计算机科学与技术学院计算机系统基础课程实验报告

|  |  |
| --- | --- |
| 实验题目：实验四 设计MIPS五级流水线  模拟器中的Cache | 学号：202000130198 |
| 班级： 20.4 | 姓名： 隋春雨 |
| Email：202000130198@mail.sdu.edu.cn | |
| 实验目的：  ①Cache 结构及功能的设计  ②了解指令流水线运行的过程  ③探究 Cache 对计算机性能的影响 | |
| 实验软件和硬件环境：  实验软件：ubuntu、Clion2020  硬件环境：多路处理器计算机教学实验系统  由四片四核龙芯3A处理器构成的16核CC-NUMA结构、内可配置  外可扩展结构的实验硬件平台。 | |
| 实验原理和方法：   1. **有关Cache的知识**     **Cache的出现是为了解决CPU越来越快、但是主存却没有跟上CPU的速度的矛盾的存在。**  Cache存储器：电脑中为高速缓冲存储器，是位于CPU和主存储器DRAM（Dynamic Random Access Memory）之间，规模较小，但速度很高的存储器，通常由SRAM（Static Random Access Memory静态存储器）组成。  CPU的速度远高于内存，当CPU直接从内存中存取数据时要等待一定时间周期，而Cache则可以保存CPU刚用过或循环使用的一部分数据，如果CPU需要再次使用该部分数据时可从Cache中直接调用，这样就避免了重复存取数据，减少了CPU的等待时间，因而提高了系统的效率。Cache又分为L1Cache（一级缓存）和L2Cache（二级缓存），L1Cache主要是集成在CPU内部，而L2Cache集成在主板上或是CPU上。  Cache的功能是提高CPU数据输入输出的速率。Cache容量小但速度快，内存速度较低但容量大，通过优化调度算法，系统的性能会大大改善，仿佛其存储系统容量与内存相当而访问速度近似Cache。   1. Cache设计原理     32位访存地址按位分为三个部分：标签(tag)、组索引(set index)、块位移(block offset)。我们定义每部 分的宽度分别为CACHE\_T、CACHE\_S、CACHE\_B（在cache.svh中定义）。另外还定义相联度为常量 CACHE\_E，其含义是每组包含CACHE\_E个缓存行(line)。 如果CACHE\_B为4且CACHE\_S为2，则每个cache行存储16个字节或4字，且整个cache分为4个组。   1. Cache工作原理     Cache的工作原理如上图所示。给定32位地址时，高速缓存首先读取该地址中的组索引(set index)字段 i ，然后检查第i组 set[i] 是否存在标记(tag)匹配的行(line)。如果存在则缓存命中(cache hit)，否则缓 存未命中(cache miss)，我们需要从内存中载入一个块(block)到缓存行中。 在现实世界中，一台32位计算机上的内存在一个时钟周期内最多可以提供32位数据。因此，若从内存中 读取数据块，则需要大量的时钟周期才能获取所有数据。 在高速缓存未命中的情况下，如果所选集合中充满了有效(valid)行，并且我们选择了要替换的脏(dirty) 行，则应首先将数据写回(writeback)到内存中，然后将新数据加载到该行中。选择要替换的行的方式有 多种，如最久未使用(LRU)、随机替换(RANDOM)、先进先出(FIFO)；   1. **实验方法** 2. 按照实验指导书要求，书写指令Cache和数据Cache的代码 3. 在pipe.c文件中将Cache文件include，对读写策略优化，目标是在数据读写时优先考虑Cache，再考虑主存。 4. 调用实验run程序，对比数值的变化。通过修改相关参数等以进一步探究不同因素对Cache速度的影响。 | |
| **实验步骤：**  **一、环境配置**  **（1）准备工作**  本实验已经提供了支持MIPS 指令集的流水线计时模拟器，通过这个模拟器你将可以更好的了解计算机是如何运行的。这台计算机主要包括两个部分：CPU 和内存。计算机要执行的程序（包括代码和数据）都存储在内存中，CPU 会将指令从内存中取出，进行解码并且执行代码所表示的操作（包括算术运算、逻辑运算以及存储器控制操作）。要注意的是，这里的CPU 所采用的指令集是MIPS 。   1. **理解书写**   **（1）阅读源码pipe.c, pipe.h, shell.c, shell.h**    **（2）按照要求书写Cache，明确替换策略**    ①dataCache ②instructionCache    其中实现的功能有初始化，写入，按地址从主存数据等。  ③策略：总是把最近最少用的那一块淘汰掉，即LRU（Least Recently Used）算法。当数据所占内存达到一定阈值时，要移除掉最近最少使用的数据。当所需数据在Cache中时，将该数据移到Cache的最前端，其他数据后移。当所需数据不在Cache中时，从内存中寻找到该数据，并将其插入Cache的最前端，其他数据后移。若此时有数据溢出，将最后端的元素写回主存（Cache中最近最小使用的），如果此位dirty位为1，说明在Cache中被修改，写回主存时需要对其进行相应的修改（因为此时已经和主存中的不一致了）。当分块局部化范围(即：某段时间集中访问的存储区)超过了Cache存储容量时，命中率变得很低。极端情况下，假设地址流是1,2,3,4,1 2,3,4,1,……，而Cache每组只有3行，那么，不管是FIFO，还是LRU算法，其命中率都为0。这种现象称为颠簸(Thrashing / PingPong)。LRU具体实现时，并不是通过移动块来实现的，而是通过给每个cache行设定一个计数器，根据计数值来记录这些主存块的使用情况。这个计数值称为LRU位。   1. 加入延时，运行程序，观察性能。（部分截图如下）   把 cache 的逻辑嵌入到现提供的 MIPS 模拟器中即可，模拟器可以正确处理流水线中数据依赖问题，可以通过增加气泡的方式绕过并正确处理他们。我们的目标是让总的 cycle 尽可能的少，发挥 cache 应有的作用。这里用自检查批处理脚本，以检查编写的正确性。  WQ~ZWA8IAJ2`NIL$JT@9$1A   1. **修改 cache 的各项参数来探究 cache 对模拟器性能的影响**   可以改变的参数有sets, block size, ways等，可以总结为Cache容量越大，命中率越高，Cycles越小；Cache的block在一定范围内越大命中率越高，若超出这个范围则恰恰相反。  WQ~ZWA8IAJ2`NIL$JT@9$1A  **①减小cache组数（set）**  结果：在实验所给数据中cycle数无变化，可能是因为所给数据的局限，并没有符合预想中的miss率上升，cycle数增加。  **②减小block size**  结果：每行存储的数据减小，运行减慢。  **③减小ways**  每组2或4行（称为2-路或4-路组相联）较常用。通常每组4行以上很少用。在较大容量的L2 Cache和L3 Cache中使用4-路以上。  结果：绝大部分测试点无变化，仅有一两个测试点缓慢减小的倍数。     1. **改变替换策略——替换(Replacement)算法**   **①随机替换算法（Random）**  　　随机法是随机地确定替换的存储块。设置一个随机数产生器，依据所产生的随机数，确定替换块。这种方法简单、易于实现，但命中率比较低。  结果：顾名思义，弹出的数据是随机的，结果竟然也是“随机”的，随机算法有的测试点比LRU快，有的测试点比LRU慢。  **②先进先出FIFO （first-in-first-out）**    这里是采取队列结构，最先进入Cache的元素将会从Cache中弹出，根据dirty位传回主存。  结果：绝大部分测试点无变化，仅有一两个测试点比LRU慢  ③最近最少用LRU （ least-recently used）  即一开始选用的方法 | |
| 结论分析与体会：  1、通过实验理解了五级流水线  流水线是将计算机指令处理过程拆分为多个步骤，并通过多个硬件处理单元并行执行来加快执行速度。在三级流水“取指-译码-执行”的基础上，引入Cache，对读取速度进一步提高；此外，因为Load/Store指令很有可能会超过一个时钟周期，所以，我们加了访存和回写，用五级流水线的方式来解决。   * 取指令(IF)：根据PC的值从存储器取出指令。 * 指令译码(ID)：产生指令执行所需的控制信号。 * 取操作数(OF)：读取存储器操作数或寄存器操作数。 * 执行(EX)：对操作数完成指定操作。 * 写回(WB)：将操作结果写入存储器或寄存器。      1. 通过实验进一步理解了Cache的相关知识   前提：大量典型程序的运行情况分析结果表明在较短时间间隔内，程序产生的地址往往集中在一个很小范围内，这种现象称为程序访问的局部性：空间局部性、时间局部性。   * 指令：指令按序存放，地址连续，循环程序段或子程序段重复执行 * 数据：连续存放，数组元素重复、按序访问   在CPU和主存之间设置一个快速小容量的存储器Cache，其中总是存放最活跃（被频繁访问）的程序和数据，由于程序访问的局部性特征，大多数情况下，CPU能直接从这个高速缓存中取得指令和数据，而不必访问主存，如果没有找到则再去主存中寻找，并根据所选策略对Cache进行更新等操作。如果没有Cache，就需要到主存中寻找，耗费较多时间。    3、解决了配置运行过程中的系列问题：  ①要注意Python版本，且须配置环境变量和工具包，否则程序无法运行  ②go之后一直在跑，就是不halted  这时候可能是从Cache中取指有误，debug一下，打印Cache中的返回值与memread的返回值对比一下。  #ifdef DEBUG // 仅在Debug时被编译  ③递归栈过深  3I56X$MQ{NWYO2T%NIZ32OL  使用python写的递归程序如果递归太深, 那么极有可能因为超过系统默认的递归深度限制而出现的错误。这是需要人为设定。  4、在Cache的设计中体会到多级结构设计的重要性  使用了结构体设计多级结构，Cache由set组成，set由line组成，line由valid,tag,data组成……这样子定义结构体层次清晰直观，在对参数和替换策略修改的时候也比较方便。  5、/\* each of these functions implements one stage of the pipeline 以下各个函数分别实现流水线的各个阶段 \*/  void pipe\_stage\_fetch();  void pipe\_stage\_decode();  void pipe\_stage\_execute();  void pipe\_stage\_mem();  void pipe\_stage\_wb();  但为什么代码要“倒着写”？  pipe\_stage\_wb(); // 回写  pipe\_stage\_mem(); // 访存  pipe\_stage\_execute(); // 执行  pipe\_stage\_decode(); // 译码  pipe\_stage\_fetch(); // 取指    由上图中的cycle5可以清晰的看出，在这个cycle内其实是按着第一步的回写，第二步的访存，……这样子执行的。可以说是阶段上倒置，指令上正置。  附录：data\_Cache.h   1. #ifndef DATA\_CACHE 2. #define DATA\_CACHE 3. #include "stdio.h" 4. #include "mips.h" 5. #include "stdbool.h" 6. #include "stdint.h" 7. typedef struct {    *// cache中的每一行* 8. uint8\_t valid;    *//是否已存入数据* 9. uint8\_t dirty; 10. uint8\_t lru;    *// 记录存入时间 太大影响运行时间* 11. uint32\_t line;  *// 行号* 12. uint8\_t data[32];   *// 32字节* 13. } data\_cache\_line; 14. typedef struct {    *// 数据cache* 15. data\_cache\_line data\_cache\_set[256][8]; *// 256组，每组8行* 16. } data\_cache; 17. void data\_cache\_init(); *// 初始化* 18. void mem\_write\_data\_cache(uint32\_t address, uint32\_t value);    *// 修改数据* 19. void data\_cache\_load(uint32\_t address); *// 将数据写入cache* 20. uint32\_t data\_cache\_read(uint32\_t address); *// 从cache中读取数据* 21. extern int data\_pause; 22. #endif   Data\_cache.c   1. #include "mips.h" 2. #include "pipe.h" 3. #include "shell.h" 4. #include <assert.h> 5. #include <stdio.h> 6. #include <stdlib.h> 7. #include <string.h> 8. #include "data\_cache.h" 9. data\_cache cache; 10. int data\_pause = 0; 11. *// 初始化* 12. void data\_cache\_init() { 13. int i, j; 14. for (i = 0; i < 256; i++) { 15. for (j = 0; j < 8; j++) { 16. cache.data\_cache\_set[i][j].valid = 0; 17. cache.data\_cache\_set[i][j].lru = 0; 18. cache.data\_cache\_set[i][j].dirty = 0; 19. } 20. } 21. } 22. *// 从cache中读取数据* 23. uint32\_t data\_cache\_read(uint32\_t address) { 24. uint32\_t line = address >> 13; 25. uint32\_t set = (address >> 5) & (0xFF); *// 组号* 26. uint32\_t inneraddress = address & 0x1F; 28. *// printf("data\_cache\_address: %x", address);* 29. int i = 0; 30. for (i = 0; i < 8; i++) { 31. if (cache.data\_cache\_set[set][i].valid == 1 && 32. cache.data\_cache\_set[set][i].line == line) {   *//当前行号和对应的主存行号相同* 33. uint32\_t tmp = address & 0x1F; *//内部地址* 34. *// uint32\_t a = (cache.data\_cache\_set[set][i].data[tmp] << 0) |* 35. *// (cache.data\_cache\_set[set][i].data[tmp+1] << 8) |* 36. *// (cache.data\_cache\_set[set][i].data[tmp+2] << 16) |* 37. *// (cache.data\_cache\_set[set][i].data[tmp+3] << 24);* 38. *// printf("res: %x", a);* 39. return 40. (cache.data\_cache\_set[set][i].data[tmp] << 0) | 41. (cache.data\_cache\_set[set][i].data[tmp+1] << 8) | 42. (cache.data\_cache\_set[set][i].data[tmp+2] << 16) | 43. (cache.data\_cache\_set[set][i].data[tmp+3] << 24); 44. } 45. } 46. data\_pause = 49;    *// 停顿50个周期* 47. data\_cache\_load(address);   *// 写入data\_cache* 48. return data\_cache\_read(address);    *// 再次读取并返回* 49. } 50. *// 修改数据* 51. void mem\_write\_data\_cache(uint32\_t address, uint32\_t value) {   *//把本应写在主存address里的write写到cache里* 53. data\_cache\_read(address);   *// 保证修改数据在cache中* 54. uint32\_t line = address >> 13;          *//取前19位* 55. uint32\_t set = (address >> 5) & (0xFF); *//取所对应的组号* 56. *// printf("mem\_address: %x", address);* 57. int i; 58. for (i = 0; i < 8; i++) { 59. if (cache.data\_cache\_set[set][i].line == line) {   *// 找到对应的主存块* 61. uint32\_t tmp = address & 0x1F; 62. cache.data\_cache\_set[set][i].dirty = 1; 63. cache.data\_cache\_set[set][i].data[tmp] = (uint8\_t)(value >> 0) & 0xFF; 64. cache.data\_cache\_set[set][i].data[tmp+1] = (uint8\_t)(value >> 8) & 0xFF; 65. cache.data\_cache\_set[set][i].data[tmp+2] = (uint8\_t)(value >> 16) & 0xFF; 66. cache.data\_cache\_set[set][i].data[tmp+3] = (uint8\_t)(value >> 24) & 0xFF; 68. return; 69. } 70. } 71. } 72. *// 将数据写入cache中* 73. void data\_cache\_load(uint32\_t address) { 74. uint32\_t line = address >> 13;          *//取前19位，要写入的line* 75. uint32\_t set = (address >> 5) & (0xFF); *//取所对应的组号* 76. uint32\_t begin = (address >> 5) << 5;        *//块头* 77. int i = 0, pos = -1; 78. for (i = 0; i < 8; i++) { 79. if (cache.data\_cache\_set[set][i].valid == 0) { 80. cache.data\_cache\_set[set][i].valid = 1; 81. cache.data\_cache\_set[set][i].line = line; 82. cache.data\_cache\_set[set][i].lru = 0; 84. pos = i; 85. break; 86. } 87. } 89. *// lru   cycle* 90. *// 2、3  925848* 91. *//  4    926721* 92. *//  5    926527* 93. *//  6    926430* 94. *//  7    926333* 95. for (i = 0; i < 8; i++) {   *// 更新lru* 96. if (pos != i && cache.data\_cache\_set[set][i].lru < 7) { 97. cache.data\_cache\_set[set][i].lru++; 98. *// printf("更新lru");* 99. *// printf("lru%d: %x %x", i, cache.data\_cache\_set[set][i].lru, cache.data\_cache\_set[set][i].dirty);* 100. } 101. *// printf("\n");* 102. } 103. *// printf("%d\n", pos);* 104. if (pos == -1) { 105. uint8\_t max = 0; 106. *// printf("---------------max-------------------\n");* 107. for (i = 0; i < 8; i++) { 108. if (max <= cache.data\_cache\_set[set][i].lru) {   *// 找到替换行* 109. max = cache.data\_cache\_set[set][i].lru; 110. pos = i; 111. printf("%d %d %d\n", max, pos, i); 112. } 113. } 114. *// printf("--------------max---------------------\n");* 115. *// printf("-----------------------%d-------------------------\n", pos);* 116. if (cache.data\_cache\_set[set][pos].dirty != 0) { *// 将脏数据写回* 117. *// printf("写回脏数据\n");* 118. uint32\_t tmp = (cache.data\_cache\_set[set][pos].line << 13) | (set << 5);    *// 替换位置* 119. for (i = 0; i < 8; i++) { 120. mem\_write\_32(tmp, data\_cache\_read(tmp)); 121. tmp += 4; 122. } 123. } 124. cache.data\_cache\_set[set][pos].valid = 1; 125. cache.data\_cache\_set[set][pos].lru = 0; 126. cache.data\_cache\_set[set][pos].line = line; 127. } 129. for (i = 0; i < 8; i++) {   *// 写入cache* 130. uint32\_t tmp = mem\_read\_32(begin); 131. cache.data\_cache\_set[set][pos].data[i\*4] = (uint8\_t)(tmp >> 0) & 0xFF; 132. cache.data\_cache\_set[set][pos].data[i\*4+1] = (uint8\_t)(tmp >> 8) & 0xFF; 133. cache.data\_cache\_set[set][pos].data[i\*4+2] = (uint8\_t)(tmp >> 16) & 0xFF; 134. cache.data\_cache\_set[set][pos].data[i\*4+3] = (uint8\_t)(tmp >> 24) & 0xFF; 135. begin += 4; 136. } 137. } | |