计算机体系结构 lab2

PB20111704 张宇昂

实验步骤

Python 脚本

本次实验中我选择在实验1中创建的gem5配置脚本，Lab1中的Cache我没有做任何修改，在此不过多赘述

下面是我在本次实验中修改的`two_level.py`配置文件代码：

```
# import the m5 (gem5) library created when gem5 is built
import m5
# import all of the SimObjects
from m5.objects import *
# from gem5.runtime import get_runtime_isa
import argparse

# Add the common scripts to our path

m5.util.addToPath("../../")

# import the caches which we made
from caches import *
# import the SimpleOpts module

from common import SimpleOpts
# Default to running 'hello', use the compiled ISA to find the binary
# grab the specific path to the binary
thispath = os.path.dirname(os.path.realpath(__file__))
default_binary = os.path.join(
    thispath,
    "../../../../",
    "lab2/cs251a-microbench-master/merge",
)
SimpleOpts.add_option("--binary", nargs="?", default=default_binary)
SimpleOpts.add_option("--cpu_type", nargs="?", default="DerivO3CPU")
SimpleOpts.add_option("--frequency", nargs="?", default="1GHz")
SimpleOpts.add_option("--mem_type", nargs="?", default="DDR3_1600_8x8")
SimpleOpts.add_option("--disable_l2", nargs="?", default=None)
SimpleOpts.add_option("--issue_width", nargs="?", default=8)
args = SimpleOpts.parse_args()
print(args)

# Binary to execute

# SimpleOpts.add_option("binary", nargs="?", default=default_binary)

# Finalize the arguments and grab the args so we can pass it on to our objects
# args = SimpleOpts.parse_args()
```

```
# create the system we are going to simulate
system = System()
# Set the clock frequency of the system (and all of its children)
system.clk_domain = SrcClockDomain()
system.clk_domain.clock = "1GHz"
if args.frequency:
    print("Setting frequency")
    system.clk_domain.clock = args.frequency
system.clk_domain.voltage_domain = VoltageDomain()
# Set up the system
system.mem_mode = "timing"  # Use timing accesses
system.mem_ranges = [AddrRange("512MB")]  # Create an address range
# Create a simple CPU
system.cpu = DerivO3CPU()
if args.cpu_type == "MinorCPU":
    print("Set CPU")
    system.cpu = MinorCPU()
elif args.cpu_type == "DerivO3CPU":
    print("Set CPU")
    system.cpu = DerivO3CPU()
    if args.issue_width:
        system.cpu.issueWidth = args.issue_width

# Create an L1 instruction and data cache
system.cpu.icache = L1ICache(args)
system.cpu.dcache = L1DCache(args)
# Connect the instruction and data caches to the CPU
system.cpu.icache.connectCPU(system.cpu)
system.cpu.dcache.connectCPU(system.cpu)
system.membus = SystemXBar()
if args.disable_l2 == True:
    print("disable l2")
    system.cpu.icache.connectBus(system.membus)
    system.cpu.dcache.connectBus(system.membus)
else:
    # Create L2 bus, because L2 only expects a single port to connect to it
    system.l2bus = L2XBar()
    system.cpu.icache.connectBus(system.l2bus)
    system.cpu.dcache.connectBus(system.l2bus)
    system.l2cache = L2Cache(args)      # Create L2 cache
    system.l2cache.connectCPUSideBus(system.l2bus)
    system.l2cache.connectMemSideBus(system.membus)

'''
# Create a memory bus, a coherent crossbar, in this case
system.l2bus = L2XBar()
# Hook the CPU ports up to the l2bus
system.cpu.icache.connectBus(system.l2bus)
system.cpu.dcache.connectBus(system.l2bus)
# Create an L2 cache and connect it to the l2bus
system.l2cache = L2Cache(args)
system.l2cache.connectCPUSideBus(system.l2bus)
# Create a memory bus
system.membus = SystemXBar()
# Connect the L2 cache to the membus
```

```
system.l2cache.connectMemSideBus(system.membus)
'''
print("before controller")
# create the interrupt controller for the CPU
system.cpu.createInterruptController()
system.cpu.interrupts[0].pio = system.membus.mem_side_ports
system.cpu.interrupts[0].int_requestor = system.membus.cpu_side_ports
system.cpu.interrupts[0].int_responder = system.membus.mem_side_ports
# Connect the system up to the membus
system.system_port = system.membus.cpu_side_ports
# Create a DDR3 memory controller
print("before memory")
system.mem_ctrl = MemCtrl()
print("Set memory")
system.mem_ctrl.dram = DDR3_1600_8x8()
system.mem_ctrl.dram.range = system.mem_ranges[0]
system.mem_ctrl.port = system.membus.mem_side_ports
# system.workload = SEWorkload.init_compatible(args.binary)
# Create a process for a simple "Hello World" application
print("before process")
process = Process()
# Set the command
# cmd is a list which begins with the executable (like argv)
final_path = os.path.join(
            thispath,
            "../../../../",
            args.binary,
        )
if args.binary == default_binary:
    system.workload = SEWorkload.init_compatible(args.binary)
    process.cmd = [args.binary]
else:
    final_path = os.path.join(
            thispath,
            "../../../../",
            args.binary,
        )
    system.workload = SEWorkload.init_compatible(final_path)
    process.cmd = final_path

# Set the cpu to use the process as its workload and create thread contexts
system.cpu.workload = process
system.cpu.createThreads()
# set up the root SimObject and start the simulation
root = Root(full_system=False, system=system)
# instantiate all of the objects we've created above
print("before simulation")
m5.instantiate()
print("Beginning simulation!")
exit_event = m5.simulate()
print("Exiting @ tick %i because %s" % (m5.curTick(), exit_event.getCause()))
```

先通过`SimpleOpts.add_option`添加参数，分别是

- `binary`：benchmark的路径

- `cpu_type`：CPU的类型（Micro/O3）
- `frequency`：`CPU_clock`的频率
- `issue_width`：设置CPU的issueWidth
- `disable_l2`：设置是否禁用L2 Cache
- `mem_type`：内存类型，本次实验中应该都是`DDR3_1600_8x8`

当然包括实验1中`L1 ICache`，`L1 DCache`，`L2 Cache`的大小参数

这部分的代码如下：

```
thispath = os.path.dirname(os.path.realpath(__file__))
default_binary = os.path.join(
    thispath,
    "../../../../",
    "lab2/cs251a-microbench-master/merge",
)
SimpleOpts.add_option("--binary", nargs="?", default=default_binary)
SimpleOpts.add_option("--cpu_type", nargs="?", default="DerivO3CPU")
SimpleOpts.add_option("--frequency", nargs="?", default="1GHz")
SimpleOpts.add_option("--mem_type", nargs="?", default="DDR3_1600_8x8")
SimpleOpts.add_option("--disable_l2", nargs="?", default=None)
SimpleOpts.add_option("--issue_width", nargs="?", default=8)
args = SimpleOpts.parse_args()
```

与Lab 1中的系统搭建方式类似，在此我们不多赘述同样的构造方式，我们只阐述如何通过参数来设置需要动态修改的参数设置。

频率设置的方法如下：设置默认频率`1GHz`，若参数存在，则进行修改设置

```
system.clk_domain = SrcClockDomain()
system.clk_domain.clock = "1GHz"
if args.frequency:
    print("Setting frequency")
    system.clk_domain.clock = args.frequency
```

CPU类型的设置方法如下：方法类似，由于本次实验只有两种CPU，所以只用两个type设置判断即可：

```
system.cpu = DerivO3CPU()
if args.cpu_type == "MinorCPU":
    print("Set CPU")
    system.cpu = MinorCPU()
elif args.cpu_type == "DerivO3CPU":
    print("Set CPU")
    system.cpu = DerivO3CPU()
    if args.issue_width:
        system.cpu.issueWidth = args.issue_width
```

由于只有O3CPU可以设置issueWidth，所以我们只需要在CPU类型为O3CPU时检查是否有issueWidth参数，若参数存在则进行设置

在Lab 1中介绍了不存在L2 Cache时直接将L1 Cache连接到总线的方法；以及存在L2 Cache时将L1 Cache连接到l2bus上，再将L2 Cache连接到总线上

设置L1 Cache和L2 Cache的方法如下：

```
# Create an L1 instruction and data cache
system.cpu.icache = L1ICache(args)
system.cpu.dcache = L1DCache(args)
# Connect the instruction and data caches to the CPU
system.cpu.icache.connectCPU(system.cpu)
system.cpu.dcache.connectCPU(system.cpu)
system.membus = SystemXBar()
if args.disable_l2 == True:
    print("disable l2")
    system.cpu.icache.connectBus(system.membus)
    system.cpu.dcache.connectBus(system.membus)
else:
    # Create L2 bus, because L2 only expects a single port to connect to it
    system.l2bus = L2XBar()
    system.cpu.icache.connectBus(system.l2bus)
    system.cpu.dcache.connectBus(system.l2bus)
    system.l2cache = L2Cache(args)      # Create L2 cache
    system.l2cache.connectCPUSideBus(system.l2bus)
    system.l2cache.connectMemSideBus(system.membus)
```

设置binary文件的路径的方式：如果是设置的默认路径`default_path`，直接设置即可；否则需要设置为`args.binary`中的路径

```
if args.binary == default_binary:
    system.workload = SEWorkload.init_compatible(args.binary)
    process.cmd = [args.binary]
else:
    print(args.binary)
    system.workload = SEWorkload.init_compatible(args.binary)
    process.cmd = final_path
```

经助教讲解，Compiler的设置需要在benchmark所在文件夹下的Makefile文件的编译器参数设置，这样我们就完成了所有参数的设置

输出的文件请参见`result`文件夹

运行脚本

我的运行脚本`run.sh`如下：

```
#!/bin/bash

GEM5_BASE=~/ca2023-labs/gem5-stable/gem5-stable
SRC_DIR=~/ca2023-labs/gem5-stable/lab2/cs251a-microbench-master
RESULT_DIR=~/ca2023-labs/gem5-stable/lab2/result
TARGET=~/ca2023-labs/gem5-stable/gem5-stable/configs/tutorial/part1/two_level.py
# TARGET=/root/gem5/configs/example/se.py

NUM=1

# Clean result dir
rm -rf ${RESULT_DIR}/*
```

```
for FILE in mm lfsr merge sieve spmv; do
    mkdir -p ${RESULT_DIR}/${FILE}
done

for FILE in mm lfsr merge sieve spmv; do
# for FILE in mm; do

    cd $SRC_DIR && make clean --silent && make gen_arr --silent && make all --
silent && cd $GEM5_BASE

    echo [${NUM}/35]: Task 1 [config_1] -\> $FILE
    ${GEM5_BASE}/build/X86/gem5.opt ${TARGET} --binary=${SRC_DIR}/${FILE} --
cpu_type='DerivO3CPU' \
        --l1d_size='64kB' --l1i_size='64kB' --l2_size='2MB' --issue_width=8\
        --frequency='1GHz' --disable_l2=True --mem_type='DDR3_1600_8x8'
    cp -r m5out/ ${RESULT_DIR}/${FILE}/config_1/
    ((NUM++))

    echo [${NUM}/35]: Task 2 [config_2] -\> $FILE
    ${GEM5_BASE}/build/X86/gem5.opt ${TARGET} --binary=${SRC_DIR}/${FILE} --
cpu_type='MinorCPU' \
        --l1d_size='64kB' --l1i_size='64kB' --l2_size='2MB' --issue_width=8\
        --frequency='1GHz' --disable_l2=True --mem_type='DDR3_1600_8x8'
    cp -r m5out/ ${RESULT_DIR}/${FILE}/config_2/
    ((NUM++))

    echo [${NUM}/35]: Task 3 [config 3] -\> $FILE
    ${GEM5_BASE}/build/X86/gem5.opt ${TARGET} --binary=${SRC_DIR}/${FILE} --
cpu_type='DerivO3CPU' \
        --l1d_size='64kB' --l1i_size='64kB' --l2_size='2MB' --issue_width=2\
        --frequency='1GHz' --disable_l2=True --mem_type='DDR3_1600_8x8'
    cp -r m5out/ ${RESULT_DIR}/${FILE}/config_3/
    ((NUM++))

    echo [${NUM}/35]: Task 4 [config 4] -\> $FILE
    ${GEM5_BASE}/build/X86/gem5.opt ${TARGET} --binary=${SRC_DIR}/${FILE} --
cpu_type='DerivO3CPU' \
        --l1d_size='64kB' --l1i_size='64kB' --l2_size='2MB' --issue_width=8\
        --frequency='4GHz' --disable_l2=True --mem_type='DDR3_1600_8x8'
    cp -r m5out/ ${RESULT_DIR}/${FILE}/config_4/
    ((NUM++))

    echo [${NUM}/35]: Task 5 [config 5] -\> $FILE
    ${GEM5_BASE}/build/X86/gem5.opt ${TARGET} --binary=${SRC_DIR}/${FILE} --
cpu_type='DerivO3CPU' \
        --l1d_size='64kB' --l1i_size='64kB' --l2_size='256kB' --issue_width=8\
        --frequency='1GHz' --mem_type='DDR3_1600_8x8'
    cp -r m5out/ ${RESULT_DIR}/${FILE}/config_5/
    ((NUM++))

    echo [${NUM}/35]: Task 6 [config 6] -\> $FILE
    ${GEM5_BASE}/build/X86/gem5.opt ${TARGET} --binary=${SRC_DIR}/${FILE} --
cpu_type='DerivO3CPU' \
        --l1d_size='64kB' --l1i_size='64kB' --l2_size='2MB' --issue_width=8\
        --frequency='1GHz' --mem_type='DDR3_1600_8x8'
```

```
    cp -r m5out/ ${RESULT_DIR}/${FILE}/config_6/
    ((NUM++))
    echo [${NUM}/35]: Task 7 [config 7] -\> $FILE
    ${GEM5_BASE}/build/X86/gem5.opt ${TARGET} --binary=${SRC_DIR}/${FILE} --
cpu_type='DerivO3CPU' \
        --l1d_size='64kB' --l1i_size='64kB' --l2_size='16MB' --issue_width=8\
        --frequency='1GHz' --mem_type='DDR3_1600_8x8'
    cp -r m5out/ ${RESULT_DIR}/${FILE}/config_7
    ((NUM++))

done
```

Config 1 ~ Config 7配置了每一种参数进行模拟，并复制m5out文件夹的输出文件输出到result文件夹中

问题解答

Question 1

1. 对不同`CPU_type`可采用的指标有：
    1. 分支跳转预测成功率：分支跳转预测成功更多可以显著提高系统性能，避免失败时带来的多余开销
    2. `IPC`或者总执行时间
2. 对不同`CPU_clock`可采用的指标为：
    1. `IPC`或者总执行时间
3. 对不同`issue_width`可采用的指标有：
    1. `IPC`或总执行时间
4. 对不同大小或是否存在的`L2 cache`可采用的指标有：
    1. `IPC`或总执行时间
    2. 内存访问次数

Question 2

通过对比`spmv` benchmark的tick count，我认为该基准显示出删除一个较小的`L2 Cache`是受益的：

1. no cache：17548632000
2. 256KB Cache：25266342000
3. 2MB Cache：17548632000
4. 16MB Cache：12687747000

Question 3

1. memory regularity指的是类似于空间规律性的概念，我的理解是同一块内存被访问的规律性，也就是我们课内讲到的时间局部性
2. memory locality对应的代表空间局部性的概念
3. control regularity指的是类似于控制指令流转移的规律性，我的理解是这个规律性代表着程序的跳转与我们系统的分配是否“契合”，即与系统安排的跳转错误的次数成反比

Question 4

1. 对于memory regularity，我们可以用`system.cpu.dcache.overall_hits::total`和`system.cpu.dcache.overall_misses::total`得到dCache的命中率，更高的缓存命中率意味着更好的memory regularity
2. 对于Control Regularity，我们可以用`system.cpu.branchPred.condPredicted`和`system.cpu.branchPred.condIncorrect`得到分支预测的成功率，成功率与`control regularity`成正比
3. 对于memory locality，我们可以用与memory regularity相同的指标，因为cache hit rate同样反映了空间局部性

Question 5

1. `mm`：该benchmark具有较高的memory regularity，较高的Control regularity，较高的memory locality，这是因为这个程序的内存是顺序访问的，而分支跳转是和给定的数据无关的
2. `lfsr`：该benchmark具有较低的memory regularity，较高的Control regularity，较低的memory locality，这是因为这个程序访问数组是随机访问的，但是它的循环是与数据无关的
3. `merge`：该benchmark具有较高的memory regularity，较低的Control regularity，较高的memory locality，这是因为这个程序看起来像一个归并排序，它是顺序访问内存的，但是它的跳转时依赖数据的，规律性较小
4. `sieve`：该benchmark具有较高的memory regularity，较低的Control regularity，较低的memory locality，这是因为这个程序打印小于1000000的质数，它存在数据相关的跳转，而且它的内存访问在判断质数时，时间的规律性较强，空间局部性较弱（以质数顺序访问）
5. `spmv`：该benchmark具有较低的memory regularity，较低的Control regularity，较高的memory locality，这是因为该程序的内层执行次数最多的循环是数据相关的，它的数组访问在一次循环中是序列化的，所以空间局部性较高，时间局部性较低

Question 6

对`sieve` benchmark：

1. 应用程序增强：可以通过循环展开来减少数据依赖：

   ```
   for(int p = 2; p < n; ++p) {
       total+=!notprime[p];
   }
   ```

   将上述代码改为：

   ```
   for(int p = 2; p < n; p += 2) {
       total+=(!notprime[p] + !notprime[p+1];
   }
   ```

2. ISA 增强：支持更长的SIMD操作，优化无数据依赖的循环
3. 微体系结构增强：该benchmark可通过降低L2 Cache缓存的延迟来提高性能，因为这个benchmark的数组大小为(1000000 * sizeof(char) = 976kB)，基本上可以存储在L2 Cache中，所以降低L2 Cache的缓存即可提高性能

实验中的困难与收获

1. 参数设置，本次实验中的参数设置报错信息不太明确，有时设置错误会报一些难以理解的错误，需要看很久才会发现是参数的问题
2. 文档较少，对Gem 5还是知之甚少，而且我对英文文档的理解偶尔会有误，最后是参照CSDN上Gem 5的翻译进行查找到（建议老师能否在课程上添加一些Gem 5讲解的内容？因为相较于Verilog我们对Gem 5毫无了解，在学习组成原理时也有对Verilog的讲解）
3. 一个错误：Cache的设置参数应为`kB`而不是`KB`，这个错误看了一晚上...
4. 脚本编写：本次实验需要对5个benchmark依次按照7个不同的配置进行模拟，逐个运行会很麻烦，于是我学习了一下shell脚本的编写