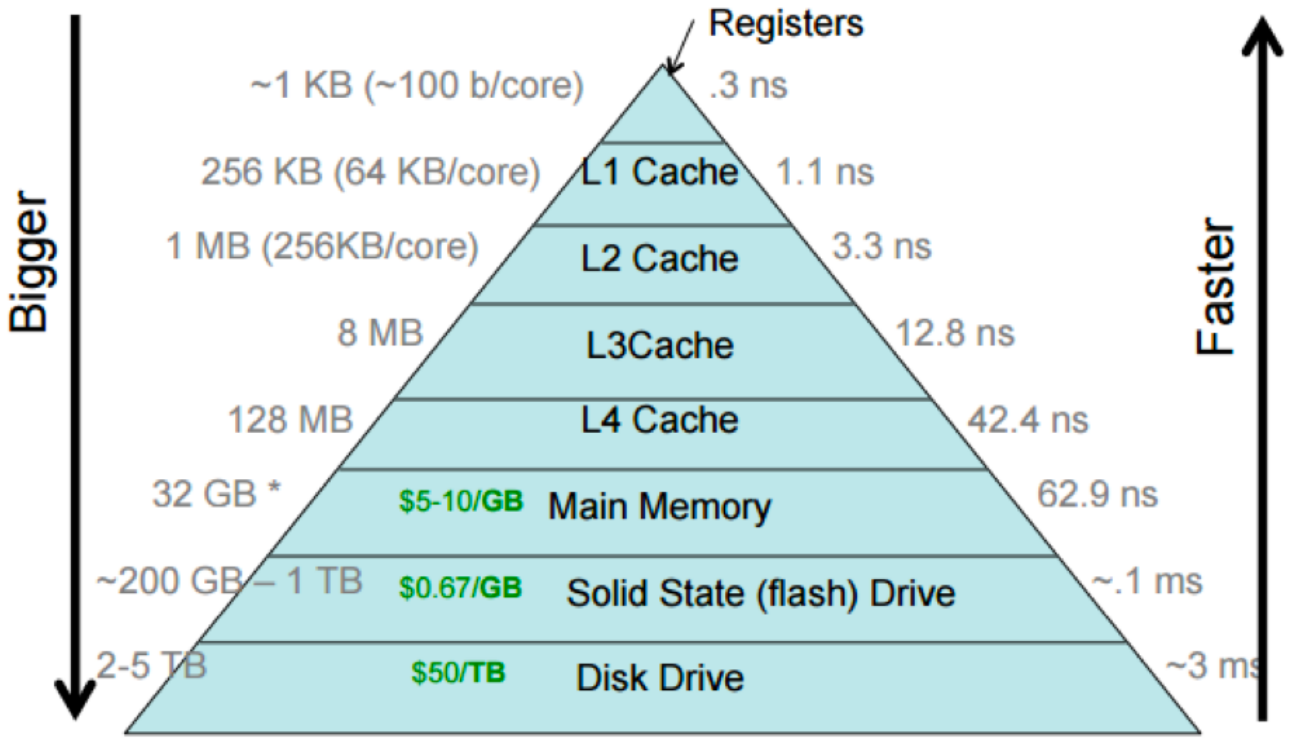
揭秘 cache 访问延迟背后的计算机原理

极客重生 2022-09-01 22:49 Posted on 广东

CPU 的 cache 往往是分多级的金字塔模型，L1 最靠近 CPU，访问延迟最小，但 cache 的容量也最小。本文介绍如何测试多级 cache 的访存延迟，以及背后蕴含的计算机原理。



图源：https://cs.brown.edu/courses/csci1310/2020/assign/labs/lab4.html

Cache Latency

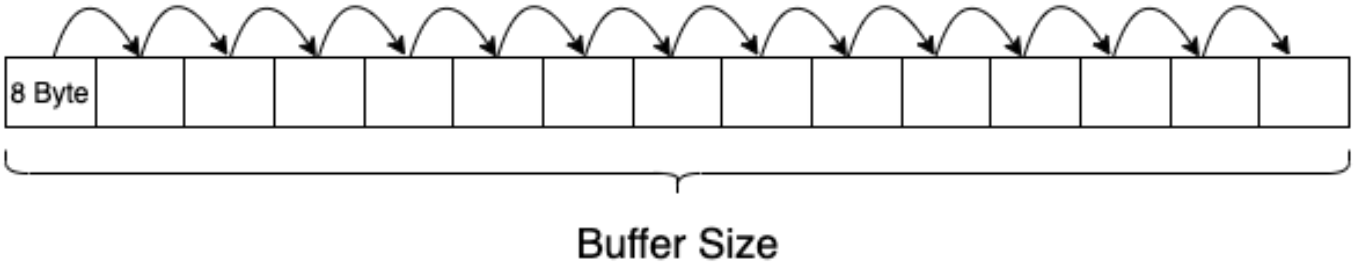
Wikichip[1] 提供了不同 CPU 型号的 cache 延迟，单位一般为 cycle，通过简单的运算，转换为 ns。以 skylake 为例，CPU 各级 cache 延迟的基准值为：

Cache/Latency	Size	Cycle	Nanosecond
L1	32 KB/core	4	1.5072
L2	1024 KB/core	14	5.2752
L3	1.375 MB/core (33 MB/socket)	50-70	18.84-26.37

CPU Frequency:2654MHz (0.3768 nanosec/clock)

设计实验

1. naive thinking



申请一个 buffer，buffer size 为 cache 对应的大小，第一次遍历进行预热，将数据全部加载到 cache 中。第二次遍历统计耗时，计算每次 read 的延迟平均值。

代码实现 mem-lat.c 如下：

```
#include <sys/types.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/mman.h>
#include <sys/time.h>
#include <unistd.h>

#define ONE p = (char **)*p;
#define FIVE    ONE ONE ONE ONE ONE
#define TEN FIVE FIVE
```

```
#define FIFTY   TEN TEN TEN TEN TEN
#define HUNDRED FIFTY FIFTY

static void usage()
{
    printf("Usage: ./mem-lat -b xxx -n xxx -s xxx\n");
    printf("   -b buffer size in KB\n");
    printf("   -n number of read\n\n");
    printf("   -s stride skipped before the next access\n\n");
    printf("Please don't use non-decimal based number\n");
}


int main(int argc, char* argv[])
{
  unsigned long i, j, size, tmp;
    unsigned long memsize = 0x800000; /* 1/4 LLC size of skylake, 1/5 of
    unsigned long count = 1048576; /* memsize / 64 * 8 */
    unsigned int stride = 64; /* skipped amount of memory before the next
    unsigned long sec, usec;
    struct timeval tv1, tv2;
    struct timezone tz;
    unsigned int *indices;

    while (argc-- > 0) {
        if ((*argv)[0] == '-') {  /* look at first char of next */
            switch ((*argv)[1]) {   /* look at second */
                case 'b':
                    argv++;
                    argc--;
                    memsize = atoi(*argv) * 1024;
                    break;

                case 'n':
                    argv++;
                    argc--;
                    count = atoi(*argv);
```

```
48                      break;
49
50                  case 's':
51                      argv++;
52                      argc--;
53                      stride = atoi(*argv);
54                      break;
55
56                  default:
57                      usage();
58                      exit(1);
59                      break;
60              }
61          }
62          argv++;
63      }
64
65
66  char* mem = mmap(NULL, memsize, PROT_READ | PROT_WRITE, MAP_PRIVATE | M
67      // trick3: init pointer chasing, per stride=8 byte
68      size = memsize / stride;
69      indices = malloc(size * sizeof(int));
70
71      for (i = 0; i < size; i++)
72          indices[i] = i;
73
74      // trick 2: fill mem with pointer references
75      for (i = 0; i < size - 1; i++)
76          *(char **)&mem[indices[i]*stride]= (char*)&mem[indices[i+1]*strid
77      *(char **)&mem[indices[size-1]*stride]= (char*)&mem[indices[0]*stride
78
79      char **p = (char **) mem;
80      tmp = count / 100;
81
82      gettimeofday (&tv1, &tz);
83      for (i = 0; i < tmp; ++i) {
84          HUNDRED;  //trick 1
```

```
    }
    gettimeofday (&tv2, &tz);
    if (tv2.tv_usec < tv1.tv_usec) {
        usec = 1000000 + tv2.tv_usec - tv1.tv_usec;
        sec = tv2.tv_sec - tv1.tv_sec - 1;
    } else {
        usec = tv2.tv_usec - tv1.tv_usec;
        sec = tv2.tv_sec - tv1.tv_sec;
    }

    printf("Buffer size: %ld KB, stride %d, time %d.%06d s, latency %.2f
            memsize/1024, stride, sec, usec, (sec * 1000000  + usec) * 10
    munmap(mem, memsize);
    free(indices);
}
```

这里用到了 3 个小技巧：

- HUNDRED 宏：通过宏展开，尽可能避免其他指令对访存的干扰。
- 二级指针：通过二级指针将buffer串起来，避免访存时计算偏移。
- char* 和 char** 为 8 字节，因此，stride 为 8。

测试方法：

```
#set -x

work=./mem-lat
buffer_size=1
stride=8

for i in `seq 1 15` ; do
    taskset -ac 0 $work -b $buffer_size -s $stride
    buffer_size=$(($buffer_size*2))
done
```

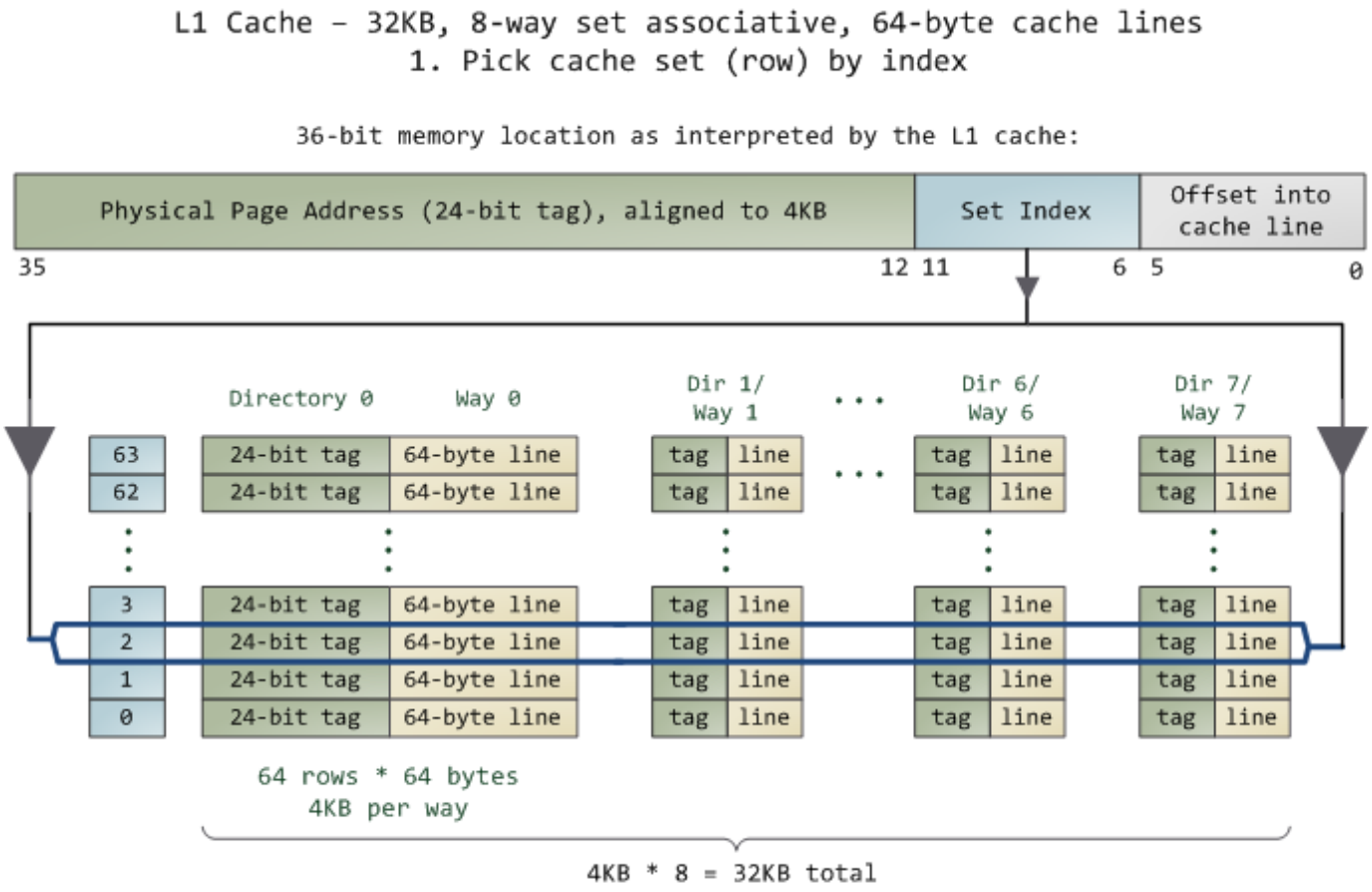
测试结果如下：

```
//L1
Buffer size: 1 KB, stride 8, time 0.003921 s, latency 3.74 ns
Buffer size: 2 KB, stride 8, time 0.003928 s, latency 3.75 ns
Buffer size: 4 KB, stride 8, time 0.003935 s, latency 3.75 ns
Buffer size: 8 KB, stride 8, time 0.003926 s, latency 3.74 ns
Buffer size: 16 KB, stride 8, time 0.003942 s, latency 3.76 ns
Buffer size: 32 KB, stride 8, time 0.003963 s, latency 3.78 ns
//L2
Buffer size: 64 KB, stride 8, time 0.004043 s, latency 3.86 ns
Buffer size: 128 KB, stride 8, time 0.004054 s, latency 3.87 ns
Buffer size: 256 KB, stride 8, time 0.004051 s, latency 3.86 ns
Buffer size: 512 KB, stride 8, time 0.004049 s, latency 3.86 ns
Buffer size: 1024 KB, stride 8, time 0.004110 s, latency 3.92 ns
//L3
Buffer size: 2048 KB, stride 8, time 0.004126 s, latency 3.94 ns
Buffer size: 4096 KB, stride 8, time 0.004161 s, latency 3.97 ns
Buffer size: 8192 KB, stride 8, time 0.004313 s, latency 4.11 ns
Buffer size: 16384 KB, stride 8, time 0.004272 s, latency 4.07 ns
```

相比基准值，L1 延迟偏大，L2 和 L3 延迟偏小，不符合预期。

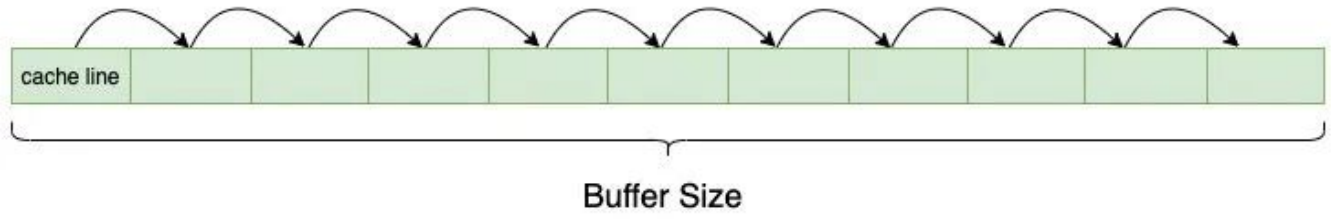
2. thinking with hardware: cache line

现代处理器，内存以 cache line 为粒度，组织在 cache 中。访存的读写粒度都是一个 cache line，最常见的缓存线大小是 64 字节。



如果我们简单的以 8 字节为粒度，顺序读取 128KB 的 buffer，假设数据命中的是 L2，那么数据就会被缓存到 L1，一个 cache line 其他的访存操作都只会命中 L1，从而导致我们测量的 L2 延迟明显偏小。

本文测试的 CPU，cacheline 大小 64 字节，只需将 stride 设为 64。



测试结果如下：

```
//L1
Buffer size: 1 KB, stride 64, time 0.003933 s, latency 3.75 ns
Buffer size: 2 KB, stride 64, time 0.003930 s, latency 3.75 ns
Buffer size: 4 KB, stride 64, time 0.003925 s, latency 3.74 ns
Buffer size: 8 KB, stride 64, time 0.003931 s, latency 3.75 ns
Buffer size: 16 KB, stride 64, time 0.003935 s, latency 3.75 ns
```

```
Buffer size: 32 KB, stride 64, time 0.004115 s, latency 3.92 ns
//L2
Buffer size: 64 KB, stride 64, time 0.007423 s, latency 7.08 ns
Buffer size: 128 KB, stride 64, time 0.007414 s, latency 7.07 ns
Buffer size: 256 KB, stride 64, time 0.007437 s, latency 7.09 ns
Buffer size: 512 KB, stride 64, time 0.007429 s, latency 7.09 ns
Buffer size: 1024 KB, stride 64, time 0.007650 s, latency 7.30 ns
Buffer size: 2048 KB, stride 64, time 0.007670 s, latency 7.32 ns
//L3
Buffer size: 4096 KB, stride 64, time 0.007695 s, latency 7.34 ns
Buffer size: 8192 KB, stride 64, time 0.007786 s, latency 7.43 ns
Buffer size: 16384 KB, stride 64, time 0.008172 s, latency 7.79 ns
```

虽然相比方案 1，L2 和 L3 的延迟有所增大，但还是不符合预期。

3. thinking with hardware: prefetch

现代处理器，通常支持预取（prefetch）。数据预取通过将代码中后续可能使用到的数据提前加载到 cache 中，减少 CPU 等待数据从内存中加载的时间，提升 cache 命中率，进而提升软件的运行效率。

Intel 处理器支持 4 种硬件预取 [2]，可以通过 MSR 控制关闭和打开：

Prefetcher	Bit# in MSR 0x1A4	Description
L2 hardware prefetcher	0	Fetches additional lines of code or data into the L2 cache
L2 adjacent cache line prefetcher	1	Fetches the cache line that comprises a cache line pair (128 bytes)
DCU prefetcher	2	Fetches the next cache line into L1-D cache
DCU IP prefetcher	3	Uses sequential load history (based on Instruction Pointer of previous loads) to determine whether to prefetch additional lines

这里我们简单的将 stride 设为 128 和 256，避免硬件预取。测试的 L3 访存延迟明显增大：

```
// stride 128
Buffer size: 1 KB, stride 256, time 0.003927 s, latency 3.75 ns
Buffer size: 2 KB, stride 256, time 0.003924 s, latency 3.74 ns
Buffer size: 4 KB, stride 256, time 0.003928 s, latency 3.75 ns
Buffer size: 8 KB, stride 256, time 0.003923 s, latency 3.74 ns
Buffer size: 16 KB, stride 256, time 0.003930 s, latency 3.75 ns
Buffer size: 32 KB, stride 256, time 0.003929 s, latency 3.75 ns
Buffer size: 64 KB, stride 256, time 0.007534 s, latency 7.19 ns
Buffer size: 128 KB, stride 256, time 0.007462 s, latency 7.12 ns
Buffer size: 256 KB, stride 256, time 0.007479 s, latency 7.13 ns
Buffer size: 512 KB, stride 256, time 0.007698 s, latency 7.34 ns
Buffer size: 512 KB, stride 128, time 0.007597 s, latency 7.25 ns
Buffer size: 1024 KB, stride 128, time 0.009169 s, latency 8.74 ns
Buffer size: 2048 KB, stride 128, time 0.010008 s, latency 9.55 ns
Buffer size: 4096 KB, stride 128, time 0.010008 s, latency 9.55 ns
Buffer size: 8192 KB, stride 128, time 0.010366 s, latency 9.89 ns
Buffer size: 16384 KB, stride 128, time 0.012031 s, latency 11.47 ns

// stride 256
```

```
Buffer size: 512 KB, stride 256, time 0.007698 s, latency 7.34 ns
Buffer size: 1024 KB, stride 256, time 0.012654 s, latency 12.07 ns
Buffer size: 2048 KB, stride 256, time 0.025210 s, latency 24.04 ns
Buffer size: 4096 KB, stride 256, time 0.025466 s, latency 24.29 ns
Buffer size: 8192 KB, stride 256, time 0.025840 s, latency 24.64 ns
Buffer size: 16384 KB, stride 256, time 0.027442 s, latency 26.17 ns
```

L3 的访存延迟基本上是符合预期的，但是 L1 和 L2 明显偏大。

如果测试随机访存延迟，更加通用的做法是，在将buffer指针串起来时，随机化一下。

```
    // shuffle indices
    for (i = 0; i < size; i++) {
        j = i +  rand() % (size - i);
        if (i != j) {
            tmp = indices[i];
            indices[i] = indices[j];
            indices[j] = tmp;
        }
    }
```

可以看到，测试结果与 stride 为 256 基本上是一样的。

```
Buffer size: 1 KB, stride 64, time 0.003942 s, latency 3.76 ns
Buffer size: 2 KB, stride 64, time 0.003925 s, latency 3.74 ns
Buffer size: 4 KB, stride 64, time 0.003928 s, latency 3.75 ns
Buffer size: 8 KB, stride 64, time 0.003931 s, latency 3.75 ns
Buffer size: 16 KB, stride 64, time 0.003932 s, latency 3.75 ns
Buffer size: 32 KB, stride 64, time 0.004276 s, latency 4.08 ns
Buffer size: 64 KB, stride 64, time 0.007465 s, latency 7.12 ns
Buffer size: 128 KB, stride 64, time 0.007470 s, latency 7.12 ns
Buffer size: 256 KB, stride 64, time 0.007521 s, latency 7.17 ns
Buffer size: 512 KB, stride 64, time 0.009340 s, latency 8.91 ns
Buffer size: 1024 KB, stride 64, time 0.015230 s, latency 14.53 ns
Buffer size: 2048 KB, stride 64, time 0.027567 s, latency 26.29 ns
Buffer size: 4096 KB, stride 64, time 0.027853 s, latency 26.56 ns
Buffer size: 8192 KB, stride 64, time 0.029945 s, latency 28.56 ns
```

```
Buffer size: 16384 KB, stride 64, time 0.034878 s, latency 33.26 ns
```

4. thinking with compiler: register keyword

解决掉 L3 偏小的问题后，我们继续看 L1 和 L2 偏大的原因。为了找出偏大的原因，我们先反汇编可执行程序，看看执行的汇编指令是否是我们想要的：

```
objdump -D -S mem-lat > mem-lat.s
```

- `-D`: Display assembler contents of all sections.
- `-S`：Intermix source code with disassembly. (gcc编译时需使用`-g`，生成调式信息)

生成的汇编文件 mem-lat.s：

```
    char **p = (char **)mem;
    400b3a:    48 8b 45 c8              mov    -0x38(%rbp),%rax
    400b3e:    48 89 45 d0              mov    %rax,-0x30(%rbp) // push stack

//...
      HUNDRED;
    400b85:    48 8b 45 d0              mov    -0x30(%rbp),%rax
    400b89:    48 8b 00                 mov    (%rax),%rax
    400b8c:    48 89 45 d0              mov    %rax,-0x30(%rbp)
    400b90:    48 8b 45 d0              mov    -0x30(%rbp),%rax
    400b94:    48 8b 00                 mov    (%rax),%rax
```

首先，变量 mem 赋值给变量 p，变量 p 压入栈`-0x30(%rbp)`。

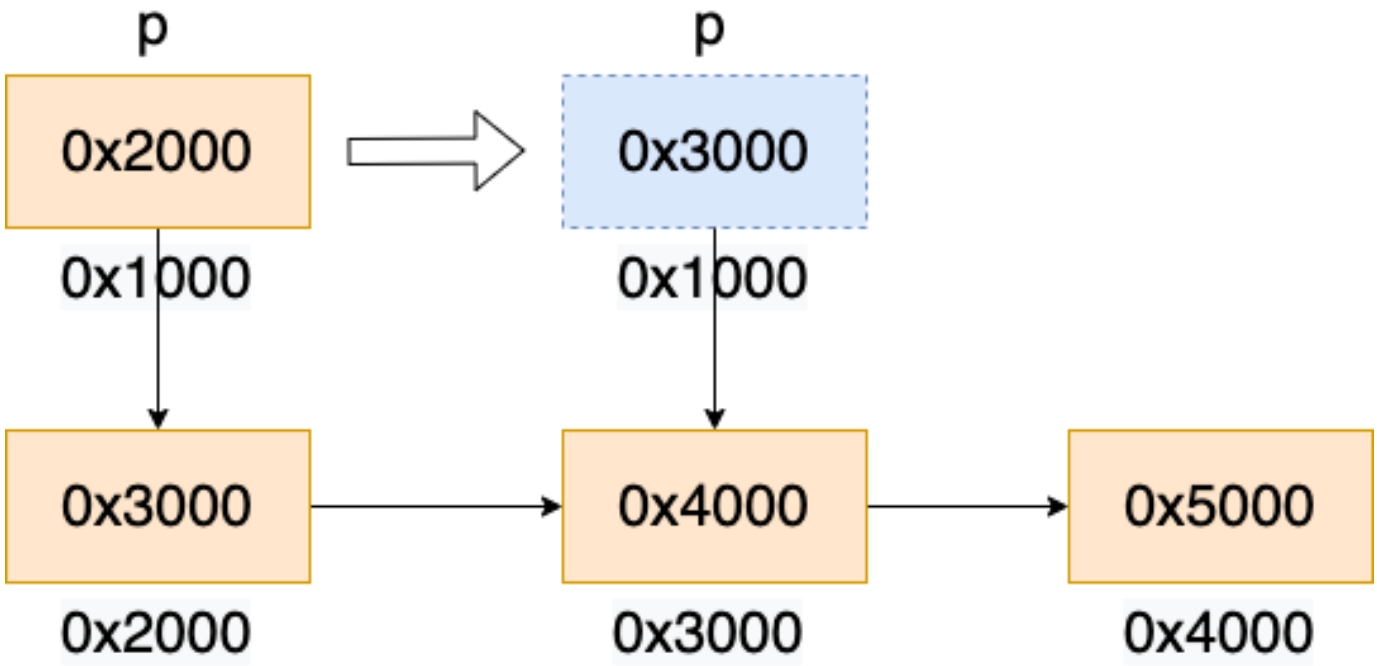
```
    char **p = (char **)mem;
    400b3a:    48 8b 45 c8              mov    -0x38(%rbp),%rax
    400b3e:    48 89 45 d0              mov    %rax,-0x30(%rbp)
```

访存的逻辑：

```
          HUNDRED;  // p = (char **)*p
    400b85:    48 8b 45 d0              mov    -0x30(%rbp),%rax
    400b89:    48 8b 00                 mov    (%rax),%rax
```

```
4    400b8c:    48 89 45 d0                 mov     %rax,-0x30(%rbp)
```

- 先从栈中读取指针变量 p 的值到`rax`寄存器（变量 p 的类型为`char **`，是一个二级指针，也就是说，指针 p 指向一个`char *`的变量，即 p 的值也是一个地址）。下图中变量 p 的值为 0x2000。
- 将`rax`寄存器指向变量的值读入`rax`寄存器，对应单目运算`*p`。下图中地址 0x2000的值为 0x3000，rax 更新为 0x3000。
- 将`rax`寄存器赋值给变量p。下图中变量p的值更新为0x3000。



根据反汇编的结果可以看到，**期望的 1 条 move 指令被编译成了 3 条，cache 的延迟也就增加了 3 倍**。

C 语言的 register 关键字，可以让编译器将变量保存到寄存器中，从而避免每次从栈中读取的开销。

> It's a hint to the compiler that the variable will be heavily used and that you recommend it be kept in a processor register if possible.

我们在声明 p 时，加上 register 关键字。

```
1  register char **p = (char **)mem;
```

测试结果如下：

```
// L1
Buffer size: 1 KB, stride 64, time 0.000030 s, latency 0.03 ns
Buffer size: 2 KB, stride 64, time 0.000029 s, latency 0.03 ns
Buffer size: 4 KB, stride 64, time 0.000030 s, latency 0.03 ns
Buffer size: 8 KB, stride 64, time 0.000030 s, latency 0.03 ns
Buffer size: 16 KB, stride 64, time 0.000030 s, latency 0.03 ns
Buffer size: 32 KB, stride 64, time 0.000030 s, latency 0.03 ns
// L2
Buffer size: 64 KB, stride 64, time 0.000030 s, latency 0.03 ns
Buffer size: 128 KB, stride 64, time 0.000030 s, latency 0.03 ns
Buffer size: 256 KB, stride 64, time 0.000029 s, latency 0.03 ns
Buffer size: 512 KB, stride 64, time 0.000030 s, latency 0.03 ns
Buffer size: 1024 KB, stride 64, time 0.000030 s, latency 0.03 ns
// L3
Buffer size: 2048 KB, stride 64, time 0.000030 s, latency 0.03 ns
Buffer size: 4096 KB, stride 64, time 0.000029 s, latency 0.03 ns
Buffer size: 8192 KB, stride 64, time 0.000030 s, latency 0.03 ns
Buffer size: 16384 KB, stride 64, time 0.000030 s, latency 0.03 ns
```

访存延迟全部变为不足 1 ns，**明显不符合预期**。

5. thinking with compiler: Touch it!

重新反汇编，看看哪里出了问题，编译代码如下：

```
      for (i = 0; i < tmp; ++i) {
  40155e:   48 c7 45 f8 00 00 00    movq   $0x0,-0x8(%rbp)
  401565:   00
  401566:   eb 05                   jmp    40156d <main+0x37e>
  401568:   48 83 45 f8 01          addq   $0x1,-0x8(%rbp)
  40156d:   48 8b 45 f8             mov    -0x8(%rbp),%rax
  401571:   48 3b 45 b0             cmp    -0x50(%rbp),%rax
  401575:   72 f1                   jb     401568 <main+0x379>
          HUNDRED;
      }
      gettimeofday (&tv2, &tz);
  401577:   48 8d 95 78 ff ff ff    lea    -0x88(%rbp),%rdx
  40157e:   48 8d 45 80             lea    -0x80(%rbp),%rax
```

```
401582:   48 89 d6                mov    %rdx,%rsi
401585:   48 89 c7                mov    %rax,%rdi
401588:   e8 e3 fa ff ff          callq  401070 <gettimeofday@plt>
```

HUNDRED 宏没有产生任何汇编代码。涉及到变量 p 的语句，并没有实际作用，只是数据读取，大概率被编译器优化掉了。

```
        register char **p = (char **) mem;
        tmp = count / 100;

        gettimeofday (&tv1, &tz);
        for (i = 0; i < tmp; ++i) {
            HUNDRED;
        }
        gettimeofday (&tv2, &tz);

        /* touch pointer p to prevent compiler optimization */
        char **touch = p;
```

反汇编验证一下：

```
            HUNDRED;
    401570:   48 8b 1b                mov    (%rbx),%rbx
    401573:   48 8b 1b                mov    (%rbx),%rbx
    401576:   48 8b 1b                mov    (%rbx),%rbx
    401579:   48 8b 1b                mov    (%rbx),%rbx
    40157c:   48 8b 1b                mov    (%rbx),%rbx
```

HUNDRED 宏产生的汇编代码只有操作寄存器 rbx 的 mov 指令，高级。

延迟的测试结果如下：

```
// L1
Buffer size: 1 KB, stride 64, time 0.001687 s, latency 1.61 ns
Buffer size: 2 KB, stride 64, time 0.001684 s, latency 1.61 ns
Buffer size: 4 KB, stride 64, time 0.001682 s, latency 1.60 ns
Buffer size: 8 KB, stride 64, time 0.001693 s, latency 1.61 ns
```

```
Buffer size: 16 KB, stride 64, time 0.001683 s, latency 1.61 ns
Buffer size: 32 KB, stride 64, time 0.001783 s, latency 1.70 ns
// L2
Buffer size: 64 KB, stride 64, time 0.005896 s, latency 5.62 ns
Buffer size: 128 KB, stride 64, time 0.005915 s, latency 5.64 ns
Buffer size: 256 KB, stride 64, time 0.005955 s, latency 5.68 ns
Buffer size: 512 KB, stride 64, time 0.007856 s, latency 7.49 ns
Buffer size: 1024 KB, stride 64, time 0.014929 s, latency 14.24 ns
// L3
Buffer size: 2048 KB, stride 64, time 0.026970 s, latency 25.72 ns
Buffer size: 4096 KB, stride 64, time 0.026968 s, latency 25.72 ns
Buffer size: 8192 KB, stride 64, time 0.028823 s, latency 27.49 ns
Buffer size: 16384 KB, stride 64, time 0.033325 s, latency 31.78 ns
```

L1 延迟 1.61 ns，L2 延迟 5.62 ns，终于，符合预期！

写在最后

本文的思路和代码参考自 lmbench[3]，和团队内其他同学的工具 mem-lat。最后给自己挖个坑，在随机化 buffer 指针时，没有考虑硬件 TLB miss 的影响，如果有读者有兴趣，待日后有空补充。

参考文献：

[1] https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)

[2]https://software.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html

[3]McVoy L W, Staelin C. lmbench: Portable Tools for Performance Analysis[C]//USENIX annual technical conference. 1996: 279-294.

往期精彩推荐

深入理解Cache工作原理

深入理解Linux 的Page Cache

熬过绝望低谷，你便无人能敌