处理器性能波动检测的计时方法及评价指标

廖秋承 左思成 王一超 林新华
（上海交通大学高性能计算中心 上海 200240）

摘 要 超级计算机中的性能波动通常表现为软件在同一硬件上运行得忽快忽慢，或在配置相同的硬件上运行得快慢不一.在多种性能波动来源中，处理器性能波动隐蔽性强且危害巨大，可导致超级计算机整机性能急剧下降.然而，当前处理器性能波动研究面临两大难题.首先，现有工具难以检测微小的性能波动.为了准确检测纳秒级的处理器性能波动，计时方法需要具有很高的精度和灵敏度.然而，现有工具在真实应用中用于计时测量时，计时结果波动可达数万拍，难以检测处理器性能波动.其次，现有方法难以客观评价不同工具的性能波动检测能力，缺乏量化评价指标.一次性能波动检测包含大量计时结果，其分布可能受性能波动和计时波动的共同影响.然而，现有方法无法评价这些测量结果是否真实反映了性能波动的特征.为解决第一个问题，本文对PAPI在不同缓存状态下的计时波动进行了测量和原因分析.随后，基于x86和Armv8指令集的内存屏障和序列化指令，设计了序列化屏障计时方法，用以抑制计时波动.为解决第二个问题，本研究对计时波动进行建模，首次提出了跨平台的计时方法精度和灵敏度指标及评价方法，定量评估了计时方法对微小时间波动的测量能力，为性能波动的检测和判定提供了依据.实验表明，在英特尔Xeon 6248和华为鲲鹏920-6426处理器上，与PAPI相比，序列化屏障计时方法的精度提高了2.2～30.2倍，灵敏度提高了1.9～44.8倍，并且能够检测到纳秒级别的性能波动.

关键词 高性能计算；处理器微架构；性能波动；性能分析；性能评测
中图法分类号 TP301 **DOI号** 10.11897/SP.J.1016.2024.00456

Timing Method and Evaluation Metrics for CPU Performance Variation Detections

LIAO Qiu-Cheng ZUO Si-Cheng WANG Yi-Chao LIN Xin-Hua
（*Center for High Performance Computing*, *Shanghai Jiao Tong University*, *Shanghai* 200240）

Abstract Performance variation is characterized by inconsistencies in run times on the same hardware or by periods of unaligned performance on identical hardwares. The CPU performance variation is one of the most harmful and insidious causes of performance degradation. Even a tiny variation can negatively affect the overall performance of a supercomputer. CPU performance variation detections currently face two challenges. First, identifying tiny processor performance variation is difficult with existing profiling tools like PAPI. The magnitude of processor performance variations can be as low as the nanosecond level. To accurately detect such variations, timing methods need to have high precision and sensitivity. However, researchers have found that tools like PAPI and LIKWID, when used for timing measurements in real applications, have large overheads and fluctuations that can reach tens of thousands of cycles, making it difficult to capture nanosecond-level run time changes. Second, existing methods struggle to objectively evaluate the performance variation detection capabilities of different tools.

收稿日期：2023-02-16；在线发布日期：2023-12-07. 本课题得到国家自然科学基金（62072300）资助.**廖秋承**，学士，助理工程师，中国计算机学会（CCF）会员，主要研究领域为高性能计算. E-mail：keyliao@sjtu.edu.cn. **左思成**，硕士，主要研究领域为高性能计算. **王一超**，硕士，高级工程师，中国计算机学会（CCF）高级会员，主要研究领域为高性能计算. **林新华**（通信作者），博士，高级工程师，中国计算机学会（CCF）杰出会员，主要研究领域为高性能计算. E-mail：james@sjtu.edu.cn.

A single performance variation detection consists of thousands of timing results. The distribution characteristics of these measurements include variations in the runtime of the tested code, fluctuations in the overhead of the timing method itself, and the impact of the timing operation on the tested code. However, current methods cannot determine whether the timing results truly reflect the distribution of the code's runtime. To address the first problem, this study first focused on PAPI, the most commonly used performance measurement tool at present. By simulating the cache environment of real applications, we measured and analyzed PAPI's timing fluctuations under different cache states for the first time. The experimental results showed that when measuring the run time of a computation process that does not change, PAPI's measurements exhibited significant long-tail deviations. Combining performance counter analysis, the main causes of PAPI's timing fluctuations included timing overhead, operating system noise, out-of-order execution, and cache misses. Subsequently, this study designed a serialized barrier timing method based on the memory barrier and serialized instructions of the x86 and Armv8 instruction sets, which suppressed timing fluctuations. In comparative experiments, the amplitude of timing fluctuations of the serialized barrier timing method was significantly lower than that of PAPI. To address the second problem, this study combined experiments and modeling to perform qualitative and quantitative analyses of the sources of instability in timing fluctuations and their impact on measurement values. For the first time, this paper proposed cross-platform precision and sensitivity indicators for timing methods, along with evaluation methods aimed at detecting processor performance variation. This paper suggests that in performance variation detection, the shorter the time that can be accurately measured, the higher the precision; the smaller the amplitude of performance variation that can be accurately distinguished, the higher the sensitivity. The precision and sensitivity indicators quantitatively evaluated the timing methods' ability to measure minute time fluctuations, thereby providing a basis for the detection and determination of performance variation. According to our evaluations, on the Intel Xeon 6248 and Huawei Kunpeng 920-6426 processors, compared to PAPI, the serialized barrier timing method was 2.2～30.2 times more precise and 1.9～44.8 times more sensitive, and is able to detect nanosecond-level performance variation.

Keywords high performance computing; microarchitecture; performance variation; performance analysis; performance evaluation

1 引 言

性能波动是指软件在同一硬件上运行得忽快忽慢，或在配置相同的硬件上运行得快慢不一. 超级计算机软硬件结构复杂，可引发性能波动的来源众多，如操作系统[1-2]、存储系统[3]、高速互连网络[2,4-5]和处理器[6-7]等. 其中，处理器性能波动隐蔽性强且危害巨大. 微观上，多核处理器的性能波动无法避免，导致指令执行时延存在不确定性[8-9]. 宏观上，随着超算并行规模增加，不同来源的性能波动在经过耦合放大后，可能会导致超算整机性能大幅下降[1-2,10]. 以美国的Stampede-2超算(2017年11月TOP500排名第12)为例，在装机测试时发现Linpack效率仅为45.7%，远低于70%的平均水平. 在耗费8个月进行仔细排查后，才最终将性能波动源定位到Intel处理器缓存设计缺陷导致的内存流量波动[10]. 在进行针对性修复后，Stampede-2超算的整机Linpack性能提升了2.3PFLOPS.

检测性能波动是研究处理器性能波动的前提，关键要解决两个问题：第一，现有工具难以检测微小的性能波动. 处理器性能波动的幅度可低至纳秒级别[9,11]，为了准确检测到这种波动，计时方法需要具有很高的精度和灵敏度. 在常规性能检测中，若被测循环耗时太短，计时误差太大，人们通常采用增加循环次数，取平均数等方法来获得稳定的测量时

间.然而,根据中心极限定律,这类数据处理方法会使数据分布趋向正态分布,掩盖微小的性能异常[12].因此,为了检测处理器性能波动,计时方法应当精度高,稳定性好,能准确检测出被测代码运行时间的微小变化.然而,Röhl等人[13]发现PAPI[14]和LIKWID[15]等工具用于真实应用的计时测量时,自身开销大,误差范围可达数万拍(cycle),难以捕捉到纳秒级性能异常.

第二,现有方法难以客观评价不同工具的性能波动检测能力.一次性能波动检测,由成千上万个计时结果组成.这些测量值的分布特征同时包含了被测代码运行时间的波动、计时方法自身开销的波动和计时操作对被测代码产生的影响.然而,现有方法无法判断计时结果是否真实反映了被测代码运行时间分布.当前研究[12,16-17]在评价计时准确性时,主要关注计时开销占程序运行时间的比例.这不仅忽略了计时方法本身存在的波动,还缺乏对检测精度和灵敏度的量化依据,故难以客观评价不同工具的性能波动检测能力.

为解决问题一,本研究首先以目前最常用的性能测量工具PAPI为研究对象,模拟真实应用的缓存环境,首次对PAPI在不同缓存状态下的计时波动进行了测量和原因分析.实验结果显示,在测量运行时间不变的计算过程时,PAPI的测量结果出现了显著的长尾偏差.结合性能计数器分析,引起PAPI计时波动的主要原因包括计时开销、操作系统噪声(OS噪声)、乱序执行和缓存不命中.随后,本研究基于x86和Armv8指令集的内存屏障和序列化指令,设计了序列化屏障计时方法,对计时波动进行了抑制.在对比实验中,序列化屏障计时方法的计时波动的幅度显著低于PAPI.

为解决问题二,本研究结合实验和建模,对计时波动的不稳定性来源和计时波动对测量值的影响进行了定性和定量分析.面向处理器性能波动检测,首次提出了跨平台的计时方法精度和灵敏度指标及评价方法.本文提出,在性能波动检测中,能准确测量的时间越短,则精度越高;能准确分辨的性能波动幅度越小,则灵敏度越高.精度和灵敏度指标定量评价了计时方法对微小时间波动的测量能力,从而为性能波动的检测和判定提供依据.

实验结果显示,在不同处理器和不同缓存冲刷大小下,序列化屏障计时方法精度和灵敏度指标均优于PAPI计时方法.以墙钟时间为例,当内存读写位于L1、L2和L3缓存时,序列化屏障计时方法在Intel Xeon 6248处理器上的精度分别为162 ns、34 ns和85 ns(PAPI为532 ns、490 ns和2563 ns),灵敏度分别为3 ns、3 ns、4 ns(PAPI为28 ns、28 ns和124 ns);在鲲鹏920-6426上,精度分别为150 ns、390 ns和610 ns(PAPI为1560 ns、2570 ns和9470 ns),灵敏度为10 ns、10 ns和13 ns(PAPI为66 ns、119 ns、582 ns).上述结果说明,序列化屏障计时拥有在两个实验平台上检测纳秒级时间波动的能力.

本文主要有以下三个贡献:

(1)通过实验发现PAPI的计时波动过大,导致结果准确性欠佳,难以用于检测处理器性能波动,并分析了PAPI计时波动产生的原因(详见第3节).

(2)基于内存屏障和序列化指令,针对x86和Armv8架构的处理器,提出了可用于检测处理器性能波动的序列化屏障计时方法(详见第4节).

(3)构建计时波动的解析模型,并据此提出评价处理器性能波动检测精度和灵敏度指标(详见第5节),并通过实验验证了有效性(详见第6节).

2 研究背景与动机

2.1 可用于处理器性能波动检测的计时工具

处理器性能检测工具包括侵入式和非侵入式两类.侵入式工具主要包括PAPI[14]、LIKWID[15]、Score-P[18]、Vampir[19]和Scalasca[20]等.这类工具需要在源代码的特定位置("桩点")添加检测函数("插桩"),当程序执行到桩点时,读取时间戳.这类工具无需额外进程,对被测程序影响较小.非侵入式工具主要包括Linux Perf[21]、Intel Vtune[22]和HPCToolKit[23]等.它们会额外启动采样进程,干扰被测程序运行,不适用于处理器性能波动检测.

在上述工具中,最常用于微架构研究的性能检测工具是PAPI和LIKWID,其中,PAPI不仅可以单独使用,还被Score-P、Scalasca和Vampir等工具所使用,用于计时和计数器读取.PAPI使用*PAPI_get_real_nsec*和*PAPI_read*函数,在程序内获得计时结果.相比之下,LIKWID在插桩后,不仅需要借助*LIKWID_MARKER_START*、*LIKWID_MARKER_END*和*LIKWID_MARKER_GET*三个预定义宏来计时和获取结果,还需要额外运行*likwid-perfctr*守护进程来进行周期性采样,产生了大量额外开销.根据Röhl等[13]人的研究,在干净的缓存下,LIKWID的*LIKWID_MARKER_START*和*LIKWID_MARKER_END*宏开销为约4000拍,

而PAPI的*PAPI_read*函数在多种处理器下都可实现数百拍的额外开销[24].

综上,本文以PAPI为主要分析对象,对计时波动开展研究,并用于与序列化屏障计时方法进行比较. PAPI可以在x86和Armv8指令集下,读取两种时钟数据:墙钟数据和CPU时钟数据. 前者是真实世界时间,以秒为单位;后者是处理器时间,以时钟周期(cycle)或"拍"为单位,本文按中文习惯以"拍"为单位.

PAPI读取这两种计时数据的过程如下:

(1)墙钟数据:PAPI支持多个墙钟计时函数,其中精度最高且最常用的是*PAPI_get_real_nsec*. 该函数多次跳转后,最终由系统调用*sys_clock_gettime*在内核态下读取纳秒精度的计时数据;

(2)CPU时钟数据:最常用的库函数为*PAPI_read*. 它经过多层调用,最终通过读取硬件性能计数器得到CPU时钟数据. 在早期版本的Linux内核上,PAPI读取硬件计数器需通过系统调用*perf_event_open*,开销高达数千纳秒. 对x86 CPU,在版本4.13之后的linux内核上,PAPI支持使用*rdpmc*指令直接读取硬件性能计数器,避免使用系统调用,使读取开销下降90%[25]. 但是,对Arm CPU,PAPI尚不支持使用汇编读取硬件计数器,仍需使用高开销的*perf_event_open*系统调用.

2.2　研究动机

目前,检测微小的处理器性能波动面临两个问题:(1)现有工具难以检测微小的处理器性能波动;(2)现有方法难以客观评价不同工具检测处理器性能波动的能力.

现有工具的计时波动已被发现多年[13,26]. 以PAPI、LIKWID为代表的底层检测库,在脏缓存的影响下,其计时开销的波动范围从数百拍至数万拍不等,使人们很难判断测量值的波动是来自处理器性能波动还是计时波动. 部分学者在研究中使用内嵌汇编,使用内存屏障,或单独编写微型评测程序,以应对计时波动[10,27]. 然而,这些方法无法应用于不同指令集的处理器,且读取的计数器种类均不一致,均只能满足各自研究的需求.

此外,在性能波动检测能力的评价上,现有方法也存在缺陷. 目前,Hoefler等人提出了评价性能测量准确性的方法[12],包含两步:(1)测量计时开销在被测代码运行时间中的占比是否足够小. (2)确认计时器每跳(tick)所经过的时间是否足够短. Hoefler等人建议在进行性能测量时,计时开销占比应小于5%,测量值应至少是时钟每跳的10倍[12]. 然而,这一方法仅为半定量方法,给出的精度和灵敏度参考过于笼统,未考虑到CPU负载的复杂性. 此外,这一方法未经过数学论证,未考虑到计时与被测代码的相互作用,也未考虑到计时开销受到脏缓存等性能波动源的影响,发生计时波动的情况. 因此,其难以用于量化评价计时方法在性能波动检测中的精度和灵敏度.

本研究观察了PAPI的计时波动,并分析了其原因(详见第3节). 随后,利用x86和Armv8指令集的特性,实现了物理时钟和墙上时钟的序列化屏障计时方法(详见第4节). 为了进一步量化评价计时方法的性能波动检测能力,本研究首次对计时波动的产生和影响进行了定性分析和数学建模,并根据模型推导出计时下限和波动分度,作为精度和灵敏度指标(详见第5节). 最后,在多种缓存冲刷大小下,定量对比了Intel Xeon 6248和鲲鹏920-6426处理器平台上序列化屏障计时和PAPI计时方法的精度和灵敏度(详见第6节).

3　PAPI计时波动的危害及原因分析

3.1　PAPI的计时波动现象及其危害

使用如图1所示的程序,检测PAPI的计时波动. 实验平台使用Intel Xeon 6248处理器,运行Linux操作系统,具体配置见表4. 为避免CPU变频,使用*cpupower*命令将主频固定在2.5 GHz,并使用*taskset*命令将代码绑定到单个核心上运行.

如图1,被测代码块是一个K次*for*循环,循环体内是K条存在数据依赖的*ADD*指令. 使用*PAPI_get_real_nsec*和*PAPI_read*分别读取墙钟和CPU时钟. *ADD*指令调用CPU加法器,耗时为1拍,硬件逻辑简单,无内存操作,可视作无性能波动. 被测代码块由K条*ADD*和分支指令(*JMP*)组成. 由于*JMP*可使被测代码的耗时不稳定,因此考虑使用循环展开来减少*JMP*. 但过度的循环展开会触发Icache Miss和TLB Miss. 考虑到一条*ADD*为4字节,且实验环境中OS页大小为4 KiB,L1指令缓存为32 KiB,故展开次数设置为256,这样既能减少*JMP*,又能避免Icache Miss和TLB Miss.

测试程序的缓存冲刷代码(第14行)用于模拟真实应用程序运行时的处理器缓存状态. 变量*ARRLEN*控制缓存冲刷数组的大小,用来模拟不同类型应用中进行性能波动检测时的缓存占用情况. 共进行9组实验,每组实验进行1万次测量(N=

```
1  long long ns1, ns2, cy1, cy2;
2  long long time_ns[N], time_cy[N];
3  for (int i = 0; i < N; ++i) {
4      ns1 = wall_clock_timing();--------------①
5      cy1 = cpu_clock_timing(); --------------②
6      #pragma GCC unroll 256
7      for (int j = 0; j < K; ++j) {
8          asm volatile ("add $1, %%rax" ::: "rax");
9      }
10     ns2 = wall_clock_timing();--------------①
11     cy2 = cpu_clock_timing(); --------------②
12     time_ns[i] = ns2 - ns1;
13     time_cy[i] = cy2 - cy1;
14     flush_cache(ARRLEN);       --------------③
15 }
```

被测代码

```
PAPI:                                                    ①
ns1/ns2←PAPI_get_real_nsec();
序列化屏障计时(x86):
ns1/ns2←asm volatile("RDTSCP");
序列化屏障计时(Armv8):
ns1/ns2←asm volatile("ISB");
        asm volatile("MRS %0, CNTVCT_EL0");
```

```
PAPI:                                                    ②
cy1/cy2←PAPI_read();
序列化屏障计时(x86):
cy1/cy2←asm volatile("RDPMC");
        asm volatile("LFENCE");
序列化屏障计时(Armv8):
cy1/cy2←asm volatile("MRS %0, PMCCNTR_EL0");
        asm volatile("ISB");
```

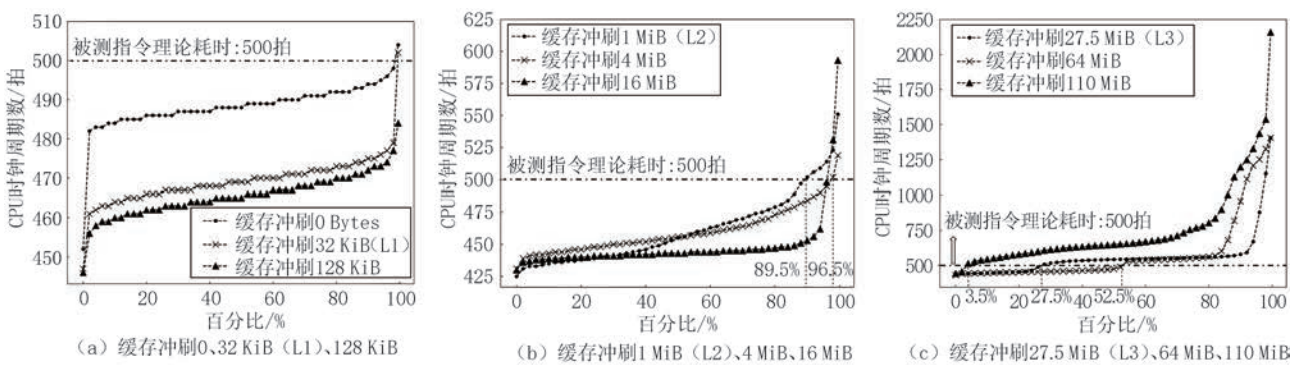
```
                                                         ③
for (int k = 0; k < ARRLEN; k += 8) {
    y[k] += scalar * x[k];
}
```

图1 用于测量计时波动的实验代码

10 000)，每次测量对500条*ADD*指令进行计时(K=500). 9组缓存冲刷数组大小*ARRLEN*分别为0、32 KiB(L1)、128 KiB、1 MiB(L2)、4 MiB、16 MiB、27.5 MiB(L3)、64 MiB和110 MiB. 缓存冲刷数组越大，计时指令运行就越容易发生缓存不命中，处理器进行内存操作的延迟就越长. 对于PAPI等存在用户态和内核态切换和内存拷贝的计时方法而言，缓存冲刷范围越大，计时波动的程度也越严重.

每组实验的测量值百分位累积分布图如图2所示. 为避免极端值对绘图和计算的影响，数据处理中舍去了0.5%的极大值(50个测量值).

观察图2数据可知，当缓存中存在脏数据时，PAPI的计时结果与理论值存在显著偏差. 具体现象有两点：(1)图2(b)中，当缓存冲刷16 MiB时，前96.5%的数据都低于理论值500拍，且均值447.1拍显著低于理论值；(2)图2(c)中，当缓存冲刷110 MiB时，后96.5%的数据都大于500拍，且均值737.7拍显著大于理论值，部分数据甚至超过2000拍以上.



(a) 缓存冲刷0、32 KiB (L1)、128 KiB (b) 缓存冲刷1 MiB (L2)、4 MiB、16 MiB (c) 缓存冲刷27.5 MiB (L3)、64 MiB、110 MiB

图2 在Intel Xeon 6248处理器上使用PAPI计时得到的计时结果累积分布图

结果表明，PAPI的计时波动使计时结果中出现大量严重偏离理论值的数据，使其无法有效检测出真实应用程序中微小的性能异常.

3.2 PAPI计时波动原因分析

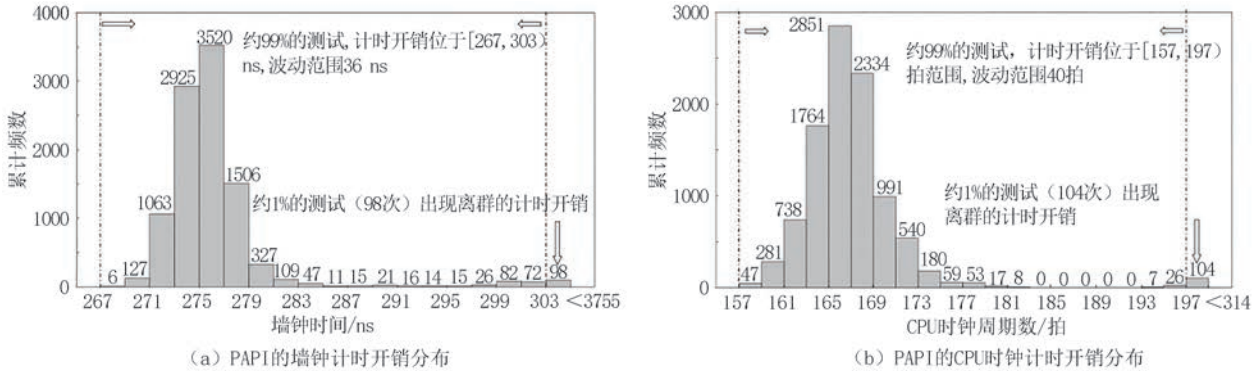
通过两个实验探究计时波动产生的原因.

3.2.1 实验1：计时开销和OS噪声引发计时波动

为检测由计时函数本身的开销以及OS噪声引发的计时波动，本实验删除图1中的被测代码和缓存冲刷代码，此时两组计时调用之间没有代码，也不进行缓存冲刷，因此计时结果为计时开销. 每组实验进行1万次测量，结果如图3所示.

首先，将测量结果作柱状图，如图3(a)、3(b)所示. 不考虑1%的极大值，PAPI的墙钟和CPU时钟计时开销的波动幅度分别为36 ns和40拍. PAPI计时开销存在波动主要有2个原因：(1)计时函数多次跳转，增加了执行过程中的不确定性；(2)通过系统调用读取墙钟数据，计时开销受不同内存地址空间来回拷贝和系统中断的干扰，容易发生波动.

其次，注意到在图3(a)的最右端，部分结果超过3000 ns，超过正常值的10倍. 根据现有研究结



（a）PAPI的墙钟计时开销分布　　（b）PAPI的CPU时钟计时开销分布

图3　PAPI墙钟和CPU时钟的计时开销在Intel Xeon 6248处理器上的分布

果[28]，这种幅度超过1000 ns的波动由OS噪声引起，故OS噪声也是PAPI计时方法波动的原因之一. 在这两个原因的影响下，即便计时指令和数据位于L1缓存，PAPI也会出现在267 ns至303 ns（或157拍至197拍）之间波动的额外开销，导致计时结果偏高且存在波动，而且，还会有约1%的计时结果受操作系统噪声影响，偏高数百至数千纳秒.

3.2.2　实验2：乱序执行和缓存不命中引发计时波动

在实际应用中，乱序执行和缓存不命中是引发PAPI计时波动的两个重要因素. 我们采用与3.1节中相同的配置，在PAPI计时后额外采集指令数量，L3缓存不命中数量和内存读取挂起周期数，如表1. 结合图2和表2的实验结果，可得出2个结论：

表1　PAPI采集的3个硬件事件

硬件事件名	缩写	事件简介
inst_retired.any_p	inst	处理器执行完成的指令数量
mem_load_retired.l3_miss	L3_miss	触发L3缓存不命中的LOAD指令数量
cycle_activity.cycles_mem_any	cy_mem	等待LOAD指令完成的周期数

表2　PAPI采集的硬件事件数据的结果（均值±标准差）

冲刷量 \ 事件名	inst	L3_miss	cy_mem
0 Bytes	1050.9±10.9	0±0	267.3±21.9
32 KiB(L1)	1074.4±20	0±0	290.2±26.2
128 KiB	1079.5±18.8	0±0	294.6±19.2
1 MiB(L2)	1088.6±37.9	0±0	365.3±26.6
4 MiB	1036.5±23.6	0±0	326.9±42.1
16 MiB	1033.3±22	0±0	361.4±53.5
27.5 MiB(L3)	1066.4±41.5	1.4±1.1	486.6±135.5
64 MiB	1073.1±38.3	5.7±0.9	476.7±179.9
110 MiB	1052.5±48	4.1±1.1	500.3±165.3

（1）乱序执行是引发计时波动的原因之一. 现代处理器的流水线延迟通常小于1 ns，测试代码结构简单，理想情况下，500条加法指令的测量值与500拍的差距应小于10拍，且保持稳定无波动. 乱序执行发生时，计时区域内部分*ADD*指令在计时点后被执行，导致测量结果低于理论值（图4情况1），因此在图2(b)中，当缓存冲刷16 MiB时，可以看到前96.5%的测量值都小于500拍，均值为447.1拍，显著低于理论值. 此外，表2中inst的测量值提示计时区域内指令

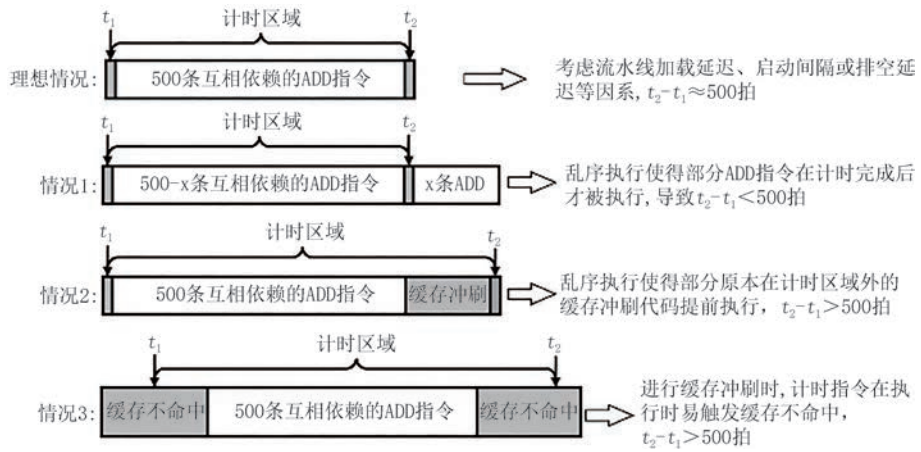


图4　实验代码在运行过程中可能出现的4种情况

数量不稳定，进一步证实了乱序执行的存在.

乱序执行会改变计时指令与被测指令顺序. 计时区域内的指令可能在计时区域外执行(图4情况1)，计时区域外的指令也可能在计时点前执行(图4情况2)，导致测量值发生波动.

(2)缓存不命中是引发计时波动的原因之一. 当表2中数据量从16 MiB增加到27.5 MiB后，不仅出现L3缓存不命中，且内存挂起次数(cy_mem)显著增加. 由于被测代码的*ADD*不涉及内存操作，因此只有两种可能：(1)对应图4情况2，计时区域外的指令触发L3缓存不命中，并被乱序调度到计时区域内；(2)对应图4情况3，PAPI计时函数的访存指令触发L3缓存不命中. 对比图2(b)、图2(c)可知，当数据量从16 MiB增加到27.5 MiB后，不仅计时数据均值从447.1拍增加到546.7拍，且大于500拍的数据比例也从3.5%增加到72.5%.

PAPI的计时波动由缓存不命中和指令乱序共同引起. 计时开始时，由于计时指令(数据)和被测程序上下文之间无依赖，局部性不佳，故计时指令取指和数据写回均容易使缓存不命中. 不命中发生后，当处理器在下级缓存或主存中进行长延迟的内存操作时，会同时通过指令乱序技术，打乱计时指令与上下文其它指令的原有顺序，掩盖内存操作的延迟. 在此过程中，计时指令真正读取时间戳的时刻，既取决于内存操作的延迟，又取决于指令执行的真实顺序，无法被准确测量. 最终，缓存不命中和指令乱序共同导致PAPI对同一段代码的计时结果出现波动，且波动的分布特征随缓存冲刷量改变呈现出不稳定性，如图2所示.

4 序列化屏障计时方法

鉴于PAPI的计时波动会干扰性能波动检测，我们基于x86和Armv8指令集的序列化屏障指令[27,29]，设计了序列化屏障计时方法.

4.1 序列化屏障计时方法的实现

在x86处理器上，根据Intel官方文档[30]，*rdpmc*指令可以读取硬件性能计数器；*rdtscp*兼具计时和指令保序的功能；*lfence*能同时保持指令顺序和内存操作顺序，因此利用上述指令设计计时方法. 如图1所示，*rdtscp*和*rdpmc*指令分别用于读取墙钟和CPU时钟数据，*rdtscp*和*lfence*用于抑制乱序执行，确保计时区域内被测指令序列的稳定.

在Armv8处理器上，根据Arm官方文档[31]，*isb*能同时起到指令保序和内存屏障作用；*cntvct_el0*和*pmccntr_el0*两个时钟寄存器分别保存了墙钟和CPU时钟数据. 如图1所示，*mrs*用于从2个时钟寄存器中分别读取墙钟和CPU时钟数据，*isb*用于指令保序.

4.2 序列化屏障计时相对PAPI计时的两个优势

4.2.1 减少由计时指令开销和OS噪声引起的波动

如表3所示，序列化屏障计时方法直接使用汇编指令完成计时功能，避免使用系统调用，缩短了调用路径，压缩了计时开销.

表3 序列化屏障计时方法和PAPI的对比

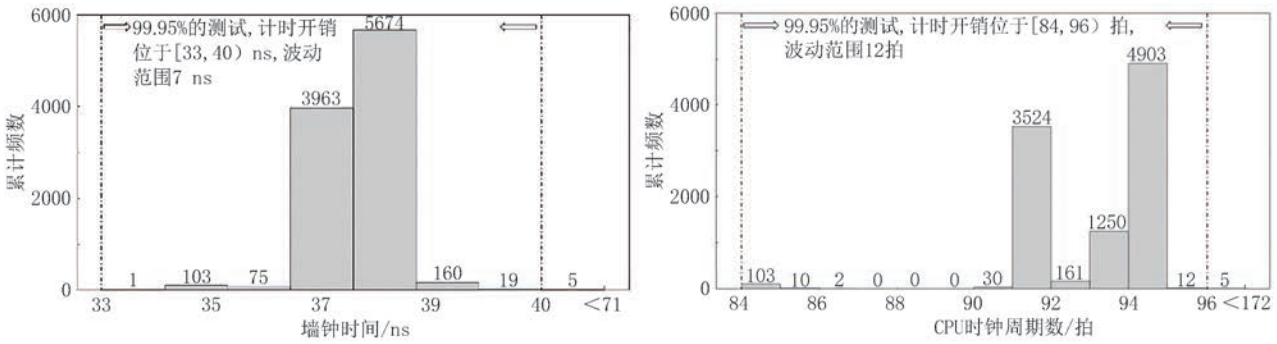
指令集和计时方法 \ 时钟		墙钟	CPU时钟
x86	PAPI计时	系统调用 *clock_gettime*	*rdpmc*
	序列化屏障计时	*rdtscp* + *lfence*	*rdpmc* + *lfence*
Armv8	PAPI计时	系统调用 *clock_gettime*	*perf_event_open*
	序列化屏障计时	*mrs cntvct_el0* + *isb*	*mrs pmccntr_el0* + *isb*

为检验优势是否成立，我们参照第3.2.1节进行对比实验，结果如下：据图3，PAPI计时前99%的墙钟和CPU时钟数据波动幅度为36 ns和40拍；据图5，序列化屏障计时方法前99.9%的墙钟和CPU时钟数据波动幅度仅为7 ns和12拍. 此外，图5中未出现偏离正常值超过1000 ns的异常值. 综上，序列化屏障计时方法的开销波动幅度显著低于PAPI，且更不容易受OS噪声干扰.

4.2.2 抑制由乱序执行和缓存不命中引起的波动

为抑制乱序执行和缓存不命中引起的波动，序列化屏障计时方法做了两点改进：(1)如表3所示，使用序列化和内存屏障指令来抑制乱序执行；(2)直接使用汇编完成计时功能，减少函数跳转次数和访存微指令数量，降低触发缓存不命中的次数. 为检验优势是否成立，我们在Intel Xeon 6248的不同缓存层次下对比两种计时方法的稳定性. 对于L1缓存下的对比，设置缓存冲刷数据量为0，此时计时指令及数据都在L1缓存中；对于L2缓存下的对比，设置缓存冲刷数据量为128 KiB(4倍L1)，此时计时指令及数据都在L2缓存中. 依次类推，对于L3缓存和主存下的对比，分别设置缓存冲刷数据量为4 MiB(4倍L2)和110 MiB(4倍L3).

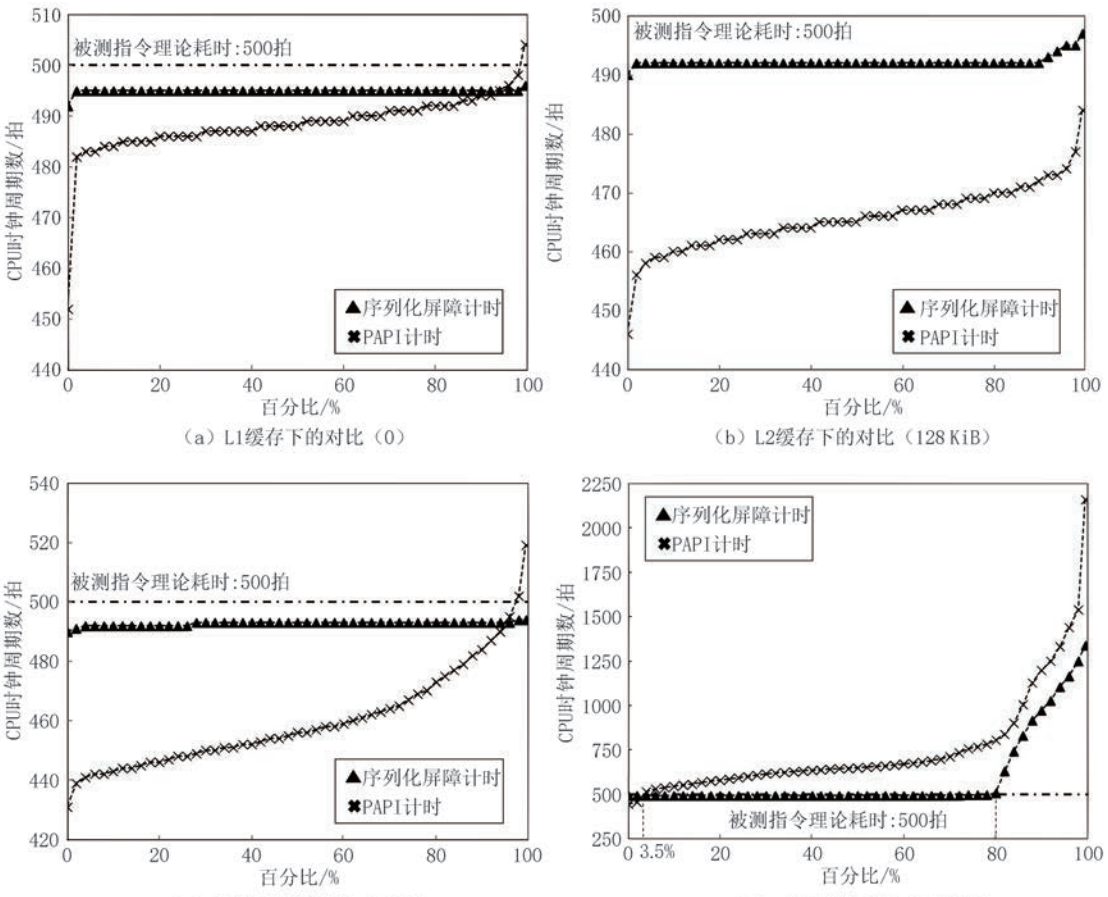
如图6(a)-(c)所示，在不同缓存层次下，序列化



（a）序列化屏障计时方法的墙钟计时开销分布　　（b）序列化屏障计时方法的CPU时钟的计时开销分布

图5　序列化屏障计时方法墙钟和CPU时钟的计时开销在Intel Xeon 6248上的分布

屏障计时方法相比PAPI计时方法，与理论值吻合得更好，且计时数据的波动幅度更低．如图6(d)中，序列化屏障计时80%的数据分布在500拍附近，而PAPI则存在大量偏离500拍的数据．综上，序列化屏障计时方法能有效抑制由乱序执行和缓存不命中引发的波动．



（a）L1缓存下的对比（0）　（b）L2缓存下的对比（128 KiB）

（c）L3缓存下的对比（4 MiB）　（d）主存下的对比（110 MiB）

图6　Intel Xeon 6248上，两种计时方法在不同缓存层次下的计时波动对比

5　计时波动的解析模型与评价指标

为定量评价和对比不同计时方法的精度和灵敏度，我们分析了计时过程中测量值、真值和计时波动之间的关系，构建了计时波动的解析模型．并基于该模型，提出了用于评价计时方法精度和灵敏度的量化指标：计时下限$t_{\min}$和波动分度t_{diff}，说明了指标测量方法．

5.1　计时波动来源的定性分析

在图4的基础上，我们进一步将计时指令和被测指令表示为如图7所示的简化的双端口乱序流水

线模型. 计时波动表现为:对相同工作量的计算过程进行反复测量时,测量值与真值之间的误差不稳定,上下波动. 在测量结果中,计时波动与性能波动叠加,使微小的性能波动难以被检测出来.

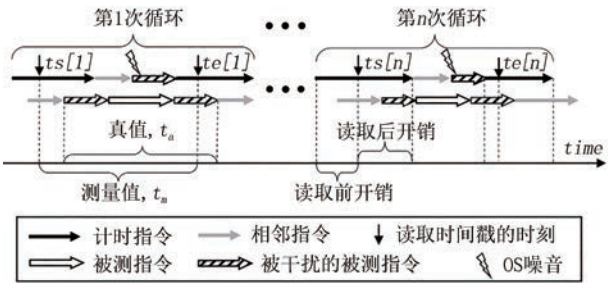


图7 插桩后计时指令与被测指令的执行模型

一方面,计时波动来源于计时指令与被测指令的相互干扰. 如图7,以时间戳被读取的时刻为界,计时指令开销分为前后两部分. 对PAPI而言,读取前开销主要来自计时函数运行与跳转,用户态到内核态切换;读取后开销主要来自内核态到用户态切换,时间戳的写回等. 上述过程中的OS噪声、缓存不命中和乱序执行等因素,不仅为PAPI的计时开销增添了不确定性,还与被测程序发生互相干扰. 根据Röhl等人[13]的研究,在对4核心STREAM基准测试的计时中,PAPI的启动和停止会产生约20 000拍的额外开销,导致测量结果出现约15 000拍幅度的波动. 序列化屏障计时方法减少了函数跳转,消除了内核态切换,从而减少了计时开销,降低了波动程度. 然而,计时指令和数据的内存操作是不可控的,缓存不命中无法完全避免,故实验中仍能观察到缓存不命中引起的计时波动(图6(d)).

另一方面,计时波动还来源于计时区间与被测区间的错位. 如图7所示,时间戳真正被读取的位置与被测代码的起止位置错位. 这一方面由于计时开销存在波动,时间戳的实际位置不稳定;另一方面由于指令乱序执行机制,计时指令与被测指令的先后顺序不稳定. 对于PAPI而言,计时开销的波动、缓存不命中和指令乱序,都会导致这种错位. 本文的序列化屏障计时方法通过加入内存屏障,强制处理器后端在屏障指令前完成读写操作,从而抑制了乱序执行,降低了错位程度. 然而,计时开销的波动无法完全消除,因此在实验中,仍能观察到测量值与理论值发生偏差或波动的情况(图6).

综上所述,序列化屏障计时方法克服了PAPI计时方法的部分缺陷,但两种计时方法都会由于计时指令与被测指令的相互干扰,以及计时点与被测区间错位,产生计时波动. 因此,需要进一步建立数学模型,以便量化评价不同计时方法在不同的性能波动检测场景下的精度与灵敏度.

5.2 计时波动的解析模型

在反复测量一段代码的运行时间时,由于计时波动的存在,测量值与真值之间必然存在偏差. 为简化模型,假设处理器主频、温度等状态不变,且只运行被测程序和OS进程,同时被测程序每次运行的计算过程相同.

对于存在性能波动的被测程序和存在计时波动的计时方法,每次测量可看作是从被测程序、计时方法、OS噪声三者各自波动范围组成的样本空间中采样. 根据图4,对于一次计时测量,真值t_a、计时开销t_e、测量值t_m和OS噪声τ_{os},存在如式1所示的关系.

$$\begin{cases} t_a = t_{a,std} + \tau_a, t_a \in T_a \\ t_e = t_{e,std} + \tau_e, t_e \in T_e \\ t_m = t_a + t_e + \tau_{os}, t_m \in T_m, \tau_{os} \in T_{os} \end{cases} \tag{1}$$

式1中各参数具体说明如下:

t_a:真值. 被测程序的真实运行时间,由程序理论最短运行时间$t_{a,std}$与程序由于性能波动而增加的耗时τ_a组成. τ_a恒大于0.

t_e:计时开销. 由计时指令产生的额外耗时,由计时理论最短开销$t_{e,std}$与计时波动导致的时间波动τ_e组成. 由图4,τ_e可正可负.

t_m:测量值. 通过计时测得的程序运行时间;

τ_{os}:OS噪声耗时. 即测量中由OS噪声产生的额外时间开销. τ_{os}恒大于0.

T_m,T_a,T_e和T_{os}:分别代表测量值、真值、计时开销和OS噪声耗时所有样本的集合.

根据现有研究[1,28],在耗时极短的计算过程中,受OS噪声影响的测量值在波动幅度和出现频率上与其它测量值有显著差异,故设计算法将包含OS噪声的测量值去除(见5.4.2节). 此外,计时理论最短开销可测量得到(见5.4.1节). 于是,可以从式1导出更逼近真值的测量结果$\hat{t}_m \in \hat{T}_m$:

$$\begin{aligned} \hat{t}_m &= t_m - t_{e,std} \\ &= t_a + \tau_e \\ &= t_{a,std} + \tau_a + \tau_e, \left(\text{当且仅当} \tau_{os} = 0\right) \end{aligned} \tag{2}$$

用联合概率分布表达测量值,测量值$\hat{t}_m = \hat{t}_{m,x}$,计时波动范围$\tau_e \in (\tau_{e,min}, \tau_{e,max})$,则测量值在样本中出现的概率可以表示为:

$$\mathrm{P}\left(\hat{t}_m=\hat{t}_{m,x}\right)=\sum_{\tau_{e,x}=\tau_{e,min}}^{\tau_{e,max}}P\left(t_a=\hat{t}_{m,x}-\tau_{e,x},\tau_e=\tau_{e,x}\right) \tag{3}$$

式2与式3联立为计时波动的解析模型，描述在不考虑OS噪声和计时理论最短开销时，计时波动对测量值的影响. 据式2，计时波动是导致测量值偏离真值的主要原因；据式3，计时精度和灵敏度与性能波动检测能力直接相关，可用于定量评价不同计时方法的性能波动检测能力.

5.3　评价指标的推导与计算

精度和灵敏度是评价性能波动检测能力的关键. 在性能波动检测中，能准确测量的时间越短，则精度越高；能准确分辨的性能波动幅度越小，则灵敏度越高. 为评价检测方法的精度和灵敏度，我们根据解析模型分析精度和灵敏度受限的原因，对关键评价指标进行推导与计算.

5.3.1　精度与灵敏度指标的推导

进行性能波动检测时，首先需要通过反复计时对 $\hat{T}_m$ 反复采样，然后观察多个样本，并将计时结果的波动幅度与程序运行时间比较. 据式2，对测量值 $\hat{t}_{m,i}$ 和 $\hat{t}_{m,j}$，测量值波动幅度与真值之比为

$$v_{i,j}=\frac{\hat{t}_{m,i}-\hat{t}_{m,j}}{t_{a,std}}=\frac{\left(\tau_{a,i}-\tau_{a,j}\right)+\left(\tau_{e,i}-\tau_{e,j}\right)}{t_{a,std}} \tag{4}$$

根据式4，可以得出限制性能波动检测精度和灵敏度的主要因素，并据此得出关键评价指标.

(1)指标1：计时下限

根据式4，$t_{a,std}$ 下降时，由于计时波动不可控，$\left(\tau_{e,i}-\tau_{e,j}\right)$ 对 $t_{a,std}$ 的影响将增加，使程序运行时间无法被准确测量，这对检测精度造成限制. 因此只有 $t_{a,std}$ 高于一下限时，才能将计时波动对测量造成的误差控制在可接受范围内. 称这一下限值为计时下限 t_{min}，表示计时方法能够准确测量的最短程序耗时.

(2)指标2：波动分度

跟据式2，τ_a 波动幅度下降时，波动检测所需灵敏度提高，但计时波动带来的干扰对 $\hat{t}_m$ 的影响也随之增加，使多个测量值之间的微小差异难以被区分，对检测灵敏度造成了限制. 故只有 τ_a 的变化幅度高于一下限时，才能在测量结果中被准确辨别. 称 $(\tau_{a,i}-\tau_{a,j})$ 的下限为波动分度 t_{diff}，表示计时方法能够准确分辨的最小时间差异.

5.3.2　指标计算1：计时下限

计时下限的计算可转变为最优化问题. 首先进行误差来源分析，令式4中 $(\tau_{a,i}-\tau_{a,j})=0$，构造不存在性能波动、理论耗时为 $t_{a,std}$ 的程序. 此时 $\hat{t}_m=t_{a,std}+\tau_e$，$\hat{t}_m\in\hat{T}_m$，测量值的误差来源只有计时波动 τ_e. 若增大 $t_{a,std}$，则由计时波动造成的误差比例将相应减少. 对于一个给定的可接受误差范围，计时下限的求解就是在 $t_{a,std}$ 不断增加的过程中，判断 $\hat{T}_m$ 的波动范围是否满足误差要求.

接下来，使用变异系数(CV)量化 $\hat{T}_m$ 中样本的波动程度. 一组测量值 $\hat{t}_{m,i}\in\hat{T}_m$ 的变异系数为

$$\mathrm{CV}\left(\hat{T}_m\right)=\frac{\mathrm{std}\left(\hat{T}_m\right)}{\mathrm{mean}\left(\hat{T}_m\right)} \tag{5}$$

$\mathrm{std}\left(\hat{T}_m\right)$ 和 $\mathrm{mean}\left(\hat{T}_m\right)$ 分别表示测量样本的标准差和均值. 用 CV 衡量计时波动对测量造成的误差，本文设阈值 $\epsilon=0.01$，当 $CV<\epsilon$ 时，计时波动对测量造成的误差不足1%. 据此，设置目标函数

$$\mathrm{F}\left(t_{a,std}\right)=\left|\epsilon-\mathrm{CV}\left(\hat{T}_m\right)\right| \tag{6}$$

计时下限的测量即在与实际测量相同的测量环境下，寻找 $t_{a,std}$，使式6中的 $\mathrm{F}(t_{a,std})$ 最小. 该问题的解即为计时下限 t_{min}. 对于单次性能波动检测，若被测时间大于 t_{min}，则意味着计时波动的幅度可以被忽略. t_{min} 越小，性能波动检测精度越高.

5.3.3　指标计算2：波动分度

波动分度的计算也可以转化为最优化问题. 由式3，对同一真值多次测量会得到不同测量值. 若多次测量的时间差很小，便无法判断这种差异是源自计时波动还是性能波动. 此时，为分辨性能波动 τ_a，检测方法在不同真值上测得的测量值分布应存在显著性差异. 设两份无性能波动的代码A0、A1的理论耗时分别为 $t_{A0,std}$ 和 $t_{A1,std}$，令 $t_{A1,std}>t_{A0,std}$. 由式2可得测量值：

$$\begin{cases}\hat{t}_{A0,m}=t_{A0,std}+\tau_{\mathrm{e}},\hat{t}_{A0,m}\in\hat{T}_{A0,m}\\ \hat{t}_{A1,m}=t_{A1,std}+\tau_{\mathrm{e}},\hat{t}_{A1,m}\in\hat{T}_{A1,m}\end{cases} \tag{7}$$

根据3.2.2节中的分析，$\hat{T}_{A0,m}$、$\hat{T}_{A1,m}$ 和 T_{e} 中的样本均不一定为正态分布. 因此，我们参考连续概率分布中，概率密度曲线重合面积的意义，定义显著性指标α，令 $\hat{T}_{overlap}=\hat{T}_{A0,m}\cap\hat{T}_{A1,m}$，则有

$$\begin{cases}\alpha_{A0}=P\left(\hat{t}_{A0,m}\in\hat{T}_{overlap}\right)\\ \alpha_{A1}=P\left(\hat{t}_{A1,m}\in\hat{T}_{overlap}\right)\end{cases} \tag{8}$$

根据全概率定理，$\alpha=\alpha_{A0}=\alpha_{A1}$. 根据Hoefler等人的研究[12]，本文设显著性指标 $\alpha_{std}=0.05$，即对于具有显著差异的两组测量值组成的集合，要求交集

元素数量少于集合元素数量的5%. 接下来,令$\Delta t=t_{A1,std}-t_{A0,std}$,固定$t_{A0,std}$,那么$\alpha$成为以$\Delta t$为自变量的函数$\alpha(\Delta t)$. 设置目标函数

$$F(\Delta t)=\alpha_{std}-\alpha(\Delta t),\ \alpha<\alpha_{std} \tag{9}$$

测量波动分度,即在与实际测量相同的测量环境下,搜索Δt,使得对于任意$t_{A0,std}>t_{min}$,$F(\alpha)$最小. 该问题的解是波动分度t_{diff}. 当两个真值差距超过t_{diff}时,它们测量值的重合次数不超过测量次数的5%. t_{diff}越小,性能波动检测灵敏度越高.

5.4 评价指标测量方法

测量评价指标主要有个4步骤:去除计时开销,OS噪声过滤,测量计时下限和测量波动分度.

5.4.1 第1步:去除计时理论最短开销

使用3.2.1节的方式,去除计时理论最短开销(*filter_cost*). 据图3、图5,PAPI计时方法在Intel Xeon 6248上的墙钟和CPU时钟开销分别为267 ns和157拍,序列化屏障计时方法的开销为33 ns和84拍.

5.4.2 第2步:过滤OS噪声

为过滤OS噪声导致的异常数据,我们基于孤立森林算法[32]设计了OS噪声过滤(*filter_os*)算法. 算法的输入为N个去除$t_{e,\text{std}}$后的测量数据点,每个点都有墙钟和CPU时钟测量值两个特征维度,主要步骤如下:

(1)使用孤立森林计算每个点的异常分数,值域为(−1, 0),分值越低,数据异常程度越大. 将异常分数分界值设为β,只有异常分数小于该值的点才标记为异常点.

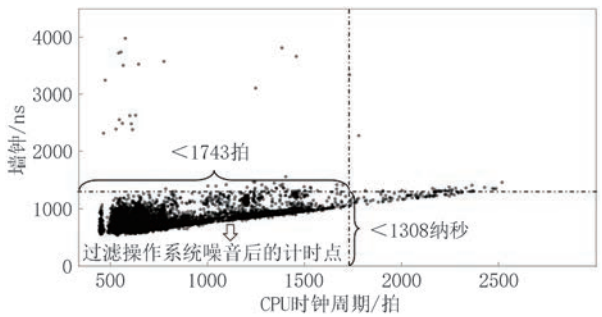


图8 OS噪声过滤算法对PAPI计时数据的过滤效果

(2)将β搜索上界固定为经验值−0.6,下界固定为上一步计算出的异常分数最小值. 以−0.6为起点,每步递减0.01,遍历所有分界值.

(3)以每个遍历到的异常分数为界,在测量值中过滤异常值,保留正常值中的最大值. 随着遍历进行,上述最大值的极值数组是单调递增的.

(4)对极值数组进行前向差分得到差分数组,计算该数组的均值. 从头遍历差分数组,判断当前值是否大于上述均值. 若为真,则认为其对应的分界值即为最佳分界值.

(5)使用上一步得到的最佳分界值对测量数据进行异常过滤,得到过滤OS噪声后的数据.

5.4.3 第3步:测量计时下限

据5.3.2节,测量$t_{\min}$,即让$t_{a,std}$不断增加,直至满足条件$CV(\hat{T}_m)<\epsilon$. 如算法1所示,首先通过改变ADD指令数,构建理论运行时间为$t_{a,std}$的被测计算过程(第7行),其次通过不断判断测量值的波动程度是否满足最优化问题的条件来搜索$t_{\min}$.

由于从0开始递增搜索速度过慢,故这里采用二分查找,不断缩小搜索步长*step*和搜索区间$[start,end]$,直至$t_{a,std}$满足条件(5-14行). 然后对$t_{a,std}$进行p次重复测量,以确认其满足最优化问题的条件,若不满足,则继续增加$t_{a,std}$(15-31行). 通过这种方式来减小因测量过程的随机性而导致的结果波动,得到稳定的结果. 根据经验,取$p=30$可以在兼顾算法效率的同时,收敛到稳定的结果,如图8所示.

算法1. 计时下限测量算法.

输入:变异系数阈值$\epsilon=0.01$,计时开销$t_{e,std}$,重复确认次数p,重复计时次数N,缓存冲刷大小size

输出:计时下限t_{min}

```
1.  start←0, end←0, t_{a,std}←0, step←10 000
2.  WHILE step ⩾ 1 DO
3.    flag←0
4.    WHILE flag == 0 DO
5.      t_{a,std}←t_{a,std} + step
6.      FOR i←1 to N DO
7.        cy_i, ns_i←timing(t_{a,std} x ADD)
8.        flush_cache(size)
9.      tm_arr ←[(cy_1, ns_1),…(cy_N, ns_N)];
10.     tm_arr←filter_os(filter_cost(t_{e,std}, tm_arr))
11.     计算tm_arr的墙钟数据变异系数cv_ns
12.     计算tm_arr的CPU时钟数据变异系数cv_cy
13.     IF cv_cy > ε AND cv_ns > ε THEN
14.       start←t_{a,std}
15.     ELSE
16.       cnt←0;
17.       FOR i←1 to p DO
18.         FOR i←1 to N DO
19.           cy_i, ns_i←timing(t_{a,std} x ADD)
20.           flush_cache(size)
21.         tm_arr←[(cy_1, ns_1),…(cy_N, ns_N)]
22.         tm_arr←filter_os (filter_cost (t_{e,std}, tm_
            arr))
```

```
23.            计算 tm_arr 的墙钟数据变异系数 cv_ns
24.            计算 tm_arr 的CPU时钟数据变异系数 cv_cy
25.            IF cv_cy > ϵ AND cv_ns > ϵ THEN
26.              BREAK
27.            cnt←cnt + 1
28.          IF cnt == p THEN
29.            flag←1, end←t_a,std
30.          ELSE
31.            start←t_a,std
32.        t_a,std←start, step←step/10
33.    t_min←t_a,std
34.    RETURN t_min
```

5.4.4　第4步：测量波动分度

据5.3.3节，测量 t_{diff} 即选定 $t_{A0,std}$，让Δt不断增加，直至满足显著性条件 α_{std}=0.05. 在算法2中，为加速测量过程，基于二分查找思想搜索 t_{diff}，每一轮都将搜索步长 *step* 以及搜索区间(left, right]长度缩小为原来的十分之一，直至得到满足显著性条件的最小值 t_{diff}. 注：算法2以CPU时钟的符号为例，测量墙钟指标的算法与此相同.

由于存在无穷多对满足条件的($t_{A0,std}$, $t_{A1,std}$)，故需要对解空间抽样. 在算法2的第7-28行中，选取 q 组($t_{A0,std}$, $t_{A1,std}$)进行显著性差异判断，只有每组($t_{A0,std}$, $t_{A1,std}$)都满足显著性条件时，才接受当前的波动分度 t_{diff}，否则继续增加 Δt. 根据经验，取 q=80可以在兼顾算法效率的同时，收敛到稳定的结果.

算法2.　波动分度测量算法.

输入：显著水平 α=0.05，计时开销 $t_{e,std}$，t_{min}，抽样次数 q，重复计时次数 N，缓存冲刷大小 *size*

输出：波动分度 t_{diff}

```
1.   start←0, end←0, t_diff←0, step←100;
2.   WHILE step ⩾ 1 DO
3.     flag←0
4.     WHILE flag == 0 DO
5.       t_diff← t_diff + step
6.       cnt ← 0
7.       FOR i←1 to q do
8.         t_A0,std← t_min + (i − 1) x t_diff
9.         t_A1,std← t_min + i x t_diff
10.          FOR i ← 1 to N DO // 测量A0
11.            cy_A0,i ← timing (t_A0,std x ADD);
12.            flush_cache (size)
13.          FOR i ← 1 to N DO // 测量A1
14.            cy_A1,i ← timing (t_A1,std x ADD);
15.            flush_cache(size)
16.          cy_A0_arr←filter_os(filter_cost(cy_A0_arr))
17.          cy_A1_arr←filter_os(filter_cost(cy_A1_arr))
18.          cy_A0_arr←sort_ascend(cy_A0_arr)
19.          cy_A1_arr←sort_ascend(cy_A1_arr)
20.          max_A0 = max(cy_A0_arr);
21.          overlap_cnt ← 0 // 统计重叠测量点数量
22.          FOREACH cy in cy_A1_arr[1…M] DO
23.            IF cy < max_A0 then
24.              overlap_cnt ← overlap_cnt + 1
25.          ovlp←overlap_cnt/M;
26.          IF ovlp > α THEN
27.            BREAK
28.          cnt←cnt + 1;
29.        IF cnt == q THEN
30.          end←t_diff ,flag←1
31.        ELSE
32.          start←t_diff
33.      t_diff←start, step←step/10
34.    RETURN t_diff
```

6　实验验证与结果分析

我们通过实验，从精度和灵敏度两方面，对序列化屏障计时方法与PAPI计时方法进行定量对比.

6.1　实验环境

实验基于Intel Xeon 6248和华为鲲鹏920-6426处理器，具体软硬件配置如表4所示.

表4　实验平台软硬件配置

	至强平台	鲲鹏平台
处理器	Intel Xeon Gold 6248	华为鲲鹏920-6426
指令集	x86	Armv8.2
主频	2.5 GHz	2.6 GHz
操作系统	CentOS 7.7.1908	CentOS 7.6.1810
内核版本	4.14.246	4.18.20
内存配置	192GiB DDR4-2666	256GiB DDR4-2933
PAPI版本	PAPI-6.0.0.1	
编译器	GCC-9.3.0	

6.2　实验方法与结果处理

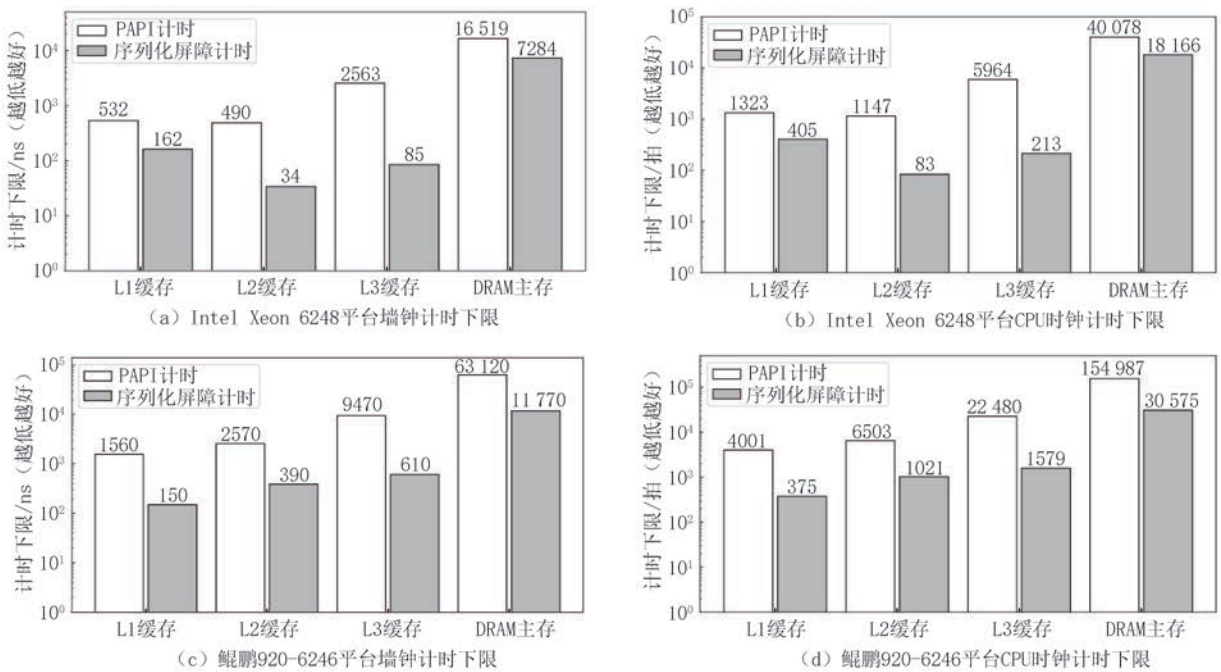
与第4.2.2节类似，实验在多个缓存层次下进行. 对应L1、L2、L3缓存和主存，分别设置缓存冲刷数据量为0、4倍L1、4倍L2和4倍L3缓存. 在Xeon6248上，缓存冲刷量分别为0 KiB、128 KiB、4 MiB和110 MiB；在鲲鹏920上，缓存冲刷量分别为0 KiB、256 KiB、2 MiB和256 MiB.

采用计时下限 t_{min} 来评价计时方法精度，采用波动分度 t_{diff} 来评价计时方法灵敏度. 测量 t_{min} 时，取误

差水平 $\varepsilon=0.01$. t_{min}越小，计时方法精度越高. 测量t_{diff}时，取显著水平$\alpha_{std}=0.05$，t_{diff}越小，计时方法灵敏度越高.

6.3 实验1：对比计时方法精度

图9(a)和图9(b)是Intel Xeon 6248处理器上的实验结果. 从图中可以看出，序列化屏障计时方法在各个缓存层次下的t_{min}都显著小于PAPI. 最好情况下(L3)，PAPI的墙钟和CPU时钟的计时下限分别为2563 ns和5964拍，而序列化屏障计时方法低至85 ns和213拍，序列化屏障计时方法的墙钟和CPU时钟精度分别为PAPI的30.2倍和28.0倍；最差情况下(主存)，PAPI计时下限分别为16 519 ns和40 078拍，而序列化屏障计时方法分别为7284 ns和18 166拍，序列化屏障计时方法的墙钟和CPU时钟精度分别为PAPI的2.3倍和2.2倍.



(a) Intel Xeon 6248平台墙钟计时下限　(b) Intel Xeon 6248平台CPU时钟计时下限

(c) 鲲鹏920-6246平台墙钟计时下限　(d) 鲲鹏920-6246平台CPU时钟计时下限

图9　两种计时方法在Intel Xeon 6248和鲲鹏920-6426不同缓存层次下的计时下限对比

图9(c)和图9(d)是鲲鹏920-6426处理器上的实验结果. 与Intel Xeon 6248上的结果类似，序列化屏障计时方法的精度同样显著优于PAPI. 最好情况下(L3)，序列化屏障计时方法在墙钟和CPU时钟下的精度分别为PAPI的15.5倍和14.2倍；最差情况下(主存)，序列化屏障计时方法在墙钟和CPU时钟下的精度分别为PAPI的5.4倍和5.1倍.

以上结果表明，相比PAPI计时方法，序列化屏障计时方法的计时下限更低，即精度更高.

6.4 实验2：对比计时方法灵敏度

图10(a)和图10(b)是Intel Xeon 6248处理器上的实验结果. 从图中可以看出，序列化屏障计时方法在各个缓存层次下的灵敏度都比PAPI计时要高. 最好情况下(L3缓存)，PAPI波动分度为124 ns和77拍，而序列化屏障波动分度低至4 ns和7拍. 序列化屏障计时方法的墙钟和CPU时钟灵敏度分别为PAPI的31.0倍和11.0倍. 在最差情况下(主存)，PAPI波动分度为683 ns和772拍，而序列化屏障波动分度低至208 ns和300拍，序列化屏障计时方法的墙钟和CPU时钟灵敏度分别为PAPI的3.3倍和2.6倍.

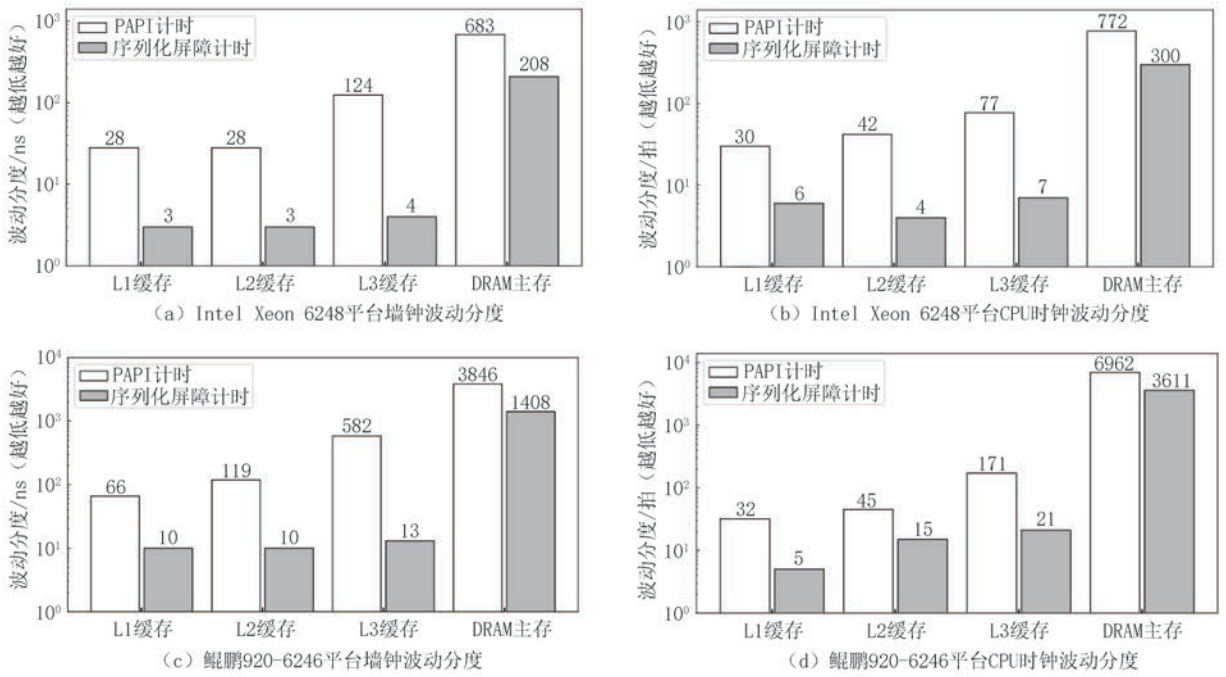
图10(c)和图10(d)是鲲鹏920-6426处理器上的实验结果. 从图中可以看出，序列化屏障计时方法在鲲鹏920-6426上同样具有更高的灵敏度. 最好情况下(L3缓存)，序列化屏障计时方法的墙钟和CPU时钟灵敏度分别为PAPI的44.8倍和8.1倍. 最差情况下(主存)，序列化屏障计时方法的墙钟和CPU时钟灵敏度分别为PAPI的2.7倍和1.9倍.

以上结果表明，相比PAPI计时方法，序列化屏障计时方法的波动分度更低，即灵敏度更高.

7 相关工作

7.1 处理器性能检测中存在的测量结果波动

已有一些研究指出处理器性能检测中存在测量波动与测量误差. Mytkowicz等人[33]发现，使用PAPI



（a）Intel Xeon 6248平台墙钟波动分度　（b）Intel Xeon 6248平台CPU时钟波动分度

（c）鲲鹏920-6246平台墙钟波动分度　（d）鲲鹏920-6246平台CPU时钟波动分度

图10　两种计时方法在Intel Xeon 6248和鲲鹏920-6426不同缓存层次下的波动分度对比

对被测程序进行插桩测量时，测量结果会随着被测指标的数量、被测程序和环境变量数量等因素的变化而发生剧烈波动. 之后，他们[34]又进一步发现，测量环境的微小变化也会导致测量结果出现很大偏差. 他们认为波动产生的原因主要来自主观因素，包括测量工具对被测程序的干扰、操作系统设置不稳定和实验设计不合理等，并提出通过控制变量和改进实验设计可以减少测量波动. 与之相反，本文从性能测量的硬件原理出发，指出计算机系统的测量波动是客观存在，无法避免的，其产生原因包括OS噪声、乱序执行和缓存不命中等. Weaver等人[26,35-36]指出处理器性能计数器普遍存在计数波动，且受OS噪声和程序内存排布等多种因素影响. 这些工作主要关注程序每次运行后得到的测量结果是否稳定. 相比之下，本文则更进一步，检测了处理器架构中的计时波动，并证明这些波动对指令计时造成了干扰，且具体分析了原因.

7.2　处理器性能波动的检测方法与工具

检测处理器性能波动主要有两种方法. 一是代码插桩. Tang等人[17]提出了一种能够识别源代码中固定工作量的循环结构算法，从而实现自动化代码插桩和性能波动检测. 然而，这一工作未公开其使用的计时测量方法. Afzal等人[37]在测量并行计算中的计算与通信过程中不同步的现象时，使用了C++Chrono库中高精度时钟进行插桩检测. 上述方法适用于运行时间较长的程序，对精度和灵敏度要求较低. 相比之下，本文提出的序列化屏障方法适用于对精度和灵敏度要求更高的性能波动检测.

二是重复采样. 通过反复运行性能评测程序，对比每次运行时间，判断是否存在性能波动[2,6,38-39]. VarBench[40]和HPAS[41]则进一步通过在小型计算程序中主动触发或人为制造处理器性能波动的方式来检测性能波动对程序造成的性能影响. 但这些方法通用性较差，无法对程序中的部分代码进行性能波动检测. McCalpin[10]研究Stampede-2超算的处理器性能波动时，通过对性能计数器进行定时采样，定位性能波动来源. 然而此方法也无法精确定位到某段代码，因而通用性不强. 相比之下，本文提出的方法可以插入到复杂应用程序的任意位置，对任一代码段进行性能波动检测，通用性更好.

7.3　处理器性能波动检测方法的模型评价

已有不少研究对计算机系统性能测量中的误差与不确定性进行了统计分析并提出了对策. Hoefler等人[12]和Chen等人[16]都提出了通过改善实验设计和改进统计学分析方法，能减少性能测试中的不确定性，从而使性能对比更合理. 有一些研究在判断计时测量准确性时，以计时测量开销占被测程序运行时间的比例作为关键指标[17,42]. 与上述工作不同，本文首次分析了计时方法自身的波动性引起的测量误差和不确定性，并进一步提出计时波动的解析模型和评价指标.

许多研究对超级计算机的性能波动进行了数学

建模与性能预测，包括操作系统和网络的波动建模预测[1]、存储系统的波动建模[43]、任务间互相干扰的性能波动建模与预测[44]和性能波动对扩展性影响的建模预测[39,42]等．部分处理器性能波动相关的研究也进行了建模分析，如Chang等人[45]在解决HPL性能波动时采用了多因子模型对平均每秒浮点操作数的波动进行建模，然而已有工作都是针对特定条件下的特定应用进行建模．相比之下，本文提出的计时波动解析模型体现了性能波动与测量波动对任意计算过程的影响，通用性更好．

8 总结与展望

本研究解决了处理器波动检测中的两个问题：(1)现有工具难以检测微小的性能波动；(2)现有方法难以客观评价不同工具的性能波动检测能力．首先，通过实验，证明了PAPI在不同的缓存状态下，会得到不稳定的计时结果，并分析了其产生计时波动的原因．第二，设计了跨平台的序列化屏障计时方法，对计时波动进行了抑制，在实验中观察到比PAPI幅度更低的计时波动．第三，对计时波动的不稳定性和其对测量值的影响进行了定性分析和建模．面向处理器性能波动检测，首次提出了跨平台的计时方法精度和灵敏度指标及评价方法．第四，在对比评测中量化了序列化屏障计时方法和PAPI计时方法的精度和灵敏度，证明序列化屏障计时能够在多种缓存冲刷大小下检测纳秒级性能波动．

下一步工作，我们将考虑多核环境下由并行资源竞争引起的处理器性能波动，进一步完善本文提出的序列化屏障计时方法及两个评价指标．

参 考 文 献

[1] Hoefler T, Schneider T, Lumsdaine A. Characterizing the influence of system noise on large-scale applications by simulation//Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. New Orleans, USA, 2010: 1-11

[2] Chunduri S, Harms K, Parker S, et al. Run-to-run variability on xeon phi based cray xc systems//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. Denver, USA, 2017: 1-13

[3] Maricq A, Duplyakin D, Jimenez I, et al. Taming performance variability//Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18. Carlsbad, USA, 2018: 409-425

[4] De Sensi D, Di Girolamo S, Hoefler T. Mitigating network noise on dragonfly networks through application-aware routing//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. New York, USA, 2019: 1-32

[5] Bhatele A, Thiagarajan J J, Groves T, et al. The case of performance variability on dragonfly-based systems//Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS). New Orleans, USA, 2020: 896-905

[6] Acun B, Miller P, Kale L V. Variation among processors under turbo boost in hpc systems//Proceedings of the 2016 International Conference on Supercomputing. New York, USA, 2016: 1-12

[7] Gottschlag M, Bellosa F. Reducing avx-induced frequency variation with core specialization//Proceedings of 9th Workshop on Systems for Multicore and Heterogeneous Architectures. Dresden, Germany, 2019:23-31

[8] Zi-Cong W, Chen X W, Yang G. Research on cache access equalization in chip multi-processor. Chinese Journal of Computers, 2019, 42(11):2403-2416 (in Chinese)
(王子聪 陈小文 郭阳. 片上多核处理器Cache访问均衡性研究，计算机学报，2019, 42(11):2403-2416)

[9] Lin J, Xu Z, Cai L, et al. Evaluating the sw26010 many-core processor with a micro-benchmark suite for performance optimizations. Parallel Computing, 2018, 77:128-143

[10] McCalpin J D. Hpl and dgemm performance variability on the xeon platinum 8160 processor//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. Dallas, USA, 2018: 225-237.

[11] Liu N, Zang B, Chen H. No barrier in the road: a comprehensive study and optimization of arm barriers//Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. San Diego, USA, 2020: 348-361

[12] Hoefler T, Belli R. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results// Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. Austin, USA, 2015: 1-12

[13] Röhl T, Treibig J, Hager G, et al. Overhead analysis of performance counter measurements//Proceedings of the 2014 43rd International Conference on Parallel Processing Workshops. Minneapolis, USA. 2014: 176-185

[14] Terpstra D, Jagode H, You H, et al. Collecting performance data with papi-c//Proceedings of the Tools for High Performance Computing 2009. Dresden, Germany, 2010: 157-173

[15] Treibig J, Hager G, Wellein G. Likwid: A lightweight performance oriented tool suite for x86 multicore environments//Proceedings of the 39th International Conference on Parallel Processing Workshop. San Diego, USA, 2010: 207-216

[16] Chen T, Guo Q, Temam O, et al. Statistical performance comparisons of computers. IEEE Transactions on Computers, 2014, 64(5):1442-1455

[17] Tang X, Zhai J, Qian X, et al. vsensor: leveraging fixed-workload snippets of programs for performance variance detection//Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Montreal, Canada, 2018: 124-136

[18] Knüpfer A, Rössel C, Biersdorff S, et al. Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir//Proceedings of the Tools for High Performance Computing 2011. Dresden, Germany, 2012: 79-91

[19] Knüpfer A, Brunst H, Doleschal J, et al. The vampir performance analysis tool-set//Proceedings of the Tools for High Performance Computing: Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing. Stuttgart, Germany. 2008: 139-155

[20] Geimer M, Wolf F, Wylie B J, et al. The scalasca performance toolset architecture. Concurrency and Computation: Practice and Experience, 2010, 22(6):702-719

[21] Gleixner T, Molnar I. Linux kernel. GitHub repository, 2008. https://github.com/torvalds/linux/blob/master/tools/perf/design.txt

[22] Reinders J. Vtune performance analyzer essentials: volume 9. Santa Clara, USA,: Intel Press Santa Clara, 2005

[23] Adhianto L, Banerjee S, Fagan M, et al. Hpctoolkit: Tools for performance analysis of optimized parallel programs. Concurrency and Computation: Practice and Experience, 2010, 22(6):685-701

[24] Liu Y. Optimizing papi for low-overhead counter measurement [Master Thesis]. University of Maine, Maine, USA, 2017

[25] Liu Y., Weaver V. M. (2019). Enhancing PAPI with Low-Overhead rdpmc Reads//Bhatele, A., Boehme, D., Levine, J., Malony, A., Schulz, M (Eds.). Programming and Performance Visualization Tools, LNCS 11027. Dallas, USA: Springer, 2019: 16-33

[26] Weaver V M, Terpstra D, Moore S. Non-determinism and over count on modern hardware performance counter implementations//Proceedings of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Austin, USA, 2013: 215-224

[27] Abel A, Reineke J. nanobench: A low-overhead tool for running micro benchmarks on x86 systems//Proceedings of the 2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). 2020: 34-46

[28] Beckman P, Iskra K, Yoshii K, et al. The influence of operating systems on the performance of collective operations at extreme scale//Proceedings of the 2006 IEEE International Conference on Cluster Computing. Barcelona, Spain, 2006: 1-12

[29] Paoloni G. How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures. Intel Corporation, 2010, 123:170

[30] Intel. Intel 64 and IA-32 architectures software developer's manual: Volume 3 (3A, 3B, 3C & 3D): System programming guide. 81 ed. Santa Clara, USA: Intel Corporation, 2023

[31] Seal D. Arm architecture reference manual. Pearson Education, 2001

[32] Liu F T, Ting K M, Zhou Z H. Isolation forest//Proceedings of the 8th IEEE International Conference on Data Mining. Pisa, Italy. 2008: 413-422

[33] Mytkowicz T, Diwan A, Hauswirth M, et al. We have it easy, but do we have it right? //Proceedings of the 2008 IEEE International Symposium on Parallel and Distributed Processing. Seattle, USA, 2008: 1-7

[34] Mytkowicz T, Diwan A, Hauswirth M, et al. Producing wrong data without doing anything obviously wrong! ACM Sigplan Notices, 2009, 44(3):265-276

[35] Weaver V M, McKee S A. Can hardware performance counters be trusted? //Proceedings of the 2008 IEEE International Symposium on Workload Characterization. Seattle, USA, 2008: 141-150

[36] Weaver V, Dongarra J. Can hardware performance counters produce expected, deterministic results//Proceedings of Third Workshop on Functionality of Hardware Performance Monitoring. Atlanta, USA, 2010: 12-22

[37] Afzal A, Hager G, Wellein G. Analytic modeling of idle waves in parallel programs: communication, cluster topology, and noise impact//Proceedings of the International Conference on High Performance Computing. Online, 2021: 351-371

[38] Weisbach H, Gerofi B, Kocoloski B, et al. Hardware performance variation: a comparative study using lightweight kernels//Proceedings of the International Conference on High Performance Computing. Frankfurt, Germany, 2018: 246-265

[39] Dominguez-Trujillo J, Haskins K, Khouzani S J, et al. Lightweight measurement and analysis of hpc performance variability//Proceedings of the 2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS). Oregon, USA: IEEE, 2020: 50-60

[40] Kocoloski B, Lange J. Varbench: an experimental framework to measure and characterize performance variability//Proceedings of the 47th International Conference on Parallel Processing. Oregon, USA, 2018: 1-10

[41] Ates E, Zhang Y, Aksar B, et al. Hpas: An hpc performance anomaly suite for reproducing performance variations//Proceedings of the 48th International Conference on Parallel Processing. Kyoto, Japan, 2019: 1-10

[42] Kocoloski B. Scalability in the presence of variability [Ph. D. Thesis]. University of Pittsburgh, Pittsburgh, USA., 2018

[43] Xu L, Wang Y, Lux T, et al. Modeling i/o performance variability in high-performance computing systems using mixture distributions. Journal of Parallel and Distributed Computing, 2020, 139:87-98

[44] Mondragon O H, Bridges P G, Levy S, et al. Understanding performance interference in next-generation hpc systems//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. Salt Lake City, USA, 2016: 384-395

[45] Chang T H, Larson J, Watson L T. Multiobjective optimization of the variability of the high-performance linpack solver//Proceedings of the 2020 Winter Simulation Conference (WSC). Online, 2020: 3081-3092

LIAO Qiu-Cheng, B. S., assistant engineer. His research interest is high performance computing.

ZUO Si-Cheng, M. S. His research interest is high performance computing.

WANG Yi-Chao, M. S., senior engineer. His research interest is high performance computing.

LIN Xin-Hua, Ph. D., senior engineer. His research interest is high performance computing.

Background

Supercomputers are fundamental to modern computational science. Using millions of computing cores in synchrony, supercomputers provide significantly higher computing performance than normal servers. However, performance variability is a major factor hindering a supercomputer's full performance. This variability is characterized by inconsistencies in running times on the same hardware or by unexpected running times on identical hardware. Over the past decade, researchers have identified and studied various sources of performance variability in supercomputers.

This project studied processor performance variation, a harmful and insidious cause of performance variability. Variations in processor performance lead to performance degradation in many supercomputers on the TOP500 list. However, detecting performance variations is challenging due to the fast running speeds of processors, making it difficult to identify the root cause and understand how these variations propagate and couple. In detecting processor performance variations, high-precision timing was identified as the primary challenge. Current detection methods face two challenges: (1) Identifying tiny processor performance anomalies is difficult with existing profiling tools like PAPI; (2) Assessing the capabilities of different tools and methods for identifying processor performance anomalies.

To address these challenges, we conducted three studies in this paper: (1) We experimentally tested the PAPI timing method and found that PAPI struggles to detect performance variations due to its timing fluctuations; (2) We developed a serialized barrier timing method with low timing fluctuations, based on memory barrier instructions and serialized instructions. This new method can detect tiny anomalies in processor performance; (3) By building an analytical model of timing fluctuations, we proposed two metrics to evaluate timing methods: the least timing limit and the variation division index. We compared our serialized barrier timing method with the PAPI timing method on Intel Xeon 6248 and Huawei Kunpeng 920-6426 processors. The timing methods were assessed for precision and sensitivity using both metrics. According to the evaluation, the serialized barrier timing method is capable of detecting nanosecond-level performance variations on both of our experimental platforms when memory access is within the cache. Furthermore, compared to PAPI, our method was 2.2～30.2 times more precise and 1.9～44.8 times more sensitive.

This research is supported by an NSFC program (No. 62072300)