P3: Profiling Serial Merge Sort

杨浩然
10195501441

写在前面（我的一些屁话）

之前把双启动的 Ubuntu 搞崩了，最初的缘由是想在 Ubuntu 上用搜狗输入法，就安装了一个搜狗，然后系统就无法启动了，倒腾到晚上两点，差点一气之下就换电脑了，还好忍住了。第二天把 Ubuntu 删掉了，把 Windows10 升到了 Windows11，用 WSL 装了一个 Ubuntu 20.04，不过毛病也很多。首先它是不带 Perf 的，要去 GitHub 上把内核源码拉下来编译。其次，WSL 上的 Perf 被阉割了（因为硬件的虚拟化），很多信息拿不到。情急之下，我开了台云主机……

哈哈，云主机虚拟化更厉害（😅）

Checkoff Item 1:

Make note of the bottleneck.

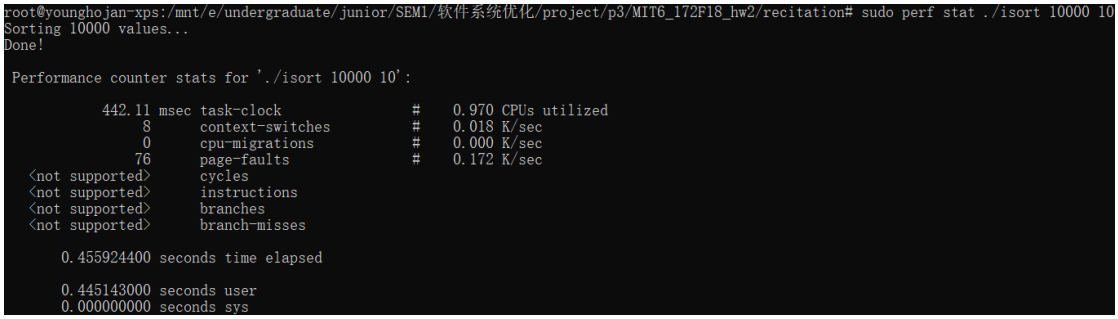
这个 Checkoff Item 要求“Make note of one bottleneck”，接下来的部分先 identify and analyse the hottest execution spots，再 diagnose。

> *perf 是 Linux 的性能分析工具，能够进行函数级与指令级的热点查找。它由一个叫“Performance counters”的内核子系统实现，基于事件采样原理，以性能事件为基础，支持针对处理器相关性能指标与操作系统相关性能指标的性能剖析，可用于性能瓶颈的查找与热点代码的定位。可以使用 perf 提供的工具分析程序的性能，也可以在 perf 之上构建自己的工具。*
>
> *perf 通过系统调用 `sys_perf_event_open` 进入到内核中，内核根据 perf 提供的信息在 PMU 上初始化一个 PMC（Performance Monitoring Counter）。PMC 随着指定硬件事件的发生而自动累加。在 PMC 溢出时，PMU 触发一个 PMI（Performance Monitoring Interrupt）中断。内核在 PMI 中断的处理函数中保存 PMC 的计数值，触发中断时的指令地址，当前时间戳以及当前进程的 PID，TID，comm 等信息。这些信息被统称为一个采样（sample）。内核会将收集到*

> *的 sample 放入用于跟用户空间通信的 Ring Buffer。用户空间里的 perf 分析程序采用 mmap 机制从 ring buffer 中读入采样，并对其解析。*
>
> *perf支持两种模式：计算模式和采样模式。perf stat 使用的是计数模式，perf record 使用的是采样模式。*

先用 `perf stat` 看下整体概况：

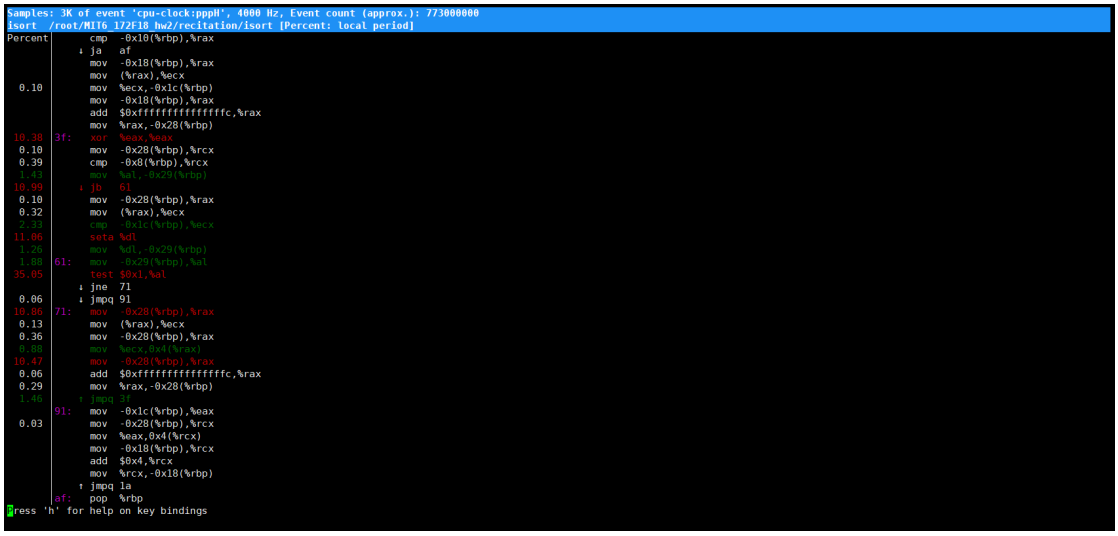
```
sudo perf stat ./isort 10000 10
```



cycles, instructions, branches, branch-missed 是拿不到的。

在我的 Ubuntu 还没崩掉的时候，曾做了一点点实验（截图随着系统的消失而消失），记得 branch-misses 是比较正常的，只占了所有分支预测的 0.01%。所以先排除 bottleneck 出在 branch-misses 上的可能性。

`perf report` 看看：



大量时间都消耗花在 isort 上，具体是 `test` 判断。如果非要说这是 bottleneck 的话，那只能换排序算法，用其他效率更好、性能更高的算法了。

思来想去，虽然云主机抓不到 cycles，但我感觉 bottleneck 就出在硬件上，可能时钟频率太低了（不然就把排序算法重构一下）。

Checkoff Item 2:

Run sum under cachegrind to identify cache performance. It may take a little while. In the output, look at the D1 and LLd misses. D1 represents the lowest-level cache (L1), and LL represents the last (highest) level data cache (on most machines, L3). Do these numbers correspond with what you would expect? Try playing around with the values N and U in sum.c. How can you bring down the number of cache misses?

cachegrind 模拟程序如何与机器的缓存层次结构和分支预测器交互。它模拟一台具有独立的一级指令和数据缓存（I1 和 D1），并由统一的二级缓存（L2）支持的机器。

Checkoff Item 2 在我本地的 VMware Workstation 中的 Ubuntu 虚拟机上完成，云实例还是不太适合做这个实验，它的性能浮动比较大，硬件不停被其他用户共享。

先 `lscpu` 把 cpu 信息打出来看下：

```
youngbojan@ubuntu:~/MIT6_172F18_hw2/recitation$ lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   45 bits physical, 48 bits virtual
CPU(s):                          2
On-line CPU(s) list:             0,1
Thread(s) per core:              1
Core(s) per socket:              1
Socket(s):                       2
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           158
Model name:                      Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
Stepping:                        10
CPU MHz:                         2592.006
BogoMIPS:                        5184.01
Hypervisor vendor:               VMware
Virtualization type:             full
L1d cache:                       64 KiB
L1i cache:                       64 KiB
L2 cache:                        512 KiB
L3 cache:                        24 MiB
```

在 cachegrind 下 run sum：

```
sudo valgrind --tool=cachegrind --branch-sim=yes ./sum
```

```
youngbojan@ubuntu:~/MIT6_172F18_hw2/recitation$ sudo valgrind --tool=cachegrind --branch-sim=yes ./sum
==7176== Cachegrind, a cache and branch-prediction profiler
==7176== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==7176== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==7176== Command: ./sum
==7176==
--7176-- warning: L3 cache found, using its data for the LL simulation.
--7176-- warning: specified LL cache: line_size 64  assoc 16  total_size 12,582,912
--7176-- warning: simulated LL cache: line_size 64  assoc 24  total_size 12,582,912
Allocated array of size 10000000
Summing 100000000 random values...
Done. Value = 938895920
==7176==
==7176== I   refs:      3,540,234,693
==7176== I1  misses:            1,221
==7176== LLi misses:            1,207
==7176== I1  miss rate:          0.00%
==7176== LLi miss rate:          0.00%
==7176==
==7176== D   refs:        610,072,284  (400,055,242 rd   + 210,017,042 wr)
==7176== D1  misses:      100,548,109  ( 99,922,223 rd   +     625,886 wr)
==7176== LLd misses:       69,277,719  ( 68,651,897 rd   +     625,822 wr)
==7176== D1  miss rate:          16.5% (       25.0%     +         0.3%  )
==7176== LLd miss rate:          11.4% (       17.2%     +         0.3%  )
==7176==
==7176== LL refs:         100,549,330  ( 99,923,444 rd   +     625,886 wr)
==7176== LL misses:        69,278,926  ( 68,653,104 rd   +     625,822 wr)
==7176== LL miss rate:            1.7% (        1.7%     +         0.3%  )
==7176==
==7176== Branches:        210,044,272  (110,043,749 cond + 100,000,523 ind)
==7176== Mispredicts:           5,326  (      5,105 cond +         221 ind)
==7176== Mispred rate:            0.0% (        0.0%     +         0.0%   )
```

L1 cache 的 miss rate 远低于 L3 cache 的 miss rate，低出了一个数量级，这显然是符合直觉的。

下面搞搞 N and U in sum.c，希望把 cache miss 降下来。

我觉得 N 没有必要修改，同时修改两个变量会使实验结果失去说服力。并且我认为这个程序的目的就是做 N 次加法，不应该修改 N，而应依据对 cache 的观察，对 U 进行修改以降低 cache miss。

根据 `lscpu` 的结果，cache 的大小应该在 $64/1024 + 512/1024 + 24 = 24.5625$ MiB 左右（如果不算 L1i cache）。如果缩小 N 使得整个数组可以存进 cache 中，cache miss 可能会减少。

```
const int U = 10000000;   // size of the array. 10 million vals ~= 40MB
const int N = 100000000;  // number of searches to perform
```

注释表示，数组长度为 10 million 时，占用空间约为 40MB，所以当长度为 6000000 时，理论上 cache 能装载整个数组。修改 `U = 6000000`，查看结果：

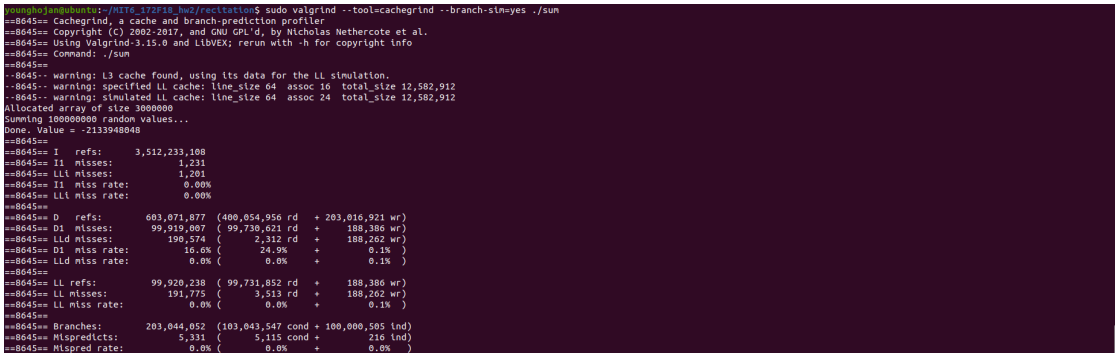
```
younghojan@ubuntu:~/MIT6_172F18_hw2/recitation$ sudo valgrind --tool=cachegrind --branch-sim=yes ./sum
==8633== Cachegrind, a cache and branch-prediction profiler
==8633== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==8633== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==8633== Command: ./sum
==8633==
--8633-- warning: L3 cache found, using its data for the LL simulation.
--8633-- warning: specified LL cache: line_size 64  assoc 10  total_size 12,582,912
--8633-- warning: simulated LL cache: line_size 64  assoc 24  total_size 12,582,912
Allocated array of size 6000000
Summing 100000000 random values...
Done. Value = -1940022480
==8633==
==8633== I   refs:      3,524,233,108
==8633== I1  misses:            1,231
==8633== LLi misses:            1,217
==8633== I1  miss rate:          0.00%
==8633== LLi miss rate:          0.00%
==8633==
==8633== D   refs:        606,071,877  (400,054,956 rd   + 206,016,921 wr)
==8633== D1  misses:      100,242,792  ( 99,866,906 rd   +     375,886 wr)
==8633== LLd misses:       48,031,875  ( 47,656,053 rd   +     375,822 wr)
==8633== D1  miss rate:          16.5% (       25.0%     +         0.2%  )
==8633== LLd miss rate:           7.9% (       11.9%     +         0.2%  )
==8633==
==8633== LL refs:         100,244,023  ( 99,868,137 rd   +     375,886 wr)
==8633== LL misses:        48,033,092  ( 47,657,270 rd   +     375,822 wr)
==8633== LL miss rate:            1.2% (        1.2%     +         0.2%  )
==8633==
==8633== Branches:        206,044,052  (106,043,547 cond + 100,000,505 ind)
==8633== Mispredicts:           5,331  (       5,115 cond +         216 ind)
==8633== Mispred rate:            0.0% (         0.0%     +         0.0%   )
```

各级 cache miss 有一定程度的降低，但是不明显。

注意到输出结果中有这样一句：

```
warning: simulated LL cache: line_size 64  assoc 24  total_size 12,582,912
```

google 了一下，没搜到这句话的意思（看来大家都不在意啊...）。我猜的话，可能是 cachegrind 模拟的 L3 cache 的大小是 12,582,912 Bytes？正好是12 MB，所以我把 N 改为了 3000000，再试试：



可以看到，L1 cache 的 miss 情况没有特别大的好转，但 L2 cache 和 L3 cache 的 miss 都大大降低（在 cache 访问次数基本不变的情况下）：

- L2 miss 从 **7.9%**（48,031,875）降低到了 **0.0%**（190,574）
- L3 miss 从 **1.2%**（48,033,092）降低到了 **0.0%**（191,775）

这个结果还是很令我惊喜的！证明缩小数组长度使其能完全放入 cache，是减少 L2、L3 cache miss 的一个可行的办法（在 checkoff 2 限定的条件下）。

解释一下我为什么又换 WSL 完成下面的 homework...
因为 VMware 里面的虚拟机操作太不方便了...VMware tool 好像出问题了，没办法在客户机和虚拟机之间共享文件夹以及复制粘贴字符...

Write-up 1:

Compare the Cachegrind output on the DEBUG=1 code versus DEBUG=0 compiler optimized code. Explain the advantages and disadvantages of using instruction count as a substitute for time when you compare the performance of different versions of this program.

Write-up 1 要求我们比较 `DEBUG=1` 和 `DEBUG=0` 条件下程序的性能差异。Makefile 中这样规定：

```
ifeq ($(DEBUG),1)
CFLAGS := -DDEBUG -O0 $(CFLAGS)
else
CFLAGS := -O3 -DNDEBUG $(CFLAGS)
endif
CFLAGS := $(CFLAGS)
```

即 `DEBUG=0` 意味着 O3 级别的优化，`DEBUG=1` 意味着 O0 级别的优化。

(程序运行参数给的 `10000 10`) `DEBUG=0`：

```
Running test #0...
Generating random array of 10000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_a          : Elapsed execution time: 0.024598 sec
sort_a repeated : Elapsed execution time: 0.024358 sec
Generating inverted array of 10000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_a          : Elapsed execution time: 0.047908 sec
sort_a repeated : Elapsed execution time: 0.047547 sec

Running test #1...
 --> test_zero_element at line 245: PASS

Running test #2...
 --> test_one_element at line 266: PASS
Done testing.
==1289==
==1289== I   refs:      231,806,631
==1289== I1  misses:          1,594
==1289== LLi misses:          1,504
==1289== I1  miss rate:        0.00%
==1289== LLi miss rate:        0.00%
==1289==
==1289== D   refs:       88,020,229  (53,507,235 rd   + 34,512,994 wr)
==1289== D1  misses:        317,166  (   174,257 rd   +    142,909 wr)
==1289== LLd misses:          5,160  (     2,435 rd   +      2,725 wr)
==1289== D1  miss rate:         0.4% (       0.3%     +        0.4%  )
==1289== LLd miss rate:         0.0% (       0.0%     +        0.0%  )
==1289==
==1289== LL refs:           318,760  (   175,851 rd   +    142,909 wr)
==1289== LL misses:           6,664  (     3,939 rd   +      2,725 wr)
==1289== LL miss rate:          0.0% (       0.0%     +        0.0%  )
==1289==
==1289== Branches:       38,039,753  (36,338,899 cond +  1,700,854 ind)
==1289== Mispredicts:     2,501,484  ( 2,501,131 cond +        353 ind)
==1289== Mispred rate:          6.6% (       6.9%     +        0.0%   )
```

DEBUG=1：

```
Running test #0...
Generating random array of 10000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_a          : Elapsed execution time: 0.044268 sec
sort_a repeated : Elapsed execution time: 0.043779 sec
Generating inverted array of 10000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_a          : Elapsed execution time: 0.087897 sec
sort_a repeated : Elapsed execution time: 0.087322 sec

Running test #1...
 --> test_zero_element at line 245: PASS

Running test #2...
 --> test_one_element at line 266: PASS
Done testing.
==1314==
==1314== I   refs:      414,979,703
==1314== I1  misses:          1,573
==1314== LLi misses:          1,493
==1314== I1  miss rate:        0.00%
==1314== LLi miss rate:        0.00%
==1314==
==1314== D   refs:      263,689,964  (196,283,241 rd   + 67,406,723 wr)
==1314== D1  misses:        315,384  (   173,323 rd   +    142,061 wr)
==1314== LLd misses:          5,147  (     2,424 rd   +      2,723 wr)
==1314== D1  miss rate:         0.1% (       0.1%     +        0.2%  )
==1314== LLd miss rate:         0.0% (       0.0%     +        0.0%  )
==1314==
==1314== LL refs:           316,957  (   174,896 rd   +    142,061 wr)
==1314== LL misses:           6,640  (     3,917 rd   +      2,723 wr)
==1314== LL miss rate:          0.0% (       0.0%     +        0.0%  )
==1314==
==1314== Branches:       42,777,226  ( 41,076,390 cond +  1,700,836 ind)
==1314== Mispredicts:     3,451,913  (  3,451,570 cond +        343 ind)
==1314== Mispred rate:          8.1% (       8.4%     +        0.0%   )
```

根据对结果的比较，可以看出在运行时间的意义上，`DEBUG=0` 编译的程序运行用时仅为 `DEBUG=1` 的 50% 左右，branch miss 也有所降低（降低 1.5%）。

使用 callgrind 查看指令数（用 perf 看应该更好，但是虚拟机不支持）：

`DEBUG=0`：

```
Running test #0...
Generating random array of 10000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_a          : Elapsed execution time: 0.036705 sec
sort_a repeated : Elapsed execution time: 0.036416 sec
Generating inverted array of 10000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_a          : Elapsed execution time: 0.072452 sec
sort_a repeated : Elapsed execution time: 0.072108 sec

Running test #1...
 --> test_zero_element at line 245: PASS

Running test #2...
 --> test_one_element at line 266: PASS
Done testing.
==1345==
==1345== Events    : Ir
==1345== Collected : 231806674
==1345==
==1345== I   refs:      231,806,674
```

`DEBUG=1`：

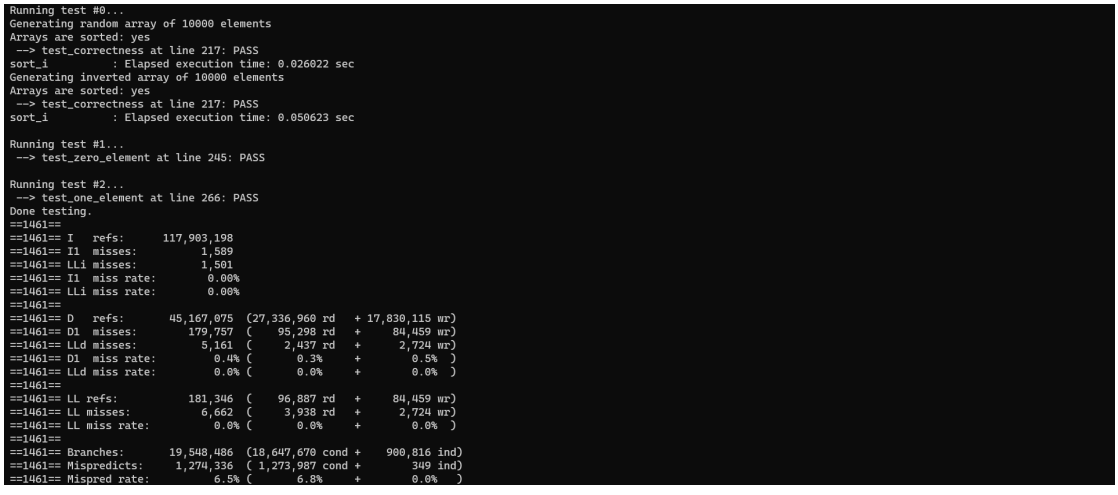
`DEBUG=0` 编译的程序运行的指令数仅为 `DEBUG=1` 的 50% 左右，和运行时间基本一致。

使用指令数作为性能指标可以看得更清楚，可以认为指令数是比运行时间更加“本质”的指标。但使用运行时间作为指标，对性能差异的展现更为直观一些，更符合直觉。

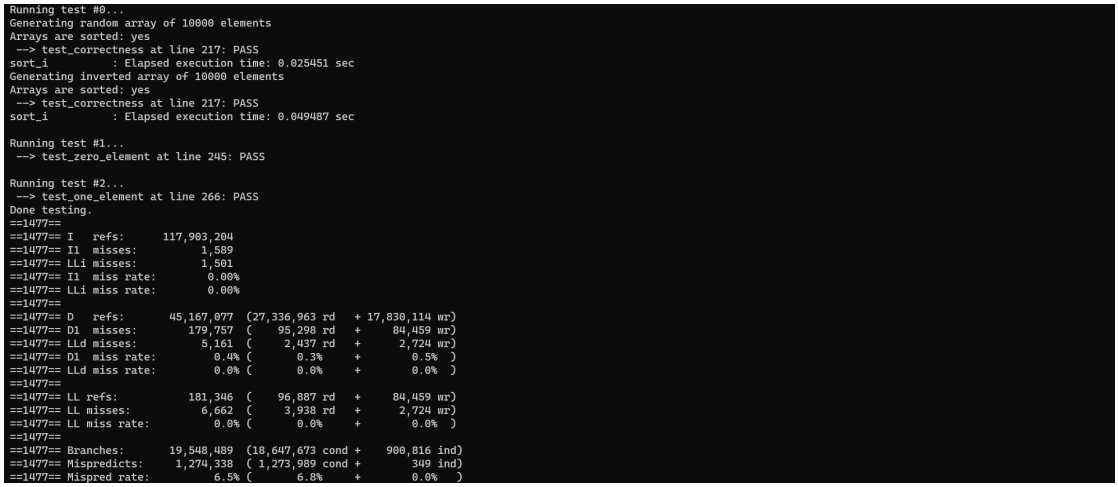
Write-up 2:

Explain which functions you chose to inline and report the performance differences you observed between the inlined and uninlined sorting routines.

先不加 `inline`，用 cachegrind 看看：

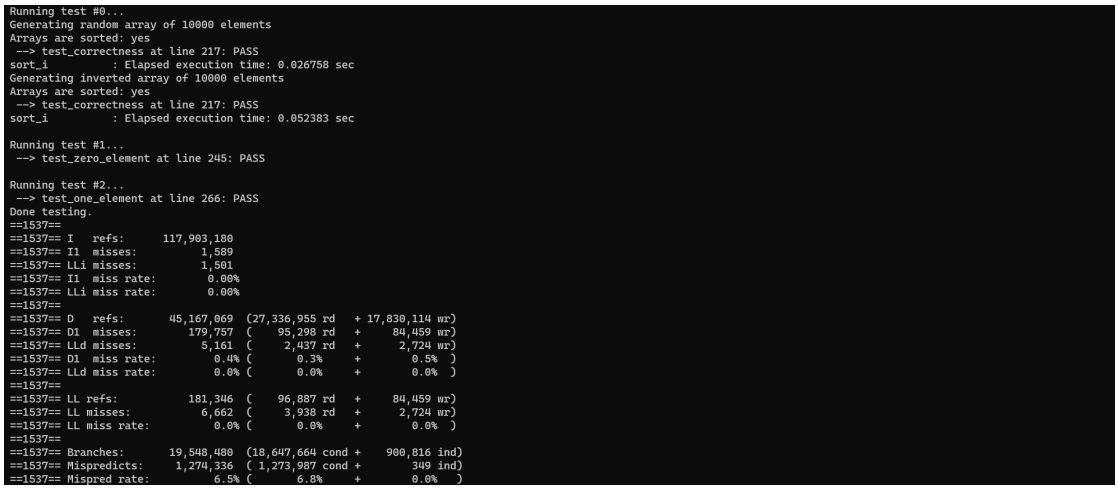


下面给 `merge_i()` 和 `copy_i()` 都加上 `inline`：



发现运行时间有了略微提升，cache miss 和 branches mispredicts 几乎一样，没有改变。

我对 `inline` 粗略的理解是，`inline` 的引入是为了解决一些频繁调用的小函数大量消耗栈空间的问题，根据对代码的观察，我尝试给多次被调用的 `mem_free()` 和 `mem_alloc()` 也加上 `inline`：



性能似乎没有提升(?!)

Write-up 3:

Explain the possible performance downsides of inlining recursive functions. How could profiling data gathered using cachegrind help you measure these negative performance effects?

首先，`inline` 函数的规范只是一个提示，编译器可以完全忽略 `inline` 限定符。如果编译器尝试插入 `inline` 函数递归，它就会创建无限大的代码。所以，大多数现代编译器都会认识到这一点。它们可以：

1. 不内联这个函数
2. 将其内联到一定深度，如果到那时它还没有终止，那么使用标准函数调用约定调用函数的单独实例。这可以以高性能的方式处理许多常见的情况，同时为罕见的情况留下一个大调用深度的退路。

内联可能会导致代码段的重复，影响到 instruction cache 的性能，所以 cachegrind 可以表现 negative performance effects...??

Write-up 4:

Give a reason why using pointers may improve performance. Report on any performance differences you observed in your implementation.

这部分应该无需观察 cache miss 和 branch mispredict，用 callgrind 看看运行时间和指令数。

不使用指针：

```
Running test #0...
Generating random array of 10000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_i          : Elapsed execution time: 0.035444 sec
Generating inverted array of 10000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_i          : Elapsed execution time: 0.070350 sec

Running test #1...
 --> test_zero_element at line 245: PASS

Running test #2...
 --> test_one_element at line 266: PASS
Done testing.
==1586==
==1586== Events    : Ir
==1586== Collected : 117903207
==1586==
==1586== I   refs:      117,903,207
```

使用指针：

```
Running test #0...
Generating random array of 10000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_p          : Elapsed execution time: 0.036346 sec
Generating inverted array of 10000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_p          : Elapsed execution time: 0.072106 sec

Running test #1...
 --> test_zero_element at line 245: PASS

Running test #2...
 --> test_one_element at line 266: PASS
Done testing.
==1571==
==1571== Events    : Ir
==1571== Collected : 118347223
==1571==
==1571== I   refs:      118,347,223
```

没有什么区别？甚至还更慢了？

我的理解是，使用指针的话，计算内存地址只需要加一个 offset，应该是更快的。可能是开了 O3 优化，这部分被自动优化掉了，下面开 O0 试试。

不使用指针：

使用指针：

现在指令数和时间减少一些了！说明使用指针还是更快的

Write-up 5:

Explain what sorting algorithm you used and how you chose the number of elements to be sorted in the base case. Report on the performance differences you observed.

在 Write-up 5 中做的事情是贴近现实的。我记得在金澈清老师在给我们上《算法设计与实现》这门课时说到，比如 Python 或者 C++ 内置的快速排序算法，排序过程中 base case 一定小时，会直接用插入排序，用迭代替代了递归。这里我们也这样做，我使用 isort，并且把 base case 的阈值设到 100：

性能提升相当明显：对随机数组排序的时间减少了约 80%，对反向数组排序的时间减少了约 70%，总指令数减少了约 68%。

Write-up 6:

Explain any difference in performance in your sort_m.c. Can a compiler automatically make this optimization for you and save you all the effort? Why or why not?

```
Running test #0...
Generating random array of 10000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_m          : Elapsed execution time: 0.014102 sec
Generating inverted array of 10000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_m          : Elapsed execution time: 0.035240 sec

Running test #1...
 --> test_zero_element at line 245: PASS

Running test #2...
 --> test_one_element at line 266: PASS
Done testing.
==1652==
==1652== Events    : Ir
==1652== Collected : 68613806
==1652==
==1652== I   refs:      68,613,806
```

貌似没变快...?

我觉得对于编译器来讲，无法达到这种算法层面上的优化。如果编译器能做到上面的优化，就有点篡改代码语义的嫌疑了。

Write-up 7:

Report any differences in performance in your sort_f.c, and explain the differences using profiling data.

```
Running test #0...
Generating random array of 10000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_a          : Elapsed execution time: 0.021242 sec
sort_a repeated : Elapsed execution time: 0.020911 sec
sort_i          : Elapsed execution time: 0.020276 sec
sort_p          : Elapsed execution time: 0.021691 sec
sort_c          : Elapsed execution time: 0.006733 sec
sort_m          : Elapsed execution time: 0.006643 sec
sort_f          : Elapsed execution time: 0.006422 sec
Generating inverted array of 10000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_a          : Elapsed execution time: 0.041742 sec
sort_a repeated : Elapsed execution time: 0.041373 sec
sort_i          : Elapsed execution time: 0.040213 sec
sort_p          : Elapsed execution time: 0.042351 sec
sort_c          : Elapsed execution time: 0.015983 sec
sort_m          : Elapsed execution time: 0.015489 sec
sort_f          : Elapsed execution time: 0.014387 sec
```

对于归并排序，每次递归都会开辟一个临时数组，这里一次性开辟一个大数组，是一种时空权衡，运行速度上升。

总结

写到这儿已是新年，紧赶慢赶还是把作业写完了，明天又要早起去复习其他科目。感谢阅读到这儿，祝新年快乐。

其实如果时间再多一点的话，这次作业我可以做得更好一些，我自我感觉比较喜欢做系统相关的工作，觉得更贴近现实，实用性强，给我带来的成就感也很大。最后也没有太多想总结的，心得体会都写在每次作业报告里面了，由衷感谢这门课上给予我帮助的两位老师和两位助教!