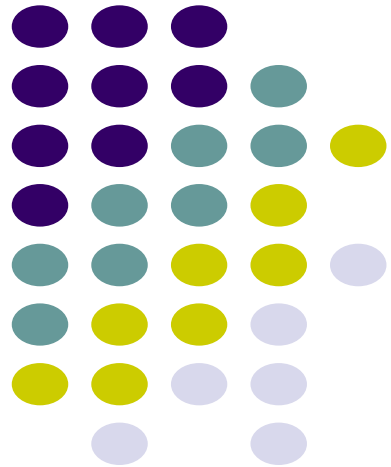
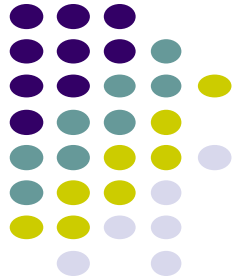
《计算机系统基础（四）：编程与调试实践》

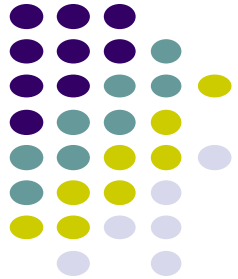
程序性能分析与优化





性 能 分 析

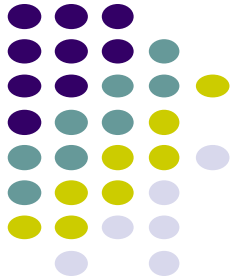
性能分析的几个基本步骤



1. 设置合理的workload，通常是程序目标的应用场景
2. 在这些实际的应用场景中把程序运行起来，并且记录性能相关的信息
3. 对程序做出相应的性能优化

本节分析若干个memcpy实现的性能，在实践中体会性能分析的基本原理。

memcpy函数

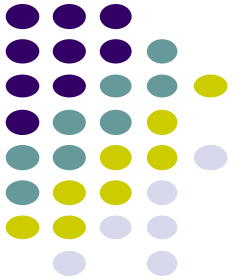


memcpy的函数声明：

```
void *memcpy(void *dst, const void *src, size_t n);
```

- memcpy总共有三个参数：
  - 第一个参数是一个void*参数，dst是内存复制的目标地址
  - 第二个参数const void *src是内存复制的源地址
  - 第三个参数size_t n代表内存复制需要复制的字节数
- memcpy返回一个void*类型的指针，是输入的第一个参数，也就是复制的目标地址

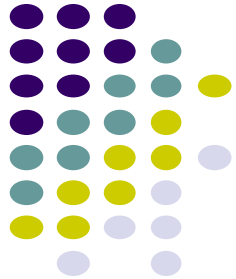
memcpy函数



- memcpy的功能是实现src到dst的内存复制，总共复制n个字节，并且memcpy会假设两段内存并不重叠。如果src和dst可能发生重叠，则需要使用memmove进行复制。

- 有时src和dst会用C语言的restrict关键字修饰，即把参数声明为void *restrict，或const void *restrict。restrict关键字的功能是告知编译器可以假设内存不重叠，并且做出相应的优化。

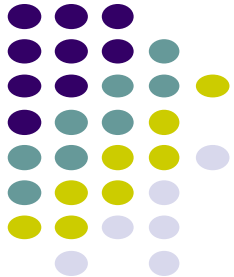
```
void *memcpy(void *restrict, const void *restrict,
size_t n);  告知编译器可以假设内存不重叠，并且做出相应的优化。
```



- memcpy是C语言标准库(libc)的一部分，也是很多程序非常常用的库函数

- 如果已经有一个应用程序，那么如何得到这个应用程序在实际运行中到底调用了多少次memcpy？以及每一次memcpy所复制的字节数？
  - ✓ 使用ltrace工具对应用程序的memcpy进行profiling

ltrace工具



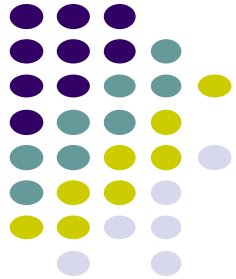
- ltrace是一种trace工具，也就是追踪工具，它能够打印出对库函数的调用

```
ltrace -f -s 0 -e memcpy -- python3 -c '2**10000000' 2>&1 >/dev/null | sed -r -n 's/.*, ([0-9]+)\).*$/\1/p'
```

  - 第一个参数-f代表不仅要追踪主进程，还要追踪它的子进程
  - 第二个参数-s 0表示不打印字符串的任何信息，以免干扰之后的处理
  - 第三个参数-e memcpy代表需要追踪的是memcpy函数
  - 接下来的两个减号代表ltrace参数结束

（接下页）

ltrace工具



```
ltrace -f -s 0 -e memcpy -- python3 -c '2**10000000' 2>&1 >/dev/null | sed -r -n 's/.*, ([0-9]+)\).*$/\1/p'
```

（接上页）

- 绿色部分是ltrace实际追踪的命令——调用python3 -c执行一段脚本，计算2的10的七次方的值
- 蓝色部分2>&1 代表将标准错误输出重定向到标准输出，即将ltrace的trace输出重定向到标准输出，之后>/dev/null表示把源程序的输出清空
- 竖线管道将ltrace的输出连接到sed进行过滤，打印出每一次memcpy的大小

memcpy性能分析相关代码

- 可以用tree命令查看当前目录树下的文件，以了解项目的代码组织结构
  - 本项目是一个C和Python混合项目，其中C代码主要是memcpy的实现和运行，而Python代码则是用于绘图运行等辅助生成工具

```
$ tree
.
├── Makefile
├── harness.c
├── impl
│   ├── asm_64b.c
│   ├── asm_8b.c
│   ├── libc.c
│   ├── simple.c
│   └── unroll.c
├── perftune.h
├── plot.py
└── run.py
```

- .h文件中声明了若干个memcpy的版本，其中有libc、simple、inline assembly和unroll，分别表示libc本地实现、一个教科书上的简单实现、使用内联汇编的实现和使用循环展开的实现
  - 这些实现位于impl目录中

memcpy性能分析相关代码

- 包含main函数的harness.c文件用于运行这些memcpy的实际代码。其中，我们从2的20次方开始，到2的28次方，也就是从1MB到128MB内存，这些大小运行memcpy

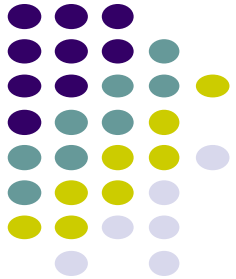
```c
#include <sys/time.h>
#include <perftune.h>

void run(size_t size, int round);

int main() {
  for (int i = 20; i < 28; i++) {
    size_t size = 1UL << i;
    run(size, 100 * (1UL << (27 - i))); // 运行100 * 2^(27-i)次
  }
}

void run(size_t size, int round) {
  void *volatile src = malloc(size);
  void *volatile dst = malloc(size);
  uint64_t runtime = UINT64_MAX;

  assert(src && dst); // 确保malloc()成功

  memset(src, -1, size);
  memset(dst, 0, size);
  my_memcpy(dst, src, size);
  if (memcmp(src, dst, size) != 0) { // 测试my_memcpy()的正确性
    fprintf(stderr, "Wrong implementation!\n");
    exit(1);
  }

  for (int rnd = 0; rnd < round; rnd++) {
    struct timeval st, ed;
    uint64_t elapse;
    void *memcpy_ret;
"harness.c" 51L, 1237C
```

memcpy性能分析相关代码

- **Run函数**
  - run函数的第一个参数size是memcpy的大小，第二个参数是run的代表memcpy被运行的次数（我们至少运行100次）
  - run函数中首先使用malloc分配size大小的内存。其中，src和dst被声明为了volatile，防止编译器对其进行优化
  - 我们将统计的运行时间设置为int 64最大值。同时我们使用assert确保malloc分配成功，如果malloc可分配失败，将会引起程序的崩溃

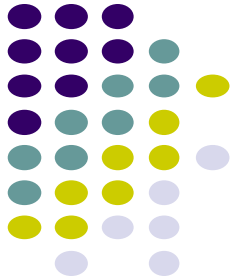
```
13 void run(size_t size, int round) {
14   void *volatile src = malloc(size);
15   void *volatile dst = malloc(size);
16   uint64_t runtime = UINT64_MAX;
17
18   assert(src && dst); // 确保malloc()成功
19
20   memset(src, -1, size);
21   memset(dst, 0, size);
22   my_memcpy(dst, src, size);
23   if (memcmp(src, dst, size) != 0) { // 测试my_memcpy()的正确性
24     fprintf(stderr, "Wrong implementation!\n");
25     exit(1);
26   }
27
28   for (int rnd = 0; rnd < round; rnd++) {
29     struct timeval st, ed;
30     uint64_t elapse;
31     void *memcpy_ret;
32
33     gettimeofday(&st, NULL); // 获取memcpy执行前的时间戳
34     memcpy_ret = my_memcpy(dst, src, size);
35     gettimeofday(&ed, NULL); // 获取memcpy执行后的时间戳
36
37     assert(memcpy_ret == dst);
38
39     elapse = (ed.tv_sec - st.tv_sec) * 1000000ULL +
40       (ed.tv_usec - st.tv_usec); // 计算本次运行耗时
41
42     // 记录所有运行中最短耗时
43     runtime = elapse < runtime ? elapse : runtime;
```

memcpy性能分析相关代码



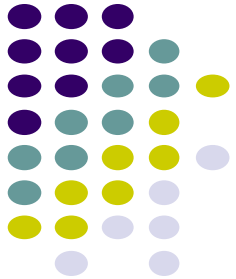
- **Run函数**（续）
  - 接下来测试memcpy是否正确。首先在原内存中使用memset，用-1对内存进行填充；然后将目标内存用零进行填充，并且调用memcpy；最终使用memcmp对拷贝前后的内存进行比较——如果拷贝成功，程序将会继续执行；如果拷贝失败，将会出错退出。
- 接下来，运行run多次。每次循环会用两个gettimeofday将memcpy的调用包裹起来，并且用assert确保memcpy拷贝成功
- 然后计算memcpy所用的时间，记录所有运行当中最短的一次的耗时，作为本次memcpy的耗时。进一步将运行结果打印到屏幕上，并且回收已经分配的内存。

项目编译 - Makefile



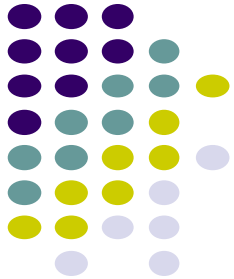
```
1  $(shell mkdir -p build/)
2
3  CFLAGS  := -Wall -Werror -I.
4  TARGETS := libc simple asm_8b asm_64b unroll
5
6  all: $(TARGETS)
7
8  %: harness.c impl/%.c perftune.h
9          for opt in 0 1 2 3; do \
10           gcc $(CFLAGS) -O$$opt impl/$*.c harness.c -Dmy_memcpy=memcpy_$* -o build/$*-gcc-O$$opt.out; \
11         done
12 #        gcc $(CFLAGS) -O1 impl/$*.c harness.c -Dmy_memcpy=memcpy_$* -o build/$*-gcc-O1.out
13 #        gcc $(CFLAGS) -O2 impl/$*.c harness.c -Dmy_memcpy=memcpy_$* -o build/$*-gcc-O2.out
14 #        gcc $(CFLAGS) -O3 impl/$*.c harness.c -Dmy_memcpy=memcpy_$* -o build/$*-gcc-O3.out
15
16 clean:
17         rm -rf build/
```

项目编译 - Makefile



- **Makefile内容：**
  - 创建一个名为build的目录
  - 指定CFLAGS编译的方式：-Wall -Werror表示输出所有的警告，并且将警告作为错误处理；-I.代表将当前目录作为include path
  - 声明多个目标：libc、simple、inline assembly、unroll，分别代表了每一个memcpy的实现
    - 每一个memcpy实现依赖于harness.c、运行测试用例的代码、以及具体的memcpy实现和头文件

项目编译方法



- 对于每一个编译目标使用gcc编译时，使用-o选项设置不同的编译优化等级：
  - o0不优化
  - o1、o2、o3分别代表三种类型的优化进行实现

- -D编译参数：可用来指定my_memcpy作为某一个具体的memcpy实现
  - 例如：对libc将定义为-Dmy_memcpy=memcpy_libc，从而引用正确的代码

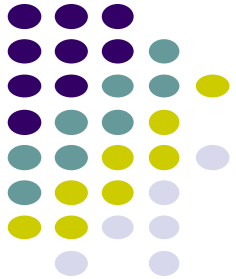
项目编译

- 使用make对项目进行编译时，将会执行脚本进行编译，并且会在buid的目录下生成相应的二进制文件
- 可以运行其中的例如asm_64b O0.out的程序，将会输出复制相应字节数的内存所需的时间，例如复制一兆内存需要110微秒

```
$ make
for opt in 0 1 2 3; do \
  gcc -Wall -Werror -I. -O$opt impl/libc.c harness.c -Dmy_memcpy=memcpy_libc -o build/libc-gcc-O$opt.out; \
done
for opt in 0 1 2 3; do \
  gcc -Wall -Werror -I. -O$opt impl/simple.c harness.c -Dmy_memcpy=memcpy_simple -o build/simple-gcc-O$opt.out; \
done
for opt in 0 1 2 3; do \
  gcc -Wall -Werror -I. -O$opt impl/asm_8b.c harness.c -Dmy_memcpy=memcpy_asm_8b -o build/asm_8b-gcc-O$opt.out; \
done
for opt in 0 1 2 3; do \
  gcc -Wall -Werror -I. -O$opt impl/asm_64b.c harness.c -Dmy_memcpy=memcpy_asm_64b -o build/asm_64b-gcc-O$opt.out; \
done
for opt in 0 1 2 3; do \
  gcc -Wall -Werror -I. -O$opt impl/unroll.c harness.c -Dmy_memcpy=memcpy_unroll -o build/unroll-gcc-O$opt.out; \
done
$ ls build/
asm_64b-gcc-O0.out  asm_8b-gcc-O0.out  libc-gcc-O0.out  simple-gcc-O0.out  unroll-gcc-O0.out
asm_64b-gcc-O1.out  asm_8b-gcc-O1.out  libc-gcc-O1.out  simple-gcc-O1.out  unroll-gcc-O1.out
asm_64b-gcc-O2.out  asm_8b-gcc-O2.out  libc-gcc-O2.out  simple-gcc-O2.out  unroll-gcc-O2.out
asm_64b-gcc-O3.out  asm_8b-gcc-O3.out  libc-gcc-O3.out  simple-gcc-O3.out  unroll-gcc-O3.out
```

```
$ ./build/asm_64b-gcc-O0.out
(1048576, 110)
(2097152, 244)
(4194304, 320)
(8388608, 776)
(16777216, 1982)
(33554432, 4220)
```

memcpy的libc实现



- memcpy的libc实现较为简单：直接调用libc中的memcpy函数，并且返回该函数的返回值

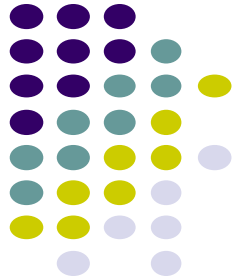
```c
#include <perftune.h>

void *memcpy_libc(void *dst, const void *src, size_t size) {
  return memcpy(dst, src, size);
}
```

memcpy的另一简单实现

- 与教科书上的实现较为类似
- 在第六行的while循环中，每当剩余需要拷贝的字节数大于零的时候，就会将字节数减1并进入循环
- 在循环体中，程序将拷贝的目标地址和原地址强制类型转换为char*，也就是指向字符型的指针，并且将原地址取出的一个字节赋予目标地址

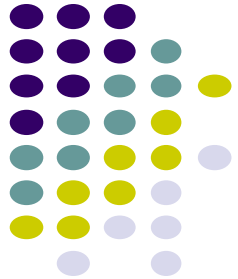
```
1  #include <perftune.h>
2
3  void *memcpy_simple(void *dst, const void *src, size_t size) {
4    void *ret = dst;
5
6    while (size--) {
7      (*(char *)dst++) = (*(char *)src++);
8    }
9
10   return ret;
11 }
```

memcpy的内联汇编实现



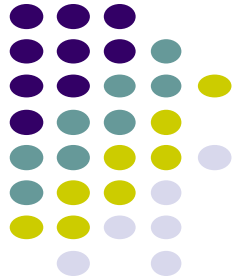
```
1 #include <perftune.h>
2
3 void *memcpy_asm_8b(void *dst, const void *src, size_t size) {
4   void *ret = dst;
5
6   asm volatile("cld; rep movsb"
7     : "=c"(size), "=D"(dst), "=S"(src) // clobbers
8     : "c"(size), "D"(dst), "S"(src) // %rcx ← size; %rdi ← dst; %rsi ← src;
9     : "memory" // clobbers memory
10  );
11
12  return ret;
13 }
```

memcpy的内联汇编实现



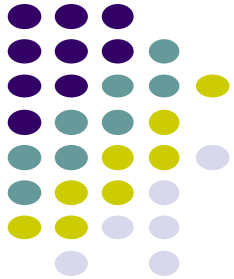
- 内联汇编的主体是第6行的asm关键字所包含的一系列代码。其中最主要的汇编部分由两条指令构成
  - 第1条指令**cld**，其中的**cl**代表清除处理器的eflags中的标志位，**d**代表清除标志位当中的df，也就是使处理器在执行重复指令时，按照从小到大的方向进行
  - 第2条指令**rep**，也就是重复前缀的移动拷贝指令。mov代表数据的移动，s代表字符串，b代表以字节为单位拷贝。这条指令在执行前，指定rcx寄存器中保存复制的size，rdi寄存器保存拷贝的目标地址，rsi寄存器保存拷贝的源地址

memcpy的内联汇编实现



- 正确设置寄存器之后，第6行的movsb指令把rsi寄存器指向的地址中的值赋予rdi寄存器指向的地址处，并且重复拷贝rcx寄存器所指定数量的字节
- 内联汇编还指定了我们的实现会修改rdi、rsi、rcx寄存器，以及可能会修改内存 —— 关于内联汇编更多的知识请参考相关文档

64位拷贝实现



- 按照64位进行拷贝的实现假设内存已按照8字节进行对齐，也就是按照64 bit进行对齐
- 该实现使用movsq指令，q后缀代表按照8个字节进行拷贝

```c
#include <perftune.h>

void *memcpy_asm_64b(void *dst, const void *src, size_t size) {
  void *ret = dst;
  assert(size % 8 == 0);

  size /= 8;
  asm volatile("cld; rep movsq"
    : "=c"(size), "=D"(dst), "=S"(src) // clobbers
    : "c"(size), "D"(dst), "S"(src) // %rcx ← size; %rdi ← dst; %rsi ← src;
    : "memory" // clobbers memory
  );

  return ret;
}
```

基于循环展开的memcpy实现

- 该实现假设内存已按照64字节对齐，每一轮循环都进行64字节的内存拷贝，其中：
  - 首先，将拷贝的源地址和目标地址都转化为64位整数的指针
  - 接下来，第10行到第17行中的每一行使用一个赋值语句，将一个64位整数赋值给另一个64位整数，从而实现8个字节的拷贝——即手工展开循环。第10行到第17行共完成了64个字节的拷贝
  - 第18行和第19行将源地址和目标地址加64

```
1  #include <perftune.h>
2
3  void *memcpy_unroll(void *dst, const void *src, size_t size) {
4    void *ret = dst;
5    assert(size % 64 == 0);
6
7    for (; size; size -= 64) {
8      const uint64_t *s = src;
9      uint64_t *d = dst;
10     d[0] = s[0];
11     d[1] = s[1];
12     d[2] = s[2];
13     d[3] = s[3];
14     d[4] = s[4];
15     d[5] = s[5];
16     d[6] = s[6];
17     d[7] = s[7];
18     dst = (char *)dst + 64;
19     src = (char *)src + 64;
20   }
21
22   return ret;
23 }
```

运行测试程序

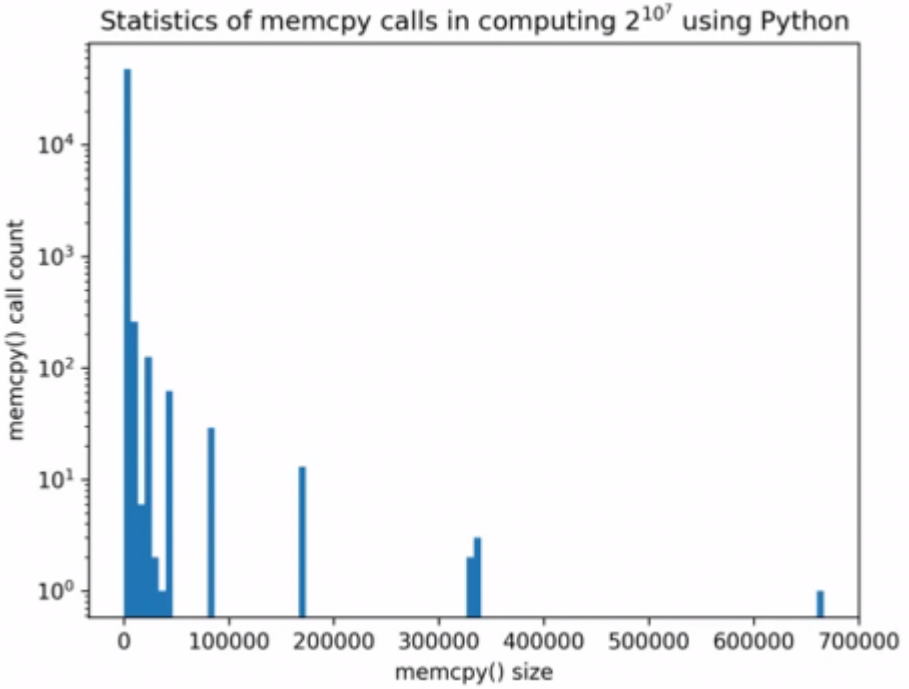
- 使用run.py当中的脚本测试、运行程序
  - 第36行到41行：对于每一个memcpy实现（libc、simple、unroll和两个inline assembly）以及每一个相应的编译器（例如在这里选用了gcc和每一个编译优化等级O0、O1、O2、O3），都进行测试运行并且输出运行的结果

```
36 for impl in ['libc', 'simple', 'unroll', 'asm_8b', 'asm_64b']:
37   for compiler in ['gcc']:
38     for opt in ['O0', 'O1', 'O2', 'O3']:
39       if impl in ['asm_8b', 'asm_64b', 'libc'] and opt ≠ 'O3': continue
40       res = subprocess.check_output([f'./build/{impl}-{compiler}-{opt}.out'])
41       report_result(impl, compiler, opt, res.decode('utf-8'))
```

- 运行的结果将会被送到plot.py当中，进一步使用matplotlib对运行的结果进行绘制

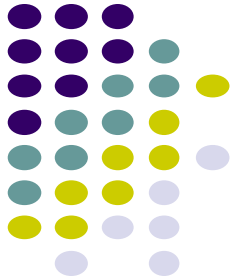
运行测试程序

- 将上述命令的结果绘制成直方图，可以发现在实际的程序执行中，memcpy可能会被调用很多次，而且虽然大部分对memcpy的调用复制的数据都非常少，但是仍然有一定数量的对较大内存的memcpy



- 这里关注较大（即1MB内存以上）的内存复制性能，也就是说假设只关注1MB以上的workload
  - 具体来说，我们关注从1MB，2MB，4MB一直到128MB memcpy的性能

运行测试程序

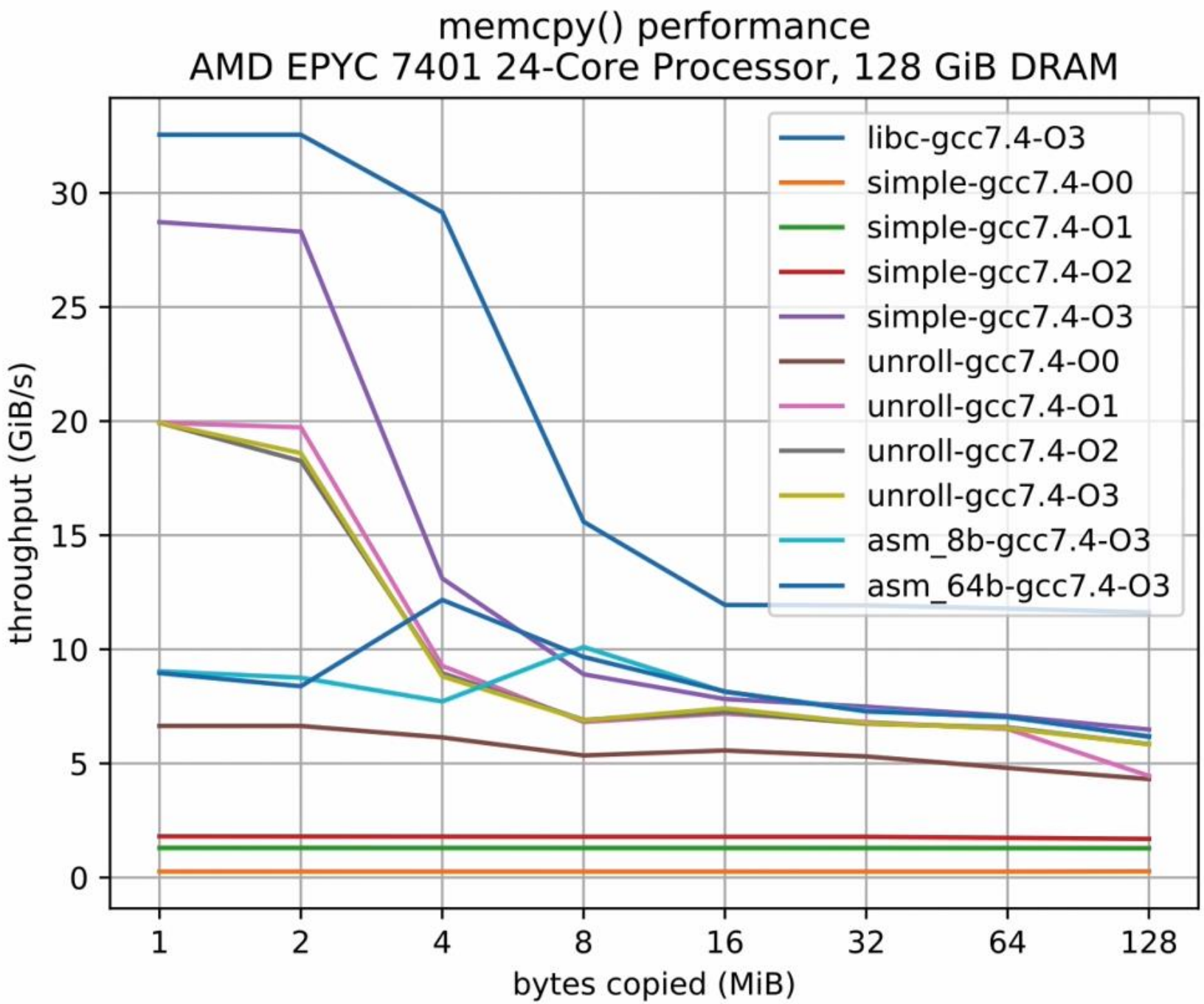


- 在多个硬件平台上对memcpy的性能进行评测
  - Intel(R) Pentium(R) CPU 4415Y低功耗处理器、AMD A10-5800K APU桌面处理器、Intel(R) i5-7500桌面处理器、AMD EPYC 7401多核服务器处理器

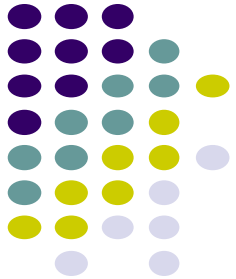
- 将memcpy的性能绘制在屏幕上，其中横坐标表示复制的内存数量，以兆为单位，从1兆到128兆，分别对应2的20次方字节和2的27次方字节；纵轴表示memcpy每秒钟能够复制的数量，以GiB每秒为单位。

AMD EPYC 7401处理器 128G内存

memcpy性能

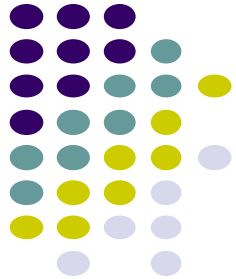


测试运行结果分析



- AMD EPYC 7401处理器+128G内存系统上的memcpy性能
  - 首先看到如图所示，最下区域有三个内存拷贝吞吐量较小的实现，它们分别是橙色、绿色和红色，代表simple（也就是用循环进行逐个字节复制）在O0、O1、O2三个编译环境下它们的性能。
  - 值得注意的是，如果使用O3对simple进行编译，编译器将会得到一个性能非常高的memcpy，它仅次于libc内部的实现，在拷贝较小内存如1MB时，相比于其他实现有很大的提升。
  - 使用循环展开实现的memcpy（棕色线条）在O0（即不进行编译优化）时，性能远远高于按字节拷贝O0、O1和O2的性能，甚至接近了系统自带的汇编实现

测试运行结果分析



- 为了理解这些memcpy实现的性能差异，我们查看一下它们的汇编代码
  - 使用**objdump**命令及其-d选项，查看二进制程序文件中的汇编代码
  - 使用**less**命令查找memcpy所在的位置

测试运行结果分析

- gcc在-O2优化级别下，simple memcpy被编译成了一个循环，其中循环最重要的主体部分是9c0和9c5的两条指令：
  - movzbl将src指针处的一个字节复制到r8d寄存器当中
  - 接着的mov指令将r8d处的一个字节复制给dst指针

```
00000000000009b0 <memcpy_simple>:
 9b0:   48 85 d2                test   %rdx,%rdx
 9b3:   48 89 f8                mov    %rdi,%rax
 9b6:   74 1a                   je     9d2 <memcpy_simple+0x22>
 9b8:   31 c9                   xor    %ecx,%ecx
 9ba:   66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
 9c0:   44 0f b6 04 0e          movzbl (%rsi,%rcx,1),%r8d
 9c5:   44 88 04 08             mov    %r8b,(%rax,%rcx,1)
 9c9:   48 83 c1 01             add    $0x1,%rcx
 9cd:   48 39 d1                cmp    %rdx,%rcx
 9d0:   75 ee                   jne    9c0 <memcpy_simple+0x10>
 9d2:   f3 c3                   repz retq
 9d4:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
 9db:   00 00 00
 9de:   66 90                   xchg   %ax,%ax
```

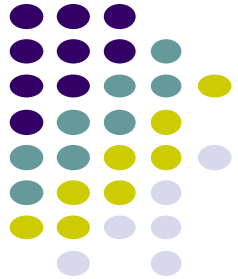
测试运行结果分析

- gcc在-O3优化级别下， simple memcpy被编译成非常复杂的二进制代码，其中较为重要的部分是循环当中的movdqa、movups指令，从每一次循环将rcx增加0x10可以知道，gcc使用了处理器的扩展指令来一次复制128位的内存，从而获得了更高的复制性能

```
00000000000009b0 <memcpy_simple>:
 9b0:   48 85 d2                test   %rdx,%rdx
 9b3:   48 89 f8                mov    %rdi,%rax
 9b6:   0f 84 9a 02 00 00       je     c56 <memcpy_simple+0x2a6>
 9bc:   48 8d 4f 10             lea    0x10(%rdi),%rcx
 9c0:   48 39 ce                cmp    %rcx,%rsi
 9c3:   48 8d 4e 10             lea    0x10(%rsi),%rcx
 9c7:   40 0f 93 c7             setae  %dil
 9cb:   48 39 c8                cmp    %rcx,%rax
 9ce:   0f 93 c1                setae  %cl
 9d1:   40 08 cf                or     %cl,%dil
 9d4:   0f 84 46 02 00 00       je     c20 <memcpy_simple+0x270>
 9da:   48 83 fa 16             cmp    $0x16,%rdx
 9de:   0f 86 3c 02 00 00       jbe    c20 <memcpy_simple+0x270>
 9e4:   48 89 f1                mov    %rsi,%rcx
 9e7:   4c 8d 5a ff             lea    -0x1(%rdx),%r11
 9eb:   48 f7 d9                neg    %rcx
 9ee:   83 e1 0f                and    $0xf,%ecx
 9f1:   48 8d 79 0f             lea    0xf(%rcx),%rdi
 9f5:   49 39 fb                cmp    %rdi,%r11
 9f8:   0f 82 42 02 00 00       jb     c40 <memcpy_simple+0x290>
 9fe:   48 85 c9                test   %rcx,%rcx
 a01:   53                      push   %rbx
 a02:   0f 84 50 02 00 00       je     c58 <memcpy_simple+0x2a8>
```

```
be0:   66 0f 6f 04 0e          movdqa (%rsi,%rcx,1),%xmm0
be5:   49 83 c0 01             add    $0x1,%r8
be9:   41 0f 11 04 09          movups %xmm0,(%r9,%rcx,1)
bee:   48 83 c1 10             add    $0x10,%rcx
```

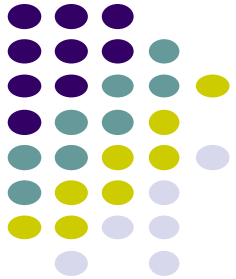
实验总结



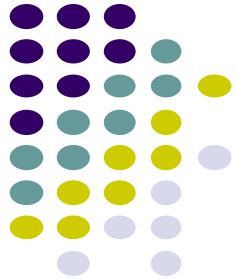
- 在不同的计算机上进行实验，结果有相当大的差异：
  - 指令数最少的程序未必运行得最快，甚至它们可能运行的相当慢，例如在AMD EPYC系列服务器处理器上
  - 在不同的CPU上，以及不同编译选项下，memcpy性能差距可能非常大

（注意：这里的实验结果仅对复制1MB-128MB内存有效）

实验总结

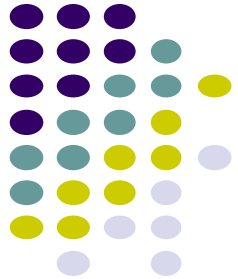


- 对于memcpy的性能分析，有以下结论：
  1. “什么样代码运行的更快”是一个很复杂的问题，和处理器的微架构、缓存、内存布局、多处理器等都有关系。一个例子是现代处理器在一个时钟周期里可以执行超过一条指令。
  2. 不要做想当然和不成熟的优化。图灵奖获得者Knuth曾经说过：Premature Optimization is the root of all evil —— 不成熟的优化是万恶之源。虽然这句话有些夸张，但是它确实是性能调优中的一个重要定律，应当相信我们的编译器，仅在必要的时候才进行性能调优。
  3. 用好工具、仔细设计实验、小心分析结果、分析时对症下药，才能实现好的性能分析。



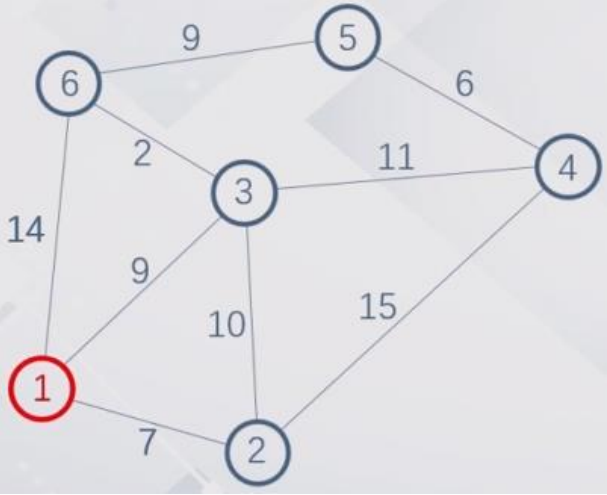
性能分析工具

概述



- 在本节中，我们通过一个多优化Dijkstra算法实现的性能分析案例，来展示性能分析工具的使用

Dijkstra算法



- Dijkstra算法用于求解有向或无向图中的单元最短路径
- 想象无向图中的每一个节点是一个石子，而每一条长度是w的边都是一根绳子，这根绳子可以被压缩，但是最多只能被拉长到w长度
- Dijkstra算法模拟了把石子和绳子放在桌上的过程：我们首先缓缓提起作为起点的某个节点（石子），然后第1个被从桌面上提起的节点就是距离起点最短的节点，第2个被提起的节点就是距离起点次近的节点，以此类推就可以得到一系列距离起点最短的节点和到达该节点的路径

Dijkstra算法实现程序

- 作为性能分析的目标程序，首先调用**build_graph**函数建立一个包含100万个节点、每个节点有100条出边的随机图，然后在随机图上调用**Dijkstra**算法，求解从编号为0的结点到编号为1的结点的最短路径

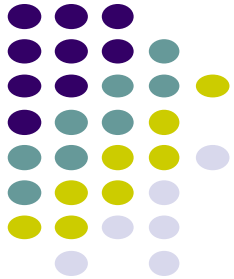
```
4  #include <string>
5  #include <iostream>
6
7  void build_graph(int n, int d);
8  std::string dijkstra(int src, int dst);
9
10 int main() {
11   build_graph(1000000, 100);
12   std::cout << dijkstra(0, 1) << std::endl;
13 }
```

Dijkstra算法实现程序

- build_graph函数首先设置随机数种子为确定值，然后对于每一个节点生成100条出边，每一条出边的权值设为0~999中的随机取值

```
25 void build_graph(int n, int d) {
26   std::srand(1); // bad but fast RNG
27   edges.resize(n);
28   for (int x = 0; x < n; x++) {
29     for (int i = 0; i < d; i++) {
30       unsigned int y = std::rand() % n;
31       unsigned int w = std::rand() % 1000;
32       edges[x].push_back(Edge(x, y, w));
33     }
34   }
35 }
```

Dijkstra算法实现程序

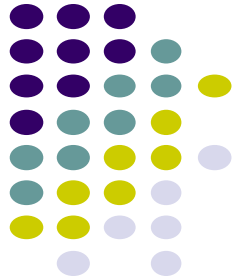


- Dijkstra算法的主要思想是维护一个优先队列，优先队列中存储每一个节点和它已知的最短路径值，并按照最短路径从小到大排序

- 初始时优先队列中仅有一个节点，编号为0，权值为0

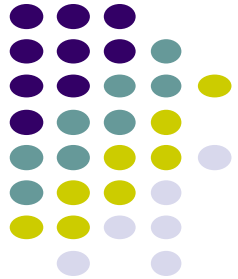
```
46 std::string dijkstra(int src, int dst) {
47   std::priority_queue< ipair, std::vector<ipair>, std::greater<ipair> > pq;
48   std::vector<int> dist(edges.size(), INF), route(edges.size(), -1);
49   std::vector<bool> visited(edges.size(), false);
50
51   pq.push(std::make_pair(0, src));
52   dist[src] = 0;
53
54   while (!pq.empty()) {
55     int dist_u = pq.top().first, u = pq.top().second; pq.pop();
56
57     for (const auto &edge: edges[u]) {
58       int v = edge.y, w = edge.w;
59       if (dist[u] + w < dist[v]) {
60         dist[v] = dist[u] + w;
61         route[v] = u;
62         pq.push(std::make_pair(dist[v], v));
63       }
64     }
65   }
66
67   return trace(route, dst);
68 }
```

Dijkstra算法实现程序



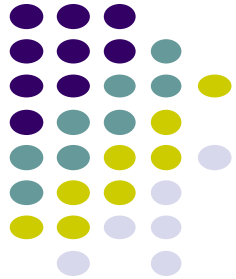
- 在每一轮循环中，从优先队列中取出当前已知最短路径的节点和它的权值，并将该节点从优先队列中删除。然后，对于该节点的每一条出边，检查是否可以由其构造比当前已知路径更短的新路径。如果能够找到更短的新路径，则用其更新最短路径，并且将该节点重新加入到优先队列中，最后返回最短路径的值

Dijkstra算法实现性能分析



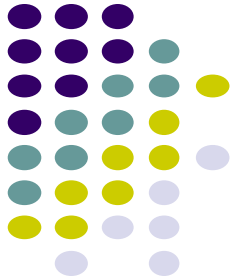
- 现在用Linux **Perf**性能分析工具，对Dijkstra算法实现进行性能分析
- Perf是一个profiler，具体来说它在运行中不断采样当前程序执行的指令，从而获得程序各部分代码的执行频率

Dijkstra算法实现性能分析



- Perf profiler的基本原理是在中断发生时，操作系统保存进程的寄存器现场，此寄存器现场包含了IP寄存器，即当前执行的指令地址，将这个指令记录下来，最后就可以推算出程序各部分执行的频率
- 现代处理器中的PMU不仅可以加速这样的采样，还可以提供更细致的profiling data，例如cache miss、分支计数、分支预测信息等

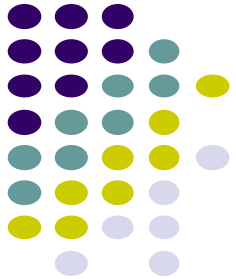
perf工具



- **perf**是Linux性能分析工具
  - Manual page手册中描述了perf相关工具的命令，如perf-stat、perf-top、perf-record、perf-report、perf-list等

```
PERF(1)                                    perf Manual                                    PERF(1)

NAME
       perf - Performance analysis tools for Linux

SYNOPSIS
       perf [--version] [--help] [OPTIONS] COMMAND [ARGS]

OPTIONS
       --debug
           Setup debug variable (see list below) in value range (0, 10). Use like: --debug verbose # sets verbose = 1
           --debug verbose=2 # sets verbose = 2

               List of debug variables allowed to set:
                 verbose          - general debug messages
                 ordered-events   - ordered events object debug messages
                 data-convert     - data convert command debug messages
```

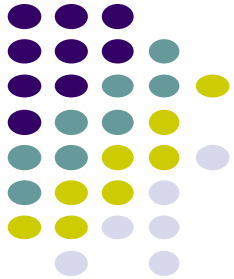
perf工具



- 为对Dijkstra算法进行性能分析，可以首先使用**perf-record**对程序的执行轨迹进行记录。在程序运行过程中，perf性能分析工具会结合软件和硬件，对运行的程序进行profiling，得到一定的profiling结果

```
$ perf record -g ./dijkstra
0 → 947739 → 768714 → 615887 → 213897 → 158540 → 576385 → 576212 → 819426 → 277894 → 919609 → 642761 → 123554 → 1
[ perf record: Woken up 12 times to write data ]
[kernel.kallsyms] with build id 34b344d82ad4582ddf3baaee3e87584ef803025b not found, continuing without symbols
[ perf record: Captured and wrote 3.036 MB perf.data (35453 samples) ]
```

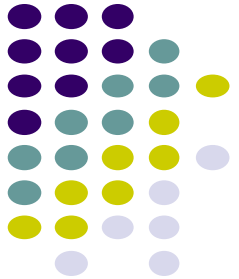
perf工具



- 可以接着使用**perf-report**命令，对程序运行的性能进行分析
  - 例如，可以展开性能分析的某一个结果，看到在某次运行中Dijkstra算法的具体实现耗费了65.93%的时间，而build_graph随机生成整个图的代码使用了20%的时间。其中，Dijkstra算法花费最多的是**堆操作**，这符合我们对Dijkstra算法的预期。另外还可以查看堆操作中相应的汇编指令

```
$ perf report
Samples: 35K of event 'cycles:u', Event count (approx.): 22138001439
  Children      Self  Command   Shared Object        Symbol
-   96.81%     0.00%  dijkstra  [unknown]            [k] 0×09f6258d4c544155
     0×9f6258d4c544155
     __libc_start_main
   - main
      - 65.93% dijkstra[abi:cxx11]
           17.80% std::__adjust_heap<__gnu_cxx::__normal_iterator<std::pair<int, int>*, std::vector<std::pair<int, int>, std::a
           0.93% 0×ffffffff868015e0
           0.81% std::__push_heap<__gnu_cxx::__normal_iterator<std::pair<int, int>*, std::vector<std::pair<int, int>, std::allo
      - 19.74% build_graph
           2.40% std::vector<Edge, std::allocator<Edge> >::_M_realloc_insert<Edge>
        6.34% __random
        3.85% __random_r
        0.86% rand
```

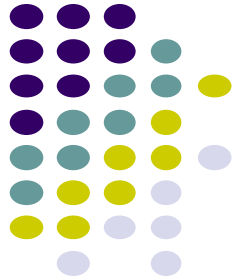
perf工具



- 我们发现，堆操作中耗时最大的、即统计达63.32%的时间都花在operator小于的比较操作中。因此，如果我们希望对Dijkstra算法的性能进行分析，就需要减少比较的次数

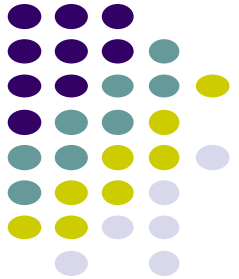
```
         |       template<typename _T1, typename _T2>
         |         inline _GLIBCXX_CONSTEXPR bool
         |         operator<(const pair<_T1, _T2>& __x, const pair<_T1, _T2>& __y)
         |         { return __x.first < __y.first
  0.10   |         mov    (%rax),%r13d
 63.32   |         lea    (%rdi,%rsi,8),%r8
  0.20   |         mov    (%r8),%r9d
         |                    || (!(__y.first < __x.first) && __x.second < __y.second); }
  7.91   |         cmp    %r9d,%r13d
  0.36   |       ↓ jg     2277 <void std::__adjust_heap<__gnu_cxx::__normal_iterator<std::pair<int, int>*, std::vector<std::pair<int,
         |       ↓ jl     226e <void std::__adjust_heap<__gnu_cxx::__normal_iterator<std::pair<int, int>*, std::vector<std::pair<int,
  1.42   |         mov    0x4(%r8),%r14d
  1.34   |         cmp    %r14d,0x4(%rax)
  0.17   |       ↓ jg     2277 <void std::__adjust_heap<__gnu_cxx::__normal_iterator<std::pair<int, int>*, std::vector<std::pair<int,
  1.95   | 6e:     mov    %rax,%r8
  3.64   |         mov    %r13d,%r9d
```

性能分析和优化



- 经过前述分析，我们发现该Dijkstra算法实现程序的性能瓶颈在于堆操作以及堆操作中的比较。因此，算法优化的主要目标就是减少堆操作和堆操作中的比较

性能分析和优化



- 观察代码，可以发现代码中有两个算法性的缺陷
  - 首先，Dijkstra算法在目标节点首次出队时，就已找到了起点到终点的最短路径。因此，可以在出队节点等于终止目标节点时，使用break退出循环以可以减少后续的堆操作
  - 另外，还可以发现在该Dijkstra算法实现中，节点可能会被反复入队，并且在反复入队后，一个节点也可能多次在55行被出队，然而按照Dijkstra算法，一个节点只在首次出队时需要更新其最短路径。因此，当节点不是首次出队时，可以直接忽略此次的更新，并在dist_u不等于已知最短路径的情况下，直接终止循环

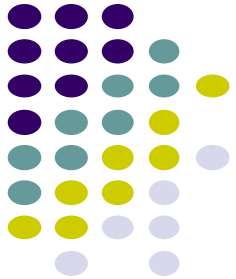
性能分析和优化

```
46 std::string dijkstra(int src, int dst) {
47   std::priority_queue< ipair, std::vector<ipair>, std::greater<ipair> > pq;
48   std::vector<int> dist(edges.size(), INF), route(edges.size(), -1);
49   std::vector<bool> visited(edges.size(), false);
50
51   pq.push(std::make_pair(0, src));
52   dist[src] = 0;
53
54   while (!pq.empty()) {
55     int dist_u = pq.top().first, u = pq.top().second; pq.pop();
56
57     if (u == dst) break;
58     if (dist_u != dist[u]) continue;
59
60     for (const auto &edge: edges[u]) {
61       int v = edge.y, w = edge.w;
62       if (dist[u] + w < dist[v]) {
63         dist[v] = dist[u] + w;
64         route[v] = u;
65         pq.push(std::make_pair(dist[v], v));
66       }
67     }
68   }
```

退出循环

终止循环

性能分析和优化

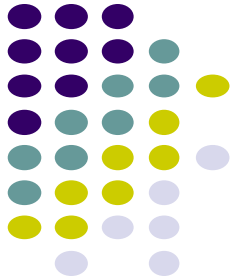


- 增加这两条优化后，再次编译并运行程序，可以看到程序的运行时间得到了一定程度的减少，同时Dijkstra算法得到了与之前完全相同的结果
- 使用perf record对程序进行性能分析并使用perf report查看性能分析的结果
  - 展开main函数中函数调用的关系，可以发现Dijkstra算法已经不是该程序的性能瓶颈：有41.84%的时间花费在建图上，并且建图上花费时间最多的是vector的插入
  - Dijkstra算法此时只占27.83%的时间，而其瓶颈主要还在堆操作上。如果我们查看汇编代码，堆操作的比较依然是性能的瓶颈

```
Samples: 18K of event 'cycles:u', Event count (approx.): 10749949667
  Children      Self  Command   Shared Object        Symbol
-   91.82%     0.00%  dijkstra  [unknown]            [k] 0x09c6258d4c544155
     0x9c6258d4c544155
     __libc_start_main
   - main
      - 41.84% build_graph
           4.85% std::vector<Edge, std::allocator<Edge> >::_M_realloc_insert<Edge>
           1.13% cfree@GLIBC_2.2.5
           0.69% malloc
      - 27.83% dijkstra[abi:cxx11]
           5.32% std::__adjust_heap<__gnu_cxx::__normal_iterator<std::pair<int, int>*, std::vector<std::pair<int, int>, std::al
           1.36% std::__push_heap<__gnu_cxx::__normal_iterator<std::pair<int, int>*, std::vector<std::pair<int, int>, std::allo
           0.99% 0xffffffff868015e0
        12.53% __random
        7.20% __random_r
        2.13% rand
```

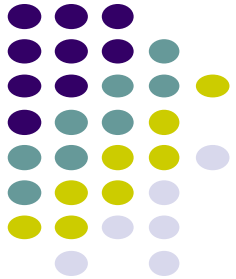
```
        operator<(const pair<_T1, _T2>& __x, const pair<_T1, _T2>& __y)
        { return __x.first < __y.first
 0.65     mov    (%rax),%r13d
66.27     lea    (%rdi,%rsi,8),%r8
 0.11     mov    (%r8),%r9d
                     || (!(__y.first < __x.first) && __x.second < __y.second); }
10.09     cmp    %r9d,%r13d
 0.33   ↓ jg     22a7 <void std::__adjust_heap<__gnu_cxx::__normal_iterator<std::pair<int, int>*, std::vector<std::pair<int,
        ↓ jl     229e <void std::__adjust_heap<__gnu_cxx::__normal_iterator<std::pair<int, int>*, std::vector<std::pair<int,
 0.65     mov    0x4(%r8),%r14d
 1.63     cmp    %r14d,0x4(%rax)
 0.87   ↓ jg     22a7 <void std::__adjust_heap<__gnu_cxx::__normal_iterator<std::pair<int, int>*, std::vector<std::pair<int,
 0.87 6e: mov    %rax,%r8
 2.71     mov    %r13d,%r9d
```

总结



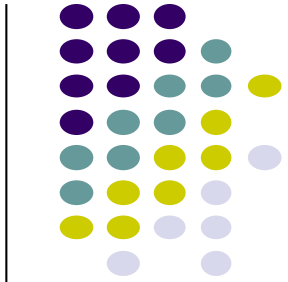
- 基于前述对Dijkstra算法实现的性能分析与优化案例，可以看出
  1. 算法的改进是至关重要的，能够大幅提升程序的性能
  2. 性能问题很可能发生在出乎意料的地方，所以应当正确使用profiler工具，避免盲目地进行性能优化

总结



- 本周课程讲解了性能分析，也就是设法获知程序的哪一部分花费了多少时间

- 在性能分析的过程中，有三点基本原则：
  1. 应当设置合理的workload，以代表应用程序的工作场景
  2. 在运行程序过程中，应记录下相应的运行时信息
  3. 应当针对这些运行时信息进行针对性的优化，避免盲目优化



END