计算机体系结构 lab 4

PB20111704 张宇昂

实验过程

1. 解压编译器：

```
xz -d gcc-linaro-6.4.1-2017.11-x86_64_aarch64-linux-gnu.tar.xz
tar -xf gcc-linaro-6.4.1-2017.11-x86_64_aarch64-linux-gnu.tar
```

2. 修改makefile文件，将路径替换为上面编译器所在路径，需要注意的是给出的makefile文件中编译器名称为`gcc-linaro-5.4.1-2017.11-x86_64_aarch64-linux-gnu`，需要修改为`gcc-linaro-6.4.1-2017.11-x86_64_aarch64-linux-gnu`

3. 对`daxpy.cc`中的函数执行循环展开，循环展开10次（由于在`daxpy.cc`中观察代码可知`N = 10000`）后，展开函数如下

```
void daxpy_unroll(double *X, double *Y, double alpha, const int N)
{
    for (int i = 0; i < N; i += 10)
    {
        Y[i] = alpha * X[i] + Y[i];
        Y[i+1] = alpha * X[i+1] + Y[i+1];
        Y[i+2] = alpha * X[i+2] + Y[i+2];
        Y[i+3] = alpha * X[i+3] + Y[i+3];
        Y[i+4] = alpha * X[i+4] + Y[i+4];
        Y[i+5] = alpha * X[i+5] + Y[i+5];
        Y[i+6] = alpha * X[i+6] + Y[i+6];
        Y[i+7] = alpha * X[i+7] + Y[i+7];
        Y[i+8] = alpha * X[i+8] + Y[i+8];
        Y[i+9] = alpha * X[i+9] + Y[i+9];
    }
}

void daxsbxpxy_unroll(double *X, double *Y, double alpha, double beta, const
int N)
{
    for (int i = 0; i < N; i += 10)
    {
        Y[i] = alpha * X[i] * X[i] + beta * X[i] + X[i] * Y[i];
        Y[i+1] = alpha * X[i+1] * X[i+1] + beta * X[i+1] + X[i+1] * Y[i+1];
        Y[i+2] = alpha * X[i+2] * X[i+2] + beta * X[i+2] + X[i+2] * Y[i+2];
        Y[i+3] = alpha * X[i+3] * X[i+3] + beta * X[i+3] + X[i+3] * Y[i+3];
        Y[i+4] = alpha * X[i+4] * X[i+4] + beta * X[i+4] + X[i+4] * Y[i+4];
        Y[i+5] = alpha * X[i+5] * X[i+5] + beta * X[i+5] + X[i+5] * Y[i+5];
        Y[i+6] = alpha * X[i+6] * X[i+6] + beta * X[i+6] + X[i+6] * Y[i+6];
        Y[i+7] = alpha * X[i+7] * X[i+7] + beta * X[i+7] + X[i+7] * Y[i+7];
        Y[i+8] = alpha * X[i+8] * X[i+8] + beta * X[i+8] + X[i+8] * Y[i+8];
        Y[i+9] = alpha * X[i+9] * X[i+9] + beta * X[i+9] + X[i+9] * Y[i+9];
    }
```

```
}

void stencil_unroll(double *Y, double alpha, const int N)
{
    for (int i = 1; i < N-10; i += 10)
    {
        Y[i] = alpha * Y[i-1] + Y[i] + alpha * Y[i+1];
        Y[i+1] = alpha * Y[i] + Y[i+1] + alpha * Y[i+2];
        Y[i+2] = alpha * Y[i+1] + Y[i+2] + alpha * Y[i+3];
        Y[i+3] = alpha * Y[i+2] + Y[i+3] + alpha * Y[i+4];
        Y[i+4] = alpha * Y[i+3] + Y[i+4] + alpha * Y[i+5];
        Y[i+5] = alpha * Y[i+4] + Y[i+5] + alpha * Y[i+6];
        Y[i+6] = alpha * Y[i+5] + Y[i+6] + alpha * Y[i+7];
        Y[i+7] = alpha * Y[i+6] + Y[i+7] + alpha * Y[i+8];
        Y[i+8] = alpha * Y[i+7] + Y[i+8] + alpha * Y[i+9];
        Y[i+9] = alpha * Y[i+8] + Y[i+9] + alpha * Y[i+10];
    }
    Y[N-9] = alpha * Y[N-10] + Y[N-9] + alpha * Y[N-8];
    Y[N-8] = alpha * Y[N-9] + Y[N-8] + alpha * Y[N-7];
    Y[N-7] = alpha * Y[N-8] + Y[N-7] + alpha * Y[N-6];
    Y[N-6] = alpha * Y[N-7] + Y[N-6] + alpha * Y[N-5];
    Y[N-5] = alpha * Y[N-6] + Y[N-5] + alpha * Y[N-4];
    Y[N-4] = alpha * Y[N-5] + Y[N-4] + alpha * Y[N-3];
    Y[N-3] = alpha * Y[N-4] + Y[N-3] + alpha * Y[N-2];
    Y[N-2] = alpha * Y[N-3] + Y[N-2] + alpha * Y[N-1];
}
```

从代码函数逻辑以及运行结果的sum相同可以看出，循环展开保持了原函数的逻辑

4. 修改`HPI.py`，按照实验文档中给出的逻辑，增加三个`HPI_FloatSimdFU`：

```
class HPI_FUPool(MinorFUPool):
    funcUnits = [HPI_IntFU(), # 0
        HPI_Int2FU(), # 1
        HPI_IntMulFU(), # 2
        HPI_IntDivFU(), # 3
        HPI_FloatSimdFU(), # 4
        HPI_MemFU(), # 5
        HPI_MiscFU(), # 6
        HPI_FloatSimdFU(), # 7
        HPI_FloatSimdFU(), # 8
        HPI_FloatSimdFU() # 9
        ]
```

5. 修改`Makefile`如下：

```
CC=/home/zya1412/gcc-linaro-6.4.1-2017.11-x86_64_aarch64-linux-gnu/bin/aarch64-linux-gnu-g++

daxpy-O1: daxpy.cc
    $(CC) -Iinclude --std=c++11 -static daxpy.cc util/m5/m5op_arm_A64.S -o daxpy -O1
daxpy-O3: daxpy.cc
    $(CC) -Iinclude --std=c++11 -static daxpy.cc util/m5/m5op_arm_A64.S -o daxpy -O3
clean:
    rm daxpy
```

实验结果及分析

修改后的`daxpy.cc`文件见`daxpy-changed.cc`，四次编译的结果分别对应为：

1. `time1`：未修改的`daxpy.cc`，编译器选项为O1
2. `time2`：修改过的`daxpy.cc`，编译器选项为O1
3. `time3`：修改过的`daxpy.cc`，编译器选项为O3

O1优化下的结果如下：

function	CPI	simTicks	committedInsts
daxpy	1.783888	35684000	80014
daxpy_unroll	2.199959	35206500	64013
daxsbxpxy	2.011040	60339750	120017
daxsbxpxy_unroll	2.314657	60189750	104015
stencil	1.962279	49055500	109995
stencil_unroll	1.948485	41890000	85995

O3优化下的结果如下：

function	CPI	simTicks	committedInsts
daxpy	1.822548	18235500	45022
daxpy_unroll	2.860901	17184000	30026
daxsbxpxy	2.071279	28492000	60023
daxsbxpxy_unroll	1.791096	17923500	45028
stencil	2.208470	33126500	69998
stencil_unroll	4.309769	36670750	49021

问题解答

Q1

从代码函数逻辑以及第一次编译和第二/三次编译的运行结果的sum相同可以看出，循环展开保持了原函数的逻辑

Q2

从CPI、simTicks、committedInst三个参数都可以看出，循环转开提升了性能，减少了Branch Hazard的数量

Q3

观察 `daxpy.cc` 可知，每个函数传入的N均为10000，经过我的测试，10次的性能是最优的，如果展开次数过少/过多都会影响程序性能，若展开次数过少，会导致消除的Branch Hazard较少，影响性能；若展开次数过多，那么运算过程中中间变量会很多,而计算机的寄存器个数是固定的,当变量个数超了寄存 器,那么变量只能存到栈中从而导致性能下降

Q4

经过对比第一次和第二次编译的结果（对原函数来说区别只有硬件改变），我个人认为对 `daxpy` 区别不是很大，但是对 `daxsbxpxy` 和 `stencil` 影响较大；我认为添加硬件可以减少数据相关和结构相关

Q5

我认为应该选择simticks或commitedInst作为指标来对性能表现进行评价

因为执行时间和指令数都可以直接反应这个大部分计算&循环跳转的函数的优化程度；而CPI不能明显反映是因为O3优化和循环展开优化对执行指令数和运行时间都进行了大幅优化，CPI不能更好的反映这两方面的同时优化，从O1表和O3表也可以看出，有些函数的CPI不升反降，这并不是因为得到了负优化，而是因为simticks和inst数都下降了，导致CPI发生了变化

Q6

我认为手动循环展开优化有一定意义，因为对特定程序（`daxsbxpxy`）进行手动循环展开还是可以获得显著的性能提升的（从CPI、运行时间、执行指令数均能证明该结论）