《计算机体系结构》

实验报告

院　　系：　计算机与信息学院

专　　业：　计算机科学与技术

年　　级：　18-3 班

课程名称：　计算机体系结构

学　　号：　2018211991

姓　　名：　余梓俊

指导教师：　李建华

2020 年 11 月 12 日

实验一、流水线相关与指令调度

1. 实验内容

一、 流水线相关

1. 用 MIPS64 汇编语言编写代码文件*.s，程序中应包括结构相关。用 WinMIPS64 模拟器运行你编写的程序，通过模拟：
   - 找出存在结构相关的指令对以及相应的结构相关的部件；
   - 记录由结构相关引起的暂停时钟周期数，计算暂停时钟周期数占总执行周期数的百分比；
   - 论述结构相关对 CPU 性能的影响，讨论解决结构相关的方法。
2. 用 MIPS64 汇编语言编写代码文件*.s，程序中应包括数据相关。在不采用定向技术的情况下，用 WinMIPS64/WinDLX 模拟器运行存在数据相关的程序。记录数据相关引起的暂停时钟周期数以及程序执行的总时钟周期数，计算暂停时钟周期数占总执行周期数的百分比。
3. 在采用定向技术的情况下，用 WinMIPS64 模拟器再次运行程序。重复上述 3 中的工作，并计算采用定向技术后性能提高的倍数。

二、 指令调度

1. 用指令调度技术解决流水线中的结构相关与数据相关

   （1） 用 MIPS64 汇编语言编写代码文件*.s，程序中应包括数据相关与结构相关（你可以自己设置各个功能单元的延迟时间）

   （2） 用 WinMIPS64 模拟器运行你所写的程序。记录程序执行过程中各种相关发生的次数、发生相关的指令组合，以及程序执行的总时钟周期数；

   （3） 采用指令调度技术对程序进行指令调度，消除相关（**手工调度^_^**）；

   （4） 用 WinMIPS64 模拟器运行调度后的程序，观察程序在流水线中的执行情况，记录程序执行的总时钟周期数；

   （5） 根据记录结果，比较调度前和调度后的性能。论述指令调度对于提高 CPU 性能的意义。
2. 用循环展开、寄存器换名以及指令调度提高性能

   （1） 用 MIPS64 汇编语言编写代码文件*.s，程序中包含一个循环次数为 4 的整数倍的简单循环；

   （2） 用 WinMIPS64 模拟器运行该程序。记录执行过程中各种相关发生的次数以及程序执行的时钟周期数；

   （3） 将循环展开 3 次，将 4 个循环体组成的代码代替原来的循环体，并对程序做相应的修改。然后,对新的循环体进行寄存器换名和指令调度；

   （4） 用 WinMIPS64 模拟器运行修改后的程序，记录执行过程中各种相关发生的次数以及程序执行的总时钟周期数；

   （5） 根据记录结果，比较循环展开、指令调度前后的性能。

2. 实验过程

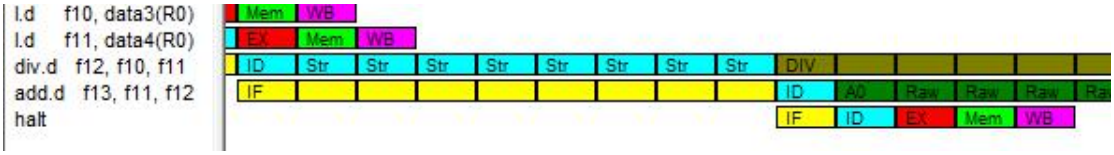
一、流水线相关

Structural and RAW:（开启 forwarding）

用 10/3 和 10.0/3.0 两个除法来构造结构冲突，加法指令紧接在除法指令后，并且用到除法的结果，构造出 RAW。第一次的两个 load word 后面也会出现一个 RAW，因为除法进入 EX 时上一个 load word 进入 MEM，尚未进入 Write Back，因此有一个 RAW。

其中 Structural 会 stall pipeline，RAW 不会。

```
.text

    ; structural and raw

    ; r12 = 10 / 3 = 3
    lw      r10, data1(R0)
    lw      r11, data2(R0)
    ; 1 raw here, wait for load word
    ; pipeline won't stalled by raw
    ddiv    r12, r10, r11

    ; f12 = 10.0 / 3.0 = 3.3
    l.d     f10, data3(R0)
    l.d     f11, data4(R0)
    ; structurals here, wait for ddiv above
    div.d   f12, f10, f11

    ; raws here, wait for ddiv to be done
    add.d   f13, f11, f12

    halt

.data
    data1:  .word   10
    data2:  .word   3
    data3:  .double 10.0
    data4:  .double 3.0
```



WAR and WAW:（开启 forwarding）

课本上描述的经典的 5 段 risc 流水线，没有 war 和 waw，是因为经典 5 段 risc 流水线的 ex 阶段是固定时长的，周期数和其他几个阶段一样。而在 winMIPS 仿真里面，是会出现 war 和 waw 的。

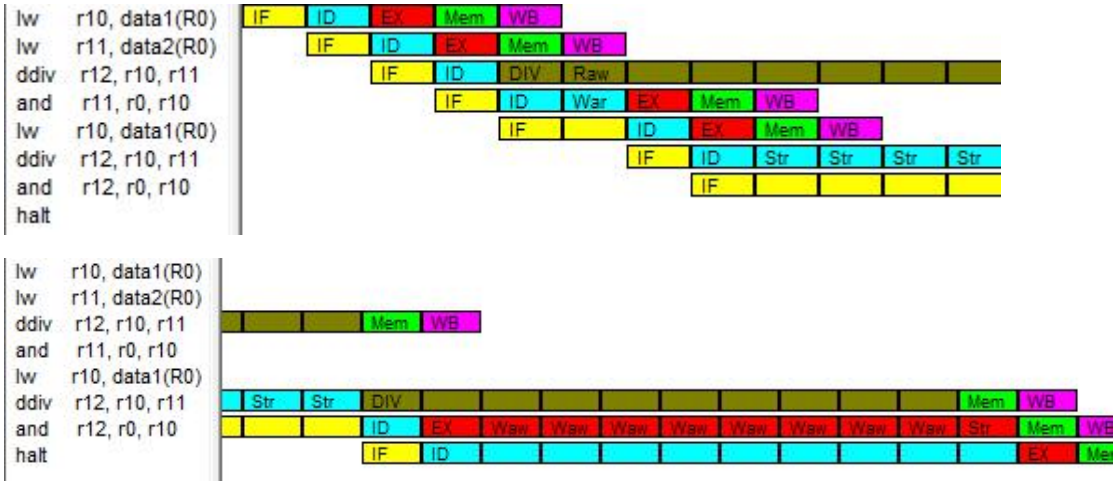
在除法后面使用一条 EX 为一周期的 and 指令，结果寄存器为除法的源寄存器，以此构造 WAR。注意，如果在 lw 后面加一条 nop，WAR 会消失，这和为什么只有一个 WAR 而不是多个 WAR（直到 ddiv 结束）是同一个道理。因为一旦进入 EX 阶段，源寄存器的内容会被保存下来，这时后面的指令修改源寄存不会导致冲突。因此此处只有一个 WAR，而且如果加了一条 nop，ddiv 不需要等待 load word 导致的一个 RAW，它会先进入 EX 阶段，之后 and 指令再进入 EX 阶段，也不会有 WAR 冲突了。（本段话基本和代码中注释对应）

同样地，在除法后面使用一条 EX 为一周期的 and 指令，结果寄存器与除法的结果寄存器相同，以次构造 WAW。此处在 WAW 结束后，还会有一个 structural，因为 and 和 ddiv 同时进入了 MEM。（本段话基本和代码中注释对应）

WAW 和 WAR 均会 stall pipeline。

```
.text

    ; war and waw

    ; war
    lw      r10, data1(R0)
    lw      r11, data2(R0)
    ; nop
    ; 1 raw
    ddiv    r12, r10, r11
    ; 1 war, pipeline stalled
    ; because r11 is used in ddiv above
    ; and ddiv hasn't gone into EX stage (see comments below)
    ; since ddiv has 1 raw
    and     r11, r0, r10

    ; if there's a nop, there won't be war.
    ; it's the same reason as why there is only 1 war instead of war
s.
    ; because once the source reg is in EX stage,
    ; it's stored somewhere and you can change the source reg.
    ; it won't collide.

    ; waw
    lw      r10, data1(R0)
    ; structurals
    ddiv    r12, r10, r11
    ; waws, pipeline stalled
```

```
    ; because it's writing to r12
    ; which is going to be written by ddiv first.
    ; also 1 structural
    ; because after ddiv is done with EX stage
    ; and and ddiv are both going to MEM stage
    and     r12, r0, r10

    halt

.data
    data1:   .word   10
    data2:   .word   3
```



统计：（开启 forwarding）

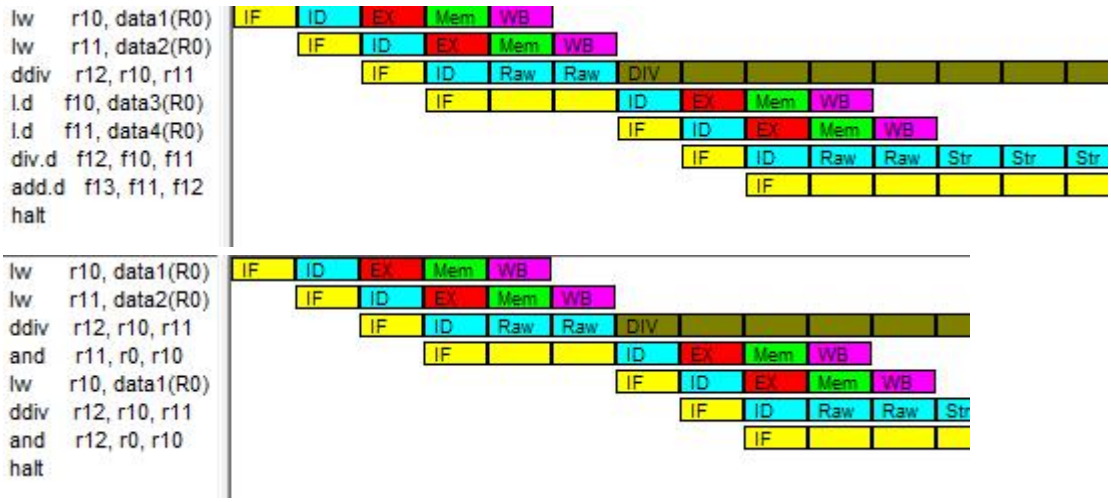
structural_and_raw.s 共 8 条指令，30 个周期，其中有 10 个 RAW，8 个 structural。结构冲突暂停 26.7%。waw_and_war.s 共 8 条指令，29 个周期，其中有 1 个 RAW，8 个 structural，1 个 WAR，8 个 WAW。结构冲突暂停 27.6%，数据冲突暂停 31.0%。

关闭 forwarding 后：

任何冲突（包括 RAW）都将 stall pipeline。

对于 RAW，必须等到 WB 结束后，RAW 才会消失（例如下面第一张图 structural_and_raw.s 中 ddiv 指令原本只需要一个 RAW，因为 forwarding 下 WB 前半周期写目的寄存器，写完后后半周期要用到该值的就可以直接读出使用，关闭 forwarding 之后就不行）。

而下面第二张图 waw_and_war.s 中原本的 WAR 会消失，一是因为 RAW 会 stall pipeline，二是由于 pipeline 被 stall，ddiv 和 and 指令进入 EX 的顺序是和指令顺序一致的，如前文所述，这就不会导致 WAR。



统计：（关闭 forwarding）

structural_and_raw.s 共 8 条指令，33 个周期（增加了 3 个），其中有 15 个 RAW（增加了 5 个），5 个 structural（减少了 3 个）。结构冲突暂停 15.1%，数据冲突暂停 45.4%。waw_and_war.s 共 8 条指令，30 个周期（增加了 1 个），其中有 4 个 RAW（增加了 3 个），6 个 structural（减少了 2 个），0 个 WAR（减少了 1 个），8 个 WAW（不变）。结构冲突暂停 20%，数据冲突暂停 33.3%。

structural_and_raw.s 和 waw_and_war.s 通过 forwarding 提高的倍数分别为 10%，3.4%。

结构冲突必然会导致流水线效率的降低，通过对部件进行流水或重复设置资源可以减少结构冲突。但过多的重复资源带来的浪费也可能比不上相应的减少的结构冲突带来的提升，因为结构冲突是不可能完全避免。

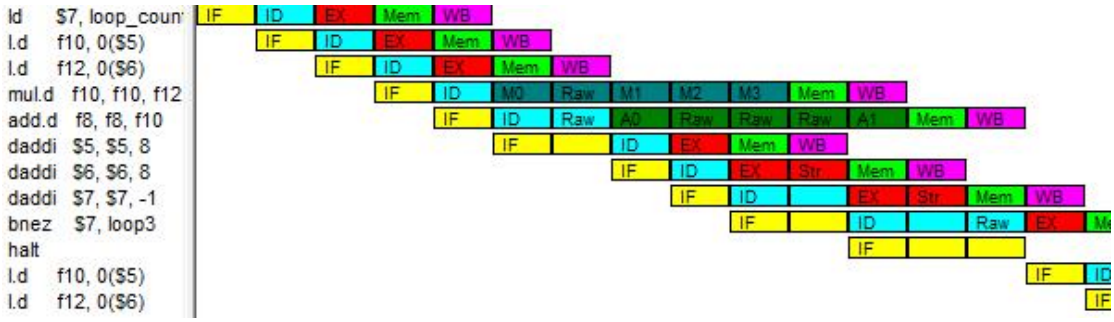
二、 指令调度

1. 指令调度：

程序说明：手动将一段 C 代码翻译为汇编代码，该代码的目的是求两个长度为 100 的向量的内积（简单起见，汇编代码中向量长度设为 4）。所用到的寄存器的功能以及其他细节可见代码注释。

```
.text
    ; This idea is from code examples in UMN.
    ; https://www.d.umn.edu/~gshute/arch/loop-unrolling.xhtml

    ; double dotProduct = 0;
    ; for (int i = 0; i < 100; i++) {
    ;   dotProduct += A[i]*B[i];
    ; }

    ; Initialize loop count ($7) to 100.
```

```
    ; Initialize dotProduct ($f10) to 0.
    ; Initialize A[i] pointer ($5) to the base address of A.
    ; Initialize B[i] pointer ($6) to the base address of B.
    ld      $7, loop_count(r0)
    loop3:
            ; before scheduling
            l.d     f10, 0($5)        ; $f10 ← A[i]
            l.d     f12, 0($6)        ; $f12 ← B[i]
            mul.d   f10, f10, f12     ; $f10 ← A[i]*B[i]
            add.d   f8, f8, f10       ; $f8 ← $f8 + A[i]*B[i]
            daddi   $5, $5, 8         ; increment pointer for A[i]
            daddi   $6, $6, 8         ; increment pointer for B[i]
            daddi   $7, $7, -1        ; decrement loop count
    test:
            bnez    $7, loop3         ; Continue if loop count > 0
    halt

.data

    loop_count:     .word   4
```



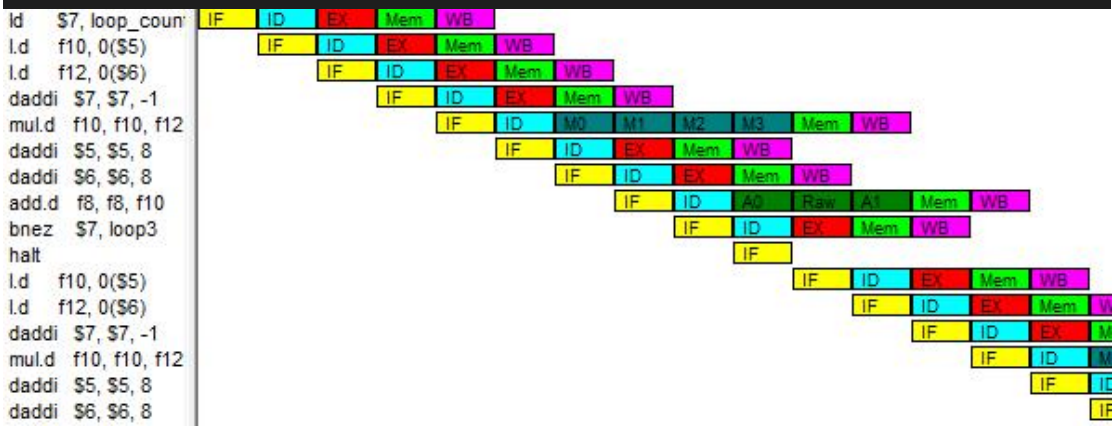

```
Execution
57 Cycles
34 Instructions
1.676 Cycles Per Instruction (CPI)
```

```
Stalls
24 RAW Stalls
0 WAW Stalls
0 WAR Stalls
8 Structural Stalls
3 Branch Taken Stalls
0 Branch Misprediction Stalls
```

显然，add.d 与 mul.d 存在数据相关，而三个 daddi 是独立的，可以将它们调到 add.d 与 mul.d 中间。

第二个 l.d 与 mul.d 之间也存在数据相关，可以将最后一个 daddi 移到这两条指令中间。同时，为了让调度结果更明显，效果更好，将模拟器的 Architecture 设为浮点加法 2 个周期，浮点乘法 4 个周期。4 个周期的浮点乘法，正好可以放下两个 daddi（实际上设为 3 个周期也可以，但设为 4 个还可以消除掉第二个 daddi 的 MEM 和乘法的 MEM 的结构冲突，调度效果更加明显）。

```
            ; after scheduling
            l.d     f10, 0($5)          ; $f10 ← A[i]
            l.d     f12, 0($6)          ; $f12 ← B[i]
            daddi   $7, $7, -1          ; decrement loop count
            mul.d   f10, f10, f12       ; $f10 ← A[i]*B[i]
            daddi   $5, $5, 8           ; increment pointer for A[i]
            daddi   $6, $6, 8           ; increment pointer for B[i]
            add.d   f8, f8, f10         ; $f8 ← $f8 + A[i]*B[i]
```




```
Execution
42 Cycles
34 Instructions
1.235 Cycles Per Instruction (CPI)
```

```
Stalls
4 RAW Stalls
0 WAW Stalls
0 WAR Stalls
1 Structural Stall
3 Branch Taken Stalls
0 Branch Misprediction Stalls
```

可以看到，调度之后数据相关从 24 个减少到了 4 个，结构相关也从 8 个减少至 1 个（仅最后一条 halt 的 MEM 发生了结构相关，可以说完全消除了 loop 的结构相关）。

调度前总 34 条指令的周期数为 57，调度后为 42，加速比达到了 1.36。

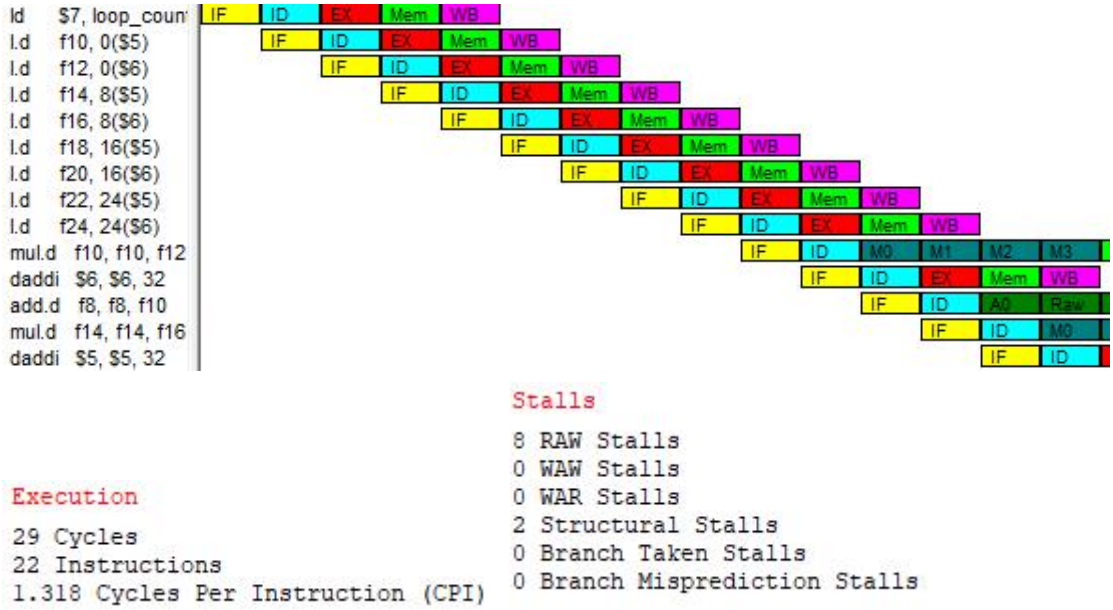
当然，并不是所有的调度都能达到这么好的效果，调度也不是那么轻松的。事实上直到在 UMN 的网站上看到点乘代码之前，我自己所写的代码所进行调度得到的效果一直非常一般。包括这段向量点积的代码，也是反复修改了模拟器的 Architecture 之后才看到这么好的效果。

2.循环展开：

同样地，修改标号 loop3 与 test 之间的循环体代码。

```
            ; loop unrolling
            l.d     f10, 0($5)          ; iteration with displacement
0
            l.d     f12, 0($6)
            l.d     f14, 8($5)          ; iteration with displacement
8
            l.d     f16, 8($6)
```

```
            l.d     f18, 16($5)         ; iteration with displacement
16
            l.d     f20, 16($6)
            l.d     f22, 24($5)         ; iteration with displacement
24
            l.d     f24, 24($6)
            mul.d   f10, f10, f12
            daddi   $6, $6, 32
            add.d   f8, f8, f10
            mul.d   f14, f14, f16
            daddi   $5, $5, 32
            add.d   f8, f8, f14
            mul.d   f18, f18, f20
            daddi   $7, $7, -4          ; decrement loop count
            add.d   f8, f8, f18
            mul.d   f22, f22, f24
            add.d   f8, f8, f22
```



Execution

29 Cycles
22 Instructions
1.318 Cycles Per Instruction (CPI)

Stalls

8 RAW Stalls
0 WAW Stalls
0 WAR Stalls
2 Structural Stalls
0 Branch Taken Stalls
0 Branch Misprediction Stalls

循环展开+寄存器换名+手工指令调度后，程序的总执行周期数减小到了 29。和原本的 57、单纯指令调度后的 42 相比，加速比分别达到了 1.97 和 1.45，效果显著。

3. 实验心得

本次实验大大加深了我对各种 hazard 以及 forwarding 的作用的理解，许多原理和实验中学到的知识点已在上文“实验过程”中阐述。实验中最费时费心的，并不是理解实验内容或对代码进行分析，而是如何设计出包含相关，并且能用指定优化方法进行优化的程序。尤其是指令调度的代码，起初我企图在课堂 PPT 中循环展开那段示例代码的基础上加以修改，使其能显著得体现指令调度的效果，但一直无法写出有能满足我要求的、有实际意义的代码。即使是最终采用的这段向量点积的代码，也花费了我大量时间去找到效果较好的指令调度的

位置。通过这次实验，我更加敬佩那些发明出各种自动化的软硬件优化算法的计算机科学家们，如此简单短小的代码才能使我能够随意地进行寄存器换名和各种调度，不难想象，编译器和 CPU 要对复杂的实际生产代码进行调度优化，有多么困难。

实验二、分支预测

1. 实验内容

本次实验使用分支预测模拟器 sim-bpred，在 4 种预测器类型及不同的参数配置下运行测试程序，并比较、分析结果，使大家加深对动态分支预测机制的理解，并了解各种分支预测实现方式的优劣。

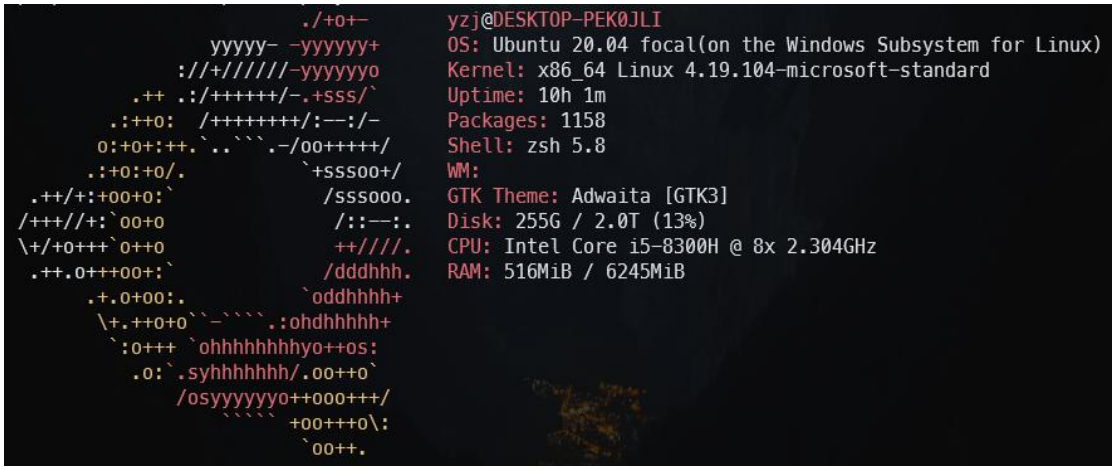
(1)进入 SimpleScalar 目录(simplesim-3.0)。

(2)用 sim-bpred 仿真器运行 spec95_little 中的测试程序：ccl, compress95, go, perl，或者从 SPEC2000 INT 中任选 4 个程序，分别采用 4 种不同的分支预测方法，即 bimod 方式，two-level adaptive 方式，always taken 方式，always not taken 方式，并对前两种分别使用下表中两种参数配置:分析仿真器输出的关于分支预测的统计参数集,填写表格，并对各仿真器的能力给出相应说明。

2. 实验过程

1. 安装 SimpleScalar

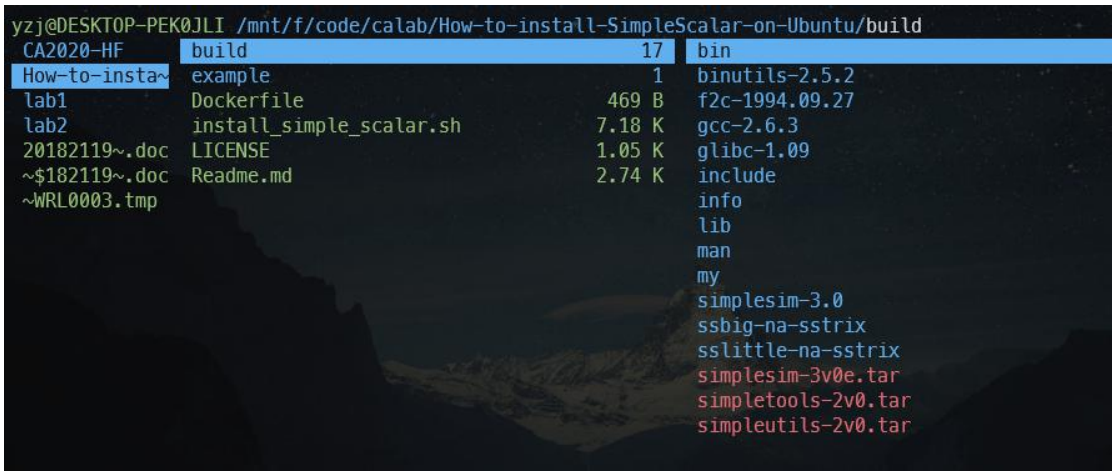
实验环境：WSL Ubuntu 20.04



安装过程：

参考 https://github.com/sdenel/How-to-install-SimpleScalar-on-Ubuntu

编译后目录结构



在实验文档中提供的 benchmark 的 SPEC2000 上运行 simple-bpred，报错，提示指令集架构错误；再尝试运行 SPEC95，提示可执行文件的大小端不正确。

在向指导老师寻求帮助并阅读了 repo 中的 install_simple_scalar.sh 以及 simplesim-3.0 目录下的 README 之后，修改 make 参数，将原本的 config-pisa 修改为 config-alpha 重新编译。

```
fatal: PISA simulator cannot run Alpha binary `./benchmark/CINT2000/181.mcf/mcf00.peak.ev6'
```

```
fatal: PISA binary `./benchmark/spec95_little/go.pisa-big' has wrong endian format
```

```
TO INSTALL:

The following details how to build and install the SimpleScalar simulators:

    a) vi Makefile

    Make sure all compile options are set for your host, we've listed
    the options for the OS/compiler combinations that were tested,
    uncomment one of these if appropriate.  You'll likely not have to
    change anything for the supported hosts, and if you need to change
    anything, it will likely be the CC variable (which specifies
    the ANSI C compiler to use to build the simulators).  NOTE: the
    simulators must be built with an ANSI-C compatible compiler, if you
    have problems with your compiler, try using GNU GCC as it is known
    to build the simulators on all the supported platforms.

    b) make config-pisa         (to build SimpleScalar/PISA)
       or
       make config-alpha            (to build SimpleScalar/Alpha)

    Execute one of the above commands to configure the SimpleScalar
    build.  The SimpleScalar/PISA build executes SimpleScalar PISA
    (Portable ISA) binaries (the old "SimpleScalar ISA"), and the
    SimpleScalar/Alpha build executes statically linked Alpha OSF
    binaries.  If building with Cygwin/32 (www.cygwin.com) on Windows,
        be sure to unpack with "tar" to set up the symbolic links.

    c) make
```

```
sed -i -e "s/NEED_sys_errPROTECTEDlist/NEED_sys_errlist/g" libiberty/strerror.c

make all
make install

cd ../simplesim*
make config-pisa
make
# You can check that SimpleScalar (not the toolchain) works with the command-line:
# ./sim-safe tests-pisa/bin.little/test-math
```

2. 编写 shell 脚本，运行模拟器

由于 SPEC2000 运行时间过长，改为测试 spec95，最初编写的 spec2000 的脚本可在 https://github.com/wine99/hfut-cs-assignments/blob/master/calab/lab2/lab2.sh 查看（与下面的 lab2_95.sh 基本相同）

```
#!/bin/bash

# put this in build/simplesim-3.0/lab2

opt=(
    " taken"
    " nottaken"
    ":bimod 512"
    ":bimod 1024"
    ":2lev 1 1024 8 0"
    ":2lev 1 64 6 1"
)

exe=(
    "cc1"
    "compress95"
    "go"
    "perl"
)

arg=(
    "-O ../benchmark/spec95_little/1stmt.i"
    "< ../benchmark/spec95_little/input.log"
    "50 9 ../benchmark/spec95_little/2stone9.in"
    "../benchmark/spec95_little/perl-tests.pl"
)

rm lab2_95.out
rm temp/lab2_95.out
```

```
cp ../benchmark/spec95_little/input.log.bak ../benchmark/spec95_litt
le/input.log

for (( i = 0 ; i < ${#exe[@]} ; i++ ))
do
    for (( j = 0 ; j < ${#opt[@]} ; j++ ))
    do
        # out of no reason compress95 will delete input.log after us
ing it when run this command in shell script
        if [ -f ../benchmark/spec95_little/input.log.Z ]; then
            rm ../benchmark/spec95_little/input.log.Z
            cp ../benchmark/spec95_little/input.log.bak ../benchmark
/spec95_little/input.log
        fi
        echo "./../sim-bpred -bpred${opt[$j]} ../benchmark/spec95_li
ttle/${exe[$i]}.alpha ${arg[$i]} >> temp/lab2_95.out 2>> lab2_95.out
"
        ./../sim-bpred -bpred${opt[$j]} ../benchmark/spec95_little/$
{exe[$i]}.alpha ${arg[$i]} >> temp/lab2_95.out 2>> lab2_95.out
    done
done
```

3. 填表

Benchmark: cc1

	Always taken	always not taken	bimod(512)	Bimod(1024)	Two level (1,1024,8,0)	Two level (1,64,6,1)
sim_total_insn	337380951	337380951	337380951	337380951	337380951	337380951
sim_total_refs	121908627	121908627	121908627	121908627	121908627	121908627
sim_num_branches	58883165	58883165	58883165	58883165	58883165	58883165
sim_elapsed_time	11	11	13	14	14	14
sim_inst_rate	30670995.5455	30670995.5455	25952380.8462	24098639.3571	24098639.3571	24098639.3571
sim_IPB	5.7297	5.7297	5.7297	5.7297	5.7297	5.7297
lookups	58883165	58883165	58883165	58883165	58883165	58883165
updates	58883165	58883165	58883165	58883165	58883165	58883165

addr_hits	35353186	37437272	49434579	50495518	51124384	51124384
dir_hirts	35353186	37437272	50889547	51952955	52595026	52595026
misses	23529979	21445893	7993618	6930210	6288139	6288139
jr_hits	6303731	6303731	5094531	5094531	5094531	5094531
jr_seen	6303731	6303731	6303731	6303731	6303731	6303731
jr_non_ras_hits.PP	6303731	6303731	534845	534845	534845	534845
jr_non_ras_seen.PP	6303731	6303731	1713359	1713359	1713359	1713359
bpred_addr_rate	0.6004	0.6358	0.8395	0.8576	0.8682	0.8682
bpred_dir_rate	0.6004	0.6358	0.8642	0.8823	0.8932	0.8932
bpred_jr_rate	1.0000	1.0000	0.8082	0.8082	0.8082	0.8082
bpred_jr_non_ras_rate.PP	1.0000	1.0000	0.3122	0.3122	0.3122	0.3122
retstack_pushes	0	0	4590377	4590377	4590377	4590377
retstack_pops	0	0	4590372	4590372	4590372	4590372
used_ras.PP	0	0	4590372	4590372	4590372	4590372
ras_hits.PP	0	0	4559686	4559686	4559686	4559686
ras_rate.PP			0.9933	0.9933	0.9933	0.9933

Benchmark: compress95

	Always taken	always not taken	bimod(512)	Bimod(1024)	Two level (1,1024,8,0)	Two level (1,64,6,1)
sim_total_insn	265510975	265510975	265510975	265510975	265510975	265510975
sim_total_refs	92890164	92890164	92890164	92890164	92890164	92890164
sim_num_branches	35816922	35816922	35816922	35816922	35816922	35816922
sim_elapsed_time	10	10	11	13	11	12
sim_inst_rate	26551097.5000	26551097.5000	24137361.3636	20423921.1538	24137361.3636	22125914.5833
sim_IPB	7.4130	7.4130	7.4130	7.4130	7.4130	7.4130
lookups	35816922	35816922	35816922	35816922	35816922	35816922
updates	35816922	35816922	35816922	35816922	35816922	35816922
addr_hits	25519618	15500973	31314962	31315290	31315296	31315296
dir_hirts	25519618	15500973	31315241	31315573	31315578	31315578

misses	10297304	20315949	4501681	4501349	4501344	4501344
jr_hits	796249	796249	796119	796119	796119	796119
jr_seen	796249	796249	796249	796249	796249	796249
jr_non_ras_hits.PP	796249	796249	397969	397969	397969	397969
jr_non_ras_seen.PP	796249	796249	398096	398096	398096	398096
bpred_addr_rate	0.7125	0.4328	0.8743	0.8743	0.8743	0.8743
bpred_dir_rate	0.7125	0.4328	0.8743	0.8743	0.8743	0.8743
bpred_jr_rate	1.0000	1.0000	0.9998	0.9998	0.9998	0.9998
bpred_jr_non_ras_rate.PP	1.0000	1.0000	0.9997	0.9997	0.9997	0.9997
retstack_pushes	0	0	398156	398156	398156	398156
retstack_pops	0	0	398153	398153	398153	398153
used_ras.PP	0	0	398153	398153	398153	398153
ras_hits.PP	0	0	398150	398150	398150	398150
ras_rate.PP			1.0000	1.0000	1.0000	1.0000

Benchmark: go

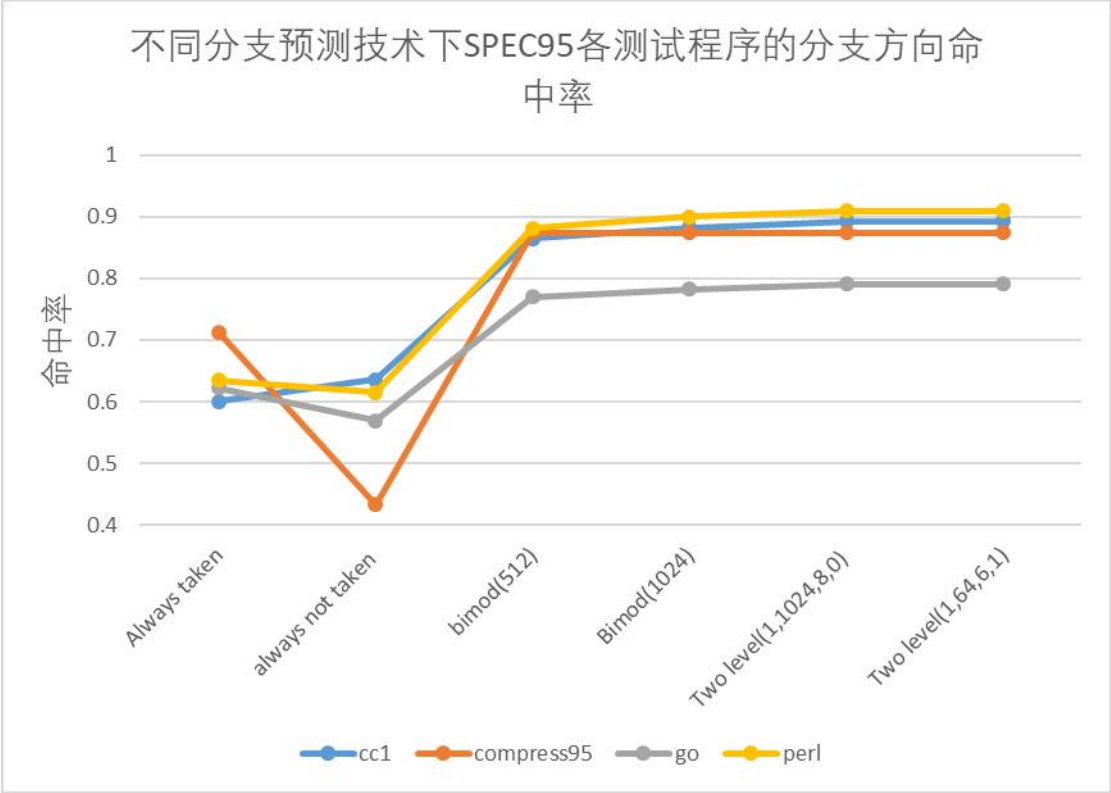
	Always taken	always not taken	bimod(512)	Bimod(1024)	Two level (1,1024,8,0)	Two level (1,64,6,1)
sim_total_insn	545812316	545812316	545812316	545812316	545812316	545812316
sim_total_refs	211690501	211690501	211690501	211690501	211690501	211690501
sim_num_branches	73904236	73904236	73904236	73904236	73904236	73904236
sim_elapsed_time	17	17	21	21	21	22
sim_inst_rate	32106606.8235	32106606.8235	25991062.6667	25991062.6667	25991062.6667	24809650.7273
sim_IPB	7.3854	7.3854	7.3854	7.3854	7.3854	7.3854
lookups	73904236	73904236	73904236	73904236	73904236	73904236
updates	73904236	73904236	73904236	73904236	73904236	73904236
addr_hits	45973424	42043847	56477279	57354137	58003198	58003198
dir_hirts	45973424	42043847	56938622	57815128	58474279	58474279
misses	27930812	31860389	16965614	16089108	15429957	15429957
jr_hits	6257514	6257514	6202347	6202347	6202347	6202347
jr_seen	6257514	6257514	6257514	6257514	6257514	6257514

jr_non_ras_hits.PP	6257514	6257514	64341	64341	64341	64341
jr_non_ras_seen.PP	6257514	6257514	117955	117955	117955	117955
bpred_addr_rate	0.6221	0.5689	0.7642	0.7761	0.7848	0.7848
bpred_dir_rate	0.6221	0.5689	0.7704	0.7823	0.7912	0.7912
bpred_jr_rate	1.0000	1.0000	0.9912	0.9912	0.9912	0.9912
bpred_jr_non_ras_rate.PP	1.0000	1.0000	0.5455	0.5455	0.5455	0.5455
retstack_pushes	0	0	6139564	6139564	6139564	6139564
retstack_pops	0	0	6139559	6139559	6139559	6139559
used_ras.PP	0	0	6139559	6139559	6139559	6139559
ras_hits.PP	0	0	6138006	6138006	6138006	6138006
ras_rate.PP			0.9997	0.9997	0.9997	0.9997

Benchmark: perl

	Always taken	always not taken	bimod(512)	Bimod(1024)	Two level (1,1024,8,0)	Two level (1,64,6,1)
sim_total_insn	25230886	25230886	25230886	25230886	25230886	25230886
sim_total_refs	8752506	8752506	8752506	8752506	8752506	8752506
sim_num_branches	4103258	4103258	4103258	4103258	4103258	4103258
sim_elapsed_time	1	1	1	1	1	1
sim_inst_rate	25230886.0000	25230886.0000	25230886.0000	25230886.0000	25230886.0000	25230886.0000
sim_IPB	6.1490	6.1490	6.1490	6.1490	6.1490	6.1490
lookups	4103258	4103258	4103258	4103258	4103258	4103258
updates	4103258	4103258	4103258	4103258	4103258	4103258
addr_hits	2606989	2524282	3530736	3610135	3646550	3646550
dir_hirts	2606989	2524282	3615892	3695132	3731570	3731570
misses	1496269	1578976	487366	408126	371688	371688
jr_hits	432711	432711	358410	358410	358410	358410
jr_seen	432711	432711	432711	432711	432711	432711
jr_non_ras_hits.PP	432711	432711	10562	10562	10562	10562
jr_non_ras_seen.PP	432711	432711	83073	83073	83073	83073
bpred_addr_rate	0.6353	0.6152	0.8605	0.8798	0.8887	0.8887
bpred_dir_rate	0.6353	0.6152	0.8812	0.9005	0.9094	0.9094
bpred_jr_rate	1.0000	1.0000	0.8283	0.8283	0.8283	0.8283
bpred_jr_non_ras_rate.PP	1.0000	1.0000	0.1271	0.1271	0.1271	0.1271
retstack_pushes	0	0	349675	349675	349675	349675
retstack_pops	0	0	349638	349638	349638	349638
used_ras.PP	0	0	349638	349638	349638	349638
ras_hits.PP	0	0	347848	347848	347848	347848
ras_rate.PP			0.9949	0.9949	0.9949	0.9949

4. 画图



3. 结果分析

SimpleScalar 分支预测的实现方法：先进行分支方向探测，即是否采取分支（当然跳转指令和调用返回指令不用作这一步），接着是生成分支地址，对于调返指令，直接在 RAS 上作相关操作，普通分支指令则要利用 BTB 来进行地址探测，命中则生成地址。然后对两步综合，地址命中且分支预测为采取，返回分支目标地址；地址不命中且分支预测为采取，返回 1;只要分支预测为不采取，就返回 0。

从上述描述可知，为了更合理地比较不同预测技术对分支的预测效果，应当选取 bpred_dir_rate，即对分支的方向的预测正确率，这个统计量来绘图分析，如上所示。

可以看到，两种静态预测 taken 和 nottaken 效果明显较差，其中 taken 略好于 nottaken，尤其是对 compress95 测试程序来说，这是因为代码中的许多分支是倾向于接受的，例如各种循环过程的判断条件。

而对于两种动态预测方法 bimod 和 two level 而言，它们的预测能力大致相当，且并没有随着参数的改变而获得太多的性能提升。

4. 实验心得

这次实验不仅加深了我对分支预测的理解，也使我学到了其他实用的技能。例如学会了编写简单的 shell 脚本，学会了怎么写循环，如何定义和使用变量，输出和追加输出重定向等等。

实验中也遇到了不少的困难。例如安装 SimplaScalar，起初模拟器的 ISA 问题一直没有解决，直到开始耐心浏览安装包各处的 README 以及寻求了老师的帮助之后，才成功地修改了 ISA 并重新编译。

实验中也有许多细节尚未完全弄懂。例如为什么静态预测方法的 addr_hit 和 dir_hit 是相同的，为什么 updates 数目和 lookups 数目相同而不是更少，为什么两个不同参数的 two level 预测结果的各个统计量几乎一模一样，two level 预测原理的细节是怎样的，ras、pp 等统计量的含义是什么，等等。由于精力和水平有限，这些问题只能期待以后进一步的学习研究了。

lab2_95.out 原始数据见：
https://github.com/wine99/hfut-cs-assignments/blob/master/calab/lab2/lab2_95.out。

实验三、缓存性能分析

1. 实验内容

通过实验和结果分析，理解缓存的各种参数对缓存性能的影响。

1. 安装和测试 SimpleScalar 模拟器(利用模拟器自带的测试程序进行测试)。

2. 在基本配置情况下运行 SPEC 2000 基准测试集下面的 4 个程序(请指明自己选的是哪些测试程序)统计 Cache 失效次数，并统计 L2 缓存的失效次数（注：配置二级缓存结构，指令和数据合在一起）。

3. 改变 Cache 容量（*2，*4，*8，*64），运行相同的测试程序，并统计 L2 缓存的失效次数计算失效率，并对结果进行总结分析。

4. 改变 Cache 的相联度（2 路，4 路，8 路，16 路，64 路），运行 1 中所选择的测试程序，并统计 L2 缓存的失效次数计算失效率，并对结果进行分析。

5. 改变 Cache 块大小（*2，*4，*8，*64），运行 1 中所选择的测试程序，并统计 L2 缓存的失效次数计算失效率，并进行分析。

2. 实验过程

```
#
# -option        <args>           #    <default> # description
#
-config          <string>         #       <null> # load configuration from a file
-dumpconfig      <string>         #       <null> # dump configuration to a file
-h               <true|false>     #        false # print help message
-v               <true|false>     #        false # verbose operation
-d               <true|false>     #        false # enable debug message
-i               <true|false>     #        false # start in Dlite debugger
-seed            <int>            #            1 # random number generator seed (0 for timer seed)
-q               <true|false>     #        false # initialize and terminate immediately
-chkpt           <string>         #       <null> # restore EIO trace execution from <fname>
-redir:sim       <string>         #       <null> # redirect simulator output to file (non-interactive only)
-redir:prog      <string>         #       <null> # redirect simulated program output to file
-nice            <int>            #            0 # simulator scheduling priority
-max:inst        <uint>           #            0 # maximum number of inst's to execute
-cache:dl1       <string>         # dl1:256:32:1:l # l1 data cache config, i.e., {<config>|none}
-cache:dl2       <string>         # ul2:1024:64:4:l # l2 data cache config, i.e., {<config>|none}
-cache:il1       <string>         # il1:256:32:1:l # l1 inst cache config, i.e., {<config>|dl1|dl2|none}
-cache:il2       <string>         #          dl2 # l2 instruction cache config, i.e., {<config>|dl2|none}
-tlb:itlb        <string>         # itlb:16:4096:4:l # instruction TLB config, i.e., {<config>|none}
-tlb:dtlb        <string>         # dtlb:32:4096:4:l # data TLB config, i.e., {<config>|none}
-flush           <true|false>     #        false # flush caches on system calls
-cache:icompress <true|false>     #        false # convert 64-bit inst addresses to 32-bit inst equivalents
-pcstat          <string list...> #       <null> # profile stat(s) against text addr's (mult uses ok)

  The cache config parameter <config> has the following format:

    <name>:<nsets>:<bsize>:<assoc>:<repl>
```

阅读打印出来的帮助信息，可知默认的 cache 配置为：

```
-cache:dl1 # dl1:256:32:1:l
-cache:dl2 # ul2:1024:64:4:l
-cache:il1 # il1:256:32:1:l
-cache:il2 # dl2
```

一级数据缓存和一级指令缓存配置为 256 组，相联度为 1，亦即全相联，每块 32B，总大小为 8KB。二级缓存数据和指令存放在一起（il2 指向 dl2），共 1024 组，相联度为 4，块大小为 64B，总大小为 256KB。

按照实验要求编写脚本，由于 SPEC2000 运行时间过长，改用 SPEC95。

一级指令缓存保持默认配置，一级数据缓存在默认配置的基础上进行修改。修改某一变量时保持其他变量不变。例如增大相联度和增大块大小时相应地减小组数，以保持总容量不变。

二级缓存的块大小由默认的 64 修改为 256，因为需要测试不同缓存块大小的影响，而实验过程中发现，当一级缓存块大小超过二级缓存块大小时，将出现以下错误。

```
sim: ** starting functional simulation w/ caches **
fatal: cache: access error: access spans block, addr 0x1ff96f80

sim: ** simulation statistics **
sim_num_insn                  2 # total number of instructions executed
```

```
#!/bin/bash

# put this in build/simplesim-3.0/lab3

opt=(
    "-cache:dl1 dl1:256:32:1:l"
    # change capacity *2 *4 *8 *64
    "-cache:dl1 dl1:512:32:1:l"
    "-cache:dl1 dl1:1024:32:1:l"
    "-cache:dl1 dl1:2048:32:1:l"
    "-cache:dl1 dl1:16384:32:1:l"
    # change association 2 4 8 16 64
    "-cache:dl1 dl1:128:32:2:l"
    "-cache:dl1 dl1:64:32:4:l"
    "-cache:dl1 dl1:32:32:8:l"
    "-cache:dl1 dl1:16:32:16:l"
    "-cache:dl1 dl1:4:32:64:l"
    # change block size *2 *4 *8 *64
    "-cache:dl1 dl1:128:64:1:l"
    "-cache:dl1 dl1:64:128:1:l"
    "-cache:dl1 dl1:32:256:1:l"
    # 2048 is bigger than the block size of l2 cache
    # "-cache:dl1 dl1:4:2048:1:l"
)

exe=(
    "cc1"
    "compress95"
    "go"
```

```
    "perl"
)

arg=(
    "-O ../benchmark/spec95_little/1stmt.i"
    "< ../benchmark/spec95_little/input.log"
    "50 9 ../benchmark/spec95_little/2stone9.in"
    "../benchmark/spec95_little/perl-tests.pl"
)

rm lab3_95.out
rm temp/lab3_95.out
cp ../benchmark/spec95_little/input.log.bak ../benchmark/spec95_litt
le/input.log

for (( i = 0 ; i < ${#exe[@]} ; i++ ))
do
    for (( j = 0 ; j < ${#opt[@]} ; j++ ))
    do
        # out of no reason compress95 will delete input.log after us
ing it when run this command in shell script
        if [ -f ../benchmark/spec95_little/input.log.Z ]; then
            rm ../benchmark/spec95_little/input.log.Z
            cp ../benchmark/spec95_little/input.log.bak ../benchmark
/spec95_little/input.log
        fi
        echo "./../sim-cache ${opt[$j]} -cache:dl2 ul2:1024:256:4:l
-cache:il2 dl2 ../benchmark/spec95_little/${exe[$i]}.alpha ${arg[$i]}
 >> temp/lab3_95.out 2>> lab3_95.out"
        ./../sim-cache ${opt[$j]} -cache:dl2 ul2:1024:256:4:l -cache:
il2 dl2 ../benchmark/spec95_little/${exe[$i]}.alpha ${arg[$i]} >> te
mp/lab3_95.out 2>> lab3_95.out
    done
done
```

不同缓存容量下的缺失率

	dl1:256:32:1:l	**dl1:512:32:1:l**	**dl1:1024:32:1:l**	**dl1:2048:32:1:l**	**dl1:16384:32:1:l**
cc1	0.0736	0.0479	0.0305	0.0193	0.0029
compress95	0.1643	0.1496	0.1201	0.1011	0.0251
Go	0.0765	0.0422	0.0197	0.0079	0.0001

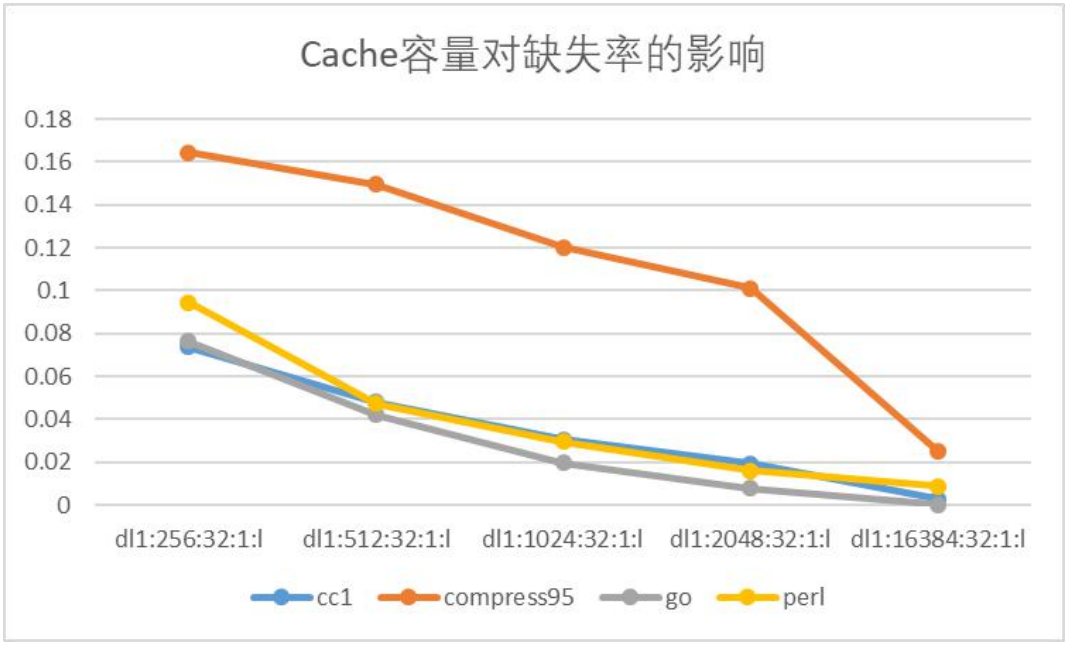
perl	0.0945	0.0474	0.0295	0.0159	0.0089

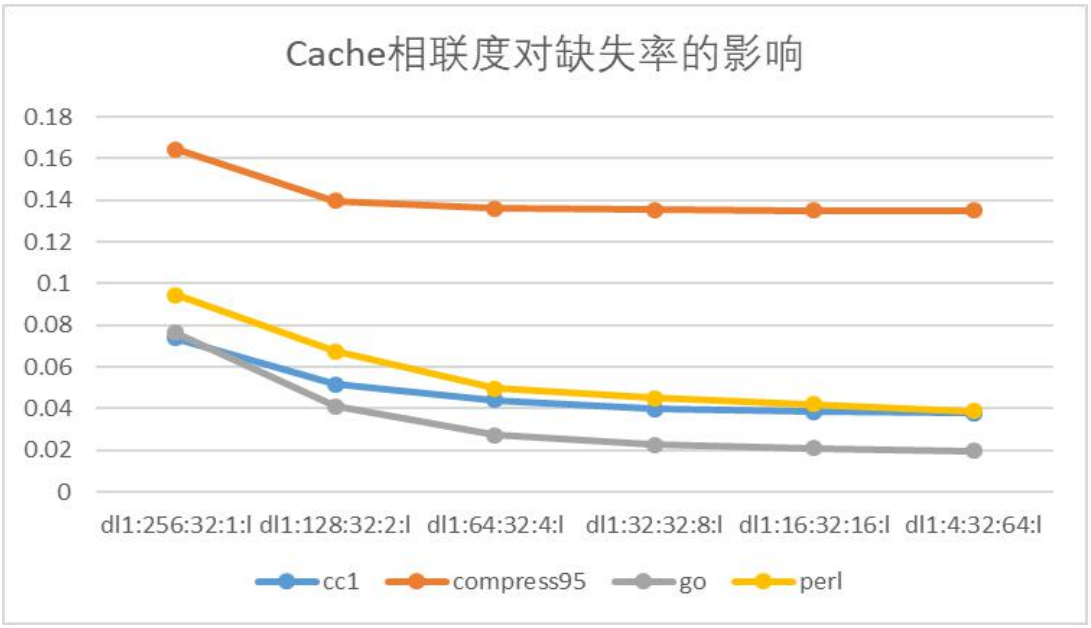
不同相联度下的缺失率

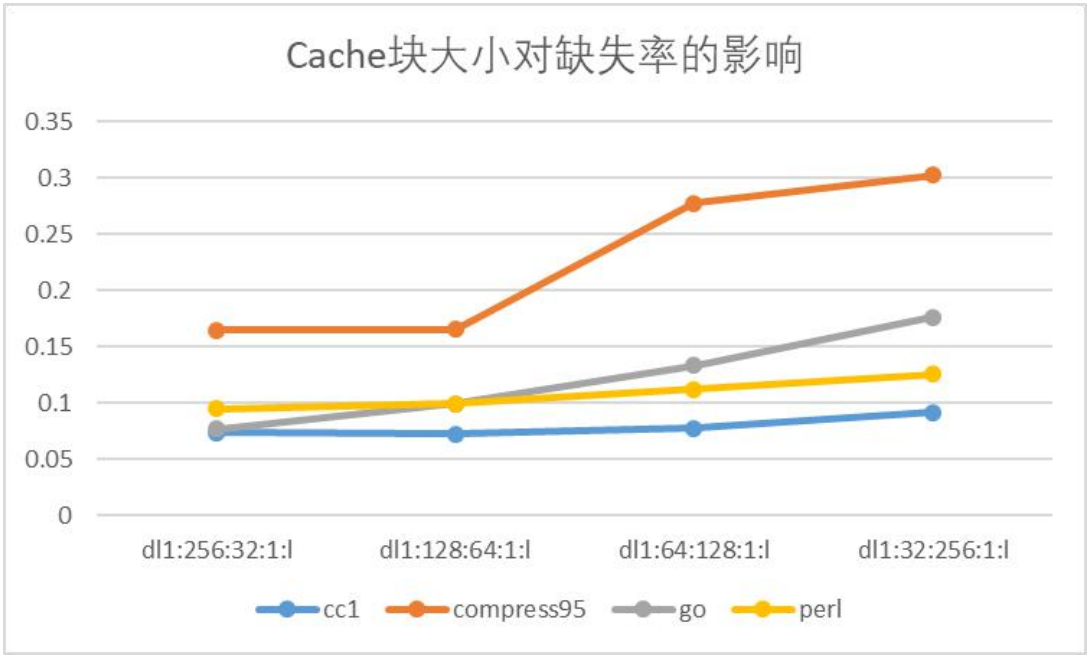
	dl1:256:32:1:l	**dl1:128:32:2:l**	**dl1:64:32:4:l**	**dl1:32:32:8:l**	**dl1:16:32:16:l**	**dl1:4:32:64:l**
cc1	0.0736	0.0517	0.0439	0.0398	0.0385	0.0379
compress95	0.1643	0.1398	0.136	0.1353	0.1352	0.1351
go	0.0765	0.0412	0.0273	0.0227	0.0208	0.0195
perl	0.0945	0.0675	0.0498	0.0452	0.0422	0.0389

不同块大小下的缺失率

	dl1:256:32:1:l	**dl1:128:64:1:l**	**dl1:64:128:1:l**	**dl1:32:256:1:l**
cc1	0.0736	0.0722	0.0776	0.0917
compress95	0.1643	0.1649	0.2772	0.3024
go	0.0765	0.0988	0.1329	0.1759
perl	0.0945	0.0988	0.112	0.1255







3. 结果分析

从结果来看，虽然增加 Cache 容量只能减小容量不命中，但却是对命中率提高是最有效的（在控制其他条件不变的情况下），但 Cache 容量显然不能无限提高，而且这也会加大命中时间。

增加相联度可以减小冲突不命中，但也会增加命中时间。实验结果也和理论相符，简单的 4 路组相联已经能获得不错的命中率的提升，继续增加相联度带来的增益实际情况下很可能不如带来的额外硬件开销。

单纯的增加块大小，由于加强了空间局部性，会使强制不命中率下降，但由于保持了 Cache 总容量不变，块大小的增加导致了块数地减小，冲突不命中率将上升，并且当块大小

增加到一定程度后，进入块中的数据很可能也不再符合局部性规律。对于本次实验的测试而言，只有一个 cc1 测试程序在块大小*2 后缺失率略微有所下降，其余程序在块大小增大而总容量不变的情况下，总体缺失率均在上升。

4. 实验心得

由于有了实验二的脚本编写经验，实验三的进度加快了不少。通过此次实验，我更加深刻地理解了各种 Cache 参数对缺失率的影响，也一定程度上验证了课本上所列的结论。但块大小对一级数据缓存的减益效果是我没有预料到的。

lab3_95.out 原始数据见：
https://github.com/wine99/hfut-cs-assignments/blob/master/calab/lab3/lab3_95.out。

成绩评定：

指导老师：李建华