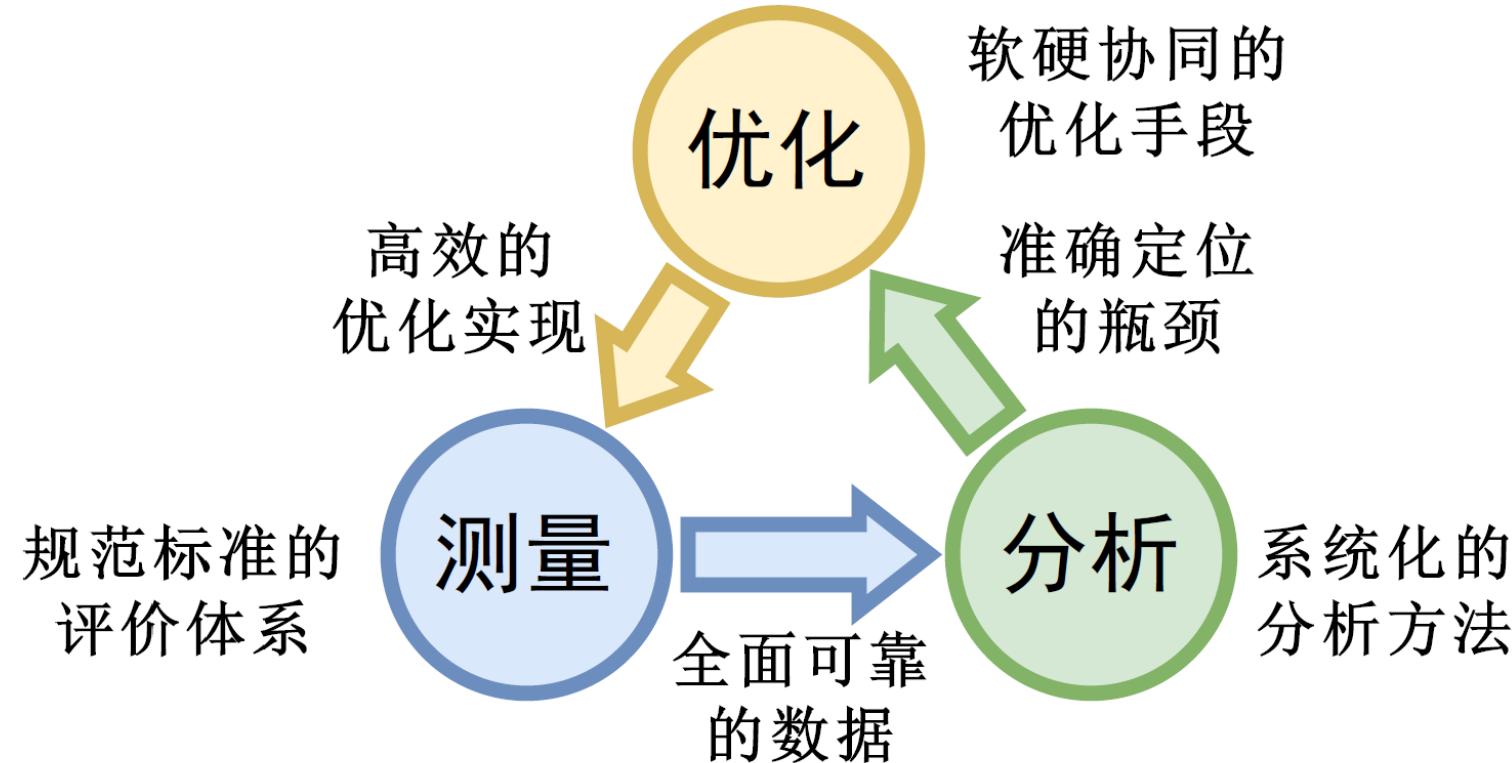


# 性能测量

刘通宇, 郭健美

2024年秋

# 数据驱动的系统优化方法



- 可靠的性能测量是进行准确性能分析的基础，也是评估性能优化效果的关键

# 大纲

- **关键问题：**如何对软件系统进行可靠的性能测量？

## 测量方法

- 外部测量
- 内部（插桩）测量
- 仿真测量

## 数据收集策略

- 计数型
- 采样型
- 追踪型

## 性能波动

- 系统静默

\* 其他：测量开销，测量误差

# 大纲

- **关键问题：**如何对软件系统进行可靠的性能测量？

## 测量方法

- 外部测量
- 内部（插桩）测量
- 仿真测量

## 数据收集策略

- 计数型
- 采样型
- 追踪型

## 性能波动

- 系统静默

# 测量方法

- 待测程序：  
判断指定区间素数个数
- 编译与运行

```
$ gcc -o count_prime count_prime.c
$ ./count prime 1 20000000
Found 1270607 prime(s) in interval
[1, 20000000].
```

在待测机器上，程序运行时间约为 10 s

代码: count\_prime.c

```
#include <stdio.h>
#include <stdlib.h>

int is_prime(int n) {
    if (n <= 1) return 0;
    if (n <= 3) return 1;
    if (n % 2 == 0 || n % 3 == 0) return 0;
    int i = 5;
    while (i * i <= n) {
        if (n % i == 0) return 0;
        i++;
    }
    return 1;
}

int count_prime(int start, int end) {
    int num, count = 0;
    for (num = start; num <= end; num++)
        if (is_prime(num) == 1) count++;
    return count;
}

int main(int argc, char *argv[]) {
    if (argc != 3) printf("This program should be called with 2 arguments.\n");
    else {
        int start = atoi(argv[1]);
        int end = atoi(argv[2]);
        int count = count_prime(start, end);
        printf("Found %d prime(s) in interval [%d, %d].\n", count, start, end);
    }
}
```

# 外部测量

- 外部测量 (external measurement) 是通过外部工具或系统级别的方法来测量程序的性能，通常不需要修改程序代码。
- 常用工具 (Linux系统)：
  - time ( /usr/bin/time )
  - Linux perf
  - Intel EMON
  - ...

# 案例：外部测量

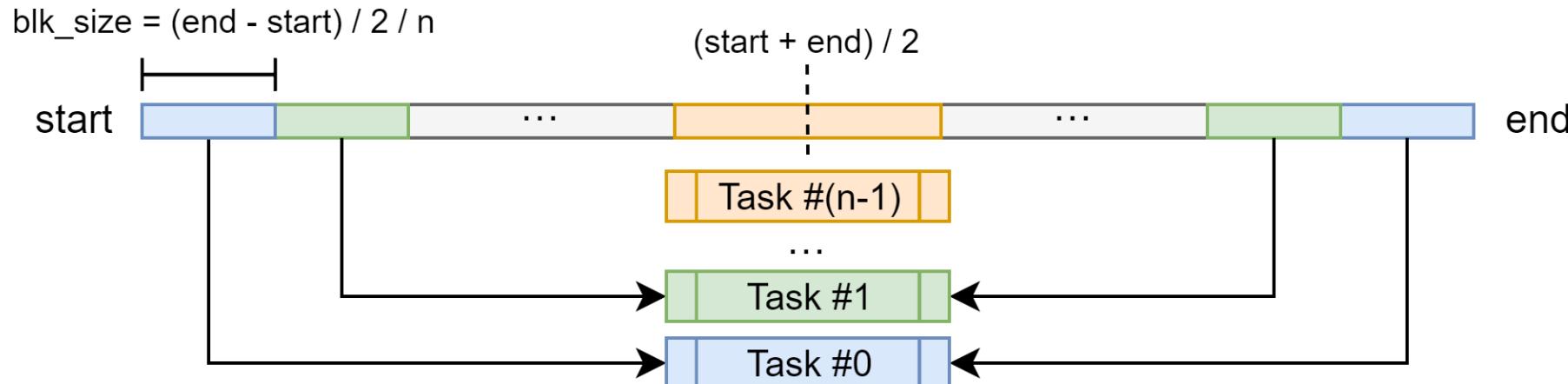
```
$ time ./count_prime 1 20000000
Found 1270607 prime(s) in interval [1, 20000000].
real    0m10.509s
user    0m10.505s
sys     0m0.004s
```

- ◆ **真实时间 (real time)** : 也称挂钟时间 (wall-clock time) , 指程序开始到结束真实世界里消逝的时间
- ◆ **CPU时间 (CPU time)** : 指程序占用CPU进行运算的时间:  
根据CPU的状态, 还以细分为:
  - 用户态时间
  - 内核态时间

- 思考: real 一定等于 user + sys 吗? user + sys 会不会大于 real?

# 案例：并行处理

- MPI (*Message Passing Interface*) 是一种进程间通信的编程接口，常用于并行计算领域
- 将“统计区间素数个数”的代码改写成并行程序



## 代码: count\_prime\_parallel.c

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
...
int main(int argc, char *argv[])
{
    if (argc != 3) printf("This program should be called with 2
arguments.\n");
    else {
        int my_rank, comm_sz, block_size, local_start_1, local_end_1,
            local_start_2, local_end_2, local_count, source,
total_count;

        int start = atoi(argv[1]);
        int end = atoi(argv[2]);

        MPI_Init(NULL, NULL);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
        MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

        block_size = (end - start) / (comm_sz * 2);

        local_start_1 = start + block_size * my_rank;
        local_end_1 = local_start_1 + block_size - 1;
        local_end_2 = end - block_size * my_rank;
        local_start_2 = local_end_2 - block_size + 1;

        if (my_rank == comm_sz - 1)
            local_count = count_prime(local_start_1, local_end_2);
        else {
            local_count = count_prime(local_start_1, local_end_1)
                + count_prime(local_start_2, local_end_2);
        }

        if (my_rank != 0)
            MPI_Send(&local_count, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        else {
            total_count = local_count;
            for (source = 1; source < comm_sz; source++) {
                MPI_Recv(&local_count, 1, MPI_INT, source, 0,
MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
                total_count += local_count;
            }
            printf("Found %d prime(s) in interval [%d, %d].\n",
total_count, start, end);
        }
        MPI_Finalize();
    }
    return 0;
}
```

编译

```
$ mpicc -o count_prime_parallel count_prime_parallel.c
```

运行

工作进程数量

```
$ mpirun -np <n> ./count_prime_parallel <start> <end>
```

# 真实时间与CPU时间

“判断指定区间素数个数” 程序

串行版本

```
$ time ./count_prime 1 20000000
Found 1270607 prime(s) in interval [1, 20000000].

real    0m10.509s
user    0m10.505s
sys     0m0.004s
```

并行版本 (启动 8 个工作进程)

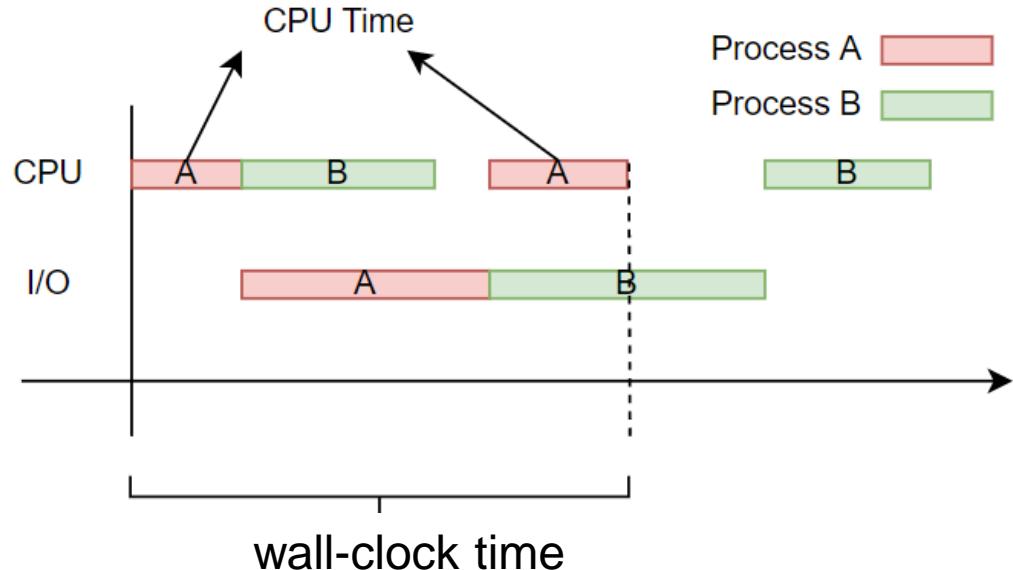
```
$ time mpirun -np 8 ./count_prime_parallel 1 20000000
Found 1270607 prime(s) in interval [1, 20000000].

real    0m1.573s
user    0m10.921s
sys     0m0.156s
```

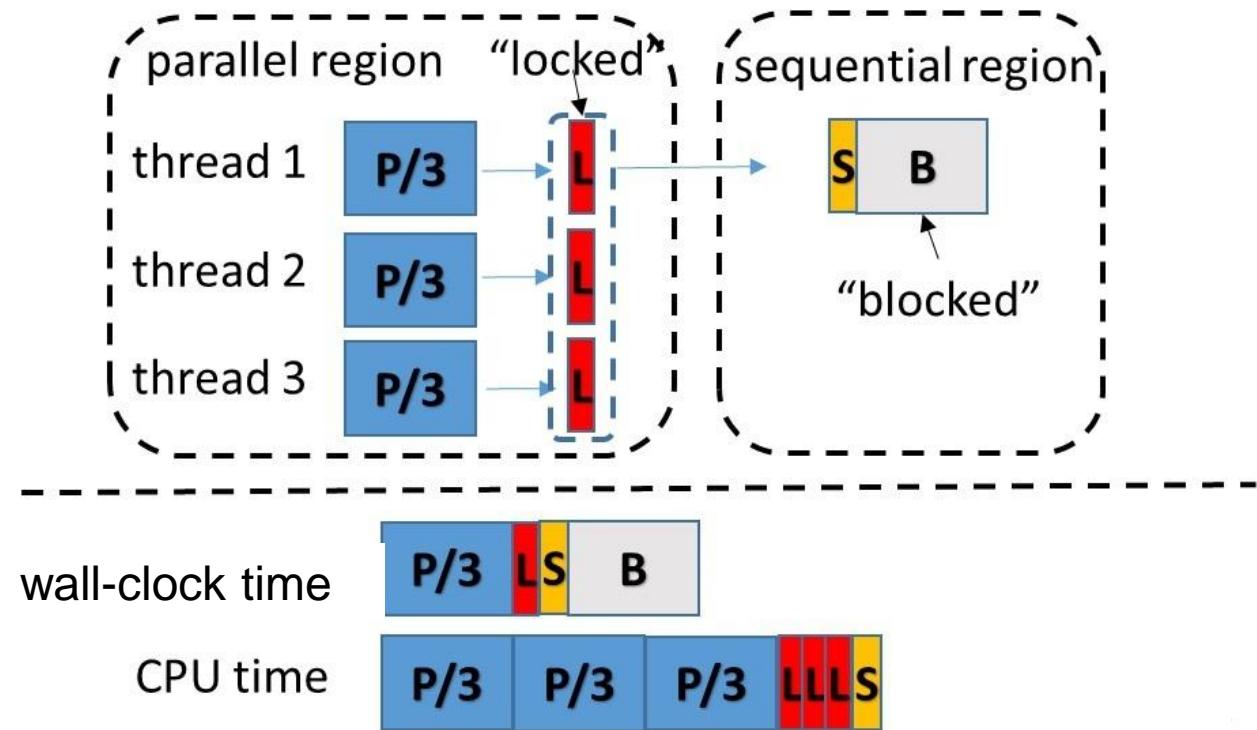
- 对于并行程序，该程序消耗的CPU时间相当于所有进程（线程）占用CPU进行运算的时间之和。

# 真实时间与CPU时间

例1：单处理器，多任务系统，单线程程序



例2：多处理器，并行程序



# 内部（插桩）测量

- 内部测量 (internal measurement) , 或叫插桩 (instrumentation) , 是通过修改程序的源代码或二进制文件，在程序中插入时间测量代码或性能分析工具来测量程序的性能。
- 适用范围：只对程序某一部分的性能感兴趣，而非整个程序。

# 间隔计时器 (interval timer)

- 在需测量程序代码段的开始和结尾各插入一个计时器（插桩点， instrumentation point），用两次计时的间隔来表示所测量程序的运行时间。
- 例：只关心 count\_prime() 函数的运行时间

代码: count\_prime\_gettime.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
...
int main(int argc, char *argv[])
{
    if (argc != 3)
        printf("This program should be called with 2 arguments.\n");
    else
    {
        int start = atoi(argv[1]);
        int end = atoi(argv[2]);
        struct timespec t_start, t_end;

        clock_gettime(CLOCK_MONOTONIC, &t_start);
        int count = count_prime(start, end);
        clock_gettime(CLOCK_MONOTONIC, &t_end);

        printf("Found %d prime(s) in interval [%d, %d].\n", count, start, end);

        double t_diff = (t_end.tv_sec - t_start.tv_sec)
                      + 1e-9 * (t_end.tv_nsec - t_start.tv_nsec);
        printf("time: %.9lf\n", t_diff);
    }
}
```

函数 clock\_gettime() 的静态库

clock\_gettime() 返回的结构体

```
struct timespec {  
    long tv_sec; /* seconds */  
    long tv_nsec; /* nanoseconds */  
}
```

被测代码

运行结果 time: 10.491397467

# rdtsc 机器指令

- x86 架构的处理器支持时间戳计数器 (Time-Stamp Counter, TSC)
  - 64位硬件计数器 (寄存器) , 可通过 rdtsc 机器指令在用户态直接读取计数值
  - 从系统启动开始, 以处理器基频 (base frequency) 计数

例: Model name: Intel(R) Xeon(R) Gold 6326 CPU @ **2.90GHz** ← 基频

使用嵌入汇编的方式在C语言代码中直接调用机器指令

```
static __inline__ unsigned long long rdtsc(void) {  
    unsigned hi, lo;  
    __asm__ __volatile__ ("rdtsc" : "=a"(lo), "=d"(hi));  
    return ( ((unsigned long long)lo) | (((unsigned long long)hi) << 32) );  
}
```

rdtsc 命令无操作数, 执行该命令后,  
高32位数据存放在 EDX 寄存器,  
低32位数据存放在 EAX 寄存器

# 计时器的选择

- ① `clock_gettime(CLOCK_MONOTONIC, ...)`

```
#include <time.h>
...
struct timespec t_start, t_end;
clock_gettime(CLOCK_MONOTONIC, &t_start);
// code to be measured
clock_gettime(CLOCK_MONOTONIC, &t_end);
```

`clock_gettime()` 返回的结构体

```
struct timespec {
    long tv_sec; /* seconds */
    long tv_nsec; /* nanoseconds */
}
```

## 运行结果

time: 10.491397467

- ② `gettimeofday()`

```
#include <sys/time.h>
...
struct timespec t_start, t_end;
gettimeofday(&t_start, NULL);
// code to be measured
gettimeofday(&t_end, NULL);
```

`gettimeofday()` 返回的结构体

```
struct timeval {
    long tv_sec; /* seconds */
    long tv_usec; /* microsecond */
}
```

time: 10.493744

- ③ `rdtsc` 机器指令

```
unsigned long long t_start, t_end;
t_start = rdtsc();
// code to be measured
t_end = rdtsc();
```

将 TSC 的计数值依据处理器基频转换为时间

```
double const BASE_FREQ = 2900000000;
double t_diff = (t_end - t_start) /
BASE_FREQ;
printf("time: %.9lf\n", t_diff);
```

time: 10.224389530

# 计时器的选择

Don't Use Lousy Timers!

- 推荐使用 ① `clock_gettime(CLOCK_MONOTONIC, ...)`
  - 时钟类型 `CLOCK_MONOTONIC` 表示该时钟是从系统启动开始计时，计数值单调不变（不会回退）
  - 能获得 ns 级别的时间戳
- 不推荐使用 ② `gettimeofday()` 或 ③ `rdtsc`
  - `gettimeofday()` 使用的时钟类型是 `CLOCK_REALTIME`，会随着时区改变而改变，并且用户可以手动修改
  - `gettimeofday()` 只能获得  $\mu\text{s}$  级别的时间戳
  - TSC 计数器的值在处理器不同核心上计数值是不同的
  - TSC 计数差值到真实时间的换算有时并不容易
  - TSC 是硬件计数器，可能会溢出

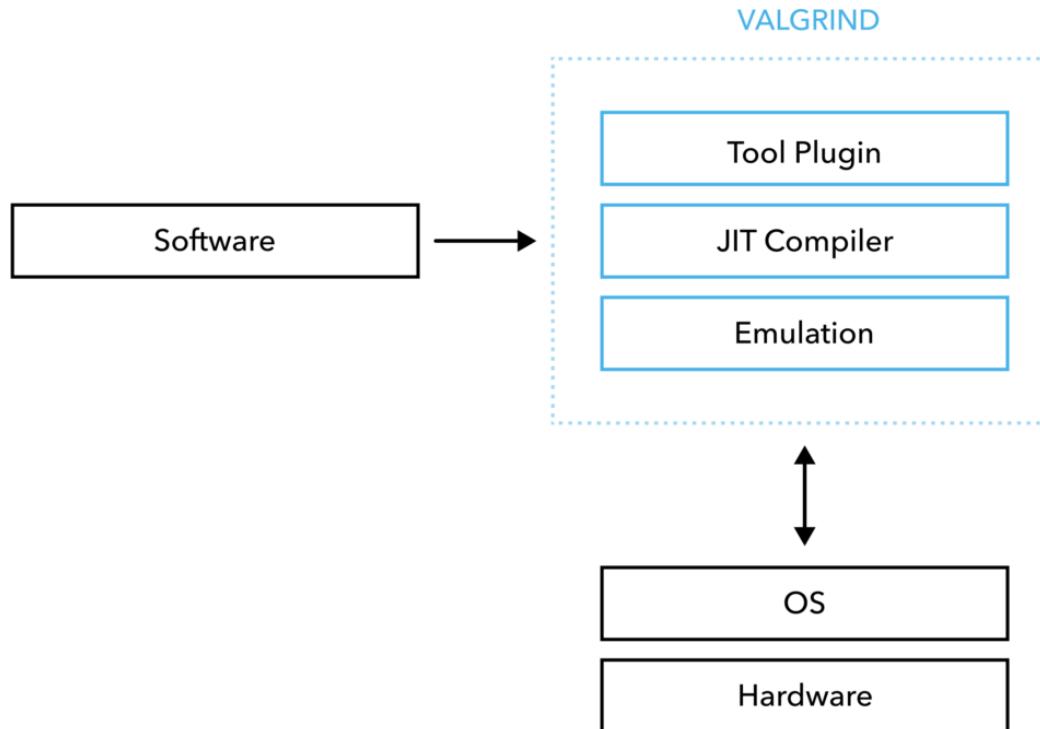
# 仿真测量

- **仿真测量** (*simulation measurement*) 是使用计算机模型或仿真工具来模拟程序、系统或硬件在不同条件下的行为，以评估其性能和稳定性等行为。
- 与实际运行程序不同，仿真测量是在虚拟环境中进行的，通过模拟真实世界的情况来预测程序性能或评估系统的行为。
- 常用工具：Valgrind

# 仿真测量

Valgrind：用于软件内存泄漏检测、性能分析的一套工具

本质：使用即时编译（Just-In-Time, JIT）技术的虚拟机



- 包含 cachegrind, memgrind 等
- 动态二进制翻译，获得平台无关的中间表示
- 从宿主机获取架构信息，进行仿真，收集性能数据
- 仿真结束后，宿主机执行程序

# 案例：仿真测量

```
$ valgrind --tool=cachegrind --time-stamp=yes ./prime 1 20000000
==00:00:00:00.000 48807== Cachegrind, a cache and branch-prediction profiler
==00:00:00:00.000 48807== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==00:00:00:00.000 48807== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==00:00:00:00.000 48807== Command: ./prime 1 20000000
==00:00:00:00.000 48807== --00:00:00:00.000 48807-- warning: L3 cache found, using its data for the LL simulation.
Found 1270607 prime(s) in interval [1, 20000000].
==00:00:02:16.859 48807==
==00:00:02:16.861 48807== I refs:      51,696,993,359
==00:00:02:16.861 48807== I1 misses:        1,340
==00:00:02:16.861 48807== LLi misses:       1,317
==00:00:02:16.861 48807== I1 miss rate:    0.00%
==00:00:02:16.861 48807== LLi miss rate:   0.00%
==00:00:02:16.861 48807==
==00:00:02:16.861 48807== D refs:      23,456,869,742  (23,390,189,673 rd + 66,680,069 wr)
==00:00:02:16.861 48807== D1 misses:      2,164  ( 1,539 rd + 625 wr)
==00:00:02:16.861 48807== LLd misses:     1,910  ( 1,322 rd + 588 wr)
==00:00:02:16.861 48807== D1 miss rate:  0.0% ( 0.0% + 0.0% )
==00:00:02:16.861 48807== LLd miss rate: 0.0% ( 0.0% + 0.0% )
==00:00:02:16.861 48807==
==00:00:02:16.861 48807== LL refs:       3,504  ( 2,879 rd + 625 wr)
==00:00:02:16.861 48807== LL misses:     3,227  ( 2,639 rd + 588 wr)
==00:00:02:16.861 48807== LL miss rate: 0.0% ( 0.0% + 0.0% )
```

# 大纲

- **关键问题：**如何对软件系统进行可靠的性能测量？

## 测量方法

- 外部测量
- 内部（插桩）测量
- 仿真测量

## 数据收集策略

- 计数型
- 采样型
- 追踪型

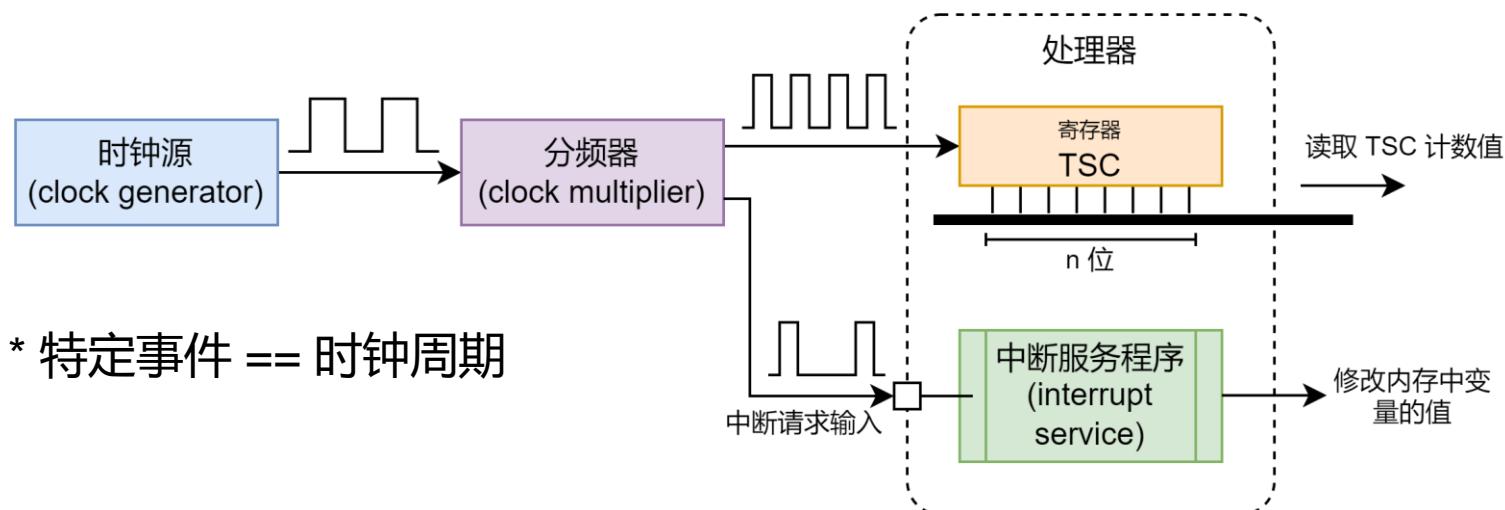
## 性能波动

- 系统静默

以外部测量为例

# 计数型

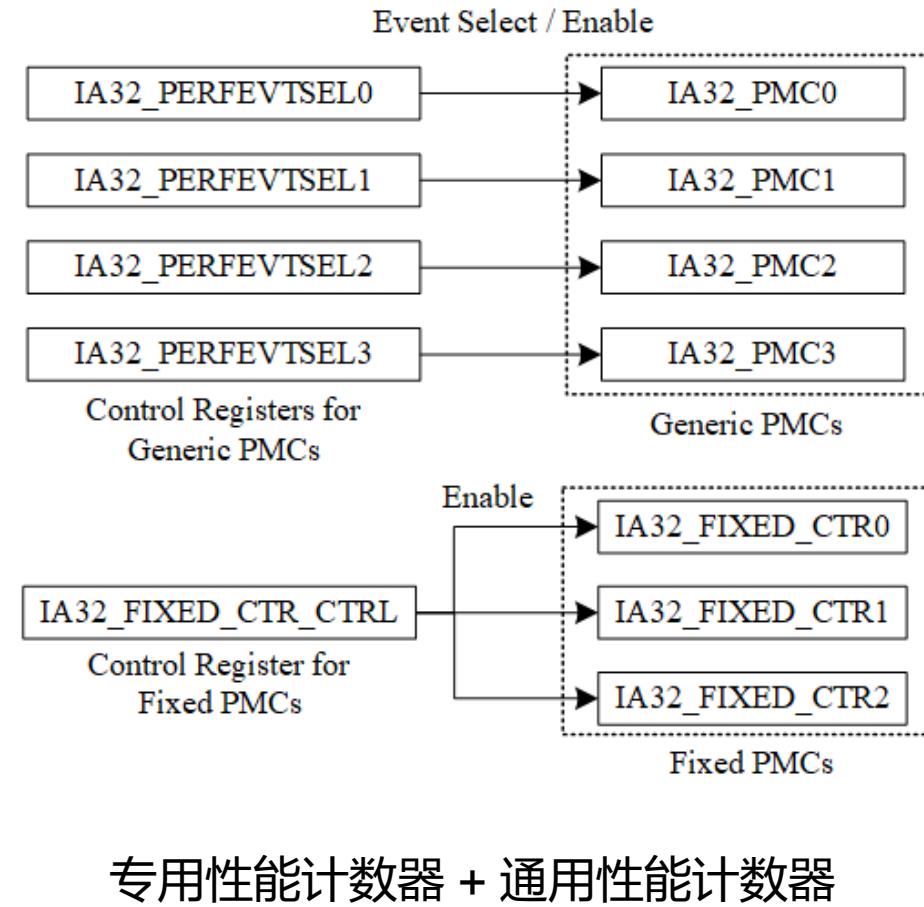
- 计数 (counting) 型策略是用**计数器** (counter) 对特定事件发生的绝对次数进行收集。
- 计数器
  - 软件计数器：操作系统内核或用户进程在内存中维护的变量
  - 硬件计数器：计算机硬件内真实存在的寄存器



# 计数型

- 计数器并非只能监控时钟周期，还能监测其他性能事件
  - 软件计数器
    - 缺页中断
    - 进程上下文切换
    - 处理器迁移次数 等
  - 硬件计数器
    - 指令数
    - 缓存 (L1/L2/L3) 未命中
    - TLB 未命中 等

性能监测单元  
(Performance Monitoring Unit, PMU)  
*Intel CascadeLake Processor*



# 案例：计数型

工具：Linux perf，使用 stat 选项对待测程序以计数方式收集性能数据

- 目标：评估指令流水线效率
- 指标：IPC（每时钟周期的平均指令数）
- 测量事件：
  - 时钟周期 (cycles)
  - 指令 (instructions)

```
$ perf stat -e cycles,instructions ./prime 1 20000000
Found 1270607 prime(s) in interval [1, 20000000].

Performance counter stats for './prime 1 20000000':
    34,535,493,614      cycles
    51,709,581,852      instructions          #    1.50  insn per cycle

    10.504566736 seconds time elapsed

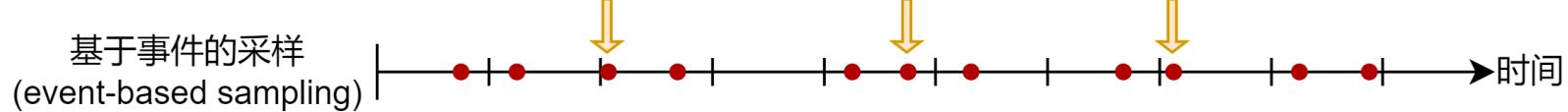
    10.504132000 seconds user
    0.000000000 seconds sys
```

# 采样型

- **采样 (sampling)** 型策略是在程序运行过程中，对系统有规律地进行采样，采样时刻将收集系统的状态信息形成若干样本，最终形成性能数据的统计信息。

- 特点：开销可控

- 采样策略：



● 发生特定事件      ↓ 采样

# 案例：采样型

工具：Linux perf，使用 record 选项对待测程序以采样方式收集性能数据

含义：当 cpu-clock 事件（操作系统内核维护的统计CPU时间的事件，单位为ns）发生一定次数时触发采样，其次数会使得采样频率尽可能地接近 99 Hz

```
$ perf record -F 99 -e cpu-clock ./prime 1 20000000
Found 1270607 prime(s) in interval [1, 20000000].
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.058 MB perf.data (1041 samples) ]
```

口 思考：为什么采样频率要设为 99 Hz，而不是 100 Hz 这种较为规整的数字？

生成名为 perf.data 的二进制文件

使用 report 选项解析样本

```
$ perf report
```

```
Samples: 1K of event 'cpu-clock', Event count (approx.): 10515151410
Overhead  Command  Shared Object  Symbol
 99.52%  prime      prime          [.]
           prime      prime          [.]
  0.48%  prime      prime          [.]
           prime      prime          [.]
```

👉 热点 (hotspot) 函数

# 案例：采样型

使用 perf report 的“Annotate”功能观察落在该函数内部的样本分布情况，识别**热点指令**；但可能存在**滑移（skid）**问题，即标记的热点指令与真实的热点指令存在一个小小的滑移误差。

```
Samples: 1K of event 'cpu-clock', 99 Hz, Event count (approx.): 10515151410
is_prime  /home/tongyu/project/ssoTextbook/chpt.2/prime [Percent: local period]
Percent   Address           Instruction
          mov    %edx,%eax
          sub    %ecx,%eax
          test   %eax,%eax
          ↓ jne   5c
          0.10   55:   mov    $0x0,%eax
          ↓ jmp   8d
          0.10   5c:   movl   $0x5,-0x4(%rbp)
          ↓ jmp   7d
          0.10   65:   mov    -0x14(%rbp),%eax
          cltd
          idivl  -0x4(%rbp)
          84.36   mov    %edx,%eax
          test   %eax,%eax
          ↓ jne   79
          0.29   mov    $0x0,%eax
          0.48   ↓ jmp   8d
          13.51  79:   addl   $0x1,-0x4(%rbp)
          7d:   mov    -0x4(%rbp),%eax
          imul   %eax,%eax
          cmp    %eax,-0x14(%rbp)
          ↑ jge   65
```

# 追踪型

- **追踪 (tracing) 型**策略记录了程序的执行过程中的每个事件和函数调用，提供了详细的时间序列信息。
- **特点：**
  - 事件驱动，需要记录特定事件发生的时间以及其他用户关心的状态信息（计数型仅需要更新计数器）
  - 如果事件发生较为频繁，追踪将造成显著开销
- **工具：**
  - Linux perf (probe 选项)
  - strace
  - bpftrace

# 案例：追踪型

工具：strace，跟踪程序运行时系统调用事件的工具

- 特定事件：Linux内核系统调用
- 记录信息：系统调用发生的时间，系统调用名与参数，返回值

```
$ strace -tt ./prime 1 20000000
00:36:11.446959 execve("./prime", ["./prime", "1", "20000000"], 0x7ffea13bb9b8 /* 42 vars */) = 0
00:36:11.447274 brk(NULL)                  = 0x55d8c23b5000
00:36:11.447314 arch_prctl(0x3001 /* ARCH_??? */, 0x7ffe322e350) = -1 EINVAL (Invalid argument)
00:36:11.447435 mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7d79f50000
00:36:11.447469 access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
...
00:36:21.953784 brk(NULL)                  = 0x55d8c23b5000
00:36:21.953835 brk(0x55d8c23d6000)      = 0x55d8c23d6000
00:36:21.953900 write(1, "Found 1270607 prime(s) in interv...", 50) = 50
00:36:21.953972 exit_group(0)            = ?
00:36:21.954085 +++ exited with 0 +++
```

# 大纲

- **关键问题：**如何对软件系统进行可靠的性能测量？

## 测量方法

- 外部测量
- 内部（插桩）测量
- 仿真测量

## 数据收集策略

- 计数型
- 采样型
- 追踪型

## 性能波动

- 系统静默

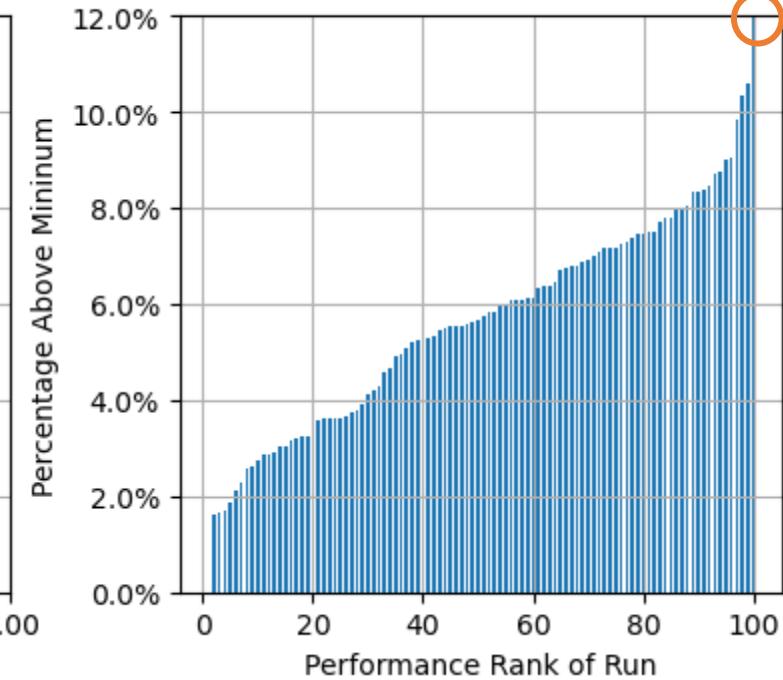
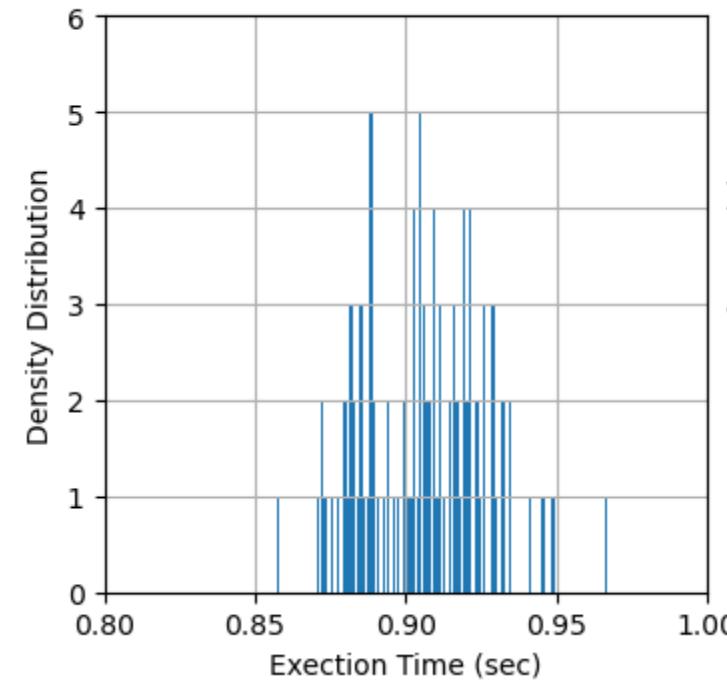
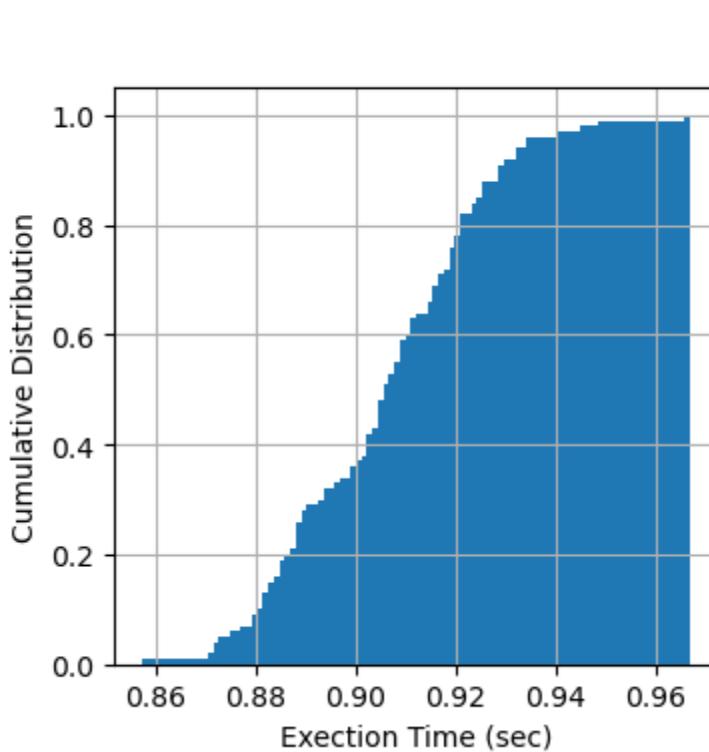
# 性能波动

- 同一套软件在同一台硬件设备上多次重复运行时，每次性能测量的结果可能不同，从而形成性能波动（performance variation）。
- 性能波动是计算机系统不可回避的一个特性，它对于系统可复现性（reproducibility）研究和定量比较不同系统的性能具有重要影响。

# 案例：性能波动

- 启动 16 个工作进程，并行统计区间内素数个数
- 令程序运行 100 次，每一次的运行时间为 1 s

最大的性能差异有 12%



# 性能波动的来源

- 软件层面
  - 后台任务
  - 中断
  - 进程/线程运行时调度
  - CPU或内存绑定
  - 多租户 (multitenancy)
  - 代码或数据对齐 等
- 硬件层面
  - 超线程 (HyperThreading, HT)
  - 睿频 (Turbo Boost)
  - 动态电压和频率调节 (Dynamic Voltage and Frequency Scaling, DVFS)
  - 网络流量 等

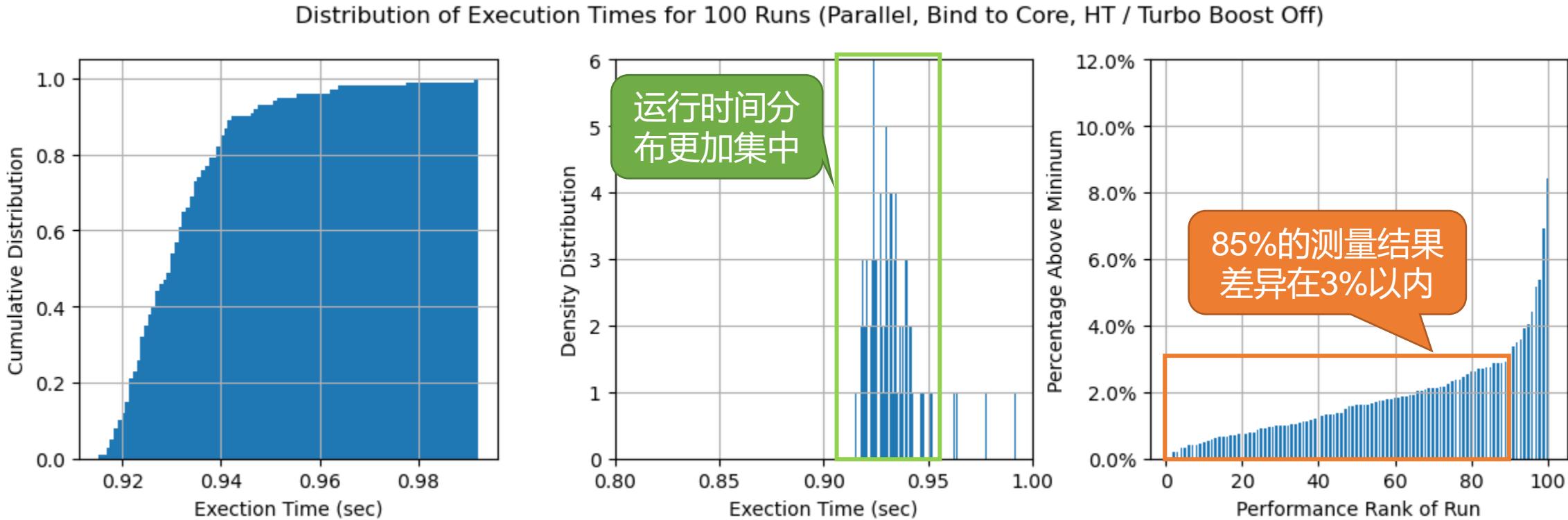
思考：如何控制性能波动？

# 控制性能波动 → 系统静默

- 软件层面
    - 后台任务
    - 中断
    - 进程/线程运行时调度
    - CPU或内存绑定
    - 多租户 (multitenancy)
    - 代码或数据对齐 等
  - 硬件层面
    - 超线程 (HyperThreading, HT)
    - 睿频 (Turbo Boost)
    - 动态电压和频率调节 (Dynamic Voltage and Frequency Scaling, DVFS)
    - 网络流量 等
- 
- 系统静默 (quiescing)
    - 软件层面
      - 关闭后台任务
      - 绑定工作进程/线程 (taskset或numactl) 等
    - 硬件层面
      - 关闭 HT / Turbo Boost / DVFS 等

# 案例：控制性能波动

- 软件层面：将工作线程绑定到固定的处理器上
- 硬件层面：关闭超线程与睿频



# 小结

- 可靠的性能测量是性能工程的基础，需要结合具体的性能测量需求，包括测量开销、测量精度、需要测量的性能数据等，选择合适的测量方法与性能数据收集方法。
- 性能测量需要考虑到性能波动，事实上我们几乎不可能保证得到完全稳定的性能测量结果，但是我们可以明确性能波动的主要来源，并尽可能地控制它们。
- 要特别关注性能数据的质量，并且正确、科学地报告性能数据，这一点需要牢记于心。