山东大学 软件 学院

计算机系统原理 课程实验报告

学号：	姓名：	班级：AI17
实验题目：Cache 仿真模拟		
实验学时：32	实验日期：2019.04.28	
实验目的： 理解 cache 的工作原理，以及相应参数与算法对程序性能的影响		
硬件环境：PC、笔记本电脑		
软件环境：Linux 系统下 gem5 模拟器		

实验步骤与内容：

1 实验步骤

1.1 搭建环境

在 Linux 操作系统下编译 gem5 相关文件。

1.2 获得 trace

编写矩阵乘源码，编译成可执行文件，在 gem5 模拟运行，获得 trace 文件。通过设置不同的 debug-flag，得到不同的 trace，本次实验主要和访存相关，flag 设置成 MMU、MemoryAccess 和 DRAM。

MMU 是虚拟地址和物理地址的映射相关硬件，当 CPU 访问内存时需要完成地址的转换，把虚拟地址映射到物理地址。使用 MMU 标志所获得的 trace 文件记录这一转换过程，其文件的基本格式：

19632000: system.cpu.workload: Translating: 0x8a350->0x7b350

即：

时钟数：相关模块标识：操作：Vaddr->Paddr

设置 DRAM 所获得的 trace 文件记录访问内存的信息，文件内部基本格式是：

5041500: system.mem_ctrls: recvAtomic: ReadReq 0x28188

即：

时钟数：相关模块标识：操作：ReadReq/WriteReq address

需要引起注意的是，这里的地址究竟是访问内存的物理地址还是从 MMU 转换出的物理地址，这两个物理地址是不同的，CPU 访问内存是给出虚拟地址，由 MMU 将其转换成与其对应的物理地址，利用这个物理地址访问 Cache，若 Cache 没有命中，才访问内存，并把这个物理地址相邻的一块空间存入 Cache 中。CPU 给出

的虚拟地址结果 MMU 转换后的物理地址通常只是用于访问一个字大小的数据单元，访问内存通常是以 Cache line 为单位对数据块访问，这两个地址显然不同。但是通过比较 MMU 标志的 trace 文件和 DRAM 标志得到的 trace 文件，发现他们的物理地址极其相似，甚至和 MemoryAccess 标志得到的 trace 文件内部访存信息也极其类似，这个观察让人十分惊讶。这意味这三个 trace 文件都包含以字节为单位访问物理内存的地址，这个发现对之后的处理会带来很多便利。

设置 MemoryAccess 标志所获得的 trace 文件记录了关于访存更加详细的信息，这个文件区分指令访存和数据访存，并且还有读入的数据大小等有用信息。我们研究 Cache 命中率时，所关注的只是 CPU 访问内存的物理地址，关于 MemoryAccess 的 trace 文件里面的信息似乎过于冗余，尤其是在 trace 开头关于程序加载的部分。

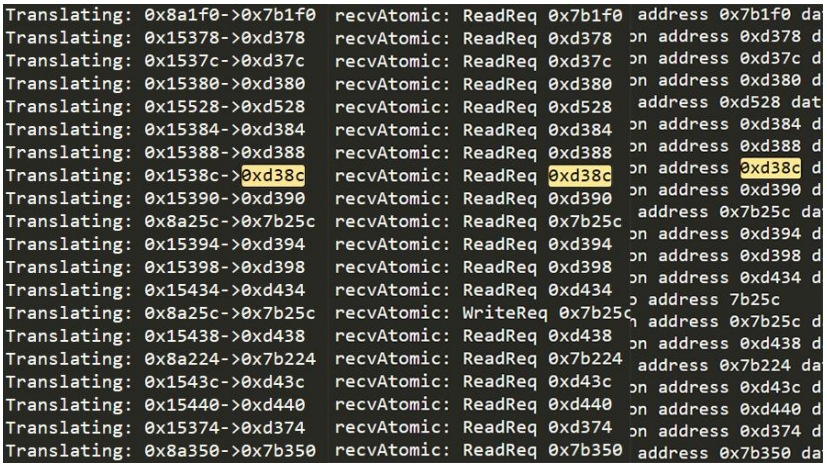

图 1：trace 文件内部地址关系

从左到右，依次是 MMU、DRAM、MemoryAccess 所获得的 trace 文件内容，

通过比较三个标志得出的 trace 文件，与 0xd38c 相邻的访存信息完全一致，其它地方也与之类似。

1.3 压缩 trace

实验表明，一个中等规模的矩阵乘程序所获得的 trace 文件已经大到不可接受，有必要对其进行压缩处理。gem5 官网提出的一个压缩方法是，将 trace 文件的名字改成*.gz 格式，这样产生的 trace 文件是压缩后的文件，而且这个压缩比通常非常高（比如：90%）。这个方法对我们来说，没有解决根源问题，我们处理的是解压后的 trace 文件。

另外一种解决办法是，消除 trace 文件的冗余信息。之前提到，探究 Cache 命中率时，对我们有用信息只是访问内存的物理地址。trace 文件有太多的冗余信息，尤其是 MemoryAccess 标志所获得的 trace。实际上，DRAM 标志所获得的 trace 含有的信息最为简洁，处理 DRAM 获得的 trace 通常是我们的首选。但是仍然无法满足我们的要求，像时钟数，模块标识等冗余信息依然存在。下一步通过修改 gem5 源码（阅读源码，并修改源码是一个极其艰难的工作，但是使用一些工具可以快速定位相关文件），减少冗余信息的产生。在 gem5 源码中找出有关 trace 信息打印的一些函数，修改 trace.hh,dram_ctrl.hh,abstract_mem.cc 等相关文件后，得到的 trace 文件内容如图 2 所示。

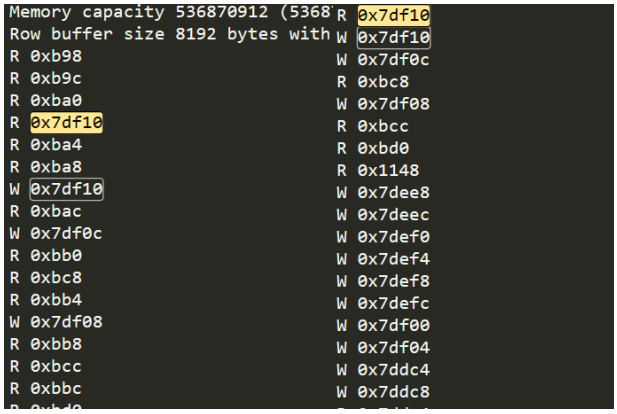

图 2：压缩后 trace 文件内容

消除冗余信息之后 trace 文件内容，左侧是关于 DRAM 的 trace，右侧是关于 MemoryAccess 的 trace

经过处理后，trace 文件只保留了有用信息，时钟数、模块标识等无用信息被删去，保留物理地址和读写标识。DRAM 标识所得的 trace 文件包含指令和数据的访存，将 MemoryAccess 的 trace 文件有关指令访存信息过滤掉之后，所得的 trace 文件经过处理后只得到关于数据的访存信息，由于不考虑 MMU 虚拟地址和物理地址的映射关系，MMU 所得的 trace 文件不做考虑。

经过压缩处理之后的 trace 简洁性大大增强。为之后的处理带来极大便利。

1.4 编写 Cache 仿真模拟器

本次 Cache 仿真模拟器重在分析不同替换策略、不同映射关系、Cache 内部参数对 Cache 性能的影响。为了简化 Cache 模拟器的实现，在编写过程中忽略 Cache 的内部细节，比如不区分 Cache 地址译码时间和数据传输时间，而将两者合并为访问 Cache 的总时间。关于具体仿真器将在之后介绍。

1.5 分析 Cache 策略对 Cache 的影响

用写好的 Cache 模拟器运行不同的 trace 文件，并分析 Cache 替换策略、Cache 行映射关系、Cache 内部参数对 Cache 的影响。数组的访问可以非常简便地模拟对物理内部数据块的访问，所以这里测试 Cache 性能使用的是关于数组访问的 trace（因为数组 trace 有问题，改用矩阵乘 trace）。设置数组访问的步长和数组的大小，模拟不同情形下对内存的访问方式。

1.6 探究不同版本矩阵乘性能的优劣

在我们已学习的知识范围内，关于矩阵乘应该有 9 种版本，分别是 ijk，ikj，jik，jki，kij，kji，分块乘，转置乘和分块转置乘。不同方式的矩阵乘区别在于访问矩阵的顺序不同，通过提高 Cache 命中率来减少访存次数，从而提高整体的效率。

2 Cache 仿真器

2.1 程序框架

Cache 分为以下几种类型：

按照映射方式，可分为直接映射、全相联映射、组相联映射。

按照替换策略，可分为随机替换策略、最不经常使用策略、近期最少使用策略、先到先被替换策略。

按照写方式，可分为写直达方式和写回方式。

这几种方式组合而形成的Cache类型多达十几种，为每一种Cache写一个类，不太现实。本程序采用继承机制设计四个类，分别是 Cache 基类、DMCache（Directed Mapping Cache）类、FAMCache(Fully Associated Mapping Cache)类和 GAMCache（Group Associated Mappint Cache）类，后面三个类共同继承自 Cache 基类。关于替换策略和写策略使用不同的函数来实现。具体类定义如图3所示。

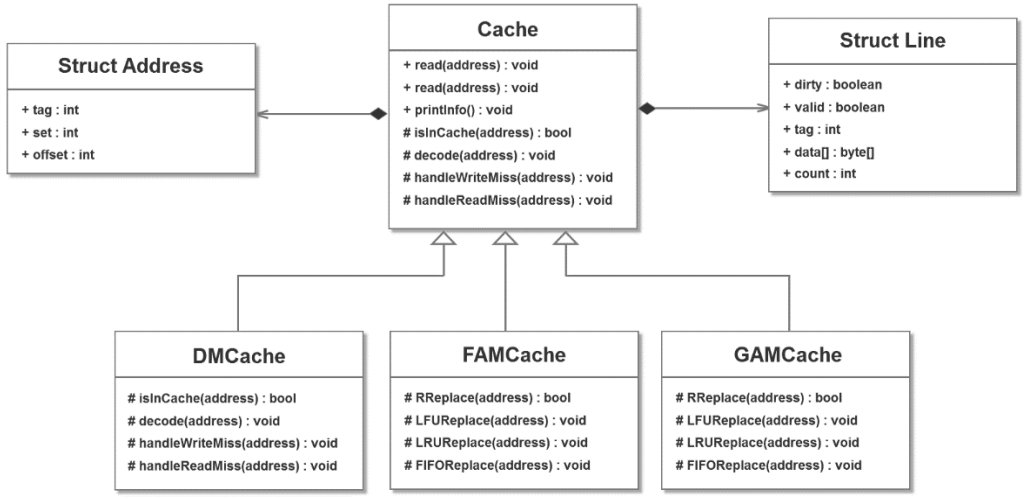

图 3：Cache 相关类 UML 图

Cache 最主要的操作是读写操作，Cache 定义了诸如 isInCache 的虚函数，有其不同子类具体实现。通过设置 replaceStrategy 和 writeStrategy 控制 Cache 使用的策略。

2.2 具体细节

Cache 内部有许多参数需要设置，这些参数由使用者通过构造函数传递给 Cache 类。一些重要的参数有 Cache 的容量、字大小、Cache Line 大小、访问 Cache 的时间、访问内存的时间等，如果使用组映射 Cache 还需要设置组大小。描述 Cache 性能的参数有命中率、吞吐量、平均读写时间、加速比等。

Cache 内部的方法实际上就是接收输入的地址，计算内部参数的过程。read

和 write 方法中，根据给定的地址，首先对地址进行译码，地址译码实际上就是对地址按照字节内偏移字段、set 字段、tag 字段进行划分，并把处理之后的各个字段封装在 Address 结构体内。关于地址的划分不同映射关系的 Cache 有不同的划分方式，承担划分地址的 decode 函数由不同的 Cache 各自实现。得到译码的地址后,判断这个地址是否在 Cache 中,判断地址是否在 Cache 中由 isInCache 函数实现，这个函数返回一个 boolean 值，如果在 Cache 内则返回 true，否则返回 false。判断地址是否在 Cache 中不同 Cache 有不同的判断方法，在全相联 Cache 中,需要将地址的 tag 字段的值和 Cache 内部所有的地址的 tag 进行比较；直接映射中需要根据 set 字段的值，和特定位置的地址中的 tag 值进行比较；组相联 Cache 中，需要到指定组和组内的 Cache Line 的所有 tag 比较一边。假设关于 tag 的比较有硬件完成，并且各个 tag 之间可以并行比较，这些不同映射方式的 Cache 在比较 tag 是均具有相同的时间，并且把这个时间统一算在访问 Cache 的总时间上。经过比较后，如果在 Cache 内，则命中，更新记录命中的计数值，并且访问数据，访问数据的具体细节在程序中没有实现，只是简单把访问一次 Cache 的时间加到读或写的总时间上。如果没有命中，则调用 handleMiss 函数，对缺失进行处理，并且更新相关的计数值和统计读写时间的变量。

在处理缺失时，需要访问内存，并把相关数据移入 Cache 中，在本程序的实现过程中,忽略内存和 Cache 的传输细节,只是简单更新统计总访存时间的变量。在把内存数据放入到 Cache 中时，需要解决碰撞。

在直接映射中，如果地址所对应的 set 内已经存在数据则发生碰撞，需要进行替换，对于直接映射来说，替换策略只有一种，即替换指定的 set 内的 Cache Line。

在全相联映射中,如果 Cache 内部的每一个 Cache Line 的 valid 均为 true，则发生碰撞，解决碰撞的策略有随机替换、LFU 替换、LRU 替换和 FIFO 替换。随机替换中，通过抛出一个随机值确定替换的位置。LFU 策略需要为每一个 Cache Line 安排一个计数值，访问一次这个 Cache Line 中的数据，这个计数值增 1，发生冲突时替换计数值最小的那个 Cache Line，我们假设这个计数值在 Cache Line 中由专门的区域存储，并且有相应的硬件实现自增功能，所以计数器自增的操作也不额外记录在访存时间上。但是寻找最小的计数值假定需要 CPU 参与，对每一个 Cache Line 的计数值比较的时间为 CPU 一个周期。LRU 策略和 LFU 策略类似需要为每一个 Cache Line 设立一个计数值,只不过当访问某一 Cache Line 时，只有其它的计数值增 1。这个计数值反映的这个 Cache Line 的年龄，刚刚新进入 Cache 的 Cache Line 的计数值为 0，在 Cache 内部呆地越久并且始终没有访问,则这个 Cache Line 计数值越大,替换时只替换计数值最大的 Cache Line。寻找最大计数值的过程也需要 CPU 开销，但是自增由硬件实现。FIFO 替换策略中，需要额外记录一个指针，这个指针记录上次加载 Cache Line 的位置，下次加载的位置位于这个之后，更新指针需要 CPU 参与，占用额外的时间。

组相联映射是全相联和直接映射的折中，全局使用直接映射，组内使用全相联映射。它的替换策略和什么替换过程类似，只是在小范围内比较，在组内替换。

写策略有两种，写直达方式和写回方式。不同的写策略在替换策略的实现和读操作上有些许不同，写直达方式中，对 Cache 中的数据进行写操作时，需要立即同步内存和 Cache 中的数据,每次写操作不管是否命中都需要比写回策略多一个访问内存的时间消耗。在替换时，写回策略需要判断 Cache Line 的 dirty 位，而写直达方式无需关心 dirty 位，如果这个 dirty 位置为 1，替换时需要多增加

一次访问内存的时间。

关于 Cache 具体实现请见所附源码。这里不再赘述。

3 Cache 性能分析

3.1 映射关系对 Cache 性能的影响

探究映射关系对Cache性能的影响所使用的trace是不同步长访问数组所得到的 trace，不知是何原因所得到的数据结果和期望相差很大。不管如何设置步长，Cache 的命中率居高不下，步长成倍变化，但是 Cache 的命中率却只是变化几个百分点。这给我们的分析带来非常大的困难，之后不得不通过手动生成地址以得到理想情况的数据。

第一个实验是探究访问位置对不同映射关系下 Cache 命中率的影响。生成一个地址序列，每个地址相差 stride 个存储单元。stride 的值大于 Cache Line 的大小，每次访问的地址均位于不同的 Cache Line 中。Cache 的参数如下。

表 1：Cache 参数设置

	Cache 大小	Cache Line 大小	字大小	行数	路数
全相联	256B	16B	4B	16 行	\
直接映射	256B	16B	4B	16 行	\
组相联	256B	16B	4B	16 行	4

统一采用 FIFO 替换、写回策略。循环访问间距为 stride 的元素一千次，得到表 2 所示的 Cache 的命中率(HR)和空间利用率(SU)与 stride 的关系。

表 2：不同 stride 访问内存时 Cache 的命中率

Array		GAM		DM		FAM	
Size	**Stride**	**HR(%)**	**SU(%)**	**HR(%)**	**SU(%)**	**HR(%)**	**SU(%)**
512	**4**	0	100	0	100	0	100
512	**8**	0	50	0	50	98.4	100
512	**16**	0	25	0	25	99.2	50
512	**32**	99.6	25	0	12.5	99.6	25
512	**64**	99.8	12.5	0	6.25	99.8	12.5
512	**128**	99.9	6.25	99.9	6.25	99.9	6.26

数组的大小为 512× 4B，访问数据的间距为 stride。Cache 的命中率只有两种取值，要么为 99%，要么为 0。Cache 命中率取 99%说明访问的元素可以同时存放在 Cache 中，命中率取值为 0 说明访问的元素在 Cache 中被映射到同一物理空间。当 stride 为 4 时，Cache 的命中率为 0，这是由于访问元素的个数已经超过 Cache 所能存放元素的最大值，当访问上一轮访问过的元素时，它已经被替换掉。逐步增大 Stride 时，Cache 可以装满需要访问的元素，由于映射策略不同，在

stride 等于 16 时，即使 Cache 有剩余空间可以放置需要访问的元素，但是由于 Cache 把需要访问的元素映射到同一物理空间，需要访问的元素不停在内存和 Cache 间替换，命中率降低。

综合来看，全相联映射具有最好的空间利用率，而直接映射的空间利用率最差，最坏的利用率仅为 6.25%。组相联映射介于两者之间。映射策略不同造成空间利用率相差很大，全相联映射方式具有最强的灵活性，每一条 Cache Line 和其它 Cache Line 均可以同时存在 Cache 中。直接映射方式，被映射到同一组的 Cache Line 互不相容。组相联映射，被映射到同一组的 Cache Line 可以共存一定数目，组与组之间没有冲突。全相联虽然具有较好的空间利用率，但是需要额外的硬件支持，将全相联和直接映射相结合的组相联方式硬件开销没有全相联那么大，而且具有较好的空间利用率。

3.2 替换策略对 Cache 性能的影响

已知的替换策略有四种：随机替换、最不经常使用、最近不常使用和 FIFO 替换策略。对于直接映射方式来说，每个 Cache Line 所在的位置是固定的，不存在替换策略一说。替换策略主要存在于全相联和组相联。

测试替换策略所使用的 trace 是 ijk 版本的矩阵乘，使用设置 DRAM 标识的 trace 文件，这个文件物理访问既包含数据访存又包含指令访存。写策略使用写回方式以突出替换的代价。使用 Cache 模拟器测试之后得到如下如下结果。

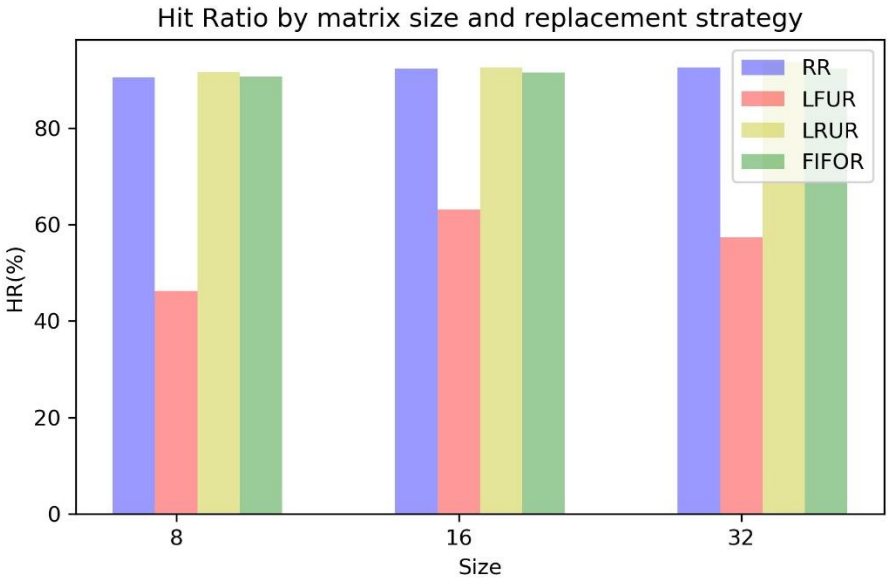

图 4：不同替换策略下 GAMCache 的命中率比较

图 4 是组相联 Cache 下的命中率和替换策略的比较。横坐标是矩阵乘所用方阵的大小。可以很明显地看出，最不经常使用替换策略大大降低了 Cache 的命中率，其它策略的命中率几乎相当。最不经常替换很可能把刚刚进入 Cache 的行给替换掉，所以具有较低的命中率。在全相联中亦是如此，如图 5 所示。

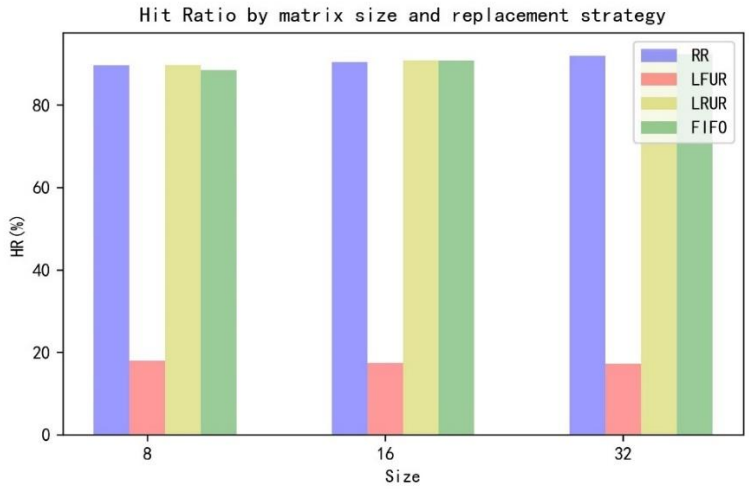

图 5：不同替换策略对 FAMCache 命中率的影响

替换策略对于全相联 Cache 命中率和组相联 Cache 有些许不同，LFU 策略对全相联 Cache 的影响更大。无论对于组相联 Cache 还是对于全相联 Cache，随机替换、邻近最少使用和 FIFO 替换具有近乎相同的命中率。关于其它三个策略仍需要比较其他参数。

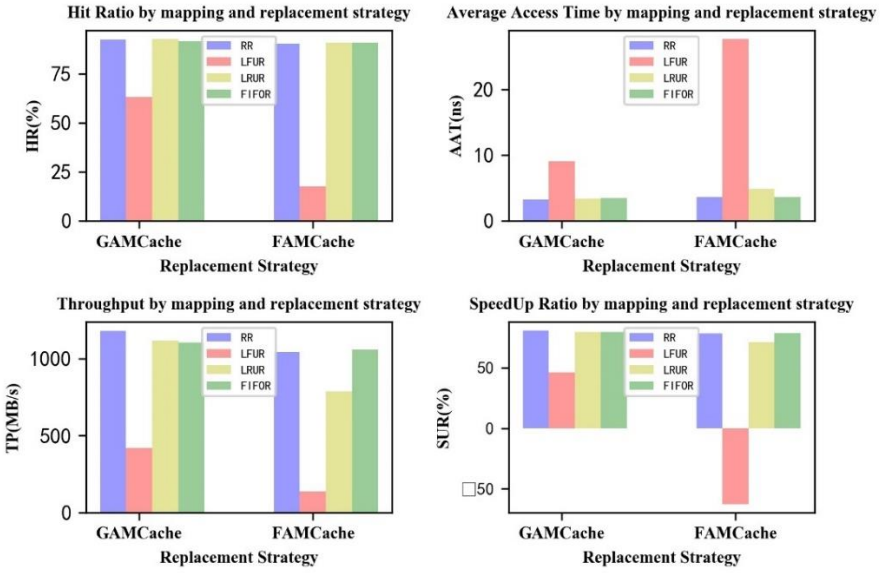

图 6：不同替换策略下 Cache 的性能指标

图 6 是在 16×16ijk 矩阵乘 trace 下其它性能参数的比较，左上方是命中率的指标，之前已经讨论过。右上方是平均访问时间的指标，左下方是吞吐量的指标，右下方是加速比的指标。这几个指标并不是相互独立，高命中率、低访问时间的 Cache 吞吐量自然高。通常只需观察命中率和访问时间指标就足够了。LFU 替换策略具有低命中率、高访问时间、低吞吐量、低加速比的特点，无论在那一方面都不如其他的替换方案。其它三种方案对不同的映射方式有不同的影响。在组相联 Cache 中其它三种方式无论在那一指标下效果都基本均衡，但是在全相联 Cache 下三种策略有略微差别。LRU 略逊于另外两个方案，LRU 策略在替换时由

于需要比较每一个 Cache Line 的计数值，所以平均访问时间比较长。但是对于组相联 Cache 来说，LRU 只需要比较组内各个 Cache Line 的计数值，所以它的平均访问时间明显比全相联的时间少。以上的分析是基于 ijk 矩阵乘，不具有一般性，在一些环境下结果可能会与上面的分析具有较大的差异。总之，对于 ijk 矩阵乘，LFU 是最影响 Cache 的策略，其它三种策略对 Cache 具有几乎相同的影响，LRU 需要额外的硬件支持，它不如随机替换和 FIFO 策略。

3.3 写策略对 Cache 性能的影响

目前已知的两种简单写策略是写直达和写回策略，当写命中时，写直达要求同时关系内存和 Cache，写回策略要求在发生替换时写回内存。使用 16×16ijk 矩阵乘、LFU 替换策略，测得实验结果如下。

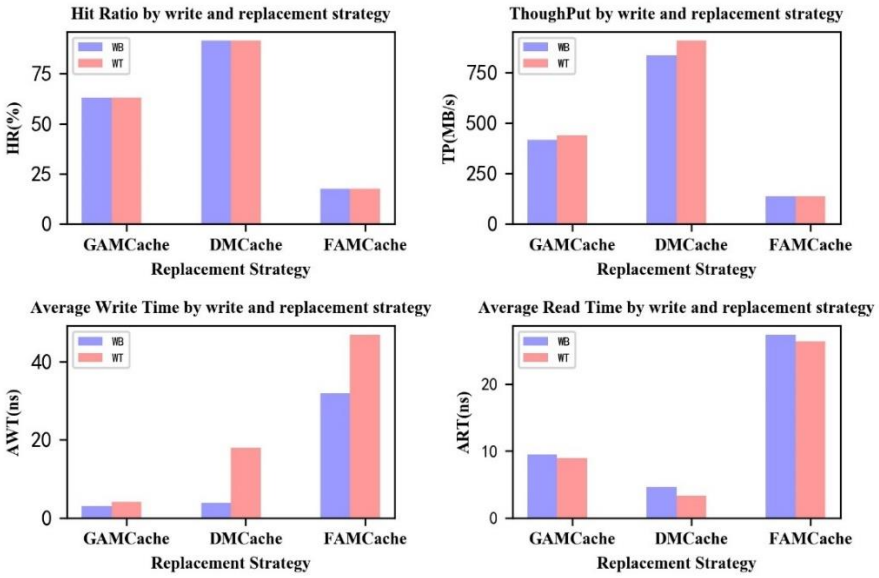

图 7：不同写和替换策略下 Cache 性能指标

图 7 重点在于比较不同写策略对 Cache 性能的影响，将三种映射策略同时画在图内是为了说明写策略对性能的影响与映射关系无关。图 7 左上角可以看出写策略对 Cache 的命中率没有影响，写策略对 Cache 的影响主要体现在访问 Cache 的时间，直写方案中，写入操作不管是否命中都需要访问一次内存，但是当读操作时，发生碰撞后直接替换，无需更新内存的值，相反在写回策略中，写命中不需要访问内存，替换时需要注意 Cache Line 的 dirty 位，如果 dirty 值为 1 需要更新内存，所以直写方案比写回方案在写操作所用时间长，读操作所用时间短。如图 7 的下方两幅图，实验结果与我们的分析相符，直接映射替换的次数比较多，写策略对 Cache 的平均写时间的影响程度大于另外两种映射策略。

3.4 Cache 内部参数对 Cache 性能的影响

Cache 最重要的一个参数是 Cache 的容量，它的大小和命中率密切相关，首先讨论 Cache 容量对 Cache 命中率的影响。使用模拟器测试 32×32 大小的矩阵

乘的 trace，Cache 行大小固定为 4 个字，为了使命中率的变换更加明显，Cache 的替换策略使用 LFU，使用直写策略。记录 Cache 的命中率和平均访问时间，得到下图。

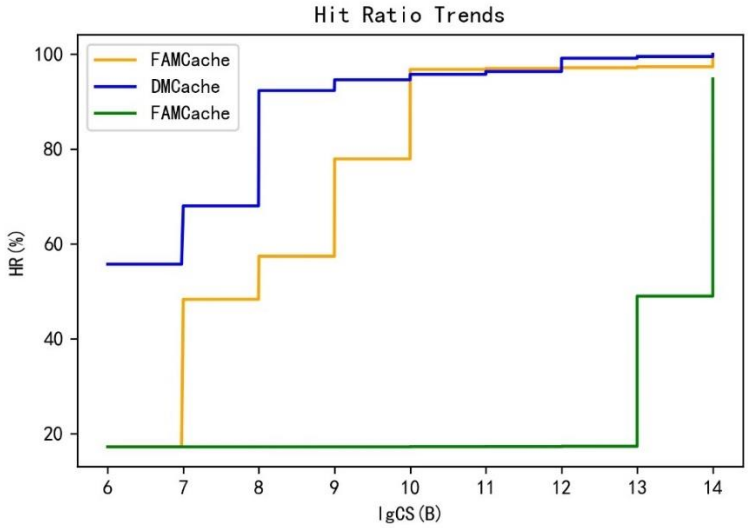

图 8：Cache 命中率和 Cache 大小的关系

Cache 的大小必须是 2 的幂次倍，图 8 的折线呈现阶梯型。随着 Cache 的容量增大，Cache 的击中率不断增大，不同映射方式的 Cache 容量增加的敏感程度不同，可能是由于使用 LFU 策略的缘故，全相联 Cache 的命中率迟迟不肯增长。在容量大到基本可以容纳数组内所有元素时才增长到 99%。Cache 命中率在 lgCS 取 7-10 时增长最快，当 lgCS 大于 10 时，命中率仍然在增长，但增长幅度很小，这就是为什么 Cache 的容量不宜过大的原因，当命中率达到百分之九十多后，将 Cache 命中率增大一点需要花费百倍的努力。

除了 Cache 容量可以影响 Cache 命中率之外，Cache 内部物理内存单元的划分也会影响 Cache 命中率，取 Cache 的容量为 1KB，探究 Cache Line 的大小与 Cache 命中率的关系。

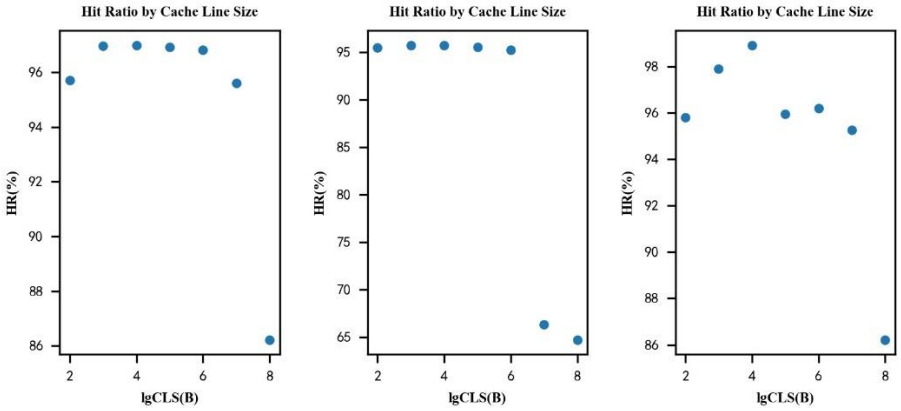

图 9：Cache 命中率和 Cache Line 大小的关系

使用 32×32 大小的矩阵乘的 trace，Cache 的容量设为 1KB，以 2 的幂次倍变化 Cache Line 的大小得到图 9。从左到右依次是 GAMCache，DMCache 和 FAMCache

测试的结果。从图 9 可以看出，Cache Line 过大会影响 Cache 的命中率，当 Cache 比较小时，影响不是很明显，最优的 Cache Line 的大小在2^4B 左右。最后讨论组相联中组数对 Cache 性能的影响。

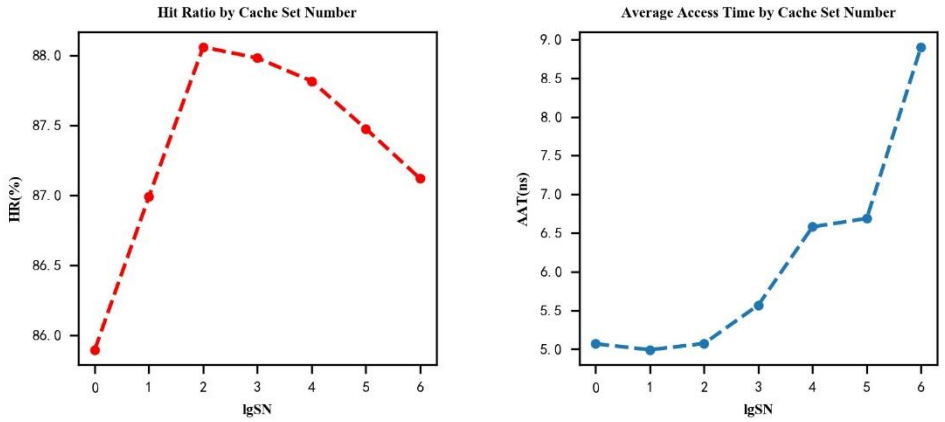

图 10：Cache 组内行数对 Cache 性能的影响

使用 Cache 模拟器测试一段 trace，得到图 10 结果。从图 10 左图可以看出当组内行数过大或过小都为降低 Cache 命中率，当 lgSN=2 时，命中率最高。Cache 的替换策略使用 LRU，替换时需要便利组内所有计数值，所以当组内行数增多时，Cache 平均访问时间较长。

4 矩阵乘的优化策略

4.1 简介

在不降低矩阵乘的复杂度的前提下，适当改变访问矩阵的顺序以提高程序的局部性，利用 Cache 访存的特点减少访存次数，从而到达提升矩阵乘的效率。目前我们已知的关于提高 Cache 命中率的矩阵优化有三种，第一种是改变访问矩阵的顺序，把按行访问搬移到最内层循环；第二种是矩阵分块，将矩阵分成若干小块，每一小块的大小可以存放在 Cache 中，块内进行局部矩阵乘；第三种是，将先将矩阵转置，转置后矩阵乘按行访问。

在编写矩阵乘代码时，为了突出矩阵乘访存部分，生成 trace 时把有关矩阵初始化操作删去，尽量保留矩阵乘最纯净的部分。为了对比矩阵大小对乘法效率的影响，我们组生成了不同版本 trace。按照不同矩阵的大小，将 trace 分为 S8，S16，S32，S64，S128。分块矩阵乘按照分块有可以分为 B8，B16，B32，B64。按照乘法策略分为 ijk，jik，ikj，jki，kij，kji，transpose，block。实验要求生成 500×500 的矩阵乘，经过很多次尝试后，虽然成功生成了 500×500 的 trace，可是这个 trace 达到难以在短时间内处理。即使经过之前所提到的 trace 压缩处理后，只留下访问数据的 trace，而且 trace 不多一个字符。达到极致后的 trace 解压后依然有 19G。处理 500×500 的矩阵不太现实，我们组集中处理小尺寸的矩阵。

4.2 实验结果

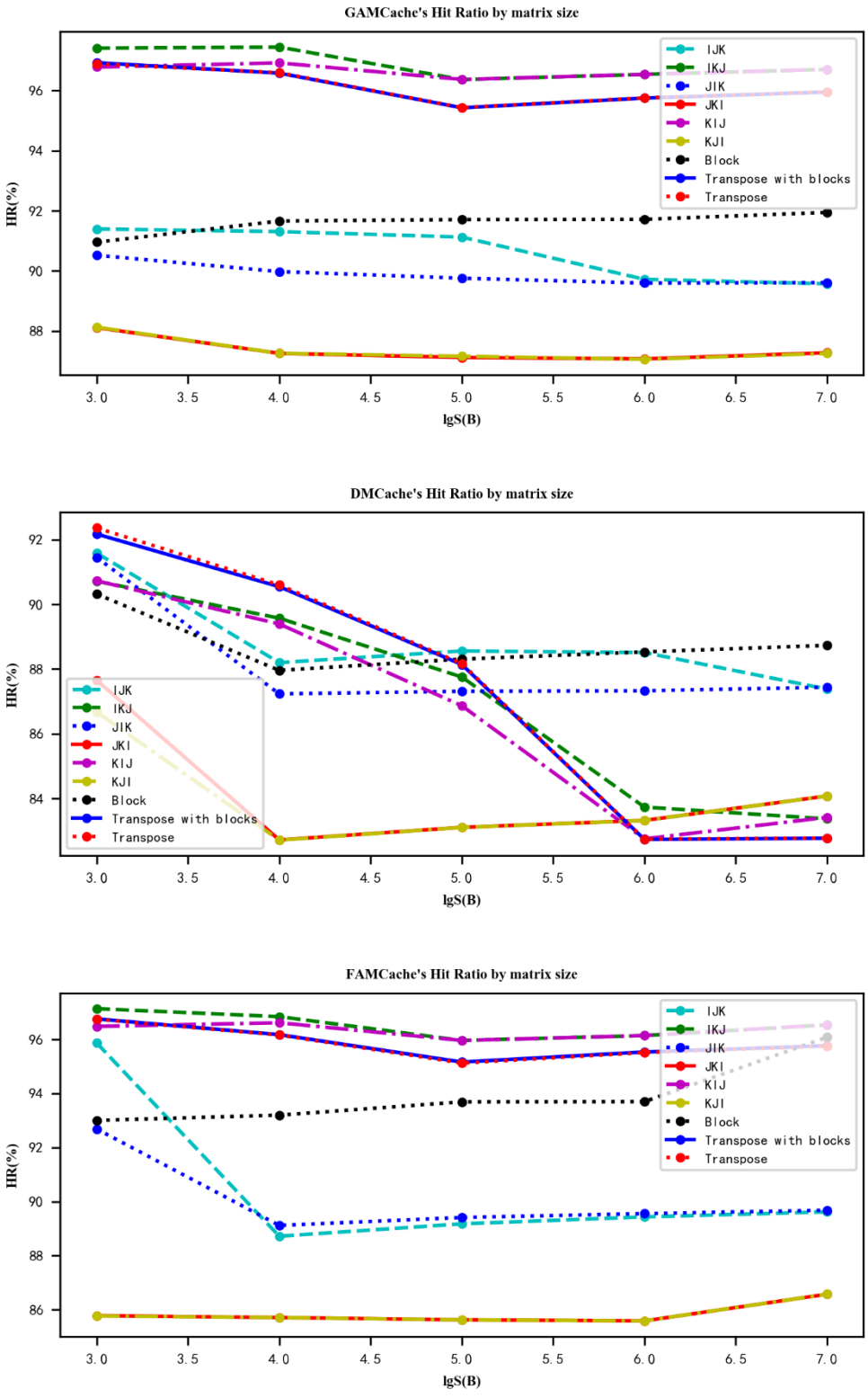

图 11：不同乘法策略对 Cache 命中率的影响

使用 Cache 模拟器测试 S8，S16，S32，S64，S128 矩阵乘所得到的 trace，

得到 Cache 的命中率与矩阵乘的关系如图 11。Cache 的大小为 256B，行大小为 16B，组相联每组含有 4 条 Cache Line，替换策略为 LRU，写策略为写回，使用的 trace 只包含取数据的访存操作。得到的结果与预想的差别很大，Cache 命中率过高，即使矩阵的大小成倍增加，Cache 的命中率依然居高不下，这种情况在之前的访问数组的 trace 也出现过，猜测可能的原因是循环计数值的访问操作或者取指令中的操作数操作冲淡了访问数组的操作。即使这样，通过微小的命中率变化仍然可以找出矩阵乘策略与 Cache 命中率的关系。比较图 11 的 9 条折线高低，这几种策略的排序大概是这样的，HR(JKI)=HR(KJI)<{HR(IJK)，HR(JIK)，HR(Block)}<HR(Transpose)=HR(Transpose with block)<HR(KIJ)=HR(IKJ)，IJK 等结果和预期的相同，KIJ 和 IKJ 在最内层循环按行访问数组，Cache 的命中率最高。IJK 和 JIK 最内层循环一个按行访问一个按列访问，它们介于 KJI、JKI 和 KIJ、IKJ 之间，JKI 和 KJI 最内层循环都是按列访问，它们的命中率最低。另外，值得注意的是，转置矩阵乘无论是直接转置还是分块转置对结果均具有相同的影响，这个结果令人惊讶。分块矩阵乘也没有期望的那么好，它不如 KIJ 和 IKJ。是由于块的大小不合适造成的？之后又多增加几组实验。测试不同分块矩阵乘的 trace，得到如下结果。

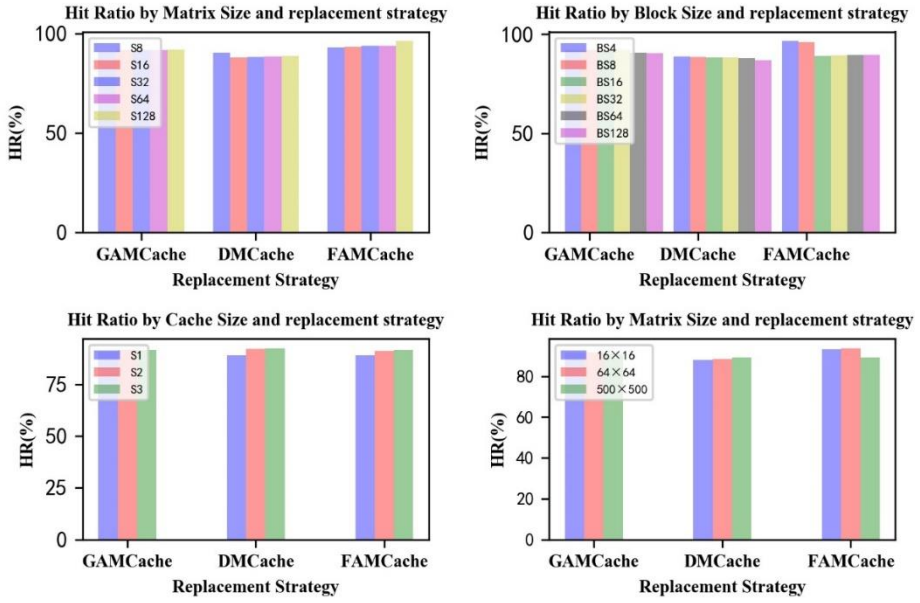

图 12：分块矩阵乘中 Cache 命中率的影响

图 12 中左上图是探究矩阵大小和 Cache 命中率的关系，Cache 的各个参数和之前一样，从图中可以看出分块矩阵乘的矩阵大小对 Cache 的命中率没有影响，即使使用 500×500 的矩阵，命中率也在 90%左右，右下图是 500×500 矩阵和小矩阵直接的比较，分块大小为 16×16。右上图是矩阵分块大小与 Cache 命中率的关系，块大小对 Cache 命中率影响在 1%之内，可能是由于矩阵大小和 Cache 不相上下导致矩阵的一行可以完全放在 Cache 中，如果增设 500×500 矩阵分块大小和命中率的关系，可能得到的就是另一个结果。左下图是关于 500×500 矩阵乘的命中率，S1-3 分别表示不同大小的矩阵，S1：CS(Cache Size)=256B,CLS(Cache Line Size)=16B；S2：CS=512B,CLS=32B；S3：CS=1024B,CLS=64B。Cache 容量成倍增加，命中率有微小的增长趋势。

结论分析与体会：

Cache 工作原理是利用程序的局部性减少访存次数以提高程序性能。为了提高 Cache 的命中率同时较少访问 Cache 的平均时间，出现了不同的策略。

内存物理块和 Cache 存储单元块的映射关系分为直接映射，全相联映射，组相联映射。经过测试不同 trace 的 Cache 命中率，我们发现全相联映射具有较大的灵活性，Cache 命中率最高，直接映射的命中率较差，组相联介于两者之间，在《实验数据》表 18 可以明显体现处这一点。全相联映射虽然具有较高的命中率，但是它也是有很大缺陷，一方面是硬件上的要求，另一方面，它不能和 LFU 和 LRU 一起使用，因为这两种替换策略在发生替换时需要比较 Cache 每一行的计数值，如果全相联和 LFU、LRU 一起搭配使用，Cache 命中率虽然很高，但是 Cache 的平均访问时间却大大提高，图 7 的下面两幅图说明了这一点，图 7 左上关于全相联 Cache 命中率非常低的原因是由于 LFU 替换策略造成的，各个进入 Cache 的行很容易被替换掉，但是 LFU 对直接映射基本没有影响，因为直接映射不存在替换策略，LFU 对组相联的影响介于两者之间。把 Cache 分成若干组，组内实现全相联，组建直接映射，兼具直接相联和全相联的优点。

替换策略有随机替换、LFU 策略、LRU 策略和 FIFO 策略，随机替换和 FIFO 替换最为简单，LFU、LRU 除了要求特殊的硬件之外还需要占用 CPU。LFU 是最差的替换策略，它经常会替换刚刚进入 Cache 的数据，具有分成低的命中率，图 6 体现出这一点。其它策略对 Cache 具有近乎相同的影响，无论是在命中率还是访问时间上，这种情况可能与 trace 有关。

写策略有直写策略和写回策略，写策略影响的不是 Cache 的命中率，而是 Cache 的平均访问时间，图 7 表明直写方式具有较大的访问时间，降低 Cache 的吞吐量，写回方式与之相反。但是写回方式的缺点是，内存和 Cache 数据不一致，这给数据共享带来不便。

Cache 内部参数的设定没有统一的规则，需要进行实验，Cache 容量过小，命中率很低，稍微提升一点，命中率迅速提升，当命中率上升到一定程度后，再提升需要耗费千倍的努力。Cache 行的大小和组相联中组内行数既不能过小也不能过大，一般取中间某个值。

矩阵乘有三种不同的方法，它们的目的都是为了在影响结果的正确性和复杂度的前提下提高 Cache 的命中率以提高效率，图 11 表明 KIJ 和 IKJ 具有较好的性能，JKI 和 KJI 性能最差，IKJ 和 JIK 介于两者之间。转置乘时分块转置和直接转置具有相同的效率，这可能与我们所用矩阵的大小有关，矩阵大小相比 Cache 非常大时，分块转置可能比直接转置要好。最后讨论了分块矩阵乘，由于矩阵过小和一些关于 trace 不明的原因，没有得到期望的结果。

经过本次实验，对 Cache 内部的工作原理有更深刻的印象。在实验过程中一定要仔细，尤其是测量数据时，数据不准确可能会导致错误的结论。