编译原理

实验指导手册

V1.0

华为技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。

本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

华为技术有限公司

地址： 深圳市龙岗区坂田华为总部办公楼 邮编：518129

网址： https://e.huawei.com

目录

前　言 3
简介 3
内容描述 3
读者知识背景 3
实验环境说明 3
1 使用毕昇编译器进行 SIMD 矢量优化 4
1.1 实验目的 4
1.2 实验设备 4
1.3 实验原理 4
1.3.1 矢量化 4
1.3.2 编译器自动矢量化 5
1.4 实验任务操作指导 5
1.5 思考题 6
2 基于 openEuler 构建第三方库的实验 7
2.1 实验目的 7
2.2 实验设备 7
2.3 实验原理 7
2.4 实验任务操作指导 7
2.5 思考题 8
3 Autotuner 9
3.1 实验目的 9
3.2 实验设备 9
3.3 实验原理 9
3.4 实验任务操作指导 10
3.5 思考题 12
4 PGO 13
4.1 实验目的 13
4.2 实验设备 13
4.3 实验原理 13
4.4 实验任务操作指导 14

4.5 思考题 17

5 术语和缩略语表 18

前 言

简介

本实验指导手册为《编译原理》课程的实验指导，本课程的实验环境依赖毕昇编译器和鲲鹏920处理器，适用于对编译原理、操作系统、计算机原理等有一定基础，并希望深入熟悉掌握编译优化技术原理和效果的读者。

内容描述

本实验指导手册共包含4个实验，以毕昇编译器为基础，探究从编译器或者编译技术角度来向大家介绍编译优化相关的技术，包括SIMD、PGO、Autotuner等。

- 实验一：使用毕昇编译器进行SIMD矢量优化，获得更好的应用性能。
- 实验二：使用毕昇编译器构建openEuler第三方库，掌握毕昇编译openEuler库的方法。
- 实验三：使用毕昇编译器进行autotuner自动化调优，提升程序性能。
- 实验四：使用毕昇编译器进行PGO反馈优化，深度提升程序性能。

读者知识背景

本课程为编译原理基础课程，为了更好地掌握本书内容，阅读本书的读者应首先具备以下基本条件：

具备基本的Linux命令能力；

了解毕昇编译器并会使用毕昇编译器；

具备基本的汇编语言编码能力；

具备基本的C/C++语言编码能力。

实验环境说明

华为鲲鹏云主机，openEuler 20.03操作系统；

毕昇编译器；

每套实验环境可供一名学员上机操作。

1 使用毕昇编译器进行 SIMD 矢量优化

1.1 实验目的

探究如何使用毕昇编译器进行 SIMD 矢量优化，以获得更好的程序性能表现。

1.2 实验设备

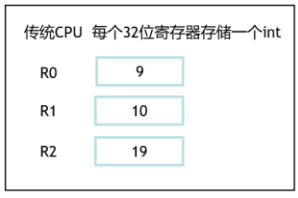
表1-1 实验设备清单

操作系统	CPU类型
openEuler 20.03 LTS SP1	鲲鹏920处理器
CentOS Linux release 7.6 aarch64	鲲鹏920处理器
CentOS Linux release 8.1 aarch64	鲲鹏920处理器
Ubuntu 18.04	鲲鹏920处理器

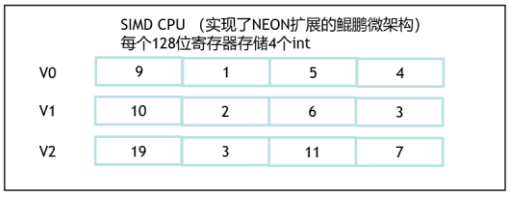
1.3 实验原理

1.3.1 矢量化

SIMD (Single Instruction Multiple Data) 即单指令流多数据流，是一种采用一个控制器来控制多个处理器，同时对一组数据（又称“数据向量”）中的每一个分别执行相同的操作从而实现空间上的并行性的技术。简单来说就是一个指令能够同时处理多个数据。




```
add r0, r1, r2
```




```
add v0.4s, v1.4s, v2.4s
```

图1-1 硬件基础：ARM NEON 指令集扩展

NEON 是 ARM 平台上的一种基于 SIMD 思想的技术，可以提供 128bit 的向量运算能力。NEON 技术从 ARMv7 开始被采用，鲲鹏 920 微架构已经使能 ARM NEON 指令集，可以进行矢量化计算操作。

1.3.2 编译器自动矢量化

在并行计算中，自动矢量化是自动并行化的一种特殊情况。在编译过程中，可以自动将计算机程序中一次计算操作处理一个数据的标量实现，转换为一个操作同时处理多个数据的矢量实现，以实现自动矢量优化。这种自动矢量优化在一些具有循环计算的场景下提升显著。

1.4 实验任务操作指导

使用毕昇编译器时添加-O2 或-O3 选项可以开启自动矢量优化，使用 #pragma clang loop vectorize(disable)可以强制关闭相应矢量优化。

test.c：

```
#include <math.h>
#include <time.h>

void f(int N, float *arr1, float *arr2, float *result){
// 删除或注释 pragma 行即可开启自动矢量化
#pragma clang loop vectorize(disable)
    for(int i = 0; i < N; i++) {
        result[i] = arr1[i] + arr2[i];
    }
}

int main()
{
    clock_t start, finish;
    int loop = 10000;
    int len = 1000000;
    float *a = (float*)malloc(sizeof(float) * len);
    float *b = (float*)malloc(sizeof(float) * len);
    float *d = (float*)malloc(sizeof(float) * len);
    for (int i = 0; i < len; i++) {
        //随机生成数组
        a[i] = rand() * 1.8570f - 2.0360f;
        b[i] = rand() * 8.7680f - 6.3840f;
    }

    start = clock();
```

```
    for (int i = 0; i < loop; i++) {
        f(len, a, b, d);
    }

    finish = clock();

    double time = (double)(finish - start) / CLOCKS_PER_SEC;
    printf("time: %f s\n", time);

    a[0] = d[0];//调用结果 d 防止编译器将计算过程优化掉
    free(a);
    free(b);
    free(d);
    return 0;
}
```

编译运行

```
clang test.c -O2 && ./a.out
```

去掉 pragma 语句编译运行，比较运行时间差异。

注释后：1.930684 s

注释前：1.930684 s

```
$ clang    test.c -O2 && ./a.out
time: 3.851565 s
$ clang    test.c -O2 && ./a.outtime: 1.925672 s
```

1.5 思考题

假设每个 NEON 指令可以处理 128 bit（4 个 float 类型）的数据，如果一个循环需要处理的 float 类型数据长度不是 4 的倍数，这部分数据该如何处理?

参考答案：

编译器会为循环的尾块生成标量的循环，即单步迭代运算。毕昇编译器的后续版本将会支持 SVE 指令集，支持对变长数据的矢量化，这样对于循环尾块依然可以矢量化。

2 使用毕昇编译器构建 openEuler 第三方库

2.1 实验目的

探究如何使用毕昇编译器构建 openEuler 第三方库，掌握毕昇编译 openEuler 库的方法。

2.2 实验设备

表2-1 实验设备清单

操作系统	CPU类型
openEuler 20.03 LTS SP1	鲲鹏920处理器

2.3 实验原理

RPM 是 Redhat Package Manager 的简称，是由 redhat 公司研制，用在 Linux 系统下的系统包管理工具。RPM 包目的：是使软件包的安装和卸载过程更容易，简化软件包的建立分发过程，并能用于不同的体系结构，RPM 系统已成为现在 Linux 系统下包管理工具事实上的标准，并且已经移植到很多商业的 unix 系统之下。rpm 打包可以通过编写 spec 文件，使用 rpmbuild 来完成一个 rpm 的打包，进而自动化的完成第三方库的构建。

2.4 实验任务操作指导

环境上安装 rpmbuild 工具。

这里用 zlib-1.2.11-19.oe2203.src.rpm 第三方库作为示范。

```
# 获取 rpm 安装依赖
rpm_path=/home/zlib-1.2.11-19.oe2203.src.rpm
deps=$(rpm -qp $rpm_path --requires | awk -v dep="$1" "NF==1{print $dep}")
# 安装依赖软件
yum install $deps -y
# 安装 src 包
rpm -ivh $rpm_path
# 设置毕昇编译器环境变量
export PATH=$BOOLE_HOME/bin:$PATH
export LD_LIBRARY_PATH=$BOOLE_HOME/lib:$LD_LIBRARY_PATH
```

```
cd $BOOLE_HOME/bin
ln -s clang gcc;ln -s clang g++;ln -s flang gfortran;ln -s clang cc

cd -
# rpmbuild 进行构建
rpmbuild    -ba /root/rpmbuild/SPECS/zlib.spec 2>&1 |tee zlib.log
```

查看是否有编译的兼容性问题并解决处理。

2.5 思考题

rpm 与 src.rpm 区别?

参考答案:

xxxxxxxxx.rpm　　<==RPM 的格式，已经经过编译且包装完成的 rpm 档案;

xxxxx.src.rpm　　<==SRPM 的格式，包含未编译的原始码资讯。

例如 rp-pppoe-3.1-5.i386.rpm 命名的意义为:

rp-pppoe - 3.1 - 5 .i386 .rpm

套件名称　套件的版本资讯 释出的次数 适合的硬体平台 副档名

.src.rpm 结尾的，这类软件包是包含了源代码的 rpm 包，在安装时需要进行编译。

3 使用毕昇编译器进行 autotuner 自动化调优

3.1 实验目的

探究如何基于毕昇编译器使能 autotuner 自动调优功能，可以更细粒度地进行程序性能优化。

3.2 实验设备

表3-1 实验设备清单

操作系统	CPU类型
openEuler 20.03 LTS SP1	鲲鹏920处理器
CentOS Linux release 7.6 aarch64	鲲鹏920处理器
CentOS Linux release 8.1 aarch64	鲲鹏920处理器
Ubuntu 18.04	鲲鹏920处理器

3.3 实验原理

调优流程（如图 Autotuner 调优流程所示）由两个阶段组成：初始编译阶段 （initial compilation）和调优阶段 （tuning process）。

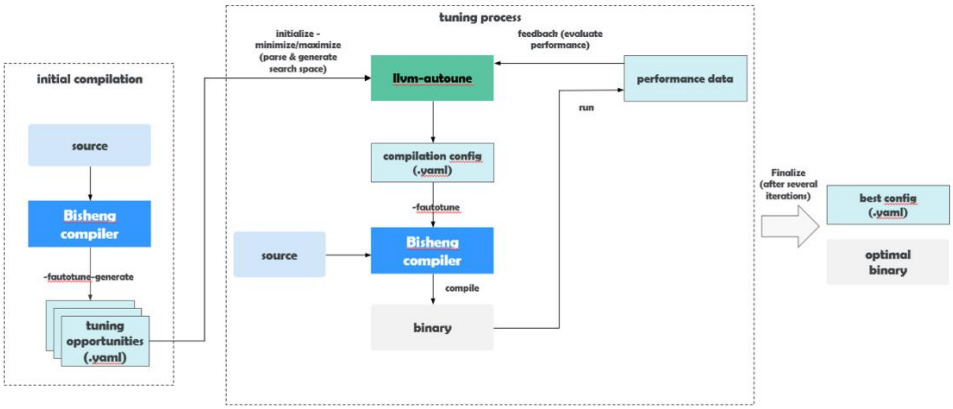


图3-1 Autotuner 调优流程

初始编译阶段

初始编译阶段发生在调优开始之前，Autotuner 首先会让编译器对目标程序代码做一次编译，在编译的过程中，毕昇编译器会生成一些包含所有可调优结构的 YAML 文件，告诉我们在这个目标程序中哪些结构可以用来调优，比如文件（module）、函数（function）、循环（loop）。例如，循环展开是编译器中最常见的优化方法之一，它通过多次复制循环体代码，达到增大指令调度的空间，减少循环分支指令的开销等优化效果。若以循环展开次数（unroll factor）为对象进行调优，编译器会在 YAML 文件中生成所有可被循环展开的循环作为可调优结构。

调优阶段

当可调优结构顺利生成之后，调优阶段便会开始：Autotuner 首先读取生成好的可调优结构的 YAML 文件，从而产生对应的搜索空间，也就是生成针对每个可调优代码结构的具体的参数和范围；调优阶段会根据设定的搜索算法尝试一组参数的值，生成一个 YAML 格式的编译配置文件（compilation config），从而让编译器编译目标程序代码产生二进制文件；最后 Autotuner 将编译好的文件以用户定义的方式运行并取得性能信息作为反馈；经过一定数量的迭代之后，Autotuner 将找出最终的最优配置，生成最优编译配置文件，以 YAML 的形式储存。

3.4 实验任务操作指导

Autotuner 已包括在毕昇编译器的发布软件包里。若您已经安装毕昇编译器，只需配置毕昇编译器的环境变量即可直接使用。否则，请先安装毕昇编译器。

配置毕昇编译器的环境变量

```
export PATH=/opt/compiler/bisheng-compiler-1.3.3-aarch64-linux/bin:$PATH
```

注意：以上步骤是以/opt/compiler 目录举例，若您的安装目录不同，请以实际目录为准。

测试是否安装成功

使能 Python3.10 以上版本

执行如下命令：

```
install-autotuner.shllvm-autotune -hauto-tuner -h
```

执行完毕后，界面如果显示相应的帮助信息则表示安装成功。

用户可根据自身需求，编写调优脚本。我们将以 coremark 为示例展示如何运行自动调优，毕昇编译器的发布包里没有自带 coremark，请从社区获取 coremark。以下为以 20 次迭代调优 coremark 的脚本示例：

```
export AUTOTUNE_DATADIR=/tmp/autotuner_data/
CompileCommand="clang -Ilinux64 -I. -g -DFLAGS_STR=\"\"    -DITERATIONS=300000    core_list_join.c core_main.c core_matrix.c core_state.c core_util.c linux64/core_portme.c -O2 -o coremark"
```

```
$CompileCommand -fautotune-generate;
llvm-autotune minimize;
for i in $(seq 20)
do
   $CompileCommand -fautotune ;
   time=`/usr/bin/time -p ./coremark    0x0 0x0 0x66 300000    2>&1 1>/dev/null | grep real | awk '{print $2}'`;
   echo "iteration: " $i "cost time:" $time;
   llvm-autotune feedback $time;
done
llvm-autotune finalize;
```

以下为分步说明：

步骤 1　配置环境变量

使用环境变量 AUTOTUNE_DATADIR 指定调优相关的数据的存放位置。

```
export AUTOTUNE_DATADIR=/tmp/autotuner_data/
```

步骤 2　初始编译

添加毕昇编译器选项-fautotune-generate，编译生成可调优代码结构。

```
cd    examples/coremark/
clang -Ilinux64 -I. -DFLAGS_STR=\"\"    -lrt"\" -DITERATIONS=300000 core_list_join.c core_main.c core_matrix.c
core_state.c core_util.c linux64/core_portme.c -O2 -g -o coremark -fautotune-generate
```

注意：建议仅将此选项应用于需要重点调优的热点代码文件。若应用的代码文件过多（超过 500 个文件），则会生成数量庞大的可调优代码结构的文件，进而可能导致 3 的初始化时间长（可长达数分钟）；以及巨大的搜索空间导致的调优效果不显著，收敛时间长等问题。

步骤 3　初始化调优

运行 llvm-autotune 命令，初始化调优任务。生成最初的编译配置供下一次编译使用。

```
llvm-autotune minimize
```

minimize 表示调优目标，旨在最小化指标（例如程序运行时间）。也可使用 maximize，旨在最大化指标（例如程序吞吐量）。

步骤 4　调优编译

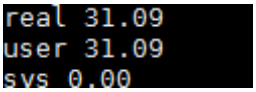
添加毕昇编译器选项-fautotune，读取当前 AUTOTUNE_DATADIR 配置并编译。

```
clang -Ilinux64 -I. -DFLAGS_STR=\"\"    -lrt"\" -DITERATIONS=300000 core_list_join.c core_main.c core_matrix.c
core_state.c core_util.c linux64/core_portme.c -O2 -g -o coremark -fautotune
```

步骤 5　性能反馈

用户运行程序，并根据自身需求获取性能数字，使用 llvm-autotune feedback 反馈。例如，如果我们想以 coremark 运行速度为指标进行调优，可以采用如下方式：

```
time -p ./coremark    0x0 0x0 0x66 300000    2>&1 1>/dev/null
```




```
llvm-autotune feedback 31.09
```

注意：建议在使用 llvm-autotune feedback 之前，先验证步骤 4 是否正常，及编译好的程序是否运行正确。若出现编译或者运行异常的情况，请输入相应调优目标的最差值（例如，调优目标为 minimize，可输入 llvm-autotune feedback 9999；maximize 可输入 0 或者 -9999）。若输入的性能反馈不正确，可能会影响最终调优的结果。

步骤 6　调优迭代

根据用户设定的迭代次数，重复步骤 4 和步骤 5 进行调优迭代。

步骤 7　结束调优

进行多次迭代后，用户可选择终止调优，并保存最优的配置文件。配置文件会被保存在环境变量 AUTOTUNE_DATADIR 指定的目录下。

```
llvm-autotune finalize
```

步骤 8　最终编译

使用步骤 7 得到最优配置文件，进行最后编译。在环境变量未改变的情况下，可直接使用-fautotune 选项：

```
clang -Ilinux64 -I. -DFLAGS_STR="\"    -lrt"\" -DITERATIONS=300000 core_list_join.c core_main.c core_matrix.c
core_state.c core_util.c linux64/core_portme.c -O2 -g -o coremark -fautotune
```

或者使用 -mllvm -auto-tuning-input= 直接指向配置文件。

```
clang -Ilinux64 -I. -DFLAGS_STR="\"    -lrt"\" -DITERATIONS=300000 core_list_join.c core_main.c core_matrix.c
core_state.c core_util.c linux64/core_portme.c -O2 -g -o coremark -mllvm -auto-tuning-
input=/tmp/autotuner_data/config.yaml
```

3.5 思考题

Autotuner 适合什么场景的应用?

参考答案：

Autotuner 使用循环通过搜索算法对搜索空间进行分析优化，适合运行时间较短，数据结果稳定的应用来使用。

4 使用毕昇编译器进行 PGO 反馈优化

4.1 实验目的

- 了解反馈优化的概念
- 熟悉毕昇编译器使用反馈优化的流程
- 了解反馈优化对二进制的影响

4.2 实验设备

表4-1 实验设备清单

操作系统	CPU类型
openEuler 20.03 LTS SP1	鲲鹏920处理器
CentOS Linux release 7.6 aarch64	鲲鹏920处理器
CentOS Linux release 8.1 aarch64	鲲鹏920处理器
Ubuntu 18.04	鲲鹏920处理器

4.3 实验原理

PGO 是 Profile Guided Optimization 的缩写，即档案导引优化。是一种自适应优化手段，不需要对软件代码本身做出改进，即可获得性能的提升，目前已经应用于 GCC、VC++、Clang 等常见的编译器中。

配置文件信息可以帮助编译器更好地优化。例如，知道分支被非常频繁地执行，有助于编译器在**排序基本块**时做出更好的决策；知道函数 foo 的调用频率比另一个函数条的调用频率更高，有助于**内联**。此外，配置文件还能指导编译器更好的进行**冷热分区、函数分区、寄存器分配、循环展开和剥离、增强代码和数据的局部相关性、Switch 语句扩展**。建议使用**-O2** 以上的优化等级使用 **PGO**。

毕昇支持两种类型的 **PGO** 采样。一种是使用采样分析器，可以在牺牲很小运行时开销情况下生成配置文件。二是重新构建代码获取一个插桩版本的二进制以收集更详细的配置文件信息。

这两种配置文件都可以提供代码中指令的执行计数，以及关于所采取的分支和函数调用的信息。

使用时需要注意，插桩编译和反馈编译时的源代码要完全一致，有时编译选项也要一致，否则可能导致无法关联 profile 数据或做出糟糕的优化选择，最好是在代码通过了单元测试和回归测试后再使用。另外，若程序大部分时间花在函数调用和分支跳转时，**PGO** 优化收益大，若大部分时间在循环和计算时，**PGO** 收益较小。如图是 PGO 的优化过程：

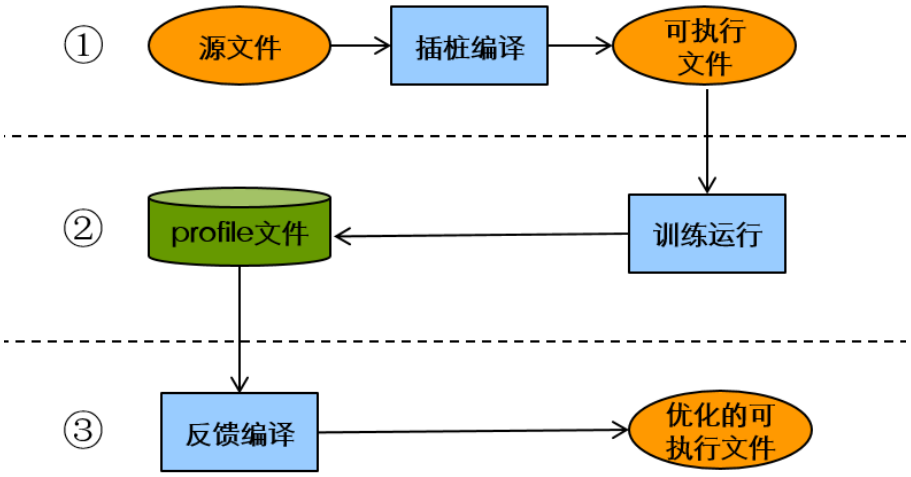


图4-1 PGO 优化过程

4.4 实验任务操作指导

步骤 1　获取未优化的情况

```
clang code.cc -o noOpt.exe
./noOpt.exe
# Average: 5736 ms
```

说明：默认使用**-O0**，是最安全，但也是性能最差的。

步骤 2　获取基准值

```
clang code.cc -O2 -o base.exe
./base.exe
# Average: 1482 ms
```

说明：相比于**-O3** 的一些激进不稳定的优化，**-O2** 是应用程序被推荐的优化等级，可以使用很多已知且稳定的优化。

步骤 3　测试 PGO 效果

```
clang++ -O2 -fprofile-generate=profile code.cc -o sample.exe
./sample.exe
llvm-profdata merge -output=code.profdata profile
clang++ -O2 -fprofile-use=code.profdata code.cc -o pgo.exe
./pgo.exe
#Average: 1440 ms
```

说明：相比于步骤 2 的结果，这个更进一步得拿到了一些收益。

步骤 4　对比使用 PGO 优化前后热点情况

```
perf record -e cycles -o base.perf.data ./base.exe
perf record -e cycles -o pgo.perf.data ./pgo.exe
perf report -i base.perf.data
perf report -i pgo.perf.data
```

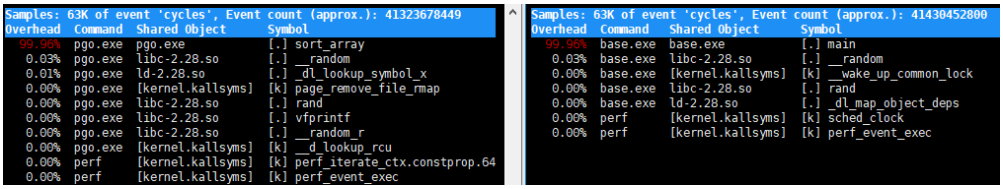


图4-2 对比使用 PGO 优化前后热点情况

说明：这个场景中，具体指令的热点高度相似，但是在函数层面，还是能看到，在对待函数的内联，使用 PGO 优化前后策略不一样(右图把 C 文件涉及的函数全部内联了)。

步骤 5　对比使用 PGO 优化前后冒泡排序函数 **bubble_sort** 的反汇编结果

```
gdb -batch -ex 'file base.exe' -ex 'disassemble bubble_sort' &> base.exe.dis
gdb -batch -ex 'file pgo.exe' -ex 'disassemble bubble_sort' &> pgo.exe.dis
vim -d base.exe.dis pgo.exe.dis
```

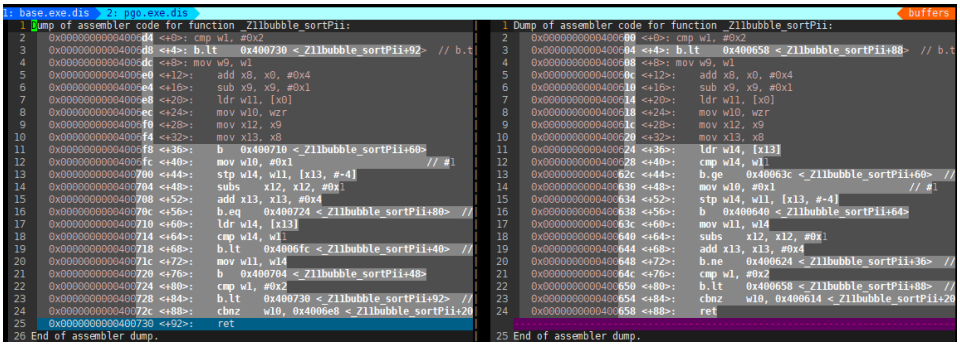


图4-3 对比反汇编结果

说明：生成二进制时，我们倾向于尽量减少跳转。代码中 **if (a[i] < a[i - 1])**表达式为真或假的概率影响了我们将交换指令放置位置的决策。图中可以看到的使用 PGO 前后跳转的策略是不一样的。

code.cc

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#define ARRAY_LEN 30000
static struct timeval tm1;
static inline void start() { gettimeofday(&tm1, NULL); }
static inline void stop() {
    struct timeval tm2;
    gettimeofday(&tm2, NULL);
    unsigned long long t =
            100 * (tm2.tv_sec - tm1.tv_sec) + (tm2.tv_usec - tm1.tv_usec) / 1000;
    printf("Average: %llu ms\n", t);
}
void bubble_sort(int *a, int n) {
    int i, t, s = 1;
    while (s) {
        s = 0;
        for (i = 1; i < n; i++) {
            if (a[i] < a[i - 1]) {
                t = a[i];
                a[i] = a[i - 1];
                a[i - 1] = t;
                s = 1;
            }
        }
    }
}
void sort_array() {
    printf("Bubble sorting array of %d elements\n", ARRAY_LEN);
    int data[ARRAY_LEN], i;
    for (i = 0; i < ARRAY_LEN; ++i) {
        data[i] = rand();
    }
    bubble_sort(data, ARRAY_LEN);
}
int main() {
    start();
    for (int i=0;i<10;i++)
        sort_array();
    stop();
    return 0;
}
```

4.5 思考题

反馈优化对于程序性能提升除了上文提到的一些点，还有很多的方向值得探索，你是否了解或能想到其他一些优化点?

参考答案:

反馈优化是社区最近的热门话题，关于它的探索一直在继续，比如 CSPGO 和 BOLT。前者在常规 PGO 的基础上考虑了函数调用上下文，对内联函和代码生成有更进一步的优化，后者是将优化后的二进制作为输入，使用 LLVM API 来反汇编、解析调试信息和链接以重新构建二进制，两者都能在常规 PGO 的基础上拿到更进一步的收益。

5 术语和缩略语表

缩略语	英文全称	中文释义
SIMD	Single Instruction Multiple Data	单指令流多数据流
PGO	Profile Guided Optimization	档案导引优化