深 圳 大 学 实 验 报 告

课 程 名 称：计算机系统(3)

实验项目名称：处理器结构实验二

学　　院：计算机与软件学院

专　　业：软件工程（腾班）

指 导 教 师：王　毅

报告人：黄亮铭　学号：2022155028　班级：腾班

实 验 时 间：2024 年 11 月 22 日

实验报告提交时间：2024 年 12 月 06 日

教务处制

一、实验目的

1. 了解控制冒险分支预测的概念
2. 了解多种分支预测的方法，动态分支预测更要深入了解
3. 理解什么是 BTB（Branch Target Buffer），并且学会用 BTB 来优化所给程序
4. 利用 BTB 的特点，设计并了解在哪种状态下 BTB 无效
5. 了解循环展开，并于 BTB 功能进行对比
6. 对 WinMIPS64 的各个窗口和操作更加熟悉

二、实验内容

按照下面的实验步骤及说明，完成相关操作记录实验过程的截图：

首先，给出一段矩阵乘法的代码，通过开启 BTB 功能对其进行优化，并且观察流水线的细节，解释 BTB 在其中所起的作用；

其次，自行设计一段使得即使开启了 BTB 也无效的代码。

第三，使用循环展开的方法，观察流水因分支停顿的次数减少的现象，并对比采用 BTB 结构时流水因分支而停顿的次数。

（选做：在 x86 系统上编写 C 语言的矩阵乘法代码，用 perf 观察分支预测失败次数，分析其次数是否与你所学知识吻合。再编写前面第二部使用的令分支预测失败的代码，验证 x86 是否能正确预测，并尝试做解释）

三、实验环境

硬件：桌面 PC

软件：Windows

四、实验步骤及说明

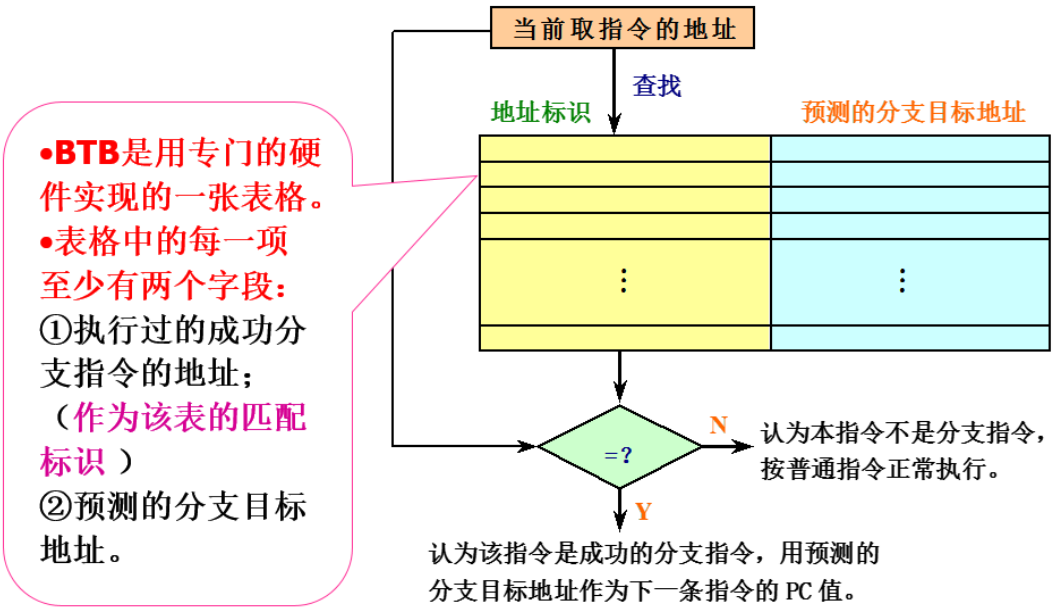
背景知识

在遇到跳转语句的时候，我们往往需要等到 MEM 阶段才能确定这条指令是否跳转（通过硬件的优化，可以极大的缩短分支的延迟，将分支执行提前到 ID 阶段，这样就能够将分支预测错误代价减小到只有一条指令），这种为了确保预取正确指令而导致的延迟叫控制冒险（分支冒险）。

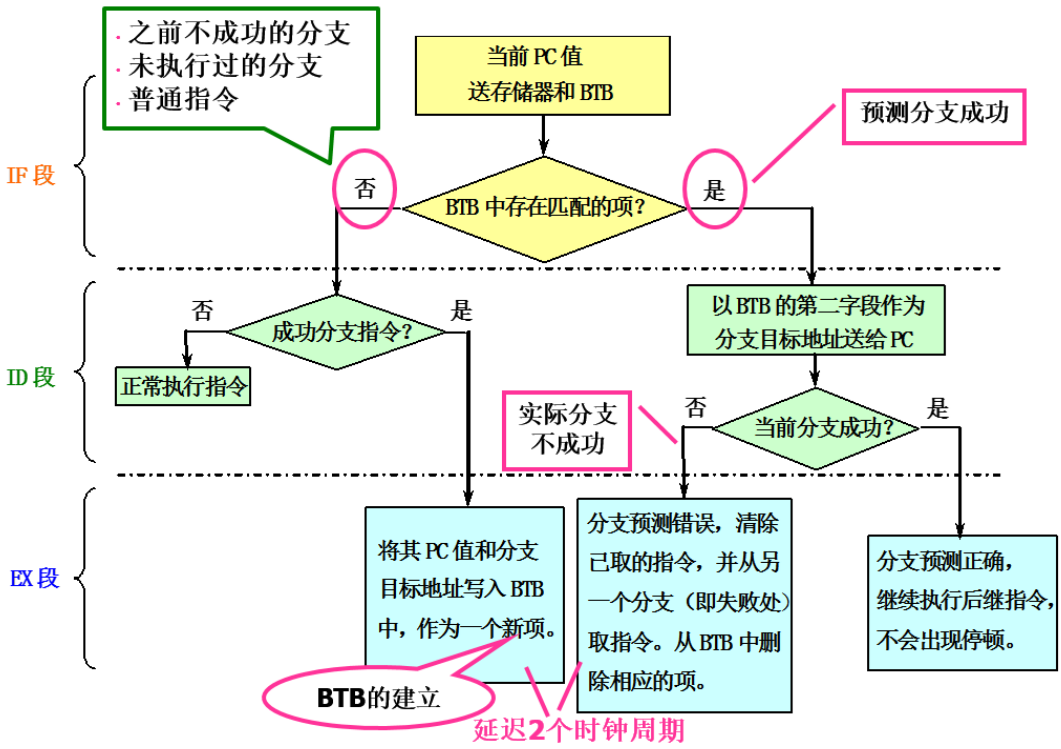
为了降低控制冒险所带来的性能损失，一般采用分支预测技术。分支预测技术包含编译时进行的静态分支预测，和执行时进行的动态分支预测。这里，我们着重介绍动态分支预测中的 BTB（Branch Target Buffer）技术。

BTB 即为分支目标缓冲器，它将分支指令（对应的指令地址）放到一个缓冲区中保存起来，当下次再遇到相同的指令（跳转判定）时，它将执行和上次一样的跳转（分支或不分支）预测。

一种可行的 BTB 结构示意图如下：



在采用了 BTB 之后，在流水线各个阶段所进行的相关操作如下：



注意，为了填写 BTB，需要额外一个周期。

（一） 矩阵乘法及优化

在这一阶段，我们首先给出矩阵乘法的例子，接着将流水线设置为不带 BTB 功能（configure->enable branch target buffer）直接运行，观察结果进行记录；然后，再开启 BTB 功能再次运行，观察实验结果。将两次的实验结果进行对比，观察 BTB 是否起作用，如果有效果则进一步观察流水线执行细节并且解释 BTB 起作用原因。

矩阵乘法的代码如下：

```
        .data
str:  .asciiz "the data of matrix 3:\n"
mx1:   .space 512
mx2:   .space 512
mx3:   .space 512

            .text
initial:    daddi r22,r0,mx1  #这个initial模块是给三个矩阵赋初值
            daddi r23,r0,mx2
            daddi r21,r0,mx3
input:      daddi r9,r0,64
            daddi r8,r0,0
loop1:      dsll r11,r8,3 #将矩阵mx1初始化为2，4，6…16，mx2初始化为3，6，9…24
            dadd r10,r11,r22
            dadd r11,r11,r23
            daddi r12,r0,2
            daddi r13,r0,3
            sd r12,0(r10)
            sd r13,0(r11)

            daddi r8,r8,1
            slt r10,r8,r9
            bne r10,r0,loop1

mul:        daddi r16,r0,8
            daddi r17,r0,0    # r17：I r18：j r19：k
loop2:      daddi r18,r0,0    #这个循环是执行for(int i = 0, i < 8; i++)的内容
loop3:      daddi r19,r0,0    #这个循环是执行for(int j = 0, j < 8; j++)的内容
            daddi r20,r0,0    #r20存储在计算result[i][j]过程中每个乘法结果的叠加值
loop4:      dsll r8,r17,6     #这个循环的执行计算每个result[i][j]
            dsll r9,r19,3
            dadd r8,r8,r9
            dadd r8,r8,r22
            ld r10,0(r8)      #取mx1[i][k]的值
            dsll r8,r19,6
            dsll r9,r18,3
            dadd r8,r8,r9
```

```
        dadd r8,r8,r23
        ld r11,0(r8)       #取mx2[k][j]的值
        dmul r13,r10,r11   #mx1[i][k]与mx2[k][j]相乘
        dadd r20,r20,r13   #中间结果累加

        daddi r19,r19,1
        slt r8,r19,r16
        bne r8,r0,loop4

        dsll r8,r17,6
        dsll r9,r18,3
        dadd r8,r8,r9
        dadd r8,r8,r21     #计算result[i][j]的位置
        sd r20,0(r8)       #将结果存入result[i][j]中

        daddi r18,r18,1
        slt r8,r18,r16
        bne r8,r0,loop3

        daddi r17,r17,1
        slt r8,r17,r16
        bne r8,r0,loop2

        halt
```

不设置 BTB 功能，运行该程序，观察 Statistics 窗口的结果截屏并记录下来。

① 首先将上述代码存储到martix-multiply.s文件中，然后使用asm.exe验证代码是否存在错误，得到结果如下所示，说明代码没有错误。

```
Pass 2 completed with 0 errors
Code Symbol Table
             initial = 00000000
               input = 0000000c
               loop1 = 00000014
                 mul = 0000003c
               loop2 = 00000044
               loop3 = 00000048
               loop4 = 00000050
Data Symbol Table
                 str = 00000000
                 mx1 = 00000018
                 mx2 = 00000218
                 mx3 = 00000418

E:\winmips64>
```

图1 asm验证结果

② 将martix-multiply.s文件导入到winmips64中，注意要在菜单栏的Configure项取消勾选Enable Branch Target Buffer。

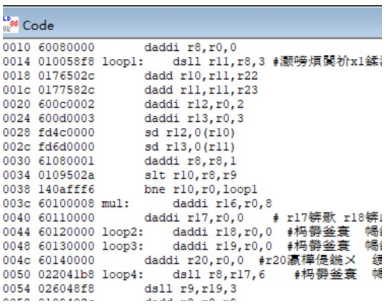


图2a导入的文件（部分）

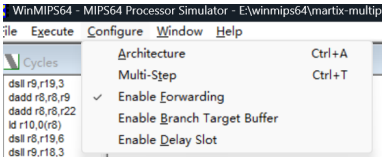


图2b取消勾选

③ 按F4执行代码，结果如下图所示，一共发生了574次Branch Taken Stalls。

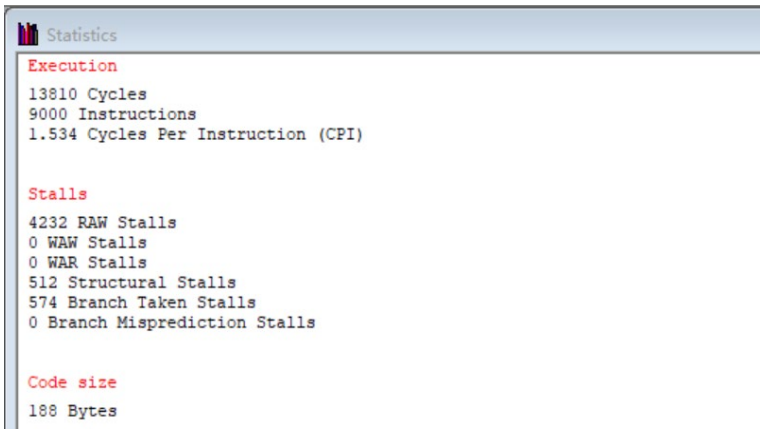


图3代码执行结果

接着，设置 BTB 功能（在菜单栏处选择 Configure 项，然后在下拉菜单中为 Enable Branch Target Buffer 选项划上钩）。并在此运行程序，观察 Statistics 窗口的结果并截屏记录下来。

① 重新载入上述文件，在菜单栏的Configure项中勾选Enable Branch Target Buffer。

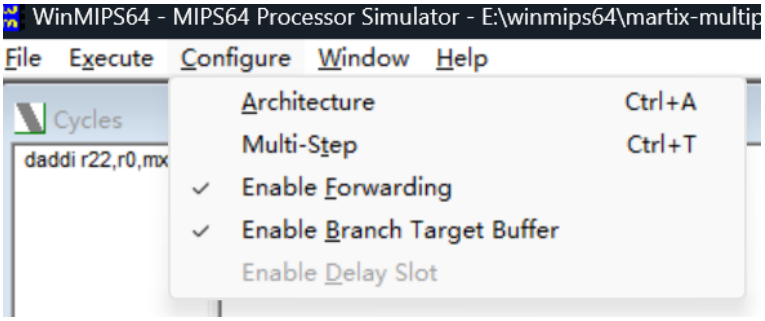


图4勾选相应选项

② 按F4执行代码，结果如下图所示，发生了148次Branch Taken Stalls和148次Branch Misprediction Stalls。

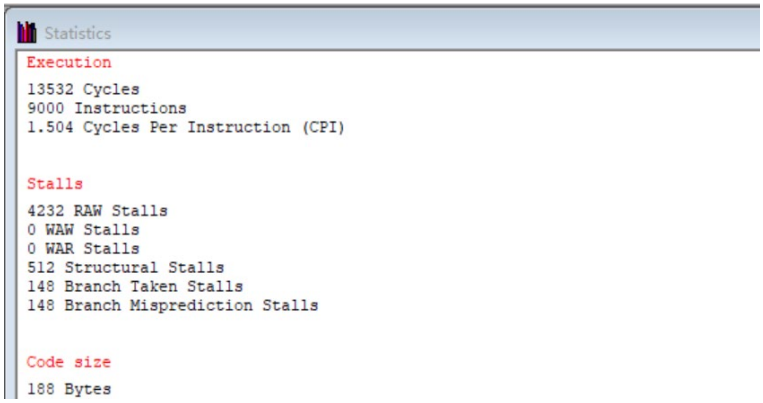


图5代码执行结果

在这里，我们仅仅观察比较 Stalls 中的最后两项------Branch Taken Stalls（后称 BTS）和 Branch Misprediction Stalls（后称 BMS）。

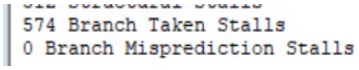


图 6a 未勾选

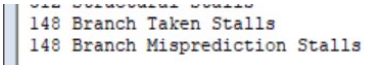


图 6b 已勾选

接下来，对比其结果。我们就结合流水线执行细节分析造成这种情况发生的原因。

对于未勾选的情况：winmips64 默认预测不跳转。

① Init 部分：给三个 8*8 的矩阵赋值，一共循环 64 次，只有最后一次不跳转，所以发生 8*8-1=63 次 BTS。

② Mul 部分：与 init 部分一样，每一重循环仅有最后一次预测正确，因此第一重循环发生 7 次 BTS，第二重循环发生 8*7=56 次 BTS，第三重循环发生 8*8*7=448 次 BTS。

③ 两部分一共发生 63+7+56+448=574 次 BTS。

对于勾选的情况：开始时发生 1 次 BTS，分支预测结果被写入 BTB 中；结束时发生 1 次 BMS，并删除 BTB 中的预测结果。综上所述，BTS 的次数=BMS 的次数。

① Init 部分：循环开始和结束时各发生 1 次 BTS 和 1 次 BMS。

② Mul 部分：第一重循环会被执行 1 次，发生 1 次 BTS 和 1 次 BMS；第二重循环会被执行 8 次，发生 8 次 BTS 和 8 次 BMS；第三重循环会被执行 64 次，发生 64 次 BTS 和 64 次 BMS。

③ 两部分的总次数=(1+1+8+64)*2=148 次。

（二）设计使 BTB 无效的代码

在这个部分，我们要设计一段代码，这段代码包含了一个循环。根据 BTB 的特性，我们设计的这个代码将使得 BTB 的开启起不到相应的优化作用，反而会是的性能大大降低。

提示：一定要利用 BTB 的特性，即它的跳转判定是根据之前跳转成功与否来决定的。

给出所用代码以及设计思路，给出运行结果的截屏证明代码实现了目标。

设计思路：根据第一部分的分析，我知道了 BTS 主要发生在 mul 部分。此外，由 BTB 特性，我需要设计循环，让循环第一次跳转成功，第二次跳转失败。所以我需要设计一个双重循环计算矩阵乘法，且矩阵的大小为 2*2。除了上述的 mul 部分，我还需要考虑 init 部分。一个解决办法为将单重循环初始化改为双重循环初始化。简单起见，我将 init 部分直接删除。

所用代码：如图下所示。

```
            .data
str:    .asciiz "the data of matrix 3:\n"
mx1:    .space 32
mx2:    .space 32
mx3:    .space 32

                .text
initial:    daddi r22,r0,mx1    #这个 initial 模块是给三个矩阵赋初值
                daddi r23,r0,mx2
                daddi r21,r0,mx3
input:          daddi r9,r0,16
                daddi r8,r0,0

mul:            daddi r16,r0,2
                daddi r17,r0,0      # r17：I r18：j r19：k
loop2:          daddi r18,r0,0 #这个循环是执行 for(int i = 0, i < 8; i++)的内容
loop3:          daddi r19,r0,0 #这个循环是执行 for(int j = 0, j < 8; j++)的内容
                daddi r20,r0,0 #r20 存储在计算 result[i][j]过程中每个乘法结果的叠加值
loop4:          dsll r8,r17,4       #这个循环的执行计算每个 result[i][j]
                dsll r9,r19,3
                dadd r8,r8,r9
                dadd r8,r8,r22
                ld r10,0(r8)        #取 mx1[i][k]的值
                dsll r8,r19,4
                dsll r9,r18,3
                dadd r8,r8,r9
                dadd r8,r8,r23
                ld r11,0(r8)        #取 mx2[k][j]的值
                dmul r13,r10,r11    #mx1[i][k]与 mx2[k][j]相乘
                dadd r20,r20,r13    #中间结果累加

                daddi r19,r19,1
                slt r8,r19,r16
                bne r8,r0,loop4

                dsll r8,r17,4
                dsll r9,r18,3
                dadd r8,r8,r9
                dadd r8,r8,r21 #计算 result[i][j]的位置
                sd r20,0(r8)        #将结果存入 result[i][j]中

                daddi r18,r18,1
                slt r8,r18,r16
                bne r8,r0,loop3
```

```
        daddi r17,r17,1
        slt r8,r17,r16
        bne r8,r0,loop2

        halt
```

图 7 设计的代码

① 首先将上述代码存储到 martix-multiply-2.s 文件中，然后使用 asm.exe 验证代码是否存在错误。结果如下图所示，说明代码没有错误。

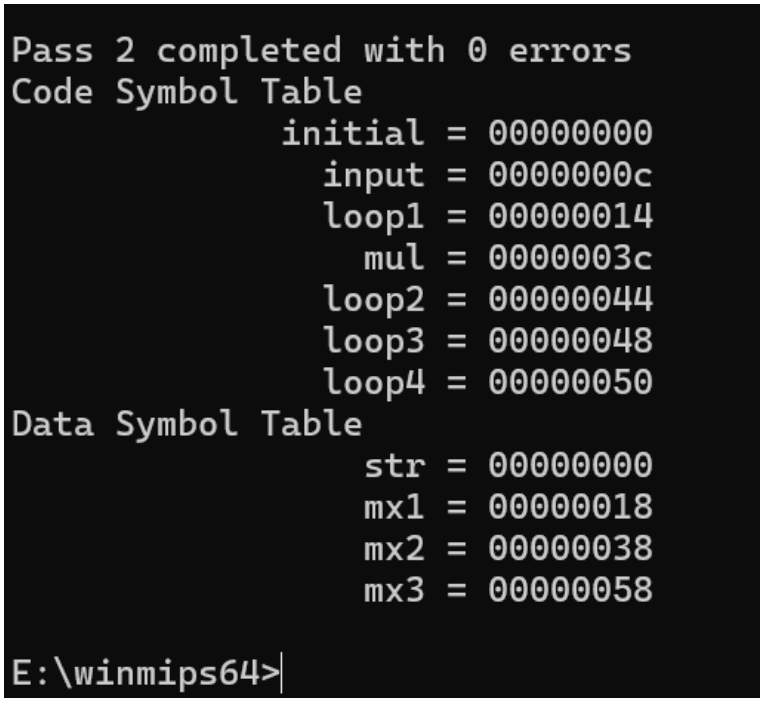


图 8 asm 验证结果

② 将上述文件载入 winmips64 中，不开启 BTB 功能，运行结果如下图所示，结果显示有 7 次 BTS。

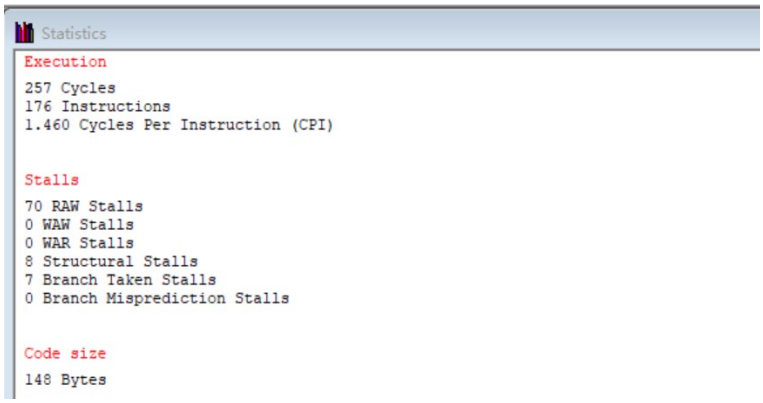


图 9 不开启 BTB 运行结果

③ 将文件重新载入 winmips64，开启 BTB 功能，运行结果如下图所示，结果显示有 14 次 BTS。

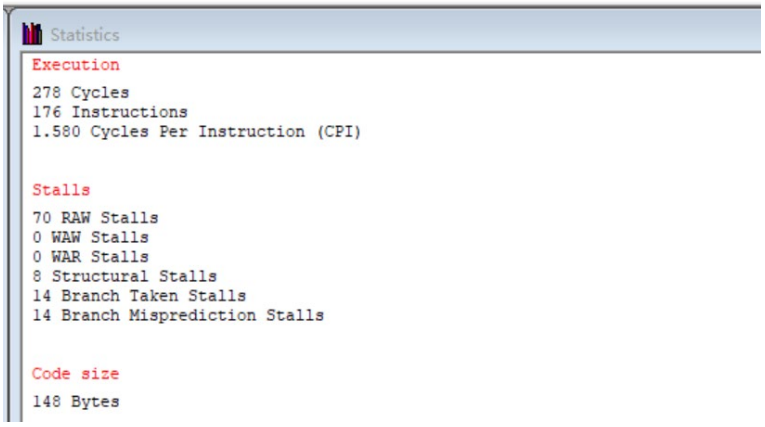


图 10 开启 BTB 运行结果

④ 不开启 BTB 有 7 次 BTS，开启 BTB 有 14 次 BTS。实验结果表明，开启 BTB 不一定能提高性能，还有可能导致程序性能下降，这与程序的具体实现有关。例如，上述程序实现不开启 BTB 比开启 BTB 的性能强大一倍。

（三）循环展开与 BTB 的效果比对

首先，我们需要对循环展开这个概念有一定的了解。

什么是循环展开呢？所谓循环展开就是通过在每次迭代中执行更多的数据操作来减小循环开销的影响。其基本思想是设法把操作对象线性化，并且在一次迭代中访问线性数据中的一个小组而非单独的某个。这样得到的程序将执行更少的迭代次数，于是循环开销就被有效地降低了。

接下来，我们就按照这种思想对上述的矩阵乘法程序进行循环展开。要求将上述的代码通过循环展开将最里面的一个执行迭代 8 次的循环整个展开了，也就是说，我们将矩阵相乘的三个循环通过代码的增加，减少到了两个循环。

展开思路：循环展开实际上是将最里层循环（第三层循环）循环体里的内容重复8次，即复制粘贴7次。通过上述方法就可以实现循环展开。

循环展开代码：如下图所示。

```
            .data
str:    .asciiz "the data of matrix 3:\n"
mx1:    .space 512
mx2:    .space 512
mx3:    .space 512


                .text
initial:        daddi r22,r0,mx1        #这个initial模块是给三个矩阵赋初值
                daddi r23,r0,mx2
                daddi r21,r0,mx3
input:          daddi r9,r0,64
                daddi r8,r0,0
loop1:          dsll r11,r8,3 #将矩阵mx1初始化为2，4，6…16，mx2初始化为3，6，9…24
                dadd r10,r11,r22
                dadd r11,r11,r23
```

```
                daddi r12,r0,2
                daddi r13,r0,3
                sd r12,0(r10)
                sd r13,0(r11)

                daddi r8,r8,1
                slt r10,r8,r9
                bne r10,r0,loop1

mul:            daddi r16,r0,8
                daddi r17,r0,0      # r17：I r18：j r19：k
loop2:          daddi r18,r0,0 #这个循环是执行for(int i = 0, i < 8; i++)的内容
loop3:          daddi r19,r0,0 #这个循环是执行for(int j = 0, j < 8; j++)的内容
                daddi r20,r0,0 #r20存储在计算result[i][j]过程中每个乘法结果的叠加值
loop4:
                # k=0
                dsll r8,r17,6       #这个循环的执行计算每个result[i][j]
                dsll r9,r19,3
                dadd r8,r8,r9
                dadd r8,r8,r22
                ld r10,0(r8)        #取mx1[i][k]的值
                dsll r8,r19,6
                dsll r9,r18,3
                dadd r8,r8,r9
                dadd r8,r8,r23
                ld r11,0(r8)        #取mx2[k][j]的值
                dmul r13,r10,r11    #mx1[i][k]与mx2[k][j]相乘
                dadd r20,r20,r13    #中间结果累加
                daddi r19,r19,1
                # k=1
                dsll r8,r17,6       #这个循环的执行计算每个result[i][j]
                dsll r9,r19,3
                dadd r8,r8,r9
                dadd r8,r8,r22
                ld r10,0(r8)        #取mx1[i][k]的值
                dsll r8,r19,6
                dsll r9,r18,3
                dadd r8,r8,r9
                dadd r8,r8,r23
                ld r11,0(r8)        #取mx2[k][j]的值
                dmul r13,r10,r11    #mx1[i][k]与mx2[k][j]相乘
                dadd r20,r20,r13    #中间结果累加
                daddi r19,r19,1
                # k=2
```

```
dsll r8,r17,6       #这个循环的执行计算每个result[i][j]
dsll r9,r19,3
dadd r8,r8,r9
dadd r8,r8,r22
ld r10,0(r8)        #取mx1[i][k]的值
dsll r8,r19,6
dsll r9,r18,3
dadd r8,r8,r9
dadd r8,r8,r23
ld r11,0(r8)        #取mx2[k][j]的值
dmul r13,r10,r11    #mx1[i][k]与mx2[k][j]相乘
dadd r20,r20,r13    #中间结果累加
daddi r19,r19,1
# k=3
dsll r8,r17,6       #这个循环的执行计算每个result[i][j]
dsll r9,r19,3
dadd r8,r8,r9
dadd r8,r8,r22
ld r10,0(r8)        #取mx1[i][k]的值
dsll r8,r19,6
dsll r9,r18,3
dadd r8,r8,r9
dadd r8,r8,r23
ld r11,0(r8)        #取mx2[k][j]的值
dmul r13,r10,r11    #mx1[i][k]与mx2[k][j]相乘
dadd r20,r20,r13    #中间结果累加
daddi r19,r19,1
# k=4
dsll r8,r17,6       #这个循环的执行计算每个result[i][j]
dsll r9,r19,3
dadd r8,r8,r9
dadd r8,r8,r22
ld r10,0(r8)        #取mx1[i][k]的值
dsll r8,r19,6
dsll r9,r18,3
dadd r8,r8,r9
dadd r8,r8,r23
ld r11,0(r8)        #取mx2[k][j]的值
dmul r13,r10,r11    #mx1[i][k]与mx2[k][j]相乘
dadd r20,r20,r13    #中间结果累加
daddi r19,r19,1
# k=5
dsll r8,r17,6       #这个循环的执行计算每个result[i][j]
dsll r9,r19,3
```

```
dadd r8,r8,r9
dadd r8,r8,r22
ld r10,0(r8)        #取mx1[i][k]的值
dsll r8,r19,6
dsll r9,r18,3
dadd r8,r8,r9
dadd r8,r8,r23
ld r11,0(r8)        #取mx2[k][j]的值
dmul r13,r10,r11    #mx1[i][k]与mx2[k][j]相乘
dadd r20,r20,r13    #中间结果累加
daddi r19,r19,1
# k=6
dsll r8,r17,6       #这个循环的执行计算每个result[i][j]
dsll r9,r19,3
dadd r8,r8,r9
dadd r8,r8,r22
ld r10,0(r8)        #取mx1[i][k]的值
dsll r8,r19,6
dsll r9,r18,3
dadd r8,r8,r9
dadd r8,r8,r23
ld r11,0(r8)        #取mx2[k][j]的值
dmul r13,r10,r11    #mx1[i][k]与mx2[k][j]相乘
dadd r20,r20,r13    #中间结果累加
daddi r19,r19,1
# k=7
dsll r8,r17,6       #这个循环的执行计算每个result[i][j]
dsll r9,r19,3
dadd r8,r8,r9
dadd r8,r8,r22
ld r10,0(r8)        #取mx1[i][k]的值
dsll r8,r19,6
dsll r9,r18,3
dadd r8,r8,r9
dadd r8,r8,r23
ld r11,0(r8)        #取mx2[k][j]的值
dmul r13,r10,r11    #mx1[i][k]与mx2[k][j]相乘
dadd r20,r20,r13    #中间结果累加
daddi r19,r19,1

dsll r8,r17,6
dsll r9,r18,3
dadd r8,r8,r9
dadd r8,r8,r21 #计算result[i][j]的位置
```

```
sd r20,0(r8)          #将结果存入result[i][j]中

daddi r18,r18,1
slt r8,r18,r16
bne r8,r0,loop3

daddi r17,r17,1
slt r8,r17,r16
bne r8,r0,loop2

halt
```

图11矩阵乘法循环展开代码

比较，通过对比循环展开（未启用BTB）、使用BTB（未进行循环展开）以及未使用BTB且未作循环展开的运行结果。比较他们的Branch Tanken Stalls和Branch Misprediction Stalls的数量，并尝试给出评判。

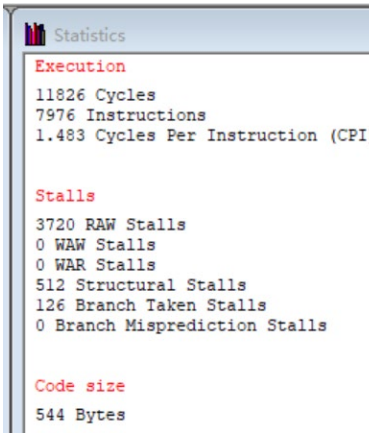


图 12a 循环展开（不启用 BTB）

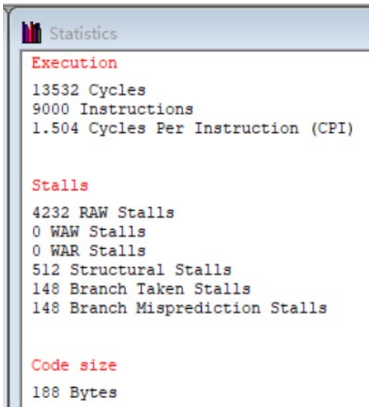


图 12b 未循环展开（启用 BTB）

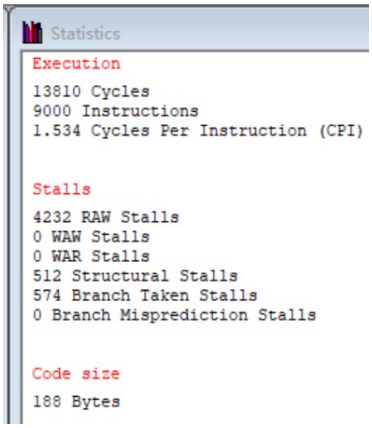


图 12c 未循环展开（不启用 BTB）

循环展开（不启用 BTS）时发生了 126 次 BTS；未循环展开（启用 BTS）时发生了 148 次 BTS；未循环展开（不启用 BTS）时发生了 574 次 BTS。

对比循环展开（不启用 BTS）和未循环展开（不启用 BTS），前者相较于后者减少了 8*8*7=448 次 BTS，恰好为矩阵乘法第三重循环发生的 BTS 次数。

综合上述分析，循环次数较多时开启 BTB 可以提高程序性能；但是循环重数过多时，越内层的循环发生的 BTS 次数越多，开启 BTB 会导致性能下降，此时可以将内层中循环次数较少的循环进行展开操作，减少分支预测错误造成的性能损失，提高程序性能。

五、实验结果

（一） 矩阵乘法及优化

对于不启用 BTB 的情况：winmips64 默认预测不跳转。

① Init 部分：给三个 8*8 的矩阵赋值，一共循环 64 次，只有最后一次不跳转，所以发生 8*8-1=63 次 BTS。

② Mul 部分：与 init 部分一样，每一重循环仅有最后一次预测正确，因此第一重循环发生 7 次 BTS，第二重循环发生 8*7=56 次 BTS，第三重循环发生 8*8*7=448 次 BTS。

③ 两部分一共发生 63+7+56+448=574 次 BTS。

对于启用 BTB 的情况：开始时发生 1 次 BTS，分支预测结果被写入 BTB 中；结束时发生 1 次 BMS，并删除 BTB 中的预测结果。综上所述，BTS 的次数=BMS 的次数。

① Init 部分：循环开始和结束时各发生 1 次 BTS 和 1 次 BMS。

② Mul 部分：第一重循环会被执行 1 次，发生 1 次 BTS 和 1 次 BMS；第二重循环会被执行 8 次，发生 8 次 BTS 和 8 次 BMS；第三重循环会被执行 64 次，发生 64 次 BTS 和 64 次 BMS。

③ 两部分的总次数=(1+1+8+64)*2=148 次。

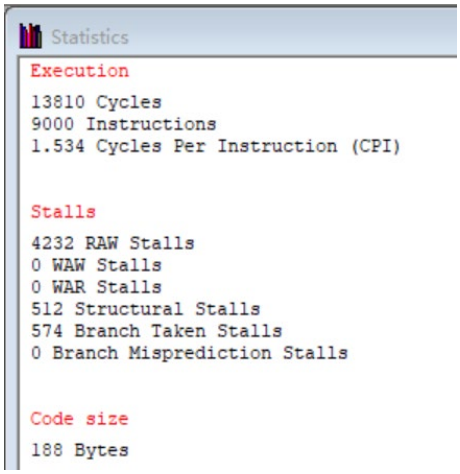


图 13a 关闭 BTB

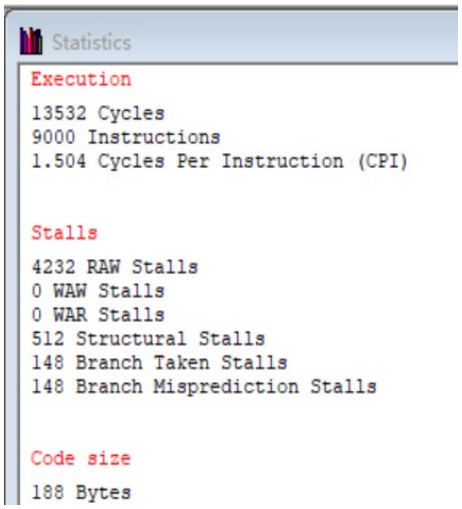


图 13b 开启 BTB

（二） 设计使 BTB 无效的代码

实验结果表明，开启 BTB 不一定能提高性能，还有可能导致程序性能下降，这与程序的具体实现有关。例如，第二部分的程序实现不开启 BTB 比开启 BTB 的性能强大一倍。

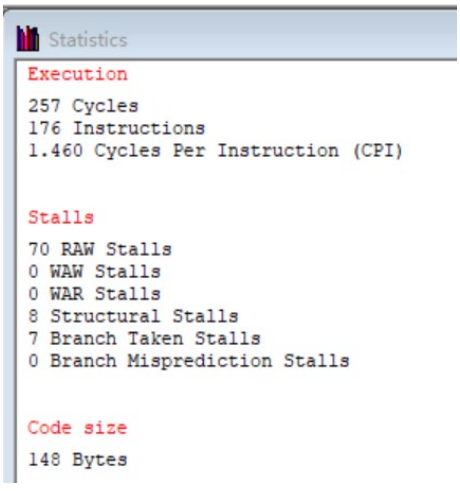


图 14a 关闭 BTB

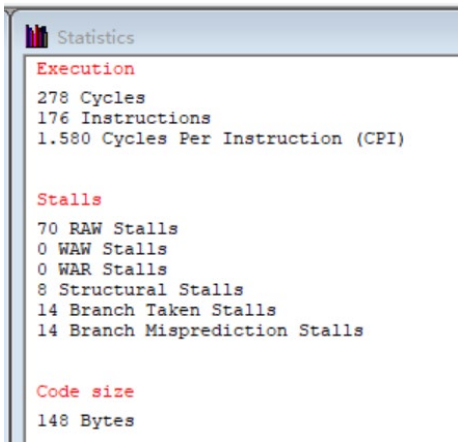


图 14b 开启 BTB

（三） 循环展开与 BTB 的效果比对

循环次数较多时开启 BTB 可以提高程序性能；但是循环重数过多时，越内层的循环发生的 BTS 次数越多，开启 BTB 会导致性能下降，此时可以将内层中循环次数较少的循环进行展开操作，减少分支预测错误造成的性能损失，提高程序性能。

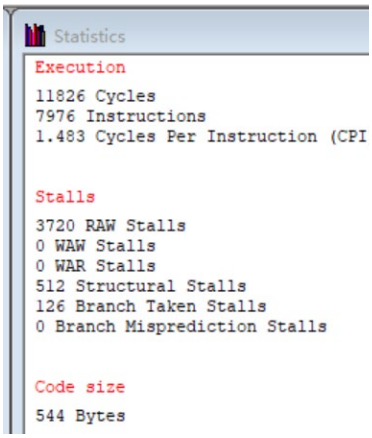


图 15a 循环展开（不启用 BTB）

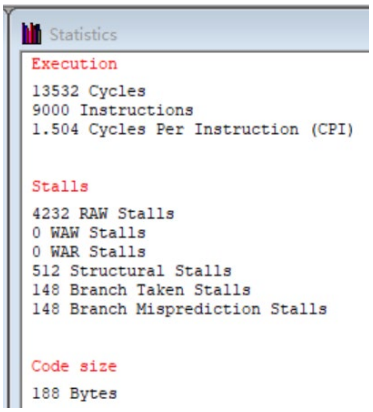


图 15b 未循环展开（启用 BTB）

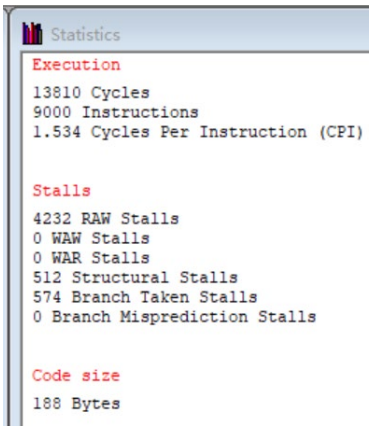


图 15c 未循环展开（不启用 BTB）

六、实验总结与体会

1. 通过本次实验，我对控制冒险、分支预测技术以及 BTB 技术有了更深入的理解。
2. 在进行实验的第一部分中，我明白了 BTB 是通过动态预测分支是否跳转成功从而对程序进行加速的。
3. 在进行实验的第二部分中，我了解到 BTB 不是总是能提高程序性能的，在某些情况下反而会导致程序性能急剧下降。
4. 在进行实验的第三部分中，我明白了 BTB 在循环次数较多时可以提高程序性能，在循环重数过多时会导致程序性能下降，解决办法为将内层循环进行循环展开操作。
5. 这次实验不仅加深了我对处理器结构和优化技术的理解，也提高了我的实践操作能力和问题解决能力。
6. 性能优化是一个复杂的过程，没有单一技术或者手段能适用于所有情况，需要根据程序的代码实现的特点选择合适的优化策略才能真正提高程序的性能。

指导教师批阅意见： **成绩评定：** 指导教师签字：王毅 2024 年 12 月 22 日
备注：

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。
2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。