RaceTrack Memory Explorations 实验报告

1200012863 刘敏行

1200012864 刘子渊

一、 实验概述

RaceTrack Memory 是一种新型的非易失性存储结构，相比于传统 Memory 的优势是拥有更高的存储密度(storage density)，因此如果将它用于 L2 cache 将大大提高效率。RaceTrack Memory 的具体特性在这里不再赘述。

本次实验，我们在理解 RaceTrack Memory 特性的基础上，首先编写一个模拟器模拟 RaceTrack Memory 的行为，然后在给定参数(baseline)基础上，对其进行优化，减少它的总时间延迟(latency)。

二、 实验内容

1、 模拟器的实现

模拟器主要包括两大部分。第一部分是通过分析待访问的地址得到该地址对应在 cache 中的 index 和 tag，从而定位到某一 cache 块，这一部分和我们以前学的 cache 完全一样。第二部分是找到对应 cache 块在新型 cache 中的位置，即哪一个 group 的哪个 domain，然后再移动条带对对应位置的数据进行操作。

下面简单阐述一下我们实现的针对 baseline 的模拟器的思路。

总共定义了三个结构体，cache_t 代表新型 cache 的每一个块，Set 代表一个 set 的一行对应的 group 和 domain 的位置，GroupPort 代表一个 group 上端口的类型和分布。

在 main 函数的最开始，首先是对以上三个结构体信息的初始化(init_RTcache, init_portmap, init_setmap)，其中端口摆放和 set 的划分遵照了 baseline 给的要求。然后从 trace 文件中不断读入请求进行处理(readFile)，更新时间滴答(time_tick)后访问 cache(visitCache)，完成有关 cache 的基本操作：得到 Index，Tag 以及替换(LRU)。在确定一个数据的位置后，移动条带进行具体操作(RT_shift)，在其中计算在此过程中的延迟(latency)，更新时间滴答。

这里有几点需要额外说明。首先，我们假设 Cache 的 Tag 比较过程是由另一个 SRAM 而非我们的 RaceTrack Memory 完成的，因此 Tag 的比较时间在模拟器中没有体现。另外，当 cache miss 时，实际上要去

memory 处取数据，但本实验中将其忽略，否则这个时间当成为主要的时间延迟，对实验意义不大。对此，我们计算 miss rate，在优化时会保证这个量不会太大。另外，读写时间、L2 cache 延迟时间都是无法优化的“固定成本”，因此我们计算的总时间延迟就是总的**条带移动**带来的延迟。

2、 优化策略

之前提到，我们把条带移动总时间作为性能评判标准，因此减少在访问数据时产生的条带的移动成了优化性能的关键。我们根据对 RaceTrack Memory 特性的理解，设计出了一系列优化策略。我们首先分别从端口排布、Set 的划分与编址、读写策略等方面思考优化点，然后再通过调整将这些优化结合、联系起来，最终形成一个总的优化策略。本节将主要介绍我们想到以及实现的优化策略，具体效果将在下一节中讨论。

1） 静态策略

① 条带端口排布(“Port Distribution”)

显然，条带上端口的类型、个数以及排布位置会对总的性能有不小影响。对于类型来说，根据输入 trace 读/写比例的不同，安排对应比例的读/写端口；对于个数来说，一般来说个数越多越好，因为可以减少访问 domain 的平均移动次数。但是一方面是要满足空间限制，另一方面也要在硬件资源和性能之间做出相对取舍；对于排布位置来说，当然要尽量把端口排布得均匀一些，从而可以减少访问 domain 的平均移动次数。

基于以上思路，我们提出了三种排布策略：

A. 端口个数、种类不变，调整位置

Baseline 将 4 个读/写端口放在了 0、16、32、48 处，我们觉得这个排布不合理，因为原则上 4 个读/写头的最大移动开销是 64/4/2=8，而针对这个排布，如果访问 63 位置，则移动距离是 15。因此原来的端口排布是不均衡的，取而代之，将 4 个端口分别放在 7，23，40，56 处，则做到了均衡，降低了最大移动开销的值。

B. 尽量增加读端口个数

“在 4 个读/写端口基础上只加入**读**端口”：在位置 7、23、40、56 放读/写端口，在位置 0、15、31、48、61 放读端口。（这个排布是符合要求的，如要看具体设计方案请看“port_design.xlsx”）

C. 尽量增加写端口个数

“在 4 个读/写端口基础上只加入**写**端口”：在位置 11、25、38、50 放置读/写端口，在位置 5、31、62 放置写端口。（这个排布是符合要求的，如要看具体设计方案请看“port_design.xlsx”）

具体实现上，只需对 init_portmap 函数进行修改。

② 端口选择策略优化(“Port Selection”)

在访问一个数据时，需要选择对应的端口，从而进行条带的移动。Baseline 中的策略是选择离访问位置最近的端口，这是选择了一个局部最优解。但是在特殊情况下这种策略可能会导致不好的结果。例如，条带碰巧每次都向一个方向移动，这样移动几次之后端口严重偏向一边，如果突然来了一个访问反方向的最远的位置的请求，那么移动的开销就会很大。为了避免这种情况，我们进行了一个小优化，当移动距离相等时，选择“使得条带回归原来位置”的端口。

另一种更大刀阔斧的解决方法是放弃“每次找最短距离”的策略，而是预先给每个端口分配一个“管辖范围”，即某一位置确定由某一端口来访问，与上述提到的一样，也是避免条带在特殊访问条件下会偏离得太多。

基于以上思路，我们对 RT_shift 函数进行了修改。

③ Set 的编址与划分(“Set Initiation”)

Baseline 中将 set 在 group 中按纵向编址，而根据程序的空间局部性和时间局部性，往往会访问相邻 set 的数据，且数据有可能会被反复访问到，如果按照 baseline 的分法则有可能频繁对一个 group 的条带进行反复操作，造成大量时间延迟。因此最好将相邻

set 分到不同的 group 中去，而且尽量分得开一些。一种比较直观的方法是将 set 在 group 中按横向编址，如图所示：

group0	group1	…	group1023
Set0	Set1	…	Set1023
Set1024	Set1025	…	Set2047
…	…	…	…
Set7168	Set7169	…	Set8191

目前我们也就想到了这种编址方式。

有关 set 的划分，baseline 中每一个 set 是 1*8 的，即只横跨一个 group，由此出发，我们想到可以使得 set 横跨 2 个/4 个/8 个 group，如图所示（每个图表示一个 set，只画了横跨 1 个/2 个的情况，其他 2 种情况类似）：

span=1

group a

0
1
2
3
4
5
6
7

span=2

group a	group a+1
0	1
2	3
4	5
6	7

这样的划分实际上是在不相邻 set 分散的基础上，又把一个 set 的不同块也尽量分散，这样的一个好处是当频繁访问某一个 set 时，对不同块的访问不会导致同一个条带的反复移动。

基于以上思路，我们对 init_setmap 函数进行了修改。

2） 动态策略

① 条带预先移动(“Pre-shift”)

这个策略的思想是在当前指令尚未结束的时候，预测下一条指令的地址以及类型。由于同时可以有两个条带同时移动，因此在执

行当前指令的同时，同时预先执行预测的下一条指令，只不过执行预测的指令并不会改变 cache 状态，也不读写数据，而只是移动对应的条带而已。

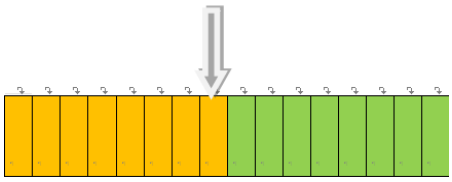
如何进行预测呢？一种最直观的预测方法是上次当前指令的下一条是什么样的指令，这次就仍然预测这一指令。具体的做法是定义一个 map，key 是一条指令（用<address, type>对可以唯一表征），value 仍然是一条指令。当一条指令来临时，就将上一条指令的 value 赋为当前指令。要预测时，key 为当前指令的对应的 value 就是预测的指令。

具体实现主要是修改 readFile 函数和 RT_shift 函数，同时添加了 pre_visitCache 函数。在 readFile 函数中取出预测的指令，pre_visitCache 函数得到指令后进行类似的 cache 访问，区别是不真正修改 cache 的值。之后给 RT_shift 加了一个参数 is_predict，为真时说明是在进行预移动，只需要模拟移动的过程，不需要计算任何时间开销。

为了使得计算简单，我们还限制预移动的位移不能大于并行执行的指令的时间开销，这样，预移动本身就不会带来任何额外时间开销。

② 条带回位(“Eager-shift”)

本策略的思路是充分利用 1.指令之间的空闲时间和 2.当前指令执行的时间，将端口不在原来位置上的条带移动归位，从而在大多数时间里保证所有条带呈现初始的状态。这样做的原因首先是秉承之前提到的一个思想，即希望条带不要向一个方向偏离太多，尽量保持端口均匀分布。这个策略显然能保证这一点。另一方面，当端口位置基本固定后，就可以利用这个不变性做出一些针对性的优化了。例如，对于一个 group 中的两个相邻 set：



黄色和绿色各代表一个 set，还有一个端口处于箭头位置。默认情况下，当需要插入新数据时，是从第 0 个 block 开始遍历的，不妨设最左边为第 0 个 block，那么向黄色 set 加入数据时就会把数据

优先放在离端口很远的 block，很不必要。如果端口固定，那么就可以进行确定的优化：对于黄色的 set，优先从第 7 个 block 开始遍历；对于绿色的 set，优先从第 0 个 block 开始遍历。从而数据尽量存放在端口的附近，访问时减少端口的移动。

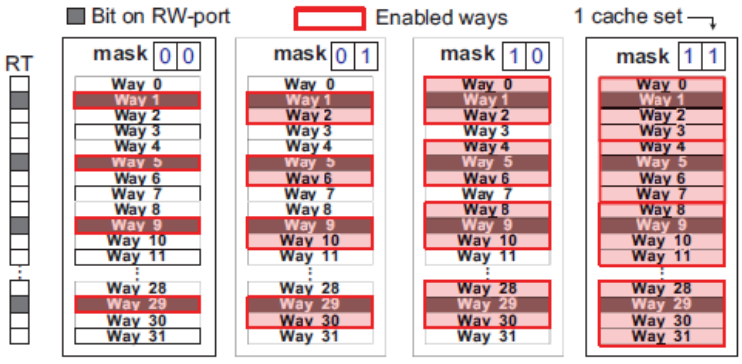
具体的实现主要是对 readFile 函数以及 RT_shift 函数进行修改。在 RT_shift 函数中记录上一条指令对应移动的条带以及移动的个数，在 readFile 函数中，首先在下一条指令来临时计算空闲时间，在这段时间内可以对两个条带进行归位(resetTwoGroup)，同时在执行当前指令时可以对另一个条带进行归位(resetSingleGroup)。这样的话，这个策略自身不会带来任何额外时间延迟。

而上面提到的基于这个策略的优化，是对 visitCache 函数进行修改。

③ 动态组相连度(“Dynamic Associativity Control”)

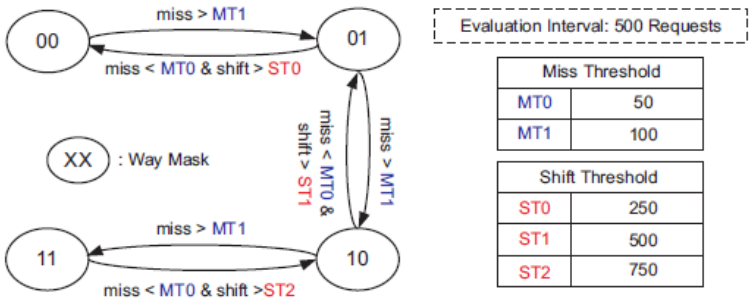
我们最希望看到的情形是要访问的数据正好就在端口的附近。上面的“条带回位”策略已经做好铺垫，帮助我们把端口固定了，下面要做的就是有针对性地把数据放在端口周围了。上面的小优化已经是一个不错的策略，然而我们从论文《Design exploration of racetrack lower-level caches》(作者 Zhenyu Sun, Xiuyuan Bi, Alex K. Jones and Hai Li)中看到了一个更厉害的策略，即动态改变一个 set 的组相连度：在 miss rate 不高、条带移动数高的时候，只开放那些在端口附近的块，当 miss rate 升高时，则应该多开放块，从而减少 miss rate。在这样一个动态机制下，可以尽量将数据控制在端口附近，减少条带的移动，同时又保证不会太高的 miss rate。

该策略的示意图如下，用的是论文中的图，虽然我们实验中具体参数不同，但大体类似。



(图：显示一个 set 的情况，mask 为 0 时开放 1/8 的块，为 1 时开放 1/4 的块，为 2 时开放 1/2 的块，为 3 时开放全部块)

状态转换图类似于我们在转移预测时学到的 2-bit 状态转换图：



(图：当一组的 miss 数和条带移动数达到一定阈值时进行状态转换)

当然，这个图中的数值以及状态数需要根据情况而设定，整个策略的具体实现也比较依赖端口排布、set 编址与划分等，后面会讨论到。

在程序具体实现方面，需要给每个 cache 块加上一个 bool 量 available,以判断这个块是否开放。当一个块被开放时，置 available 为 true，否则置为 false，同时还要置 valid 为 false；需要加上 check_assoc 函数，这个函数每处理 500 条指令被调用一次，作用是对所有 set 进行遍历，依据上面的状态转换图判断是否需要进行状态转换。当需要转换时，调用 change_assoc 函数实现具体的操作，开放/关闭相应 set 的块。

（番外）动态组间映射(“Set Remapping”)(未实现)

这个想法我们由于种种困难没有实现，但是我认为还是可以不妨把大致思路说一下。这个策略与前面的策略相同，依然是希望数据能够尽量地聚集在端口附近。因此首先根据端口的排布情况给所有的 set 划分两个优先级，端口附近的 set 认为是高优先级，其他的 set 则是低优先级。当一个数据访问来临时，如果发现对应的 set 是高优先级的 set，那么照常访问；否则，将该数据从低优先级 set 映射到一个高优先级的 set，如果遇到高优先级 set 已满，则可以把替换掉的数据放入一个低优先级的 set。大致的想法就是强行希望所有的访问都在端口附近，只不过众多细节实现起来较为麻烦，且时间有限，而且多多少少与我们现有的策略矛盾，故没有实现。

三、 实验过程与数据分析

在这部分中，我们将针对之前提到的各种优化策略的效果进行评测，同时对结果进行分析。

我们从给的 trace 中筛选出了 8 个大小、读写比例、cache miss rate 比较正常的 trace 作为我们的实验评测数据，它们分别是 401.bzip2, 403.gcc, 410.bwaves, 435.gromacs, 450.soplex, 453.povray, 456.hmmer, 471.omnetpp。它们的基本情况如下表：

trace	#of Reads	# of Writes	miss rate
401.bzip2	1231218	2008160	2.86%
403.gcc	1729038	573313	4.29%
410.bwaves	1124559	9440220	9.95%
435.gromacs	1882219	20953	0.63%
450.soplex	983162	779705	1.46%
453.povray	918009	395699	3.90%
456.hmmer	4193814	8360421	0.17%
471.omnetpp	2073726	409846	0.39%

1) Baseline

在 Baseline 上运行这些 trace，结果如下表

trace	latency
401.bzip2	17678095
403.gcc	13453357
410.bwaves	24408015
435.gromacs	15299748
450.soplex	9670882
453.povray	7522577
456.hmmer	13545379
471.omnetpp	13311132

2) Port 排布(“Port Distribution”)

A. 实现上一节方案 A(RW_PORT @ 7,23,40,56)

在 baseline 基础上只改变这一点，运行实验 trace，与 baseline 的 latency 比较如下：(miss rate 没有变化，省略)

trace	Baseline	Port_opt1
401.bzip2	17678095	16822378
403.gcc	13453357	13051666

410.bwaves	24408015	24369220
435.gromacs	15299748	15306456
450.soplex	9670882	9300664
453.povray	7522577	7361707
456.hmmer	13545379	14374757
471.omnetpp	13311132	13043297

可以看出，几乎所有 trace 都有一定程度的优化。不过，优化的幅度比较小，有两个 trace 甚至时间反而变多。我们认为虽然新的端口分布更平均，但 trace 的访问并非平均，如果 trace 集中访问某些区域，那么这个优化的效果就不会很明显。

B. 实现上一节方案 B(RW_PORT @ 7,23,40,56, R_PORT @ 0,15,31,48,61)

在 baseline 基础上只改变这一点，运行实验 trace，与 baseline 的 latency 比较如下：(miss rate 没有变化，省略)

trace	Baseline	Port_opt2
401.bzip2	17678095	10816433
403.gcc	13453357	4574339
410.bwaves	24408015	20832186
435.gromacs	15299748	2082564
450.soplex	9670882	5409910
453.povray	7522577	2754445
456.hmmer	13545379	11568605
471.omnetpp	13311132	2907783

为了更体现这个方案对读操作的优化，贴出对应的 read_latency(消耗在读操作上的 latency)：

trace	Baseline	Port_opt2
401.bzip2	9555157	1891821
403.gcc	10371148	1569523
410.bwaves	3883787	912199
435.gromacs	15060152	1881661
450.soplex	5282972	1245037
453.povray	5290317	734994
456.hmmer	3715517	1326618
471.omnetpp	11100153	1071837

可以看出，这一方案使得所有 trace 得到优化，有些 trace 的优化程度更是令人咋舌，优化一个数量级（黄色标注）。实际上，优

化程度很大的 trace，读写比例都较大（>5:1）。再看直接比较 read_latency 的表格，发现 read_latency 的优化幅度基本与之前总的 latency 优化类似，因此证明这里的优化确实基本来自于对于读操作的优化。

因此，这一方案对于读操作很多的 trace，优化效果明显。

C. 实现上一节方案 C(RW_PORT @ 11,25,38,50 W_PORT @ 5,31,62)

在 baseline 基础上只改变这一点，运行实验 trace，与 baseline 的 latency 比较如下：(miss rate 没有变化，省略)

trace	Baseline	Port_opt3
401.bzip2	17678095	11883633
403.gcc	13453357	9043729
410.bwaves	24408015	8991876
435.gromacs	15299748	11082898
450.soplex	9670882	7123100
453.povray	7522577	5287624
456.hmmer	13545379	13995581
471.omnetpp	13311132	8759539

为了更体现这个方案对读操作的优化，贴出对应的 write_latency(消耗在写操作上的 latency)：

trace	Baseline	Port_opt3
401.bzip2	8122938	4446452
403.gcc	3082209	1500108
410.bwaves	20524228	6753086
435.gromacs	239596	80423
450.soplex	4387910	2194463
453.povray	2232260	1061566
456.hmmer	9829862	6895495
471.omnetpp	2210979	1015324

同上，我们也看到了被疯狂优化的 trace（黄色标注），没错，它的读写比例很低（约 1:9）。同样，再看直接比较 write_latency 的表格，发现 write_latency 的优化幅度基本与之前总的 latency 优化类似，因此证明这里的优化确实基本来自于对于写操作的优化。

因此，这一方案对于写操作很多的 trace，优化效果明显。

总之，调整端口的排布、种类和个数、选择策略，可以产生一定的优化。从实验结果来看，针对 trace 的读写比例，加入尽可能多的端口可能会造成大幅度的优化。然而，我们认为这种方法比较暴力，同时具有一定的偶然性，再加上为了后期能更好地看出别的策略的优化，我们在之后的实验中只选择**4 个读写端口分别在 7、23、40、56 这种排布方案**作为基础，此版本以后简称 cache2.0。

3) Set 编址与划分("Set Initiation")

上一节中提出要将 set 按横向而非横向排布，又根据横跨 group 个数不同衍生出 4 种方案。

A. 横跨 1 个 group

在 cache2.0 基础上只改变这一点，运行实验 trace，与 cache2.0 的 latency 比较如下：(miss rate 没有变化，省略)

trace	Cache2.0	Span=1
401.bzip2	16822378	13066352
403.gcc	13051666	6672537
410.bwaves	24369220	39913446
435.gromacs	15306456	10155609
450.soplex	9300664	4335628
453.povray	7361707	2909676
456.hmmer	14374757	3261070
471.omnetpp	13043297	5193406

B. 横跨 2 个 group

在 cache2.0 基础上只改变这一点，运行实验 trace，与 cache2.0 的 latency 比较如下：(miss rate 没有变化，省略)

trace	Cache2.0	Span=2
401.bzip2	16822378	13613543
403.gcc	13051666	6666116
410.bwaves	24369220	43703642
435.gromacs	15306456	10136454
450.soplex	9300664	4788147
453.povray	7361707	2988140
456.hmmer	14374757	12864842
471.omnetpp	13043297	6501608

C. 横跨 4 个 group

在 cache2.0 基础上只改变这一点，运行实验 trace，与 cache2.0 的 latency 比较如下：(miss rate 没有变化，省略)

trace	Cache2.0	Span=4
401.bzip2	16822378	12980811
403.gcc	13051666	6470696
410.bwaves	24369220	40757792
435.gromacs	15306456	10127761
450.soplex	9300664	4748205
453.povray	7361707	3496155
456.hmmer	14374757	32420775
471.omnetpp	13043297	7942643

D. 横跨 8 个 group

在 cache2.0 基础上只改变这一点，运行实验 trace，与 cache2.0 的 latency 比较如下：(miss rate 没有变化，省略)

trace	Cache2.0	Span=8
401.bzip2	16822378	12158822
403.gcc	13051666	6421851
410.bwaves	24369220	33421275
435.gromacs	15306456	10110468
450.soplex	9300664	4895066
453.povray	7361707	3546576
456.hmmer	14374757	32334752
471.omnetpp	13043297	8322391

可以看到，除了个别 trace（黄色标注）以外，这四种方案相比原来的 set 编址方案要有一定的提高，这说明程序总体来说确实依照了局部性原理，短时间内会对相邻的 set 进行访问，从而使得横向放置减少了相应的移动开销。四种横向放置策略横向比较的话，除了个别 trace（绿色标注）以外，优化幅度大致相似，说明对一个 set 内部块的访问并没有特别强的局部性。如果非要分出个高低的话，方案 A，即只横跨 1 个 group 的方案平均 latency 最小，而且这个方案也方便后面优化策略的设计，因此在后面的实验中，我们以 **cache2.0+横向放置 set 且一个 set 横跨一个 group** 为基础再进行其他优化，以后简称此版本为 cache3.0。

4) Port 选择(“Port Selection”)

① 首先，加入小优化：原策略如果找到多个满足条件的端口，选取使得条带偏离原来位置最小的端口。

在 cache3.0 基础上只改变这一点，运行实验 trace，与 cache3.0 的 latency 比较如下：(miss rate 没有变化，省略)

trace	Cache3.0	Port_sel_opt1
401.bzip2	13066352	13062016
403.gcc	6672537	6679284
410.bwaves	39913446	39650282
435.gromacs	10155609	10137600
450.soplex	4335628	4315932
453.povray	2909676	2892624
456.hmmer	3261070	3367295
471.omnetpp	5193406	5416174

总的来看，还是能做到一些小幅度优化的，看来尽量使条带保持在原来位置还有些道理。

② 将 baseline 的端口选择策略进行更改，由原来的“动态取最短”改为“静态分配”。

在 cache3.0 基础上只改变这一点，运行实验 trace，与 cache3.0 的 latency 比较如下：(miss rate 没有变化，省略)

trace	Cache3.0	Port_sel_opt2
401.bzip2	13066352	13915897
403.gcc	6672537	7450286
410.bwaves	39913446	49947446
435.gromacs	10155609	10113385
450.soplex	4335628	5053663
453.povray	2909676	3204483
456.hmmer	3261070	3299139
471.omnetpp	5193406	5381909

令人遗憾，“静态分配”策略效果更差。其实也并不奇怪，像这个端口选择策略的效果本身就和其他因素（比如 set 编址）息息相关。“静态分配”策略是能够保证条带不会偏离初始位置太多，然而却牺牲了局部最优性。当 trace 的访问比较平均时（也许是大多数情况），那么这个策略只能带来劣势。总的来看，这个策略在这里实践是失败的，但也并非一无是处，例如，有些 trace（黄色标注）上确实有优化。

5) 指令预取(“Pre-shift”)

在一条指令执行时，根据历史信息预测下一条指令，并提前移动条带。

在 cache3.0 基础上只改变这一点，运行实验 trace，与 cache3.0 的 latency 比较如下：(miss rate 没有变化，省略；后两列分别表示进行预测的指令比例以及预测的正确比例)

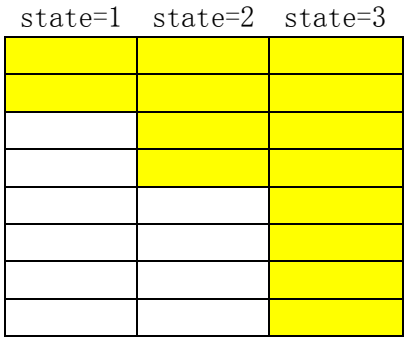
trace	Cache3.0	Pre-shift	Predict_ratio	Right_ratio
401.bzip2	13062016	13468393	49.1%	7.4%
403.gcc	6679284	5891977	59.3%	44.1%
410.bwaves	39650282	41846080	50.0%	10.4%
435.gromacs	10137600	5219943	66.1%	97.6%
450.soplex	4315932	4286612	65.2%	25.0%
453.povray	2892624	2551474	66.2%	44.9%
456.hmmer	3367295	3253979	95.0%	57.9%
471.omnetpp	5416174	4739939	75.1%	43.3%

可以看出，对于大部分 trace 而言得到了优化，道理很简单，就是预测正确了呗！从后面两列可以看出，得到优化的预测正确率较高，而变得更差的（黄色标注）的预测正确率确实十分可怜。不难想到，如果基本预测错了，虽说当次移动并不直接计入时间开销中，但条带的端口分布就被打乱了，很可能会偏离一边很多，这种情况可能会导致一些大的移动的出现。**Pre-shift** 总的来说是成功的，我们将其加入主线中，以后简称 cache4.0。

6) 条带回位(“Eager-shift”)+动态相连度(“Dynamic Associativity Control”)

正像在第二节讨论策略的原理写的那样，这两个策略是需要绑在一起才能看出来威力的，更为苛刻的是后者，不但需要“条带回位”策略保驾护航，而且阈值参数、状态转换时的动作的设计都需要依照端口排布、set 编址和划分的情况的定。还好，之前我们已经确定了端口排布、set 编址和划分，基于此，我们在原来思想的基础上，进行了改进调整。

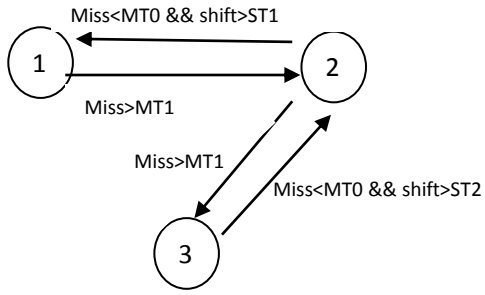
首先，为了不使 miss rate 变得太高，我们选择在一开始就给每个 set 开放两个块，后续可根据情况变为四个块、八个块，如图所示：



（图：表示一个 set，维护三个状态，黄色标注开放的块）

要注意的时，开放新的块时，要从离端口近的块开始；关闭块时，要从离端口远的块开始。这个思想在第二节中对应部分已经涉及过了。

另外，新的转换状态图如下：



其中，经过一些实验，将阈值分别定为 MT0=1, MT1=3, ST0=10, ST0=20。

在 cache4.0 基础上进行上述策略的实现，运行实验 trace，与 cache4.0 的 latency 比较如下：

trace	Cache 4.0	Eager-shift+DAC
401.bzip2	13468393	3622611
403.gcc	5891977	2003275
410.bwaves	41846080	10500209
435.gromacs	5219943	645819
450.soplex	4286612	1655538
453.povray	2551474	1372319
456.hmmer	3253979	10838793
471.omnetpp	4739939	1747205

这一策略必然增大 miss rate，我们也将 miss rate 进行了比较如下：

trace	Cache 4.0	Eager-shift+DAC
401.bzip2	2.86%	16.39%
403.gcc	4.29%	6.36%
410.bwaves	9.95%	10.18%
435.gromacs	0.63%	0.63%
450.soplex	1.46%	2.09%
453.povray	3.90%	6.35%
456.hmmer	0.17%	0.30%
471.omnetpp	0.39%	0.40%

我们先来看看 latency 方面，令人欣喜，这一策略果然厉害，可以看到，不少 trace 甚至遭遇到了数量级的优化。（关于出现了莫名的大幅度反弹情

况的 trace（黄色标注），我们心有不甘对此进行了研究，发现问题主要出在 Eager-shift 上。可以看到这个 trace 的 miss rate 极低，它总是反复对一些数据进行访问，结果就是条带每次归位后实际上又要访问归位前的地方，结果造成大量的不必要的浪费。这种特殊情况我们确实没有考虑到。我们对此 trace 去掉了 Eager-shift 策略，得到 latency 为 782588，也得到大幅优化，呵呵了。）

另一方面，Miss rate 会不会被提高很多呢？事实证明也没有。实际上，我们在阈值设定上是相当保守的，一旦 miss 超过 3，就开放更多的块，而只有当没有 miss 且条带累计移动较大时才会关闭一些块。

因此，这个策略在大幅减少 latency 的情况下没有导致 miss rate 的大幅提高，还是很成功的。在原来的基础上加入 **Eager-shift 和 DAC** 策略，形成了我们的最终版本 cache 5.0。

四、 实验结论

最后，我们将 baseline 和 cache5.0 进行对比，latency 比较如下：

trace	Baseline	Cache5.0
401.bzip2	17678095	3622611
403.gcc	13453357	2003275
410.bwaves	24408015	10500209
435.gromacs	15299748	645819
450.soplex	9670882	1655538
453.povray	7522577	1372319
456.hmmer	13545379	10838793
471.omnetpp	13311132	1747205

Miss rate 比较如下：

trace	Baseline	Cache5.0
401.bzip2	2.86%	16.39%
403.gcc	4.29%	6.36%
410.bwaves	9.95%	10.18%
435.gromacs	0.63%	0.63%
450.soplex	1.46%	2.09%
453.povray	3.90%	6.35%
456.hmmer	0.17%	0.30%
471.omnetpp	0.39%	0.40%

因此，总的来说，我们首先正确实现了 Baseline 模拟器功能，其次从多个方面一步步进行优化，最终在 miss rate 升高不多的情况下，大幅度降低了 latency（平均下来约为原来的 25%）。

注意到，我们选取的最终优化数据全部来自 cache5.0，前面也讨论过，对于有些 trace，5.0 版本并未最优，然而我们在本次实验中并非想一味地追求对于某些 trace 的 latency 优化，而是更注重优化策略的探寻与实践。我们最终形成了一整套策略实现在 5.0 中，是一个普适的版本。我们也就将此作为我们的最终版本了。

五、 反思与体会

很感谢老师和助教给我们精心布置了这样一个大作业题目。这个大作业和我们之前做的都不一样，以前基本上都是其他人做的很多的一些问题，做的时候基本会完全借鉴别人的想法，只不过自己再来实现一遍罢了。而这次对 RaceTrack Memory 的探索，全世界上已有的成果都很少，更没有一些所谓的正统的、成熟的方法，因此优化策略、实现细节等都需要我们自己琢磨、讨论。虽然这一过程注定伴随很多困难与艰辛，但是当我们完成时，心里很有成就感，感觉这是自己的东西。而且，我们觉得这次大作业对我们今后做自己做 research 有很大的帮助，很有意义。

再次感谢老师和助教！

六、 附录（源代码说明）

基本与实验报告中提到的版本号相对应：

cache1.0.cpp 是 baseline 的实现

cache2.0.cpp 加入 port 排布的所有优化

cache3.0.cpp 加入 set 划分和编址的所有优化，其中还有被注释的 port 选择优化

cache4.0.cpp 加入 pre-shift 优化

cache5.0.cpp 加入 eager-shift 和 DAC 优化，是最终版本