

体系结构相关及性能测试

杜忱莹 周辰霏

2021 年 3 月

安祺 曾泉胜

2022 年 2 月

目录

1 体系结构相关实验分析——cache 优化	3
1.1 实验介绍	3
1.2 实验设计指导	3
1.2.1 题目分析	3
1.2.2 算法设计与编程	4
1.3 程序编译和运行	5
1.4 实验结果分析及其他思考	5
2 体系结构相关实验分析——超标量优化	6
2.1 实验介绍	6
2.2 实验设计指导	6
2.2.1 题目分析	6
2.2.2 算法设计与编程	7
2.3 程序编译、运行、结果分析及更多思考	7
3 使用 VTune 剖析程序性能	8
3.1 VTune 简要介绍	8
3.2 VTune 的安装及使用	8
3.2.1 VTune 安装	8
3.2.2 VTune 使用	8
3.3 体系结构相关的程序性能剖析	9
3.3.1 cache 命中率	9
3.3.2 超标量	12
4 使用 Perf 剖析程序性能	14
4.1 Perf 简要介绍	14
4.2 Perf 的安装及使用	15
4.2.1 Perf 安装	15
4.2.2 Perf 使用	15
4.3 体系结构相关的程序性能剖析	17
4.3.1 cache 命中率	17
4.3.2 超标量	18

1 体系结构相关实验分析——cache 优化

1.1 实验介绍

给定一个 $n \times n$ 矩阵，计算每一列与给定向量的内积，考虑两种算法设计思路：逐列访问元素的平凡算法和 cache 优化算法，进行实验对比：

1. 对两种思路的算法编程实现；
2. 练习使用高精度计时测试程序执行时间，比较平凡算法和优化算法的性能。

1.2 实验设计指导

1.2.1 题目分析

本题与讲义中矩阵求和的例子非常相似，都是关于如何设计算法，令访存模式具有更好的空间局部性，从而发挥 cache 的能力。

平凡算法逐列访问矩阵元素，一步外层循环（内存循环一次完整执行）计算出一个内积结果；cache 优化算法则改为逐行访问矩阵元素，一步外层循环计算不出任何一个内积，只是向每个内积累加一个乘法结果。后者的访存模式具有很好空间局部性，令 cache 的作用得以发挥。

需要注意以下几点：

1. 实验设计（问题规模的确定、测试数据的生成等）。

对于本问题，测试数据人为设定固定值即可，例如 $A[i][j] = i + j$ ，方便程序正确性检查。

关于测试用例规模的确定，对本题而言，程序性能与 cache 相关，而现代 CPU cache 有多个层次，每个层次有固定规模（结合体系结构调研作业），因此，观察问题规模与 cache 大小不同关系时的程序性能变化，是有意义的。此外，在《实验教学指导书-1》中，我们介绍了，性能测试中应设置一系列问题规模，以研究程序性能随问题规模变化的趋势。因此，可设定一系列问题规模，其中体现于实验用 CPU 的各级 cache 大小对应的规模。

总之，实验的目的是为了说明本文提出的方法有好的性能，应围绕这一目的设计实验，明确每个实验想要说明什么，如 A 方法比 B 方法速度快（测试不同问题规模下两种方法的时间）、A 方法加入算法策略 X 后可加速（测试不同问题规模下 A 方法和 A 方法 + 策略 X 的时间），再如对于本问题设置对应各级 cache 大小的问题规模来研究 cache 对性能的影响等。

在实验报告中，在“实验”一节应详细描述实验设计。

2. 程序分析。

一方面，可对算法进行理论分析（包括程序性能随问题规模变化的趋势，以及问题达到与各级 cache 大小相对应规模时程序性能变化等），与性能测试结果相对照。与大家熟悉的算法分析不同，这里除了考虑计算次数外，还需考虑访存开销。

另一方面，可采用 VTune 等工具获取程序运行时系统层面的一些指标（如 cache 命中率等），揭示性能表现的内在原因，同时也可与理论分析对照。VTune 的使用见后面小节。

3. 程序测试（运行时间测量）。

参考《实验教学指导书-1》中的程序运行时间测量方法。由于本问题计算较为简单，当矩阵规模较小时，程序运行时间很短。因此，可采用重复运行待测函数、延长计时间隔的方法，来解决计时函数精度不够、影响测量精度的问题。

1.2.2 算法设计与编程

编程思路与讲义中矩阵列求和的例子几乎一致：

• 平凡算法

```
1 // 逐列访问矩阵元素：一步外层循环（内存循环一次完整执行）计算出一个内积结果
2 for(i = 0; i < n; i++){
3     sum[i] = 0.0;
4     for(j = 0; j < n; j++)
5         sum[i] += b[j][i] * a[j];
6 }
```

• 优化算法

```
1 // 改为逐行访问矩阵元素：一步外层循环计算不出任何一个内积，只是向每个内积累加一个乘法结果
2 for(i = 0; i < n; i++)
3     sum[i] = 0.0;
4 for(j = 0; j < n; j++)
5     for(i = 0; i < n; i++)
6         sum[i] += b[j][i] * a[j];
```

后者的访存模式与行主存储匹配，具有很好空间局部性，令 cache 作用得以发挥。

1.3 程序编译和运行

程序的编译、运行和结果查看可参考《实验教学指导书-1》。在前几次实验中，我们都是进行单节点并行编程练习，不涉及 MPI。因此，程序的编译不使用 mpicc，直接使用 gcc、clang（毕升编译器）即可。**程序提交 PBS** 作业系统、编写提交脚本时，只申请一个节点，不需要将程序拷贝到各节点，程序执行直接调用可执行文件、无须使用 mpiexec (mpirun) 命令（用 perf 等工具进行 profiling 的话直接写 perf 命令）。

在实验报告“实验”一节的开始，要清晰描述程序编译、运行的平台的详细信息，如 CPU 型号、频率、各级 cache 大小（与本问题相关）、内存大小等。

1.4 实验结果分析及其他思考

在科技论文中，实验结果的分析是非常重要的。

首先，**切忌以截图的方式呈现实验结果**。应记录下实验结果数据，绘制图、表格的形式呈现。表格用于呈现详细结果数据，图通常用来呈现变化趋势（如程序运行时间随问题规模变化趋势等）。如需在一张图中呈现较多方法的实验结果，可采用柱状图，每种方法的每个实验结果用一个柱呈现。

其次，正文中至少要做到“**看图说话**”，即，对图、表中的实验结果进行描述。如不同方法的性能对比、文章提出的新方法较之 baseline 性能提升幅度 xx%、随着问题规模变化性能变化趋势、最高性能最低性能是多少、一些算法策略对性能影响幅度等等。

更重要的，需要**对实验结果进行合理分析解释**。为什么 A 方法比 B 方法好、为什么加入策略 X 后 A 方法性能有明显提升、为什么随着问题规模增大 A 方法时间增加幅度比 B 方法慢等等，从算法设计的角度分析产生这样的实验结果的原因。最好能与算法设计和分析小节中对算法的理论分析呼应上，这样，理论和实验相互印证会有更强的说服力。另外，采用 profiling 工具分析程序运行过程中硬件、系统软件的一些监测值来说明产生程序性能的底层原因，也是一种好的方法。总之，实验结果的分析一般是为了说明本文提出方法的确有好的性能，并通过与理论分析呼应等方式增强说服力。

另外，如作业中所描述，可以尝试不同的算法设计和实验方式。

2 体系结构相关实验分析——超标量优化

2.1 实验介绍

计算 n 个数的和，考虑两种算法设计思路：逐个累加的平凡算法（链式）；适合超标量架构的指令级并行算法（相邻指令无依赖），如最简单的两路链式累加，再如递归算法——两两相加、中间结果再两两相加，依次类推，直至只剩下最终结果。完成如下作业：

1. 对两种算法思路编程实现；
2. 练习使用高精度计时测试程序执行时间，比较平凡算法和优化算法的性能。

2.2 实验设计指导

2.2.1 题目分析

本题需要关注以下几点：

1. **测试数据的生成**。同上一个题目一样，数据生成人为指定即可，元素个数 n 取 2 的幂即可，方便递归算法设计。

2. **循环处理**。几个算法基本实现方式都是采用循环，但可能带来严重的额外开销——每个元素只进行一次加法，但却需要进行循环判定、归纳变量递增等多个额外操作。可采用循环展开（unroll）策略——每个循环步进行多次加法运算，相当于将多个循环步的工作展开到一个循环步，从而大幅度降低簿记操作的比例。甚至可以采用宏/模板将循环完全去掉。但要注意，不同算法尽量保持相同的展开比例，保证性能对比的公平性。循环展开可结合指令级并行，即，合并到一个循环步中的多个计算通过合理设计令它们相互不依赖，可同时由多条流水线处理。

3. **中间结果处理**。递归算法每个步骤会得到大量中间结果，可在输入数组中原地保存（输入的元素不再被使用的话），也可分配一个辅助数组保存。元素访问顺序要小心设计，注意空间局部性。

4. **问题规模（元素个数）设置**。同样可考虑流水线条数、cache 大小等系统参数来设置实验中的问题规模。

5. **较小问题规模执行**。当问题规模较小时执行时间可能很短，可将核心计算重复多次，以提高性能测试和 profiling 的精度，如上一题。

2.2.2 算法设计与编程

- 平凡算法

```
1 // 链式：将给定元素依次累加到结果变量即可
2 for (i = 0; i < n; i++)
3     sum += a[i];
```

- 优化算法

```
1 // 多链路式
2 sum1 = 0; sum2 = 0
3 for (i = 0; i < n; i += 2) {
4     sum1 += a[i];
5     sum2 += a[i + 1];
6 }
7 sum = sum1 + sum2;
8
9 // 递归：
10 1. 将给定元素两两相加，得到n/2个中间结果；
11 2. 将上一步得到的中间结果两两相加，得到n/4个中间结果；
12 3. 依此类推，log(n)个步骤后得到一个值即为最终结果。
13
14 // 实现方式1：递归函数，优点是简单，缺点是递归函数调用开销较大
15 function recursion(n)
16 {
17     if (n == 1)
18         return;
19     else
20     {
21         for (i = 0; i < n / 2; i++)
22             a[i] += a[n - i - 1];
23         n = n / 2;
24         recursion(n);
25     }
26 }
27
28 // 实现方式2：二重循环
29 for (m = n; m > 1; m /= 2) // log(n)个步骤
30     for (i = 0; i < m / 2; i++)
31         a[i] = a[i * 2] + a[i * 2 + 1] // 相邻元素相加连续存储到数组最前面
32 // a[0]为最终结果
```

2.3 程序编译、运行、结果分析及更多思考

参见上一题和作业内容。

一个可以探索的问题，如果是进行浮点运算，计算次序的改变可能会导致结果变化（计算机表示浮点数精度有限导致），而本问题的指令级并行算法与串行算法中元素累加顺序是不同的，可对此进行探索。

3 使用 VTune 剖析程序性能

3.1 VTune 简要介绍

VTune 是 Intel 推出的一款可视化的性能剖析（profiling）工具，支持的平台包括：Windows，Linux 和 macOS。本实验展示的是在 Windows 系统上的应用，在 Linux 上的安装和使用可参考[官方文件](#)。所谓 profiling 是指通过对目标收集采样或快照来归纳目标特征。例如分析 CPU 的使用率时，可以通过对程序计数器采样，或者跟踪栈来找到消耗 CPU 周期的代码路径，从而找到程序中的占用 CPU 使用率高的函数。通过对程序的性能分析，可以帮助开发人员针对系统资源的使用来优化代码。常见的剖析工具有：DTrace、perf、VTune 等。这里我们介绍 VTune 的使用。

3.2 VTune 的安装及使用

3.2.1 VTune 安装

VTune 在 Intel 官网即可免费下载安装：<https://software.intel.com/en-us/vtune/choose-download>，这里展示使用的是 VTune_Amplifier_2019 版本在 windows 系统上的使用。

3.2.2 VTune 使用

VTune 需要以管理员的身份打开，打开进入页面并创建新项目。

点击 Configure Analysis 进入分析界面，左下角 Launch Application 中选中需要进行测试的程序以及输入参数，右侧提供了：Hotspots, Microarchitecture, Parallelism 等多种分析类型。设置 CPU 的采样间隔时间，以及额外的测试内容即可测试对应程序性能。

以 Hotspots 为例，选中并进行测试，可以采集到 collection log, Summary, Bottom-up, Caller/Callee, Top-down Tree Platform 数据。如图 3.4 所示，Summary 主要分析的数据有：执行时间，高热点部分，CPU 使用直方图以及收集信息和平台信息。在这里可以看到总开销时间，程序中最耗时的

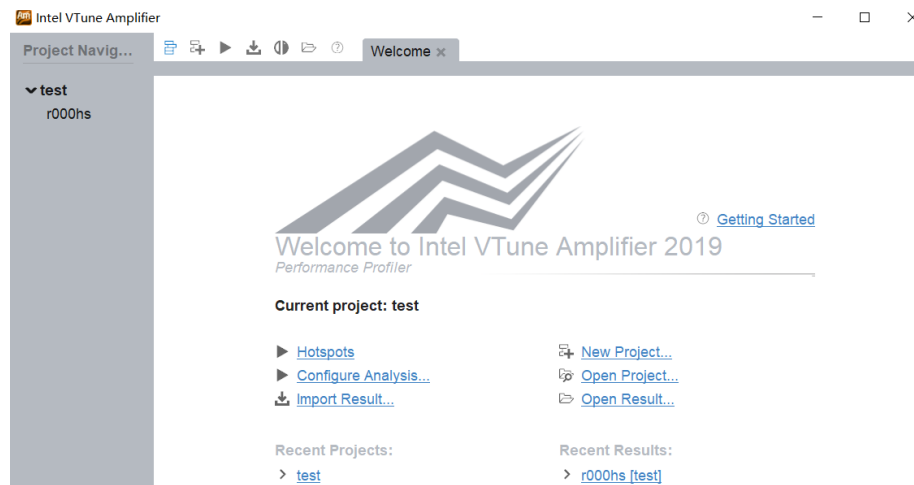


图 3.1: VTune 主页面

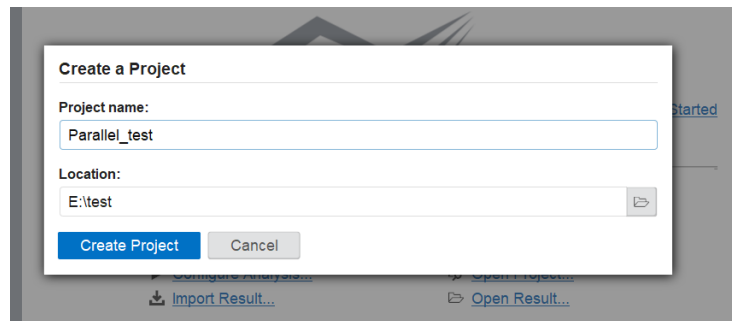


图 3.2: 创建项目

部分等内容。Bottom-top 可以查看函数/线程调用时间。具体各数据类型大家可以自己进行查看，在这里不一一列举。接下来我们举两个例子来展示 VTune 的基本使用。

3.3 体系结构相关的程序性能剖析

3.3.1 cache 命中率

接下来我们以数组列求和的例子展示 VTune 分析程序性能。

```
1 for (i = 0; i < 1000; i++)  
2   column_sum[i] = 0.0;
```

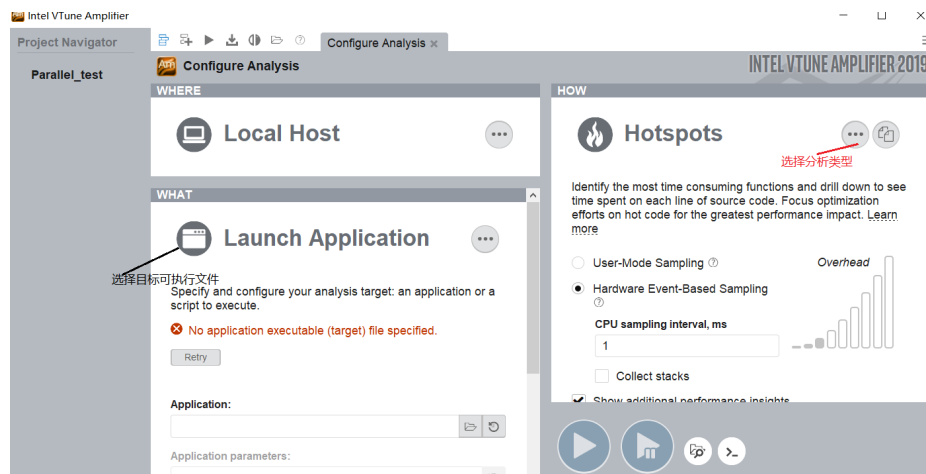


图 3.3: 配置分析项目

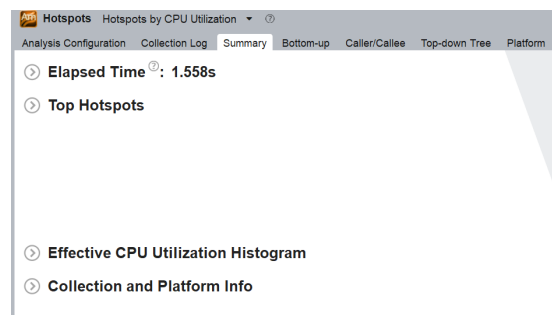


图 3.4: Hotspot 简单分析结果

```
3   for (j = 0; j < 1000; j++)  
4       column_sum[i] += b[j][i];
```

二维数组在 C/C++ 中为行主存储方式，这样按列访问数组可能会造成 cache 频繁 miss 的从而影响到程序执行的时间。

为分析此程序，我们希望看到程序执行时间以及 cache miss 次数。后者需要额外加入额外的 Event，如图 3.5，在 Hotspots 窗口中选择 “Hardware Event-Based Sampling”。然后编辑希望采样的事件，如图 3.6 所示，在 Events configured for CPU 中勾选 MEM_LOAD_RETIURED.L1_HIT、MEM_LOAD_RETIURED.L1_MISS、MEM_LOAD_RETIURED.L2_HIT、MEM_LOAD_RETIURED.L2_MISS、MEM_LOAD_RETIURED.L3_HIT、

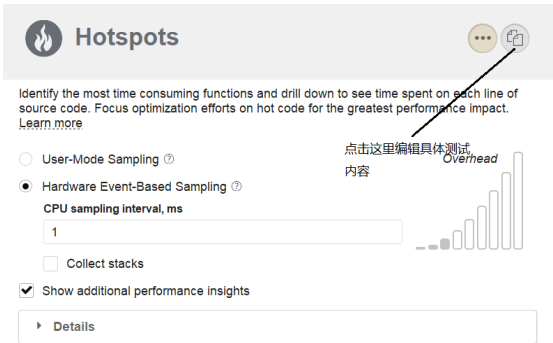


图 3.5: 选择硬件事件采集

MEM_LOAD_RETIURED.L3_MISS。VTune 就会采样 CPU L1,L2,L3 cache 的 HIT 以及 MISS 的次数。图 3.7给出了测试结果。

Events configured for CPU: Intel(R) microarchitecture code named Coffeelake

NOTE: For analysis purposes, Intel VTune Amplifier 2019 may adjust the Sample After values in the table below by a multiplier. The multiplier depends on the value of the Duration time estimate option specified in the target configuration window.

MEM_LOAD_RETIRED			Explain
Event Name	Sample ...	Description	
<input checked="" type="checkbox"/> MEM_LOAD_RETIRED.L1...	2000003	Retired load instructions with L1 ca...	
<input checked="" type="checkbox"/> MEM_LOAD_RETIRED.L1...	100003	Retired load instructions missed L1 ...	
<input checked="" type="checkbox"/> MEM_LOAD_RETIRED.L2...	100003	Retired load instructions with L2 ca...	
<input checked="" type="checkbox"/> MEM_LOAD_RETIRED.L2...	50021	Retired load instructions missed L2 ...	
<input checked="" type="checkbox"/> MEM_LOAD_RETIRED.L3...	50021	Retired load instructions with L3 ca...	
<input checked="" type="checkbox"/> MEM_LOAD_RETIRED.L3...	100007	Retired load instructions missed L3 ...	
<input type="checkbox"/> MEM_LOAD_RETIRED.F...	100007	Retired load instructions which data...	
<input type="checkbox"/> MEM_LOAD_RETIRED.F...	100003	Retired load instructions which data...	
<input type="checkbox"/> MEM_LOAD_RETIRED.L1...	2000003	Retired load instructions with L1 ca...	
<input type="checkbox"/> MEM_LOAD_RETIRED.L1...	100003	Retired load instructions missed L1 ...	
<input type="checkbox"/> MEM_LOAD_RETIRED.L2...	100003	Retired load instructions with L2 ca...	
<input type="checkbox"/> MEM_LOAD_RETIRED.L2...	50021	Retired load instructions missed L2 ...	

图 3.6: 选择采样事件

上述的代码中按列访问会导致较高的 cache miss 情况，我们重写代码：

```
1 for (i = 0; i < 1000; i++)
2     column_sum[i] = 0.0;
3 for (j = 0; j < 1000; j++)
4     for (i = 0; i < 1000; i++)
5         column_sum[i] += b[j][i];
```

这样的访问方式与行主存储器匹配，进行测试可以得到结果 cache miss 的次数更少。测试结果如图 3.8所示。

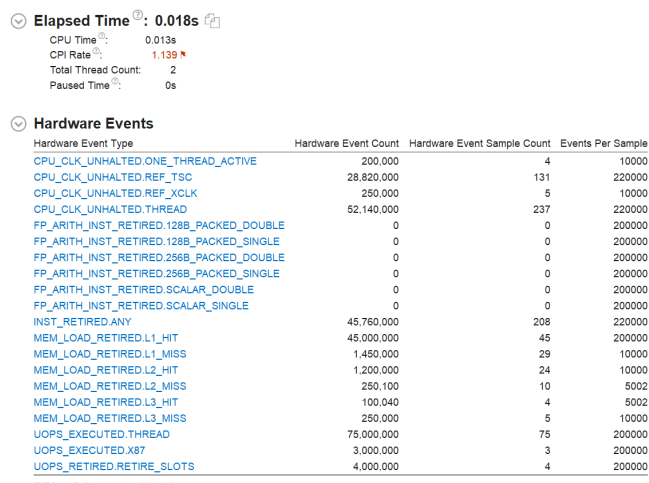


图 3.7: 列主次序访问算法的分析结果

3.3.2 超标量

以课堂介绍的 n 个数求和问题为例，两类算法设计思路：逐个累加的平凡算法（链式）；超标量优化算法，如最简单的两路链式累加，再如递归算法——两两相加、中间结果再两两相加，依次类推，直至只剩下最终结果。

我们比较两路链式累加算法和普通的链式算法的性能。对比普通的链式算法，两路链式算法能更好地利用 CPU 超标量架构，两条求和的链可令两条流水线充分地并发运行指令。有一个评价指标 IPC（Instruction Per Clock），即每个时钟周期执行的指令数，可以直观地比较这两种算法的区别。我们可以想到，两种算法所需的指令数大致相同，且两路链式算法同一时间令两条流水线充满，那么其 IPC 应该明显优于链式算法。接下来我们利用 VTune 来分析这两种算法的性能，验证我们的分析。

如图 3.9 所示，我们选择 Microarchitecture Exploration 类型（在 Bottom-up 中可以看到具体执行的周期数），Summary 数据下我们可以看到总体执行的周期数（Clockticks），执行指令数（Instructions Retired）以及 CPI（IPC 的倒数，每条指令执行的周期数）。接下来我们进入 Bottom-up 数据栏，如图 3.10 所示，在这一页面中我们可以看到具体每个函数执行的 CPU 时间，周期数以及执行指令数（gcc 编译时记得 -g 来加入调试程序使用的附加信息）。在 chain_unroll（链式算法实现函数）一栏，我们可以看到执行的指令数为 3,640,000（4096 个元素求和 500 次），而执行的周期数为 2,080,000。

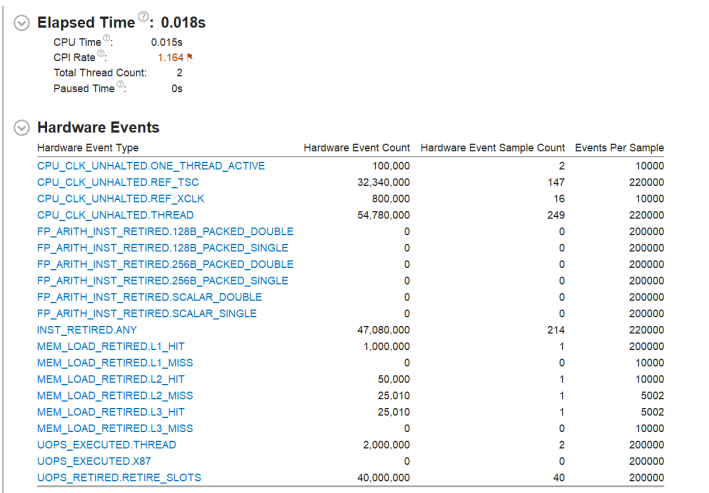


图 3.8: 行主次序访问算法的分析结果

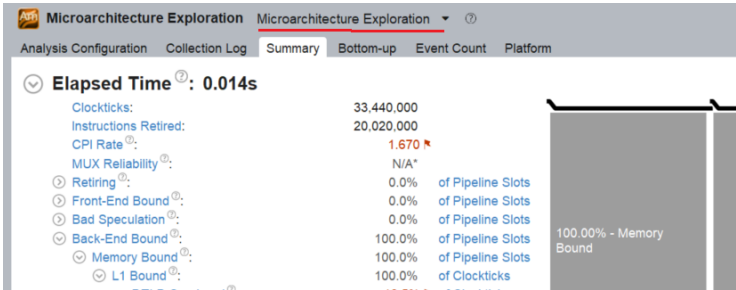


图 3.9: 超标量分析结果

其 CPI 为 0.571。

对比两路链式算法分析结果，如图 3.11所示，我们可以看到其执行指令数为 3,640,000，执行周期数为 1,300,000。其 CPI 为 0.357，明显优于链式算法的 0.571，这与我们之前的分析相同。大家可以将自己代码的测试结果与上述结果对比，看看有何异同。

VTune 的功能很强大，不仅能看到每个函数的执行情况，还可以看到具体每段代码以及对应汇编代码的执行情况。我们在图 3.11中双击 chain_2，可以看到 chain_2 函数中具体每段代码执行的次数以及执行周期数：

同样的，第一个例子中对 cache 命中率的测试也可以查看具体每个函数每段代码的 cache 命中率。这里只介绍了两个简单的程序性能分析示例，

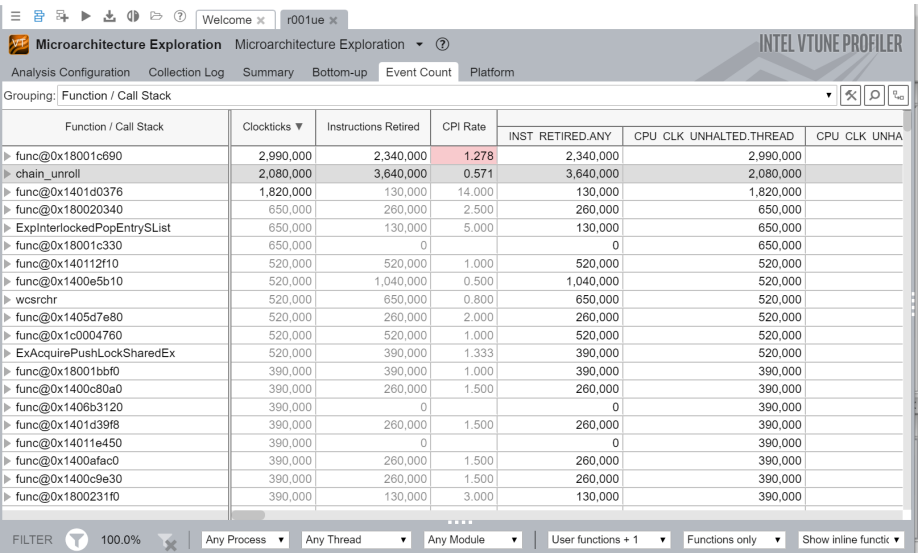


图 3.10: 链式算法详细结果

详细的 VTune 使用手册大家可以在网站：<https://software.intel.com/en-us/vtune-help> 官方文档中查询具体每一种数据的测试和分析。希望大家举一反三，灵活使用 VTune（或其他 profiling 工具）剖析程序性能。

4 使用 Perf 剖析程序性能

4.1 Perf 简要介绍

Perf(Performance Events for Linux) 是一个用于基于 Linux 2.6+ 的系统的分析器工具，它抽象出 Linux 性能测量中的 CPU 硬件差异，并提供了一个简单的命令行界面。该工具基于 Linux 内核提供的perf_events接口实现。

下文中，当命令行以 \$ 开头时，表示以非 root 用户执行，以 # 开头时，表示需要 root 权限。[关于 Ubuntu 中的 root 用户。](#)

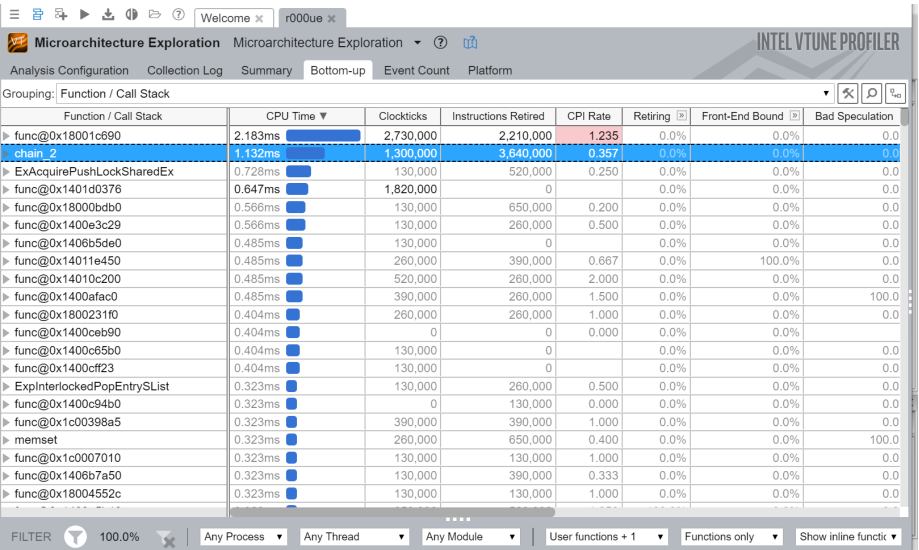


图 3.11: 两路链式算法详细结果

4.2 Perf 的安装及使用

4.2.1 Perf 安装

以 Ubuntu 20.04 为例，在 linux 系统中可以使用以下命令安装 perf 工具，如果下载速度过慢，可以考虑更换国内软件源，可参见[镜像源帮助页](#)。

```
sudo apt install linux-tools-$(uname -r) linux-tools-generic
```

注意，若 linux 系统安装在虚拟机中，需要打开虚拟机的“虚拟化 CPU 性能计数器”或“虚拟化 PMU”功能，否则 perf 将不支持硬件事件的采样。目前发现，VirtualBox、WSL2 暂不支持类似功能。

4.2.2 Perf 使用

使用 perf 对程序性能进行剖析，主要借助对事件的收集测量。事件分软件事件和硬件事件两类，软件事件主要包括上下文切换等 linux 内核提供的事件，硬件事件主要包括周期数、指令数、缓存未命中等体系结构相关事件，需要依赖硬件单元 PMU（Performance Monitoring Unit）。

可使用list命令查看在你的系统上有哪些事件，下面举几个例子。

```
perf list -h          #列出帮助
```

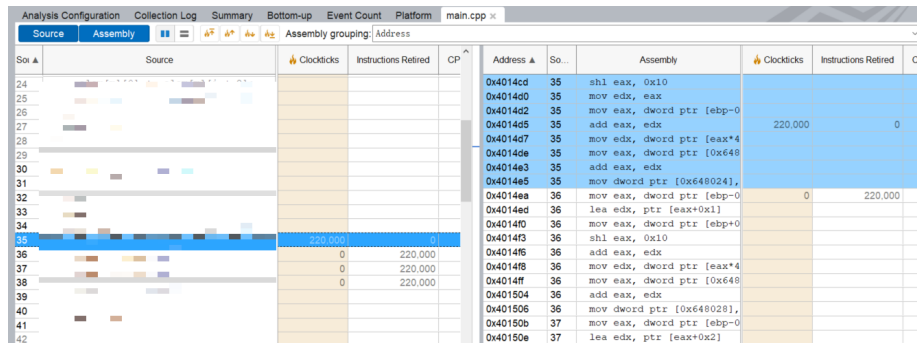


图 3.12: 两路链式算法执行执行分析

```

perf list hw          #列出硬件事件
perf list cache       #列出L1 cache相关事件
perf list pmu         #列出pmu相关事件
perf list l2_cache    #列出L2 cache相关事件
perf list pmu | grep cache #过滤包含"cache"的行

```

perf 提供了一系列命令来跟踪程序和分析性能，其中重点介绍stat, record, report。

Perf 有两种模式：

1. Counting 模式下（例如使用perf stat命令）会数出某事件在一段时间内一共发生了多少次
2. Sampling 模式下（例如使用perf record命令），在计数一定数量的时间后产生采样中断

perf stat对程序运行的事件数进行统计,给出最终结果,默认会对周期数,指令数等几项进行统计。可以使用-e <event1>,<event2>...选项指定对其他事件的统计,使用-r n选项进行n次重复统计,输入perf stat -h查看帮助。

perf record通过采样对程序运行信息进行手机,结果默认保存在当前目录的perf.data文件中。使用选项-e <event>指定收集的事件,使用-g记录运行过程中函数调用关系,使用-F <freq>指定采样的频率,使用选项-a则对所有 CPU 进行记录,使用选项-c n指定每n次事件发送进行一次采样,更多选项可输入perf record -h获得帮助。

perf report对收集得到的perf.data进行分析,默认进入交互式页面,初始界面按h获得帮助。而使用选项--stdio可一次性全部输出。

下一节就具体例子对重点功能做介绍说明，而更详细的使用说明可以参考下列文档。

1. linux perf 官方 WIKI:<https://perf.wiki.kernel.org/index.php/Tutorial>
2. 官方 WIKI 的一篇译文:<https://segmentfault.com/a/1190000021465563>
3. 一篇详细的 perf 用例(推荐):<https://www.brendangregg.com/perf.html>

此外，有几个方法可对 perf 进行可视化分析，有兴趣可进一步了解。

1. 借助脚本生成火焰图对 perf.data 进行可视化分析。
2. 据[intel 官方文档](#)说明，vtune 工具支持对 perf.data 进行导入分析。
3. 使用一个可以将perf.data可视化的工具[hotspot](#)。

4.3 体系结构相关的程序性能剖析

4.3.1 cache 命中率

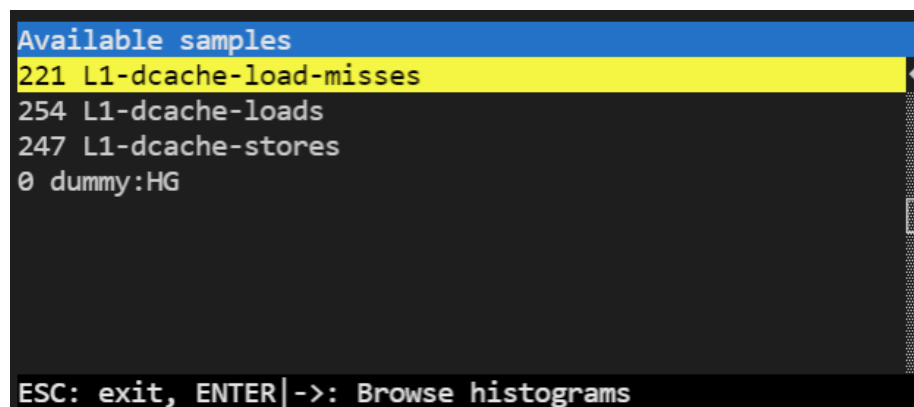
仍然以前文所述的数组列求和为例，在下文的示例程序中，函数col_major按列优先计算，而函数row_major按行优先计算。

使用某个编译器对代码进行编译，以 gcc 为例，为了在分析过程中保留更详细的符号和源代码信息，编译时使用调试信息选项-g。确保生成的可执行文件可以正常执行后，使用 perf 针对 cache 相关事件进行采样。依次输入指令如下

```
1 $ gcc column_sum.c -g -o column_sum
2 # perf record -e L1-dcache-load-misses,L1-dcache-loads,L1-dcache-stores -ag ./row_column
3 # perf report
```

如图 4.13所示，显示了所选择三个事件的采样数，并可依次按下 Enter 查看对应的直方图。

如图 4.14所示，显示了在所有采样的 L1-dcache-load-misses 事件中，各个函数的占比，在当前条件下col_major比row_major的 L1 数据缓存载入不命中率要高很多。继续按下 Enter，并选择 Annotate，可查看该函数具有源代码注释的反汇编代码，与各指令上的事件采样比率，由于硬件平台不同，可能产生完全不一样的反汇编代码，图 4.15展示了x86_64平台上的反汇编结果。此外，如果安装了可视化的 hotspot 工具，运行# hotspot perf.data，



```
Available samples
221 L1-dcache-load-misses
254 L1-dcache-loads
247 L1-dcache-stores
0 dummy:HG

ESC: exit, ENTER|->: Browse histograms
```

图 4.13: perf report 结果 1

可以更加清晰地展示火焰图、函数调用关系等。如图 4.16 所示, 为 L1-dcache-load-misses 事件的火焰图, 图中颜色深浅无明显含义, 关键看各个函数的宽度, “平原” 为该程序的瓶颈处。

影响实验结果的因素有很多, 在理解 cache 工作原理的基础上, 首先要清楚所用机器的各项参数, 在 Linux 上使用命令 `$ getconf -a | grep CACHE` 可以得到各级 cache 的大小 (SIZE 单位 Byte), 相联度 (ASSOC), 缓存行大小 (LINESIZE 单位 Byte)。改变计算规模, 或者在不同的机器上, 实验结果可能有很大差异。

4.3.2 超标量

与 VTune 一节同样, 比较两路链式累加算法和普通的链式算法的性能, 两路链式累加算法 IPC 应该明显优于普通的链式算法。下例程序对 4096 元素求和 500 次, perf 重复统计 instruction 和 cycles 共 100 次。

Listing 1: 平凡算法

```
$ perf stat -e instructions,cycles -r 100 ./liner

Performance counter stats for './liner' (100 runs):

    25,310,778      instructions          #    1.00   insn per cycle     ( +- 0.00% )
    25,327,056      cycles                               ( +- 0.05% )

    0.0058768 +- 0.0000205 seconds time elapsed ( +- 0.35% )
```

Samples: 221 of event 'L1-dcache-load-misses', Event count (approx.): 510398				
Children	Self	Command	Shared Object	Symbol
+ 84.03%	0.00%	row_column	libc-2.31.so	[.] __libc_start_main
+ 84.03%	0.00%	row_column	row_column	[.] main
+ 75.01%	70.66%	row_column	row_column	[.] col_major
+ 11.21%	0.00%	swapper	[kernel.kallsyms]	[k] secondary_startup_64_no_verify
+ 11.21%	0.00%	swapper	[kernel.kallsyms]	[k] cpu_startup_entry
+ 11.21%	0.00%	swapper	[kernel.kallsyms]	[k] do_idle
+ 10.79%	0.00%	swapper	[kernel.kallsyms]	[k] call_cpuidle
+ 10.79%	0.00%	swapper	[kernel.kallsyms]	[k] cpuidle_enter
+ 10.79%	0.00%	swapper	[kernel.kallsyms]	[k] cpuidle_enter_state
+ 10.22%	0.00%	swapper	[kernel.kallsyms]	[k] acpi_idle_enter
+ 9.02%	7.66%	row_column	row_column	[.] row_major
+ 8.97%	2.42%	swapper	[kernel.kallsyms]	[k] native_safe_halt
+ 7.68%	0.00%	swapper	[kernel.kallsyms]	[k] start_secondary
+ 7.59%	1.25%	swapper	[kernel.kallsyms]	[k] asm_sysvec_apic_timer_interrupt
+ 6.33%	0.00%	swapper	[kernel.kallsyms]	[k] sysvec_apic_timer_interrupt
+ 3.53%	0.00%	swapper	[kernel.kallsyms]	[k] x86_64_start_kernel
+ 3.53%	0.00%	swapper	[kernel.kallsyms]	[k] x86_64_start_reservations
+ 3.53%	0.00%	swapper	[kernel.kallsyms]	[k] start_kernel
+ 3.53%	0.00%	swapper	[kernel.kallsyms]	[k] arch_call_rest_init
+ 3.53%	0.00%	swapper	[kernel.kallsyms]	[k] rest_init
+ 3.17%	0.00%	row_column	[kernel.kallsyms]	[k] asm_exc_page_fault
+ 3.10%	0.00%	swapper	[kernel.kallsyms]	[k] irq_exit_rcu
Cannot load tips.txt file, please install perf!				

图 4.14: perf report 结果 2

Listing 2: 2 路展开累加

\$ perf stat -e instructions,cycles -r 100 ./2way				
Performance counter stats for './2way' (100 runs):				
22,235,765	instructions	#	1.71 insn per cycle	(+- 0.01%)
12,984,925	cycles			(+- 0.05%)
0.0030010 +- 0.0000124 seconds time elapsed (+- 0.41%)				

对比普通的链式算法和两路链式累加算法,指令数 25,310,778 与 22,235,765 相近,周期数 25,327,056 与 12,984,925 翻倍,IPC (CPI 倒数) 1.00 和 1.71 有较大差距。

这里也可使用 `perf record` 对函数详细剖析,分别收集 instruction 和 cycle 数计算 CPI。

