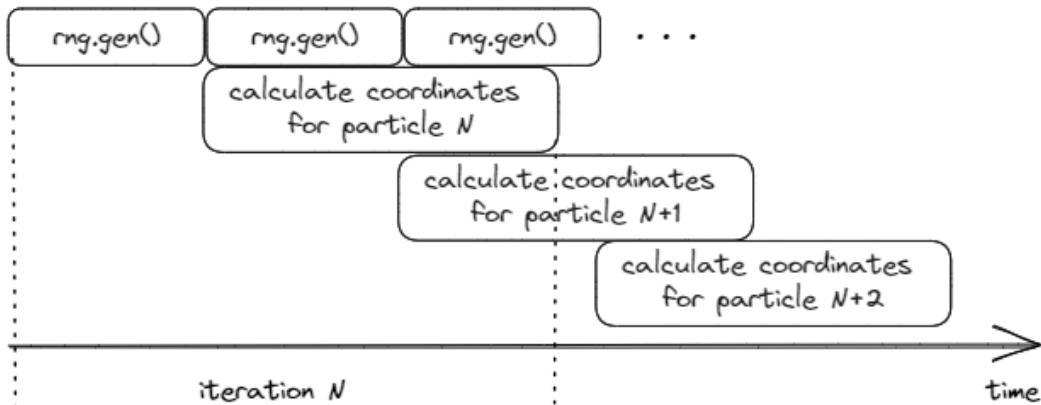


Figure 66: 在 Listing 8.1 中依赖执行的可视化

每个粒子坐标的计算代码彼此独立，因此向左拉伸它们以更多地重叠它们的执行可能会有益。您可能想知道：“但是这三个（或六个）指令如何拖慢整个循环？”确实，循环中还有许多其他“繁重”指令，例如 `fmul` 和 `fmadd`。然而，它们不在关键路径上，因此可以与其他指令并行执行。并且由于现代 CPU 非常宽，它们将同时执行来自多个迭代的指令。这允许 OOO 引擎在循环的不同迭代中有效地找到并行性（独立指令）。

让我们做一些粗略的计算。<sup>149</sup> 每个 `eor` 和 `ls1` 指令的延迟为 2 个周期，一个周期用于移位，一个周期用于异或。我们有三个依赖的 `eor + ls1` 对，因此生成下一个随机数需要 6 个周期。这是我们这个循环的绝对最小值，我们不能以每迭代 6 个周期以上的速度运行。后续代码至少需要 20 个周期延迟才能完成所有 `fmul` 和 `fmadd` 指令。但这没关系，因为它们不在关键路径上。重要的是这些指令的吞吐量。经验法则：如果一条指令处于关键路径上，请查看其延迟；如果它不在关键路径上，请查看其吞吐量。在每个循环迭代中，我们有 5 个 `fmul` 和 4 个 `fmadd` 指令，它们都

<sup>149</sup> 苹果公司没有公布他们产品的指令延迟和吞吐量，但有一些实验可以说明这一点，这里有一个这样的研究：<https://dougallj.github.io/applecpu/firestorm-simd.html>。由于这是非官方的数据来源，你应该有所保留。

在同一组执行单元上执行。M1 处理器可以每周期运行 4 条这种类型的指令，因此发出所有 `fmul` 和 `fmadd` 指令至少需要  $9/4 = 2.25$  个周期。因此，我们有两个性能限制：第一个由软件强制（每个迭代 6 个周期，由于依赖链），第二个由硬件强制（每个迭代 2.25 个周期，由于执行单元的吞吐量）。现在我们受第一个限制约束，但我们可以尝试打破依赖链以接近第二个限制。

解决这个问题的一种方法是使用额外的 RNG 对象，使其一个为循环的偶数迭代提供数据，另一个为奇数迭代提供数据，如 Listing 149 所示。请注意，我们还手动展开循环。现在我们有两个独立的依赖链，可以并行执行。有人可能会说这改变了程序的功能，但用户无法分辨，因为粒子的运动是随机的。另一种解决方案是选择一个内部依赖链更少的 RNG。

代码清单：二维表面上的随机粒子运动

```
void particleMotion(vector<Particle> &particles,
                     uint32_t seed1, uint32_t seed2) {
    XorShift32 rng1(seed1);
    XorShift32 rng2(seed2);
    for (int i = 0; i < STEPS; i++) {
        for (int j = 0; j + 1 < particles.size(); j += 2) {
            uint32_t angle1 = rng1.gen();
            float angle_rad1 = angle1 * DEGREE_TO_RADIAN;
            particles[j].x += cosine(angle_rad1) * particles[j].velocity;
            particles[j].y += sine(angle_rad1) * particles[j].velocity;
            uint32_t angle2 = rng2.gen();
            float angle_rad2 = angle2 * DEGREE_TO_RADIAN;
            particles[j+1].x += cosine(angle_rad2) * particles[j+1].velocity;
            particles[j+1].y += sine(angle_rad2) * particles[j+1].velocity;
        }
        // remainder (not shown)
    }
}
```

完成此转换后，编译器开始自动矢量化循环主体，即它将两个链粘合在一起并使用 SIMD 指令并行处理它们。为了隔离打破依赖链的影响，我们禁用了编译器矢量化。

为了测量改变的影响，我们运行了“之前”和“之后”版本，观察到运行时间从每个迭代 19 毫秒降至每个迭代 10 毫秒。这几乎是 2 倍的加速。IPC 也从 4.0 上升到 7.1。为了尽职尽责，我们还测量了其他指标，以确保性能不会因其他原因意外提升。在原始代码中，MPKI 为 0.01，BranchMispredRate 为 0.2%，这意味着程序最初没有遭受缓存未命中或分支预测错误。这里还有另一个数据点：在英特尔的 Alderlake 系统上运行相同的代码时，它显示了 74% 的 Retiring 和 24% 的 Core Bound，这证实了性能受计算限制。

通过一些额外的更改，您可以将此解决方案通用化，以拥有您想要的任意数量的依赖链。对于 M1 处理器，测量结果表明拥有 2 个依赖链足以非常接近硬件限制。拥有超过 2 个链路的性能提升可以忽略不计。然而，一个趋势是 CPU 变得越来越宽，即它们越来越能够并行运行多个依赖链。这意味着未来的处理器可以受益于拥有超过 2 个依赖链。与往常一样，您应该测量并找到您的代码将在其上运行的平台的最佳点。

有时仅打破依赖链还不够。想象一下，您没有一个简单的 RNG，而是一个非常复杂的加密算法，它长达 10'000 个指令。因此，现在我们不再是一个非常短的 6 个指令依赖链，而是有 10'000 个指令位于关键路径上。您立即进行上述相同的更改，期待获得 2 倍的加速。只会看到性能稍有提升。发生了什么？

这里的问题是 CPU 无法“看到”第二个依赖链开始执行它。回想一下第 3 章，预留站 (RS) 的容量不足以提前看到

10'000 条指令，因为它比这小得多。因此，CPU 将无法重叠执行两个依赖链。为了修复它，我们需要交错这两个依赖链。使用这种方法，您需要更改代码，以便 RNG 对象同时生成两个数字，函数 gen() 中的每个语句都重复并交错。即使编译器内联所有代码并且可以清楚地看到两个链，它也不会自动交错它们，因此您需要小心这一点。您在执行此操作时可能遇到的另一个限制是寄存器压力。并行运行多个依赖链需要保持更多状态，因此需要更多寄存器。如果您用完寄存器，编译器会将它们溢出到堆栈，从而减慢程序速度。

作为结束时的思考，我们想强调找到关键依赖链的重要性。这并不总是容易的，但要知道您的循环、函数或代码段的关键路径上是什么至关重要。否则，您可能会发现自己在修复次要问题，而这些问题几乎没有影响。

## 8.2 内联函数

如果你是那种经常查看汇编代码的开发人员，你可能见过 CALL、PUSH、POP 和 RET 指令。在 x86 指令集中，CALL 和 RET 指令用于调用和返回函数。PUSH 和 POP 指令用于将寄存器值保存到堆栈上并恢复它。

函数调用的微妙之处由调用约定描述，即如何传递参数和顺序，如何返回结果，调用的函数必须保留哪些寄存器以及工作在调用方和被调用方之间如何分配。基于调用约定，当调用者调用一个函数时，它期望在被调用者返回后一些寄存器将保持相同的值。因此，如果被调用者需要更改应保留的寄存器之一，它需要在返回给调用者之前保存（PUSH）和恢复（POP）它们。一系列 PUSH 指令称为序言，一系列 POP 指令称为尾声。

当一个函数很小的时候，调用函数的开销（序言和尾声）可能非常明显。通过将函数体内联到调用位置，可以消除这种开销。函数内联是将对函数 F 的调用替换为对实际参数专门化的 F 代码的过程。内联是最重要的编译器优化之一。不仅因为它消除了调用函数的开销，而且还使其他优化变得可能。这是因为当编译器内联一个函数时，编译器分析的范围会扩大到一个更大的代码块。然而，也有缺点：内联可能会增加代码大小和编译时间<sup>150</sup>。

许多编译器中函数内联的主要机制依赖于成本模型。例如，在 LLVM 编译器中，它基于为每个函数调用（调用点）计算成本。内联函数调用的成本基于该函数中指令的数量和类型。如果成本低于阈值，通常是固定的阈值，则会进行内联，但在某些情况下可以变化<sup>151</sup>。除了通用成本模型之外，还有许多启发式方法可以在某些情况下覆盖成本模型的决策。例如：

- 微小的函数（包装器）几乎总是被内联。
- 只有一个调用点的函数是内联的首选候选项。
- 大型函数通常不会被内联，因为它们会膨胀调用函数的代码。

此外，有些情况下内联会有问题：

- 递归函数不能内联到自身。
- 通过指针引用的函数可以内联到直接调用的地方，但是该函数必须保留在二进制文件中，即它不能完全内联和消除。对于具有外部链接的函数也是如此。

正如我们之前所说，编译器在决定是否内联函数时倾向于使用成本模型方法，这在实践中通常效果很好。一般来说，依靠编译器做出所有内联决策并在需要时进行调整是一个很好的策略。成本模型无法考虑到每种可能的情况，这为改进留下了空间。有时候编译器需要开发人员的特殊提示。一种找到程序中潜在内联候选项的方法是查看分析数据，特别是函数序言和尾声的热度。下面是一个函数剖面的示例，其中序言和尾声消耗了函数时间的~50%：

开销 | 函数 `foo` 的源代码和反汇编

(%) |

```
3.77 : 418be0: push r15      # 序言
4.62 : 418be2: mov r15d,0x64
```

<sup>150</sup> 参见文章：<https://aras-p.info/blog/2017/10/09/Forced-Inlining-Might-Be-Slow/>。

<sup>151</sup> 例如，1) 当函数声明带有内联提示时，2) 当存在函数的分析数据时，或者 3) 当编译器优化大小 (-Os) 而不是性能 (-O2) 时。

```

2.14 : 418be8: push r14
1.34 : 418bea: mov r14,rsi
3.43 : 418bed: push r13
3.08 : 418bef: mov r13,rdi
1.24 : 418bf2: push r12
1.14 : 418bf4: mov r12,rcx
3.08 : 418bf7: push rbp
3.43 : 418bf8: mov rbp,rdx
1.94 : 418bfb: push rbx
0.50 : 418bfc: sub rsp,0x8
...
#                                     # 函数体
...
4.17 : 418d43: add rsp,0x8 # 尾声
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

```

当你看到热的 PUSH 和 POP 指令时，这可能是一个很强的指示，即函数

序言和尾声的时间可能会被节省，如果我们内联函数的话。请注意，即使序言和尾声很热，也不一定意味着内联函数会有利可图。内联会触发许多不同的更改，因此很难预测结果。在强制编译器内联函数之前，请始终测量更改代码的性能。

对于 GCC 和 Clang 编译器，可以使用 C++11 的 `[[gnu::always_inline]]` 属性作为内联 `foo` 的提示，如下面的代码示例所示。对于较早的 C++ 标准，可以使用 `__attribute__((always_inline))`。对于 MSVC 编译器，可以使用 `__forceinline` 关键字。

```

[[gnu::always_inline]] int foo() {
    // foo body
}

```

### 8.3 循环优化

循环是几乎所有高性能程序的核心。由于循环代表了执行大量次的代码片段，因此它们是执行时间花费最多的部分。在这样一个关键代码段进行微小的更改可能会对程序的性能产生重大影响。这就是为什么仔细分析程序中热点循环的性能并了解改进它们的可能方法非常重要。

要有效地优化循环，关键是要了解性能瓶颈。一旦找到占用大部分时间的循环，就尝试确定限制其性能的因素。通常，它将是以下一种或多种情况：内存延迟、内存带宽或机器的计算能力。Roofline 性能模型（Section 5.6）是评估不同循环相对于硬件理论最大值的性能的一个良好起点。自上而下的微架构分析（Section ??）也可以成为有关瓶颈的另一个很好的信息来源。

在本节中，我们将研究针对上述瓶颈类型最著名的循环优化。我们首先讨论低级别的优化，这些优化只会在一个循环中移动代码。此类优化通常有助于提高循环内部计算的有效性。接下来，我们将研究重构循环的高级别优化，这

些优化通常会影响多个循环。第二类优化通常旨在改进内存访问，消除内存带宽和内存延迟问题。请注意，这不是所有已知循环转换的完整列表。有关下面讨论的每个转换的更详细信息，读者可以参考 [Cooper & Torczon, 2012]。

编译器可以自动识别执行某些循环转换的机会。然而，有时需要开发人员干预才能达到所需的结果。在本节的第二部分，我们将分享一些有关如何发现循环优化机会的想法。了解对给定循环进行了哪些转换以及编译器未能进行哪些优化是成功性能调优的关键之一。最后，我们将考虑使用多面体框架优化循环的另一种方法。

### 8.3.1 低级别优化

首先，我们将考虑将代码转换为单个循环内的简单循环优化：循环不变式代码运动、循环展开、循环强度降低和循环取消交换。此类优化通常有助于提高具有高算术强度的循环的性能（请参见 Section 5.6），即当循环受 CPU 计算能力限制时。通常，编译器擅长进行此类转换；但是，在某些情况下，编译器可能需要开发人员的支持。我们将在后续部分讨论这个问题。

**循环不变式代码运动 (LICM)**: 在循环中评估且从未改变的表达式称为循环不变式。由于它们的价值不会跨循环迭代而改变，我们可以将循环不变式表达式移出循环。我们通过将结果存储在临时变量中并在循环内部使用它来实现这一点（请参见 Listing 8.3.1）。如今，所有优秀的编译器在大多数情况下都能成功执行 LICM。

代码清单: 循环不变的代码运动

```
for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)    =>
        a[j] = b[j] * c[i];
                                for (int i = 0; i < N; ++i) {
                                    auto temp = c[i];
                                    for (int j = 0; j < N; ++j)
                                        a[j] = b[j] * temp;
                                }
```

**循环展开 (Loop Unrolling)**: 循环变量是在循环中其值是循环迭代次数的函数的变量。例如， $v = f(i)$ ，其中  $i$  是迭代次数。在每次迭代中修改归纳变量可能是不必要的并且代价高昂。相反，我们可以展开循环并为归纳变量的每个增量执行多次迭代（请参见 Listing ??）。

代码清单: 循环展开 ~~~~ {#lst:Unrol.cpp} for (int i = 0; i < N; ++i) for (int i = 0; i+1 < N; i+=2) { a[i] = b[i] \* c[i]; => a[i] = b[i] \* c[i]; a[i+1] = b[i+1] \* c[i+1]; } ~~~~

循环展开的主要好处是每次迭代执行更多的计算。在每个迭代结束时，索引值必须递增、测试，并且如果还有更多迭代要处理，则控制权回到循环顶部。这项工作可以看作是循环的“税收”，可以减少。通过将 Listing ?? 中的循环展开 2 倍，我们可以将执行的比较和分支指令数量减少一半。

循环展开是一种众所周知的优化；然而，许多人仍然对此感到困惑并尝试手动展开循环。我建议任何开发人员都不要手动展开任何循环。首先，编译器非常擅长这样做，并且通常可以最佳地进行循环展开。第二个原因是，处理器由于其乱序推测执行引擎而具有“嵌入式展开器”（参见 Chapter 3）。当处理器等待来自第一个迭代的长时间延迟指令完成（例如加载、除法、微代码指令、长依赖链）时，它将推测性地开始执行来自第二个迭代的指令，并且只等待循环携带的依赖性。这扩展到多个迭代之前，有效地在指令重新排序缓冲区 (ROB) 中展开循环。

**循环强度降低 (LSR)**: 用更便宜的指令替换昂贵的指令。此类转换可以应用于所有使用归纳变量的表达式。强度降低经常应用于数组索引。编译器通过分析变量值如何在循环迭代中演变来执行 LSR。在 LLVM 中，它被称为标量演化 (SCEV)。在 Listing 8.3.1 中，编译器相对容易证明内存位置  $b[i*10]$  是循环迭代次数  $i$  的线性函数，因此它可以使用更便宜的加法替换昂贵的乘法。

代码清单: 循环强度降低

```
for (int i = 0; i < N; ++i)
    a[i] = b[i * 10] * c[i];    =>    int j = 0;
                                         for (int i = 0; i < N; ++i) {
```

```

        a[i] = b[j] * c[i];
        j += 10;
    }
}

```

**循环取消交换 (Loop Unswitching):** 如果循环内部有一个条件语句并且它是不可变的，我们可以将其移出循环。我们通过复制循环主体并将其一个版本放置在条件语句的每个 `if` 和 `else` 子句中来实现这一点（参见 Listing 8.3.1）。虽然循环取消交换可能会使编写代码的数量翻倍，但每个新循环现在都可以单独优化。

代码清单: 循环取消交换

```

for (i = 0; i < N; i++) {
    a[i] += b[i];
    if (c)
        b[i] = 0;
}
if (c)
    => for (i = 0; i < N; i++) {
        a[i] += b[i];
        b[i] = 0;
    }
else
    for (i = 0; i < N; i++) {
        a[i] += b[i];
    }
}

```

### 8.3.2 高级优化

还有一类循环转换会改变循环的结构，并且通常会影响多个嵌套循环。我们将研究循环交换、循环块化（切分）和循环融合和分配（裂变）。这套转换旨在改进内存访问并消除内存带宽和内存延迟瓶颈。从编译器的角度来看，证明此类转换的合法性并证明其性能收益是非常困难的。从这个意义上说，开发人员处于更有利的位置，因为他们只需要关心他们特定代码片段中转换的合法性，而不必关心可能发生的每种情况。不幸的是，这也意味着我们通常必须手动进行此类转换。

**循环交换:** 是交换嵌套循环循环顺序的过程。内循环使用的归纳变量切换到外循环，反之亦然。Listing 8.3.2 显示了交换嵌套循环 `i` 和 `j` 的示例。循环交换的主要目的是对多维数组的元素进行顺序内存访问。通过遵循元素在内存中布局的顺序，我们可以提高内存访问的空间局部性，使我们的代码更加缓存友好。此转换有助于消除内存带宽和内存延迟瓶颈。

代码清单: 循环交换

```

for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        a[j][i] += b[j][i] * c[j][i];
for (j = 0; j < N; j++)
    for (i = 0; i < N; i++)
        a[j][i] += b[j][i] * c[j][i];

```

仅当循环是完美嵌套时，循环交换才是合法的。完美嵌套的循环是所有语句都在最内层循环中的循环。交换不完美的循环嵌套更难做到，但仍然可行，请查看 Codee: <https://www.codee.com/catalog/glossary-perfect-loop-nesting/><sup>152</sup> 目录中的示例。

**循环阻塞 (切分) (Loop Blocking (Tiling)):** 这种转换的思想是将多维执行范围拆分为更小的块（块或切片），以便每个块都能适合 CPU 缓存。如果一个算法使用大型多维数组并对其元素执行跨步访问，则很有可能缓存利用率低下。每次这样的访问都可能将将来会被请求的数据从缓存中推出（缓存逐出）。通过将算法划分为较小的多维块，我们可以确保循环中使用的数据在重用之前保留在缓存中。

<sup>152</sup> Codee: 完美循环嵌套 - [https://www.codee.com/catalog/glossary-perfect-loop-nesting/]([https://www.codee.com/catalog/glossary-perfect-loop-nesting/])

在 Listing 152 中显示的示例中，算法对数组  $a$  的元素执行行主遍历，同时对数组  $b$  执行列主遍历。循环嵌套可以划分为更小的块，以最大限度地重用数组  $b$  中的元素。

代码清单: 阻塞循环

```
// linear traversal                                // traverse in 8*8 blocks
for (int i = 0; i < N; i++)                      for (int ii = 0; ii < N; ii+=8)
    for (int j = 0; j < N; j++)      =>        for (int jj = 0; jj < N; jj+=8)
        a[i][j] += b[j][i];                for (int i = ii; i < ii+8; i++)
                                            for (int j = jj; j < jj+8; j++)
                                                a[i][j] += b[j][i];
```

循环阻塞是优化通用矩阵乘法 (GEMM) 算法的众所周知的方法。它可以提高内存访问的缓存重用率，并改善算法的内存带宽和内存延迟。

通常，工程师会针对每个 CPU 核心专有的缓存大小（英特尔和 AMD 的 L1 或 L2，Apple 的 L1）优化平铺算法。然而，专用缓存的大小会随着每一代发生变化，因此硬编码块大小会带来其自身的挑战。作为替代解决方案，可以使⽤ cache-oblivious: [https://en.wikipedia.org/wiki/Cache-oblivious\\_algorithm<sup>153</sup>](https://en.wikipedia.org/wiki/Cache-oblivious_algorithm) 算法，其⽬标是在任何缓存⼤⼩下都能合理地⼯作。

**循环融合和分配 (裂变) (Loop Fusion and Distribution (Fission)):** 当单独的循环迭代相同范围并且不引用彼此的数据时，可以将它们融合在一起。Listing 153 显示了循环融合的示例。相反的过程称为循环分配 (裂变)，即循环被分成单独的循环。

代码清单: 循环融合和分配

```
for (int i = 0; i < N; i++)          for (int i = 0; i < N; i++) {
    a[i].x = b[i].x;                  a[i].x = b[i].x;
                                         a[i].y = b[i].y;
=>                               }
for (int i = 0; i < N; i++)          }
    a[i].y = b[i].y;
```

循环融合有助于降低循环开销（类似于循环展开），因为两个循环都可以使用相同的归纳变量。此外，循环融合还可以帮助改善内存访问的时间局部性。在 Listing 153 中，如果结构体的  $x$  和  $y$  成员恰好位于同一个缓存线上，那么最好将两个循环融合，因为我们可以避免两次加载相同的缓存线。这将减少缓存占用并提高内存带宽利用率。

但是，循环融合并不总是能提高性能。有时，将循环拆分为多个通道、预过滤数据、排序和重新组织数据等会更好。通过将大型循环分配到多个更小的循环中，我们可以限制每个循环迭代所需的数据量，有效地提高内存访问的时间局部性。这有助于出现高缓存争用情况，通常发生在大循环中。循环分配还可以减少寄存器压力，因为每个循环迭代中执行的操作更少。此外，将大循环分解成多个较小的循环可能会因更好的指令缓存利用率而有利于 CPU 前端的性能。最后，分布后，每个小循环都可以由编译器进一步单独优化。

**循环展开和卡住 (Loop Unroll and Jam):** 要执行此转换，需要首先展开外循环，然后将多个内循环组合在一起，如 Listing 153 所示。此转换增加了内循环的 ILP（指令级并行性），因为内循环中执行了更多独立的指令。在代码示例中，内循环是一个归约操作，它累积数组  $a$  和  $b$  元素之间的差值。当我们将其循环嵌套展开和卡住 2 倍时，我们实际上同时执行初始外循环的 2 次迭代。这一点通过具有 2 个独立的累加器来强调，它打破了初始变体中  $\text{diffs}$  上的依赖链。

代码清单: 循环展开和阻塞

<sup>153</sup> 缓存遗忘算法 - [https://en.wikipedia.org/wiki/Cache-oblivious\\_algorithm](https://en.wikipedia.org/wiki/Cache-oblivious_algorithm)

```

for (int i = 0; i < N; i++)
    for (int j = 0; j < M; j++)
        diffss += a[i][j] - b[i][j]; =>
for (int i = 0; i+1 < N; i+=2)
    for (int j = 0; j < M; j++) {
        diffss1 += a[i][j] - b[i][j];
        diffss2 += a[i+1][j] - b[i+1][j];
    }
diffss = diffss1 + diffss2;

```

当外循环没有跨迭代依赖关系时，可以执行循环展开和卡住，换句话说，内循环的两次迭代可以并行执行。此外，当内循环具有与外循环索引（例如此处的 *i*）步进的内存访问时，这种转换才有意义，否则其他转换可能更适用。循环展开和卡住尤其适用于内循环执行次数较少的情况，例如少于 4 次。通过执行此转换，我们将更多的独立操作打包到内循环中，从而提高 ILP（指令级并行性）。

有时，循环展开和卡住转换对于外循环矢量化非常有用，而矢量化在撰写本文时编译器还无法自动完成。当编译器看不到内循环的执行次数时，它仍然可以矢量化原始内循环，希望它执行足够的迭代次数以命中矢量化代码（有关矢量化的更多信息，请参见下一部分）。但是，如果执行次数很少，程序将使用循环的慢速标量版本。一旦执行循环展开和卡住，我们允许编译器以不同的方式矢量化代码：现在将内循环中的独立指令“粘合”在一起（也称为 SLP 矢量化）。

### 8.3.3 发现循环优化机会

正如我们在本节开头所讨论的，编译器将负责优化循环的繁重工作。您可以依靠它们对循环代码进行所有明显的改进，例如消除不需要的工作、执行各种窥孔优化等。有时编译器足够聪明，可以默认生成快速版本的循环，而其他时候我们必须自己进行一些重写来帮助编译器。正如我们之前所说，从编译器的角度来看，合法且自动地进行循环转换是非常困难的。通常，当编译器无法证明转换的合法性时，它们必须保守。

考虑 Listing 8.3.3 中的代码。编译器无法将表达式 `strlen(a)` 移出循环体外。因此，循环每次迭代都会检查是否到达字符串结尾，这显然很慢。编译器无法提升调用的原因是，数组 *a* 和 *b* 的内存区域可能重叠。在这种情况下，将 `strlen(a)` 移出循环体外是非法的。如果开发人员确信内存区域不会重叠，他们可以使用 `restrict` 关键字声明函数 *foo* 的两个参数，即 `char* __restrict__ a`。

代码清单: 不能将 `strlen` 移出循环

```

void foo(char* a, char* b) {
    for (int i = 0; i < strlen(a); ++i)
        b[i] = (a[i] == 'x') ? 'y' : 'n';
}

```

正如我们在本节开头讨论的那样，编译器将负责优化循环的繁重工作。您可以依靠它们对循环代码进行所有明显的改进，例如消除不需要的工作、进行各种窥孔优化等。有时编译器足够聪明，可以默认生成快速版本的循环，而其他时候我们必须自己进行一些重写来帮助编译器。正如我们之前所说，从编译器的角度来看，合法且自动地进行循环转换是非常困难的。通常，当编译器无法证明转换的合法性时，它们必须保守。

以下是一些有关如何自己发现循环优化机会的提示：

- 找到热点循环：使用性能分析器识别占用大量执行时间的循环。首先关注优化这些循环。
- 分析循环结构：了解循环内部执行的操作类型、内存访问模式和循环执行次数。这将帮助您识别可能有利的循环转换类型。
- 不要害怕尝试：有时，找到最佳循环转换的最好方法是尝试不同的方法并找出效果最好的方法。有许多不同的循环优化技术可用，所以不要害怕尝试找到适合您特定情况的技术。

- 寻找编译器提示: 一些编译器提供提示或警告, 可以帮助您识别潜在的循环优化机会。例如, 编译器可能会警告您一个可以矢量化的循环。

请记住, 循环优化既是一门艺术, 也是一门科学。没有一刀切的解决方案, 优化循环的最佳方法是进行试验并找出最适合您的特定情况的方法。

#### 8.3.4 循环优化框架

多年来, 研究人员开发了确定循环转换合法性和自动转换循环的技术。其中一项发明是多面体框架: [https://en.wikipedia.org/wiki/Loop\\_optimization#The\\_polyhedral\\_or\\_constraint-based\\_framework](https://en.wikipedia.org/wiki/Loop_optimization#The_polyhedral_or_constraint-based_framework)<sup>154</sup>。GRAPHITE: <https://gcc.gnu.org/wiki/Graphite><sup>155</sup> 是最早集成到生产编译器中的多面体工具之一。GRAPHITE 基于从 GCC 的低级中间表示 GIMPLE 提取的多面体信息, 执行一系列经典循环优化。GRAPHITE 证明了该方法的可行性。

后来, LLVM 编译器开发了自己的多面体框架, 称为 Polly: <https://polly.llvm.org/><sup>156</sup>。Polly 是 LLVM 的高级循环和数据局部性优化基础架构。它使用基于整数多面体的抽象数学表示来分析和优化程序的内存访问模式。Polly 执行经典循环转换, 尤其是切分和循环融合, 以改善数据局部性。该框架在许多著名的基准测试上显示了显著的加速 [Grosser et al., 2012]。下面是一个例子, 说明 Polly 如何将来自 Polybench 2.0: <https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/><sup>157</sup> 基准套件的通用矩阵乘法 (GEMM) 内核的运行速度提高了近 30 倍:

```
$ clang -O3 gemm.c -o gemm clang
$ time ./gemm clang
real 0m6.574s
$ clang -O3 gemm.c -o gemm.polly -mllvm -polly
$ time ./gemm.polly
real 0m0.227s
```

Polly 是一个强大的循环优化框架; 但是, 它仍然会错过一些常见和重要的情况。<sup>158</sup> 它没有启用 LLVM 基础架构中的标准优化管道, 并且要求用户提供显式编译器选项才能使用它 (-mllvm -polly)。在寻找加速循环的方法时, 使用多面体框架是一个可行的选择。

## 8.4 向量化

在现代处理器上, 使用 SIMD 指令可以比普通非向量化 (标量) 代码获得更大的速度提升。在进行性能分析时, 软件工程师的首要任务之一是确保代码的热点部分被向量化。本节将指导工程师发现向量化机会。有关现代 CPU 的 SIMD 功能的回顾, 读者可以参考 Section 3.4。

通常, 向量化会自动发生, 无需任何用户干预, 这称为自动向量化。在这种情况下, 编译器会自动识别从源代码生成 SIMD 机器代码的机会。自动向量化可能是一个方便的解决方案, 因为现代编译器可以为各种程序生成快速的向量化代码。

但是, 在某些情况下, 如果没有软件工程师的干预, 自动向量化可能不会成功, 这可能是基于他们从编译器或性能分析数据中获得的反馈<sup>159</sup>。在这种情况下, 程序员需要告诉编译器特定的代码区域是可向量化的, 或者向量化是有益的。现代编译器有一些扩展, 允许高级用户控制自动向量化过程, 并确保代码的某些部分被高效地向量化。但是, 这种控制是有限的。在后续章节中, 我们将看到几个使用编译器提示的例子。

<sup>154</sup> 多面体框架 - [https://en.wikipedia.org/wiki/Loop\\_optimization#The\\_polyhedral\\_or\\_constraint-based\\_framework](https://en.wikipedia.org/wiki/Loop_optimization#The_polyhedral_or_constraint-based_framework).

<sup>155</sup> GRAPHITE 多面体框架 - <https://gcc.gnu.org/wiki/Graphite>.

<sup>156</sup> Polly - <https://polly.llvm.org/>.

<sup>157</sup> Polybench - <https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/>.

<sup>158</sup> Why not Polly? - <https://sites.google.com/site/parallelizationforllvm/why-not-polly>.

<sup>159</sup> 例如, 编译器优化报告, 参见 Section 5.8。

值得注意的是，SIMD 很重要的许多问题领域中，自动向量化根本不起作用，并且在不久的将来也不太可能起作用。读者可以在 [Mula & Lemire, 2019] 中找到一个例子。目前编译器不会尝试外循环自动向量化。它们不太可能向量化浮点代码，因为结果在数值上会有差异。涉及跨向量通道的排列或改组的代码也 less 可能自动向量化，这对于编译器来说可能仍然很困难。

自动向量化还有一个更微妙的问题。随着编译器的不断发展，它们所做的优化也在发生变化。以前编译器版本中成功进行的自动向量化代码，在下一个版本中可能无法工作，反之亦然。此外，在代码维护或重构过程中，代码的结构可能会发生变化，导致自动向量化突然失败。这可能发生在原始软件编写很久之后，因此此时修复或重新实现代码的代价会更高。

如果绝对需要生成特定的汇编指令，就不要依赖编译器的自动向量化。在这种情况下，可以使用编译器内在函数编写代码，我们将在 Section 8.5 中讨论这一点。在大多数情况下，编译器内在函数提供 1 对 1 映射到汇编指令。内在函数比内联汇编更容易使用，因为编译器会处理寄存器分配，并且允许程序员对代码生成保持相当程度的控制。然而，它们仍然经常冗长且难以阅读，并且容易出现不同编译器之间行为差异甚至错误。

在低成本但不可预测的自动向量化和冗长/不可读但可预测的内在函数之间，可以使用围绕内在函数的包装库。这些库往往更易读，可以将编译器修复集中在库中，而不是分散在用户代码中，并且仍然允许开发人员控制生成的代码。许多这样的库存存在，它们支持最新或“奇特”操作的范围以及它们支持的平台数量各不相同。据我们所知，Highway 是目前唯一完全支持可扩展向量（如 SVE 和 RISC-V V 指令集）的库。需要注意的是，本文作者之一是该库的技术负责人。它将在 Section 8.5 中介绍。

请注意，使用内在函数或包装库时，仍然建议使用 C++ 编写初始实现。这样可以通过将原始代码的结果与新的向量化实现进行比较，从而快速原型化和验证正确性。

在本节的其余部分，我们将讨论几种方法，尤其是内循环向量化，因为它是自动向量化最常见的类型。其他两种类型，外循环向量化和 SLP (Superword-Level Parallelism) 向量化，将在附录 B 中提到。

#### 8.4.1 编译器自动向量化

多种障碍可能会阻碍自动向量化，其中一些与编程语言的语义固有相关。例如，编译器必须假设无符号循环索引可能会溢出，这可能会阻止某些循环转换。另一个例子是 C 编程语言所做的假设：程序中的指针可能指向重叠的内存区域，这可能会使程序分析非常困难。另一个主要障碍是处理器的设计本身。在某些情况下，处理器没有针对某些操作的高效向量指令。例如，大多数处理器上都无法执行基于谓词 (bitmask 控制) 的加载和存储操作。另一个例子是向量宽度格式在有符号整数和双精度浮点数之间的转换，因为结果在不同大小的向量寄存器上进行操作。尽管存在所有挑战，软件开发人员可以克服许多挑战并启用向量化。稍后在本节中，我们将提供有关如何与编译器协作并确保热门代码被编译器向量化的指南。

向量器通常由三个阶段组成：合法性检查、盈利性检查和转换本身：

- **合法性检查：**在此阶段，编译器检查将循环（或其他类型的代码区域）转换为使用向量的合法性。循环向量器检查循环的迭代是否连续，这意味着循环线性进行。向量器还确保循环中的所有内存和算术运算都可以扩展为连续运算。循环的控制流在所有通道上都是一致的，并且内存访问模式也是一致的。编译器需要检查或确保生成的代码不会触及不应该触及的内存，并且操作顺序将被保留。编译器需要分析指针的可能范围，如果缺少一些信息，它必须假设转换是非法的。合法性阶段收集了向量化循环合法的必要条件列表。
- **盈利性检查：**接下来，向量器检查转换是否有利可图。它比较不同的向量化因素，并找出哪个向量化因素执行速度最快。向量器使用成本模型来预测不同操作的成本，例如标量加法或向量加载。它需要考虑将数据洗牌到寄存器的附加指令，预测寄存器压力，并估计循环保护的成本，这些保护确保满足允许向量化的前提条件。检查盈利性的算法很简单：1) 累加代码中所有操作的成本，2) 比较每个代码版本的成本，3) 将成本除以预期的执行次数。例如，如果标量代码花费 8 个周期，而矢量化代码花费 12 个周期，但一次执行 4 次循环迭代，那么矢量化版本的循环可能更快。

- 转换：最后，当向量器确定转换合法且有利可图时，它会转换代码。此过程还包括插入启用向量化的守卫。例如，大多数循环使用未知的迭代次数，因此编译器必须生成循环的标量版本和矢量化版本，以处理最后几个迭代。编译器还必须检查指针是否重叠等等。所有这些转换都使用在合法性检查阶段收集的信息完成。

#### 8.4.2 发现向量化机会

阿姆达尔定律<sup>160</sup>告诉我们，我们应该只花时间分析程序执行过程中使用最多的代码部分。因此，性能工程师应该关注由分析工具突出显示的代码热点部分。正如前面提到的，向量化最常应用于循环。

发现改善向量化的机会应该从分析程序中的热点循环开始，并检查编译器执行了哪些优化。查看编译器向量化备注（参见 Section 5.8）是最简单的方法。现代编译器可以报告某个循环是否已矢量化，并提供其他详细信息，例如矢量化因子（VF）。如果编译器无法矢量化循环，它还可以告诉原因。

使用编译器优化报告的另一种方法是检查汇编输出。最好分析来自分析工具的输出，该工具显示给定循环的源代码和生成的汇编指令之间的对应关系。这样，您只关注重要的代码，即热点代码。但是，理解汇编语言比像 C++ 这样的高级语言要困难得多。弄清楚编译器生成的指令的语义可能需要一些时间。但这项技能非常有益，并且经常提供有价值的见解。经验丰富的开发人员只需查看指令助记符和这些指令使用的寄存器名称，就可以快速判断代码是否已矢量化。例如，在 x86 ISA 中，向量指令操作打包数据（因此在其名称中带有“P”），并使用“XMM”、“YMM”或“ZMM”寄存器，例如“VMULPS XMM1, XMM2, XMM3”将“XMM2”和“XMM3”中的四个单精度浮点数相乘并将结果保存在“XMM1”中。但要小心，人们经常看到使用“XMM”寄存器，就认为它是向量代码——不一定。例如，“VMULSS”指令只会乘以一个单精度浮点数，而不是四个。

开发人员在尝试加速可矢量化代码时经常会遇到一些常见情况。下面我们介绍四种典型场景，并针对每种情况提供一般指导。

**8.4.2.1 向量化是非法的** 在某些情况下，遍历数组元素的代码根本不可矢量化。向量化备注非常有效地解释了出了什么问题以及为什么编译器无法矢量化代码。Listing 161 显示了一个循环内部的依赖关系，该依赖关系阻止了向量化<sup>161</sup>。

代码清单: 向量化: 读写后写依赖。

```
void vectorDependence(int *A, int n) {
    for (int i = 1; i < n; i++)
        A[i] = A[i-1] * 2;
}
```

一些循环由于上述硬件限制无法矢量化，但另一些循环在放松特定约束时可以矢量化。有时，编译器无法矢量化循环仅仅是因为它无法证明这样做是合法的。编译器通常非常保守，只会确信不会破坏代码时才进行转换。此类软限制可以通过向编译器提供额外的提示来放松。

例如，当转换执行浮点运算的代码时，矢量化可能会改变程序的行为。浮点加法和乘法是交换的，这意味着您可以交换左手侧和右手侧而不改变结果： $(a + b == b + a)$ 。但是，这些操作不是关联的，因为舍入发生在不同的时间： $((a + b) + c) != (a + (b + c))$ 。Listing 161 中的代码无法由编译器自动矢量化。原因是矢量化会将变量 sum 更改为矢量累加器，这将改变运算顺序，并可能导致不同的舍入决策和不同的结果。

代码清单: 向量化: 浮点运算。

```
1 // a.cpp
2 float calcSum(float* a, unsigned N) {
```

<sup>160</sup> 阿姆达尔定律 - [https://en.wikipedia.org/wiki/Amdahl's\\_law](https://en.wikipedia.org/wiki/Amdahl's_law)。

<sup>161</sup> 一旦展开几个循环迭代，很容易发现读后写依赖关系。请参见 Section 5.8 中的示例。

```

3   float sum = 0.0f;
4   for (unsigned i = 0; i < N; i++) {
5     sum += a[i];
6   }
7   return sum;
8 }
```

然而，如果程序可以容忍最终结果有一点轻微的不准确（通常情况下是这样），我们可以将此信息传达给编译器以启用矢量化。Clang 和 GCC 编译器都有一个标志 `-ffast-math`<sup>162</sup>，它允许这种转换：

```

$ clang++ -c a.cpp -O3 -march=core-avx2 -Rpass-analysis=.*
...
a.cpp:5:9: remark: loop not vectorized: cannot prove it is safe to reorder floating-point
operations; allow reordering by specifying '#pragma clang loop vectorize(enable)' before the
loop or by providing the compiler option '-ffast-math'. [-Rpass-analysis=loop-vectorize]
...
$ clang++ -c a.cpp -O3 -ffast-math -Rpass=.*
...
a.cpp:4:3: remark: vectorized loop (vectorization width: 4, interleaved count: 2)
[-Rpass=loop-vectorize]
...
```

不幸的是，此标志涉及微妙且潜在危险的行为变化，包括非数字 (NaN)、带符号零、无穷大和次正规数。由于第三方代码可能还没有准备好应对这些影响，因此不应在不仔细验证结果（包括边缘情况）的情况下在大段代码中启用此标志。

让我们看另一个典型情况，编译器可能需要开发人员的支持才能执行矢量化。当编译器无法证明循环操作的是具有非重叠内存区域的数组时，它们通常会选择更安全的选项。让我们重新审视 Section 5.8 中 Listing 5.8 提供的示例。当编译器尝试矢量化 Listing 162 中呈现的代码时，它通常无法做到这一点，因为数组 a、b 和 c 的内存区域可能重叠。

代码清单: a.c

```

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

这是由 GCC 10.2 提供的优化报告（使用 `-fopt-info` 启用）：

```

$ gcc -O3 -march=core-avx2 -fopt-info
a.cpp:2:26: optimized: loop vectorized using 32 byte vectors
a.cpp:2:26: optimized: loop versioned for vectorization because of possible aliasing
```

GCC 识别到数组 a、b 和 c 的内存区域可能存在重叠，并创建了相同循环的多个版本。编译器插入了运行时检查<sup>163</sup>来检测内存区域是否重叠。基于这些检查，它会在矢量化和标量<sup>164</sup>版本之间进行调度。在这种情况下，矢量化会带

<sup>162</sup> 编译器标志 `-Ofast` 启用 `-ffast-math` 以及 `-O3` 编译模式。

<sup>163</sup> 请参阅 easyperf 博客上的示例: [https://easyperf.net/blog/2017/11/03/Multiversioning\\_by\\_DD.](https://easyperf.net/blog/2017/11/03/Multiversioning_by_DD.)

<sup>164</sup> 但循环的标量版本仍然可以展开。

来插入可能代价高昂的运行时检查的成本。

如果开发人员知道数组 a、b 和 c 的内存区域不会重叠，可以在循环前面插入 `#pragma GCC ivdep165` 或使用 `__restrict__` 关键字，如 Listing 5.8 所示。此类编译器提示将消除 GCC 编译器插入上述运行时检查的需要。

编译器本质上是静态工具：它们只根据所使用的代码进行推理。例如，一些动态工具（例如 Intel Advisor）可以检测跨迭代依赖或访问具有重叠内存区域的数组等问题是否确实出现在给定的循环中。但要记住，这类工具只提供建议。不加思索地插入编译器提示可能会导致实际问题。

**8.4.2.2 向量化无益** 在某些情况下，编译器可以矢量化循环，但认为这样做没有好处。在 Listing 8.4.2.2 中呈现的代码中，编译器可以矢量化对数组 A 的内存访问，但需要将对数组 B 的访问拆分为多个标量加载。这种分散/收集模式相对昂贵，并且能够模拟操作成本的编译器经常会决定避免矢量化具有这种模式的代码。

代码清单：向量化：没有好处。

```
1 // a.cpp
2 void stridedLoads(int *A, int *B, int n) {
3     for (int i = 0; i < n; i++)
4         A[i] += B[i * 3];
5 }
```

下面是 Listing 8.4.2.2 中的代码的编译器优化报告：

```
$ clang -c -O3 -march=core-avx2 a.cpp -Rpass-missed=loop-vectorize
a.cpp:3:3: remark: the cost-model indicates that vectorization is not beneficial
      [-Rpass-missed=loop-vectorize]
      for (int i = 0; i < n; i++)
      ^

```

用户可以使用 `#pragma` 提示强制 Clang 编译器矢量化循环，如 Listing 166 所示。但是，请记住，矢量化是否真正有利很大程度上取决于运行时数据，例如循环的迭代次数。编译器无法获得这些信息，<sup>166</sup> 因此它们往往保守行事。开发人员可以在寻找性能提升空间时使用此类提示。

代码清单：向量化：没有好处。

```
1 // a.cpp
2 void stridedLoads(int *A, int *B, int n) {
3 #pragma clang loop vectorize(enable)
4     for (int i = 0; i < n; i++)
5         A[i] += B[i * 3];
6 }
```

开发人员应注意使用矢量化代码的隐藏成本。使用 AVX 和特别是 AVX-512 向量指令可能会导致频率降频或启动开销，在某些 CPU 上还会影响后续代码数微妙。矢量化部分的代码应该足够热门，以证明使用 AVX-512 的合理性。<sup>167</sup> 例如，排序 80 KiB 的数据被发现足以摊销这种开销，使矢量化变得有价值。<sup>168</sup>

<sup>165</sup> 它是 GCC 特定的编译器指令。对于其他编译器，请查阅相应的手册。

<sup>166</sup> 除了配置文件引导优化（参见 Section 10.5）。

<sup>167</sup> 有关更多细节，请阅读这篇博客文章：<https://travisdowns.github.io/blog/2020/01/17/avxfreq1.html>。

<sup>168</sup> AVX-512 降频研究：在 VQSort README 中：<https://github.com/google/highway/blob/master/hwy/contrib/sort/README.md#study-of-avx-512-downclocking>

**8.4.2.3 循环已矢量化，但使用标量版本** 在某些情况下，编译器可以成功地矢量化代码，但矢量化代码不会显示在性能分析器中。检查循环的对应汇编时，通常很容易找到循环体的矢量化版本，因为它使用了向量寄存器（程序其他部分不常用），并且代码是展开的，并填充了检查和多个版本以启用不同的边缘情况。

如果生成的代码没有执行，一个可能的原因是编译器生成的代码假设循环执行次数高于程序使用的次数。例如，要在现代 CPU 上高效地进行矢量化，程序员需要矢量化并利用 AVX2，并且还要将循环展开 4-5 次，为管道化的 FMA 单元生成足够的工作。这意味着每次循环迭代都需要处理大约 40 个元素。许多循环可能运行的循环执行次数低于这个值，并可能退回到使用标量剩余循环。很容易检测这些情况，因为标量剩余循环会在性能分析器中亮起，而矢量化代码将保持冷状态。

解决这个问题的方法是强制矢量器使用较低的矢量化因子或展开计数，以减少循环处理的元素数量，并使更多具有较低执行次数的循环访问快速矢量化的循环体。开发人员可以使用 `#pragma` 提示来实现这一点。对于 Clang 编译器，可以使用 `#pragma clang loop vectorize_width(N)`，如 easyperf 博客上的文章<sup>169</sup> 所示。

**8.4.2.4 循环以次优方式矢量化** 当您看到一个循环被矢量化并在运行时执行时，该部分程序可能已经运行良好。但是，也有一些例外情况。有时，人类专家可以编写出性能优于编译器生成的代码。

由于以下几个因素，最佳矢量化因子可能并不直观。首先，人类很难在脑海中模拟 CPU 的操作，除了尝试多种配置之外别无其他方法。涉及多个向量通道的向量洗牌可能比预期的更昂贵或更便宜，这取决于许多因素。其次，在运行时，程序可能会表现出不可预测的行为，这取决于端口压力和许多其他因素。这里的建议是尝试强制矢量器选择一个特定的矢量化因子和展开因子，并测量结果。矢量化编译器指令可以帮助用户枚举不同的矢量化因子并找出性能最佳的因子。每个循环可能的配置相对较少，并且在典型输入上运行循环是人类可以做到的，而编译器却做不到。

最后，有时循环的标量非矢量化版本比矢量化版本性能更好。这可能是因为使用了昂贵的向量操作，例如“gather/scatter”加载、掩码、洗牌等，编译器必须使用这些操作才能进行矢量化。性能工程师也可以尝试通过不同的方式禁用矢量化。对于 Clang 编译器，可以通过编译器选项 `-fno-vectorize` 和 `-fno-slp-vectorize` 或针对特定循环的提示（例如，`#pragma clang loop vectorize(enable)`）来实现。

**8.4.2.5 使用显式矢量化的语言** 矢量化还可以通过用专用于并行计算的编程语言重写程序的部分来实现。这些语言使用特殊的结构和对程序数据的了解，将代码高效地编译成并行程序。最初，这些语言主要用于将工作卸载到特定处理单元，例如图形处理单元 (GPU)、数字信号处理器 (DSP) 或现场可编程门阵列 (FPGA)。然而，其中一些编程模型也能够针对您的 CPU（例如 OpenCL 和 OpenMP）。

其中一种这样的并行语言是 Intel® Implicit SPMMD 程序编译器 (ISPC)(<https://ispc.github.io/>)<sup>170</sup>，我们将在本节稍作介绍。ISPC 语言基于 C 编程语言，并使用 LLVM 编译器基础架构为许多不同的架构生成优化代码。ISPC 的关键特性是“接近金属”的编程模型和跨 SIMD 架构的性能可移植性。它要求从编写程序的传统思维方式转变，但为程序员提供了更多控制 CPU 资源利用率的手段。

ISPC 的另一个优点是其互操作性和易用性。ISPC 编译器生成标准对象文件，可以与传统 C/C++ 编译器生成的代码链接。ISPC 代码可以很容易地插入任何原生项目，因为用 ISPC 编写的函数可以像 C 代码一样调用。

Listing 170 显示了我们之前在 Listing 161 中介绍的一个简单函数，用 ISPC 重写。ISPC 考虑程序将在并行实例中运行，基于目标指令集。例如，当使用 SSE 与 `ffloat` 一起使用时，它可以并行计算 4 个操作。每个程序实例将在 `i` 的向量值上操作，分别是  $(0, 1, 2, 3)$ 、然后是  $(4, 5, 6, 7)$ ，以此类推，一次有效地计算 4 个和。正如您所看到的，使用了一些非典型 C 和 C++ 的关键字：

- `export` 关键字意味着该函数可以从 C 兼容的语言调用。
- `uniform` 关键字意味着变量在程序实例之间共享。

<sup>169</sup> 使用 Clang 的优化编译器指令 - [https://easyperf.net/blog/2017/11/09/Multiversioning\\_by\\_trip\\_counts](https://easyperf.net/blog/2017/11/09/Multiversioning_by_trip_counts)

<sup>170</sup> ISPC 编译器: <https://ispc.github.io/>.

- `varying` 关键字意味着每个程序实例都有自己的局部变量副本。
- `foreach` 与经典的 `for` 循环相同，除了它会将工作分配到不同的程序实例。

代码清单:ISPC 版本的数组元素求和。

```
export uniform float calcSum(const uniform float array[],
                             uniform ptrdiff_t count)
{
    varying float sum = 0;
    foreach (i = 0 ... count)
        sum += array[i];
    return reduce_add(sum);
}
```

由于函数 `calcSum` 必须返回单个值（一个 `uniform` 变量），而我们的 `sum` 变量是一个 `varying`，因此我们需要使用 `reduce_add` 函数来收集每个程序实例的值。ISPC 还负责生成剥离和剩余循环，以考虑未正确对齐或不是向量宽度倍数的数据。

**“接近硬件”编程模型:** 传统 C 和 C++ 语言的一个问题是编译器并不总是矢量化代码的关键部分。通常情况下，程序员会使用编译器内部函数（参见 Section 8.5），这绕过了编译器自动矢量化，但通常很困难，并且需要在出现新指令集时进行更新。ISPC 通过默认假设每个操作都是 SIMD 来帮助解决这个问题。例如，ISPC 语句 `sum += array[i]` 被隐含地认为是一个 SIMD 操作，可以并行进行多次加法运算。ISPC 不是一个自动矢量化编译器，它不会自动发现矢量化机会。由于 ISPC 语言与 C 非常相似，它比使用内部函数好得多，因为它可以让您专注于算法而不是低级指令。此外，据报道，它在性能方面与手写的内部函数代码相匹配或优于手写的内部函数代码<sup>171</sup>。

**性能可移植性:** ISPC 可以自动检测您 CPU 的特性，以充分利用所有可用资源。程序员可以编写一次 ISPC 代码，并编译到许多向量指令集，例如 SSE4、AVX 和 AVX2。ISPC 还可以为不同的架构（如 x86 CPU、ARM NEON）生成代码，并具有实验性的 GPU 卸载支持。

## 8.5 使用编译器内部函数

某些类型的应用程序具有值得重点调整的热点。但是，对于这些热点生成的代码，编译器并不总是按照我们想要的去做。例如，程序在循环中进行一些计算，而编译器以次优方式对其进行矢量化。这通常涉及一些棘手的或专门的算法，我们可以为其设计更好的指令序列。使用 C 和 C++ 语言的标准结构来让编译器生成所需的汇编代码可能非常困难甚至不可能。

希望我们可以强制编译器生成特定的汇编指令，而无需编写低级汇编语言。为了实现这一点，可以使用编译器内部函数，编译器内部函数会转换成特定的汇编指令。内部函数与内联汇编具有同样的好处，而且还提高了代码的可读性，允许编译器进行类型检查，辅助指令调度，并有助于减少调试。Listing 8.5 中的示例展示了如何通过编译器内部函数（函数 `bar`）对函数 `foo` 中的相同循环进行编码。

代码清单: 编译器内部函数

```
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
```

<sup>171</sup> 使用 SIMD 内部函数的虚幻引擎的一些部分使用 ISPC 重写，从而获得了速度提升: <https://software.intel.com/content/www/us/en/develop/articles/unreal-engines-new-chaos-physics-system-screams-with-in-depth-intel-cpu-optimizations.html>

```

6 #include <xmmmintrin.h>
7
8 void bar(float *a, float *b, float *c, unsigned N) {
9     __m128 rA, rB, rC;
10    int i = 0;
11    for (; i + 3 < N; i += 4){
12        rA = _mm_load_ps(&a[i]);
13        rB = _mm_load_ps(&b[i]);
14        rC = _mm_add_ps(rA,rB);
15        _mm_store_ps(&c[i], rC);
16    }
17    for (; i < N; i++) // remainder
18        c[i] = a[i] + b[i];
19 }

```

当编译为 SSE 目标时, `foo` 和 `bar` 都将生成类似的汇编指令。但是, 有几个注意事项。首先, 当依赖自动矢量化时, 编译器会插入所有必要的运行时检查。例如, 它将确保有足够的元素来填充向量执行单元。其次, 函数 `foo` 将有一个处理循环剩余部分的标量版本作为备用。最后, 大多数向量内部函数假设数据是对齐的, 因此为 `bar` 生成 `movaps` (对齐加载), 而为 `foo` 生成 `movups` (未对齐加载)。考虑到这一点, 开发人员使用编译器内部函数时必须自己注意安全方面。

使用非可移植的平台特定内部函数编写代码时, 开发人员还应该为其他架构提供备选项。有关英特尔平台的所有可用内部函数列表, 请参见此参考: <https://software.intel.com/sites/landingpage/IntrinsicsGuide/><sup>172</sup>。

### 8.5.1 内部函数的包装库

ISPC 的一次编写, 多目标运行模式很有吸引力。然而, 我们可能希望更紧密地集成到 C++ 程序中, 例如与模板互操作, 或者避免单独的构建步骤并使用相同的编译器。相反, 内部函数提供更多的控制, 但开发成本更高。

我们可以结合两者的优势并避免这些缺点, 使用所谓的嵌入式领域特定语言, 其中向量操作表示为普通的 C++ 函数。您可以将这些函数视为“可移植的内部函数”, 例如 `Add` 或 `LoadU`。即使多次编译您的代码 (每个指令集一次), 也可以在普通 C++ 库中完成, 方法是使用预处理器在不同的编译器设置下“重复”您的代码, 但位于独特的命名空间中。一个例子是之前提到的 Highway 库, <sup>173</sup> 它只要求 C++11 标准。

与 ISPC 一样, Highway 也支持检测最佳可用指令集, 这些指令集分为“簇”, 在 x86 上对应于 Intel Core (S-SSE3)、Nehalem (SSE4.2)、Haswell (AVX2)、Skylake (AVX-512) 或 Icelake/Zen4 (AVX-512 with extensions)。然后它从相应命名空间调用您的代码。

与内部函数不同, 代码保持可读性 (每个函数没有前缀/后缀) 和可移植性。

代码清单: 对数组元素求和的高速版本。

```

#include <hwy/highway.h>

float calcSum(const float* HWY_RESTRICT array, size_t count) {
    const ScalableTag<float> d; // type descriptor; no actual data
    auto sum = Zero(d);
    size_t i = 0;

```

<sup>172</sup> 英特尔内部函数指南 - <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>。

<sup>173</sup> Highway 库: <https://github.com/google/highway>

```

for ( ; i + Lanes(d) <= count; i += Lanes(d)) {
    sum = Add(sum, LoadU(d, array + i));
}
sum = Add(sum, MaskedLoad(FirstN(d, count - i), d, array + i));
return ReduceSum(d, sum);
}

```

注意在循环处理向量大小 `Lanes(d)` 的倍数之后的显式余数处理。虽然这更加冗长，但它使实际发生的事情变得清晰可见，并允许进行优化，例如覆盖最后一个向量而不是依赖于 `MaskedLoad`<sup>174</sup>，甚至当已知 `count` 是向量大小的倍数时完全跳过余数。

Highway 支持超过 200 个操作，可以分为以下类别：

- 初始化
- 获取/设置通道
- 获取/设置块
- 打印
- 元组
- 算术
- 逻辑
- 掩码
- 比较
- 内存
- 缓存控制
- 类型转换
- 组合
- 旋转/排列
- 在 128 位块内旋转
- 减少
- 加密

有关操作的完整列表，请参阅其文档<sup>174</sup> 和 FAQ: <https://github.com/google/highway/blob/master/g3doc/faq.md>。您也可以在线编译器探索器: <https://gcc.godbolt.org/z/zP7MYe9Yf> 中进行试验。

其他库包括 Eigen、nsimd、SIMDe、VCL 和 xsimd。需要注意的是，C++ 标准化工作从 Vc 库开始，导致了 `std::experimental::simd`，但它提供了一组非常有限的操作，截至撰写本文时仅在 GCC 11 编译器上支持。

## 问题和练习

1. 使用本章讨论的技术解决以下实验作业：
  - `perf-ninja::function_inlining_1`
  - `perf-ninja::vectorization 1 & 2`
  - `perf-ninja::dep_chains 1 & 2`
  - `perf-ninja::compiler_intrinsics 1 & 2`
  - `perf-ninja::loop_interchange 1 & 2`
  - `perf-ninja::loop_tiling_1`
2. 描述您将采取哪些步骤来找出应用程序是否利用了所有利用 SIMD 代码的机会？

<sup>174</sup> Highway 快速参考 - [https://github.com/google/highway/blob/master/g3doc/quick\\_reference.md](https://github.com/google/highway/blob/master/g3doc/quick_reference.md)

3. 尝试在实际代码上手动进行循环优化（但不要提交）。确保所有测试仍然通过。
4. 假设您正在处理一个 IpCall（每次调用指令）指标非常低的应用程序。您将尝试应用/强制哪些优化？
5. 每天运行您正在使用的应用程序。找到程序中最热的循环。它是矢量化的吗？可以强制编译器自动矢量化吗？  
附加问题：循环是由依赖链还是执行吞吐量导致瓶颈？

## 章节总结

- 低效的计算是现实世界应用程序中瓶颈的重要部分。现代编译器通过执行许多不同的代码转换，非常擅长消除不必要的计算开销。尽管如此，我们仍然有很大的机会做得比编译器提供的更好。
- 在 Section ?? 中，我们展示了如何通过强制某些代码优化来搜索程序中的性能提升空间。我们讨论了诸如函数内联、循环优化和矢量化等流行的转换。

---

## 9 优化分支预测

到目前为止，我们一直在讨论优化内存访问和计算。然而，还有另一个重要的性能瓶颈类别，我们尚未讨论。它与推测执行有关，这是现代高性能 CPU 核心中普遍存在的一项功能。为了提醒你，可以参考 Section 3.3.3 中我们讨论了如何利用推测执行来提高性能。在本章中，我们将探讨减少分支预测错误次数的技术。

一般来说，现代处理器非常擅长预测分支结果。它们不仅遵循静态预测规则，还检测动态模式。通常，分支预测器保存先前分支结果的历史记录，并尝试猜测下一个结果。然而，当模式变得难以跟踪时，CPU 分支预测器可能会影响性能。

当分支预测错误时，会导致显著的速度惩罚。当这种事件经常发生时，CPU 需要清除所有预测性工作，后来证明是错误的。它还需要清空流水线，并开始填充正确路径的指令。通常，现代 CPU 由于分支预测错误而经历 10 到 20 个周期的惩罚。准确的周期数取决于微架构设计，即流水线的深度和从错误预测中恢复的机制。

分支预测器使用缓存和历史寄存器，因此容易受到与缓存相关的问题的影响，即三个 C：

- 强制缺失：当使用静态预测并且没有动态历史记录可用时，可能会在分支的第一次动态出现时发生预测错误。
- 容量缺失：由于程序中分支数量非常高或动态模式过长而导致的预测错误。
- 冲突缺失：分支被映射到缓存桶（关联集）中，使用它们的虚拟和/或物理地址的组合。如果太多活动分支映射到同一集合，则可能丢失历史记录。另一个冲突缺失的实例是错误共享，当两个独立分支被映射到同一个缓存条目并相互干扰时，可能会降低预测历史记录。

程序总会发生非零数量的分支预测错误。您可以通过查看 TMA 的 Bad Speculation 指标来了解程序受分支预测错误的影响程度。对于通用应用程序来说，Bad Speculation 指标在 5-10% 的范围内是正常的。我们建议一旦该指标超过 10%，就要密切关注。

由于分支预测器擅长发现模式，因此以前用于优化分支预测的建议不再适用。过去，开发人员可以通过在分支指令的编码前缀中提供预测提示来为处理器提供选择 (0x2E: Branch Not Taken, 0x3E: Branch Taken)。这可能会提高旧微架构（如 Pentium 4）的性能。虽然在现代处理器上仍然使用这些分支前缀会生成有效的 x86/x64 汇编，但不会在现代处理器上产生性能提升。[TODO]

减少分支预测错误的一种间接方法是使用基于源代码和编译器的技术来简化代码。PGO 和 BOLT 通过提高顺序执行率来减少分支预测错误，从而缓解了分支预测器结构的压力。我们将在下一章中讨论这些技术。

因此，也许唯一直接消除分支预测错误的方法是消除分支本身。在接下来的两个小节中，我们将看看如何用查找表和预测来替代分支。

有一个常规智慧认为，从预测的角度来看，从不被采取的分支对分支预测是透明的，不会影响性能，因此删除它们并没有太多意义。然而，与这种智慧相反的是，BOLT 优化器的作者进行的一项实验表明，在大型代码库应用程序（例如 Clang C++ 编译器）中，将从不被采取的分支替换为相同大小的空操作，可以在现代 Intel CPU 上带来约 5% 的加速。因此，尽管还是值得尝试消除所有分支。

### 9.1 用查找表替换分支

避免频繁出现分支预测错误的一种方法是使用查找表。在 Listing 9.1 中展示了一个可能会受益于这种转换的代码示例。与往常一样，原始版本在左侧，改进版本在右侧。函数 `mapToBucket` 将 [0-50) 范围内的值映射到对应的五个桶中，并对超出此范围的值返回 -1。对于均匀分布的 `v` 值，`v` 有相等的概率落入任何一个桶中。在原始版本的生成的汇编中，我们可能会看到许多分支，这可能会导致高的分支预测错误率。希望可以通过单个数组查找来重写函数 `mapToBucket`，如右侧所示。

代码清单: 用查找表替换分支。

```
int8_t mapToBucket(unsigned v) {
    if (v >= 0 && v < 10) return 0;
    if (v >= 10 && v < 20) return 1;
    if (v >= 20 && v < 30) return 2;      =>
    if (v >= 30 && v < 40) return 3;
    if (v >= 40 && v < 50) return 4;
    return -1;
}

int8_t buckets[50] = {
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
    3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4};

int8_t mapToBucket(unsigned v) {
    if (v < (sizeof(buckets) / sizeof(int8_t)))
        return buckets[v];
    return -1;
}
```

对于右侧改进版本的 `mapToBucket`, 编译器很可能会生成一个单个分支指令, 以防止对 `buckets` 数组进行越界访问。这个函数的典型热路径将执行未采取的分支和一个加载指令。由于我们期望大多数输入值都落在 `buckets` 数组覆盖的范围内, CPU 分支预测器将很好地预测这个分支。查找速度也很快, 因为 `buckets` 数组很小且很可能位于 L1-d 缓存中。

如果我们需要映射一个更大范围的值, 比如 [0-1M], 分配一个非常大的数组是不现实的。在这种情况下, 我们可以使用区间映射数据结构, 它们可以使用更少的内存来实现这个目标, 但查找复杂度是对数级的。读者可以在 [Boost<sup>175</sup>](#) 和 [LLVM<sup>176</sup>](#) 找到现有的区间映射容器实现。

## 9.2 用算术替换分支

在某些情况下, 可以用算术来替换分支。如 Listing 9.1 中的代码, 也可以使用简单的算术公式重写, 如 Listing 9.2 所示。对于这段代码, Clang-17 编译器用更便宜的乘法运算替换了昂贵的除法运算。

代码清单: 用算术替换分支。

```
int8_t mapToBucket(unsigned v) {
    constexpr unsigned BucketRangeMax = 50;
    if (v < BucketRangeMax)
        return v / 10;
    return -1;
}
```

截至 2023 年, 编译器通常无法自行找到这些捷径, 因此由程序员手动完成。如果你能找到用算术替换分支的方法, 你很可能会看到性能改进。不幸的是, 这并不总是可能的。

## 9.3 用谓词替换分支

某些分支可以通过执行分支的两部分, 然后选择正确的结果 (谓词) 来有效地消除。当这种转换可能有利可图时, 代码示例显示在 Listing 9.3 中。如果 TMA 提示 `if (cond)` 分支具有非常高的误判率, 您可以尝试通过执行右侧显示的转换来消除分支。

代码清单: 谓词分支。

<sup>175</sup> C++ Boost interval\_map - [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>176</sup> LLVM's IntervalMap - [https://llvm.org/doxygen/IntervalMap\\_8h\\_source.html](https://llvm.org/doxygen/IntervalMap_8h_source.html)

```

int a;
if (cond) { /* frequently mispredicted */ =>
    a = computeX();
} else {
    a = computeY();
}

```

```

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

```

对于右侧的代码，编译器可以替换来自三元运算符的分支，并生成 CMOVx86 指令。CMOVcc 指令检查 EFLAGS 寄存器 (CF、OF、PF、SF 和 ZF) 中一个或多个状态标志的状态，并在标志处于特定状态或条件下执行移动操作。可以使用 FCMOVcc, VMAXSS/VMINSS 指令对浮点数字进行类似的转换。Listing 9.3 显示了原始版本和无分支版本的汇编列表。

代码清单：谓词分支 - x86 汇编代码。

| # original version         | # branchless version                                 |
|----------------------------|------------------------------------------------------|
| 400504: test edi,edi       | 400537: mov eax,0x0                                  |
| 400506: je 400514          | 40053c: call <computeX> # compute x; a = x           |
| 400508: mov eax,0x0        | 400541: mov ebp,eax # ebp = x                        |
| 40050d: call <computeX> => | 400543: mov eax,0x0                                  |
| 400512: jmp 40051e         | 400548: call <computeY> # compute y; a = y           |
| 400514: mov eax,0x0        | 40054d: test ebx,ebx # test cond                     |
| 400519: call <computeY>    | 40054f: cmovne eax,ebp # override a with x if needed |
| 40051e: mov edi,eax        |                                                      |

与原始版本相比，无分支版本没有跳转指令。然而，无分支版本独立计算 x 和 y，然后选择其中一个值并丢弃另一个值。虽然这种转换消除了分支预测错误的惩罚，但它可能比原始代码做了更多的工作。在这种情况下，性能改进在很大程度上取决于 computeX 和 computeY 函数的特性。如果函数很小，编译器能够内联它们，那么它可能会带来明显的性能提升。如果函数很大，执行这两个函数的成本可能比承担分支预测错误的成本更低。

需要注意的是，谓词并不总是能提高应用程序的性能。谓词的问题在于它限制了 CPU 的并行执行能力。对于原始版本的代码，CPU 可以预测分支将被取走，推测性地调用 computeX 并继续执行程序的其余部分。对于无分支版本，这种类型的推测是不可能的，因为 CPU 必须等待 CMOVNE 指令的结果才能继续进行。

在选择代码的常规版本和无分支版本之间进行权衡时，典型的例子是二分查找<sup>177</sup>：

- 对于无法放入 CPU 缓存的大型数组的搜索，基于分支的二分查找版本性能更好，因为分支预测错误的惩罚与内存访问的延迟相比很低（由于缓存未命中，延迟很高）。由于存在分支，CPU 可以推测其结果，从而允许同时从当前迭代和下一个迭代加载数组元素。它并没有就此结束：推测仍在继续，您可能同时有多个加载正在进行。
- 对于适合 CPU 缓存的小型数组，情况则相反。正如前面所解释的，无分支搜索仍然将所有内存访问序列化。但这一次，加载延迟很小（只有少数周期），因为数组适合 CPU 缓存。基于分支的二分查找会不断遭受误判，其成本大约为 10-20 个周期。在这种情况下，误判的成本远高于内存访问的成本，因此推测执行的优势会受到阻碍。无分支版本通常在这种情况下更快。

二分查找是一个很好的例子，它展示了如何在选择标准实现和无分支实现时进行推理论证。现实世界的场景可能更难分析，因此，再次测量以找出在您的情况下替换分支是否有益。

如果没有性能分析数据，编译器就无法了解误判率。因此，编译器通常默认生成分支（即原始版本）。它们在使用谓词方面比较保守，即使在简单的情况下也可能抵制生成 CMOV 指令。同样，权衡也很复杂，如果没有运行时数据，就很难做出正确的决定。基于硬件的 PGO（参见 [#sec:secPGO]）将是这里的一大进步。此外，还有一种方法可以通

<sup>177</sup> 关于无分支二分查找的讨论 - <https://stackoverflow.com/a/54273248>。

过硬件机制向编译器指示分支条件是不可预测的。从 Clang-17 开始，编译器现在支持 `__builtin_unpredictable`，它可以非常有效地将不可预测的分支替换为 CMOV x86 指令。例如：

```
if (__builtin_unpredictable(x != 2))
    y = 0;
if (__builtin_unpredictable(x == 3))
    y = 1;
```

## 问题和练习

1. 使用本章讨论的技术解决以下实验作业：

- `perf-ninja::branches_to_cmov_1`
- `perf-ninja::lookup_tables_1`
- `perf-ninja::virtual_call_mispredict`
- `perf-ninja::conditional_store_1`

2. 运行您每天使用的应用程序。收集 TMA 细分并检查 `BadSpeculation` 指标。查看代码中哪个部分归因于最多的分支预测错误。是否可以使用本章讨论的技术来避免分支？

**编码练习：**编写一个微基准，使其经历 50% 的预测错误率或尽可能接近 50%。您的目标是用代码编写，使其所有分支指令中有一半被预测错误。这并不像您想象的那么简单。一些提示和想法：- 分支预测错误率计算为 `BR_MISP_RETIRED.ALL_BRANCHES / BR_INST_RETIRED.ALL_BRANCHES`。- 如果使用 C++ 编码，您可以 1) 使用类似于 perf-ninja 的 google benchmark，或 2) 编写常规控制台程序并使用 Linux `perf` 收集 CPU 计数器，或 3) 将 libpfm 集成到微基准中（参见 Section 5.4）。- 无需发明复杂的算法。一种简单的方法是在范围 [0;100] 中生成一个伪随机数，并检查它是否小于 50。随机数可以提前预生成。- 请记住，现代 CPU 可以记住较长（但仍然有限）的分支结果序列。

## 章节总结

- 现代处理器在预测分支结果方面非常出色。因此，我们建议只有当 TMA 报告显示高 `Bad Speculation` 指标时才开始修复分支预测错误的工作。
- 当分支结果模式变得难以让 CPU 分支预测器跟踪时，应用程序的性能可能会受到影响。在这种情况下，无分支版本的算法可能表现更好。本章展示了如何用查找表、算术和谓词替换分支。在某些情况下，还可以使用编译器内联函数消除分支，如 [Kapoor, 2009] 所示。
- 无分支算法并不是普遍有利的。始终测量以找出最适合您特定情况的方法。

# 10 机器代码布局优化

中央处理器的前端 (FE) 负责提取和解码指令，并将它们传递给无序后端 (BE)。随着新一代处理器执行“马力”的提升，CPU FE 也需要变得更加强大，以保持机器的平衡。如果 FE 无法跟上提供指令的速度，BE 将无法充分利用，整体性能将会下降。这就是为什么 FE 设计为始终超前于实际执行，以平滑可能发生的任何 hiccup 并始终准备好要执行的指令。例如，2016 年发布的 Intel Skylake 每周期可以提取最多 16 条指令。

大多数情况下，CPU FE 的低效率可以描述为 BE 等待执行指令的情况，但 FE 无法提供它们。因此，CPU 周期被浪费而没有做任何实际的有用工作。回想一下，现代 CPU 可以每周期处理多个指令，现在范围从 4 到 8 宽。并非所有可用插槽都填满的情况经常发生。这代表了数据库、编译器、网络浏览器等许多领域的应用程序低效性的来源。

TMA 方法在“前端绑定”指标中捕获 FE 性能问题。它表示 CPU FE 无法向 BE 提供指令（而 BE 本可以接受它们）的周期百分比。大多数现实世界的应用程序都经历了一个非零的“前端绑定”指标，这意味着运行时间的某个百分比将浪费在次优的指令获取和解码上。低于 10% 是常态。如果您看到“前端绑定”指标超过 20%，那么花时间处理它是绝对值得的。

FE 无法将指令传递给执行单元可能有多种原因。大多数情况下，这是由于代码布局不佳导致的，从而导致 I-cache 和 ITLB 利用率低下。拥有庞大代码库（例如数百万行代码）的应用程序尤其容易出现 FE 性能问题。在本章中，我们将介绍一些典型的优化方法来改善机器代码布局并提高程序的整体性能。

## ## 机器代码布局

当编译器将源代码转换为机器代码时，它会生成一个线性的字节序列。Listing 10 显示了一个小段 C++ 代码的二进制布局示例。一旦编译器完成生成汇编指令，它需要对它们进行编码并按顺序排列在内存中。

### 代码清单: 机器代码布局示例

| C++ Code    | Assembly Listing | Disassembled Machine Code |
|-------------|------------------|---------------------------|
| .....       | .....            | .....                     |
| if (a <= b) | ; a is in edi    | 401125 cmp esi, edi       |
| bar();      | ; b is in esi    | 401128 jb 401131          |
| else        | cmp esi, edi     | 40112a call bar           |
| baz();      | jb .label1       | 40112f jmp 401136         |
|             | call bar()       | 401131 call baz           |
|             | jmp .label2      | 401136 ...                |
| .label1:    |                  |                           |
|             | call baz()       |                           |
| .label2:    |                  |                           |
|             | ...              |                           |

代码在二进制文件中放置的方式称为 机器代码布局。请注意，对于同一个程序，可以以许多不同的方式布局代码。对于 Listing 10 中的代码，编译器可能决定反转分支，以便首先调用 `baz`。此外，函数 `bar` 和 `baz` 的主体可以以两种不同的顺序放置：我们可以先在二进制文件中放置 `bar`，然后放置 `baz`，或者反转顺序。这会影响指令在内存中放置的偏移量，进而可能会影响生成的二进制文件的性能，正如您将在后面看到的。在本章的以下部分中，我们将介绍一些机器代码布局的典型优化。

## 10.1 基本块

基本块是一系列具有单个入口和单个出口的指令。图 @fig:BasicBlock 显示了一个基本块的简单示例，其中 MOV 指令是一个入口，而 JA 是一个退出指令。虽然一个基本块可以有一个或多个前驱和后继，但中间的指令不能进入或退出基本块。

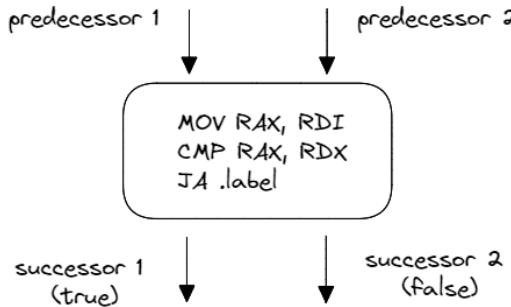


Figure 67: 汇编指令的基本块。

保证基本块中的每条指令都会被执行一次。这是一个重要的属性，被许多编译器转换所利用。例如，它极大地减少了控制流分析和转换的问题，因为对于某些问题类别，我们可以将基本块中的所有指令视为一个实体。

## 10.2 基本块布局

假设我们在程序中有一个热路径，其中有一些错误处理代码 (coldFunc)：

```
// 热路径
if (cond)
    coldFunc();
// 再次热路径
```

图 68 显示了这段代码的两种可能的物理布局。如果没有提供任何提示，图 68a 是大多数编译器默认会生成的布局。如果我们反转条件 cond 并将热代码放置为 fall through，则可以实现图 68b 中所示的布局。

哪种布局更好？这取决于 cond 通常是真的还是假的。如果 cond 通常为真，那么我们最好选择默认布局，因为否则我们会执行两次跳转而不是一次。另外，在一般情况下，如果 coldFunc 是一个相对较小的函数，我们希望它被内联。但是，在这个特定的例子中，我们知道 coldFunc 是一个错误处理函数，可能不会经常执行。通过选择布局 68b，我们在代码的热部分之间保持 fall through，并将 taken branch 转换为 not taken branch。

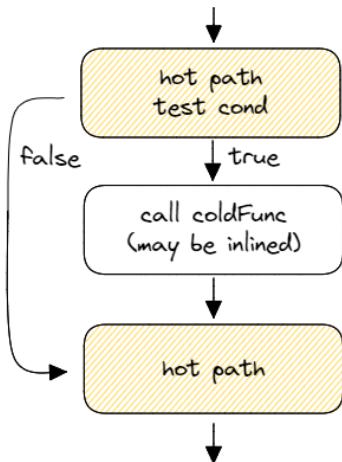
图 68b 中呈现的布局性能更好，原因有几个。首先，图 68b 中的布局更好地利用了指令和 μop-cache (DSB，参见 Section 3.8.1)。所有热代码都连续，没有缓存行碎片：L1I-cache 中的所有缓存行都被热代码使用。μop-cache 也是如此，因为它也是基于底层代码布局进行缓存的。其次，taken branch 对于 fetch 单元来说也更昂贵。CPU 前端会连续获取字节块，因此每次 taken jump 都意味着 jump 之后的字节是无用的。这会降低最大有效提取吞吐量。最后，在某些架构上，not taken branch 比 taken branch 便宜。例如，Intel Skylake CPU 每周期可以执行两个 untaken branch，但每两周期只能执行一个 taken branch。<sup>178</sup>

为了建议编译器生成改进版本的机器代码布局，可以使用 `[[likely]]`<sup>179</sup> 和 `[[unlikely]]` 属性提供提示，该属性从 C++20 开始可用。使用此提示的代码如下所示：

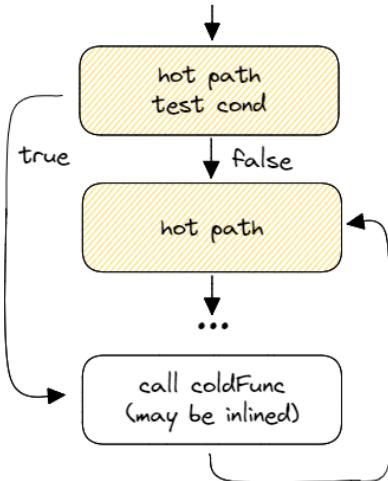
```
// 热路径
```

<sup>178</sup> 不过，有一个特殊的微型循环优化，可以使非常小的循环每个周期执行一个 taken branch。

<sup>179</sup> C++ 标准 `[[likely]]` 属性：<https://en.cppreference.com/w/cpp/language/attributes/likely>。



(a) 默认布局



(b) 改进的布局

Figure 68: 上面代码片段的两种机器代码布局版本。

```
if (cond) [[unlikely]]
    coldFunc();
// 再次热路径
```

在上面代码中，`[[unlikely]]` 提示将指示编译器 `cond` 不太可能为真，因此编译器应相应地调整代码布局。在 C++20 之前，开发人员可以使用 `__builtin_expect`: <https://llvm.org/docs/BranchWeightMetadata.html#builtin-expect<sup>180</sup>> 构造，他们通常自己创建 `LIKELY` wrapper 提示以使代码更具可读性。例如：

```
#define LIKELY(EXPR) __builtin_expect((bool)(EXPR), true)
#define UNLIKELY(EXPR) __builtin_expect((bool)(EXPR), false)
// 热路径
if (UNLIKELY(cond)) // NOT
    coldFunc();
// 再次热路径
```

优化编译器不仅会在遇到“`likely/unlikely`”提示时改进代码布局。他们还会在其他地方利用这些信息。例如，当应用 `[[unlikely]]` 属性时，编译器将阻止内联 `coldFunc`，因为它现在知道它不太可能经常被执行，并且优化它的大小更有利，即只留下一个 `CALL` 到这个函数。插入 `[[likely]]` 属性对于 `switch` 语句也是可能的，如 Listing 180 所示。

代码清单:switch 语句中可能使用的属性

```
for (;;) {
    switch (instruction) {
        case NOP: handleNOP(); break;
        [[likely]] case ADD: handleADD(); break;
        case RET: handleRET(); break;
        // handle other instructions
    }
}
```

使用此提示，编译器将能够稍微重新排序代码并优化热交换以更快地处理 `ADD` 指令。

## 基本块对齐

有时，性能会根据指令在内存中的偏移量而发生显着变化。考虑 Listing 180 中提供的简单函数以及使用 `-O3 -march=core-avx2 -fno-unroll-loops` 编译时对应的机器码。为了说明这个想法，循环展开被禁用了。

代码清单: 基本的块对齐

```
void benchmark_func(int* a) {
    for (int i = 0; i < 32; ++i)
        a[i] += 1;
}                                00000000004046a0 <_Z14benchmark_funcPi>:
                                4046a0: mov rax,0xffffffffffff80
                                4046a7: vpcmpeqd ymm0,ymm0,ymm0
                                4046ab: nop DWORD [rax+rax+0x0]
                                4046b0: vmovdqu ymm1,YMMWORD [rdi+rax+0x80] # loop begins
                                4046b9: vpsubd ymm1,ymm1,ymm0
                                4046bd: vmovdqu YMMWORD [rdi+rax+0x80],ymm1
                                4046c6: add rax,0x20
                                4046ca: jne 4046b0                      # loop ends
                                4046cc: vzeroupper
```

<sup>180</sup> 有关 `builtin-expect` 的更多信息，请参见此处：<https://llvm.org/docs/BranchWeightMetadata.html#builtin-expect>。

4046cf: ret

代码本身相当合理，但布局并不完美（见图 69a）。对于循环的指令用黄色斜线突出显示。与数据缓存一样，指令缓存行长度为 64 字节。在图 69 中，粗框表示缓存行边界。请注意，循环跨越多个缓存行：它从缓存行 0x80-0xBF 开始，并在缓存行 0xC0-0xFF 结束。为了获取在循环中执行的指令，处理器需要读取两个缓存行。这些情况通常会导致 CPU 前端的性能问题，尤其是对于上面呈现的小循环。

为了解决这个问题，我们可以使用 NOP 将循环指令向前移动 16 个字节，以便整个循环位于一个缓存行中。图 69b 显示了使用以蓝色突出显示的 NOP 指令执行此操作的效果。有趣的是，即使您在微基准测试中只运行这个热循环，性能影响也是可见的。这有点令人困惑，因为代码量很小，它不应该在任何现代 CPU 上占用 L1I 缓存的大小。图 69b 中布局性能更好的原因解释起来并不简单，并且会涉及大量微体系结构细节，我们在本书中不讨论这些细节。感兴趣的读者可以在 easyperf 博客上的文章“Code alignment issues: [https://easyperf.net/blog/2018/01/18/Code\\_alignment\\_issues](https://easyperf.net/blog/2018/01/18/Code_alignment_issues)”中找到更多信息。<sup>181</sup>

|      | 0    | 1    | 2    | 3    | 4    | 5    | 6    | 7     | 8     | 9     | A     | B     | C     | D     | E    | F    |
|------|------|------|------|------|------|------|------|-------|-------|-------|-------|-------|-------|-------|------|------|
| 0x80 |      |      |      |      |      |      |      |       |       |       |       |       |       |       |      |      |
| 0x90 |      |      |      |      |      |      |      |       |       |       |       |       |       |       |      |      |
| 0xa0 | mov  | vpcmp | vpcmp | vpcmp | vpcmp | nop   | nop   | nop   | nop  | nop  |
| 0xb0 | vmov | vpsub | vpsub | vpsub | vpsub | vmov  | vmov  | vmov  | vmov | vmov |
| 0xc0 | vmov | vmov | vmov | vmov | vmov | vmov | add  | add   | add   | add   | jne   | vzero | vzero | vzero | ret  |      |
| 0xd0 |      |      |      |      |      |      |      |       |       |       |       |       |       |       |      |      |
| 0xe0 |      |      |      |      |      |      |      |       |       |       |       |       |       |       |      |      |
| 0xf0 |      |      |      |      |      |      |      |       |       |       |       |       |       |       |      |      |

(a) 默认布局

|      | 0     | 1     | 2     | 3    | 4    | 5    | 6    | 7     | 8     | 9     | A     | B     | C     | D    | E    | F    |
|------|-------|-------|-------|------|------|------|------|-------|-------|-------|-------|-------|-------|------|------|------|
| 0x80 |       |       |       |      |      |      |      |       |       |       |       |       |       |      |      |      |
| 0x90 |       |       |       |      |      |      |      |       |       |       |       |       |       |      |      |      |
| 0xa0 | mov   | mov   | mov   | mov  | mov  | mov  | mov  | vpcmp | vpcmp | vpcmp | vpcmp | nop   | nop   | nop  | nop  | nop  |
| 0xb0 | nop   | nop   | nop   | nop  | nop  | nop  | nop  | nop   | nop   | nop   | nop   | nop   | nop   | nop  | nop  | nop  |
| 0xc0 | vmov  | vmov  | vmov  | vmov | vmov | vmov | vmov | vmov  | vmov  | vpsub | vpsub | vpsub | vpsub | vmov | vmov | vmov |
| 0xd0 | vmov  | vmov  | vmov  | vmov | vmov | vmov | add  | add   | add   | jne   | jne   | nop   | nop   | nop  | nop  | nop  |
| 0xe0 | vzero | vzero | vzero | ret  |      |      |      |       |       |       |       |       |       |      |      |      |
| 0xf0 |       |       |       |      |      |      |      |       |       |       |       |       |       |      |      |      |

(b) 改进布局

Figure 69: Listing 180 中循环的两种不同的代码布局。

默认情况下，LLVM 编译器会识别循环并将它们对齐到 16B 边界，如图 69a 所示。为了达到我们示例的所需代码位置，如图 69b 所示，可以使用 `-mllvm -align-all-blocks=5` 选项，该选项将在目标文件中将每个基本块对齐到 32 字节边界。但是，请谨慎使用此选项，因为它很容易在其他地方降低性能。此选项在执行路径上插入 NOP，这可能会增加程序的开销，尤其是当它们位于关键路径上时。NOP 不需要执行；但是，它们仍然需要从内存中获取、解码和退出。后者还会消耗 FE 数据结构和缓冲区中的空间用于簿记，类似于所有其他指令。LLVM 编译器中还有其他不那么侵入性的选项可用于控制基本块对齐，您可以在 easyperf 博客 post: [https://easyperf.net/blog/2018/01/25/Code\\_alignment\\_options\\_in\\_llvm](https://easyperf.net/blog/2018/01/25/Code_alignment_options_in_llvm) 中查看这些选项。<sup>182</sup>

<sup>181</sup> “Code alignment issues” - [https://easyperf.net/blog/2018/01/18/Code\\_alignment\\_issues](https://easyperf.net/blog/2018/01/18/Code_alignment_issues)

<sup>182</sup> “Code alignment options in llvm” - [https://easyperf.net/blog/2018/01/25/Code\\_alignment\\_options\\_in\\_llvm](https://easyperf.net/blog/2018/01/25/Code_alignment_options_in_llvm)

LLVM 编译器最近的新增功能是 `[[clang::code_align()]]` 循环属性，它允许开发人员在源代码中指定循环的对齐方式。这提供了对机器代码布局非常细粒度的控制。在引入此属性之前，开发人员不得不求助于一些不太实用的解决方案，例如在源代码中注入内联汇编语句 `asm(".align 64;")`。以下代码展示了如何使用新的 Clang 属性将循环对齐到 64 字节边界：

```
void benchmark_func(int* a) {
    [[clang::code_align(64)]]
    for (int i = 0; i < 32; ++i)
        a[i] += 1;
}
```

尽管 CPU 架构师努力将机器代码布局的影响最小化，但在某些情况下，代码放置（对齐）仍然会影响性能。机器代码布局也是性能测量的主要噪音源之一。它使区分真正的性能改进或回归与意外发生的回归（由代码布局的改变引起）变得更加困难。

### 10.3 函数拆分

函数拆分的思想是将热点代码与冷代码分开。这种转换通常也被称为函数轮廓化。这种优化对于具有复杂控制流图和热路径中存在大量冷代码块的相对较大函数非常有益。Listing 10.3 中显示了这种转换可能会有好处的代码示例。为了将热路径中的冷基本块移除，我们将它们剪切并粘贴到一个新函数中，并创建一个调用它们的调用。

代码清单：函数拆分：将冷代码轮廓化到新函数。

|                                                                                                                                                                       |                                                                                                                                                                                                                                                                               |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>void foo(bool cond1, bool cond2) {     // 热路径     if (cond1) {         /* 大量的冷代码 (1) */     }     // 热路径     if (cond2) {         /* 大量的冷代码 (2) */     } }</pre> | <pre>void foo(bool cond1, bool cond2) {     // 热路径     if (cond1) {         cold1();     }     // 热路径     if (cond2) {         cold2();     } } void cold1() __attribute__((noinline)) { /* 大量的冷代码 (1) */ } void cold2() __attribute__((noinline)) { /* 大量的冷代码 (2) */ }</pre> |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

请注意，我们通过使用 `noinline` 属性禁用了冷函数的内联。因为如果没有这个属性，编译器可能会决定内联它，这实际上会撤销我们的转换。或者，我们可以在 `cond1` 和 `cond2` 分支上都应用 `[[unlikely]]` 宏（参见 Section 10.2），以传达给编译器不希望内联 `cold1` 和 `cold2` 函数。

图 @fig:FunctionSplitting 给出了这种转换的图形表示。因为我们在热路径中只留下了一个 `CALL` 指令，所以下一个热指令很可能会与上一个指令驻留在同一个缓存行中。这提高了 CPU 前端数据结构（如 I-cache 和 DSB）的利用率。

轮廓化的函数应该创建在 `.text` 段之外，例如在 `.text.cold` 中。如果函数从不被调用，这样做可以提高内存占用，因为它在运行时不会加载到内存中。

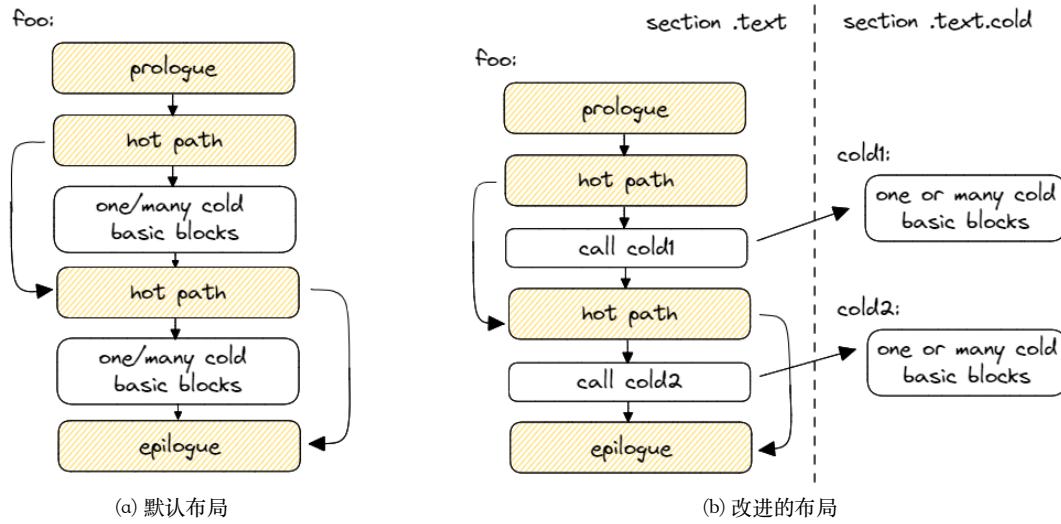


Figure 70: 将冷代码拆分到单独的函数中。

## 10.4 函数重排序

根据前面各节描述的原则，可以将热点函数分组，进一步提高 CPU 前端缓存的利用率。当热点函数被分组在一起时，它们开始共享缓存行，这减少了代码占用空间，即 CPU 需要获取的缓存行总数。

图 @fig:FunctionGrouping 给出了重新排列热点函数 foo、bar 和 zoo 的图形表示。图像上的箭头显示了最频繁的调用模式，即 foo 调用 zoo，然后 zoo 调用 bar。在默认布局中（见图 @fig:FuncGroup\_default），热点函数不相邻，它们之间有一些冷函数。因此，两个函数调用的序列（foo -> zoo -> bar）需要读取四个缓存行。

我们可以重新排列函数的顺序，使得热点函数彼此靠近（见图 @fig:FuncGroup\_better）。在改进的版本中，foo、bar 和 zoo 函数的代码适合于三个缓存行。另外，注意函数 zoo 现在根据函数调用的顺序被放置在 foo 和 bar 之间。当我们从 foo 调用 zoo 时，zoo 的开始已经在 I-cache 中。

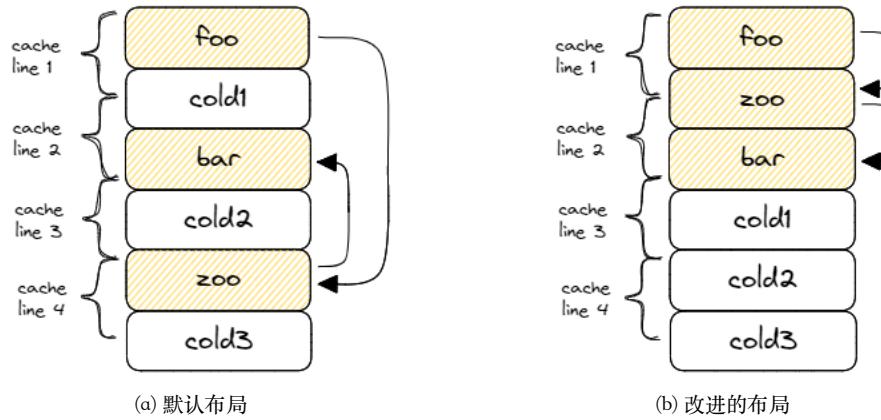


Figure 71: 重新排列热点函数。

与之前的优化类似，函数重排序提高了 I-cache 和 DSB-cache 的利用率。当存在许多小型热点函数时，这种优化效果最佳。

链接器负责将程序的所有函数布局在生成的二进制输出中。虽然开发人员可以尝试自己重新排列程序中的函数，但不能保证所需的物理布局。几十年来，人们一直使用链接器脚本来实现这个目标。如果你使用的是 GNU 链接器，这

仍然是一种行之有效的方法。Gold 链接器 (`ld.gold`) 对这个问题有了更简单的解决方案。要在 Gold 链接器中获得二进制文件中函数的所需顺序，可以首先使用 `-ffunction-sections` 标志编译代码，这会将每个函数放入一个单独的节中。然后使用 `--section-ordering-file=order.txt` 选项提供一个带有按所需最终布局排序的函数名称列表的文件。LLD 链接器也具有相同的特性，它是 LLVM 编译器基础设施的一部分，并通过 `--symbol-ordering-file` 选项访问。

解决将热点函数分组在一起的问题的一个有趣方法是由 Meta 的工程师在 2017 年引入的。他们实现了一个名为 HFSort 的工具<sup>183</sup>，它根据分析数据自动生成节顺序文件 [Ottoni & Maher, 2017]。使用这个工具，他们观察到大型分布式云应用程序，如 Facebook、百度和维基百科的性能提高了 2%。HFSort 已经集成到了 Meta 的 HHVM、LLVM BOLT 和 LLD 链接器中<sup>184</sup>。从那时起，该算法首先被 HFSort+ 取代，最近又被 Cache-Directed Sort (CDSort<sup>185</sup>) 所取代，对于具有大型代码占用空间的工作负载带来了更多改进。

## 10.5 使用配置分析文件引导的优化 (Profile Guided Optimizations)

编译程序和生成最佳汇编代码都是关于启发式的。代码转换算法有很多特殊情况，旨在在特定情况下实现最佳性能。对于编译器做出的许多决策，它会尝试根据一些典型案例猜测最佳选择。例如，当决定是否内联特定函数时，编译器可能会考虑该函数被调用的次数。问题是编译器事先并不知道这一点。它首先需要运行程序才能找出答案。如果没有运行时信息，编译器就必须猜测。

这就是分析 (profiling) 信息派上用场的时候。有了分析 (profiling) 信息，编译器可以做出更好的优化决策。大多数编译器中有一组转换，可以根据反馈给他们的分析 (profiling) 数据调整算法。这组转换称为分析 (profiling) 优化 (PGO)。当分析 (profiling) 数据可用时，编译器可以用它来指导优化。否则，它将退回到使用其标准算法和启发式。有时在文献中，您可以找到术语反馈定向优化 (FDO)，它指的是与 PGO 相同的东西。

图 72 显示了使用 PGO 的传统工作流程，也称为插桩化的 (instrumented) PGO。首先，您编译您的程序并告诉编译器自动检测代码。这将在函数中插入一些记账代码以收集运行时统计信息。第二个步骤是使用代表应用程序典型工作负载的输入数据运行插桩化的 (instrumented) 二进制文件。这将生成分析 (profiling) 数据，一个包含运行时统计信息的新文件。它是一个原始转储文件，其中包含有关函数调用计数、循环迭代计数和其他基本块命中计数的信息。此工作流程的最后一步是使用分析 (profiling) 数据重新编译程序以生成优化的可执行文件。

开发人员可以通过使用 `-fprofile-instr-generate` 选项构建程序来启用 LLVM 编译器中的 PGO 检测 (步骤 1)。这将指示编译器检测代码，从而在运行时收集分析 (profiling) 信息。之后，LLVM 编译器可以使用 `-fprofile-instr-use` 选项使用分析 (profiling) 数据重新编译程序并输出经过 PGO 调整的二进制文件。clang 中使用 PGO 的指南在文档: <https://clang.llvm.org/docs/UsersManual.html#profiling-with-instrumentation> 中进行了描述。<sup>186</sup> GCC 编译器使用不同的选项集: `-fprofile-generate` 和 `-fprofile-use`，如文档: <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options> 所述。<sup>187</sup>

PGO 帮助编译器改进函数内联、代码放置、寄存器分配和其他代码转换。PGO 主要用于具有大型代码库的项目，例如 Linux 内核、编译器、数据库、网络浏览器、视频游戏、生产力工具等。对于拥有数百万行代码的应用程序，它是改善机器代码布局的唯一实践方法。使用分析 (profiling) 优化使生产工作负载的性能提高 10-25% 并不少见。

虽然许多软件项目将插桩化的 (instrumented) PGO 作为其构建过程的一部分，但采用率仍然很低。有几个原因。主要原因是插桩化的 (instrumented) 可执行文件的巨大运行时开销。运行插桩化的 (instrumented) 二进制文件和收集分析 (profiling) 数据经常会降低 5-10 倍的速度，这使构建步骤更长，并且阻止直接从生产系统（无论是在客户端设

<sup>183</sup> HFSort - <https://github.com/facebook/hhvm/tree/master/hphp/tools/hfsort>

<sup>184</sup> LLD 中的 HFSort - <https://github.com/llvm-project/lld/blob/master/ELF/CallGraphSort.cpp>

<sup>185</sup> LLVM 中的 Cache-Directed Sort - <https://github.com/llvm/llvm-project/blob/main/llvm/lib/Transforms/Utils/CodeLayout.cpp>

<sup>186</sup> Clang 中的 PGO - <https://clang.llvm.org/docs/UsersManual.html#profiling-with-instrumentation>

<sup>187</sup> GCC 中的 PGO - <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options>

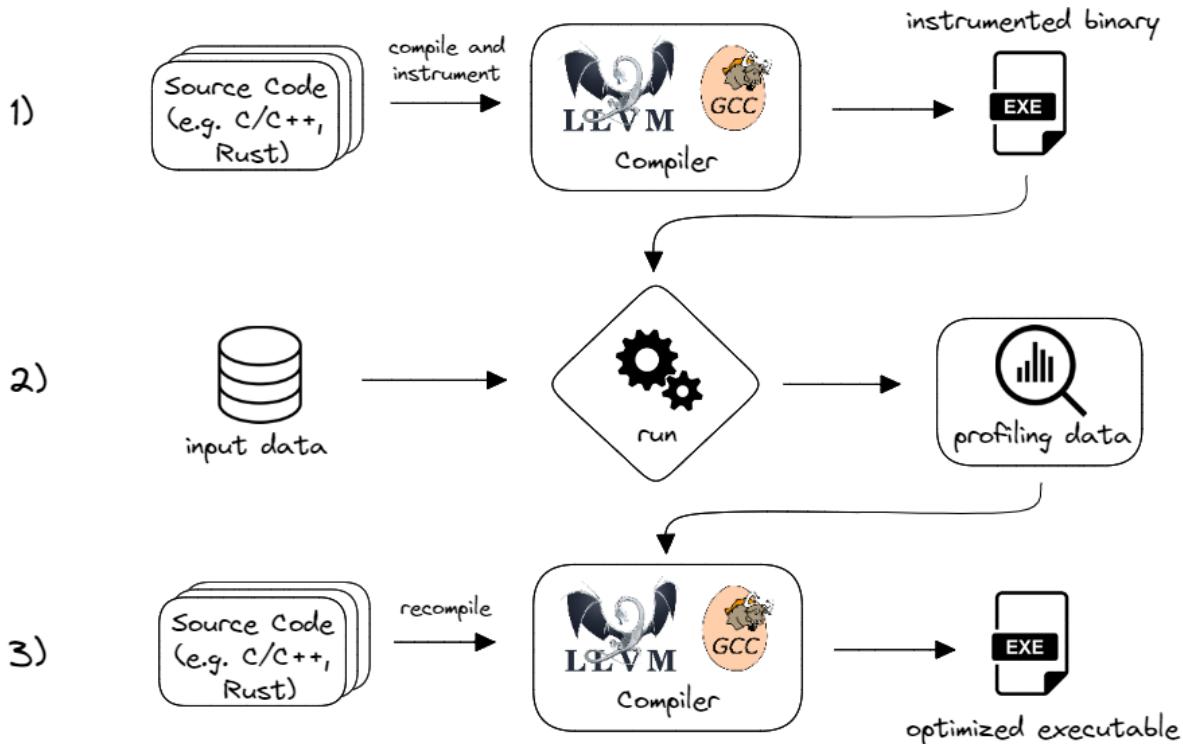


Figure 72: 使用工具 PGO 的工作流程.

备还是云端) 收集配置文件。不幸的是, 您无法一次收集分析 (profiling) 数据并将其用于所有未来的构建。随着应用程序源代码的演变, 配置文件数据会变得陈旧 (不同步), 需要重新收集。

PGO 流程的另一个注意事项是, 编译器应该只使用应用程序将如何使用的代表性场景进行训练。否则, 您最终可能会降低程序的性能。编译器会“盲目地”使用您提供的配置文件数据。它假设程序无论输入数据是什么都始终表现相同。PGO 的用户应该谨慎选择用于收集配置文件数据 (步骤 2) 的输入数据, 因为在改进应用程序的一个用例的同时, 其他用例可能会被悲观化。幸运的是, 它不必仅仅是单个工作负载, 因为来自不同工作负载的配置文件数据可以合并在一起, 以代表应用程序的一组用例。

谷歌在 2016 年率先提出了另一种基于样本的 PGO 解决方案。[\[Chen et al., 2016\]](#) 除了检测代码之外, 还可以从标准配置文件工具 (例如 Linux perf) 的输出中获取配置文件数据。谷歌开发了一个名为 AutoFDO: <https://github.com/google/autofdo><sup>188</sup> 的开源工具, 可以将 Linux perf 生成的采样数据转换为 GCC 和 LLVM 等编译器可以理解的格式。

与带仪器的 PGO 相比, 这种方法有一些优点。首先, 它消除了 PGO 构建工作流程的一个步骤, 即步骤 1, 因为无需构建带有仪器的二进制文件。其次, 配置文件数据收集运行在已经优化的二进制文件上, 因此运行时开销要低得多。这使得可以在生产环境中更长时间地收集配置文件数据。由于这种方法基于硬件收集, 它还支持使用带仪器的 PGO 无法实现的新型优化。一个例子是分支到 cmov 转换, 这是一个用条件移动替换条件跳转以避免分支预测错误开销的转换 (参见 Section 9.3)。为了有效地执行此转换, 编译器需要知道原始分支的错误预测频率。此信息可在现代 CPU (Intel Skylake+) 上的基于样本的 PGO 中获得。

下一个创新想法来自 Meta, 它在 2018 年年中开源了其名为 BOLT: <https://code.fb.com/data-infrastructure/accelerate-large-scale-applications-with-bolt/> 的二进制优化工具。<sup>189</sup> BOLT 在已经编译的二进制文件上工作。它首先反汇编代码, 然后使用 Linux perf 等采样分析器收集的配置文件信息进行各种布局转换, 然后重新链接二进制文件。

<sup>188</sup> AutoFDO - <https://github.com/google/autofdo>

<sup>189</sup> BOLT - <https://code.fb.com/data-infrastructure/accelerate-large-scale-applications-with-bolt/>

[Panchenko et al., 2018] 截至今天，BOLT 拥有超过 15 个优化通道，包括基本块重新排序、函数拆分和重新排序等。与传统 PGO 类似，BOLT 优化的主要候选是遭受许多指令缓存和 iTLB 未命中折磨的程序。自 2022 年 1 月起，BOLT 成为 LLVM 项目的一部分，并作为独立工具提供。

谷歌在 BOLT 引入几年后开源了其名为 Propeller: [https://github.com/google/llvm-propeller/blob/plo-dev/Propeller\\_RFC.pdf](https://github.com/google/llvm-propeller/blob/plo-dev/Propeller_RFC.pdf) 的二进制重新链接工具。它具有类似的目的，但它不反汇编原始二进制文件，而是依赖于链接器输入，因此可以分布在多个机器上以实现更好的扩展和更少的内存消耗。BOLT 和 Propeller 等后链接优化器可以与传统 PGO (和 LTO) 结合使用，通常可以提供额外 5-10% 的性能提升。此类技术开辟了基于硬件遥测的新型二进制重写优化。

## 10.6 减少 ITLB 未命中

调整前端效率的另一个重要领域是内存地址的虚拟到物理地址转换。这些转换主要由 TLB (参见 Section 3.7.1) 提供服务，TLB 在专用条目中缓存最近使用的内存页转换。当 TLB 无法处理翻译请求时，将进行耗时的内核页表页面遍历，为每个引用的虚拟地址计算正确的物理地址。每当您在 TMA 摘要中看到高比例的 ITLB 开销时，本节中的建议可能派上用场。

通常情况下，相对较小的应用程序不易受到 ITLB 未命中的影响。例如，Golden Cove 微架构可以在其 ITLB 中覆盖高达 1MB 的内存空间。如果您的应用程序的机器代码适合 1MB，您应该不会受到 ITLB 未命中的影响。当应用程序中频繁执行的部分分散在内存周围时，问题就开始出现。当许多函数开始频繁相互调用时，它们会开始争夺 ITLB 中的条目。一个例子是 Clang 编译器，在撰写本文时，它的代码部分大约为 60MB。在运行主流 Intel CoffeeLake 处理器的笔记本电脑上，ITLB 开销约为 7%，这意味着 7% 的周期被浪费在处理 ITLB 未命中上：执行要求苛刻的页面遍历和填充 TLB 条目。

另一组经常受益于使用大页的大内存应用程序包括关系数据库（例如 MySQL、PostgreSQL、Oracle）、托管运行时（例如 Javascript V8、Java JVM）、云服务（例如网络搜索）、网络工具（例如 node.js）。将代码段映射到大页可以将 ITLB 未命中数量减少高达 50% [Suresh Srinivas, 2019]，从而为某些应用程序带来高达 10% 的加速。但是，与许多其他功能一样，大页并不适用于所有应用程序。可执行文件只有几 KB 大的小程序最好使用常规 4KB 页面而不是 2MB 大页；这样，内存可以更有效地利用。

减少 ITLB 压力的总体思路是将应用程序性能关键代码的部分映射到 2MB（大页）上。但通常情况下，整个应用程序的代码部分会重新映射以简化操作，或者如果您不知道哪些函数是热门函数。要进行这种转换，关键要求是代码部分与 2MB 边界对齐。在 Linux 上，这可以通过两种不同的方式实现：使用附加链接器选项重新链接二进制文件或在运行时重新映射代码部分。这两个选项都展示在 easyperf.net 博客<sup>190</sup> 上。据我们所知，它在 Windows 上是不可能的，因此我们只展示如何在 Linux 上进行操作。

第一个选项可以通过使用 `-Wl,-zcommon-page-size=2097152 -Wl,-zmax-page-size=2097152` 选项链接二进制文件来实现。这些选项指示链接器将代码段放在 2MB 边界，以便启动时由加载器将其放置在 2MB 页面上。这种放置的缺点是链接器将被迫插入多达 2MB 的填充（浪费）字节，使二进制文件更加臃肿。以 Clang 编译器为例，它使二进制文件的大小从 111MB 增加到 114MB。重新链接二进制文件后，我们在 ELF 二进制头中设置一个特殊位，该位确定文本段是否应默认由大页支持。最简单的方法是使用 libhugetlbf: <https://github.com/libhugetlbf/libhugetlbf/blob/master/HOWTO><sup>191</sup> 软件包中的 hugeedit 或 hugetlbf 工具。例如：

```
# 永久设置 ELF 二进制头中的特殊位。
$ hugeedit --text /path/to/clang++
# 代码段将默认使用大页加载。
$ /path/to/clang++ a.cpp
```

<sup>190</sup> “使用大页为代码带来的性能优势” - <https://easyperf.net/blog/2022/09/01/Utilizing-Huge-Pages-For-Code>.

<sup>191</sup> libhugetlbf - <https://github.com/libhugetlbf/libhugetlbf/blob/master/HOWTO>.

```
# 在运行时覆盖默认行为。
```

```
$ hugectl --text /path/to/clang++ a.cpp
```

第二个选项是在运行时重新映射代码段。此选项不需要代码段与 2MB 边界对齐，因此可以在不重新编译应用程序的情况下工作。这种方法背后的 idea 是在程序启动时分配大页并将所有代码段转移到那里。该方法的参考实现是在 iodlr: <https://github.com/intel/iodlr><sup>192</sup> 中实现的。一个选项是从您的 main 函数调用该功能。另一个更简单的方法是构建动态库并在命令行中预加载它：

```
$ LD_PRELOAD=/usr/lib64/liblppreload.so clang++ a.cpp
```

虽然第一种方法只能使用显式大页，但使用 iodlr 的第二种方法既适用于显式大页，也适用于透明大页。有关如何在 Windows 和 Linux 上启用大页的说明，请参见附录 C。

除了采用大页之外，还可以使用优化 I-cache 性能的标准技术来改善 ITLB 性能。例如，重新排序函数以更好地定位热门函数，通过链接时优化 (LTO/IPO) 减少热门区域的大小，使用概要引导优化 (PGO) 和 BOLT，以及减少激进内联。

BOLT 提供了 -hugify 选项，可以根据配置文件数据自动将大页用于热门代码。使用此选项时，llvm-bolt 将注入代码，在运行时将热门代码放在 2MB 页面上。该实现利用了 Linux 透明大页 (THP)。这种方法的优点是只有少部分代码映射到大页，所需的大页数量最小化，因此页面碎片减少。

## 10.7 案例研究：测量代码足迹

正如我们在本章中多次提到的，代码布局优化对具有大量代码的应用程序影响最大。阐明程序中热代码大小不确定的最佳方法是测量其代码足迹，我们将其定义为程序执行期间触及的机器指令的字节/缓存行/页面数量。

大型代码足迹本身并不一定会对性能产生负面影响。代码足迹不是决定性的指标，它不会立即告诉您是否存在问题。尽管如此，它已被证明作为性能分析中一个有用的额外数据点。结合 TMA 的“前端绑定”、“L1 指令缓存未命中率”和其他指标，它可能会加强投资时间优化应用程序机器代码布局的论据。

目前，可以可靠地测量代码足迹的工具还很少。在本案例研究中，我们将展示 perf-tools: <https://github.com/aayasin/perf-tools><sup>193</sup>，一个基于 Linux perf 构建的开源性能分析工具集。为了估计<sup>194</sup> 代码足迹，perf-tools 利用了英特尔的 LBR（参见 Section 5.11），因此目前它不能在 AMD 或 ARM 架构的系统上工作。下面是一个收集代码足迹数据的示例命令：

```
$ perf-tools/do.py profile --profile-mask 100 -a <your benchmark>
```

其中，--profile-mask 100 启动 LBR 采样，-a 允许您指定要运行的程序。此命令将收集代码足迹以及其他各种数据。我们不显示工具的输出，好奇的读者可以自行尝试。

我们选取了一组四个基准测试：Clang C++ 编译、Blender 光线追踪、Cloverleaf 流体动力学和 Stockfish 国际象棋引擎；这些工作负载您应该已经从 Section 4.11 分析其性能特征的地方熟悉了。我们在基于英特尔 Alderlake 的处理器上使用与 Section 4.11 中相同的命令运行它们。正如预期的那样，在基于 Skylake 的机器上运行相同基准测试获得的代码足迹数字与 Alderlake 运行的结果非常相似。代码足迹取决于程序和输入数据，而不是特定机器的特性，因此结果在不同架构上应该看起来相似。

四个基准测试的结果分别列在表 8 中。二进制和 .text 大小是使用标准的 Linux readelf 工具获得的，而其他指标是使用 perf-tools 收集的。如果一个指令内存位置被命中至少一次，该工具将其视为“非冷”，因此，“非冷代码足

<sup>192</sup> iodlr 库 - <https://github.com/intel/iodlr>.

<sup>193</sup> perf-tools - <https://github.com/aayasin/perf-tools>

<sup>194</sup> perf-tools 收集的代码足迹数据是不精确的，因为它是基于 LBR 记录的采样。不幸的是，其他工具，如英特尔的 sde -footprint，不提供代码占用。然而，自己编写一个基于 pin 的工具来测量准确的代码占用并不难。

迹 [KB]”是程序触及的包含机器指令的千字节数。指标“非冷代码 4KB 页”告诉我们程序触及的包含机器指令的非冷 4KB 页数。它们一起帮助我们理解这些非冷内存位置的密度或稀疏性。一旦我们深入研究这些数字，这一点就会变得清晰。最后，我们还提供了前端绑定百分比，这是一个您应该已经从 Section ?? 中了解的关于 TMA 的指标。

Table 8: 案例研究中使用的基准测试的代码足迹。

| Metric                          | Clang17 编译 | Blender | CloverLeaf | Stockfish |
|---------------------------------|------------|---------|------------|-----------|
| Binary size [KB]                | 113844     | 223914  | 672        | 39583     |
| .text size [KB]                 | 67309      | 133009  | 598        | 238       |
| non-cold code footprint [KB]    | 5042       | 313     | 104        | 99        |
| non-cold code 4KB-pages         | 6614       | 546     | 104        | 61        |
| Frontend Bound, Alderlake-P [%] | 52.3       | 29.4    | 5.3        | 25.8      |

解释：

- 二进制大小 (Binary size): 指的是应用程序的可执行文件大小。
- .text 大小: 指的是可执行文件中代码段的大小。
- 非冷代码足迹 (non-cold code footprint): 指的是程序执行过程中触及的代码部分的大小，以千字节为单位。
- 非冷代码 4KB 页 (non-cold code 4KB-pages): 指的是非冷代码占用的 4KB 页数。
- 前端绑定 (Frontend Bound) Alderlake-P: 指的是由于等待指令而导致的性能瓶颈，以百分比表示。

观察：

- Clang17 编译的非冷代码足迹最大，其次是 Stockfish 和 CloverLeaf。
- Blender 虽然 .text 大小很大，但其非冷代码足迹相对较小。
- CloverLeaf 的非冷代码 4KB 页利用率最高，其次是 Stockfish 和 Clang17。
- Clang17 的前端绑定问题最为严重，其次是 Stockfish 和 Blender。

分析：

- 代码足迹可以帮助我们了解应用程序对 CPU 前端的压力。
- 非冷代码 4KB 页利用率可以反映代码布局的紧凑程度。
- 前端绑定指标可以帮助我们识别性能瓶颈。

注意：

- 此分析不考虑应用程序外部的代码，例如动态链接库。

让我们先来看看二进制和.text的大小。与 Clang17 和 Blender 相比，CloverLeaf 是一个非常小的应用程序；Stockfish 嵌入的神经网络文件占了大部分二进制文件，但其代码部分相对较小；Clang17 和 Blender 拥有庞大的代码库。.text size 指标是我们的应用程序的上限，即我们假设<sup>195</sup> 代码足迹不应超过 .text 大小。

通过分析代码足迹数据，我们可以做出一些有趣的观察。首先，尽管 Blender 的 .text 部分非常大，但不到 1% 的 Blender 代码是非冷的：133 MB 中只有 313 KB。因此，仅仅因为二进制文件很大，并不意味着应用程序会遭受 CPU 前端瓶颈。真正重要的是热代码的数量。对于其他基准测试，这个比率更高：Clang17 7.5%，CloverLeaf 17.4%，Stockfish 41.6%。从绝对数字来看，Clang17 编译触及的机器指令字节数比其他三个应用程序高出一个数量级。

其次，让我们检查表中的“非冷代码 4KB 页”行。对于 Clang17，5042 KB 的非冷代码分布在 6614 个 4KB 页面上，这给我们提供了  $5042 / (6614 * 4) = 19\%$  的页面利用率。这个指标告诉我们代码的热点部分的密度/稀疏性。每个热点缓存行越靠近另一个热点缓存行，就需要更少的页面来存储热点代码。页面利用率越高越好。本章前面讨

<sup>195</sup> 这并不总是正确的：一个应用程序本身可能很小，但需要调用多个其他动态链接的库，或者它可能大量使用内核代码。

论的基本块放置和函数重新排序是提高页面利用率的转换的完美例子。对于其他基准测试，比率为：Blender 14%，CloverLeaf 25%，Stockfish 41%。

现在我们已经量化了四个应用程序的代码足迹，可能会考虑 L1 指令缓存和 L2 缓存的大小以及热代码是否适合。在我们的 Alderlake 处理器上，L1-I 缓存只有 32 KB，不足以完全覆盖我们分析过的任何基准测试。但请记住，在本节开头我们说过，大型代码足迹并不直接指向问题。是的，大型代码库会给 CPU 前端带来更大的压力，但指令访问模式也对性能至关重要。与数据访问相同的局部性原则也适用。这就是为什么我们将其与 Topdown 分析中的前端绑定指标结合在一起。

对于 Clang17，5 MB 的非冷代码导致了一个巨大的 52.3% 的前端绑定性能瓶颈：超过一半的周期都在等待指令。在所有呈现的基准测试中，它从 PGO 类型优化中获益最多。CloverLeaf 不存在指令获取效率低下的问题；它 75% 的分支都是向后跳转，这表明它们可能是反复执行的相对较小的循环。Stockfish 虽然与 CloverLeaf 具有大致相同的非冷代码足迹，但却对 CPU 前端提出了更大的挑战 (25.8%)。它有更多的间接跳转和函数调用。最后，Blender 比 Stockfish 具有更多的间接跳转和调用。由于进一步的调查超出了本案例研究的范围，因此我们在此停止分析。对于有兴趣继续分析的读者，我们建议根据 TMA 方法深入研究前端绑定类别，并查看诸如 ICache\_Misses, ITLB\_Misses, DSB coverage 等指标。

另一个研究代码足迹的有用工具是 llvm-bolt-heatmap: <https://github.com/llvm/llvm-project/blob/main/bolt/docs/Heatmaps.md><sup>196</sup>，它是 llvm 的 BOLT 项目的一部分。该工具可以生成代码热图，让您细粒度地了解应用程序的代码布局。其主要用途是 1) 评估原始代码布局是否紧凑以及是否可以优化，2) 确保优化后的代码布局在实际负载下仍然紧凑。

## 问题和练习

### 1. 解决 perf-ninja::pgo 实验作业：

不幸的是，我没有关于 perf-ninja::pgo 实验作业的具体细节。如果你能提供更多上下文或分享涉及的指令/任务，我将很高兴指导你完成该过程或提供关于基于 PGO 的优化的通用建议。

### 2. 尝试将大页面用于代码段：

以下是如何使用大页面为大型应用程序的代码段进行实验的综合方法：

准备：

- 应用程序选择：选择一个二进制大小超过 100MB 的应用程序（访问源代码有利但不是必需的）。
- 环境设置：确保您的系统支持大页面（检查内核文档和 /proc/meminfo）。您可能需要 root 权限才能重新映射内存。
- 性能分析：使用性能分析工具（例如 perf、gprof）收集基准性能数据，以测量与 ITLB 相关的指标，例如 ITLB 加载/未命中、CPU 周期和分支错误预测。

大页面重新映射：

- 方法选择：选择 Section 10.6 部分描述的方法之一（例如，链接器选项、使用 libhugetlbfs 的运行时重新映射）。
- 重新映射：根据所选方法，将应用程序的代码段重新映射到大页面上。有关详细说明，请参考特定方法的文档。

分析：

- 性能测量：重新运行应用程序的基准测试或工作负载，并将性能指标与基准数据进行比较。寻找执行时间、ITLB 未命中率以及花在内存相关操作上的 CPU 周期方面的改进。

<sup>196</sup> llvm-bolt-heatmap - <https://github.com/llvm/llvm-project/blob/main/bolt/docs/Heatmaps.md>

- 大页面分配：检查 `/proc/meminfo` 以验证大页面分配和使用情况。它应该显示 `HugePages_Allocated` 和 `HugePages_Rsvd` 的增加。
- 其他指标：考虑监控其他相关指标，例如缓存未命中、TLB 未命中和分支停顿，以更深入地了解对内存层次结构性能的影响。

解释和细化：

- 性能影响：分析性能变化。如果您观察到显著的改进，那么大页面可能对该应用程序有利。
- 微调：如果结果混合或不明确，请尝试使用不同的大页面大小或重新映射技术以找到最佳配置。
- 权衡：要意识到潜在的权衡（例如，内存碎片增加、外部碎片）。大页面可能并不适用于所有应用程序。

### 3. 优化循环中的 C++ switch 语句：

给定一个 C++ `switch` 语句，该语句 70% 的时间被使用，其他情况很少使用，以下是优化策略：

- 案例排序：将最频繁的案例（70%）移动到 `switch` 语句的顶部。这可以提高分支预测准确性并减少分支错误预测。
- 跳转表：如果适用，请考虑将 `switch` 语句转换为跳转表。当案例具有连续整数值时，这尤其有效。
- 基于范围的选择：如果案例对应于一系列值，请探索使用范围比较而不是 `switch` 语句进行基于范围的选择。
- 配置文件引导优化 (PGO)：如果可能，请使用 PGO 收集执行配置文件并指导编译器生成针对热点案例更高效的代码。
- 替代数据结构：根据上下文，您可以探索使用哈希表或二叉搜索树等替代数据结构进行更快的查找，尤其是在案例没有连续整数值的情况下。

### 4. 使用 PGO、llvm-bolt 或 Propeller 进行性能提升：

将 PGO、llvm-bolt 或 Propeller 应用于您日常使用的应用程序：

- 选择工具：选择最符合您的需求和专业知识的工具。考虑易用性、性能分析功能和提供的优化技术等因素。
- 性能分析：使用所选工具的性能分析功能收集准确的执行配置文件。确保配置文件代表真实世界的负载。
- 优化：根据配置文件生成优化后的代码。llvm-bolt 和 Propeller 提供链接时优化，而 PGO 在编译期间指导编译器。
- 评估：使用基准测试或真实世界的负载比较优化后应用程序与原始版本的性能。分析性能分析数据和优化报告以了解性能变化。

请记住，这些技术的有效性取决于各种因素，例如应用程序特性、硬件和编译器版本。进行实验并仔细评估，以找到最适合您的特定场景的方法。

## 章节总结

CPU 前端优化总结见表 9。

Table 9: CPU 前端优化总结。

| 转换                               | 如何转换？            | 为什么有用？          | 最适合于             | 由谁完成  |
|----------------------------------|------------------|-----------------|------------------|-------|
| 基本块放置<br>(Basic block placement) | 维护热点代码           | 未取分支更便宜更好的缓存利用率 | ；任何代码，尤其是大量分支的代码 | 具有编译器 |
| 基本块对齐<br>(Basic block alignment) | 使用 NOPs 对齐移动热点代码 | 更好的缓存利用         | 率热循环             | 编译器   |

| 转换                            | 如何转换？              | 为什么有用？    | 最适合于                       | 由谁完成   |
|-------------------------------|--------------------|-----------|----------------------------|--------|
| 函数拆分<br>(Function splitting)  | 将冷代码块拆分并 放置在单独的函数中 | 更好的缓存利用   | 率具有复杂 CFG 的 当热部分之间有 大块冷代码时 | 函数，编译器 |
| 函数重新排序<br>Function splitting) | 将热点函数分组在一 起        | 更好的缓存利 用率 | 许多小型热点                     | 函数 链接器 |

- 代码布局改进经常被低估，最终被忽略和遗忘。CPU 前端性能问题，例如 I-cache 和 ITLB 未命中，浪费了大量的周期，尤其是对于具有大型代码库的应用程序。但即使是小型和中型应用程序也可以从优化机器代码布局中受益。
- 当开发人员试图改善应用程序性能时，他们通常不会首先关注它。他们更喜欢从低垂的果实开始，例如循环展开和矢量化。然而，要知道仅仅通过更好的机器代码布局，您就可能获得额外的 5-10%，这仍然很有用。
- 如果您可以为您的应用程序制定一套典型用例，则通常最好使用 LTO、PGO、BOLT 和其他工具。对于大型应用程序，它是改善机器代码布局的唯一实用方法。

## 10.8 其他优化领域

在本章中，我们将探讨一些优化主题，它们与前三章中涵盖的任何类别没有特别关联，但仍然足够重要，可以在本书中占有一席之地。

## 10.9 优化输入输出

待写

- 网络和存储 IO 内核旁路
- 使用特定于操作系统的 api，例如 mmap(将文件读入地址空间) 与 c++ 流。

### Architecture-Specific Optimizations

Performance considerations on x86, ARM, and RISC-V

Major differences between ISAs

Know capabilities of your ISA

CISC vs RISC code density

Microarchitecture-specific issues

Memory ordering

Memory alignment

4K aliasing

Cache trashing

Non-temporal stores

Instruction latencies and throughput

## 10.10 低延迟优化技术

到目前为止，我们已经讨论了各种旨在改进应用程序整体性能的软件优化。在这一节中，我们将讨论在低延迟系统（例如实时处理和高频交易 (HFT)）中使用的额外优化技术。在这样的环境中，主要的优化目标是使程序的特定部分尽可能快地运行。当您在 HFT 行业工作时，每个微秒和纳秒都至关重要，因为它直接影响利润。通常，低延迟部分会实现实时或 HFT 系统的关键循环，例如移动机械臂或向交易所发送订单。优化关键路径的延迟有时会以牺牲程序其他部分为代价。一些技术甚至会牺牲整个系统的整体吞吐量。

当开发人员优化延迟时，他们会避免在热点路径上支付任何不必要的开销。这通常涉及系统调用、内存分配、I/O 以及任何其他具有非确定性延迟的内容。为了达到最低可能的延迟，热点路径需要提前准备好所有资源并可供其使用。

一个相对简单的方法是预先计算一些会在热点路径上执行的操作。这会带来使用更多内存的代价，这些内存将无法供系统中的其他进程使用，但它可以为您在关键路径上节省一些宝贵的周期。但是，请记住，有时计算内容比从内存中获取结果更快。

既然这是一本关于低级 CPU 性能的书籍，我们将跳过讨论类似于刚才提到的更高层次的技术。相反，我们将讨论如何在关键路径上避免页面错误、缓存未命中、TLB 驱逐和核心节流。

### 10.10.1 避免轻微页面错误

虽然“轻微”这个词存在，但轻微页面错误对运行时延迟的影响并不轻微。回想一下，当用户代码分配内存时，操作系统只会承诺提供一个页面，但不会立即通过提供一个已清零的物理页面来执行承诺。相反，它会等到用户代码第一次访问它时，然后操作系统才会履行其职责。第一次访问新分配的页面会触发轻微页面错误，这是一个由操作系统处理的硬件中断。轻微错误的延迟影响可以从不到 1 微秒到几微秒，尤其是在您使用具有 5 层页表而不是 4 层页表的 Linux 内核时。

如何检测应用程序中运行时的轻微页面错误？一种简单的方法是使用 `top` 实用程序（添加 `-H` 选项以查看线程级别视图）。将 `vMn` 字段添加到默认的显示列选择中，以查看每次显示刷新间隔发生的轻微页面错误数量。Listing 10.10.1 显示了在编译大型 C++ 项目时使用 `top` 命令查看排名前 10 的进程的转储。附加的 `vMn` 列显示了过去 3 秒内发生的轻微页面错误数量。

代码清单：编译大型 C++ 项目时，带有附加 `vMn` 字段的 Linux `top` 命令的转储。

| PID    | USER     | PR | NI | VIRT   | RES    | SHR   | S | %CPU | %MEM | TIME+   | COMMAND | vMn |
|--------|----------|----|----|--------|--------|-------|---|------|------|---------|---------|-----|
| 341763 | dendiba+ | 20 | 0  | 303332 | 165396 | 83200 | R | 99.3 | 1.0  | 0:05.09 | c++     | 13k |
| 341705 | dendiba+ | 20 | 0  | 285768 | 153872 | 87808 | R | 99.0 | 1.0  | 0:07.18 | c++     | 5k  |
| 341719 | dendiba+ | 20 | 0  | 313476 | 176236 | 83328 | R | 94.7 | 1.1  | 0:06.49 | c++     | 8k  |
| 341709 | dendiba+ | 20 | 0  | 301088 | 162800 | 82944 | R | 93.4 | 1.0  | 0:06.46 | c++     | 2k  |
| 341779 | dendiba+ | 20 | 0  | 286468 | 152376 | 87424 | R | 92.4 | 1.0  | 0:03.08 | c++     | 26k |
| 341769 | dendiba+ | 20 | 0  | 293260 | 155068 | 83072 | R | 91.7 | 1.0  | 0:03.90 | c++     | 22k |
| 341749 | dendiba+ | 20 | 0  | 360664 | 214328 | 75904 | R | 88.1 | 1.3  | 0:05.14 | c++     | 18k |
| 341765 | dendiba+ | 20 | 0  | 351036 | 205268 | 76288 | R | 87.1 | 1.3  | 0:04.75 | c++     | 18k |
| 341771 | dendiba+ | 20 | 0  | 341148 | 194668 | 75776 | R | 86.4 | 1.2  | 0:03.43 | c++     | 20k |
| 341776 | dendiba+ | 20 | 0  | 286496 | 147460 | 82432 | R | 76.2 | 0.9  | 0:02.64 | c++     | 25k |

另一种检测运行时轻微页面错误的方法是使用 `perf stat -e page-faults` 附加到正在运行的进程。

在 HFT 世界中，任何超过 0 的页错误都是一个问题。但是对于其他业务领域的低延迟应用程序来说，每秒钟出现 100-1000 次错误的持续情况应该进一步调查。调查运行时轻微页面错误的根本原因可以像启动 `perf record -e page-faults` 然后 `perf report` 一样简单，以定位有问题的源代码行。

为了在运行时避免页面错误惩罚，您应该在启动时预先为应用程序分配所有内存。一个示例代码可能看起来像这样：

```
char *mem = malloc(size);
int pageSize = sysconf(_SC_PAGESIZE)
for (int i = 0; i < size; i += pageSize)
    mem[i] = 0;
```

首先，这段示例代码像往常一样在堆上分配了 `size` 大小的内存。然而，紧接着在分配之后，它会逐个页面访问新分配的内存，确保每个页面都加载到 RAM 中。这种方法有助于避免未来访问时因轻微页面错误造成的运行时延迟。

请看 Listing 197 中更全面的方法，它结合了 `mlock/mlockall` 系统调用对 glibc 分配器进行调整（摘自“实时 Linux Wiki”<sup>197</sup>）。

代码清单：调整 glibc 分配器以锁定内存中的页，并防止将它们释放到操作系统。

<sup>197</sup> Linux 基金会 Wiki：实时应用程序内存 - <https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/memory>

```

#include <malloc.h>
#include <sys/mman.h>

mallopt(M_MMAP_MAX, 0);
mallopt(M_TRIM_THRESHOLD, -1);
mallopt(M_ARENA_MAX, 1);

mlockall(MCL_CURRENT | MCL_FUTURE);

char *mem = malloc(size);
for (int i = 0; i < size; i += sysconf(_SC_PAGESIZE))
    mem[i] = 0;
//...
free(mem);

```

代码 Listing 197 调整了三个 glibc malloc 设置: M\_MMAP\_MAX、M\_TRIM\_THRESHOLD 和 M\_ARENA\_MAX。

- 将 M\_MMAP\_MAX 设置为 0 会禁用底层 mmap 系统调用用于大分配 - 这是必要的，因为当库尝试将 mmap 过的段释放回操作系统时，mlockall 可能会被 munmap 撤销，从而挫败我们的努力。
- 将 M\_TRIM\_THRESHOLD 设置为 -1 可以防止 glibc 在调用 free 后将内存返回给操作系统。正如之前所说，此选项对 mmap 过的段没有影响。
- 最后，将 M\_ARENA\_MAX 设置为 1 可以防止 glibc 通过 mmap 分配多个 arena 以容纳多个内核。请记住，后者会妨碍 glibc 分配器多线程可扩展性特性。

结合起来，这些设置将 glibc 强制转换为堆分配，直到应用程序结束才会将内存释放回操作系统。因此，在上述代码中最后一次调用 free(mem) 之后，堆将保持相同的尺寸。如果在初始化时分配了足够的空间，任何后续运行时调用 malloc 或 new 都只会重用预先分配/预处理过的堆区域中的空间。

更重要的是，由于之前的 mlockall 调用，之前在 for 循环中预处理过的所有堆内存都将保留在 RAM 中 - MCL\_CURRENT 选项锁定所有当前映射的页面，而 MCL\_FUTURE 则锁定所有将来会映射的页面。这种使用 mlockall 的额外好处是，该进程生成的任何线程的堆栈也将被预处理并锁定。为了更好地控制页面锁定，开发人员应该使用 mlock 系统调用，它可以让您选择哪些页面应该保留在 RAM 中。这种技术的缺点是它减少了系统上其他进程可用的内存量。

针对 Windows 的应用程序开发人员应该研究以下 API：使用 VirtualLock 锁定页面，使用 VirtualFree 和 MEM\_DECOMMIT 避免立即释放内存，但不使用 MEM\_RELEASE 标志。

这只是防止运行时轻微错误的两种示例方法。这些技术中的一些或全部可能已经集成到 jemalloc、tcmalloc 或 mimalloc 等内存分配库中。请查看您选择的库的文档，了解可用的功能。

### 10.10.2 高速缓存预热

在某些应用程序中，延迟最敏感的部分代码是最不经常执行的。例如，高频交易应用程序会持续从证券交易所读取市场数据信号，一旦检测到有利信号，就会向交易所发送买入订单。在上例工作负载中，读取市场数据的代码路径是最常执行的，而执行买入订单的代码路径很少执行。

由于市场上的其他参与者也可能捕捉到相同的市场信号，策略的成功在很大程度上取决于我们的反应速度，换句话说，取决于我们发送订单到交易所的速度。当我们希望我们的买入订单尽快到达交易所并利用市场数据中检测到的有利信号时，我们最不想遇到的是在决定起飞的那一刻遇到障碍。

当一段代码路径一段时间没有运行时，它的指令和相关数据很可能从 I-cache 和 D-cache 中被驱逐出去。然后，就在我们需要运行这段关键的很少执行的代码时，我们遇到了 I-cache 和 D-cache 未命中惩罚，这可能会让我们输掉比赛。这就是缓存预热技术可以发挥作用的地方。

缓存预热涉及定期执行延迟敏感的代码以将其保留在缓存中，同时确保它不会执行任何不需要的操作。执行延迟敏感的代码也会通过将延迟敏感的数据带入其中来“预热”D-cache。这种技术通常用于高频交易应用程序。虽然我们不会提供示例实现，但您可以在 CppCon 2018 年闪电演讲：[https://www.youtube.com/watch?v=XzRxikGgaHI<sup>198</sup>](https://www.youtube.com/watch?v=XzRxikGgaHI) 中体验一下。

### 10.10.3 避免 TLB 驱逐

我们从前面章节了解到，TLB 是每个内核的一个快速但有限的虚拟到物理内存地址转换缓存，它减少了耗时的内核页表遍历的需要。当一个进程从一个内核被调度出去，为一个具有完全不同的虚拟地址空间的新进程让路时，属于该内核的 TLB 需要被刷新。除了批发性的 TLB 刷新之外，还有一个更具选择性的过程来使无效的 TLB 条目称为 TLB 驱逐。

与基于 MESI 的协议和每个内核的 CPU 缓存（即 L1、L2 和 LLC）不同，硬件本身无法维护核心到核心之间的 TLB 一致性。因此，这项任务必须由内核通过软件来完成。内核通过一种特定的处理器间中断 (IPI) 来实现这个角色，叫做 TLB 驱逐，在 x86 平台上通过 INVLPG 汇编指令实现。

TLB 驱逐是实现多线程应用程序低延迟时最容易忽视的陷阱之一。为什么？因为在多线程应用程序中，进程线程共享虚拟地址空间。因此，内核必须在参与线程运行的内核的 TLB 之间通信特定类型的对该共享地址空间的更新。例如，常用的系统调用，如 `munmap`（可以禁用 glibc 分配器使用，参见 Section 10.10.1）、`mprotect` 和 `madvise`，会影响内核必须在进程的组成线程之间通信的地址空间更改类型。

尽管开发人员可能避免在他的代码中显式使用这些系统调用，但 TLB 驱逐仍然可能来自外部源 - 例如，分配器共享库或操作系统设施。这种类型的 IPI 不仅会扰乱运行时应用程序性能，而且其影响的程度会随着所涉及线程数量的增加而增大，因为中断是在软件中传递的。

如何检测多线程应用程序中的 TLB 驱逐？一种简单的方法是检查 `/proc/interrupts` 中的 TLB 行。一种检测运行时连续 TLB 中断的有用方法是在查看此文件时使用 `watch` 命令。例如，您可以运行 `watch -n5 -d 'grep TLB /proc/interrupts'`，其中 `-n 5` 选项每 5 秒刷新视图，而 `-d` 则突出显示每次刷新输出之间的差异。

Listing 10.10.3 显示了在运行延迟关键线程的 CPU2 处理器上出现大量 TLB 驱逐的 `/proc/interrupts` 转储。注意其他内核数量级上的差异。在这种情况下，这种行为的罪魁祸首是 Linux 内核的一个名为自动 NUMA 平衡的功能，可以通过 `sysctl -w numa_balancing=0` 轻松禁用。

代码清单：一个`/proc/interrupts` 的转储文件，其中显示了 CPU2 上大量 TLB 被击落的情况

|      | CPU0   | CPU1    | CPU2    | CPU3    |                                    |
|------|--------|---------|---------|---------|------------------------------------|
| ...  |        |         |         |         |                                    |
| NMI: | 0      | 0       | 0       | 0       | Non-maskable interrupts            |
| LOC: | 552219 | 1010298 | 2272333 | 3179890 | Local timer interrupts             |
| SPU: | 0      | 0       | 0       | 0       | Spurious interrupts                |
| ...  |        |         |         |         |                                    |
| IWI: | 0      | 0       | 0       | 0       | IRQ work interrupts                |
| RTR: | 7      | 0       | 0       | 0       | APIC ICR <code>read</code> retries |
| RES: | 18708  | 9550    | 771     | 528     | Rescheduling interrupts            |
| CAL: | 711    | 934     | 1312    | 1261    | Function call interrupts           |
| TLB: | 4493   | 6108    | 73789   | 5014    | TLB shootdowns                     |

<sup>198</sup> 缓存预热技术：<https://www.youtube.com/watch?v=XzRxikGgaHI>

但这不是导致 TLB 驱逐的唯一来源。其他来源还包括透明大页、内存压缩、页面迁移和页面缓存回写。垃圾回收器也可以启动 TLB 驱逐。这些特性在履行其职责的过程中会重新定位页面和/或更改页面权限，这需要更新页表，从而导致 TLB 驱逐。

防止 TLB 驱逐需要限制对共享进程地址空间进行的更新次数。在源代码层面，您应该避免运行时执行上述系统调用列表，即 `munmap`、`mprotect` 和 `madvise`。在操作系统层面，禁用内核功能，这些功能会因其功能而导致 TLB 驱逐，例如透明大页和自动 NUMA 平衡。有关 TLB 驱逐的更多细致讨论，以及它们的检测和预防，请阅读 JabPerf 博客上的文章：<https://www.jabperf.com/how-to-deter-or-disarm-tlb-shootdowns/><sup>199</sup>。

#### 10.10.4 防止意外内核节流

C/C++ 编译器是工程领域的杰出成就。然而，它们有时会生成令人惊讶的结果，可能让你陷入无谓的追查中。一个现实例子是编译器优化器发出了你从未打算过的繁重的 AVX 指令。虽然在更现代的芯片上这个问题小了很多，但许多较老的 CPU（仍在本地和云端积极使用）在执行繁重的 AVX 指令时会出现严重的内核节流/降频现象。如果你的编译器在没有你的明确知情或同意的情况下产生了这些指令，你可能会在应用程序运行期间遇到无法解释的延迟异常。

针对这种情况，如果您不希望使用繁重的 AVX 指令，可以将“`-mprefer-vector-width=###`”添加到您的编译标志中，将最高宽度指令集固定为 128 或 256。同样，如果您整个服务器集群运行的是最新芯片，那么也不用太担心，因为 AVX 指令集的节流影响现在已经微乎其微。

### 10.11 缓慢的浮点运算

一些进行大量浮点值计算的应用程序容易出现一个非常微妙的问题，从而导致性能下降。这个问题出现在应用程序遇到“次规格”浮点值时，我们将在本节讨论。您还可以找到术语“去规格”浮点值，它是指同一个东西。根据 IEEE 标准 754，<sup>200</sup> 次规格值是一个非零数字，其指数小于最小规格数。<sup>201</sup> Listing 201 展示了一个非常简单的次规格值的实例。

在实际应用中，次规格值通常表示一个非常小的信号，以至于它与零无法区分。在音频中，它可能意味着一个非常安静的信号，超出了人类的听觉范围。在图像处理中，它可以表示像素的任何 RGB 颜色分量非常接近于零，等等。有趣的是，次规格值存在于许多生产软件包中，包括天气预报、光线追踪、物理模拟和建模等等。

代码清单：实例化一个正常和次正常的 FP 值

```
unsigned usub = 0x80200000; // -2.93873587706e-39 (subnormal)
unsigned unorm = 0x411a428e; // 9.641248703 (normal)
float sub = *((float*)&usub);
float norm = *((float*)&unorm);
assert(std::fpclassify(sub) == FP_SUBNORMAL);
assert(std::fpclassify(norm) != FP_SUBNORMAL);
```

如果没有次规格值，两个浮点值  $a - b$  的减法可能会溢出并产生零，即使这两个值不相等。次规格值允许计算逐渐失去精度，而不会将结果舍入为零。不过，正如我们稍后将看到的，这也是有代价的。次规格值也可能出现在生产软件中，当一个值在一个循环中不断减小或除法时出现。

从硬件的角度来看，处理次规格值比处理普通浮点值更困难，因为它需要特殊处理，通常被认为是一种特殊情况。应用程序不会崩溃，但性能会下降。生成或消耗次规格值的计算比对普通数字执行类似计算要慢得多，速度可以慢

<sup>199</sup> JabPerf 博客：TLB 驱逐 - <https://www.jabperf.com/how-to-deter-or-disarm-tlb-shootdowns/>

<sup>200</sup> IEEE 754 标准 - <https://ieeexplore.ieee.org/document/8766229>

<sup>201</sup> 次规格数 - [https://en.wikipedia.org/wiki/Subnormal\\_number](https://en.wikipedia.org/wiki/Subnormal_number)

10 倍甚至更多。例如，英特尔处理器目前使用微码 协助 处理次规格值操作。当处理器识别到次规格浮点值时，微码执行器 (MSROM) 将提供必要的微操作 ( $\mu$ ops) 来计算结果。

在许多情况下，次规格值是由算法自然生成的，因此是不可避免的。幸运的是，大多数处理器都提供了将次规格值刷新为零并一开始不生成次规格值的选项。事实上，许多用户宁愿结果稍微不那么准确，也不愿让代码变慢。不过，对于金融软件来说，相反的论点也可以成立：如果你将次规格值刷新为零，你就失去了精度，并且无法将其向上扩展，因为它仍然是零。这可能会让一些客户生气。

假设你可以接受没有次规格值，那么如何检测和禁用它们？虽然可以使用 Listing 201 所示的运行时检查，但在整个代码库中插入它们并不实际。使用 PMU（性能监控单元）检测应用程序是否生成次规格值是一种更好的方法。在英特尔 CPU 上，你可以收集 FP\_ASSIST\_ANY 性能事件，每次使用次规格值时都会增加该事件。TMA 方法将这种瓶颈归类为“Retiring”类别，是的，这是高“Retiring”值不好的情况之一。

一旦确认存在次规格值，你可以启用 FTZ 和 DAZ 模式：

- DAZ (Denormals Are Zero)。任何低于正常值的输入在使用之前都被替换为零。
- FTZ (Flush To Zero)。任何会变为低于正常值的输出都替换为零。

启用它们后，CPU 浮点运算中就不需要昂贵地处理次规格值了。在 x86 平台上，MXCSR（全局控制和状态寄存器）中有两个独立的位字段。在 ARM Aarch64 中，两种模式由 FPCR 控制寄存器的 FZ 和 AH 位控制。如果你用 `-ffast-math` 编译你的应用程序，就不用担心了，编译器会自动在程序开始时插入所需代码启用这两个标志。`-ffast-math` 编译器选项有点过载，所以 GCC 开发人员创建了一个单独的 `-mdaz-ftz` 选项，只控制次规格值的行为。如果你想从源代码控制它，Listing 201 显示了你可以使用的示例。如果你选择这个选项，请避免频繁更改 MXCSR 寄存器，因为操作相对昂贵。读取 MXCSR 寄存器有相当长的延迟，写入寄存器是一个序列化指令。

代码清单: 手动启用 FTZ 和 DAZ 模式

```
unsigned FTZ = 0x8000;
unsigned DAZ = 0x0040;
unsigned MXCSR = _mm_getcsr();
_mm_setcsr(MXCSR | FTZ | DAZ);
```

请注意，FTZ 和 DAZ 模式都与 IEEE 754 标准不兼容。它们是在硬件中实现的，以提高在溢出常见且生成非规格化结果不必要时的应用程序性能。通常，我们观察到一些使用次规格值的生产浮点应用程序速度提高了 3% - 5%，有时甚至高达 50%。

## 10.12 系统调优

在成功完成了利用 CPU 微架构所有复杂设施调整应用程序的所有艰苦工作之后，我们最不想看到的是系统固件、操作系统或内核破坏我们所有的努力。即使是最精细调整的应用程序，如果被系统管理中断 (SMI)（一种 BIOS 中断，用于停止整个操作系统以执行固件代码）间歇性地中断，也毫无意义。这样的中断每次可能会运行 10 到 100 毫秒。

公平地说，开发人员通常对应用程序执行环境几乎没有控制权。当我们运送产品时，不切实际地调整客户可能拥有的每个设置。通常，足够大的组织会有单独的运维团队来处理这类问题。尽管如此，与这些团队成员沟通时，了解还有哪些因素会限制应用程序表现出最佳性能是很重要的。

正如 Section 2.1 所示，现代系统中有许多需要调整的地方，避免系统级干扰并非易事。基于 x86 的服务器部署的性能调优手册的一个例子是 Red Hat 的 指南: <https://access.redhat.com/sites/default/files/attachments/201501-perf-brief-low-latency-tuning-rhel7-v2.1.pdf><sup>202</sup>。在那里，您将找到消除或显著减少缓存破坏中断的提示，这些中断来自系

<sup>202</sup> Red Hat 低延迟调优指南 - <https://access.redhat.com/sites/default/files/attachments/201501-perf-brief-low-latency-tuning-rhel7-v2.1.pdf>

统 BIOS、Linux 内核和设备驱动程序等许多应用程序干扰源。这些指南应作为所有新服务器构建的基础映像，然后再将任何应用程序部署到生产环境中。

当涉及调整特定系统设置时，它并不总是简单的“是”或“否”答案。例如，您的应用程序是否会从其所在环境中启用的同步多线程 (SMT) 功能中受益，一开始并不清楚。一般指导原则是仅为 IPC 相对较低的异构工作负载<sup>203</sup> 启用 SMT。另一方面，如今的 CPU 制造商提供具有如此高的内核数量的处理器，以至于 SMT 比过去少得多。然而，这只是一个通用指南，正如本书迄今强调的那样，最好是自己去测量。

大多数开箱即用的平台都配置为在可能的情况下优化吞吐量并节省电量。但是，有一些行业具有实时要求，它们更关心降低延迟而不是其他一切。这样的行业的一个例子是用于汽车装配线的机器人。此类机器人执行的动作由外部事件触发，并且通常有一个预定的时间预算来完成，因为下一个中断即将到来（通常称为“控制环路”）。满足此类平台的实时目标可能需要牺牲机器的整体吞吐量或允许其消耗更多能量。该领域流行的技术之一是禁用处理器睡眠状态<sup>204</sup>，使其时刻准备立即做出反应。另一种有趣的方法称为缓存锁定，<sup>205</sup> 其中 CPU 缓存的部分保留用于特定数据集；它有助于简化应用程序内的内存延迟。

### 10.13 案例研究：对最后一级缓存大小的敏感性

本案例研究的目的是展示如何确定应用程序是否对最后一级缓存 (LLC) 的大小敏感。利用这些信息，您可以在购买计算系统硬件组件时做出明智的决策。同样，您以后可以确定对其他因素（例如内存带宽、核心数量、处理器频率等）的敏感性，并可能购买更便宜的计算机，同时保持相同的性能水平。

在这个案例研究中，我们对同一组应用程序运行多次，LLC 大小各不相同。现代服务器处理器允许用户控制 LLC 空间分配给处理器线程。通过这种方式，用户可以限制每个线程仅使用其分配的共享资源数量。此类设施通常称为服务质量 (QoS) 扩展。它们可用于优先处理关键性能应用程序并减少与同一系统中其他线程的干扰。除了 LLC 分配之外，QoS 扩展还支持限制内存读取带宽。

我们的分析将帮助我们识别当 LLC 大小减小时性能显著下降的应用程序。我们称这样的应用程序对 LLC 大小敏感。我们还确定了不敏感的应用程序，即 LLC 大小不会影响性能。此结果可用于正确调整处理器 LLC 的大小，特别是考虑到市场上可用范围广泛。例如，我们可以确定应用程序是否可以从更大的 LLC 中受益，即投资新硬件是否合理。相反，如果应用程序对小缓存大小已足够，那么我们可以购买更便宜的处理器。

对于本案例研究，我们使用 AMD Milan 处理器，但其他服务器处理器，例如 Intel Xeon 和 ARM ThunderX，也包含硬件支持，允许用户控制 LLC 空间和内存读取带宽分配给处理器线程。

#### 目标机器：AMD EPYC 7313P

我们使用了一个服务器系统，该系统配有 16 核 AMD EPYC 7313P 处理器，代号为 Milan，AMD 于 2021 年推出。该系统的关键特性在表 10 中指定。

Table 10: 实验所用服务器的主要特性。

| 特性            | 值              |
|---------------|----------------|
| 处理器 Processor | AMD EPYC 7313P |

<sup>203</sup> 即，当兄弟线程执行不同的指令模式时

<sup>204</sup> 电源管理状态：P 状态、C 状态。详细信息请参见此处：<https://software.intel.com/content/www/us/en/develop/articles/power-management-states-p-states-c-states-and-package-c-states.html>。

<sup>205</sup> 缓存锁定。缓存锁定技术的调查 [Mittal, 2016]。将缓存的一部分伪锁定的示例，然后将其作为 Linux 文件系统中的字符设备公开并可用于 mmap：<https://events19.linuxfoundation.org/wp-content/uploads/2017/11/Introducing-Cache-Pseudo-Locking-to-Reduce-Memory-Access-Latency-Reinette-Chatre-Intel.pdf>。

| 特性                              | 值                                                                                              |
|---------------------------------|------------------------------------------------------------------------------------------------|
| 核心数 x 线程数 Cores<br>x threads    | $16 \times 2$                                                                                  |
| 配置 Configuration                | $4 \text{ CCX} \times 4 \text{ 个核心/CCX}$                                                       |
| 频率 Frequency                    | 3.0/3.7 GHz, 基础/最大                                                                             |
| L1 缓存 (I, D) L1 cache<br>(I, D) | 8 通道, 32 KiB (每个核心)                                                                            |
| L2 缓存 L2 cache                  | 8 通道, 512 KiB (每个核心)                                                                           |
| LLC                             | 16 通道, 32 MiB, 非包容式 (每个 CCX) 16-ways, 32 MiB, non-inclusive (per CCX)                          |
| 主内存 Main Memory                 | 512 GiB DDR4, 8 个通道, 标称峰值带宽: 204.8 GB/s 512 GiB DDR4, 8 channels, nominal peak BW: 204.8 GB/s, |
| TurboBoost TurboBoost           | 已禁用 Disabled                                                                                   |
| 超线程 Hyperthreading              | 已禁用 (1 个线程/核心) Disabled (1 thread/core)                                                        |
| 操作系统 OS                         | Ubuntu 22.04, 内核 5.15.0-76 Ubuntu 22.04, kernel 5.15.0-76                                      |

## 10.14 AMD Milan 7313P 处理器的集群式内存层次结构

图 73 展示了 AMD Milan 7313P 处理器的集群式内存层次结构。它由四个核心复合芯片 (CCD) 组成，这些 CCD 通过一个 I/O 芯片连接彼此和片外内存。每个 CCD 集成一个核心复合体 (CCX) 和一个 I/O 连接。反过来，每个 CCX 都有四个 Zen3 内核，可以运行八个线程，共享一个 32 MiB 的受害者 LLC，即 LLC 填充了从 CCX 的四个 L2 缓存驱逐的缓存行。

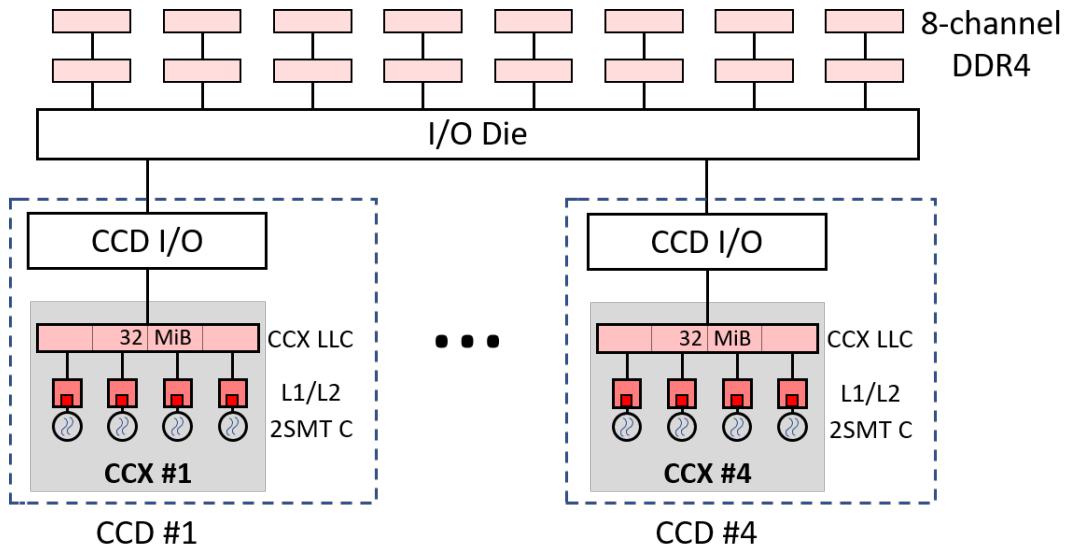


Figure 73: AMD Milan 7313P 处理器的集群式内存层次结构

虽然总共有 128 MiB 的 LLC，但 CCX 的四个内核无法将缓存行存储在除其自己的 32 MiB LLC ( $32 \text{ MiB}/\text{CCX} \times 4 \text{ CCX}$ ) 以外的 LLC 中。由于我们将运行单线程基准测试，因此我们可以关注单个 CCX。实验中 LLC 的大小将在 0 到 32 MiB 之间变化，步长为 2 MiB。

## 工作负载: SPEC CPU2017

我们使用 SPEC CPU2017 套件中的部分基准测试。<sup>206</sup> SPEC CPU2017 包含一组行业标准性能基准测试，可对处理器、内存子系统和编译器进行压力测试。它被广泛用于比较高性能系统的性能。它也广泛用于计算机架构研究。

具体来说，我们从 SPEC CPU2017 中选择了 15 个内存密集型基准测试（6 个 INT 和 9 个 FP），正如 [Navarro-Torres et al., 2019] 中建议的那样。这些应用程序使用 GCC 6.3.1 和以下编译器选项编译：-g -O3 -march=native -fno-unsafe-math-optimizations -fno-tree-loop-vectorize，正如 SPEC 在套件提供的配置文件中指定的。

### 10.14.1 控制和监控 LLC 分配

为了监控和执行 LLC 分配和内存读取带宽的限制，我们将使用 AMD64 技术平台服务质量扩展 [Advanced Micro Devices, 2022]。用户可以通过特定于模型的寄存器 (MSR) 的库来管理此 QoS 扩展。首先，必须通过写入 PQR\_ASSOC 寄存器 (MSR 0xC8F) 为线程或一组线程分配资源管理标识符 (RMID) 和服务类别 (COS)。以下是硬件线程 1 的示例命令：

```
# 写入 PQR_ASSOC (MSR 0xC8F): RMID=1, COS=2 -> (COS << 32) + RMID
$ wrmsr -p 1 0xC8F 0x20000001
```

其中 -p 1 表示硬件线程 1。我们显示的所有 rdmsr 和 wrmsr 命令都需要 root 访问权限。

LLC 空间管理通过写入每个线程的 16 位二进制掩码执行。掩码的每个位允许线程使用给定的十六分之一的 LLC（在 AMD Milan 7313P 的情况下为  $1/16 = 2 \text{ MiB}$ ）。多个线程可以使用相同的片段，这意味着竞争性地共享相同的 LLC 子集。

要设置线程 1 对 LLC 使用的限制，我们需要写入 L3\_MASK\_n 寄存器，其中 n 是 COS，即相应 COS 可以使用的缓存分区。例如，要限制线程 1 仅使用 LLC 可用空间的一半，请运行以下命令：

```
# 写入 L3_MASK_2 (MSR 0xC92): 0x00FF (LLC 空间的一半)
$ wrmsr -p 1 0xC92 0x00FF
```

现在，要监控硬件线程 1 对 LLC 的使用情况，首先我们需要将监控标识符 RMID 与 LLC 监控事件 (L3 缓存占用率监控，evtID 0x1) 关联起来。我们通过写入 QM\_EVTSEL 控制寄存器 (MSR 0xC8D) 来做到这一点。之后，我们应该读取 QM\_CTR 寄存器 (MSR 0xC8E)：

```
# 写入 QM_EVTSEL (MSR 0xC8D): RMID=1, evtID=1 -> (RMID << 32) + evtID
$ wrmsr -p 1 0xC8D 0x10000001
# 读 QM_CTR (MSR 0xC8E)
$ rdmsr -p 1 0xC8E
```

这将使我们能够估计缓存行中的 LLC 使用情况<sup>207</sup>。要将此值转换为字节，我们需要将 rdmsr 命令返回的值乘以缓存行大小。

同样，可以限制分配给线程的内存读取带宽。这是通过将无符号整数写入特定的 MSR 寄存器来实现的，该寄存器以 1/8 GB/s 的增量设置最大读取带宽。欢迎感兴趣的读者阅读 [Advanced Micro Devices, 2022] 了解更多详细信息。

## 评估指标

用于量化应用程序性能的最终指标是执行时间。为了分析内存层次结构对系统性能的影响，我们还将使用以下三个指标：1) CPI，每条指令的周期数<sup>208</sup>，2) DMPKI，每千条指令的 LLC 中的需求缺失，以及 3) MPKI，每千条指令的

<sup>206</sup> SPEC CPU® 2017 - <https://www.spec.org/cpu2017/>

<sup>207</sup> AMD 文档 [Advanced Micro Devices, 2022] 更准确地使用了术语 L3 缓存转换因子，可以通过 cpuid 指令确定。

<sup>208</sup> 我们使用 CPI 而不是每条指令的时间，因为我们假设 CPU 频率在实验过程中不会改变。

LLC 中的总缺失（需求 + 预取）。虽然 CPI 与应用程序性能直接相关，但 DMPKI 和 MPKI 不一定会影响性能。表 11 显示了用于从特定硬件计数器计算每个指标的公式。有关每个计数器的详细描述，请参见 AMD 的处理器编程参考 [Advanced Micro Devices, 2021]。

Table 11: 案例研究中使用的指标计算公式。

| 指标    | 公式                                                                                                                                                     |
|-------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| CPI   | 非暂停周期 (PMCx076) / 退休指令 (PMCx0C0) Cycles not in Halt (PMCx076) / Retired Instructions (PMCx0C0)                                                         |
| DMPKI | 需求数据缓存填充 <sup>209</sup> (PMCx043) / (退休指令 (PMCx0C0) / 1000) Demand Data Cache Fills <sup>210</sup> (PMCx043) / (Retired Instructions (PMCx0C0) / 1000) |
| MPKI  | L3 缺失 <sup>211</sup> (L3PMCx04) / (退休指令 (PMCx0C0) / 1000) L3 Misses <sup>212</sup> (L3PMCx04) / (Retired Instructions (PMCx0C0) / 1000)                |

硬件计数器可以通过 MSR 进行配置和读取。配置包括指定要监视的事件以及如何监视它。在我们的系统中，每个线程有 6 个核心计数器，每个 L3-CCX 有 6 个计数器，以及 4 个数据结构计数器。访问核心事件通过写入 `PERF_CTL[0-5]` 控制寄存器 (MSR 0xC001020[0,2,4,6,8,A]) 完成。`PERF_CTR[0-5]` 寄存器 (MSR 0xC001020[1,3,5,7,9,B]) 是与这些控制寄存器关联的计数器。例如，对于计数器 0 来收集在硬件线程 1 上运行的应用程序退休的指令数，执行以下命令：

```
$ wrmsr -p 1 0xC0010200 0x5100C0
$ rdmsr -p 1 0xC0010201
```

其中 `-p 1` 表示硬件线程 1，`0xC0010200` 是计数器 0 (`PERF_CTL[0]`) 的控制 MSR，`0x5100C0` 指定要测量的事件标识符（已退休指令，`PMCx0C0`）以及如何测量它（用户事件）。使用 `wrmsr` 完成配置后，可以使用 `rdmsr` 命令读取收集已退休指令数的计数器 0。

类似地，通过写入 L3 控制寄存器 (MSR 0xC001023[0,2,4,6,8,A]) 并读取其关联计数器 (MSR 0xC001023[1,3,5,7,9,B]) 来访问 L3 缓存事件。最后，通过写入 `DF_PERF_CTL[0-3]` 控制寄存器 (MSR 0xC001024[0,2,4,6]) 并读取其关联的 `DF_PERF_CTR[0-3]` 寄存器 (MSR 0xC001024[1,3,5,7]) 来访问数据结构事件<sup>213</sup>。

本案例研究中使用的方法在 [Navarro-Torres et al., 2023] 中更详细地描述。可以在以下公共存储库中找到重现实验所需的代码和信息：<https://github.com/agusnt/BALANCER>。

## 结果

我们在系统中单独运行一套 SPEC CPU2017 基准测试，仅使用一个实例和单个硬件线程。重复这些运行，同时将可用 LLC 大小从 0 更改为 32 MiB，步长为 2 MiB。图 74 从左到右以图形方式显示每个分配的 LLC 大小的 CPI、DMPKI 和 MPKI。对于 CPI 图表，Y 轴上的较低值意味着更好的性能。此外，由于系统上的频率是固定的，因此 CPI 图表反映了绝对分数。例如，具有 32 MiB LLC 的 520.omnetpp (虚线) 比 0 MiB LLC 快 2.5 倍。

对于 DMPKI 和 MPKI 图表，Y 轴上的值越低越好。对应于 503.bwaves (实线)、520.omnetpp (虚线) 和 554.roms (虚线) 的三条线代表了所有应用程序中观察到的三个主要趋势。我们不显示其余基准测试。

<sup>209</sup> 我们使用了 `MemIoRemote` 和 `MemIoLocal` 这两个子变量，它们从连接在远程/本地 NUMA 节点上的 DRAM 或 IO 请求填充数据缓存。

<sup>210</sup> 我们使用了 `MemIoRemote` 和 `MemIoLocal` 这两个子变量，它们从连接在远程/本地 NUMA 节点上的 DRAM 或 IO 请求填充数据缓存。

<sup>211</sup> 我们仅使用掩码计算 L3 缺失，具体为 `L3Event[0x0300C00000400104]`。

<sup>212</sup> 我们仅使用掩码计算 L3 缺失，具体为 `L3Event[0x0300C00000400104]`。

<sup>213</sup> 在我们的研究中，我们没有使用数据结构计数器。

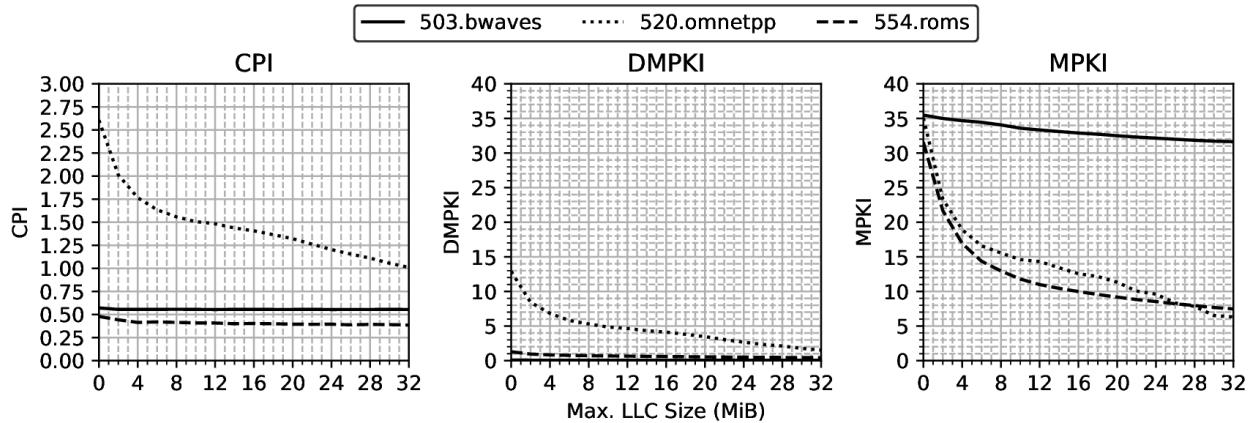


Figure 74: 随着 LLC 分配限制增加 (2 MiB 步长), CPI, DMPKI 和 MPKI.

在 CPI 和 DMPKI 图表中可以区分两种行为。一方面, 520.omnetpp 利用了其在 LLC 中的可用空间: 随着 LLC 中分配的空间增加, CPI 和 DMPKI 都显着降低。我们可以说 520.omnetpp 的行为对 LLC 可用大小敏感。增加分配的 LLC 空间可以提高性能, 因为它避免了驱逐将来会使用的缓存行。

相比之下, 503.bwaves 和 554.roms 不会利用所有可用的 LLC 空间。对于这两个基准测试, 随着 LLC 中的分配限制增加, CPI 和 DMPKI 大致保持不变。我们可以说这两个应用程序的性能对其在 LLC 中的可用空间不敏感。如果我们的应用程序表现出这种行为, 我们可以选择具有较小 LLC 尺寸的处理器而不牺牲性能。

现在让我们分析 MPKI 图表, 其中结合了 LLC 需求缺失和预取请求。首先, 我们可以看到 MPKI 值总是远高于 DMPKI 值。也就是说, 大多数块都是由预取器从内存加载到芯片内层次结构中的。这种行为是由于预取器在预加载私有缓存以供使用的数据方面非常高效, 从而消除了大多数需求缺失。

对于 503.bwaves, 我们观察到 MPKI 与 CPI 和 DMPKI 图表大致保持在相同水平。基准测试中可能没有太多数据重用和/或内存流量非常低。520.omnetpp 工作负载的行为与我们之前确定的相同: MPKI 随着可用空间的增加而减少。但是对于 554.roms, MPKI 图表显示随着可用空间增加, 总缺失急剧下降, 而 CPI 和 DMPKI 保持不变。在这种情况下, 基准测试中存在数据重用, 但这对性能无关紧要。预取器可以提前获取所需数据, 消除需求缺失, 无论 LLC 中可用空间如何。但是, 随着可用空间减少, 预取器无法在 LLC 中找到块并且必须从内存加载它们的可能性会增加。因此, 为这类应用程序提供更多 LLC 容量不会直接提升其性能, 但会使系统受益, 因为它减少了内存流量。

通过查看 CPI 和 DMPKI, 我们最初认为 554.roms 对 LLC 大小不敏感。但通过分析 MPKI 图表, 我们需要重新考虑我们的陈述并得出结论, 即 554.roms 也对 LLC 大小敏感, 因此最好不要限制其可用 LLC 空间, 以免增加内存带宽消耗。更高的带宽消耗可能会增加内存访问延迟, 进而意味着系统上运行的其他应用程序的性能下降 [Navarro-Torres et al., 2023]。

## 问题和练习

1. 完成 `perf-ninja::lto` 和 `perf-ninja::io_opt1` 实验作业。
2. 运行您日常使用的应用程序。找到热点。检查它是否可以从本章讨论的任何技术中受益。

## 章节总结

# 11 优化多线程应用

现代 CPU 每年都在增加越来越多的核心。截至 2020 年，你可以购买到一款 x86 服务器处理器，其核心数量超过 50 个！而拥有 8 个执行线程的中档台式机也是相当常见的配置。由于每个 CPU 中有如此强大的处理能力，如何高效利用所有的硬件线程成为了挑战。为了确保应用的未来成功，准备软件以便与不断增长的 CPU 核心数量良好地扩展非常重要。

多线程应用具有其自身的特点。在处理多个线程时，单线程执行的某些假设会失效。例如，我们不能再通过查看单个线程来识别热点，因为每个线程可能都有自己的热点。在流行的生产者-消费者<sup>214</sup>设计中，生产者线程可能大部分时间都在休眠。对这样一个线程进行分析不会揭示出我们的多线程应用为何扩展性不佳的原因。

## 11.1 性能扩展和开销

当处理单线程应用程序时，优化程序的某个部分通常会对性能产生积极的影响。然而，对于多线程应用程序来说，情况并非总是如此。有些应用程序中，线程 A 执行一个长时间运行的操作，而线程 B 则早早地完成了其任务，只是等待线程 A 完成。无论我们如何改进线程 B，应用程序的延迟都不会减少，因为它将受到运行时间较长的线程 A 的限制<sup>215</sup>。

这种影响被广泛称为安达尔定律<sup>216</sup>，它规定了并行程序的加速度受其串行部分的限制。图 75 描绘了理论上的加速度极限，作为处理器数量的函数。对于一个其中有 75% 是并行的程序，加速度因子将收敛到 4。

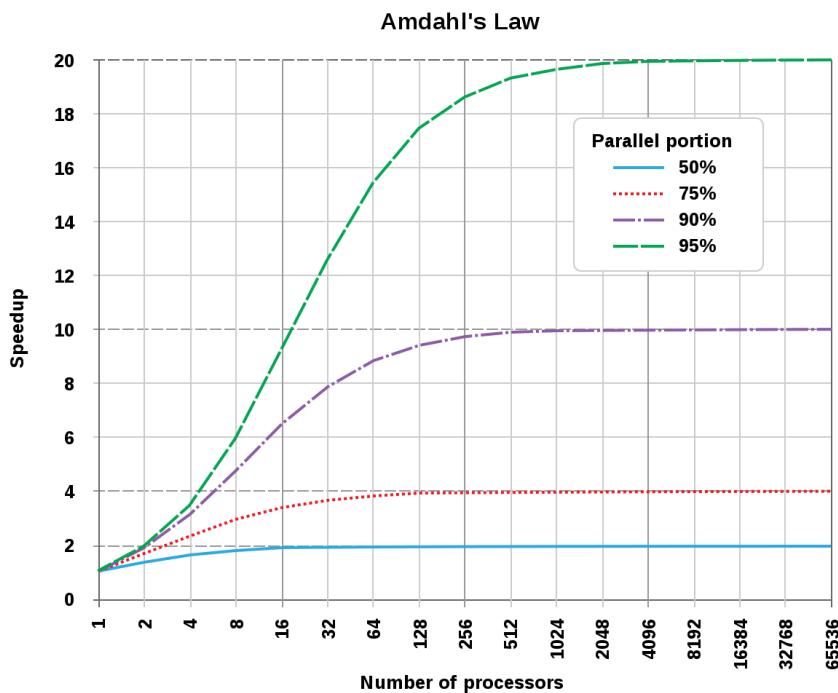


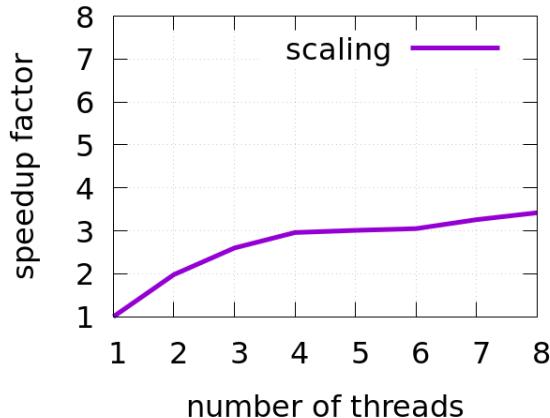
Figure 75: 根据安达尔定律，程序执行的理论加速度随处理器数量的变化。© Image by Daniels220 via Wikipedia.

<sup>214</sup> Producer-consumer pattern - [https://en.wikipedia.org/wiki/Producer-consumer\\_problem](https://en.wikipedia.org/wiki/Producer-consumer_problem)

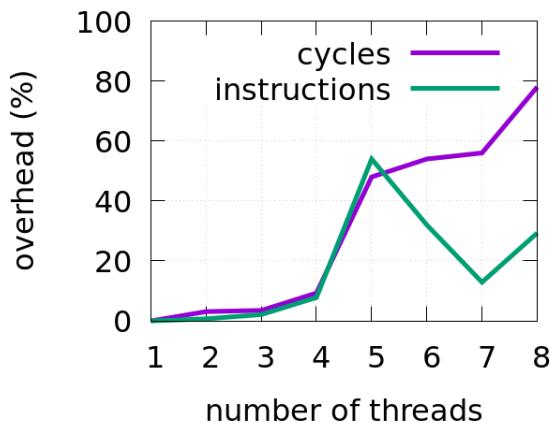
<sup>215</sup> 这不一定总是成立。例如，资源在线程/核心之间共享（如缓存）可能限制扩展性。此外，计算密集型基准测试往往只能在物理（而不是逻辑）核心数量上进行扩展，因为两个相邻的硬件线程共享同一个执行引擎。

<sup>216</sup> 安达尔定律 - <https://zh.wikipedia.org/wiki/%E5%AE%89%E8%BE%BE%E5%B0%94%E6%AE%B5%E6%97%B6%E5%AE%9A%E5%BE%8B>。

图 76a 显示了来自 Starbench 并行基准套件的 h264dec 基准测试的性能扩展情况。我在拥有 4 个核心/8 个线程的 Intel Core i5-8259U 上进行了测试。请注意，在使用了 4 个线程之后，性能并没有显著提高。很可能，获取一个拥有更多核心的 CPU 不会提高性能。<sup>217</sup>



(a) 使用不同线程数量的性能扩展。



(b) 使用不同线程数量的开销。

Figure 76: 在 Intel Core i5-8259U 上的 h264dec 基准测试的性能扩展和开销。

事实上，进一步增加计算节点可能会导致逆向加速。这种效应由 Neil Gunther 解释为通用可扩展性法则<sup>218</sup> (USL)，它是安达尔定律的一个扩展。USL 描述了计算节点（线程）之间的通信作为性能的另一个限制因素。随着系统的扩展，开销开始阻碍性能的增长。超过临界点后，系统的能力开始下降（见图 77）。USL 被广泛用于对系统的容量和可扩展性建模。

由 USL 描述的减速是由多种因素驱动的。首先，随着计算节点数量的增加，它们开始竞争资源（争用）。这导致额外的时间用于同步这些访问。另一个问题是资源在许多工作线程之间共享。我们需要在许多工作线程之间保持共享资源的一致状态（一致性）。例如，当多个工作线程频繁地更改全局可见对象时，这些更改需要广播到使用该对象的所有节点。突然之间，由于额外的一致性维护需求，通常的操作开始花费更多的时间来完成。在 Intel Core i5-8259U 上，h264dec 基准测试的通信开销可以在图 76b 中观察到。请注意，随着我们为任务分配超过 4 个线程，图表表明开销以经过的核心周期数的形式增加。<sup>219</sup>

优化多线程应用程序不仅涉及到本书迄今描述的所有技术，还涉及到检测和减轻争用和一致性的前述影响。下一小

<sup>217</sup> 然而，它会受益于频率更高的 CPU。

<sup>218</sup> 通用可扩展性法则 - [http://www.perfdynamics.com/Manifesto/USLscalability.html#tth\\_sEc1](http://www.perfdynamics.com/Manifesto/USLscalability.html#tth_sEc1)。

<sup>219</sup> 使用 5 和 6 个工作线程时，已完成的指令数量出现了一个有趣的峰值。这应该通过对工作负载进行分析来进行调查。

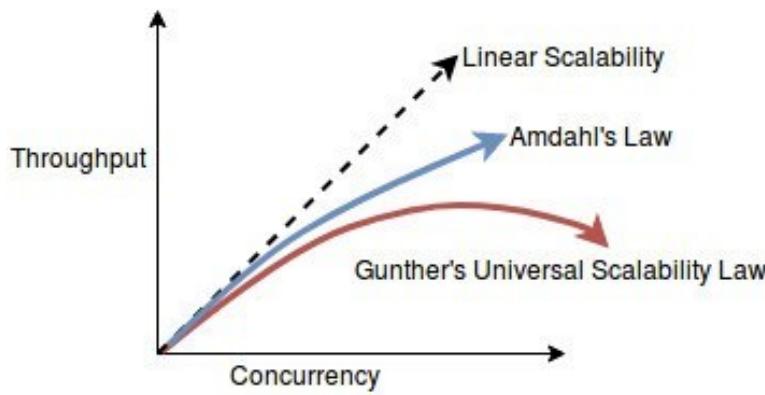


Figure 77: 通用可扩展性法则和安达尔定律。© Image by Neha Bhardwaj via Knoldus Blogs.

节将描述针对调优多线程程序的这些额外挑战的技术。

## 11.2 并行效率指标

在处理多线程应用程序时，工程师们应该谨慎分析诸如 CPU 利用率和 IPC（见 Chapter 4）等基本指标。某个线程可能表现出高 CPU 利用率和高 IPC，但实际上可能只是在一个锁上旋转。这就是为什么在评估应用程序的并行效率时，建议使用有效 CPU 利用率，该指标仅基于有效时间。<sup>220</sup>

### 11.2.1 有效 CPU 利用率

该指标表示应用程序有效地利用了可用的 CPU。它显示了系统上所有逻辑 CPU 的平均 CPU 利用率百分比。CPU 利用率指标仅基于有效时间，不包括并行运行时系统<sup>221</sup>和自旋时间引入的开销。100% 的 CPU 利用率意味着您的应用程序在运行期间始终使所有逻辑 CPU 内核保持忙碌 [Intel, 2023a]。

对于指定的时间间隔 T，可以计算有效 CPU 利用率如下：

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

### 11.2.2 线程数

应用程序通常具有可配置的线程数，这使它们能够在具有不同核心数的平台上有效运行。显然，使用比系统上可用的线程数少的线程来运行应用程序会浪费其资源。另一方面，运行过多的线程可能会导致较高的 CPU 时间，因为其中一些线程可能正在等待其他线程完成，或者时间可能被用于上下文切换。

除了实际的工作线程外，多线程应用程序通常还具有辅助线程：主线程、输入和输出线程等。如果这些线程消耗了大量时间，它们就需要专用的硬件线程。这就是为什么了解总线程数并正确配置工作线程数很重要。

为了避免线程创建和销毁的惩罚，工程师通常会分配一个线程池<sup>222</sup>，其中有多个线程等待由监督程序分配的任务以供并发执行。这对执行短暂任务特别有益。

<sup>220</sup> 诸如 Intel VTune Profiler 等性能分析工具可以区分在线程旋转时采集的分析样本。它们通过为每个样本提供调用堆栈来完成这项工作（见 Section 5.5.3）。

<sup>221</sup> 类似 pthread、OpenMP 和 Intel TBB 等线程库和 API 具有它们自己的线程创建和管理开销。

<sup>222</sup> 线程池 - [https://en.wikipedia.org/wiki/Thread\\_pool](https://en.wikipedia.org/wiki/Thread_pool)。

### 11.2.3 等待时间

等待时间发生在软件线程由于阻塞或导致同步的 API 而等待时。等待时间是每个线程的；因此，总等待时间可能超过应用程序经过的时间 [Intel, 2023a]。

线程可以由于同步或抢占而被操作系统调度程序从执行中切换掉。因此，等待时间可以进一步分为同步等待时间和抢占等待时间。大量的同步等待时间很可能表明应用程序具有高度争用的同步对象。我们将在接下来的章节中探讨如何找到它们。显着的抢占等待时间可以表明线程过度订阅问题 (oversubscription)<sup>223</sup>，这可能是由于大量的应用程序线程或与操作系统线程或系统上其他应用程序的冲突引起的。在这种情况下，开发人员应考虑减少线程总数或增加每个工作线程的任务粒度。

### 11.2.4 自旋时间

自旋时间是 CPU 忙于等待时间。当软件线程等待时，这种情况经常发生，因为同步 API 导致 CPU 轮询 [Intel, 2023a]。实际上，内核同步原语的实现更倾向于在一段时间内锁定旋转，而不是立即进行线程上下文切换（这是昂贵的）。然而，过多的自旋时间可能反映了无法进行有效工作的机会的丧失。

## 11.3 使用 Intel VTune Profiler 进行分析

Intel VTune Profiler 有一种专门针对多线程应用程序的分析类型，称为线程分析。其摘要窗口（见图 78）显示了有关整个应用程序执行的统计信息，识别了我们在 Section 11.2 中描述的所有指标。从有效 CPU 利用率直方图中，我们可以了解到有关捕获的应用程序行为的几个有趣事实。首先，平均而言，同时仅利用了 5 个硬件线程（图表中的逻辑核心）。其次，所有 8 个硬件线程同时活跃的情况非常罕见。

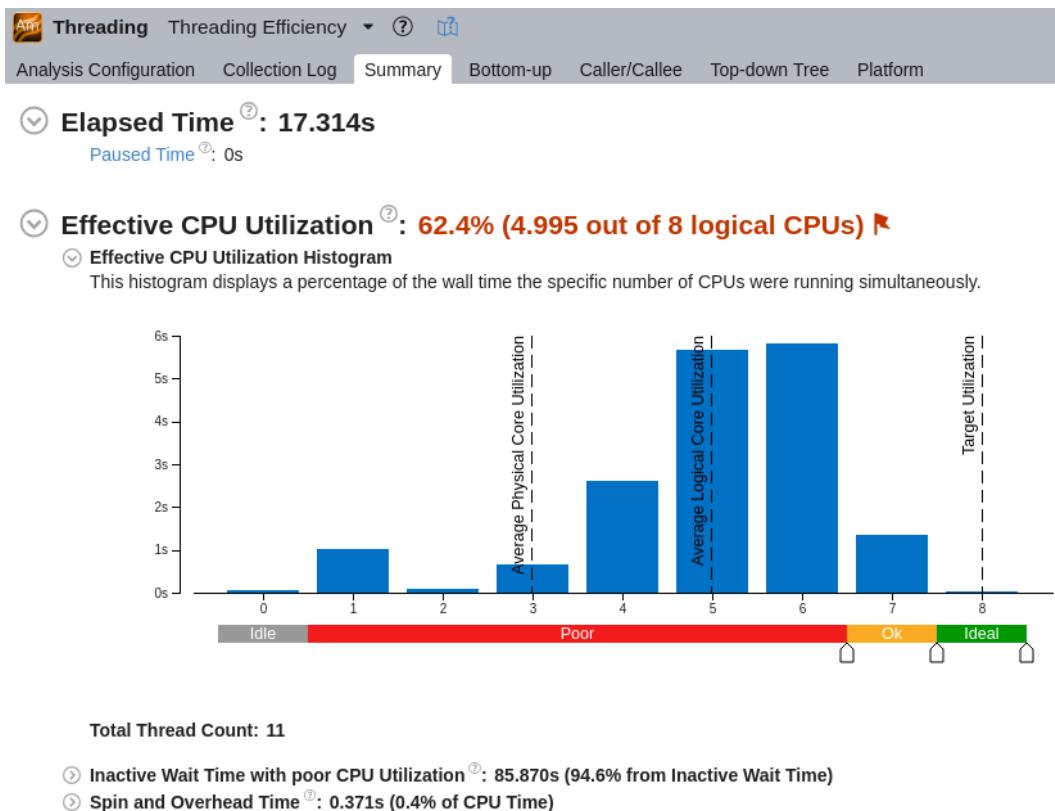


Figure 78: 来自 Phoronix 测试套件中 x264 基准测试的 Intel VTune Profiler 线程分析摘要。

<sup>223</sup> 线程过度订阅 - <https://software.intel.com/en-us/vtune-help-thread-oversubscription>。

### 11.3.1 查找开销大的锁

接下来，工作流程建议我们识别最有争议的同步对象。图 79 显示了这些对象的列表。我们可以看到 `__pthread_cond_wait` 显然突出，但由于程序中可能有几十个条件变量，我们需要知道哪一个是导致 CPU 利用率不佳的原因。

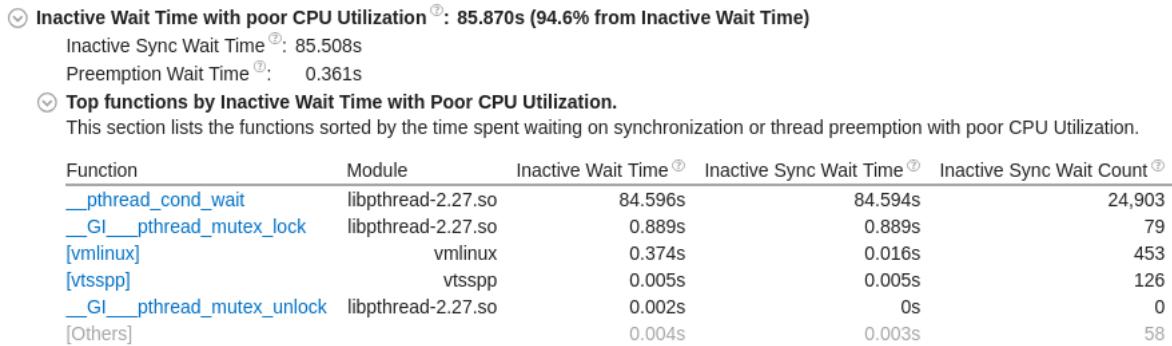


Figure 79: 显示了来自 Phoronix 测试套件中 x264 基准测试的 Intel VTune Profiler 线程分析，显示了最有争议的同步对象。

要找出原因，我们可以简单地点击 `__pthread_cond_wait`，这将带我们到底部向上视图，如图 80 所示。我们可以看到导致线程等待条件变量的最频繁路径（等待时间的 47%）：`__pthread_cond_wait <- x264_8_frame_cond_wait <- mb_analyse_init`。

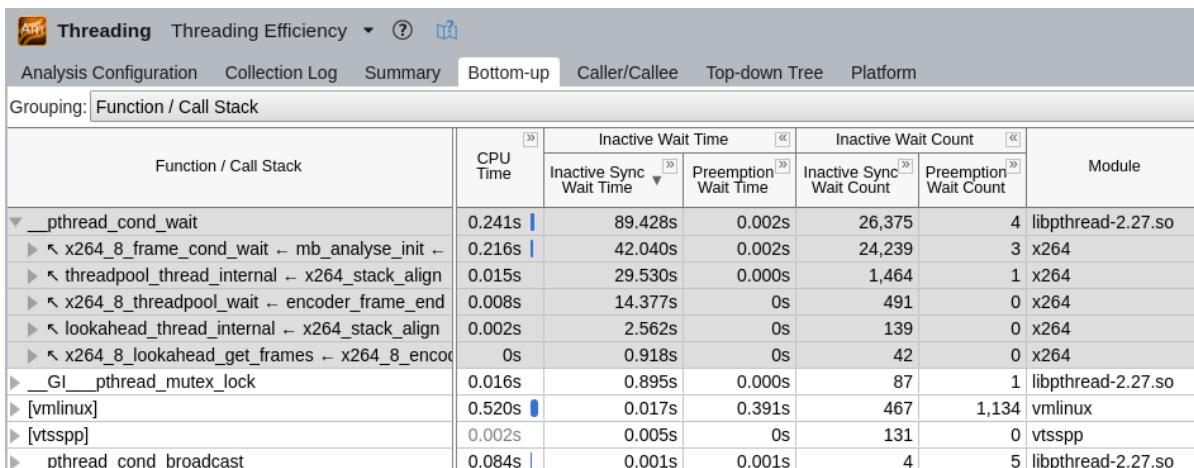


Figure 80: 显示了来自 Phoronix 测试套件中 x264 基准测试的 Intel VTune Profiler 线程分析，显示了最有争议的条件变量的调用堆栈。

接下来，我们可以通过双击分析中相应行来跳转到 `x264_8_frame_cond_wait` 函数的源代码视图，如图 81 所示。接下来，我们可以研究锁的原因以及在这个地方使线程通信更有效的可能方法。<sup>224</sup>

### 11.3.2 平台视图

Intel VTune Profiler 的另一个非常有用的功能是平台视图（见图 82），它允许我们观察程序执行的任何给定时刻每个线程在做什么。这对于理解应用程序的行为并找到潜在的性能增长空间非常有帮助。例如，我们可以看到在从 1 秒到 3 秒的时间间隔内，只有两个线程在持续地利用相应 CPU 核心的约 100%（线程 ID 分别为 7675 和 7678）。在此期间，其他线程的 CPU 利用率是突发性的。

<sup>224</sup> 我不认为这将是一条容易的道路，并且不能保证您会找到使其更好的方法。

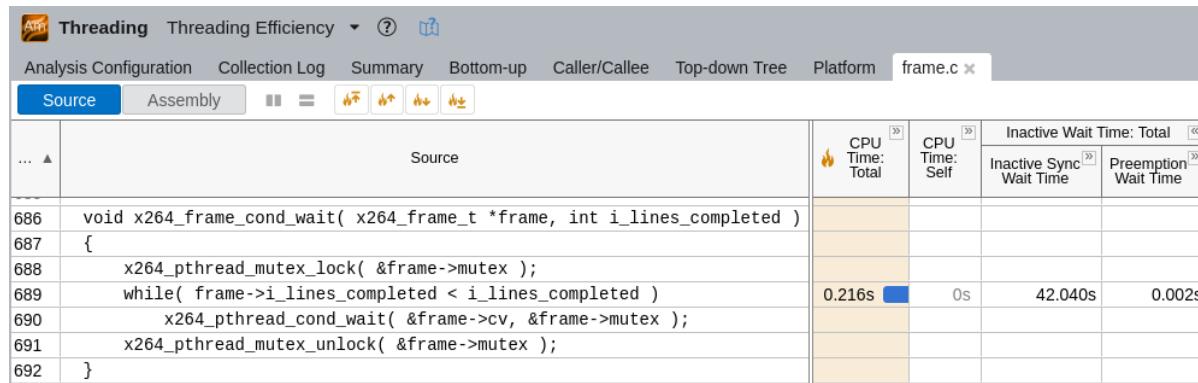


Figure 81: 显示了 Phoronix 测试套件中 x264 基准测试中 `x264_8_frame_cond_wait` 函数的源代码视图。

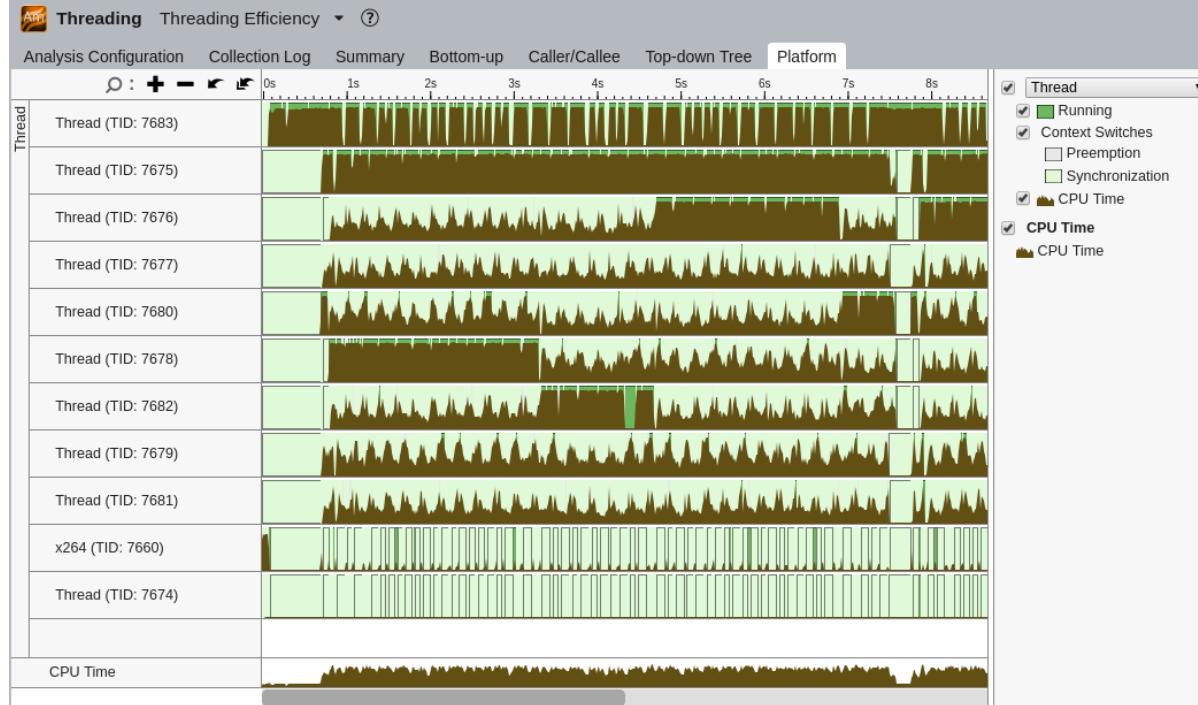


Figure 82: 显示了 Phoronix 测试套件中 x264 基准测试的 Intel VTune Profiler 平台视图。

平台视图还具有缩放和过滤功能。这使我们能够了解指定时间范围内每个线程在执行什么操作。要查看此内容，请在时间轴上选择范围，右键单击并选择“放大”和“按所选内容过滤”。Intel VTune Profiler 将显示在此时间范围内使用的函数或同步对象。

## 11.4 使用 Linux Perf 进行分析

Linux 的 perf 工具可以对应用程序可能产生的所有线程进行性能分析。它有 -s 选项，可以记录每个线程的事件计数。使用此选项，在报告的末尾，perf 列出了所有线程 ID 以及每个线程收集的样本数：

```
$ 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
```

要为特定的软件线程过滤样本，可以使用 --tid 选项：

```
$ 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 的 perf 也自动提供了我们在 Section 11.2 中讨论的一些指标：

```
$ 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
```

### 11.4.1 查找开销大的锁

要使用 Linux 的 perf 找到最有争议的同步对象，需要对调度程序上下文切换进行采样 (`sched:sched_switch`)，这是一个内核事件，因此需要 root 访问权限：

```
$ 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
```

上面的输出显示了哪些线程最频繁地被切换出执行。请注意，我们还收集了调用堆栈 (`--call-graph dwarf`，见 Section 5.5.3)，因为我们需要用它来分析导致昂贵同步事件的路径：

```
$ 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%--0xfffffffffffffff
 |  |
 |  |--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
...

```

上面的列表显示了导致等待条件变量 (`__pthread_cond_wait`) 和后续上下文切换的最频繁路径。这条路径是 `x264_8_macroblock_analyse -> mb_analyse_init -> x264_8_frame_cond_wait`。从这个输出中，我们可以得知 84% 的上下文切换都是由线程在 `x264_8_frame_cond_wait` 中等待条件变量引起的。

## 11.5 使用 Coz 进行分析

在 Section 11.1 中，我们定义了识别影响多线程程序整体性能的代码部分的挑战。由于各种原因，优化多线程程序的一部分并不总是能带来明显效果。Coz: <https://github.com/plasma-umass/coz><sup>225</sup> 是一种新型的性能分析器，解决了这个问题，填补了传统软件性能分析器留下的空白。它使用一种称为“因果性能分析”的新技术，通过在应用程序运行期间虚拟地加速代码段来预测某些优化的整体效果，从而进行实验。它通过插入减慢所有其他同时运行代码的暂停来实现这些“虚拟加速”。[Curtsinger & Berger, 2018]

将 Coz 分析器应用于 Phoronix 测试套件: <https://www.phoronix-test-suite.com/> 中的 C-Ray: <https://openbenchmarking.org/test/pts/c-ray> 基准的示例如图 83 所示。根据图表，如果我们将 c-ray-mt.c 中第 540 行的性能提高 20%，Coz 预计 C-Ray 基准整体应用程序性能将相应提高约 17%。一旦我们在这条线上的改进达到 ~45%，对其应用程序的影响就会根据 Coz 的估计开始趋于平缓。有关此示例的更多详细信息，请参阅 easyperf 博客上的文章: <https://easyperf.net/blog/2020/02/26/coz-vs-sampling-profilers><sup>226</sup>。

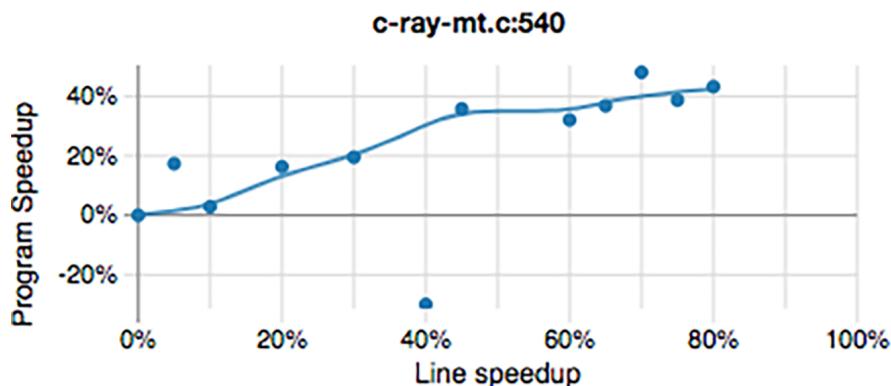


Figure 83: CozProfile 适用于 C-Ray 基准的 Coz 分析文件.

## 11.6 利用 eBPF 和 GAPP 进行分析

Linux 支持各种线程同步原语 - 互斥锁、信号量、条件变量等。内核通过 `futex` 系统调用支持这些线程原语。因此，通过追踪内核中 `futex` 系统调用的执行，同时从涉及的线程中收集有用的元数据，可以更轻松地识别争用瓶颈。

<sup>225</sup> COZ 源代码 - <https://github.com/plasma-umass/coz>。

<sup>226</sup> 博客文章“COZ 与采样性能分析器” - <https://easyperf.net/blog/2020/02/26/coz-vs-sampling-profilers>。

Linux 提供了内核跟踪/分析工具，使之成为可能，其中 eBPF (Extended Berkley Packet Filter<sup>227</sup>) 的功能最为强大。

eBPF 基于内核中运行的沙箱虚拟机，允许在内核内安全高效地执行用户定义的程序。用户定义的程序可以用 C 语言编写，并由 BCC 编译器 (<https://github.com/iovisor/bcc>)<sup>228</sup> 编译成 BPF 字节码，以便加载到内核虚拟机中。这些 BPF 程序可以编写成在某些内核事件执行时启动，并通过各种方式将原始或处理后的数据传回用户空间。

开源社区提供了许多用于通用目的的 eBPF 程序。其中一个工具是通用自动并行分析器 (Generic Automatic Parallel Profiler) (GAPP)，它有助于跟踪多线程争用问题。GAPP 使用 eBPF 通过对已识别的序列化瓶颈进行关键性排序来跟踪多线程应用程序的争用开销，收集被阻塞的线程和导致阻塞的线程的堆栈轨迹。GAPP 最好的方面是它不需要代码更改、昂贵的工具化或重新编译。GAPP 分析器的创建者能够确认已知的瓶颈，并且还揭示了 Parsec 3.0 Benchmark Suite (<https://parsec.cs.princeton.edu/index.htm>)<sup>229</sup> 和一些大型开源项目中以前未报告的新瓶颈。

## 11.7 缓存一致性问题

### 11.7.1 缓存一致性协议

多处理器系统采用缓存一致性协议来确保每个包含独立缓存的独立内核共享使用内存时的数据一致性。如果没有这样的协议，如果 CPU A 和 CPU B 都将内存位置 L 读取到各自的缓存中，然后处理器 B 随后修改其缓存值 L，那么 CPU 将具有相同内存位置 L 的不一致值。缓存一致性协议确保对缓存条目的任何更新都忠实地更新在同一位置的任何其他缓存条目中。

MESI (Modified Exclusive Shared Invalid) 是最著名的缓存一致性协议之一，用于支持现代 CPU 中使用的回写缓存。其缩写表示缓存行可以标记的四种状态（参见图 84）：

- **修改 (Modified)**: 缓存行仅存在于当前缓存中，并且已从其在 RAM 中的值进行修改
- **独占 (Exclusive)**: 缓存行仅存在于当前缓存中，并且与其在 RAM 中的值匹配
- **共享 (Shared)**: 缓存行存在于这里和其他缓存行中，并且与其在 RAM 中的值匹配
- **无效 (Invalid)**: 缓存行未使用（即不包含任何 RAM 位置）

从内存中获取时，每个缓存行都将一个状态编码到其标签中。然后，缓存行状态会从一个状态转换到另一个状态。

<sup>230</sup> 现实中，CPU 供应商通常会实现稍作改进的 MESI 变体。例如，英特尔使用 MESIF: [https://en.wikipedia.org/wiki/MESIF\\_protocol](https://en.wikipedia.org/wiki/MESIF_protocol)，<sup>231</sup> 它添加了转发 (F) 状态，而 AMD 则使用 MOESI: [https://en.wikipedia.org/wiki/MOESI\\_protocol](https://en.wikipedia.org/wiki/MOESI_protocol)，<sup>232</sup> 它添加了拥有 (O) 状态。但这些协议仍然保持了基本 MESI 协议的本质。

正如早期示例所示，缓存一致性问题会导致程序出现顺序不一致的问题。这个问题可以通过使用窥探缓存来监视所有内存事务并相互协作以保持内存一致性来缓解。不幸的是，这也伴随着成本，因为一个处理器的修改会使另一个处理器缓存中的对应缓存行失效。这会导致内存停顿并浪费系统带宽。与只能为应用程序性能设置上限的序列化和锁定问题不同，一致性问题会导致由 USL 在 Section 11.1 中描述的逆行效应。两种广为人知的缓存一致性问题是“真共享”和“假共享”，我们将在下面进一步探讨。

### 11.7.2 真共享

真共享指的是两个不同的处理器访问同一个变量（请参见 Listing 11.7.2）。

代码清单：真正的共享示例。

<sup>227</sup> eBPF 文档 - <https://prototype-kernel.readthedocs.io/en/latest/bpf/>

<sup>228</sup> BCC 编译器 - <https://github.com/iovisor/bcc>

<sup>229</sup> Parsec 3.0 基准测试套件 - <https://parsec.cs.princeton.edu/index.htm>

<sup>230</sup> 读者可以在此处观看和测试动画 MESI 协议：<https://www.scss.tcd.ie/Jeremy.Jones/vivio/caches/MESI.htm>.

<sup>231</sup> MESIF - [https://en.wikipedia.org/wiki/MESIF\\_protocol](https://en.wikipedia.org/wiki/MESIF_protocol)

<sup>232</sup> MOESI - [https://en.wikipedia.org/wiki/MOESI\\_protocol](https://en.wikipedia.org/wiki/MOESI_protocol)

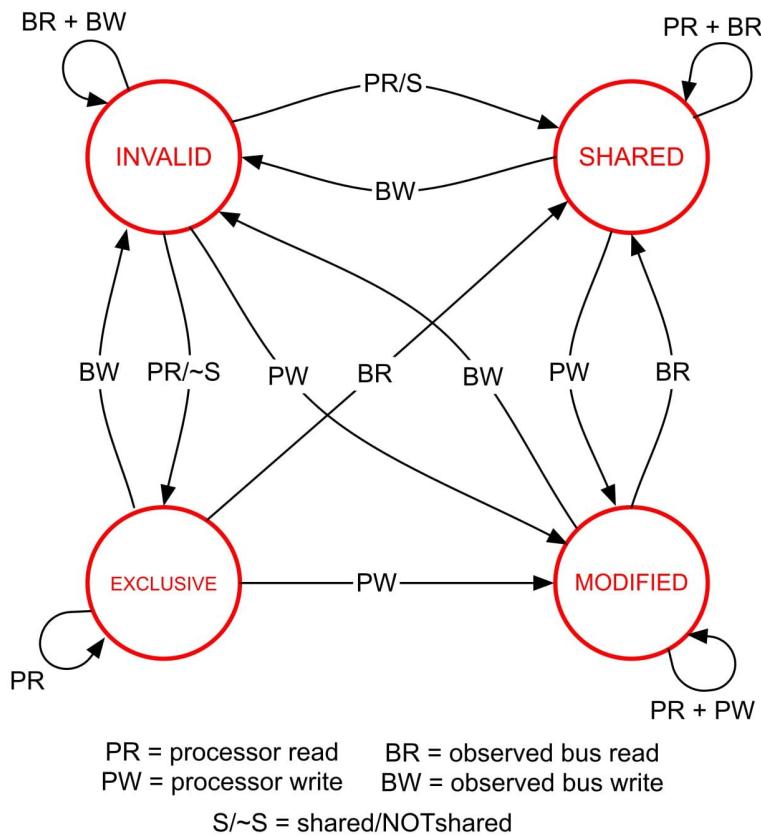


Figure 84: MESI 状态图. © Image by University of Washington via courses.cs.washington.edu.

```

unsigned int sum;
{ // parallel section
    for (int i = 0; i < N; i++)
        sum += a[i]; // sum is shared between all threads
}

```

真实共享意味着存在数据竞争，这很难被检测到。幸运的是，有一些工具可以帮助识别这类问题。Clang 的 Thread sanitizer: [https://clang.llvm.org/docs/ThreadSanitizer.html<sup>233</sup>](https://clang.llvm.org/docs/ThreadSanitizer.html) 和 helgrind: [https://www.valgrind.org/docs/manual/hg-manual.html<sup>234</sup>](https://www.valgrind.org/docs/manual/hg-manual.html) 就是其中的一些工具。为了防止 Listing 11.7.2 中的数据竞争，应该将 `sum` 变量声明为 `std::atomic<unsigned int> sum`。

当发生真实共享时，使用 C++ 原子类型可以帮助解决数据竞争问题。然而，它实际上序列化了对原子变量的访问，这可能会降低性能。解决真实共享问题的另一种方法是使用线程局部存储 (TLS)。TLS 是一种方法，允许给定多线程进程中的每个线程分配内存来存储线程特定的数据。通过这样做，线程修改自己的本地副本，而不是争用全局可用的内存位置。可以使用 TLS 类说明符 (`thread_local unsigned int sum`, 自 C++11 起) 声明 `sum` 来修复 Listing 11.7.2 中的示例。然后，主线程应该合并每个工作线程所有本地副本的结果。

## 11.8 假共享

假共享<sup>235</sup> 发生在两个不同的处理器修改位于同一缓存行上的不同变量时（参见 Listing 235）。图 85 展示了假共享问题。

代码清单: 假共享示例。

```

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];
}

```

假共享是多线程应用程序性能问题的常见来源。因此，现代分析工具内置了检测此类案例的支持。TMA 将经历真/假共享的应用程序描述为内存绑定。通常，在这种情况下，您会看到争用访问: [https://software.intel.com/en-us/vtune-help-contested-accesses<sup>236</sup>](https://software.intel.com/en-us/vtune-help-contested-accesses) 指标的高值。

<sup>233</sup> Clang 的线程消毒工具:<https://clang.llvm.org/docs/ThreadSanitizer.html>。

<sup>234</sup> Helgrind, 一个线程错误检测工具:<https://www.valgrind.org/docs/manual/hg-manual.html>。

<sup>235</sup> 值得注意的是，错误共享不仅在 C/C++/Ada 等低级语言中可以观察到，在 Java/C# 等高级语言中也可以观察到。

<sup>236</sup> 争用访问 - <https://software.intel.com/en-us/vtune-help-contested-accesses>.

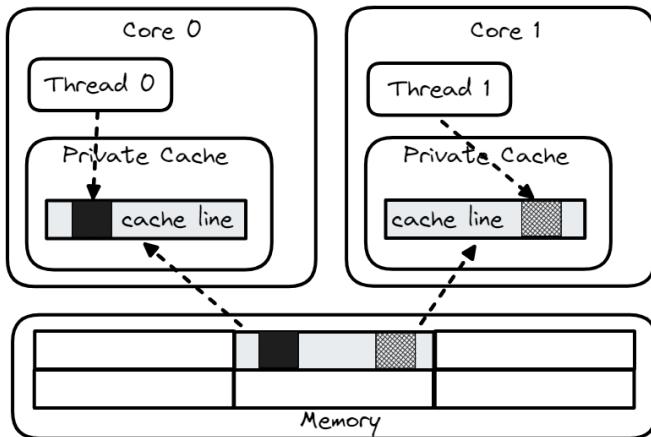


Figure 85: False 共享: 两个线程访问同一个缓存行。© Image by Intel Developer Zone via software.intel.com.

使用 Intel VTune Profiler 时，用户需要两种类型的分析来查找和消除假共享问题。首先，运行 微架构探索: [https://software.intel.com/en-us/vtune-help-general-exploration-analysis<sup>237</sup>](https://software.intel.com/en-us/vtune-help-general-exploration-analysis) 分析，该分析实施 TMA 方法来检测应用程序中是否存在假共享。正如之前提到的，争用访问指标的高值促使我们深入挖掘并运行启用了“分析动态内存对象”选项的 内存访问: <https://software.intel.com/en-us/vtune-help-memory-access-analysis> 分析。此分析有助于找出导致争用问题的对数据结构的访问。通常，这些内存访问具有高延迟，分析会揭示这一点。有关使用 Intel VTune Profiler 修复假共享问题的示例，请参见 英特尔开发者社区: [https://software.intel.com/en-us/vtune-cookbook-false-sharing<sup>238</sup>](https://software.intel.com/en-us/vtune-cookbook-false-sharing)。

Linux perf 也支持查找假共享。与 Intel VTune Profiler 一样，首先运行 TMA（请参见 Section 5.10.1）以找出程序是否经历假/真共享问题。如果是这种情况，请使用 perf c2c 工具检测具有高缓存一致性成本的内存访问。perf c2c 匹配不同线程的存储/加载地址，并查看是否命中了修改后的缓存行。读者可以在专门的博客文章: [https://joemario.github.io/blog/2016/09/01/c2c-blog/<sup>239</sup>](https://joemario.github.io/blog/2016/09/01/c2c-blog/) 中找到该过程及其如何使用工具的详细解释。

可以通过对齐/填充内存对象来消除假共享。Section 11.7.2 中的示例可以通过确保 sumA 和 sumB 不共享同一缓存行来修复（请参阅 Section 7.1.4 中的详细信息）。

从一般的性能角度来看，最重要的考虑因素是可能状态转换的成本。在所有缓存状态中，唯一不涉及昂贵的跨缓存子系统通信和 CPU 读/写操作期间的数据传输的是修改 (M) 和独占 (E) 状态。因此，缓存行保持“M”或“E”状态的时间越长（即跨缓存的数据共享越少），多线程应用程序产生的一致性成本就越低。有关如何利用此属性的示例，请参见 Nitsan Wakart 的博客文章“深入了解缓存一致性: [http://psy-lob-saw.blogspot.com/2013/09/diving-deeper-into-cache-coherency.html<sup>240</sup>](http://psy-lob-saw.blogspot.com/2013/09/diving-deeper-into-cache-coherency.html)”。

## 问题和练习

1. 完成 perf-ninja:::false\_sharing 实验练习。
2. 运行你日常使用的应用程序。它是否多线程？如果不是，请选用一些多线程基准测试。计算并行效率指标，并运行扩展性研究。观察时间线图，是否存在调度问题？识别热点锁以及导致这些锁的代码路径。你能改进锁定吗？检查应用程序性能是否受到真/假共享的影响。
3. 奖励问题：多线程应用与多进程应用相比有哪些优势？

<sup>237</sup> Vtune 一般探索分析 - <https://software.intel.com/en-us/vtune-help-general-exploration-analysis>.

<sup>238</sup> Vtune 食谱：假共享 - <https://software.intel.com/en-us/vtune-cookbook-false-sharing>.

<sup>239</sup> 关于 perf c2c 的文章 - <https://joemario.github.io/blog/2016/09/01/c2c-blog/>.

<sup>240</sup> 博客文章“深入了解缓存一致性”- <http://psy-lob-saw.blogspot.com/2013/09/diving-deeper-into-cache-coherency.html>

## 章节总结

- 没有利用现代多核 CPU 的应用程序正在落后于竞争对手。为了应用程序未来的成功，让软件能够良好地随着 CPU 内核数量的增长而扩展是非常重要的。
- 当处理单线程应用程序时，优化程序的一部分通常会对性能产生积极影响。然而，对于多线程应用程序来说，情况不一定如此。这种效应被称为阿姆达尔定律，它规定并行程序的加速受其串行部分的限制。
- 如通用可扩展性定律所解释，线程通信可能导致负面加速。这给多线程程序的调优带来了额外的挑战。优化多线程应用程序的性能还涉及检测和减轻争用和一致性的影响。
- Intel VTune Profiler 是分析多线程应用程序的常用工具。但是近年来，其他工具也出现了，它们拥有独特的功能集，例如 Coz 和 GAPP。

## 11.9 软件和硬件性能的当前和未来趋势

### 软件性能

- 人工智能和机器学习的兴起将继续推动软件性能的进步。
- 软件开发人员将需要更多地了解并利用硬件加速器来提高性能。
- 软件架构将变得更加复杂，以充分利用多核处理器和异构计算平台。

### 硬件性能

- 摩尔定律的放缓将迫使硬件制造商寻找新的方法来提高性能。
- 3D 堆叠和 chiplet 设计等新技术将使芯片能够在更小的空间内容纳更多的晶体管。
- 光子和量子计算等新兴技术可能会在未来几年内带来重大突破。

### 软件和硬件性能的协同发展

- 软件和硬件开发人员将需要更加紧密地合作，以充分利用未来的计算平台。
- 编译器和操作系统等软件工具将需要不断改进，以更好地利用硬件功能。
- 性能分析和调优将变得更加重要，以确保软件能够充分发挥硬件的潜力。

以下是一些具体的趋势示例：

- 人工智能和机器学习

人工智能和机器学习 (AI/ML) 应用程序对性能有着极高的要求。为了满足这些要求，软件开发人员将需要更多地了解并利用硬件加速器。例如，GPU 和 TPU 能够比传统的 CPU 或 CPU 更快地执行 AI/ML 任务。

- 多核处理器和异构计算平台

多核处理器和异构计算平台正在变得越来越普遍。为了充分利用这些平台，软件架构将变得更加复杂。例如，软件开发人员将需要使用线程和并行编程技术来充分利用多个内核和加速器。

- 3D 堆叠和 chiplet 设计

3D 堆叠和 chiplet 设计等新技术使芯片能够在更小的空间内容纳更多的晶体管。这将导致更高的性能和更低的功耗。

- 光子和量子计算

光子和量子计算等新兴技术可能会在未来几年内带来重大突破。这些技术有可能提供比传统计算平台高得多的性能。

### 结论

软件和硬件性能的趋势将继续快速发展。软件开发人员和硬件制造商将需要共同努力，以充分利用未来的计算平台。

### 后记

感谢您阅读完整本书。希望您喜欢并从中有所收获。如果本书能帮助您解决实际问题，我会更加高兴。在这种情况下，我将把它视为成功，并证明我的努力没有白费。在您继续努力之前，让我简要强调本书的要点并为您提供最终建议：

- 硬件性能不像几十年前那样快速增长了。性能调优变得比过去 40 年来更加重要。它将是未来获得性能提升的主要驱动力之一。
- 软件默认情况下没有最佳性能。存在某些限制阻止应用程序发挥其全部性能潜力。硬件和软件组件都存在这样的限制。
- Donald Knuth 有句名言：“过早优化是所有问题的根源”。但反过来也经常是真的。推迟性能工程工作可能为时已晚，并可能像过早优化一样造成同样的危害。在设计未来产品时，不要忽视性能方面。

- 现代 CPU 的性能不是确定性的，取决于许多因素。有意义的性能分析应该考虑噪音并使用统计方法分析性能测量。
- 了解 CPU 微架构可能有助于理解您进行实验的结果。但是，在对代码进行特定更改时，不要过分依赖这些知识。您的心智模型永远不可能像 CPU 内部实际设计那样准确。预测特定代码片段的性能几乎是不可能的。
- 一定要测量！
- 性能调优很难，因为它没有预定的步骤需要遵循，没有算法。工程师需要从不同的角度解决问题。了解可用的性能分析方法和工具（硬件和软件）。我强烈建议拥抱 Roofline 模型和 TMA 方法，如果它们在您的平台上可用。它将帮助您将您的工作引导到正确的方向。此外，要知道何时可以利用其他硬件性能监控功能，例如 LBR、PEBS 和 PT。
- 了解应用程序性能的限制因素以及可能的修复方法。第二部分涵盖了一些针对每种类型 CPU 性能瓶颈的基本优化：前端受限、后端受限、回退、错误推测。使用第 8-11 章查看当您的应用程序属于上述四种类别之一时可用的选项。
- 如果修改带来的好处可以忽略不计，您应该保持代码最简单和最干净的形式。
- 有时，在系统上提高性能的修改会在另一个系统上减慢执行速度。请确保在您关心的所有平台上测试您的更改。

我希望这本书能帮助您更好地理解应用程序的性能和整体 CPU 性能。当然，它无法涵盖您在进行性能优化时可能遇到的所有情况。我的目标是为您提供一个起点，并向您展示在现代 CPU 上进行性能分析和调整的潜在选项和策略。

如果您喜欢阅读本书，请务必将其传递给您的朋友和同事。如果您能在社交媒体平台上宣传本书，我将不胜感激。

我非常欢迎您通过电子邮件 [dendibakh@gmail.com](mailto:dendibakh@gmail.com) 提供反馈。请告诉我您对本书的想法、评论和建议。我将在我的博客 <https://easyperf.net/contact/> 上发布所有更新和未来有关本书的信息。

祝您性能调优愉快！

# 术语表

|                                                      |                                             |
|------------------------------------------------------|---------------------------------------------|
| AOS Array Of Structures                              | LLC Last Level Cache                        |
| BB Basic Block                                       | LSD Loop Stream Detector                    |
| BIOS Basic Input Output System                       | MSR Model Specific Register                 |
| CI/CD Contiguous Integration/ Contiguous Development | MS-ROM Microcode Sequencer Read-Only Memory |
| CPI Clocks Per Instruction                           | NUMA Non-Uniform Memory Access              |
| CPU Central Processing Unit                          | OS Operating System                         |
| DSB Decoded Stream Buffer                            | PEBS Processor Event-Based Sampling         |
| DRAM Dynamic Random-Access Memory                    | PGO Profile Guided Optimizations            |
| DTLB Data Translation Lookaside Buffer               | PMC Performance Monitoring Counter          |
| EBS Event-Based Sampling                             | PMI Performance Monitoring Interrupt        |
| FLOPS FLoating-point Operations Per Second           | PMU Performance Monitoring Unit             |
| FPGA Field-Programmable Gate Array                   | PT Processor Traces                         |
| GPU Graphics processing unit                         | RAT Register Alias Table                    |
| HFT High-Frequency Trading                           | ROB ReOrder Buffer                          |
| HPC High Performance Computing                       | SIMD Single Instruction Multiple Data       |
| HW Hardware                                          | SMT Simultaneous MultiThreading             |
| I/O Input/Output                                     | SOA Structure Of Arrays                     |
| IDE Integrated Development Environment               | SW Software                                 |
| ILP Instruction-Level Parallelism                    | TLB Translation Lookaside Buffer            |
| IPC Instructions Per Clock cycle                     | TMA Top-down Microarchitecture Analysis     |
| IPO Inter-Procedural Optimizations                   | TSC Time Stamp Counter                      |
| ITLB Instruction Translation Lookaside Buffer        | $\mu$ op MicroOperation                     |
| LBR Last Branch Record                               |                                             |

# 主要 CPU 微架构列表

在下面的表格中，我们展示了来自 Intel、AMD 和基于 ARM 的供应商的最新 ISA 和微架构。当然，并不是所有的设计都列在这里。我们只包括了那些在本书中引用的或者代表了平台演变中的重大转变的设计。

Table 12: 最近的英特尔 Core 微架构列表。

| 名称           | 三字母缩写 | 发布年份 | 支持的 ISA 客户端/服务<br>器芯片 |
|--------------|-------|------|-----------------------|
| Nehalem      | NHM   | 2008 | SSE4.2                |
| Sandy Bridge | SNB   | 2011 | AVX                   |
| Haswell      | HSW   | 2013 | AVX2                  |
| Skylake      | SKL   | 2015 | AVX2 / AVX512         |
| Sunny Cove   | SNC   | 2019 | AVX512                |
| Golden Cove  | GLC   | 2021 | AVX2 / AVX512         |
| Redwood Cove | RWC   | 2023 | AVX2 / AVX512         |

Table 13: 最近的 AMD 微架构列表。

| 名称           | 发布年份 | 支持的 ISA |
|--------------|------|---------|
| Streamroller | 2014 | AVX     |
| Excavator    | 2015 | AVX2    |
| Zen          | 2017 | AVX2    |
| Zen2         | 2019 | AVX2    |
| Zen3         | 2020 | AVX2    |
| Zen4         | 2022 | AVX512  |

Table 14: 最近的 ARM ISA 列表，以及它们自己和第三方的实现。

| ISA              | ISA 发布年份 | ARM 微架构 (最新)                        | 第三方微架构                                        |
|------------------|----------|-------------------------------------|-----------------------------------------------|
| ARMv8-A          | 2011     | Cortex-A73                          | Apple A7-A10; Qualcomm Kryo; Samsung M1/M2/M3 |
| ARMv8.2-A        | 2016     | Neoverse N1; Cortex-X1              | Apple A11; Samsung M4; Ampere Altra           |
| ARMv8.4-A        | 2017     | Neoverse V1                         | AWS Graviton3; Apple A13, M1                  |
| ARMv9.0-A (64 位) | 2018     | Neoverse N2; Neoverse V2; Cortex X3 | Microsoft Cobalt 100; NVIDIA Grace            |
| ARMv8.6-A (64 位) | 2019     | —                                   | Apple A15, A16, M2, M3                        |
| ARMv9.2-A        | 2020     | Cortex X4                           | —                                             |

## References

- [Advanced Micro Devices, 2021] Advanced Micro Devices (2021). Processor Programming Reference (PPR) for AMD family 19h model 01h (55898). B1 Rev 0.50. [https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/programmer-references/55898\\_B1\\_pub\\_0\\_50.zip](https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/programmer-references/55898_B1_pub_0_50.zip)
- [Advanced Micro Devices, 2022] Advanced Micro Devices (2022). AMD64 technology platform quality of service extensions. Pub. 56375, rev 1.01. [https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/other/56375\\_1\\_0\\_3\\_PUB.pdf](https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/other/56375_1_0_3_PUB.pdf)
- [Akinshin, 2019] Akinshin, A. (2019). Pro .NET Benchmarking (1 ed.). Apress. <https://doi.org/10.1007/978-1-4842-4941-3>
- [Alam et al., 2019] Alam, M., Gottschlich, J., Tatbul, N., Turek, J. S., Mattson, T., & Muzahid, A. (2019). A zero-positive learning approach for diagnosing software performance regressions. Advances in Neural Information Processing Systems 32, 11627 – 11639. Curran Associates, Inc. [http://papers.nips.cc/paper/9337-azero-positive-learning-approach-for-diagnosing-software-performance-regressions.pdf](http://papers.nips.cc/paper/9337-a-zero-positive-learning-approach-for-diagnosing-software-performance-regressions.pdf)
- [AMD, 2023] AMD (2023). AMD64 Architecture Programmer's Manual. Advanced Micro Devices, Inc. <https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf>
- [AMD, 2024] AMD (2024). AMD uProf User Guide, Revision 4.2. Advanced Micro Devices, Inc. <https://www.amd.com/content/dam/amd/en/documents/developer/version-4-2-documents/uprof/uprof-user-guide-v4.2.pdf>
- [Arm, 2022a] Arm (2022a). Arm Architecture Reference Manual Supplement Armv9. Arm Limited. <https://documentation-service.arm.com/static/632dbdace68c6809a6b41710?token=>
- [Arm, 2022b] Arm (2022b). Arm Neoverse™ V1 PMU Guide, Revision: r1p2. Arm Limited. <https://developer.arm.com/documentation/PJDOC-1063724031-605393/2-0/?lang=en>
- [Arm, 2023a] Arm (2023a). Arm Neoverse V1 Core: Performance Analysis Methodology. Arm Limited. <https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/neoverse-v1-core-performance-analysis.pdf>
- [Arm, 2023b] Arm (2023b). Arm Statistical Profiling Extension: Performance Analysis Methodology. Arm Limited. <https://developer.arm.com/documentation/109429/latest/>
- [Chen et al., 2016] Chen, D., Li, D. X., & Moseley, T. (2016). Autofdo: Automatic feedback-directed optimization for warehouse-scale applications. CGO 2016 Proceedings of the 2016 International Symposium on Code Generation and Optimization, 12 – 23. <https://ieeexplore.ieee.org/document/7559528>
- [Cooper & Torczon, 2012] Cooper, K. & Torczon, L. (2012). Engineering a Compiler. Morgan Kaufmann. Morgan Kaufmann. <https://books.google.co.in/books?id=CGTOIAEACAAJ>
- [Curtsinger & Berger, 2013] Curtsinger, C. & Berger, E. D. (2013). Stabilizer: Statistically sound performance evaluation. Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, 219–228. <https://doi.org/10.1145/2451116.2451141>
- [Curtsinger & Berger, 2018] Curtsinger, C. & Berger, E. D. (2018). Coz: Finding code that counts with causal profiling. Commun. ACM, 61(6), 91–99. <https://doi.org/10.1145/3205911>
- [Daly et al., 2020] Daly, D., Brown, W., Ingo, H., O'Leary, J., & Bradford, D. (2020). The use of change point detection to identify software performance regressions in a continuous integration system. Proceedings of the ACM/SPEC International Conference on Performance Engineering, ICPE '20, 67–75. <https://doi.org/10.1145/3358960.3375791>

- [domo.com, 2017] domo.com (2017). Data Never Sleeps 5.0. Domo, Inc. [https://www.domo.com/learn/data-never-sleeps-5?aid=ogsm072517\\_1&sf100871281=1](https://www.domo.com/learn/data-never-sleeps-5?aid=ogsm072517_1&sf100871281=1)
- [Du et al., 2010] Du, J., Sehrawat, N., & Zwaenepoel, W. (2010). Performance profiling in a virtualized environment. Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, 2. [https://www.usenix.org/legacy/event/hotcloud10/tech/full\\_papers/Du.pdf](https://www.usenix.org/legacy/event/hotcloud10/tech/full_papers/Du.pdf)
- [Fog, 2004] Fog, A. (2004). Optimizing software in c++: An optimization guide for windows, linux and mac platforms. [https://www.agner.org/optimize/optimizing\\_cpp.pdf](https://www.agner.org/optimize/optimizing_cpp.pdf)
- [Fog, 2012] Fog, A. (2012). The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers. Copenhagen University College of Engineering. <https://www.agner.org/optimize/microarchitecture.pdf>
- [Gregg, 2013] Gregg, B. (2013). Systems Performance: Enterprise and the Cloud (1st ed.). Prentice Hall Press.
- [Grosser et al., 2012] Grosser, T., Größlinger, A., & Lengauer, C. (2012). Polly - performing polyhedral optimizations on a low-level intermediate representation. Parallel Process. Lett., 22.
- [Hennessy, 2018] Hennessy, J. L. (2018). The future of computing. <https://youtu.be/Azt8Nc-mtKM?t=329>
- [Hennessy & Patterson, 2017] Hennessy, J. L. & Patterson, D. A. (2017). Computer Architecture, Sixth Edition: A Quantitative Approach (6th ed.). Morgan Kaufmann Publishers Inc.
- [Ingo & Daly, 2020] Ingo, H. & Daly, D. (2020). Automated system performance testing at mongodb. Proceedings of the Workshop on Testing Database Systems, DBTest '20. <https://doi.org/10.1145/3395032.3395323>
- [Intel, 2023a] Intel (2023a). CPU Metrics Reference. Intel® Corporation. <https://software.intel.com/en-us/vtune-help-cpu-metrics-reference>
- [Intel, 2023b] Intel (2023b). Intel® 64 and IA-32 Architectures Optimization Reference Manual. Intel® Corporation. <https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html>
- [Jimenez & Lin, 2001] Jimenez, D. & Lin, C. (2001). Dynamic branch prediction with perceptrons. Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture, 197 – 206. <https://doi.org/10.1109/HPCA.2001.903263>
- [Jin et al., 2012] Jin, G., Song, L., Shi, X., Scherpelz, J., & Lu, S. (2012). Understanding and detecting real-world performance bugs. Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, 77–88. <https://doi.org/10.1145/2254064.2254075>
- [Kanev et al., 2015] Kanev, S., Darago, J. P., Hazelwood, K., Ranganathan, P., Moseley, T., Wei, G.-Y., & Brooks, D. (2015). Profiling a warehouse-scale computer. SIGARCH Comput. Archit. News, 43(3S), 158–169. <https://doi.org/10.1145/2872887.2750392>
- [Kapoor, 2009] Kapoor, R. (2009). Avoiding the cost of branch misprediction. <https://software.intel.com/en-us/articles/avoiding-the-cost-of-branch-misprediction>
- [Khuong & Morin, 2015] Khuong, P.-V. & Morin, P. (2015). Array layouts for comparison-based searching. <https://arxiv.org/ftp/arxiv/papers/1509/1509.05053.pdf>
- [Leiserson et al., 2020] Leiserson, C. E., Thompson, N. C., Emer, J. S., Kuszmaul, B. C., Lampson, B. W., Sanchez, D., & Schardl, T. B. (2020). There's plenty of room at the top: What will drive computer performance after moore's law? Science, 368(6495). <https://doi.org/10.1126/science.aam9744>

- [Liu et al., 2019] Liu, M., Sun, X., Varshney, M., & Xu, Y. (2019). Large-scale online experimentation with quantile metrics. <https://arxiv.org/abs/1903.08762>
- [Luo et al., 2015] Luo, T., Wang, X., Hu, J., Luo, Y., & Wang, Z. (2015). Improving tlb performance by increasing hugepage ratio. 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 1139 – 1142. <https://doi.org/10.1109/CCGrid.2015.36>
- [Matteson & James, 2014] Matteson, D. S. & James, N. A. (2014). A nonparametric approach for multiple change point analysis of multivariate data. *Journal of the American Statistical Association*, 109(505), 334 – 345. <https://doi.org/10.1080/01621459.2013.849605>
- [Mittal, 2016] Mittal, S. (2016). A survey of techniques for cache locking. *ACM Transactions on Design Automation of Electronic Systems*, 21. <https://doi.org/10.1145/2858792>
- [Muła & Lemire, 2019] Muła, W. & Lemire, D. (2019). Base64 encoding and decoding at almost the speed of a memory copy. *Software: Practice and Experience*, 50(2), 89–97. <https://doi.org/10.1002/spe.2777>
- [Mytkowicz et al., 2009] Mytkowicz, T., Diwan, A., Hauswirth, M., & Sweeney, P. F. (2009). Producing wrong data without doing anything obviously wrong! Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIV, 265–276. <https://doi.org/10.1145/1508244.1508275>
- [Navarro-Torres et al., 2023] Navarro-Torres, A., Alatruey-Benedé, J., Ibáñez, P., & Viñals-Yúfera, V. (2023). Balancer: bandwidth allocation and cache partitioning for multicore processors. *The Journal of Supercomputing*, 79(9), 10252 – 10276. <https://doi.org/10.1007/s11227-023-05070-0>
- [Navarro-Torres et al., 2019] Navarro-Torres, A. et al. (2019). Memory hierarchy characterization of SPEC CPU2006 and SPEC CPU2017 on the Intel Xeon Skylake-SP. *PLOS ONE*, 1 – 24. <https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0220135>
- [Nowak & Bitzes, 2014] Nowak, A. & Bitzes, G. (2014). The overhead of profiling using pmu hardware counters. <https://zenodo.org/record/10800/files/TheOverheadOfProfilingUsingPMUhardwareCounters.pdf>
- [Ottoni & Maher, 2017] Ottoni, G. & Maher, B. (2017). Optimizing function placement for large-scale data-center applications. Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO ’17, 233–244. <https://ieeexplore.ieee.org/document/7863743>
- [Panchenko et al., 2018] Panchenko, M., Auler, R., Nell, B., & Ottoni, G. (2018). BOLT: A practical binary optimizer for data centers and beyond. *CoRR*, abs/1807.06735. <http://arxiv.org/abs/1807.06735>
- [Paoloni, 2010] Paoloni, G. (2010). How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures. Intel® Corporation. <https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf>
- [Ren et al., 2010] Ren, G., Tune, E., Moseley, T., Shi, Y., Rus, S., & Hundt, R. (2010). Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, 65 – 79. <http://www.computer.org/portal/web/csdl/doi/10.1109/MM.2010.68>
- [Sasongko et al., 2023] Sasongko, M. A., Chabbi, M., Kelly, P. H. J., & Unat, D. (2023). Precise event sampling on amd versus intel: Quantitative and qualitative comparison. *IEEE Transactions on Parallel and Distributed Systems*, 34(5), 1594 – 1608. <https://doi.org/10.1109/TPDS.2023.3257105>
- [Seznec & Michaud, 2006] Seznec, A. & Michaud, P. (2006). A case for (partially) tagged geometric history length branch prediction. *J. Instr. Level Parallelism*, 8. <https://inria.hal.science/hal-03408381/document>

- [Sharma, 2016] Sharma, S. D. (2016). Hardware-assisted instruction profiling and latency detection. *The Journal of Engineering*, 2016, 367 – 376(9). <https://digital-library.theiet.org/content/journals/10.1049/joe.2016.0127>
- [statista.com, 2018] statista.com (2018). Volume of data/information created worldwide from 2010 to 2025. Statista, Inc. <https://www.statista.com/statistics/871513/worldwide-data-created/>
- [Suresh Srinivas, 2019] Suresh Srinivas, e. a. (2019). Runtime performance optimization blueprint: Intel® architecture optimization with large code pages. <https://www.intel.com/content/www/us/en/develop/articles/runtime-performance-optimization-blueprint-intel-architecture-optimization-with-large-code.html>
- [Yasin, 2014] Yasin, A. (2014). A top-down method for performance analysis and counters architecture. 35 – 44. <https://doi.org/10.1109/ISPASS.2014.6844459>

## 附录 A. 减少测量噪声 (Reducing Measurement Noise)

以下是一些示例功能，它们可能导致性能测量的不确定性增加。有关完整讨论，请参见 Section 2.1。

### 动态频率缩放 (Dynamic Frequency Scaling)

动态频率缩放 (DFS): [https://en.wikipedia.org/wiki/Dynamic\\_frequency\\_scaling<sup>241</sup>](https://en.wikipedia.org/wiki/Dynamic_frequency_scaling) 是一种通过在运行要求苛刻任务时自动提高 CPU 运行频率来提升系统性能的技术。例如，英特尔 CPU 具有名为 睿频加速: [https://en.wikipedia.org/wiki/Intel\\_Turbo\\_Boost<sup>242</sup>](https://en.wikipedia.org/wiki/Intel_Turbo_Boost) 的 DFS 实现功能，AMD CPU 则采用 Turbo Core: [https://en.wikipedia.org/wiki/AMD\\_Turbo\\_Core<sup>243</sup>](https://en.wikipedia.org/wiki/AMD_Turbo_Core) 功能。

以下示例展示了睿频加速对在 Intel® Core™ i5-8259U 上运行单线程工作负载的影响：

```
# 睿频加速启用
$ cat /sys/devices/system/cpu/intel_pstate/no_turbo
0

$ perf stat -e task-clock,cycles -- ./a.exe
    11984.691958  task-clock (msec) #      1.000 CPUs utilized
    32,427,294,227  cycles          #      2.706 GHz
                                11.989164338 seconds time elapsed

# 睿频加速禁用
$ echo 1 | sudo tee /sys/devices/system/cpu/intel_pstate/no_turbo
1

$ perf stat -e task-clock,cycles -- ./a.exe
    13055.200832  task-clock (msec) #      0.993 CPUs utilized
    29,946,969,255  cycles          #      2.294 GHz
                                13.142983989 seconds time elapsed
```

当睿频加速启用时，平均频率明显更高。

DFS 可以永久地在 BIOS 中禁用。<sup>244</sup> 要在 Linux 系统上以编程方式禁用 DFS 功能，您需要 root 权限。下面是如何实现这一点：

```
# 英特尔
echo 1 > /sys/devices/system/cpu/intel_pstate/no_turbo
# AMD
echo 0 > /sys/devices/system/cpu/cpufreq/boost
```

### 同时多线程 (Simultaneous Multithreading)

现代 CPU 内核通常采用 同时多线程 (SMT): [https://en.wikipedia.org/wiki/Simultaneous\\_multithreading<sup>245</sup>](https://en.wikipedia.org/wiki/Simultaneous_multithreading) 方式制造。这意味着在一个物理内核中，您可以同时执行两个线程。通常，体系结构状态: <https://en.wikipedia.org/wiki/Architecture>

<sup>241</sup> 动态频率缩放 - [https://en.wikipedia.org/wiki/Dynamic\\_frequency\\_scaling](https://en.wikipedia.org/wiki/Dynamic_frequency_scaling).

<sup>242</sup> 英特尔睿频加速 - [https://en.wikipedia.org/wiki/Intel\\_Turbo\\_Boost](https://en.wikipedia.org/wiki/Intel_Turbo_Boost).

<sup>243</sup> AMD Turbo Core - [https://en.wikipedia.org/wiki/AMD\\_Turbo\\_Core](https://en.wikipedia.org/wiki/AMD_Turbo_Core).

<sup>244</sup> 英特尔睿频加速常见问题解答 - <https://www.intel.com/content/www/us/en/support/articles/000007359/processors/intel-core-processors.html>.

<sup>245</sup> SMT - [https://en.wikipedia.org/wiki/Simultaneous\\_multithreading](https://en.wikipedia.org/wiki/Simultaneous_multithreading).

`ectural_state246` 会被复制，但执行资源（ALU、缓存等）不会。这意味着如果我们在同一个内核上以“同时”方式运行两个独立的进程（在不同的线程中），它们会相互争夺资源，例如缓存空间。

SMT 可以永久地在 BIOS 中禁用。<sup>247</sup> 要在 Linux 系统上以编程方式禁用 SMT，您需要 root 权限。下面是如何在每个内核中关闭一个兄弟线程：

```
echo 0 > /sys/devices/system/cpu/cpuX/online
```

可以在以下文件中找到 CPU 线程的兄弟对：

```
/sys/devices/system/cpu/cpuN/topology/thread_siblings_list
```

例如，在具有 4 个内核和 8 个线程的 Intel® Core™ i5-8259U 上：

```
# 所有 8 个硬件线程均已启用:
$ lscpu
...
CPU(s):          8
On-line CPU(s) list: 0-7
...
$ cat /sys/devices/system/cpu/cpu0/topology/thread_siblings_list
0,4
$ cat /sys/devices/system/cpu/cpu1/topology/thread_siblings_list
1,5
$ cat /sys/devices/system/cpu/cpu2/topology/thread_siblings_list
2,6
$ cat /sys/devices/system/cpu/cpu3/topology/thread_siblings_list
3,7

# 在内核 0 上禁用 SMT
$ echo 0 | sudo tee /sys/devices/system/cpu/cpu4/online
0
$ lscpu
CPU(s):          8
On-line CPU(s) list: 0-3,5-7
Off-line CPU(s) list: 4
...
$ cat /sys/devices/system/cpu/cpu0/topology/thread_siblings_list
0
```

## 缩放调速器 (Scaling Governor)

Linux 内核可以控制 CPU 频率用于不同的目的。其中一个目的是节能，在这种情况下，Linux 内核内部的调速器<sup>248</sup>可以决定降低 CPU 运行频率。对于性能测量，建议将调速器策略设置为“性能”，以避免低于标称时钟频率。下面是如何将其设置为所有内核：

<sup>246</sup> 体系结构状态 - [https://en.wikipedia.org/wiki/Architectural\\_state](https://en.wikipedia.org/wiki/Architectural_state).

<sup>247</sup> “如何禁用超线程” - <https://www.pcmag.com/article/314585/how-to-disable-hyperthreading>.

<sup>248</sup> Linux CPU 频率调节器文档： <https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt>.

```
for i in /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor
do
    echo performance > $i
done
```

## CPU 亲和性 (CPU Affinity)

处理器亲和性: [https://en.wikipedia.org/wiki/Processor\\_affinity<sup>249</sup>](https://en.wikipedia.org/wiki/Processor_affinity) 允许将进程绑定到特定 CPU 内核。在 Linux 中，可以使用 taskset: [https://linux.die.net/man/1/taskset<sup>250</sup>](https://linux.die.net/man/1/taskset) 工具实现这一点。这里

```
# 无亲和性
$ perf stat -e context-switches,cpu-migrations -r 10 -- a.exe
      151      context-switches
      10       cpu-migrations

# 进程绑定到 CPU0
$ perf stat -e context-switches,cpu-migrations -r 10 -- taskset -c 0 a.exe
      102      context-switches
        0       cpu-migrations
```

请注意，cpu-migrations 的数量变为 0，即进程永远不会离开 core0。

或者，您可以使用 cset: [https://github.com/lpechacek/cpuset<sup>251</sup>](https://github.com/lpechacek/cpuset) 工具仅为要基准测试的程序预留 CPU。如果使用 Linux perf，请至少保留两个内核，以便 perf 在一个内核上运行，您的程序在另一个内核上运行。以下命令将从 N1 和 N2 中移动所有线程 (-k on 表示即使内核线程也会被移动出去):

```
$ cset shield -c N1,N2 -k on
```

以下命令将在隔离的 CPU 上运行 -- 后面的命令:

```
$ cset shield --exec -- perf stat -r 10 <cmd>
```

## 进程优先级 (Process Priority)

在 Linux 中，可以使用 nice 工具提高进程优先级。通过提高优先级，进程可以获得更多的 CPU 时间，并且 Linux 调度器会比具有正常优先级的进程更青睐它。优先级范围从 -20（最高优先级值）到 19（最低优先级值），默认值为 0。

请注意在上一个示例中，基准测试进程的执行被操作系统中断超过 100 次。如果我们通过使用 sudo nice -n -N 运行基准测试来提高进程优先级：

```
$ perf stat -r 10 -- sudo nice -n -5 taskset -c 1 a.exe
      0      context-switches
      0       cpu-migrations
```

请注意，上下文切换的数量变为 0，因此该进程不间断地接收所有计算时间。

<sup>249</sup> 处理器关联性 - [https://en.wikipedia.org/wiki/Processor\\_affinity](https://en.wikipedia.org/wiki/Processor_affinity).

<sup>250</sup> taskset 手册 - <https://linux.die.net/man/1/taskset>.

<sup>251</sup> cpuset 手册 - <https://github.com/lpechacek/cpuset>.

## 文件系统缓存 (Filesystem Cache)

通常，会分配一部分主内存来缓存文件系统内容，包括各种数据。这减少了应用程序访问磁盘的次数。以下示例说明了文件系统缓存如何影响简单 `git status` 命令的运行时间：

```
# 清空文件系统缓存
$ echo 3 | sudo tee /proc/sys/vm/drop_caches && sync && time -p git status
real 2,57
# 预热文件系统缓存
$ time -p git status
real 0,40
```

可以通过运行以下两个命令清除当前的文件系统缓存：

```
$ echo 3 | sudo tee /proc/sys/vm/drop_caches
$ sync
```

或者，您可以进行一次干运行以预热文件系统缓存，并将其排除在测量之外。此干运行可以与基准测试输出的验证结合使用。

## 附录 B. LLVM 向量化 (Vectorizer)

本章节描述了截至 2020 年 Clang 编译器内部 LLVM 循环向量化的状态。内部循环向量化是将最内层循环中的代码转换为使用跨多个循环迭代的矢量的代码的过程。SIMD 向量中的每个通道对连续的循环迭代执行独立算术运算。通常，循环不会处于干净状态，向量化必须猜测和假设缺少的信息并会在运行时检查细节。如果假设错误，向量化会退回到运行标量循环。下面的示例突出了一些 LLVM 向量化支持的代码模式。

### 循环次数未知

LLVM 循环向量化支持循环次数未知的情况。在下面的循环中，迭代开始和结束点未知，向量化有一个机制可以向量化不以零开始的循环。在这个例子中，`n` 可能不是向量宽度的倍数，向量化必须将最后几个迭代作为标量代码执行。保留循环的标量副本会增加代码大小。

```
void bar(float* A, float* B, float K, int start, int end) {
    for (int i = start; i < end; ++i)
        A[i] *= B[i] + K;
}
```

### 指针的运行时检查

在下面的示例中，如果指针 `A` 和 `B` 指向连续的地址，则向量化代码是非法的，因为某些 `A` 的元素会在从数组 `B` 读取之前写入。

一些程序员使用 `restrict` 关键字通知编译器指针是不相交的，但在我们的例子中，LLVM 循环向量化无法知道指针 `A` 和 `B` 是唯一的。循环向量化通过放置代码来处理这个循环，该代码在运行时检查数组 `A` 和 `B` 是否指向不相交的内存位置。如果数组 `A` 和 `B` 重叠，则执行循环的标量版本。

```
void bar(float* A, float* B, float K, int n) {
    for (int i = 0; i < n; ++i)
        A[i] *= B[i] + K;
}
```

### 归约

在这个例子中，求和变量被循环的连续迭代使用。通常，这会阻止向量化，但向量化可以检测到 `sum` 是一个归约变量。变量 `sum` 成为一个整数向量，在循环结束时，数组的元素被相加以创建正确的结果。LLVM 循环向量化支持许多不同的归约操作，例如加、乘、异或、与和或。

```
int foo(int *A, int n) {
    unsigned sum = 0;
    for (int i = 0; i < n; ++i)
        sum += A[i] + 5;
    return sum;
}
```

使用 `-ffast-math` 时，LLVM 循环向量化支持浮点归约操作。

## 归纳

在这个例子中，归纳变量 `i` 的值被保存到一个数组中。LLVM 循环向量化知道如何向量化归纳变量。

```
void bar(float* A, int n) {
    for (int i = 0; i < n; ++i)
        A[i] = i;
}
```

## If 转换

LLVM 循环向量器能够“平铺”代码中的 IF 语句并生成单个指令流。向量化支持最内层循环中的任何控制流。最内层循环可能包含复杂的 IF、ELSE 甚至 GOTO 的嵌套。

```
int foo(int *A, int *B, int n) {
    unsigned sum = 0;
    for (int i = 0; i < n; ++i)
        if (A[i] > B[i])
            sum += A[i] + 5;
    return sum;
}
```

## 指针归纳变量 {.unnumbered .unlisted}

此示例使用标准 C++ 库中的 `std::accumulate` 函数。此循环使用 C++ 迭代器，它们是指针，而不是整数索引。LLVM 循环向量化器检测指针归纳变量并可以向量化此循环。此功能很重要，因为许多 C++ 程序使用迭代器。

```
int baz(int *A, int n) {
    return std::accumulate(A, A + n, 0);
}
```

## 反向迭代器

LLVM 循环向量化器可以向量化反向计数的循环。

```
int foo(int *A, int n) {
    for (int i = n; i > 0; --i)
        A[i] += 1;
}
```

## Scatter / Gather

LLVM 循环向量化器可以向量化成为分散/收集内存的标量指令序列的代码。

```
int foo(int *A, int *B, int n) {
    for (intptr_t i = 0; i < n; ++i)
        A[i] += B[i * 4];
}
```

在许多情况下，成本模型会认为这种转换是无利可图的。

## 混合类型的向量化

LLVM 循环向量化器可以向量化具有混合类型的程序。向量化器成本模型可以估计类型转换的成本并决定向量化是否有利可图。

```
int foo(int *A, char *B, int n) {
    for (int i = 0; i < n; ++i)
        A[i] += 4 * B[i];
}
```

## 函数调用的向量化

LLVM 循环向量化器可以向量化内在数学函数。有关这些功能的列表，请参见下表。

|         |       |           |
|---------|-------|-----------|
| pow     | exp   | exp2      |
| sin     | cos   | sqrt      |
| log     | log2  | log10     |
| fabs    | floor | ceil      |
| fma     | trunc | nearbyint |
| fmuladd |       |           |

## 向量化过程中的部分展开

现代处理器具有多个执行单元，只有包含高度并行性的程序才能充分利用机器的全部宽度。LLVM 循环向量化器通过执行循环的部分展开来增加指令级并行度 (ILP)。

在下面的示例中，整个数组累积到变量 `sum` 中。这是低效的，因为处理器只能使用单个执行端口。通过展开代码，循环向量化器允许同时使用两个或多个执行端口。

```
int foo(int *A, int n) {
    unsigned sum = 0;
    for (int i = 0; i < n; ++i)
        sum += A[i];
    return sum;
}
```

LLVM 循环向量化器使用成本模型来决定何时展开循环是有利的。展开循环的决定取决于寄存器压力和生成的代码大小。

## SLP 向量化

SLP（超字级并行 Superword-Level Parallelism）向量化器试图将多个标量操作粘合在一起形成向量操作。它自下而上地处理代码，跨越基本块，以寻找要组合的标量。SLP 向量化的目标是将类似的独立指令组合成向量指令。内存访问、算术运算、比较运算都可以使用这种技术进行向量化。例如，以下函数对其输入 ( $a_1, b_1$ ) 和 ( $a_2, b_2$ ) 执行非常相似的操作。基本块向量化器可以将以下函数组合成向量运算。

```
void foo(int a1, int a2, int b1, int b2, int *A) {
    A[0] = a1*(a1 + b1);
    A[1] = a2*(a2 + b2);
    A[2] = a1*(a1 + b1);
```

```
A[3] = a2*(a2 + b2);  
}
```

## 附录 C. 启用大页面

### 11.10 Windows

要在 Windows 上使用大页面，需要启用 SeLockMemoryPrivilege 安全策略 (链接到微软文档: <https://docs.microsoft.com/en-us/windows/security/threat-protection/security-policy-settings/lock-pages-in-memory>). 这可以通过 Windows API 以编程方式完成，也可以通过安全策略 GUI 完成。

1. 点击开始 -> 搜索 “secpol.msc”，启动它。
2. 在左侧选择“本地策略”->“用户权利分配”，然后双击“锁定内存中的页面”。
3. 添加您的用户并重新启动机器。
4. 使用 RAMMap: <https://docs.microsoft.com/en-us/sysinternals/downloads/rammap> 工具检查运行时是否使用了大页面。

在代码中使用大页面：

```
void* p = VirtualAlloc(NULL, size, MEM_RESERVE |
                      MEM_COMMIT |
                      MEM_LARGE_PAGES,
                      PAGE_READWRITE);
...
VirtualFree(ptr, 0, MEM_RELEASE);
```

### 11.11 Linux

在 Linux 操作系统上，应用程序可以使用大页面的两种方式：显式和大页面透明分配。

#### 11.11.1 显式大页面 (.unnumbered)

显式大页面可以在启动时或运行时预留。要在启动时强制 Linux 内核分配 128 个大页面，请运行以下命令：

```
$ echo "vm.nr_hugepages = 128" >> /etc/sysctl.conf
```

要显式分配固定数量的大页面，可以使用 libhugetlbfs: <https://github.com/libhugetlbfs/libhugetlbfs>。以下命令预分配 128 个大页面。

```
$ sudo apt install libhugetlbfs-bin
$ sudo hugeadm --create-global-mounts
$ sudo hugeadm --pool-pages-min 2M:128
```

这大致相当于执行以下命令，不需要 libhugetlbfs (参见内核文档：链接到内核文档: <https://www.kernel.org/doc/Documentation/vm/hugetlpage.txt>):

```
$ echo 128 > /proc/sys/vm/nr_hugepages
$ mount -t hugetlbfs \
-o uid=<value>,gid=<value>,mode=<value>,pagesize=<value>,size=<value>, \
min_size=<value>,nr_inodes=<value> none /mnt/huge
```

您应该可以在 /proc/meminfo 中观察到效果。请注意，这是一个系统范围的视图，而不是针对每个进程：

```
$ watch -n1 "cat /proc/meminfo | grep huge -i"
AnonHugePages:      2048 kB
ShmemHugePages:     0 kB
FileHugePages:      0 kB
HugePages_Total:    128    <== 128 huge pages allocated
HugePages_Free:     128
HugePages_Rsvd:     0
HugePages_Surp:     0
Hugepagesize:       2048 kB
Hugetlb:           262144 kB <== 256MB of space occupied
```

开发人员可以在代码中通过调用带有 MAP\_HUGETLB 标志的 mmap 来利用显式大页面（完整示例<sup>252</sup>）：

```
void ptr = mmap(nullptr, size, PROT_READ | PROT_WRITE,
                MAP_PRIVATE | MAP_ANONYMOUS | MAP_HUGETLB, -1, 0);
...
munmap(ptr, size);
```

其他替代方案包括：

- 使用来自已挂载的 hugetlbfs 文件系统的文件进行 mmap（示例代码<sup>253</sup>）。
- 使用 SHM\_HUGETLB 标志的 shmemget（示例代码<sup>254</sup>）。

## 透明大页内存

要在 Linux 上允许应用程序使用透明大页 (THP)，需要确保 /sys/kernel/mm/transparent\_hugepage/enabled 设置为 always 或 madvise。前者启用系统范围的 THP 使用，而后者则将对哪些内存区域应使用 THP 的控制权交给用户代码，从而避免消耗更多内存资源的风险。以下是使用 madvise 方法的示例：

```
void ptr = mmap(nullptr, size, PROT_READ | PROT_WRITE | PROT_EXEC,
                MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
madvise(ptr, size, MADV_HUGEPAGE);
...
munmap(ptr, size);
```

您可以在 /proc/meminfo 下的 AnonHugePages 中观察到系统范围的效果：

```
$ watch -n1 "cat /proc/meminfo | grep huge -i"
AnonHugePages:      61440 kB    <== 30 transparent huge pages are in use
HugePages_Total:    128
HugePages_Free:     128    <== explicit huge pages are not used
```

此外，开发人员可以通过查看针对其进程的特定 smaps 文件来观察其应用程序如何利用 EHP 和/或 THP：

```
$ watch -n1 "cat /proc/<PID_OF_PROCESS>/smaps"
```

<sup>252</sup> MAP\_HUGETLB 示例 - [https://github.com/torvalds/linux/blob/master/tools/testing/selftests/vm/map\\_hugetlb.c](https://github.com/torvalds/linux/blob/master/tools/testing/selftests/vm/map_hugetlb.c).

<sup>253</sup> 已挂载的 hugetlbfs 文件系统 - <https://github.com/torvalds/linux/blob/master/tools/testing/selftests/vm/hugepage-mmap.c>.

<sup>254</sup> SHM\_HUGETLB 示例 - <https://github.com/torvalds/linux/blob/master/tools/testing/selftests/vm/hugepage-shm.c>.

## 附录 D. Intel 处理器跟踪

Intel 处理器跟踪 (PT) 是一种 CPU 功能，通过将数据包编码为高度压缩的二进制格式记录程序执行，可以在每条指令上附带时间戳，用于重构执行流。PT 具有广泛的覆盖范围和相对较小的开销，<sup>255</sup> 通常低于 5%。其主要用途是事后分析和排查性能故障的根本原因。

### 工作流程

与采样技术类似，PT 不需要对源代码进行任何修改。收集跟踪的全部需要就是在支持 PT 的工具下运行程序。一旦启用了 PT 并启动了基准测试，分析工具就会开始将跟踪数据包写入 DRAM。

与 LBR (Last Branch Records) 类似，Intel PT 通过记录分支来工作。在运行时，每当 CPU 遇到任何分支指令时，PT 就会记录该分支的结果。对于简单的条件跳转指令，CPU 会记录其是否被执行 (T) 或未被执行 (NT)，仅使用 1 位。对于间接调用，PT 将记录目标地址。请注意，由于我们静态知道无条件分支的目标，因此会忽略无条件分支。

示例中展示了一小段指令序列的编码，如图 86 所示。诸如 PUSH、MOV、ADD 和 CMP 等指令被忽略，因为它们不会改变控制流。但是，JE 指令可能会跳转到 .label，因此需要记录其结果。稍后存在一个间接调用，需要保存目标地址。

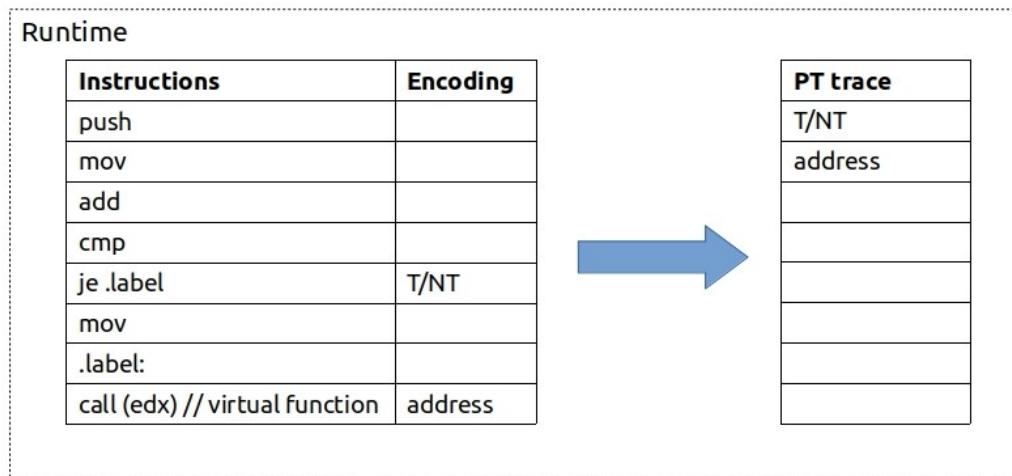


Figure 86: Intel 处理器跟踪编码

在分析时，我们需要将应用程序二进制文件和收集到的 PT 跟踪数据合并在一起。一个软件解码器需要应用程序二进制文件来重构程序的执行流程。它从入口点开始，然后使用收集到的跟踪数据作为查找参考来确定控制流。图 87 展示了 Intel 处理器跟踪的解码示例。假设 PUSH 指令是应用程序二进制文件的入口点。然后 PUSH、MOV、ADD 和 CMP 等指令被按原样重构，而无需查看编码的跟踪数据。稍后，软件解码器遇到一个 JE 指令，这是一个条件分支，需要查找结果。根据图 87 中的跟踪数据，JE 被执行 (T)，因此我们跳过下一个 MOV 指令并转到 CALL 指令。同样，CALL(edx) 是一个改变控制流的指令，因此我们在编码的跟踪数据中查找目标地址，即 0x407e1d8。在我们的程序运行时执行的指令用黄色突出显示。请注意，这是程序执行的精确重构；我们没有跳过任何指令。稍后，我们可以使用调试信息将汇编指令映射回源代码，并记录逐行执行的源代码日志。

<sup>255</sup> 有关英特尔 PT 额外开销的更多信息，请参见 [Sharma, 2016]。

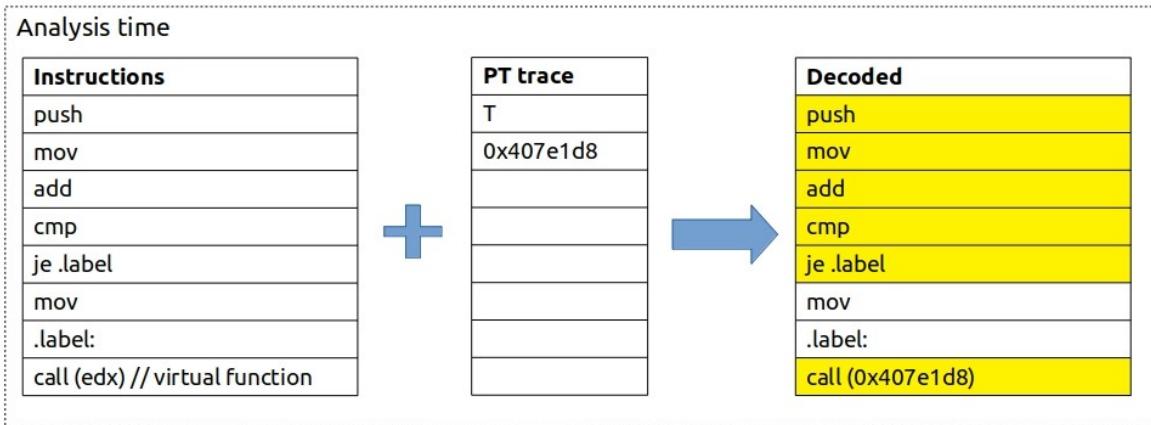


Figure 87: Intel 处理器跟踪解码

## 时间数据包

使用 Intel PT，不仅可以跟踪执行流，还可以跟踪时间信息。除了保存跳转目标外，PT 还可以发出时间数据包。图 88 提供了时间数据包如何用于恢复指令时间戳的可视化示例。与前面的示例类似，我们首先看到 JNZ 没有被执行，因此我们将其及其上方的所有指令的时间戳更新为 0 纳秒。然后我们看到一个 2 纳秒的时间更新和 JE 被执行，因此我们将其及其上方的所有指令（以及下方的 JNZ）的时间戳更新为 2 纳秒。之后是一个间接调用，但没有附加时间数据包，因此我们不更新时间戳。然后我们看到经过了 100 纳秒，并且 JB 没有被执行，因此我们将其上方的所有指令的时间戳更新为 102 纳秒。

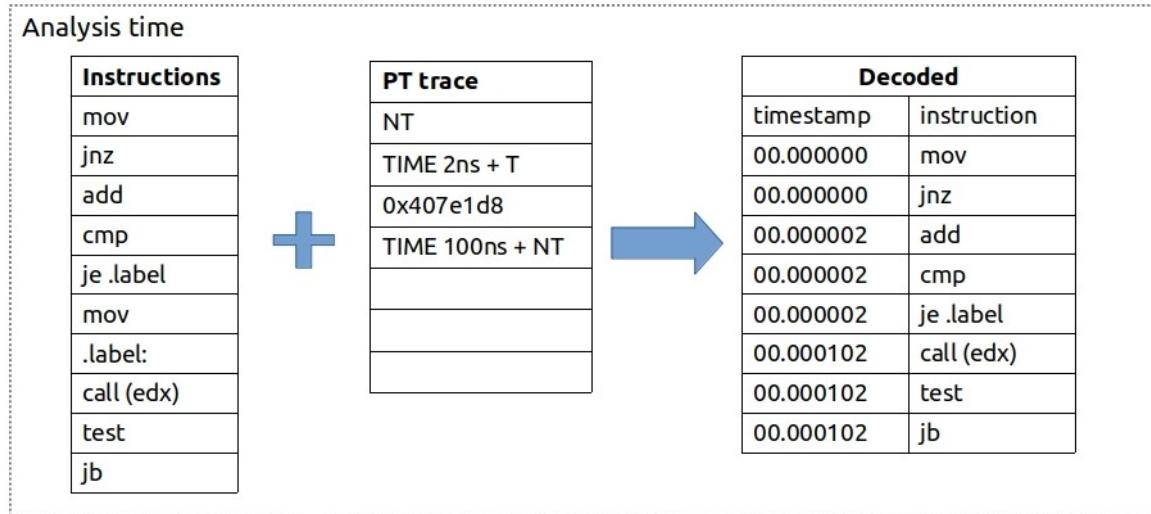


Figure 88: Intel 处理器跟踪时间数据包

在图 88 中展示的示例中，指令数据（控制流）完全准确，但时间信息不太准确。显然，CALL(edx)、TEST 和 JB 指令并不是同时发生的，但我们没有更准确的时间信息。具有时间戳使我们能够将程序的时间间隔与系统中的另一个事件对齐，并且很容易与挂钟时间进行比较。在某些实现中，跟踪时间可以通过一个循环精确模式进一步改进，其中硬件记录了常规数据包之间的周期计数（有关更多详细信息，请参阅 [Intel, 2023b, 第 3C 卷, 第 36 章]）。

## 收集和解码跟踪

使用 Linux 的 perf 工具可以轻松地收集 Intel PT 跟踪数据：

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

在上面的命令行中，我们要求 PT 机制每个周期更新一次时间信息。但是很可能，这不会显著增加我们的准确性，因为只有在与另一个控制流数据包配对时，才会发送时间数据包。

收集完跟踪数据后，可以通过执行以下命令来获取原始的 PT 跟踪数据：

```
$ perf report -D > trace.dump
```

PT 在发出时间数据包之前会捆绑最多 6 个条件分支。自 Intel Skylake CPU 一代以来，时间数据包包含自上一个数据包以来经过的周期数。然后，如果我们查看 trace.dump，可能会看到类似以下的内容：

```
000073b3: 2d 98 8c TIP 0x8c98 // 目标地址 (IP)
000073b6: 13 CYC 0x2 // 时间更新
000073b7: c0 TNT TNNTNNN (6) // 6 个条件分支
000073b8: 43 CYC 0x8 // 经过 8 个周期
000073b9: b6 TNT NTTNTT (6)
```

以上是显示的原始 PT 数据包，对性能分析并不是非常有用。要将处理器跟踪数据解码为人类可读形式，可以执行以下命令：

```
$ perf script --ns --itrace=i1t -F time,srcline,insn,srccode
```

以下是可能获得的解码跟踪的示例：

| 时间戳            | 源代码行     | 指令              | 源代码                          |
|----------------|----------|-----------------|------------------------------|
| ...            |          |                 |                              |
| 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 |                              |
| ...            |          |                 |                              |

以上只是来自长时间执行日志的一个小片段。在此日志中，我们有跟踪每个执行的指令，而我们的程序正在运行时。我们可以真正观察程序所采取的每一步。这是进一步功能和性能分析的非常强大的基础。

## 使用情况

- 分析性能故障：由于 PT 捕获了整个指令流，因此可以分析应用程序未响应的小时间段内发生了什么。在 easyperf 博客的一篇文章<sup>256</sup>中可以找到更详细的示例。
- 事后调试：PT 跟踪数据可以由像 gdb 这样的传统调试器进行回放。除此之外，PT 还提供了调用堆栈信息，即使堆栈已损坏，该信息也始终有效。<sup>257</sup>可以在远程机器上收集 PT 跟踪数据，然后在离线状态下进行分析。这在问题难以重现或系统访问受限时特别有用。
- 审查程序的执行：
  - 我们可以立即知道是否未执行某个代码路径。

<sup>256</sup> 使用英特尔 PT 分析性能故障 - <https://easyperf.net/blog/2019/09/06/Intel-PT-part3。>

<sup>257</sup> 使用英特尔 PT 进行事后调试 - <https://easyperf.net/blog/2019/08/30/Intel-PT-part2。>

- 多亏了时间戳，可以计算在尝试获取锁时等待的时间，等等。
- 通过检测特定指令模式进行安全缓解。

## 磁盘空间和解码时间

即使考虑到跟踪的压缩格式，编码后的数据也会占用大量磁盘空间。通常，每个指令少于 1 个字节，但考虑到 CPU 执行指令的速度，它仍然很多。取决于工作负载，CPU 以 100 MB/s 的速度编码 PT 非常常见。解码的跟踪可能很容易增加十倍 (~1GB/s)。这使得 PT 不适用于长时间运行的工作负载。但即使是在大工作负载上，运行一小段时间也是负担得起的。在这种情况下，用户只能在故障发生期间附加到正在运行的进程。或者他们可以使用循环缓冲区，新跟踪将覆盖旧跟踪，即始终拥有最近 10 秒左右的跟踪。

用户可以通过多种方式进一步限制收集。他们可以限制仅在用户/内核空间代码上收集跟踪。此外，还有一个地址范围过滤器，因此可以动态地选择加入和退出跟踪以限制内存带宽。这使我们能够跟踪单个函数甚至单个循环。

解码 PT 跟踪可能需要很长时间。在 Intel Core i5-8259U 机器上，对于运行 7 毫秒的工作负载，编码的 PT 跟踪大约消耗 1MB 的磁盘空间。使用 `perf script` 解码此跟踪需要大约 20 秒。使用 `perf script -F time,ip,sym,symoff,insn` 的解码输出大约占用 1.3GB 的磁盘空间。截至 2020 年 2 月，使用 `perf script -F` 以及 `+srcline` 或 `+srccode` 解码跟踪变得非常慢，不适合日常使用。应该改进 Linux perf 的实现。

## 英特尔 PT 参考资料和链接

- 英特尔® 64 和 IA-32 架构软件开发人员手册 [Intel, 2023b, 第 3 卷 C, 第 36 章]。
- 白皮书“硬件辅助指令分析和延迟检测”[Sharma, 2016]。
- Andi Kleen 在 LWN 上的文章，网址：<https://lwn.net/Articles/648154>。
- 英特尔 PT 微型教程，网址：<https://sites.google.com/site/intelptmicrotutorial/>。
- simple\_pt: Linux 上的简单英特尔 CPU 处理器跟踪，网址：<https://github.com/andikleen/simple-pt>。
- Linux 内核中的英特尔 PT 文档，网址：<https://github.com/torvalds/linux/blob/master/tools/perf/Documentation/intel-pt.txt>。
- 英特尔处理器跟踪备忘单，网址：<http://halobates.de/blog/p/410>。