Computer Architectures Project 3.0

江天源(1100016614) and 吕鑫(1100016639)

简介

本课程项目要求对Racetrack Memory（RM）这种存储技术进行研究，理解并掌握RM的基本特性并深入探索，在几种层面上提出一些优化的设计。

由于Gem5学习成本过大，难以配置，相关文档少，很难在上面重新实现基于RM的Cache。因此最终我们决定自己另外写程序，手动实现一个模拟器，这样也方便我们调试相关功能。

模拟器实现

模拟器采用Python语言编写，主要是模拟了访问L2 Cache的过程，忽略L1 Cache。

文件列表

文件名	功能
__main.py__	主程序入口
configs.py	相关参数的配置
l2cache.py	模拟l2cache功能
rm.py	模拟Racetrack memory功能, 保存数据

文件名	功能
sram.py	模拟sram功能, 保存tag等标记位
trace.py	Trace类定义
traceparser.py	对文件中trace条目的解析

模拟过程

为了仿真，模拟了CPU的实际运行过程，程序中模拟**每个周期**的执行细节，周期中检查当前指令是否执行完，若执行完则执行下一条指令。

L2Cache实现

结构

L2Cache由SRAM和Racetrack Memory组成，SRAM负责存储Tag和Valid信息，RM则负责存储数据。

图示（From *"TapeCache: A High Density, Energy Efficient Cache Based on Domain Wall Memory"*）：

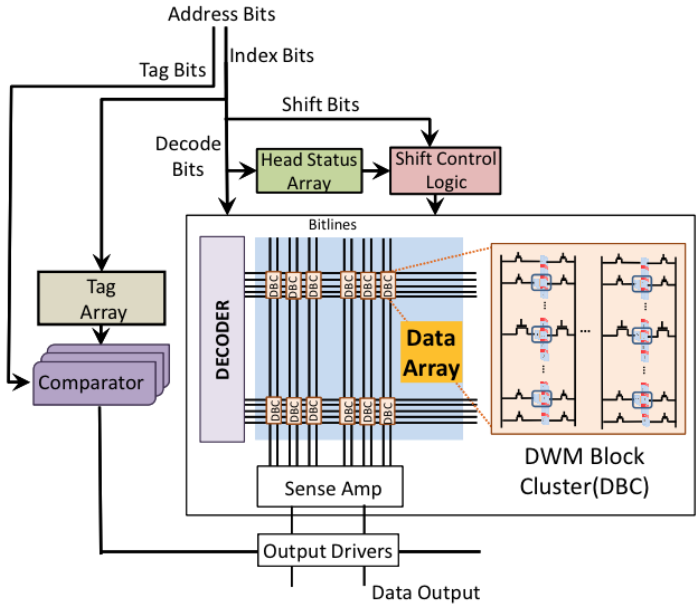


Figure 4: TapeCache organization

默认参数

参数	值
L2总大小	4 MB
L2组相连数	8
L2读写延迟	1/1 cycle
L2访问延迟	6 cycles
L2 miss延迟	100 cycles
替换策略	LRU

访存过程

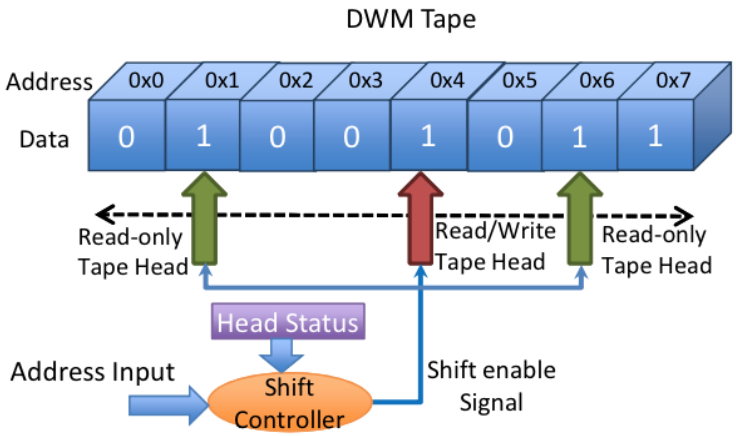
1. 根据访问地址，解析得到Index和Tag
2. 先在SRAM中比较Tag
3. Tag相同且Valid为True，则缓存命中，若为读操作，则访问RM得到数据；若为写操作，改写RM数据，在SRAM中标记dirty，替换时write-back
4. Tag不同则缓存不命中，访问主存，更新SRAM的Tag和RM的相应数据，再返回
5. SRAM中用时间戳记录每个数据的访问时间，方便LRU策略替换时使用

Racetrack Memory 实现

结构

由条带（Tape）、读写端口（Port）、条带组（Group）等组成。

图示：



默认参数

参数	值
每条带Domain数	64
每组条带数	512
Shift延迟	1 cycle
R/W端口大小	12 Domain
W端口大小	8 Domain
R端口大小	4 Domain

访存过程

1. 根据地址中的Index和Set划分策略，找到数据块（Block）所在的Set Num
2. 再根据SRAM在Tag比较过程中得到的值，确定数据的Way Num
3. 根据Set Num和Way Num确定数据所在的具体Group Num
4. 移动条带，使读/写口对准数据所在的Block

实验设计

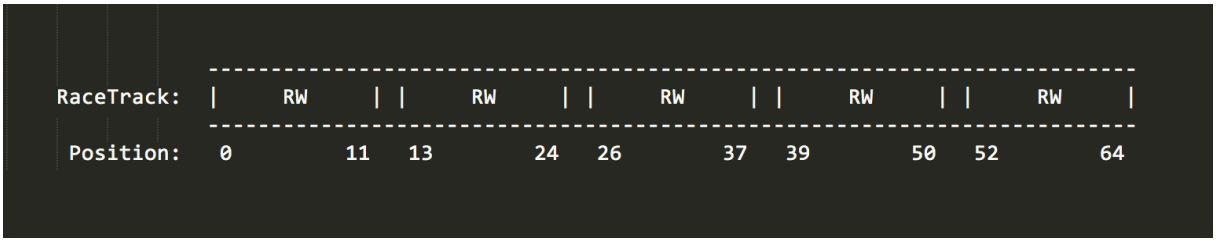
读写端口

我们在读写端口的摆放上设计几种不同的摆放策略，并进行实验，考察读写端口的摆放对于最终性能的影响。

策略1：摆满RW端口（5RW）

64个Domain的条带，尽量使大小为12Domain的RW端口平均分布，因此摆放位置为0、13、26、39、52。

图示：



策略2：摆满R、W端口（2W+12R、4W+8R）

考虑到读操作比写操作多，因此安排两个/四个写端口，剩下的都用读端口占满。

图示（两个写端口）：

图示（四个写端口）：

策略3：摆满RW、R端口（2RW+10R）

与2类似，只是用RW端口代替W端口。

图示：

策略4：摆满RW、W、R端口（2RW+2W+6R）

图示：

Set划分

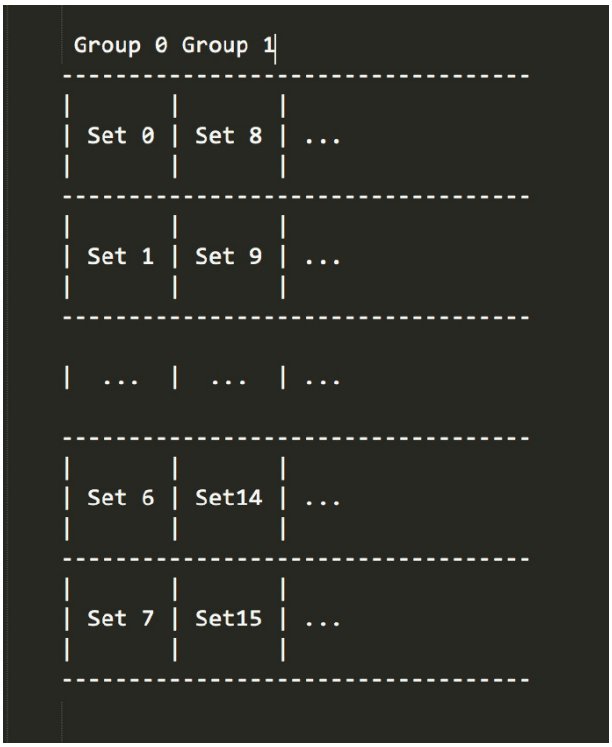
先来计算一下一些值的具体大小

```
L2 Size = 4M

Group Size = 64 bit * 512 = 4096B = 4K

Group Num = 4M/4K = 1024

L2 Assoc = 8

Set Num = 64 * 1024 /8 = 8192
```

策略1：连续划分（Continuous）

从Group 0开始，连续的分配Set，如Set0-Set7分配在Group 0上，Set8-Set15分配在Group 1上，以此类推。每 `8192 / 8 = 1024` 个Group组成一路。

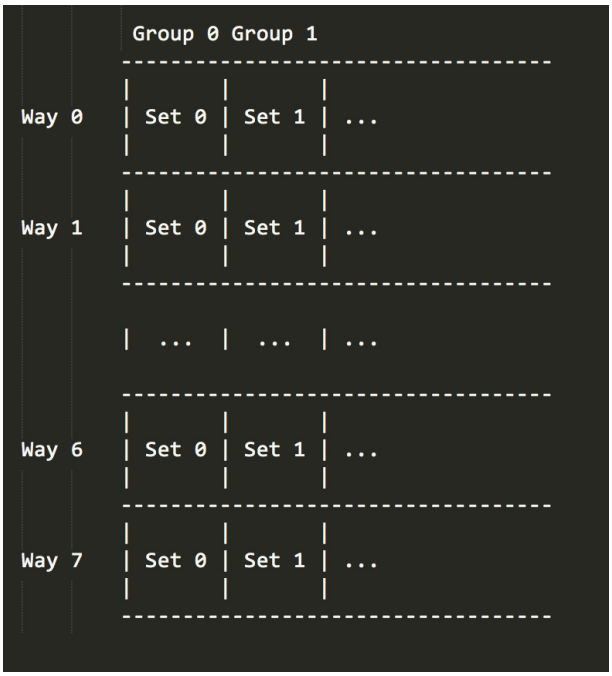
图示：



策略2：按相连度划分（Cross-Way）

横向划分，Set 0在Group 0，Set 1在Group 1，**这样每个**Group的8个Set编号相同，**每个**Way的数据遍布所有1024个Set。

图示：



端口移动

端口移动主要研究集中在两个方面：一个是访存时的移动策略，另一个是访存完的移动策略。

访存时可以**动态移动（Dynamic）**，即动态计算当前离数据最近的一个端口，也可以**静态移动（Static）**，即根据数据在条带上的位置选择既定的端口。

访存完可以**返回原位（Eager）**，即访问数据完移回原来的位置，也可以**原地不动（Lazy）**，即访问数据完保持位置，不移回。

根据这两个方面，衍生出四种策略：

策略1：DE（动态移动+返回原位）

策略2：DL（动态移动+原地不动）

策略3：SE（静态移动+返回原位）

策略4：SL（静态移动+原地不动）

其它优化

我们在实验中还实现了Preshift策略，就是当一条指令被预读取时，如果与当前指令所用条带不冲突的话，就在这个周期预先移动，这样该指令被执行时，就直接可以读数据。

这样的Preshift策略，即在前一个指令还在执行时便将下条指令提前移动好的策略，无疑能够有效地降低Shift Overhead，极端情况当所有数据都不在一个条带上时，Shift Overhead会大大降低。但相应的，每个周期内需要完成的任务也相应增多，需要额外的硬件支持。

Preshift的执行逻辑如下：

1. 若下一条指令已经就绪，且与当前指令所用条带不冲突，开始准备Preshift
2. 经过6 cycles访问L2 Cache
3. 查询SRAM确定下一条指令是否hit/miss
4. 若miss：等待当前指令执行完成后开始访存等操作
5. 若hit: 开始preshift，直到移动到对应位置或前一条指令结束

Preshift过程中可能会遇到执行了一半的时候，前一条指令突然结束了。针对这种情况的处理如下：

1. 在访问L2 Cache的过程中被打断：继续执行访问L2 Cache及其后续过程即可
2. 在shift的过程中被打断：记录shift的位置和距离，继续shift及其后继过程
3. shift完成后：直接开始读/写操作

为了测试优化的效果，我们进行了很多的实验。

实验结果

测试文件

为了方便，我们选择了trace中的六个文件作为参考依据进行实验，我们主要选择了几个较典型的trace文件：读指令很多的，写指令很多的和读写率正常的。

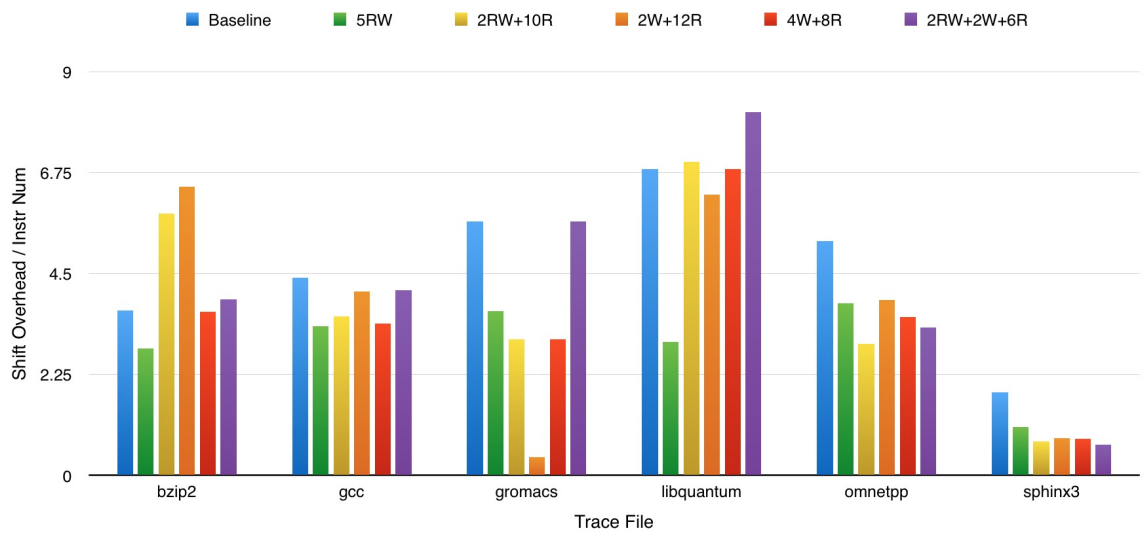
六个文件分别是：

文件名	指令数目	读比例	写比例	文件大小
401.bzip2.trace	3239378	38.01 %	61.99 %	92.8 MB
403.gcc.trace	2302351	75.1 %	24.9 %	64.7 MB
435.gromacs.trace	1903172	98.9 %	1.1 %	54.1 MB
462.libquantum.trace	1388530	0.01 %	99.99 %	38.9 MB
471.omnetpp.trace	2483572	83.5 %	16.5 %	69.5 MB
482.sphinx3.trace	1875	85.17 %	14.83 %	47 KB

我们分别对几个策略做了实验，在实验中主要考虑对Racetrack Memory的优化，因此以shift overhead/instr num作为评估依据，而对于L2访问延迟，L2 Miss Penalty等所有情况下都一样的参数，则不在实验结果分析时进行考虑。（模拟过程中并没有忽略这些延迟的影响）。

读写端口摆放实验

实验结果



结果分析

从表中可知，RW port和W port较少的策略，在写比例较少的数据上（如gromacs）表现比较优秀，远胜于Baseline。但是在写比例较多的数据上（如libquantum）表现自然差强人意，甚至不如Baseline。

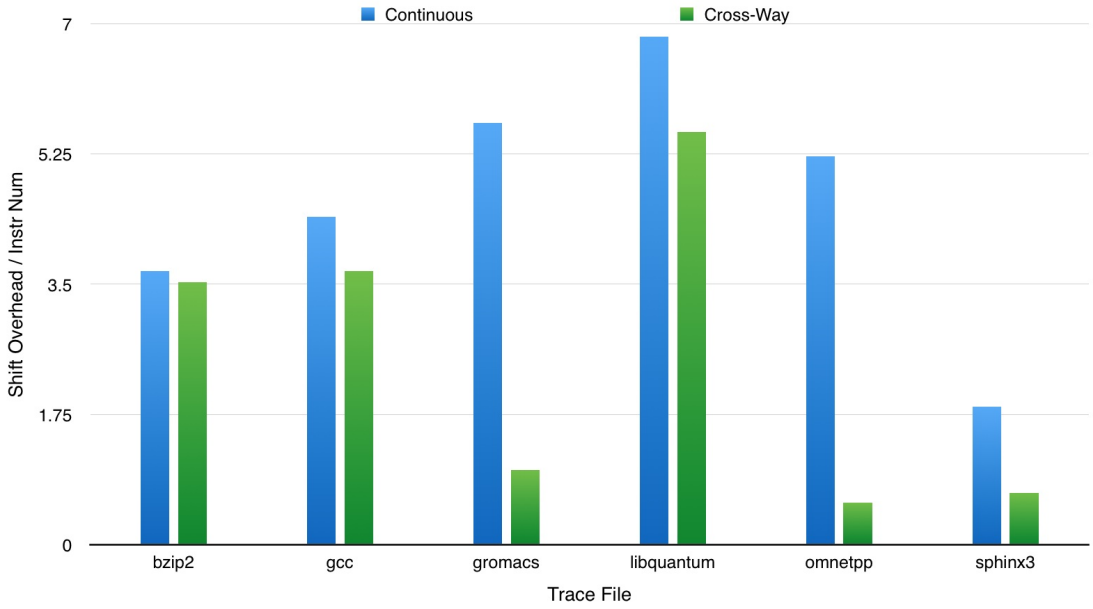
表中综合下来比较优秀的是两种策略：5RW和4W+8R，两种都是比较均匀地将读和写遍布整个条带，因此性能不会对数据集的读写比例有太大依赖，实验结果符合预期。

值得注意的是在gromacs数据上，2W+12R的策略远胜于其它策略，思考可能是该数据集读操作的地址分布主要都集中在2W+12R策略的读端口上导致移动次数较少导致的。当然gromacs数据集读比例极大也是原因之一。

综上，在所有数据集平均来看，5RW和4W+8R的策略表现较为优秀，是两种较为合理的摆放策略。

Set划分实验

实验结果



结果分析

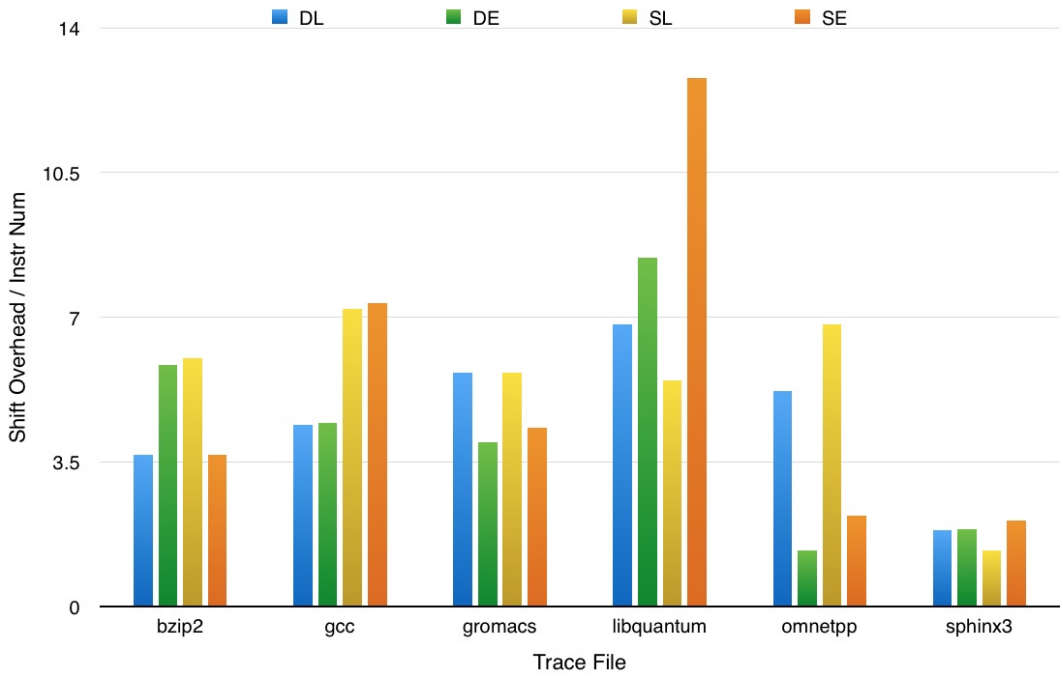
在所有数据集上，实验结果中显示Cross-Way的划分策略均要好于Continuous（Baseline）的分配策略。

我们认为这是程序空间局部性导致的。Cross-Way策略将Set平均分布在各个Group中，访问同一个Set的若干个数据，Continuous策略会可能在一个Group上来回不停的循环往复移动，而Cross-Way则只需在第一次时移动几条条带，之后便不需移动。

综上，由于空间局部性的缘故，Cross-Way的Set划分策略是一种比较合理的策略。

端口选择与移动策略实验

实验结果



结果分析

从实验结果中可以看出，DL策略有着最小的平均shift overhead，因为DL策略可以很好的利用空间局部性的优势，对于相近的数据连续访问需要的移动距离相对其他策略优秀。相对的，SL策略则在连续局部性访问的情况下会有最糟糕的表现。

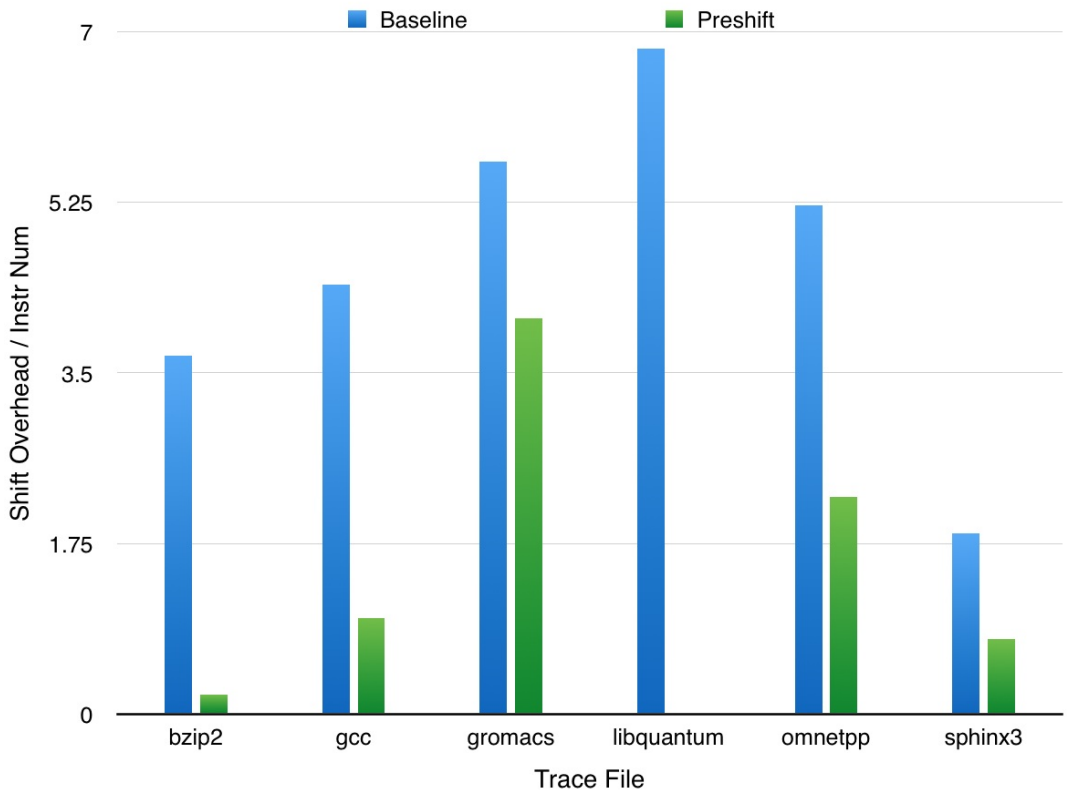
从libquantum这组数据的结果中发现此时DL的结果不如SL的结果。观察了这个trace中的内容后发现，其L2 Cache访问的局部性非常差。使用DL策略时，当前的最优选择(最近端口)，会让条带上的端口分布偏置，从而导致后面的shift overhead很大。

此外，理论上来讲如果SE策略按照统计方法选择最少移动距离的方案会跟DE完全一致，但是为了不针对某一个某一个数据进行特别优化导致过拟合，最终采取了“按照所属端口的范围”的方针进行读写端口选择。但综合来看，DE/SE方案对数据的局部性依赖比较大，在局部性较好的数据，如gromacs和omnetpp中表现甚至比DL好；而在局部性较差的数据，如libquantum中表现却非常的差。

综上，端口和移动策略的选择不能一概而论。平均情况下最好的是DL，但如果数据有着极端的局部性，或极端的没有局部性，则可以考虑使用其他几种策略。此外还可以考虑的DL策略的进一步优化(比如在下一条指令为其他条带的情况下(即最近没有该条带的操作时)移回到某固定位置)。

优化策略实验

实验结果



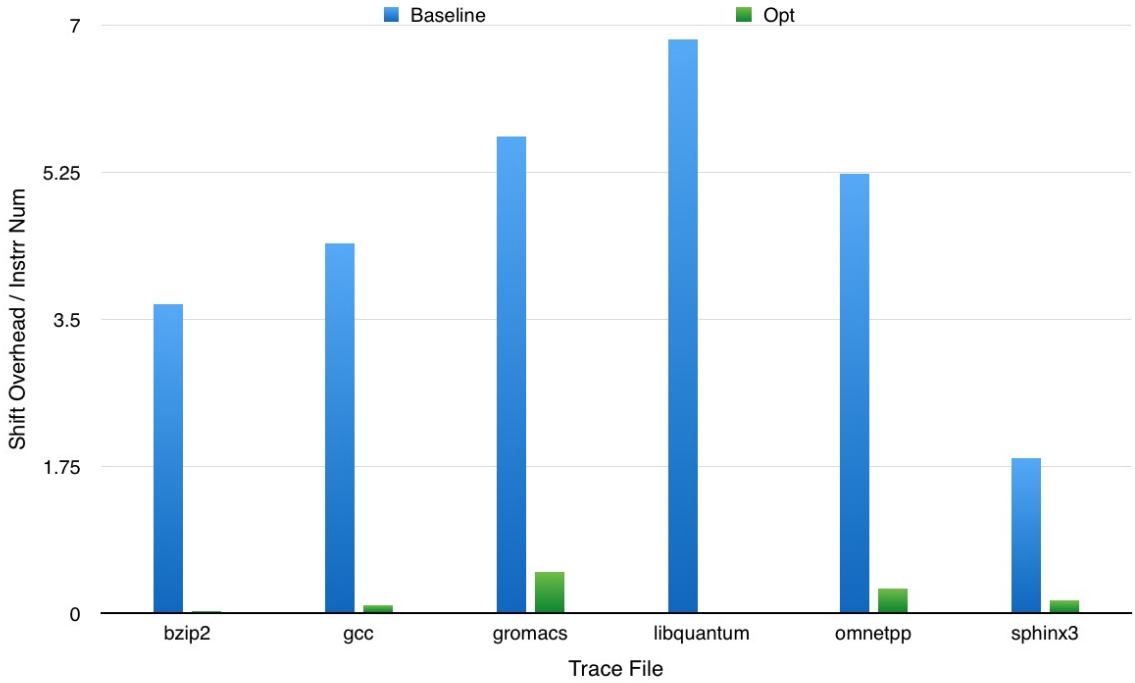
结果分析

使用Preshift策略之后优化效果拔群。对比几个数据之后发现，对于空间局部性较好的数据(如gromacs)，Preshift优化的效果没有局部性差的数据那么显著。

而对于毫无局部性的libquantum数据甚至使shift overhead几乎达到了0。这是由于Preshift的机制使得其在局部性差的情况下更有可能实现当前指令和预取指令的并行(下一条指令更可能是不同条带上的操作)。

实验总结

综合各项实验的结果，我们最终选择了“4W+8R”的端口摆放策略，Cross-way的set划分策略，DL端口选择/移动策略，并加之以Preshift优化作为我们的最终结果。



Design	bzip2	gcc	gromacs	libquantum	omnetpp	sphinx
Baseline	3.67283	4.40218	5.66777	6.82984	5.22195	1.84907
Opt	0.02736	0.08842	0.48715	0	0.29742	0.15520

在这次大作业中，我们选择了之前并不怎么熟悉的Python语言进行代码编写，在过程中遇到的最大挑战也是对Python的一些特性和功能不熟悉。但随着RM模拟器baseline的完成与之后一步步的优化，已经能比较熟练地使用Python了。

在开始编码之前，我们认真讨论了整个Racetrack Memory的设计，并阅读了相关的论文，不仅对RM有了很多深入的了解，也对内存体系、Cache的结构等很多方面的问题有了一个更清晰理解。而之后模拟器的编写过程非常顺利，没有遇到太多逻辑上的bug，这也应该归功于课程学习以及之前的充分准备吧。同时由于将项目部署到了github上进行管理，遇到问题时也很容易地进行对比和恢复。

随后最后的反复测试的结束，这次的大作业终于完成了。从最开始的不知从何下手，到最后顺利地运行出结果并且一点一点看着结果改进，感觉还是很有成就感的。

参考资料

1. M. Mao, W. Wen, Y. Zhang, H. Li, and Y. Chen, **"Exploration of GPGPU Register File Architecture Using Domain-wall-shift-write based Racetrack Memory,"** Design Automation Conference (DAC), Jun. 2014
2. Z. Sun, X. Bi, A. K. Jones, and H. Li, **"Design exploration of racetrack lower-level caches.,"** ISLPED, pp. 263–266, 2014.