内存对性能影响实验文档

王士博

wangshibo@shu.edu.cn

2024 年 3 月 30 日

1 实验环境和软件

1. 宿主机配置：Windows11，Intel(R) Core(TM) i7-11800H CPU，32.0 GB RAM，NVIDIA GeForce RTX 3070 GPU；
2. 实验环境：docker 镜像服从下面的 dockerfile 配置；

```
FROM ubuntu:22.04

ENV DEBIAN_FRONTEND=noninteractive
RUN apt -y update && apt -y upgrade && \
    apt -y install build-essential git m4 scons zlib1g zlib1g-dev \
    libprotobuf-dev protobuf-compiler libprotoc-dev libgoogle-perftools-
       dev \
    python3-dev doxygen libboost-all-dev libhdf5-serial-dev python3-
       pydot \
    libpng-dev libelf-dev pkg-config pip python3-venv black

RUN pip install mypy pre-commit
```

3. 仿真软件：Gem5[1]；

2 实验内容

2.1 Gem5 简介和安装

Gem5 对环境的要求较为严苛，在 GitHub 仓库的 Issue 中很大一部分是对编译失败的问题的讨论。所以这里展示使用 docker 镜像来安装 Gem5 的过程。由于仓库中的 Ubuntu20.04 镜像链接失效，这里使用 Ubuntu22.04 镜像安装。镜像地址为`ghcr.io/gem5/ubuntu-22.04_all-dependencies:v23-0`，dockerfile 如实验环境 2所示，使用如下的命令来下拉和创建实例，并且在宿主机使用 git 下拉 Gem5 的源码。

```
docker pull ghcr.io/gem5/ubuntu-22.04_all-dependencies:v23-0
#docker build -t your_image_name:tag . for dockerfile
git clone https://github.com/gem5/gem5
docker run --volume gem5:/gem5 -it ghcr.io/gem5/ubuntu-22.04_all-
   dependencies:v23-0
```

进入虚拟终端之后进入 gem5 目录之后，执行如下的命令使用 scons 构建 gem5，将线程数设置为宿主机的线程数，在后续的构建中可能 CPU 会长期占用 100%，但是如果线程数过少构建缓慢，实测 16 线程需要 30min 左右。

```
cd /your/path/to/gem5
python3 `which scons` /build/X86/gem5.opt -j$(nproc)
>>scons:successfully build target
```

2.2 构建模拟器

构建一个 Gem5 的模拟系统，需要下面几个部件。第一个是 CPU 及其相关的配置如下：

```
system = System()
system.clk_domain = SrcClockDomain()
if not options or not options.clk:
    system.clk_domain.clock = '1GHz'
else:
    system.clk_domain.clock = options.clk
system.clk_domain.voltage_domain = VoltageDomain()
```

第二个是内存的配置，这里使用了 DDR3_1600_8x8 的配置，如下所示：

```
system.mem_mode = 'timing'
if not options or not options.mem_size:
    system.mem_ranges = [AddrRange('512MB')]
else:
    system.mem_ranges = [AddrRange(options.mem_size)]
```

第三个需要配置的是 Cache，这里使用了 L1 和 L2 的 Cache，这里的 Cache 需要继承自 Cache 基类自己进行细节的配置，如下所示：

```
system.cpu = TimingSimpleCPU()
system.cpu.icache = L1ICache(options)
system.cpu.dcache = L1DCache(options)
system.cpu.icache.connectCPU(system.cpu)
system.cpu.dcache.connectCPU(system.cpu)
system.l2bus = L2XBar()
system.cpu.icache.connectBus(system.l2bus)
system.cpu.dcache.connectBus(system.l2bus)
system.l2cache = L2Cache(options)
system.l2cache.connectCPUSideBus(system.l2bus)
system.membus = SystemXBar()
system.l2cache.connectMemSideBus(system.membus)
if not options or not options.cache_block:
    system.cache_line_size = 64
else:
    system.cache_line_size = options.cache_block
```

第四步需要配置的是内存控制器，这里使用了 DDR3_1600_8x8 的配置，如下所示：

```
system.mem_ctrl = MemCtrl()
system.mem_ctrl.dram = DDR3_1600_8x8()
system.mem_ctrl.dram.range = system.mem_ranges[0]
system.mem_ctrl.port = system.membus.mem_side_ports
```

最后可以指定一个测试程序进行测试，这个程序一定是静态编译之后的结果，比如 test.o 文件。下面是这个程序的一个使用实例。其中需要保证测试文件要在 gem5/tests/test_progs 的目录之下以保证可以正常识别，并且上面的配置文件需要放在 gem5/configs 目录之下。在执行测试时使用构建出的/build/X86/gem5.opt 二进制可执行文件执行传入参数的配置文件进行测试。下面是一个测试的完整流程。

```
vim tests/test-progs/bandwidth_test.c
gcc -static tests/test-progs/bandwidth_test.c -o tests/test-progs/
    bandwidth_test -lm
/build/X86/gem5.opt configs/exps/cached_cpu.py --clk='1GHz' --
    l1i_size='16kB' --l1d_size='128kB' --cache_block=16 --l2_size='512
    kB' --test_dir='tests/test-progs/bandwidth_test' --mem_size='512MB
    '
```

通过执行/build/X86/gem5.opt configs/exps/cached_cpu.py --help 来查看参数情况。

```
>>Usage: cached_cpu.py [options]
  Options:
    -h, --help            show this help message and exit
    --clk=CLK             CPU clk. Default: 1GHz
    --cache_block=CACHE_BLOCK
                          CPU cache block. Default: 64
    --l1i_size=L1I_SIZE   L1 instruction cache size. Default: 16kB.
    --l1d_size=L1D_SIZE   L1 data cache size. Default: Default: 64kB.
    --l2_size=L2_SIZE     L2 cache size. Default: 256kB.
    --mem_size=MEM_SIZE   Memory size. Default: 512MB.
    --test_dir=TEST_DIR   Test directory. Default: tests/test-
                          progs/hello/bin/x86/linux/hello
    --ddr4=DDR4           DDR4 with 2400Mhz. Default:DDR3 1600Mhz
```

2.3 实验程序

1. 矩阵乘法运算来测试缓存和缓存行对性能的影响：

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <time.h>
    #define N 128
    void matrix_multiply(double **a, double **b, double **result) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                result[i][j] = 0.0;
                for (int k = 0; k < N; k++) {
```

```
                result[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
int main() {
    double **a = (double **)malloc(N * sizeof(double *));
    double **b = (double **)malloc(N * sizeof(double *));
    double **result = (double **)malloc(N * sizeof(double *));
    for (int i = 0; i < N; i++) {
        a[i] = (double *)malloc(N * sizeof(double));
        b[i] = (double *)malloc(N * sizeof(double));
        result[i] = (double *)malloc(N * sizeof(double));
    }
    srand(time(NULL));
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            a[i][j] = rand() / (double)RAND_MAX;
            b[i][j] = rand() / (double)RAND_MAX;
        }
    }
    matrix_multiply(a, b, result);
    printf("Matrix multiplication completed.\n");
    for (int i = 0; i < N; i++) {
        free(a[i]);
        free(b[i]);
        free(result[i]);
    }
    free(a);
    free(b);
    free(result);
    return 0;
}
```

2. 使用内存开辟和数组遍历来测试内存大小对性能的影响，这个程序会申请比主存更多的内存空间，这会迫使进行主存和外存的交换，将主存的内容移动到外存上，并且之后进行遍历，迫使程序从外存上读取数据：

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 80000000
int main() {
    long long *largeArray = malloc(SIZE * sizeof(long long));
    if (largeArray == NULL) {
        printf("Failed to allocate memory.\n");
```

```
            return 1;
        }
        for (long long i = 0; i < SIZE; i++) {
            largeArray[i] = i;
        }
        printf("Large dataset manipulation completed.\n");
        free(largeArray);
        return 0;
    }
```

3. 使用加法程序测试内存频率/带宽对性能的影响，因为 CPU 的加法运算非常快，而且 CPU 的时钟频率远高于内存的频率所以在 CPU 频率较高并且没有 Cache 的情况下，这个程序的瓶颈在于主存的频率（主存读取数据的快慢）：

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <time.h>
    #define ARRAY_SIZE (1024 * 1024 * 64)
    double calculateSum(double *array, int size) {
        double sum = 0.0;
        for (int i = 0; i < size; ++i) {
            sum += array[i];
        }
        return sum;
    }
    int main() {
        double *array = (double *)malloc(ARRAY_SIZE * sizeof(double));
        if (array == NULL) {
            fprintf(stderr, "Memory allocation failed.\n");
            return 1;
        }
        srand(time(NULL));
        for (int i = 0; i < ARRAY_SIZE; ++i) {
            array[i] = (double)rand() / RAND_MAX * 100;
        }
        clock_t start = clock();
        double sum = calculateSum(array, ARRAY_SIZE);
        clock_t end = clock();
        printf("Array sum: %f\n", sum);
        printf("Time taken: %f seconds\n", (double)(end - start) /
           CLOCKS_PER_SEC);
        free(array);
        return 0;
    }
```

3 实验结果

1. 矩阵乘法运算实验结果如下：

配置编号	主存大小	L1 Data Cache	L1 Instruction Cache	L2 Cache	Cache Blocks	主频率 (MHz)	运行时间 (s)	Cache 命中次数	CPI (周期/指令)
1	512MB	128KB	32KB	512KB	8	1600	0.43	47123934	4.09
2	512MB	128KB	32KB	512KB	16	1600	0.40	48365326	3.85
3	512MB	256KB	32KB	512KB	8	1600	0.43	47192301	4.10
4	512MB	128KB	32KB	512KB	8	2400	0.43	47023931	4.09
5	512MB	128KB	32KB	1024KB	8	1600	0.43	47343133	4.09
6	512MB	256KB	32KB	512KB	8	1600	0.43	47239412	4.09
7	512MB	128KB	16KB	512KB	8	1600	0.43	47297938	4.09
8	512MB	128KB	32KB	512KB	32	1600	0.38	48817136	3.71
9	512MB	16KB	32KB	16KB	8	1600	0.60	47173460	5.71

表 1: Cache 测试

对于矩阵乘法程序而言，以下配置参数对性能影响显著：

(a) **L1** 数据缓存（**L1 Data Cache**）：程序对 L1 数据缓存大小非常敏感。配置 9 表明了 L1 数据缓存小（16KB）时，执行时间长且 CPI 高，性能下降显著。

(b) 缓存块数（**Cache Blocks**）：较高的缓存块数可以改善性能。配置 2 与配置 8 虽然 L1 数据缓存大小相同，但更多的缓存块数（分别为 16 和 32）带来了更低的 CPI。

(c) **L2** 缓存（**L2 Cache**）：L2 缓存大小对性能有正面影响。配置 5 相对配置 1 的 L2 缓存加倍，使得性能得到了显著提升。

(d) 主频（**MHz**）：仅增加主频并不足以显著提高性能。配置 4 的高主频（2400MHz）并未带来预期的性能提升，这表明缓存效率更为关键。

综上所述，矩阵乘法程序对缓存参数非常敏感，适当增加缓存大小和块数可以显著提高程序执行效率。

2. 内存大小对性能的影响如下：

配置编号	主存大小 (MB)	数组长度（10^7）	运行时间 (s)
1	512MB	8	6.00
2	1024MB	8	4.12
3	512MB	4	2.01
4	512MB	4	2.04
5	512MB	12	15.12

表 2: 内存大小测试

表2展示了不同主存大小对强制内存外存交换程序性能的影响。以下为分析结果：

(a) 主存大小（**Memory Size**）：主存大小的增加能显著减少执行时间，如配置 2 相比配置 1，在故障次数相同的情况下，主存增加一倍，执行时间从 6.00 秒减少到 4.12 秒。

(b) 故障次数（**Faults**）：故障次数的减少直接影响执行时间的缩短。配置 3 和配置 4 在主存大小相同的情况下，故障次数减少，执行时间由 6.00 秒显著减少到约 2.02 秒。

(c) 执行时间（**Execution Time**）：当故障次数显著增加时，执行时间会大幅上升，如配置 5 的故障次数是配置 1 的 1.5 倍，导致执行时间从 6.00 秒增加到 15.12 秒。

配置编号	CPU 频率 (Ghz)	主存频率 (Mhz)	运行时间 (s)
1	1	DDR3/1600	1.02
2	1	DDR4/2400	0.71
3	3	DDR3/1600	1.01
4	3	DDR4/2400	0.61

表 3: 内存频率测试

3. 内存频率对性能的影响如下，对于此程序采用无缓存的单一 CPU 和主存的结构进行测试：

   在大数组求和任务中，CPU 与主存频率对性能有显著影响。表 3 提供了不同配置下的执行时间。以下为详细分析：

   (a) **CPU** 频率（**GHz**）：对于 CPU 密集型任务，理论上 CPU 频率越高，执行速度越快。然而，配置 1 和配置 3 的比较显示，即使 CPU 频率提高了两倍，执行时间的改进并不明显（1.02 秒对比 1.01 秒）。

   (b) 主存频率（**MHz**）：配置 1 和配置 2 的比较表明，提高主存频率可以显著减少执行时间（从 1.02 秒减少到 0.71 秒），突显了数据传输速率在大数组求和程序中的重要性。

   (c) 执行时间（**s**）：配置 4 提供了最短的执行时间（0.61 秒），显示了同时提高 CPU 和主存频率可以协同工作，以达到最佳的程序性能。

4 实验结论

1. 主存大小

   - 敏感应用及分析：大型数据库和虚拟机环境特别敏感于主存大小。大型数据库需要将大量数据加载到内存中以提高查询效率，而虚拟机环境中，每个虚拟机分配的内存越多，能够并行运行的虚拟机数量就越多，系统性能也越好。
   - 影响：不足的主存大小会导致系统频繁地使用磁盘空间作为虚拟内存，极大地增加了数据访问的延迟，降低了系统的整体性能。
   - 瓶颈原因：主存不足时，系统无法将所有需要的数据和程序同时加载到内存中，必须从较慢的磁盘存储中调入数据，这增加了处理时间。
   - 时间浪费：在频繁的数据交换过程中，大量时间花费在等待磁盘 I/O 操作上，CPU 在这段时间内可能处于闲置状态，导致资源浪费。

2. 主存频率（主存带宽）

   - 敏感应用及分析：视频编辑、3D 渲染和科学计算应用对主存频率特别敏感。这些应用需要快速处理和传输大量数据，高带宽可以显著提升这些应用的执行效率。
   - 影响：如果主存频率过低，即使 CPU 有高速的处理能力，数据传输的瓶颈也会限制整体性能，导致处理速度下降。
   - 瓶颈原因：主存频率低意味着数据在内存和 CPU 之间的传输速度慢，处理器需要等待数据，无法充分利用其计算能力。
   - 时间浪费：处理器在等待内存数据时的闲置时间增加，导致处理效率降低。

3. 缓存大小

- 敏感应用及分析：高性能计算（HPC）和大型在线事务处理（OLTP）系统对缓存大小极为敏感。这些系统需要频繁访问大量数据，较大的缓存可以减少对慢速主存的访问次数，提高处理速度。
- 影响：缓存大小不足会导致高缓存失效率，增加对主存的访问需求，从而增加数据访问延迟，降低系统性能。
- 瓶颈原因：当缓存无法容纳足够的数据以供 CPU 快速访问时，必须从相对较慢的主存中加载数据，导致处理速度受限。
- 时间浪费：CPU 在等待必要数据从主存传输到缓存时的闲置时间增加，效率降低。

4. 缓存级数

- 敏感应用及分析：复杂的软件应用，如数据库管理系统和大规模集成电路（VLSI）设计工具，对缓存的级数特别敏感。多级缓存可以有效地减少对主存的访问次数，提高数据访问速度。
- 影响：缓存级数较少可能导致数据访问效率降低，因为它减少了存储频繁访问数据的机会，增加了对慢速主存的依赖。
- 瓶颈原因：缺乏足够级数的缓存意味着数据在缓存和主存之间的移动次数增加，导致数据访问延迟。
- 时间浪费：数据在缓存级别之间移动所需的时间增加，减慢了处理速度，CPU 等待数据的时间增长。

5. 缓存行大小

- 敏感应用及分析：内存密集型应用，如流式处理和大数据分析，对缓存行大小特别敏感。合适的缓存行大小可以减少缓存失效次数和不必要的数据加载。
- 影响：不适当的缓存行大小会导致缓存利用率低下，增加缓存失效率，从而需要更频繁地从主存中加载数据。
- 瓶颈原因：缓存行太大可能导致加载过多不必要的数据，浪费缓存空间；太小则增加了缓存管理的开销。
- 时间浪费：不合理的缓存行大小导致的高缓存失效率增加了 CPU 等待数据的时间，降低了处理效率。

参考文献

[1] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, May 2011.