南開大學
Nankai University

计算机学院

计算机组成原理期末报告

性能评价工具集

刘新宇 2113384

许宸 2110598

杨万睿 2113447

冯思程 2112213

刘轩宇 2112487

专业：计算机科学与技术

指导老师：张金

2023 年 6 月 27 日

目录

1 概述

本次实验旨在使用自主设计的性能测试工具对鲲鹏服务器的各方面性能进行测试评估。在设计制作性能测试集时，我们结合 C++ 以及 python 两种语言中已有的测试框架，并使用索引压缩、快速傅里叶变换等算法作为测试工具，对服务器的性能做出全方位的测试。

2 设计思路

总体思路 我们的目标是对给定的鲲鹏服务器进行性能测试，并划定基准，这里我们考虑用给定的鲲鹏服务器本身作为基准，并根据测试结果实现其他的服务器性能测试目标，这里我们综合的、全面的、多方面的考虑一个服务器需要被重点关注的测试部分，如下：

- 1）服务器基础硬件参数
- 2）CPU 计算性能测试
- 3）并行性能测试
- 4）FPU 性能测试
- 5）内存与硬盘读写测试
- 6）总评分计算

细节实现 然后对于上述的每个大体部分都需要合适的、具体的程序来进行测试，每一部分所利用的测试分别如下：

- 1）服务器基础硬件参数测试：
  - CPU 测试
  - 主存测试
  - 硬盘测试
- 2）CPU 计算性能测试：
  - 整数计算性能测试（索引压缩）
  - 浮点数计算性能测试（FFT）
- 3）并行性能测试：
  - SIMD 性能测试（基于 FFT）
  - 多线程性能测试（基于 FFT）
- 4）FPU 性能测试：
  - FPU Julia 测试
  - FPU Mandel 测试
- 5）内存与硬盘读写测试：

  - 硬盘读写测试
  - 内存读写测试

- 6）总评分计算

3 测试集评估

3.1 服务器基础硬件参数

3.1.1 CPU

实验中使用的服务器 CPU 架构为 AArch64 架构，是一种 64 位 ARM 架构处理器，支持 64 位操作系统。

处理器为计算更为方便采用小端序的字节序，将较低较低的字节存储在内存的较低处。CPU 数量达到 96 个，系统中存在 2 个物理插槽将 96 个 CPU 核心分为两部分，每个物理插槽中存在 48 个 CPU 核心，使得计算性能进一步提升，并将 CPU 从 0-95 在在线列表中进行编号；每个 CPU 核心中线程数量为 1 个，在一套寄存器下只能执行一个线程。

系统采用一种多处理器系统架构 NUMA，共存在 4 个 NUMA 节点，每个节点可以访问不同的物理内存区域。

处理器最高主频达到了 2600Mhz，同时处理器最低主频为 200Mhz，较高的主频为服务器提供了充足的计算性能，每秒可以执行的虚拟指令达到了 200 万条。

处理器的一级数据缓存达到 6MB，同时一级指令缓存的大小也达到了 6MB，并且处理器的二级缓存和三级缓存大小分别达到了 48MB 和 96MB，较大的缓存大小使得在程序执行时 cache 的命中率大大提高，使得程序执行速度大大提高。

该处理器还有着较为完备的异常处理系统，处理器对于 L1tf 漏洞、Itlb multihit 漏洞、MDS 漏洞、Meltdown 漏洞、Speculative Store Bypass 漏洞、Spectre V2 漏洞、SRBDS 漏洞、TSX 异步中止漏洞能够免受其危害，同时对于 Spectre V1 漏洞处理器采取了 user pointer sanitization 的处理措施进行缓解。

该处理器支持较多特性与指令集的使用，其中包括浮点运算、ASIMD、事件流跟踪、AES 加密算法、多项式乘法、SHA1 和 SHA2 哈希算法、CRC32、原子操作等多种操作与特性。

表 1: CPU 信息表

Architecture:	aarch64
CPU op-mode(s):	64-bit
Byte Order:	Little Endian
CPU(s):	96
On-line CPU(s) list:	0-95
Thread(s) per core:	1
Core(s) per socket:	48
Socket(s):	2
NUMA node(s):	4
Vendor ID:	0x48
Model:	0
Stepping:	0x1
CPU max MHz:	2600.0000
CPU min MHz:	200.0000
BogoMIPS:	200.00
L1d cache:	6 MiB
L1i cache:	6 MiB
L2 cache:	48 MiB
L3 cache:	96 MiB
NUMA node0 CPU(s):	0-23
NUMA node1 CPU(s):	24-47
NUMA node2 CPU(s):	48-71
NUMA node3 CPU(s):	72-95
Vulnerability Itlb multihit:	Not affected
Vulnerability L1tf:	Not affected
Vulnerability Mds:	Not affected
Vulnerability Meltdown:	Not affected
Vulnerability Spec store bypass:	Not affected
Vulnerability Spectre v1:	Mitigation; __user pointer sanitization
Vulnerability Spectre v2:	Not affected
Vulnerability Srbds:	Not affected
Vulnerability Tsx async abort:	Not affected
Flags:	fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhp cpuid asimdrdm jscvt fcma dcpop asimddp asimdfhm

Listing 1: 获取 CPU 信息代码

```
import subprocess

# 执行shell命令获取CPU信息
result = subprocess.run(['lscpu'], capture_output=True, text=True)
output = result.stdout

# 输出CPU信息
print(output)
```

3.1.2 主存

主存基准同样采用服务器作为基准，存储区域总大小达到 188G，其中 1.4G 已使用，未使用存储大小为 183G。

共享内存大小达到了 3MB，共享内存较大，使得在进行并行化程序执行时对于线程间的通信有了较好的提升，加速了程序执行效率。

同时缓存区的大小达到了 3.5G，使得缓存足够大，使得命中率提升，进一步提升程序执行效率。

可利用的内存大小一共为 185G。

交换区的大小为 8.0G，使得在程序运行所需内存过大时能够及时在交换区进行数据恢复等操作，使得在处理大型程序时性能进一步提升。

表 2: 主存信息表

Type	total	used	free	shared	buff/cache	available
MEM	188Gi	1.4Gi	183Gi	3.0Mi	3.5Gi	185Gi
SWAP	8.0Gi	0B	8.0Gi			

Listing 2: 获取内存信息代码

```
import subprocess

# 执行shell命令获取内存信息
result = subprocess.run(['free', '-h'], capture_output=True, text=True)
output = result.stdout

# 输出内存信息
print(output)
```

3.1.3 硬盘

首先对文件系统的挂载点进行说明：文件系统被挂载的位置，例如，”/” 是根文件系统，”/boot” 是引导分区，”/dev/shm” 是用于共享内存的临时文件系统。

下表的数据提供了关于不同挂载点（文件系统被挂载的位置）的磁盘使用情况的详细信息，包括总容量（Size）、已使用空间（Used）、可用空间（Avail）、使用率（Use%）

现在，逐个分析关键挂载点的信息：

”/dev/mapper/ubuntu–vg-ubuntu–lv” 是根文件系统，总容量为 196G，已使用 20G，可用空间 166G，使用率为 11%。

”/boot” 是引导分区，总容量为 976M，已使用 203M，可用空间 707M，使用率为 23%。

”/boot/efi” 是 EFI 分区，总容量为 511M，已使用 3.5M，可用空间 508M，使用率为 1%。

”/snap/core18/1949” 是一个循环设备的挂载点，总容量为 49M，已使用 49M，可用空间为 0，使用率为 100%。同样，”/snap/core18/2002”、”/snap/lxd/19206”、”/snap/snapd/10709”、”/snap/lxd/19648” 和”/snap/snapd/11584” 都是循环设备挂载点，使用率均为 100%。

”/dev” 是一个临时文件系统，总容量为 95G，已使用 0，可用空间 95G，使用率为 0%。

”/dev/shm”、”/run/lock”、”/sys/fs/cgroup”、”/run/user/1314”、”/run/user/1267”、”/run/user/1214” 和”/run/user/1315” 都是临时文件系统挂载点，使用率均为 0%。

”/run” 也是一个临时文件系统的挂载点，总容量为 19G，已使用 3.8M，可用空间 19G，使用率是 1%。

表 3: 硬盘信息表

Filesystem	Size	Used	Avail	Use%	Mounted on
udev	95G	0	95G	0%	/dev
tmpfs	19G	3.8M	19G	1%	/run
/dev/mapper/ubuntu–vg-ubuntu–lv	196G	20G	166G	11%	/
tmpfs	95G	0	95G	0%	/dev/shm
tmpfs	5.0M	0	5.0M	0%	/run/lock
tmpfs	95G	0	95G	0%	/sys/fs/cgroup
/dev/sda2	976M	203M	707M	23%	/boot
/dev/loop0	49M	49M	0	100%	/snap/core18/1949
/dev/loop1	49M	49M	0	100%	/snap/core18/2002
/dev/loop2	62M	62M	0	100%	/snap/lxd/19206
/dev/sda1	511M	3.5M	508M	1%	/boot/efi
/dev/loop4	27M	27M	0	100%	/snap/snapd/10709
/dev/loop3	63M	63M	0	100%	/snap/lxd/19648
/dev/loop5	29M	29M	0	100%	/snap/snapd/11584
tmpfs	19G	0	19G	0%	/run/user/1314
tmpfs	19G	0	19G	0%	/run/user/1267
tmpfs	19G	0	19G	0%	/run/user/1214
tmpfs	19G	0	19G	0%	/run/user/1315

Listing 3: 获取硬盘信息代码

```
import subprocess

# 执行shell命令获取硬盘信息
result = subprocess.run(['df', '-h'], capture_output=True, text=True)
output = result.stdout

# 输出硬盘信息
print(output)
```

3.2 CPU 计算性能测试

使用 C++ 微秒级计时工具库 chrono 来进行运算时间的测试，在测试中，我们使用固定数据规模进行测试，将程序运行时间记录到 txt 文件中，以便于未来进行评分。

3.2.1 整数计算性能测试：索引压缩

FOR (Frequency-Oriented Reordering) 算法是一种常用的倒排索引压缩算法，其基本思想是将文档 ID 和位置信息分别存储，并按照文档 ID 的频率对倒排列表进行重排序。具体来说，FOR 算法将倒排列表分成多个块，每个块包含一组文档 ID 相同的项，每个块以文档 ID 出现的频率为关键字进行排序。在存储每个块时，它将文档 ID 编码为与前一个文档 ID 的差异，并将位置信息编码为与前一个位置的差异。

FOR 算法的主要优点是它能够有效地利用文档 ID 的重复性，从而实现更高的压缩率。但是，由于它需要对倒排列表进行排序，因此其压缩时间较长，不适用于实时压缩。此外，由于 FOR 算法会将倒排列表划分为多个块，因此在查询时需要进行多次合并操作，从而导致查询性能下降。

FOR 算法的具体步骤如下：

对于输入的数组，计算数组元素值为与前一位的差值：V(n) = V(n) - V(n-1)，其中 n=2,3,4... 第一位不变

计算数组中最大值所需占用的大小，即用二进制表示最大值所需的位数。

根据最大值所需占用的大小，计算数组是否需要拆分，如果需要拆分，则计算拆分后每组的最大值所需占用的大小并记录。

将原始数据压缩成差分数组和索引数组。差分数组中存储的是相邻元素之间的差值，索引数组中存储的是拆分后的每个子数组的最后一个元素在原数组中的位置。

对于差分数组和索引数组，分别采用不同的压缩方法进行压缩。

最后将压缩后的差分数组和索引数组合并成一个压缩后的数组。

Listing 4: 索引压缩代码

```
vector<uint32_t> result;
int first, second;
first = 18;
second = 19;
for (int i = 0; i < resodds[first].size(); i++)
{
    uint32_t cur = resodds[first][i];
    for (int j = 0; j < resodds[second].size(); j++) {
        if (cur == resodds[second][j])
        {
            result.push_back(cur);
            break;
        }
    }
}
```

3.2.2 浮点数计算性能测试：快速傅里叶变换

FFT 是一种常见的数值计算算法，对于浮点数性能的评估具有一定的代表性。在实际的应用中，FFT 中的数据往往为由浮点数实部和虚部构成的虚数，会涉及到大量的浮点数计算，因此对于基础的浮点数计算性能测试，我们采用 FFT 作为评估方法。核心计算代码如下：

Listing 5: FFT 算法核心代码

```
void fft2(vector<complex<double> >& input)
{
  int n = input.size();

  // 数据重排
  for (int i = 1, j = 0; i < n; i++)
  {
    int bit = n >> 1;
    for (; j >= bit; bit >>= 1) j -= bit;
    j += bit;
    if (i < j) swap(input[i], input[j]);
  }
```

```
    // 蝴蝶运算
    for (int k = 2; k <= n; k <<= 1)
    {
        int m = k >> 1;
        complex<double> w_m(cos(PI / m), -sin(PI / m));

        for (int i = 0; i < n; i += k)
        {
            complex<double> w(1);
            for (int j = 0; j < m; j++)
            {
                complex<double> t = w * input[i + j + m];
                input[i + j + m] = input[i + j] - t;
                input[i + j] += t;
                w *= w_m;
            }
        }
    }
}
```

3.3 并行性能测试

并行是提高计算能力的重要方式，通过同时处理多个任务或并发请求，系统可以更高效地执行计算任务。在性能测试中，评估系统的并行计算能力对于了解其计算性能至关重要。

为了测试系统的并行计算能力，主要使用 SIMD 和 OpenMP。SIMD（单指令多数据）是一种计算机体系结构概念，可以实现并行处理并提高计算效率。在 SIMD 中，单个指令同时对多个数据元素进行执行。它旨在在单个时钟周期内对多个数据点执行相同的操作，从而实现并行性并加快数据并行任务的计算速度。在 SIMD 系统中，一组数据元素（称为向量）通过将单个指令应用于所有元素来进行处理。这与标量处理相反，标量处理中每个数据元素都使用单独的指令进行处理。SIMD 指令可以在打包到矢量寄存器中的多个数据元素上操作，实现高效的加法、减法、乘法和除法等操作。

OpenMP(Open Multi-Processing)是一种并行编程接口,常用于共享内存系统。通过使用 OpenMP，可以评估 CPU 的多线程计算能力。测试中，使用并行化的整数和浮点数运算代码，并使用 OpenMP 来实现并行执行。通过测量并行计算任务的执行时间和性能指标，评估系统在多线程计算方面的表现。

3.3.1 SIMD 性能测试：快速傅里叶变换

测试计算机使用 SIMD 指令进行并行化计算 FFT 的运行时间测试计算机 SIMD 程序性能，核心计算代码如下：

Listing 6: SIMD 算法核心代码

```
void fft_Neon(COMPLEX* X, int N)
{
    if (N < 2)
    {
        // bottom of recursion.
        // Do nothing here, because already X[0] = x[0]
```

```
    }
    else
    {
        if (N < 2)
        {
            // bottom of recursion.
            // Do nothing here, because already X[0] = x[0]
        }
        else
        {
            separate(X, N);
            fft_Neon(X, N / 2);
            fft_Neon(X + N / 2, N / 2);
            for (int k = 0; k < N / 2; k++)
            {
                double coss = cos(-2. * PI * k / N);
                double sinn = sin(-2. * PI * k / N);
                float64x2_t even1, odd1;
                float64x2_t cos_2vec = { coss, coss };
                float64x2_t sin_2vec = { sinn, sinn };
                float64x2_t zero = { 0.0, 0.0 };
                double* ptr1 = (double*)&X[k];
                double* ptr2 = (double*)&X[k + N / 2];
                even1 = vld1q_f64(ptr1);
                odd1 = vld1q_f64(ptr2);
                float64x2_t intew1 = vmulq_f64(cos_2vec, odd1);
                float64x2_t intew2 = vmulq_f64(sin_2vec, odd1);
                float64x2_t intew2_neg = vnegq_f64(intew2);
                float64x2_t sf_intew2 = vextq_f64(intew2, intew2_neg, 1);
                float64x2_t real_o = vaddq_f64(intew1, sf_intew2);
                float64x2_t result_e = vaddq_f64(even1, real_o);
                float64x2_t _result_o = vsubq_f64(even1, real_o);
                vst1q_f64((double*)&X[k], result_e);
                vst1q_f64((double*)&X[k + N / 2], _result_o);
            }
        }
    }
}
```

3.3.2 多线程性能测试：快速傅里叶变换

测试计算机使用 OPENMP 进行多线程并行化计算 FFT 的运行时间测试计算机多线程程序性能，核心计算代码如下：

Listing 7: 多线程算法核心代码

```
void fft2_omp(vector<complex<double>>& input)
{
  int n = input.size();
```

```
  // 数据重排
  for (int i = 1, j = 0; i < n; i++)
  {
    int bit = n >> 1;
    for (; j >= bit; bit >>= 1) j -= bit;
    j += bit;
    if (i < j) swap(input[i], input[j]);
  }
  // 蝴蝶运算
  for (int k = 2; k <= n; k <<= 1)
  {
    int m = k >> 1;
    complex<double> w_m(cos(PI / m), -sin(PI / m));
#pragma omp parallel for num_threads(4)
    for (int i = 0; i < n; i += k)
    {
      complex<double> w(1);
      for (int j = 0; j < m; j++)
      {
        complex<double> t = w * input[i + j + m];
        input[i + j + m] = input[i + j] - t;
        input[i + j] += t;
        w *= w_m;
      }
    }
  }
}
```

3.4 FPU 性能测试

介绍 FPU 是 CPU 中浮点运算单元的简称，它通常是作为处理器的一部分集成在其中的，它拥有自己的指令集和寄存器。当计算机执行需要进行浮点数运算的指令时，这些指令被发送到 FPU 中进行处理。FPU 能够高效地执行各种浮点数运算，并提供高精度的计算结果。

FPU 的引入使得计算机能够处理更加复杂和精确的数学运算，广泛应用于科学计算、图形处理、工程领域以及需要高精度计算的应用程序中。它在计算机性能的提升和科学技术的发展中发挥着重要的作用。

测试方法 FPU 算力的高速稳定发挥，很大程度上决定了电脑的性能。因此我们针对 FPU 进行浮点运算的进一步测试，主要采用 Julia 和 Mandel 这两种测试方法。

3.4.1 FPU Julia 测试

Julia 集合是复平面上的一种数学集合，它是在复数平面上生成分形图像的一种方法。计算 Julia 集合需要进行复数运算，包括复数乘法、加法、幂等运算等。该测试侧重于评估计算机在复数运算上的性能。

Listing 8: 对于 FPU 的 Julia 测试

```
int fpuJulia(float cx, float cy, int maxIterations) {
    float zx = 0.0f;
    float zy = 0.0f;
    for (int i = 0; i < maxIterations; i++) {
        float new_zx = zx * zx - zy * zy + cx;
        float new_zy = 2.0f * zx * zy + cy;
        if (std::sqrt(new_zx * new_zx + new_zy * new_zy) > 2.0f) {
            return i;
        }
        zx = new_zx;
        zy = new_zy;
    }
    return maxIterations;
}
```

3.4.2 FPU Mandel 测试

Mandel 是另一种浮点运算测试方法，Mandelbrot 集合是复平面上的一种数学集合，它也是在复数平面上生成分形图像的一种方法。计算 Mandelbrot 集合需要进行复数运算，包括复数乘法、加法、幂等运算。

Listing 9: 对于 FPU 的 Mandel 测试

```
int mandelbrot(double real, double imag, int maxIterations) {
    std::complex<double> c(real, imag);
    std::complex<double> z(0.0, 0.0);

    for (int i = 0; i < maxIterations; i++) {
        z = z * z + c;

        if (std::abs(z) > 2.0) {
            return i;
        }
    }
    return maxIterations;
}
```

Julia 和 Mandel 两种测试方法类似，都是常用的测试计算机浮点运算性能的方法，Mandelbrot 集合和 Julia 集合的计算算法有所不同。Mandelbrot 集合的计算是通过迭代计算每个像素点的收敛性来确定其属于集合还是逃逸出集合，而 Julia 集合的计算则是通过在复平面上对每个像素点进行复数运算来生成图像。在测试浮点运算性能时，可以两者相结合得到综合结果。

3.5 内存与硬盘读写测试

3.5.1 硬盘读写

要测试硬盘读写的速度，使用 python 程序进行简单的测试，下面只给出硬盘写入的代码，硬盘读取的类似，都是对硬盘上的文件进行操作即可。

Listing 10: 硬盘读写性能测试

```python
def write_test_disk(file_path, size_mb):
    """
    执行写入测试，将指定大小的随机数据写入文件
    展示写入文件的代码，读取的类似
    """
    # 生成随机数据
    data = os.urandom(size_mb * 1024 * 1024)

    # 写入文件
    start_time = time.time()
    with open(file_path, 'wb') as file:
        file.write(data)
    end_time = time.time()

    # 计算写入时间
    write_time = end_time - start_time
    print(f"写入测试完成，写入时间: {write_time:.4f} 秒")
```

3.5.2 内存读写

内存读写性能的测试思路与硬盘读写的测试思路类似，只需要将写入硬盘文件的操作更换为写入内存即可。

Listing 11: 硬盘读写性能测试

```python
def write_test_mem(size_mb):
    """
    执行写入测试，将指定大小的随机数据写入内存
    仅展示写入
    """
    # 生成随机数据
    data = np.random.bytes(size_mb * 1024 * 1024)

    # 写入内存
    start_time = time.time()
    memory = np.frombuffer(data, dtype=np.byte)
    end_time = time.time()

    # 计算写入时间
    write_time = end_time - start_time
    print(f"写入测试完成，写入时间: {write_time:.4f} 秒")
```

4 基准测试

工具集就以鲲鹏服务器为性能基准，在鲲鹏服务器上进行多轮各项测试，记录各项程序的平均运行时间作为基准时间。当要测试其他计算机某一方面性能时，就以以下公式作为参照：

$$Performance\ of\ the\ computer\ in\ a\ certain\ aspect = \frac{Kunpeng\ specified\ aspect\ running\ time}{The\ current\ test\ computer}$$

涉及到当前计算机总的评分时，我们以当前计算机各方面加权几何平均值作为总评分参考，公式如下：

$$Overall\ computer\ performance = \sqrt[n]{weight_i \cdot x_1 \cdot x_2 \cdot \ldots \cdot x_n}$$

其中 $x_1x_2...x_n$ 为计算机单一方面性能测试结果，$weight_i$ 为计算机单一方面性能测试权重。

5 测试结果

表 4: 测试结果表

测试名称	测试时间	每秒计算次数
索引压缩	566150 微秒	-
FFT	5839056340 纳秒	-
FFT simd(1024) 测试	5.98853 毫秒	-
FFT mpi(8192) 测试	0.210438 毫秒	-
FPU Julia 测试	6.50014 秒	1.53843e+06 次
FPU Mandel 测试	1.7e-07 秒	5.88235e+11 次
内存读取	0.028 秒	-
内存写入	0.001 秒	-
硬盘读取	0.035 秒	-
硬盘写入	0.054 秒	-

6 总结

在本学期的计算机组成原理课程中，通过深入学习计算机的各个组件和原理，我们成功地实现了一个性能测试工具。该工具旨在全面测试计算机的各个部件，并以鲲鹏服务器作为测试基准。

这个性能测试工具通过利用所学知识，对计算机的硬盘、内存、处理器等关键部件进行全面的测试。它使用了精心设计的测试算法和方法，以确保准确评估计算机的性能表现。

针对硬盘性能测试，实现了一个功能强大的测试模块，用于测量硬盘的读写速度、响应时间和吞吐量。该模块通过创建临时文件、模拟随机数据的写入和读取，并使用计时工具来记录测试结果。通过多次运行测试并取平均值，我能够获得更准确的硬盘性能数据。

在内存性能测试方面，设计了一组测试用例，用于评估计算机的内存读写速度和数据处理能力。我使用随机数据生成器创建大规模数据集，并利用内存的高速读写特性来测量内存的性能。

另外，针对处理器性能测试，编写了一系列基准测试程序，包括计算密集型任务和并行处理任务。这些测试程序利用多线程和并行计算技术，对处理器的运算能力、缓存性能和多任务处理能力进行评估。通过测量执行时间和计算结果的准确性，我能够得出关于处理器性能的详细结论。

通过以上测试模块的集成，我们成功地构建了一个全面的性能测试工具，可以对计算机的各个部件进行准确评估。这个工具不仅可以对鲲鹏服务器进行性能测试，也可以应用于其他计算机系统的性

能分析。它为计算机性能优化和硬件选择提供了有力的支持，使用户能够更好地了解和利用计算机系统的潜力。

7 参考资料

- 计算机组成与设计：硬件软件接口，David A.Patterson、John L.Hennessy，2015