版本说明

版本号	修改说明	备注
V0.01	初始文档	由 Wang, Qi，Yang, Xi，Zhu, Yuhao 书写
V0.02		改正书中存在的若干问题，加入???

Contributors List

贡献者名单以字母排序，希望更多的人加入。

bbuy.tektalk	指出书中若干笔误，提出 2.4 节修改意见
Wei, Liu	指出书中多处笔误
Ma, Ling	指出 Sandy Bridge 的 Load 和 Store 的个数问题
Huang, He	与 Ma, Ling 在弯曲中讨论 LSU 的实现
Multithreaded	指出 Ref[34]和 Ref[96]的重复
Wang, Liang	指出书中若干笔误
Wang, Qi	V0.01 初始稿
Yang, Xi	V0.01 初始稿
Zhu, Yanhai	指出书中 Load Speculation 中的错误，和若干文字描述错误
Zhu, Yuhao	V0.01 初始稿
2.tektalk	指出书中若干笔误

Contents

序

近些年，我在阅读一些和处理器相关的论文与书籍，有很多些体会，留下了若干文字。其中还是有一片领域，我一直不愿意书写，这片领域是处理器系统中的 Cache Memory。我最后决定能够写下一段文字，不仅是为了这片领域，是我们这些人在受历史车轮的牵引，走向一个未知领域，所产生的一些质朴的想法。

待到动笔，总被德薄而位尊，知小而谋大，力少而任重，鲜不及矣打断。多次反复后，我几乎丢失了书写的兴趣。几个朋友间或劝说，不如将读过的经典文章列出来，有兴趣的可以去翻阅，没有兴趣的即便是写成中文也于事无补。我没有采纳这些建议，很多事情可以很多人去做，有些事情必须是有些人做。

这段文字起始于上半年，准备的时间更加久远些，收集翻译先驱的工作后，加入少许理解后逐步成文。这些文字并是留给自己的一片回忆。倘若有人从这片回忆中收益，是我意料之外的，我为这些意外为我的付出所欣慰。Cache Memory 很难用几十页字完成哪怕是一个简单的 Survey，我愿意去尝试却没有足够的能力。知其不可为而为之使得这篇文章有许多未知的结论，也缺乏必要的支撑数据。

在书写中，我不苛求近些年出现的话题，这些话题即便是提出者可能也只是抛砖引玉，最后的结果未知。很多内容需要经过较长时间的检验。即便是这些验证过的内容，我依然没有把握将其清晰地描述。这些不影响这段文字的完成。知识的积累是一个漫长的过程，是微小尘埃累积而得的汗牛充栋。再小的尘埃也不能轻易拂去。

这些想法鼓励我能够继续写下去。

熙和禺皓的加入使本篇提前完成。每次书写时我总会邀些人参与，之前出版的书籍也是如此，只是最后坚持下来只有自己。熙和禺皓的年纪并不大，却有着超越他们年纪的一颗坚持的心。与他们商讨问题时，总拿他们与多年前的自己对照，感叹着时代的进步。他们比当年的我强出很多。我希望看到这些。

个体是很难超越所处的时代，所以需要更多的人能够去做一些力所能及的，也许会对他人有益的事情。聚沙成塔后的合力如上善之水。因为这个原因，我们希望能有更多的人能够加入到 Contributors List，完善这篇与 Cache Memory 相关的文章。

Cache Memory 也被称为 Cache，是存储器子系统的组成部分，存放着程序经常使用的指令和数据，这只是 Cache 的传统定义。从广义的角度上看，Cache 是缓解访问延时的 Buffer，这些 Buffer 无处不在，只要存在着访问延时的系统，这些广义 Cache 就可以在掩盖访问延时的同时，尽可能地提高数据带宽。

在处理器系统设计中，广义 Cache 的身影随处可见。在一个系统设计中，快和慢是一个相对概念。与微架构(Microarchitecture)中的 L1/L2/L3 Cache 相比，DDR 是一个慢速设备，在磁盘 I/O 系统中却是快速设备。在磁盘 I/O 系统中，仍在使用 DDR 内存作为磁介质的 Cache。在一个微架构中，除了有 L1/L2/L3 Cache 之外，用于虚实地址转换的各级 TLB，在指令流水线中的 ROB，Register File，BTB，Reservation Station 也是一种 Cache。我们准备书写的 Cache，是狭义 Cache，是大家所熟悉的，围绕着处理器流水线和主存储器的 L1/L2/L3 Cache。这些 Cache 组成的层次结构，是微架构的设计核心。

广义 Cache 的设计可以在狭义的实现中获得帮助，却不是书中重点。在网络与存储这两个热点话题中，算法层面之外的重中之重是广义 Cache 的管理问题。与云相关的各类概念中，亟需解决的事情依然是算法与广义 Cache 管理。算法层面的实现需要考虑广义 Cache 的管理策略，反之亦然。广义 Cache 与狭义 Cache 系统没有质的区别。这些 Cache 系统都是由数据缓冲，连接缓冲的数据通路和控制逻辑这三个部分组成。

从算法角度上看，广义 Cache 的设计与实现比狭义 Cache 相比也许略微复杂一些；用实现的角度上看，狭义 Cache 的设计复杂度远远超过大多数广义 Cache。读者也许终其一生没有机会去体验狭义 Cache 系统的设计，依然可以从这些设计思想中受益。这些思想，可以应用于复杂的处理器系统中，可以解决一些细致入微的性能问题。系统开发者在不断思考探索的过程中，在挑战极限的奋斗中，在身旁最后一把利器寸寸折断后，杀虎屠龙。

我未曾想过书写一篇学术意义上完美的文章。我更愿意用工程师的语言完成写作，这并不阻碍本篇内容的有理可依。所有这些想法使我不堪重负，在整个处理器系统的设计中，几乎没有什么部件比 Cache Memory 系统更为复杂。

本篇书写的 Cache 系统以 Alpha，Power，UltraSPARC 和 x86 处理器为主线，这四个处理器目前属于 Tier 1。虽然 Alpha 处理器已退出历史舞台，但是并不影响其 Tier 1 的地位。我不会刻意去书写目前较为流行的嵌入式处理器，because yesterday's high-performance technologies are today's embedded technologies, but yesterday's embedded-systems issues are today's high-performance issues.

在 Tier 1 处理器中，本篇偏重于 Intel 的 x86 处理器实现，不管有多少资料引证 Power 和 UltraSPARC 处理器的诸多优点，x86 处理器依然是使用最为广阔，影响最为深远的处理器。单纯从结构上看，Intel 的 x86，即便是最新的 Sandy Bridge EP 处理器，也远未到达学术界意义上的完美。但是在 Cache Memory 层面，Intel 的领先是事实。

我最初曾试使用英文书写这些文字，我已经记不清楚最后一次抱着学习的目的去读中文技术类书籍是曾几何时，中文科技类书籍不能如此发展下去。我远没有书写英文小说的能力，依然有使用英文写科普文章的胆子，不用翻译的书写更加惬意。最终放弃了这个选择，因为英文世界里有这样的文章，因为 Cache Memory 之外的内容，因为中文世界需要有人去贡献一些勇气与智慧。

待到完成，总留有遗憾。我习得从这些遗憾中偷得间隙，留一片空间给自己。书不尽言，言不尽意总是无可奈何。这些文字很难给予我任何成就感，细看先驱的诸多著作后，留下的是何足道后的反思。我安于反思后的平静。

在阅读这些文字前，希望您能够仔细阅读 John 和 David 书写的“A Quantitative Approach”。我常备这本书，看了许多遍，字里行间内外的一些细节仍然不慎明了，写作时重温此书有了新的收获，借此重新审读了下列文字。

这篇文章最初的版本是 0.01，书名叫浅谈 Cache Memory。

第1章 有关 Cache 的思考

在现代处理器系统中，Cache Memory 处于 Memory Hierarchy 的最顶端，其下是主存储器和外部存储器。在一个现代处理器系统中，Cache 通常由多个层次组成，L1，L2 和 L3 Cache。CPU 进行数据访问将通过各级 Cache 后到达主存储器。如果 CPU 所访问的数据在 Cache 中命中，将不会访问主存储器，以缩短访问延时。

工艺的提高，使得主存储器的访问延时在持续缩短，访问带宽也在进一步的提高，但是依然无法与 CPU 的主频，内部总线的访问延时和带宽匹配。主存储器是一个不争气的孩子，不是如人们期望那般越来越快，是越变越胖。

主存储器膨胀的形体对 Cache Memory 提出了更高的要求，也进一步降低了主存储器所提供的带宽与访问延时之间的比率。近些年，单端信号所提供的数据传送带宽受到了各种制约，使得差分信号闪亮登场。差分信号的使用却进一步扩大了访问延时，对于这种现象，理论派亦无能为力，只是简单规定了一个公式 1-1，这只是一个无奈的选择。

$$\text{Latency Improvement}^2 \leq \text{Bandwidth Improvement} \qquad \text{公式 1-1}$$

从以上公式可以发现，延时在以平方增长，而带宽以线性增长。可以预计在不远的将来，CPU 访问主存储器的相对访问延时将进一步扩大，这是主存储器发展至今的现状，这使得在处理器设计时需要使用效率更高的 Cache Memory 系统去掩盖这些 Latency，也使得 Cache Memory 需要使用更多的层次结构以提高处理器的执行效率。在现代处理器中，一个任务的执行时间通常由两部分组成，CPU 运行时间和存储器访问延时，如公式 1-2 所示。

$$\text{Excution Time} = (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle time} \qquad \text{公式 1-2}$$

在一个处理器系统中，影响 CPU 运行时间的因素很多，即便是几千页的书籍也未必能够清晰地对其进行阐述。但是即便在一个可以容纳几千条指令的并行执行的 CPU 流水线中，采用更多提高 ILP(Instruction-Level Parallelism)的策略去进一步缩短 CPU 的运行时间，这些指令也必会因为存储器的访问延时被迫等待。

这使得如何缩短存储器访问时间受到更多的关注，合理使用 Cache 是降低存储器访问时间的有效途径。根据统计结果，在处理器执行一项任务时，在一段时间内通常会多次访问某段数据。这些任务通常具有时间局部性(Temporal Locality)和空间局部性(Spatial Locality)的特征，这使得 Cache 的引入顺理成章。这种局部性并不是程序天生具有的特性，是一个精心策划的结果。我们不排除有些蹩脚的程序时常对这两个 Locality 发起挑战。对于这样的程序，处理器微架构即便能够更加合理地安排 Cache 层次结构，使用更大的 Cache 也没有意义。

在一个任务的执行过程中，即便是同一条存储器指令在访问存储器时也可能会出现 Cache Hit 和 Cache Miss 两种情况。精确计算一个任务的每一次存储器访问时间是一件难以完成的任务，于是更多的人关注存储器平均访问时间 AMAT(Average Memory Access Time)，如公式 1-3 所示。

$$\text{Average Memory Access Time} = \text{Hit time} + \text{Miss rate} \times \text{Miss Penalty} \qquad \text{公式 1-3}$$

从公式 1-3 可以发现在一个处理器系统中，AMAT 的计算也许并不困难，只需要能够确定 Hit time，Miss Rate 和 Miss Penalty 这三个参数即可。但是如何才能才能确定这三个参数。

在一个程序的执行过程中，精确计算 Hit time，Miss Rate 和 Miss Penalty 这三个参数并不容易，即便将计算这些参数所需的环境进行了一轮又一轮的约束。近些年，我在面试一些 Candidates 的时候，通常只问他们一个问题，让他们简单描述，在任何一个他所熟悉的处理器中，一条存储器读写指令的执行全过程。我很清楚在短暂的面试时间内没有任何一个人能够说清楚这个问题，即便这个 Candidate 已经获得了处理器体系结构方向的博士学位。

我所失望的是参加面试的学生几乎全部忘记了这些可能在课堂上学过的，可能会使他们受益终身的基础内容。参加面试的学生们更多的是在其并不算长的硕士博士学习阶段，研究如何提高编程能力，和一些与操作系统具体实现相关的技巧。这些并不是学生们的错，有太多的评审官们本身就仅专注于小的技术和所谓的编程能力。

这些具体的编程能力和技巧，本不是一个学生应该在学校中练习的。在弥足珍贵的青春岁月中，学会的这些小技巧越多，这个学生在整个技术生涯中的 Potential 可能越低。重剑无锋，大巧不公。泱泱大国最为缺少的，不是能够书写程序，实现各类技巧的工程师。

1.1 Cache 不可不察也

在现代处理器中，Cache Hierarchy 一般由多级组成，处于 CPU 和主存储器之间，形成了一个层次结构，这个层次结构日趋复杂。Intel 甚至放弃使用阿拉伯字母对 Cache 的各级层次编号，而直接使用 LLC(Last-Level Cache)，MLC(Medium-Level Cache)这样的术语。

变化的称呼表明了一个事实，Cache 层次结构在整个处理器系统中愈发重要，也越发复杂。Sandy Bridge 处理器大约使用了十亿个晶体管，在其正中不再是传统的 CPU，是 Ring Bus 包裹着的最后一级 Cache [1]。

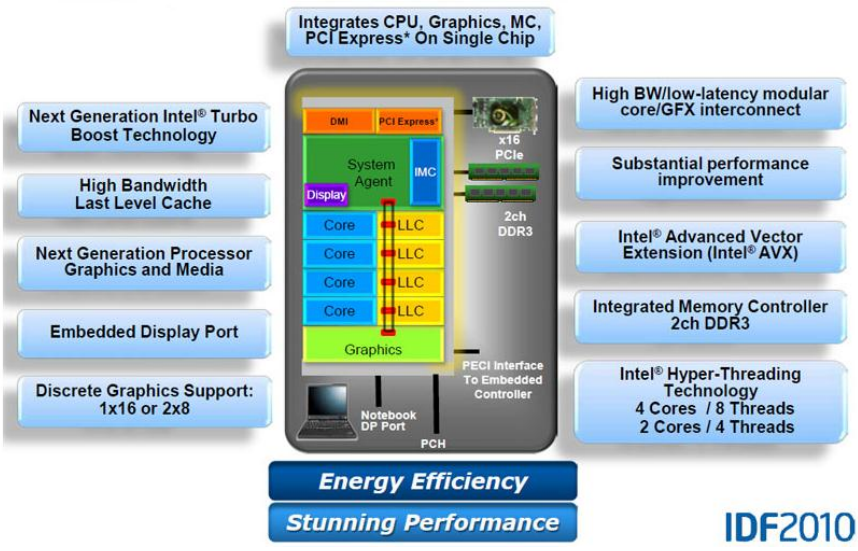

图 1-1 Sandy Bridge 的拓扑结构[1]

处理器的制作过程异常复杂。在人类历史上，其设计难度只有古埃及的金字塔可以与其媲美，即便是胡夫金字塔也只使用了 230 万个巨石，几十万个劳工而已。现代 CPU 的所耗的资源何止这些数字。在处理器这座金字塔中，Cache 层次结构是最基本的框架。

几千年前，孙子曾经说过，“兵者，国之大事，死生之地，存亡之道，不可不察也”。对于有志于站在金字塔顶峰的，即便目标只有半山腰的系统程序员，也是 Cache，不可不察也。在 Intel 的 Values->Discipline 中有一句话“Pay attention to detail”。

但是不要忘记 Devil is in the detail。准备深入理解 Cache 层次结构的读者需要时刻提醒自己真正了解什么是细节之后，才会重视细节，才能够避免因为忽视细节而引发的灾难。重视细节这个品质与你是否足够细心没有必然联系。

我们回到公式 1-3，简单探讨计算 Hit time，Miss Rate 和 Miss Penalty 这三个参数时所需要考虑的因素和相关的环境。

似乎 Hit Time 参数最容易获得。我们很快就可以从 CPU 的数据手册中找到各级 Cache Hit 后的访问时间，并从 L1 Cache 的访问时间开始计算 Hit Time。可能我们上来就错了，现代处理器大多使用了 Store-Load Forwarding 技术。存储器读操作首先要查询的并不是 L1 Cache，是在更前面执行的，还没有来得及提交的 Store 结果，这些结果保存在一段数据缓冲中，这个数据缓冲也是一种 Cache，不过比 L1 Cache 更加快速一些，也更接近 CPU。

除了数据 Cache，在现代处理器中，在指令 Cache 前还有一个 Line-Fill Buffer。Sandy Bridge 微架构中还含有一个 µops Cache[1]，计算指令 Cache 的 Hit 延时也没有想象中容易。精确计算指令与数据 Cache Hit 的延时需要注意很多细节。简而言之，在处理器系统存储层次中，L1 Cache 并不是最快的，也不是第一级。如果进一步考虑到 Load Speculation 使用的各类算法和命中率，Hit Time 参数并不容易计算清楚。

即便不考虑这些较为复杂的细节，我们仅从 L1 Cache 开始，Hit Time 参数也很难用简单的公式描述。在单处理器环境中，L1 Miss 后会逐级查找下级 Cache，直到主存储器。但是在多处理器内核环境中，情况复杂得多，一次存储器访问在自己的内核中没有命中，可能会在其他内核的 Cache 中命中，在其他内核的 Cache 中命中后，又存在数据如何传递，延时如何计算这些问题。说清楚这些问题并不容易。如果我们再进一步讨论多个 SMP 系统间 Cache 的一致性，这个 Hit Time 的计算就更加复杂。我只能选择放弃在这一节内，能够清楚地描述如何计算 Hit Time 这个参数。

Miss Rate 参数更加难以琢磨。我们真的可以用 Vtune，Perf 这样的工具精确计算出哪怕是单个任务的 Miss Rate 这样的参数吗，用这样的工具得到的统计数值有什么用途。同一片树叶，有的人一叶障目，有的人一叶知秋。不要为一叶障目而苦恼。多看几片后，必会发现春天的到来，也不要为一叶知秋而骄傲，少看几片，终会被最后一片树叶阻隔。

Miss Penalty 参数的计算仿佛容易一些。最糟糕的情况莫过于 CPU 从主存储器中获取数据。我们可以将环境进一步简化，以便于读者计算这个参数。我们可以不讨论 SMP 系统间的 Cache 一致性，甚至不讨论 SMP 之内的 Cache 一致性，仅讨论单处理器。即便如此 Miss Penalty 参数也不容易轻易计算，即便在这种情况之下，我们只讨论存储器读。

我们忽略在微架构在 Cache 前使用的各类 Queue，让存储器读操作首先对 L1 Cache 进行尝试。如果没有命中这级 Cache，这次数据访问一定可以到达 L2 Cache 吗，如果不是 L2 Cache，又是哪一级 Cache。这一切由 L1 和 L2 Cache 的关联结构决定。在一个处理器系统中，L1 与 L2 Cache 之间可能是 Inclusive，也可能是 Exclusive。如果是 Inclusive，存储器读操作将接着尝试 L2 Cache，如果不是将会跨越这级 Cache。事实并非如此简单，L1 与 L2 Cache 并不会直接相连，之间依然存在着许多 Buffer。

历经千辛万苦，数据访问最终到达最后一级 Cache，如果没有命中，就可以从主存储器中获得数据。在这种情况之下，我们仿佛可以计算出最恶劣情况之下的 Miss Penalty。但是这只是噩梦的开始。在现代处理器系统中，每一次存储器读写指令，都是由若干个步骤组成，这些步骤间具有相互联系，如果进一步考虑 Memory Consistency 层面，所涉及到的同步操作更多一些，这些操作并不能用几句话概括。

我们抛开这些复杂话题，讨论在 L1 和 L2 Cache Miss 之后从存储器获得数据这个模型。存储器读从存储器获得数据仅是一次读访问的步骤。从主存储器获得的宝贵数据不会轻易丢失，会存放在 Cache 中，需要将这些数据存放到哪一级 Cache 最为合理，LLC，MLCs 还是 FLC。

在一个正常运行的系统中，在每一个 Cache Block 中存放的数据都是有用的。新数据存放通常意味着旧数据的淘汰。值得思考的是如何进行这些淘汰操作，使用什么策略进行淘汰。从 L1 Cache 中淘汰的数据虽然暂时没有用途，但是不意味着可以轻易丢失，是否应该先进入到 L2 Cache 暂存。采用这种策略时，L2 Cache 也需要相应的进行淘汰操作。

从上文的描述可以看到，一个简单的存储器读访问带来了一系列的问题。我们首先需要为这次存储器读做基础的准备，然后进行真正的存储器读，读完成之后，还有复杂的扫尾工作。貌似容易计算的 Miss Penalty 参数即便在简化到了不能再简化的现代处理器系统中，也很难计算清楚。

我们还没有讨论存储器写对 Cache Block 的的污染与破坏，写操作可能会改变 Cache Block 的状态，使存储器读操作更加举步维艰，写操作还会带来很多 Bus Traffic，这些 Traffic 加大了存储器读的 Miss Penalty，我们没有讨论多处理器内核环境下的 Cache Coherence。

我们依然忽略了一个更加基本的细节，虚实地址转换。在现代操作系统中运行的任务，没有哪个任务可以直接使用 Physical Address(PA)，使用更多的是 Effective Address(EA)。在多数处理器系统中，EA 首先被转换为 Virtual Address(VA)，之后再转化为 PA。处理器微架构在更多的场景中直接使用的是 PA，不是 VA 更不是 EA。

虚拟化技术的引入，在略微有些复杂的 VA，PA 和 EA 的基础上又引入了 MPA(Machine Physical Address)和 GPA(Guest Physical Address)，带来了一系列地址 Mapping 机制，中断重定向等内容。虚拟化还带来了 IOMMU 和 I/O 虚拟化技术。为了能够在最小的篇幅完成这篇文章，我们忽略虚拟化技术，专注最基础的虚实地址转换。

1.2 伟大的变革

虚拟地址的出现可以追朔到上世纪六十年代的 Atlas 计算系统[2]。在当时 Atlas 计算系统是一个庞然大物，但也只有 96K 字节的内部存储器和 576K 字节的磁鼓作为外部存储器。我们很难深刻体会在计算机发展的初级阶段，计算机使用者的无奈。

当时的使用者可能身兼数职，首先是一个有钱人，不然根本没有机会去购买和使用计算机；然后是一个精巧的工匠，不过打孔技术恐怕已经失传；还必须是一个科学家，需要使用计算机；最后才可能是程序员。

Atlas 计算系统所提供的 96KB 物理地址空间很难满足程序员的需要。在当时程序员被迫显式地管理物理内存与磁鼓之间的数据交换，尽可能地利用外部存储器换入换出一些数据，以扩展物理内存地址空间。这些数据交换挫伤了程序员的编程热情。在这个大背景之下，Atlas 计算系统引入了 Virtual Memory，同时引入的还有分页机制。

技术的发展趋势惊人相似。在最具智慧的人解决了只有他能够解决的问题后，此后如潮水般涌入的人群爆发式地将其推至巅峰，等待下一位救世主的降临。虚拟地址出现之后迎来了这些变化。

在不到 40 年的时间里，虚拟存储技术遍及计算机系统的各个领域。从软件层面，多进程的引入顺理成章。与多进程相关的虚拟内存管理机制更加层出不穷，如 On-Demand 分配策略，COW(Copy On Write)策略等。从硬件层面，多线程处理器已被广泛接收，虚拟化技术更是软硬件层面的集大成者。这些变化已超出虚拟地址引入者的想象。这一切只是变化，或者是变化中的细节，终非变革。

虚拟地址的引入分离了程序员看到的地址和处理器使用的物理地址，设立了一个映射关系表存放虚拟地址与物理地址的映射关系，这个映射关系表也被称之为页表(Page Table)。最容易的想到的是使用主存储器存放这个映射关系表，但是没有程序能够忍受在使用虚拟地址访问一段物理空间时，首先需要从主存储器的页表中获得物理地址。

使用 TLB(Translation Lookaside Buffer)作为页表的缓冲是一个不错的想法，很快实现在各类处理器中。TLB 一般由多个 Entry 组成，不同处理器使用的 Entry 组成结构并不相同。下文以 Freescale 的 E500 内核为例简单介绍 TLB Entry 基本组成结构，如图 1-2 所示。

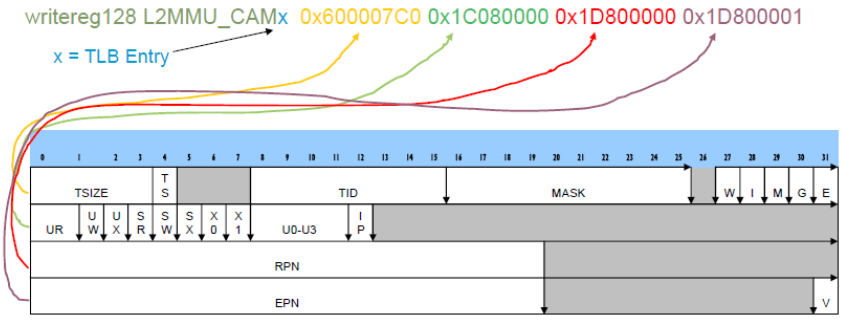

图 1-2 E500 TLB Entry 组成结构[3]

在 E500 内核的 TLB 中，一个 Entry 的必要组成部分包含 EPN(Effective Page Number)，RPN(Real Page Number)，TSIZE(窗口大小，通常为 4KB)。其中 EPN 与 TIS 和 TS 字段联合组成 VPN(Virtual Page Number)[4]；RPN 是物理地址基地址；TSIZE 记录映射窗口的大小，WIMGE 和 UWRX 为状态信息。

在一段程序访问存储器时，需要进行虚实地址转换。首先需要做的是将 EA 转换为 VA，这个过程因处理器而异，基本过程是 VA=f(EA)。函数 f 可繁可简，x86 处理器的处理方法较为复杂。E500 内核的做法较为简单，将 TS，TID 和 EA 级联即可。

在 CPU 得到 VA 后，将与 TLB 中所有 Entry 同时进行比较，如果 Hit 则获得 RPN，之后通过简单的计算，最终获得 PA；如果 Miss，就必须从 PTE 中进行查找，或者使用软件或者使用硬件手段。这些因为 Miss 引发的一系列操作会相大程度得影响 CPU 的执行效率。

TLB 最好永远不发生 Miss，这要求 TLB 需要 Cover 整个主存储器空间，硬件无法容纳这样大的 TLB。而且随着 TLB 的增大，其查找延时也越长，折中的选择是使用多级结构。TLB 也因此拆分为 L1 和 L2 TLB，与 Cache Hierarchy 的结构越发一致。TLB 也是一种广义 Cache。

多进程的频繁切换为 TLB 制造了不小的麻烦。在现代处理器系统中，每一个进程都使用各自独立的虚拟地址空间，进程的切换意味着虚拟地址空间的切换，也意味着 TLB 的刷新。进程的频繁切换导致 TLB 需要频繁预热，这个开销是难以接受的。这使得 PCID(Process Context Identifiers)的引入成为可能。

Intel 从 Westmere 处理器开始支持这一功能[5]。其原理是在 TLB Entry 中加入 PCID，并作为 VA 的一部分。使用该功能后，进程切换时，没有必要刷新 TLB，从而在一定程度上提高 TLB 的使用效率。E500 内核使用图 1-2 中的 TID 字段也可以实现同样的功能。AMD 的 Opteron 微架构使用 ASN(Address Space Number)[6]称呼这一功能。采用这种方法在减少 TLB 刷新操作的同时，进一步提高了函数 f 的复杂度和硬件负担。凡事有利有弊。

多线程处理器的引入再一次增加了 TLB 设计的负担。在多线程处理器中，存在多个逻辑 CPU。这几个逻辑 CPU 共享同一条流水线，却使用不同的虚拟地址空间，也需要使用 TLB 进行虚实地址转换，在绝大多数情况下，这些逻辑 CPU 共享 TLB，对进程切换带来的 TLB 刷新操作是 Zero-Tolerance。这使得 Logical Processor ID 也加入到 TLB Entry 之中。

TLB 经历若干变化后，其 Entry 结构日趋稳定，却迎来了更加严厉的挑战。因为主存储器的膨胀速度已经越发不可控制。程序对主存储器容量提出越来越高的要求。这使得主存储器容量几乎以每年 100%的速度膨胀，使得 TLB 的 Coverage Rate 在逐年降低，直接导致 TLB Miss Rate 的不断提高。

经典的教科书曾告诉我们，程序的 TLB Miss Rate 平均值仅为 5%左右，在某些情况之下不到 1%[7]。而近些年的研究表明，在很多应用中，TLB Miss Rate 为 30~60%[8]。这使得如何降低 TLB Miss Rate 重新受到关注。

增加 TLB 的 Coverage Rate 是降低 TLB Miss Rate 的有效手段，Coverage Rate 指 TLB 所能管理的存储器空间与主存储器容量的比值。近些年随着主存储器容量的不断扩大，TLB Miss Rate 在逐步降低。在主存储器容量不变的前提下，增加 TLB 的 Coverage Rate 有两个途径。一是增加 TLB 的 Entry 数目，这个数目已经在不断增加，依然无法与膨胀得更加快速的主存储器容量匹配；另外一种方法是增加一条 TLB Entry 所能 Cover 的 Size。

近期 Intel 的 x86 在 TLB Size 为 4K~4MB Hugepage 的基础上，提出了 1GB Superpage 的概念[5]。这一概念并非 x86 的发明，一些嵌入式处理器，如 Freescale 的 E500 内核，很早就使用 TLB1[4]支持 Superpage，使用 TLB0 支持常规页面。

增大的页面给操作系统带来了额外的负担。随着页面的增加，应用程序消耗的内存会相应增加，而且相应带来的换页开销也进一步增大。但是如果仅仅面对 4K~4MB 大小的页面，操作系统仍有能力找到通用策略，FreeBSD 从 7.0 版本起支持 4K~4MB 大小的 Hugepage，Linux 也从 2.6.23 开始为各类微架构提供 Hugepage 的支持。

真正带来挑战的是 1GB 之上的 Superpage。许多学者与工程人员试图寻求一些 Superpage 的通用管理策略[9][10][11]，依然难以解决由 Superpages 带来的 Allocation，Relocation，Promotion，Pollution 和 Fragmentation Control 等一系列问题。这使得这些所谓 Superpage 管理的通用方法几乎停留在纸面上或者实现中，很少有人直接使用操作系统提供的实现机制。

这使得更多的人开始认真思考操作系统是应该继续找寻通用解决方法，还是为专用化与定制化提供服务，是虽千万难吾独往矣，还是耐心等待着水到渠成。Intel 将 TLB Size 直接从 4MB 跨越为 1GB 的事实也在暗示着，操作系统需要进一步为应用让步，不再是全面接管，不再是继续制定放之四海而皆准的规则，而是让高效应用按照各自的轨迹前行，是为需要进一步优化的程序提供更大的空间。

Superpages 的引入极大降低了 TLB Miss Rate。还是有很多人发现 TLB 地址转换依然存在于存储器读写访问的关键路径上。在多数微架构中，一条存储器读指令，首先需要经过虚实地址转换，得到物理地址之后，才能通过若干级 Cache，最终与主存储器系统进行数据交换。如果存储器访问可以部分忽略 TLB 转换而直接访问 Cache，无疑可以缩短存储器访问在关键路径上的步骤，从而减少访问延时。Virtual Cache 为此而生，John 和 David 对其情有独钟，Virtual Cache 也在 MIPS 系列处理器中得到了大规模普及，在 Pentium 4，Opteron，Alpha21164 和有些 ARM 处理器中使用了 Virtual Cache。

采用 Virtual Cache 不是灵丹妙药，这种方法虽然缩短了存储器访问的关键路径，也带来了 Cache Synonym/Alias 这些问题，这些问题在 SMP 和 SSMP 系统中暴露出了更大的问题。解决这些问题更多需要考虑的是各种软硬件层面的权衡与取舍。

简单介绍虚实地址转换关系之后，我们首先需要关心在一个处理器系统中，存储器读写指令的执行过程。对此一无所知的读者，很难进一步理解 Cache 层次结构。也正是 Cache 层次结构的引入，加大了存储器读写指令执行的实现难度。

1.3 让指令飞

Superscalar 与 OOO(Out-of-order)的引入极大促进了现代处理器微架构的发展。已知的高性能处理器，如 Nehalem，Sandy Bridge，Opteron，Power 甚至是 ARM Cortex 系列处理器都使用了这种构架。这类方法在有效提高了 ILP(instruction level parallelism)的同时，加大了整个 Cache Memory 层次结构的实现难度。

在此我们只讨论存储器读写指令在 Superscalar 与 OOO 环境下的执行过程。存储器读写指令的执行过程似乎非常简单。即使是只写过几行汇编代码的程序员亦可对此娓娓道来。许多人认为存储器读不过是将数据从主存储器中将数据读入寄存器，存储器写是将寄存器中的数据写入到主存储器中。

这个执行过程很难用一句话回答，即便是将使用的处理器模型进行大规模的约束。在一个支持 Superscalar 和 OOO 的处理器中，一条指令的执行被分解为若干步骤。指令首先进入 Pipeline 的 Front-End，包括 Fetch 与 Decode，之后经过 Dispatch 和 Scheduler 后进入执行单元，最后 Commit 执行结果。

假设在一个微架构中，所有指令使用 In-Order 方式通过 Front-End，并采用 Out-of-Order 方式进行 Issue，之后使用 Out-of-Order Execution 和 Completion 方式，在最后进行 Commitment 时使用 In-order 的方式。其中指令 Commitment 的定义是在其执行完毕，并将最后结果更新至 ROB(re-order buffer)和 LSQ(Load-Store Queue)的过程。

现代处理器在 Commit 最后的执行结果时大多都采用 In-order 方式，这也保证了指令在经过 Out-of-Oder 的流水线后，程序员看到的最终结果与程序应有的顺序一致。多数程序员被这一假象迷惑，认为 CPU 的乱序执行仅与硬件流水线相关，并不会影响软件程序。

事实并非如此。微架构为了实现乱序执行，有些指令，比如存储器读指令，可能会提前执行，而后因为种种原因，如分支预测失败，可能会被迫重新执行。虽然乱序流水线可以保证最后的结果与程序期待的结果一致，但是无法完全抹去这条本不该执行的指令在流水线中，在存储器子系统中留下的执行痕迹。

为了进一步简化模型，我们仅讨论在经过这些约束后的 CPU 中，存储器读写指令的执行过程。与其他指令相比，这两条指令的执行过程更加显得步履蹒跚。下文以 Nehalem 微架构为参照说明存储器指令的执行过程。Nehalem 微架构 Pipeline 的组成结构如图 1-3 所示。

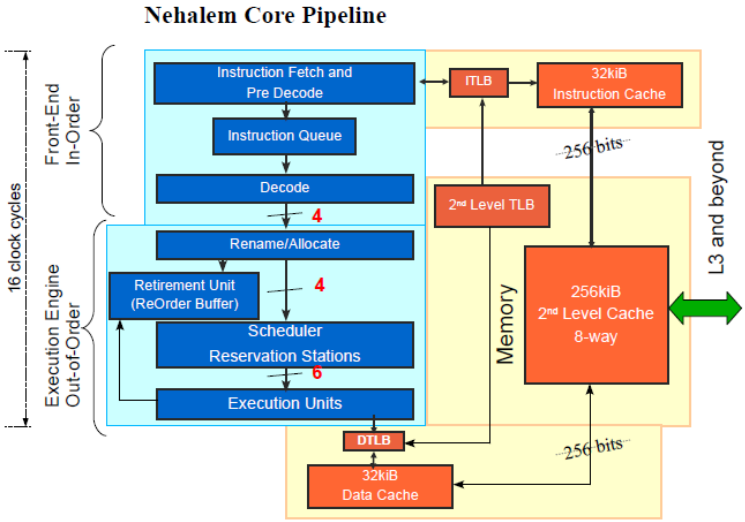

图 1-3 Nehalem 微架构 Pipeline 的组成结构[12]

存储器读写指令在经过 Front-End 阶段时进行了很多细节处理工作，尤其是对于 x86 处理器，此处不再对此做进一步的描述。这些存储器读写指令在经过 Front-End 之后，将首先通过 Rename/Allocate 部件，使用 Renaming 技术可以解决与存储器读写最直接相关的 WAW，WAR 相关，之后等待源 Operand 准备就绪，并将其 Dispatch 给 Scheduler。这些指令需要从 RS(Reservation Stations)获得可用的 Entry，对于存储器读写指令还需要从 LSQ 中预留空间，最后插上 ROB Tag 的翅膀，经 In-Order 或者 Out-of-Order 的发射(Issue)过程，自由飞翔。

这些飞翔的指令无序，而且指令流水线会让最笨的鸟尽可能的提前飞翔。存储器读写指令就是其中最笨的几只鸟。这些飞翔着的存储器读写指令将飞向第一个目的地 LSQ，还有一些执行信息会同步到 ROB 和 RS 的对应 Entry 中。

这些飞翔的指令有快有慢，有应该飞的也有不应该飞的。速度不同的指令必须在到达第一个终点 LSQ 时，按序等待提交。不应该飞的指令必须在最后的 Commitment 阶段之前被发现，然后被抛弃，重新飞翔。这也造成了在一条指令流水线中存在多个 Outstanding 的读和写指令，可并行的最大读写指令由 ROB，LSQ 和 RS 中的 Entry 数目决定。在很多现代处理器中，LSQ 的 Entry 数目多小于 ROB 和 RS 中的 Entry 数目，因此在一个 Pipeline 中可以并发的读写指令首先由 LSQ 的深度决定。

无序飞翔存储器读写指令在执行过程中，需要尽量无视对方的存在，最大可能的实现飞翔。因此也引入了读写指令的 Speculation 机制。由于 Out-of-Order Issue 的原因，后续的指令可能先于之前的指令执行，由于 Out-of-Order Execution 和 Out-of-Completion 的原因，率先发射的指令也不一定最先抵达目的地。

这些机制制造了多种乱序的可能。虽然指令的最终结果仍然是 In-Order Commitment，但是这一机制并不能保证存储器指令的执行轨迹是 in-order 的。存储器指令的执行与其执行轨迹的异步引出了一个异常沉重的话题 Memory Consistency。这个话题将有专门的篇章讨论。

我们首先分析存储器读写指令的执行过程。这两大类指令都是访问存储器的，虽然一个是进行写，另一个是进行读，但是依然有其相同点。我们首先讨论其相同点。在一个 CPU 中，读写指令在进入 Pipeline 之前，首先被分解为两个微步骤或者是两个微指令，这并不是 x86 处理器所特有的，许多为了提高存储指令执行效率的微架构都使用了这种方式。

其中一条微指令用来计算指令使用的 EA。在有些处理器微架构中，每一条 Load/Store 指令在其之前的 Store 指令的 EA 计算完毕后才能发射，这一机制有效解决了存储指令间的 RAW 类相关，但是这种方式较为 Conservative。激进的想法是先做，错了再纠正，只要错误带来的惩罚小于做对了所带来的收益即是一次有效行为。

在现代处理器微架构中，虽然程序员直接使用 EA，但是对于存储器读写指令，由于各类寻址方式的引入，Pipeline 并不会很顺利地得到最终 EA。在 x86 处理器中，EA 的计算公式如公式 1-4 所示。

$$\text{Effective Address} = \text{Base} + \text{Index} \ll \text{Scale} + \text{Disp} + \text{Segment} \qquad \text{公式 1-4}$$

这样的公式显然并不容易很快的计算出结果，这使得现代处理器多设置了 AGU(Address Generate Unit)这个专门的执行部件。AGU 部件在计算出指令使用的 EA 后，会将其传递给在 LSQ 中对应的存储器读写微指令，之后这些微指令才开始真正的存储器操作。

在流水线中指令的执行过程很难用一句话说清楚。权衡读写指令复杂度后，我们决定先讨论存储器写指令，这条指令的执行过程与读有类似之处，也有很多区别。存储器写指令的第一个目的地是 LSQ 的对应 Entry，这条指令在获得源 Operand 和 EA 后，将进行存储器写。在没有描述清楚 Cache 层次结构之前，我们无法说明存储器写如何在流水线中执行。为未来预留说明空间，我们暂时简化存储器写的执行过程。

任何一个处理器体系结构都会谨慎地处理存储器写指令的执行过程。设计者都明白一个基本道理，如果你向一个指定的存储器写入一个指定的数据后，你很难用常规的手段重新获得其历史信息。写是不能悔棋的，即便其目标是一个 Well-Behavior 的存储器。这些谨慎创造了存储器写这个胆小鬼。流水线中执行的存储器读和写指令，其胆量的差别是质的，存储器读指令在没有按序轮到其执行之前已经周游了大千世界，而存储器写指令在执行完毕那一瞬间甚至还没有离开过家门。

在多数现代处理器微架构中，存储器写指令只有在最终 Commit 之后才能真正向 Cache 层次结构发送数据。这并不意味着存储器写指令不能 Speculative，存储器写在 Commit 之前也如存储器读一样自由，而且在多数情况下写操作也需要首先读取数据，之后进行数据合并，然后进行真正的写操作。存储器写指令在最后的 Commit 阶段时异常谨慎，只有在确保一定不会出现问题之后才能提交。这一机制保证了存储器写操作在指令流水线中可以按序完成，也导致了存储器写的延时。

对于一个最终存储器类设备，在流水线中执行的存储器写操作所采用的这种方法依然不能保证到达 DDR 颗粒或者 PCIe 设备时，存储器写操作依然会按序到达。一次存储器写操作的轨迹很长，一部分在 CPU 域中，一部分会在设备域。按序 Commit 与各类 Barrier 指令只能保证在 CPU 域的序。当存储器访问到达其他设备域时，需要遵守其他序规则。这些内容超出了在这里的讨论范围。

我们重新讨论存储器写的延时。正如大家所知，存储器写可以 Posted，原本可以较快的执行完毕，只是由于 Speculative 的原因，存储器写操作很有可能会被错误地提前执行，只是在这条存储器写指令没有 Commit 之前，都有修改的余地。这个余地带来了写的延时。

这个延时在一个追求极致的微架构中是不能忍受的。最具智慧的一群人在有资格一起进行极限挑战时，差距非常微小。在这些人眼中，处理器的一个 Cycle 已经被细化为十分之一，甚至百分之一。两个追求极致的 CPU，指令平均执行效率如果相差了一个 Cycle，意味着这两个 CPU 本不应该在同一个舞台上竞争。

必须要最大程度地利用这个延时，以提高存储器读的效率。与存储器写相比，存储器读非常幸运。在正常情况下，对于 Well-Behavior 存储器的某个地址读一千次一万次，也不能将其从 0 读成 1。读操作可以更大胆一些。

存储器读指令获得 EA 后，首先需要访问的是 LSQ，检查是否在尚未 Commit 的写操作中是否具有数据副本，从而消除了存储器写带来的延时。在 LSQ 中没有命中时，存储器读指令才会访问 Cache 层次结构，并将结果放入对应的 LSQ 等待提交。在访问 Cache 层次结构时，这些读指令有快有慢，这造成了存储器读的乱序。更为糟糕的是，有些读操作可能不应该被提前执行。虽然这个存储器读的结果可以被指令流水线最终丢弃，但其执行轨迹却无法消除。

这是因为 Load/Store Speculation，也引出了数据访问序这些概念。很多系统架构师熟悉这些序，也在有意无意的利用着这些序。下文讲述的这个实例发生在在一个多处理器并行编程系统中。在这个系统中，存在 C1 和 C2 这两个 CPU，并使用共享存储器进行通信。

C1 首先向 A1 地址写入一个命令，之后将 A2 地址的数据加 1。C2 使用存储器读操作获得 A2 的数据后，首先进行数据检查，发现数据变化后，得知 A1 中已经存放了新的命令，之后再处理这个命令，其源代码示意如图 1-4 所示。

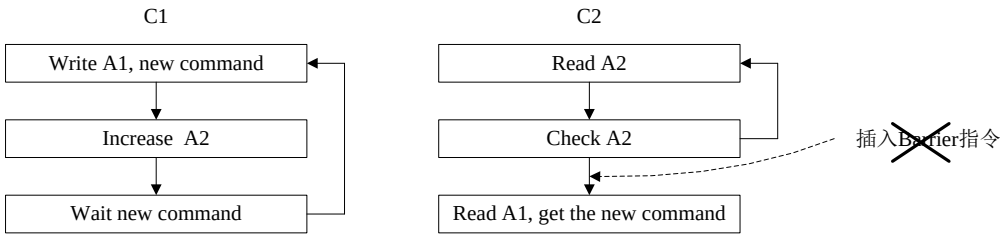

图 1-4 C1 与 C2 的数据交换

这样的代码在支持 Load/Store Speculation 的微架构中，有很多问题。一个简单的想法是 C2 的 Load Speculation 可能导致 Read A1 得到的命令并不是最新的。但是 C2 在何种情况之下才能得到旧的数据，并不是简单说说 Load Speculation 这样容易。

曾经有一个朋友与我说，买卖股票其实很容易，Just buy low and sell high，But how 难住了他。Load Speculation 引发灾难的各种可能性也很难用一两句话描述清楚。此处仅以图 1-4 说明一个简单的可能出现错误的场景。

假定 C1 没有出现问题，对 A1 和 A2 这两个地址的存储器写访问按序进行。与此同时，C2 采用 Load Speculation 技术以加速存储器读，但是 Read A2 和 Read A1 这两个操作最终依然按序完成。只是 C2 在访问 A2 时，发生了 Cache Miss，经过很长一段时间之后，假设这段延时为 m，C2 才能获得真正的数据，而 Speculative Read A1 读取的数据在 Cache 中命中，C2 将其放入 LSQ 中，等待最后的提交。

如果 C2 最终从 A2 中的获得数据没有改变(Check A2)，则重新读取 A2 重新判断，在 LSQ 中的 A1 将被抛弃。Speculative Read A1 的结果会被抛弃，并没有什么问题。只有在某个机缘巧合之下，Speculative Read A1 的结果才会错误，这个概率非常低。

假设在延时 m 这个时间段里，C1 更新了 A1 和 A2 并对于 C2 按序到达。只是在 C1 更新 A1 之前，C2 已经进行完成 Speculative Read A1，此时 C2 得到的是 A1 的一个历史值。C2 在延时 m 内最终获得了 A2，这个值已经被 C1 更新，因此 C2 将获得更新后的 A2，此时分支预测单元将误认为 C2 的 Speculative Read A1 的结果正确，不会重新对 A1 进行访问。此时造成了 C2 获得了一个最新的 A2 和旧的 A1，引发了一个致命错误。

有很多方法解决这个错误，只需要在 Check A2 之后，显示加入 Barrier 类指令，避免 Load Speculation，或者直接设置 MMU 禁止对这段存储器访问的 Load Speculation。虽然只有在某个机缘巧合之下，Speculative Read A1 的结果才会错误，而且这个概率非常低，但是我们仍然需要保证这段程序在执行过程中 100%正确。问题是我们要为不到 1%的概率，将如此沉重的 Barrier 操作强加到 99%以上的每一个正确之上。

这个 Barrier 并非不可移除。只要 C2 发现 Read A1 命中 Cache 之后，将其之前出现的 Read Miss 读操作加上 Snoop Resync Flag 的标志。标志为 Snoop Resync Flag 的读指令在 Commit 结果时，抛弃其后的读操作，就能避免这些沉重的 Barrier 操作。在这个例子中，使用这样机制后，Speculative Read A1 的结果将被流水线丢弃，之后重新执行，沉重的 Barrier 操作得以取消。AMD Opteron 使用了这种机制[13]，类似的机制也出现在其他现代处理器中。

有心人会从各类处理器的优化手册和这些公司近十年来发表的 IP，猜测出这些处理器实现的某些细节，即便这些公司敝帚自珍一些更重要的细节，从不发表这些细节的 IP。敝帚自珍总是暂时，时间将冲破任何障碍。美国中央情报局称，早在 RSA 算法发布之前，他们已经掌握了公钥体制，只是没有公开。他们的孤芳自赏没有改变这个体制的如期而至，只是成就了 Rivest，Shamir 和 Adleman 的不世之功，传世之名。

不想继续探讨有关 IP 的深重话题，但是总有要说的话。IP 自有其存在的道理。羊儿有羊儿的道理，狼儿有狼儿的道理。IP 以保护知识产权的名义诞生，历史越是悠久的公司，IP 的数量也更加多些。在很多时候，我们并不知晓有些 IP 是如何审批通过。某些行业的某些 IP 有如“天是蓝的，草是绿的”这样的常识。这些 IP 不因我们称其无耻而消亡，这些 IP 的发源地很多是来自某些如雷贯顶的巨型公司。

IP 制度最初是为了保护创新，发展至今总能找出其妨碍创新的实例。很多时候真正的创新产生于悲寒交迫饥饿着的人群中，不发生在衣食无忧领取薪水，悠哉游哉喝着咖啡聊着天的白领中。这使得很多历史悠久的公司为真正创新所做出的努力反而不如在车库中工作着的年轻人，这是一个客观存在的事实。

这些年轻人历千辛万苦的成果，却能很轻易地被“天是蓝的，草是绿的”这样的 IP 拒之门外。深入思考这些问题后剩余是无奈后的反思。天将是蓝的，草依旧是绿的。人类的生存与发展，需要的是不断进取所带来的创新，这是世界大行，是再多的魍魉魑魅，再多的阴谋诡计无法阻挡的。

1.4 Crime and Punishment

很多时候，一个架构师选择 Load/Store Speculation 的终极方法是掷硬币，只是在用一只很有技巧的手去投掷这个硬币。这些猜测是无限追求完美的人群，在屈服于最终的命运安排之后使用的赌徒方式。有人质疑这种掷硬币和闭着眼睛猜有什么区别。闭着眼睛猜确实是一种办法，只是当你睁开双眼发现迷失后，知道归时之路。

Load/Store Speculation 的结果可能正确，也可能错误。如果最终结果是正确的，是一次成功的投机，如果错误会带来一定的惩罚。如果一次投机将导致不可收拾的系统惩罚，其最终结果不如不进行这些猜测。片面追求投机而忽略惩罚只是莽夫所为。合理的预测执行与失败后可以承担的惩罚是一个大的权衡。投机是人类与生俱来的。在投机的成功率越高，相应的惩罚越低时，这个天性就更加容易暴露。

Speculation 策略需要在成功率和惩罚间进行取舍。让我无限制的悔棋，从理论上说我可以战胜一切对手。只是这个对手如果是李昌镐，至死我也下不完一盘棋，所以我将目标设为悔棋十步，去挑战那位每次只能胜我一目半目的邻居。采用十步悔棋规则后，那位邻居每次战至中盘，即与我签订城下之盟。

溯本求源，我们重新讨论为什么会出现 Load/Store Speculation。为简化起见，我们仅在此讨论 Load Speculation 而忽略 Store Speculation。在现代处理器系统中，存储器 Load 请求所需的 Latency 相对于 CPU 的主频在不断提高，使得存储器瓶颈问题更加突出一些。使用 Load Speculation 的主要原因是为了掩盖这些 Load Latency，尽量的让执行延时较长的存储器读指令笨鸟先飞早入林。如何决定让那只笨鸟先飞是一个较为复杂的预测过程。在讲述这些预测之前，我们首先讨论这些预测的基本实现机制和预测失败的后继处理方法。

我们简单回顾 Confidence Counter 机制，Confidence Counter 是一种常用的，判断是否应该进行预测的加权处理方式。除了 Load/Store Speculation 实现之外，Confidence Counter 也广泛应用于 Branch Prediction 领域，是一种已经得到证明的，行之有效的方式。其实现机制与 N-bit Saturating Counter(Bimodal Predictor)类似。Confidence Counter 由 Saturation，Predict Threshold，Misprediction Penalty 和 Increment for Correct Prediction 四个参数[14][15]组成。

我们以{31(Saturation), 30(Threshold), 15(Penalty), 1(Increment)}为例简要说明 Confidence Counter 的使用方法。假设在一个应用中，Confidence Counter 的初值为 29，此时指令流水线将不会进行 Load Speculation 操作。如果指令流水线发现执行的最终结果为真时，Confidence Counter 将加 1(Increment)；当 Confidence Counter 的值等于或者超过 Threshold 时，指令流水线开始进行 Load Speculation；如果 Confidence Counter 的值为 31(Saturation)时，结果为真时也保持不变；如果预测失败后，Confidence Counter 将一次减去 15(Penalty)，直到逐步加 1 到达 Threshold 后才能触发 Load Speculation。

在 Branch Prediction 中也使用了 Confidence Counter 机制。假设在一个分支预测系统中，使用 2-b Confidence Counter，而且 Strongly Taken, Weekly Taken, Weekly not Taken 和 Strongly not Taken 这状态位为 3，2，1 和 0 时，Taken 路径使用的 Confidence Counter 为{3, 2, 1, 1}，Not Taken 路径使用的 Confidence Counter 为{0, 1, -1, -1}。

大多数 Load Prediction 算法使用 Confidence Counter 作为是否进行预测的基本分析工具。在 Load Speculation 失败后，指令流水线大多使用两种机制进行恢复操作，分别为 Squash 和 Reexecution 机制[14][15]。

Squash 指当某条 Load 指令预测失败后，将 ROB 中在其之后的指令抛弃，并重新从指令 Cache 中 Fetch 指令。这种方式与 BTB 预测失败后采取的方式类似。Reexecution 指当某条 Load 指令预测失败后，仅仅重新执行与此指令直接或者间接相关的指令。

从直觉上看，采用 Reexecution 比 Squash 机制的效率高出许多。但是我们依然不能依次得出 Reexecution 一定优于 Squash 机制的结论。在体系结构设计中，更多要考虑的依然是权衡。Reexecution 机制需要使用更多的逻辑和数据缓冲也是不争的事实。

在已知的指令流水线设计中，Load Speculation 主要有四种算法实现，分别是 Dependency Prediction，Address Prediction，Value Prediction 和 Memory Renaming。这些 Prediction 的实现机理依然是猜测，并借用 Confidence Counter 机制和以往的经验尝试。Load Speculation 与 Brach Prediction 的实现有类似之处，本质上是一个机器的自学习过程。

其中 Dependence Prediction 算法的实现机制较为简单。对于现代处理器，能够精确判断存储器读写指令间的依赖关系至关重要。在一些支持读写指令的乱序执行的微架构中，在这些处理器中通常使用 LSQ 支持多条存储器读写执行的并发执行，以提高指令执行的并行度。为确保正确处理读写指令间依赖关系，万无一失的方法是每一条 Load 指令在乱序发射之前，遍历在 LSQ 中的 Store 指令，确保与其不发生依赖关系。如果在 LSQ 没有发现依赖关系，这条 Load 指令才可以被乱序执行。

采用这种方法虽然保证了存储器读写指令的正确执行，却带来了较大的系统延时。Store 指令在执行到一定阶段时，Load 指令才能精确地判断这种相关性，在 LSQ 中的 Store 指令并不都是准备就绪的。Dependence Prediction 算法就是为了取消这类等待延时的一种方法。该算法有 Blind Prediction，Wait Dependency Predictor 和 Store Sets 等实现策略。

最简单的策略莫过于 Blind Prediction。所谓 Blind Prediction，指 Load 指令仅检查在 LSQ 中准备就绪的 Store 指令，如果没有发生相关性，即可提前执行，如果在 LSQ 发现了 Store Alias，则直接从 Store Queue 中获取数据。对于没有就绪的 Store 指令，Blind Prediction 认为不存在相关性。如果最后的执行结果证明确实两者之间存在相关性，被错误执行的 Load 指令，将采用 Squash 或者 Reexecution 方式恢复。

Alpha 21264 采用了 Wait Dependency Predictor 策略[18]进行 Load Speculation。其实现机制是在指令 Cache 中的每一条指令加入一个 Wait 位，当 Load 指令的 EA 计算完毕后，而且相应的 Wait 位为 0 时，这条 Load 指令可以 Speculation，否则需要等待。当发生预测失败后，相应的 Wait 位需要置 1 以避免进一步的预测失败。全部 Wait 位在 100,000 个 Cycle 后将自动清零，以避免因为多次预测失败造成很多 Wait 位都置 1 的情况发生。当发生指令 Cache Miss 的时候，Wait 位将清零。

Store Set 是另一种 Dependence Prediction 实现策略。其实现机制是使用 Store Set ID 连接对同一个地址的存储器读写访问，并将这些 ID 存放在 SSIT(Store Set Identifier Table)中。在存储器读写指令预取之后，使用 PC 作为 Index 索引 SSIT 后获得对应的 ID，这个 ID 指向另一张表 LFST(Last Fetched Store Table)。在 LFST 中记录了已经访问相同地址的 Store 指令，并使用这种方法判断读写指令之间的依赖关系，没有发现依赖关系的 Load 指令，将可以进行后续的 Speculation。

以上这几种策略的相同点是利用读写指令间存在的依赖关系，提高 Load Prediction 的预测成功率。使用这类方法时，存储器读写指令都需要获得 EA 后，才能进行进行相应的预测。采用 Address Prediction 算法更加 Aggressive 一些，这类算法的本质是在存储器读写指令计算 EA 之前，预测这个 EA 的值，从而实现计算 EA 与实际访问存储器操作的并行执行。处于 Critical Patch 的存储器访问操作，其 EA 不但可以预测，而且成功率很高。

Address Prediction 算法的常用实现策略有 LVP(Last Value Prediction)，Stride 和 Context。使用 LVP 策略时，下一次存储器访问的预测地址是上一次刚刚访问过的地址；使用 Stride 策略时，预测地址是上一次刚刚访问过的地址加上某个偏移；Context 策略更具智慧，采用这一策略时，可以利用存储器访问过的历史信息 VHT，形成一个 Pattern，之后计算出一个合适的值索引另一张表 VPT，以获得策略的地址，这一方法与 BTB 中常用的 gshare 机制类似。

对于不同的应用，LVP，Stride 和 Context 策略的预测成功率并不相同，Context 策略貌似最优，但依然存在一些应用使用 LVP 和 Stride 策略预测成功率更高一些。因此在一个微架构的具体实现中可以使用不同权值的 Confidence Counter 混合使用 LVP，Stride 和 Context 策略，以获得更优的结果。

Value Prediction 算法与 Address Prediction 算法的实现策略类似。其不同之处是一个预测的是将要使用的地址，一个预测的是最终数据。显然 Value Prediction 算法更加 Aggressive 一些。Value Prediction 与 Address Prediction 的实现策略较为一致，也使用了 LVP，Stride 和 Context 策略，策略的实现方法也几乎一致。

Value Prediction 算法并不能避免存储器读操作最终穿越 Cache Hierarchy，并不能避免任何 Bus Traffic 的出现。指令流水在获得数据之后，需要进行 Check-Load 操作，确定是否发生了 Misprediction，但是采用这种算法使得指令流水可以更早的获得 Load 操作的结果，从而使其后的指令可以在猜测中继续执行。

在存储器瓶颈愈发严重的今天，Value Prediction 算法曾被多次提及。Andy Glew 有一个非常 Crazy 非常大胆的想法，他设想了一个可以容纳几千条并发执行的单发射的微架构，使用各类耸人听闻的技术，包括 Value Prediction 进一步优化 MLP(Memory-Level Parallelism)[22]。以个人浅见，使用一个更大规模，比如几百兆的最后一级 Cache 也比这些想法有实现的可能。当芯片制作工艺到达 10nm 时，处理器系统包含一个几百兆的 EDRAM 应该不是什么难事。可能在 14nm 工艺之下就可以实现这样容量的 LLC。

最后需要说明的是 Memory Renaming 算法。研究发现 Store 和 Load 指令对存储器的访问可以由硬件精确预测[19][21]，使用 Confidence Counter 即可方便地实现这一目标。当 Store 和 Load 指令建立了确定的关系之后，Load 指令不必每一次都从存储器子系统中获得数据，因为 Store 指令可以将数据直接送至建立映射关系的 Load 指令。这一方法相当于指令流水将已经与 Store 指令确立关系的 Load 指令的存储器器访问，转化为对 LSQ 进行访问，这也是 Renaming 的由来。Memory Renaming 技术也有一些具体的实现策略，而且这些策略依然可以混合使用。在此不再一一述说。

在一个微架构中可以同时使用这四大类算法，也可以只使用部分。在这四大类算法中，都有各自的实现策略，这些策略可以混合使用。Confidence Counter 使用的不同参数也决定了预测的成功率。预测失败的恢复手段还有两种选择。

这使得在一个 Load/Store Speculation 设计中有非常多的选择。在其他领域的实现中，系统设计者也会遇到许多选择。过多的选择经常令人无所适从。系统设计者很难从公说公有理婆说婆有理的，甚至有意无意去误导的各类量化分析结果中得出精确结论，做出最正确的选择。他们最后的选择是去掷硬币[①]。

“掷硬币”是面对过多选择的无奈。一个是非的结果很难只有是与非这样简单。有些选择是单纯的也是幸运得几乎不存在的，他们一次就认定了是与非。更多的是经历了大是大非，在是是非非中反复后的是与非。是可以倾听自己心跳声的人绞尽最后一滴脑汁后的感觉。

许多设计都存在这样的是非，即便这些设计出自最权威系统架构师之手。这些不可知引发了一个讨论，是否有哪怕是特定应用之下，最优微架构的存在。我们很难确定在一个具体的实现中，$O(n\times Log_2 n)$一定优于 $O(N^2)$，在一个实现中 N 是有限的，不是理论上的无限，而且不同的算法使用的实现时间并不相同。

即便使用相同的算法，具体实现依然有优劣之分。我有些机会体验过这些糟糕的实现。这些实现不仅出现在入门级别的公司，也出现在国内顶级的公司。最初对这些实现是惊讶的，在习以为常后只剩下是无奈。而后发现就是这些实现战胜了由常青藤组成的个体，团队，帝国。明白这些人更加懂得，什么是权衡，什么是掣肘，什么是地处中国的不得已。

[①] 国内的许多公司也将这个方法称为拍脑门。

这一切超越了技术层面，或者说技术层面的权衡源自于此。另一方面，我想说这样获得的成功很难让整个世界信服。有过资治通鉴的国家在人与人的争斗，人与人的相互排挤与贬低上，哪怕是少费一点点时间，整个世界也会因此更加精彩。

有些仿佛只能出现在金庸小说中称着洪教主福如东海，寿比南山这样的言语与行为，也确确实实的发生着。黑压压着一片的人群，很多行为仿佛自己是这个星球的最后一代，疯狂着掠夺。任何美丽在此面前都荡然无存。

完成了第一章的书写后，我几乎准备放弃。不是因为我已经写完了我想书写的内容，而是不知道如何继续。简单罗列好目录后，能够约定的只有终稿的日期，写到那时便是结束，没有刻意的规划。

第2章 Cache 的基础知识

很多程序员认为 Cache 是透明的，处理器可以很聪明地安排他们书写的程序。他们非常幸运，可以安逸着忽略 Cache，也安逸着被 Cache 忽略，日复一日，年复一年，机械地生产着各类代码。All of them are deceived。

貌似并不存在的 Cache，有意无意地制造了，正在制造，并必会制造着各类陷阱。也许在历经了各类苦难后，有些人能够发现 Cache 的少些特性，却愈发不可控制。掌握 Cache 的细节知识并不困难，只要能够真正做到静如止水。

2.1 Cache 的工作原理

处理器微架构访问 Cache 的方法与访问主存储器有类似之处。主存储器使用地址编码方式，微架构可以地址寻址方式访问这些存储器。Cache 也使用了类似的地址编码方式，微架构也是使用这些地址操纵着各级 Cache，可以将数据写入 Cache，也可以从 Cache 中读出内容。只是这一切微架构针对 Cache 的操作并不是简单的地址访问操作。为简化起见，我们忽略各类 Virtual Cache，讨论最基础的 Cache 访问操作，并借此讨论 CPU 如何使用 TLB 完成虚实地址转换，最终完成对 Cache 的读写操作。

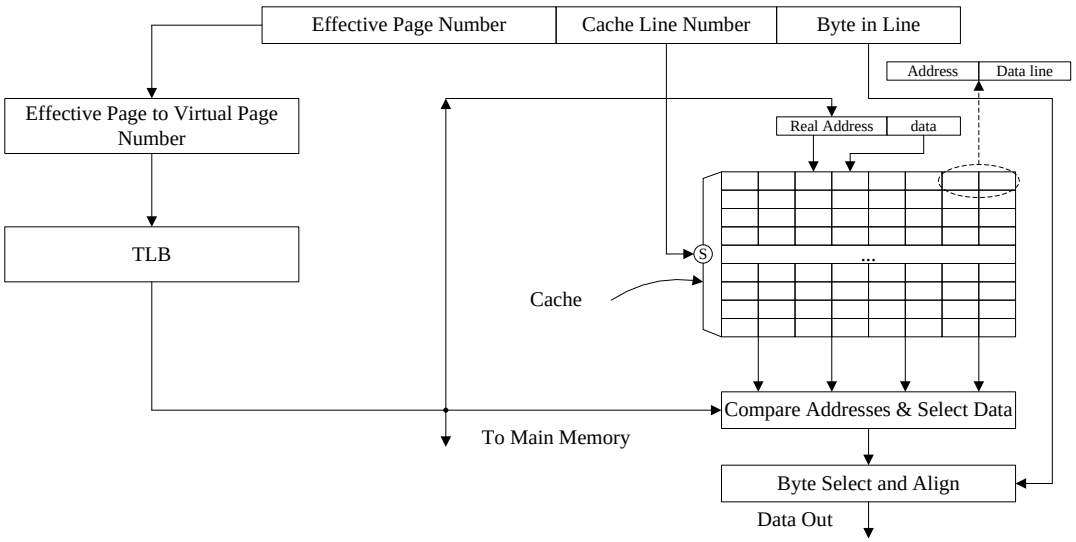

图 2-1 典型的 Cache 结构[①]

Cache 的存在使得 CPU Core 的存储器读写操作略微显得复杂。CPU Core 在进行存储器方式时，首先使用 EPN(Effective Page Number)进行虚实地址转换，并同时使用 CLN(Cache Line Number)查找合适的 Cache Block。这两个步骤可以同时进行。在使用 Virtual Cache 时，还可以使用虚拟地址对 Cache 进行寻址。为简化起见，我们并不考虑 Virtual Cache 的实现细节。

EPN 经过转换后得到 VPN，之后在 TLB 中查找并得到最终的 RPN(Real Page Number)。如果期间发生了 TLB Miss，将带来一系列的严重的系统惩罚，我们只讨论 TLB Hit 的情况，此时将很快获得合适的 RPN，并依此得到 PA(Physical Address)。

[①] 该图源自[23]的 Figure 2. A typical cache and TLB design，拷贝后过于模糊，重画这个示意图，并有所改动。

在多数处理器微架构中，Cache 由多行多列组成，使用 CLN 进行索引最终可以得到一个完整的 Cache Block。但是在这个 Cache Block 中的数据并不一定是 CPU Core 所需要的。因此有必要进行一些检查，将 Cache Block 中存放的 Address 与通过虚实地址转换得到的 PA 进行地址比较(Compare Address)。如果结果相同而且状态位匹配，则表明 Cache Hit。此时微架构再经过 Byte Select and Align 部件最终获得所需要的数据。如果发生 Cache Miss，CPU 需要使用 PA 进一步索引主存储器获得最终的数据。

由上文的分析，我们可以发现，一个 Cache Block 由预先存放的地址信息，状态位和数据单元组成。一个 Cache 由多个这样的 Cache Block 组成，在不同的微架构中，可以使用不同的 Cache Block 组成结构。我们首先分析单个 Cache Block 的组成结构。单个 Cache Block 由 Tag 字段，状态位和数据单元组成，如图 2-2 所示。

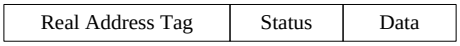

图 2-2 单个 Cache Block 的组成结构

其中 Data 字段存放该 Cache Block 中的数据，在多数处理器微架构中，其大小为 32 或者 64 字节。Status 字段存放当前 Cache Block 的状态，在多数处理器系统中，这个状态字段包含 MESI，MOESI 或者 MESIF 这些状态信息，在有些微架构的 Cache Block 中，还存在一个 L 位，表示当前 Cache Block 是否可以锁定。许多将 Cache 模拟成 SRAM 的微架构就是利用了这个 L 位。有关 MOESIFL 这些状态位的说明将在下文中详细描述。在多核处理器和复杂的 Cache Hierarchy 环境下，状态信息远不止 MOESIF。

RAT(Real Address Tag)记录在该 Cache Block 中存放的 Data 字段与那个地址相关，在 RAT 中存放的是部分物理地址信息，虽然在一个 CPU 中物理地址可能有 40，46 或者 48 位，但是在 Cache 中并不需要存放全部地址信息。因为从 Cache 的角度上看，CPU 使用的地址被分解成为了若干段，如图 2-3 所示。

Real Address Tag		Cache Line Index		Bank		Byte	
39	13	12	6	5	3	2	0

图 2-3 CPU 访问 Cache 使用的地址

这个地址也可以理解为 CPU 访问 Cache 使用的地址，由多个数据段组成。首先需要说明的是 Cache Line Index 字段。这一字段与图 2-1 中的 Cache Line Number 类似，CPU 使用该字段从 Cache 中选择一个或者一组 Entry[①]。

Bank 和 Byte 字段之和确定了单个 Cache 的 Data 字段长度，通常也将这个长度称为 Cache 行长度，图 2-3 所示的微架构中的 Cache Block 长度为 64 字节。目前多数支持 DDR3 SDRAM 的微架构使用的 Cache Block 长度都是 64 字节。部分原因是由于 DDR3 SDRAM 的一次 Burst Line 为 8[24]，一次基本 Burst 操作访问的数据大小为 64 字节。

在处理器微架构中，将地址为 Bank 和 Byte 两个字段出于提高 Cache Block 访问效率的考虑。Multi-Bank Mechanism 是一种常用的提高访问效率的方法，采用这种机制后，CPU 访问 Cache 时，只要不是对同一个 Bank 进行访问，即可并发执行。Byte 字段决定了 Cache 的端口位宽，在现代微架构中，访问 Cache 的总线位宽为 64 位或者为 128 位。

剩余的字段即为 Real Address Tag，这个字段与单个 Cache 中的 Real Address Tag 的字段长度相同。CPU 使用地址中的 Real Address Tag 字段与 Cache Block 的对应字段和一些状态位进行联合比较，判断其访问数据是否在 Cache 中命中。

[①] 如果使用 Set-Associative 方式组织 Cache 结构，此时使用 Index 字段可以获得一组 Entry。

2.2 Cache 的组成结构

由上文所述，在一个 Cache 中包含多行多列，存在若干类组成方式。在处理器体系结构的历史上，曾出现过更多的的组成结构，最后剩余下来的是我们耳熟能详的 Set-Associative 组成结构。这种结构在现代处理器微架构中得到了大规模普及。

在介绍 Set-Associative 组成结构之前，我们简单回顾另外一种 Cache 组成结构，Sector Buffer 方式[23]。假定在一个微架构中，Cache 大小为 16KB，使用 Sector Buffer 方式时，这个 16KB 被分解为 16 个 1KB 大小的 Sector，CPU 可以同时查找这 16 个 Sector。

当 CPU 访问的数据不在这 16 个 Sector 中命中时，将首先进行 Sector 淘汰操作，在获得一个新的 Sector 后，将即将需要访问的 64B 数据填入这个 Sector。如果 CPU 访问的数据命中了某个 Sector，但是数据并不包含在 Sector 时，将相应的数据继续读到这个 Sector 中。采用这种方法时，Cache 的划分粒度较为粗略，对程序的局部性的要求过高。Cache 的整体命中率不如采用 Set-Associative 的组成方式[23]。

在狭义 Cache 的设计中，这种方法已经不在使用。但是这种方法依然没有完全失效。处理器体系结构是尺有所短，寸有所长。在一些广义 Cache 的设计中，Sector Buffer 方式依然是一种行之有效的 Buffer 管理策略。有很多程序员仍在不自觉地使用这种方式。这种不自觉的行为有时是危险的，太多不自觉的存在有时会使人忘记了最基础的知识。

我曾经逐行阅读过一些工作了很多年的工程师的 Verilog 代码，在这些代码中使用了一些算法，这些算法我总感觉似曾相识，却已物是人非。他们采用的算法实际上有许多经典的实现方式，已经没有太多争论，甚至被列入了教科书中。有些工程师却忘记了这些如教科书般的经典，可能甚至没有仔细阅读过这些书籍，在原本较为完美的实现中填入蛇足。更为糟糕的是，一些应该存在的部件被他们轻易的忽略了。

有时间温故这些经典书籍是一件很幸运的事情。我手边常备着 A Quantitative Approach 和 The Art of Computer Programming 这些书籍，茶余饭后翻着，总能在这些书籍中得到一些新的体会。时间总是有的。很多人一直在抱怨着工作的忙碌，没有空余，虽然他们从未试图去挤时间的海绵，总是反复做着相同的事情。多次反复最有可能的结果是熟能生巧而为匠，却很难在这样近乎机械的重复中出现灵光一现的机枢。这些机枢可能发生在你多读了几行经文，或受其他领域的间接影响，也许并不是能够出现在日常工作的简单重复之上。

写作时回忆 Sector Buffer 机制时写下了这些文字。在现代处理器中，Cache Block 的组成方式大多都采用了 Set-Associative 方式。与 Set-Associative 方式相关的 Cache Block 组成方式还有 Direct Mapped 和 Fully-Associative 两种机制。Direct Mapped 和 Fully-Associative 也可以被认为是 Set-Associative 方式的两种特例。

在上世纪 90 年代，Direct Mapped 机制大行其道，在 Alpha 21064，21064A 和 21164 处理器中，L1 I Cache 和 D Cache 都使用了 Direct Mapped 方式和 Write Through 策略。直到 Alpha 21264，在 L1 Cache 层面才开始使用 2-Way Associative 方式和 Write Back 策略[16][17][18]。即便如此 Alpha 21264 在 L1 I Cache 仍然做出了一些独特设计，采用了 2-Way Set-Predict 结构，在某种程度上看使用这种方式，L1 I Cache 相当于工作在 Direct Mapped 方式中。

在 90 年代，世界上没有任何一个微架构能够与 Alpha 相提并论，没有任何一个公司有 DEC 在处理器微架构中的地位，来自 DEC 的结论几乎即为真理，而且这些结论都有非常深入的理论作为基础。与 Fully-Associative 和 Set-Associative 相比，Direct Mapped 方式所需硬件资源非常有限，每一次存储器访问都固定到了一个指定的 Cache Block。这种简单明了带来了一系列优点，最大的优点是在 200~300MHz CPU 主频的情况下，Load-Use Latency 可以是 1 个 Cycle。天下武功无坚不破，唯快不破。

至今很少有微架构在 L1 层面继续使用 Direct Mapped 方式，但是这种实现方式并没有如大家想象中糟糕。围绕着 Direct Mapped 方式，学术界做出了许多努力，其中带来最大影响的是 Normal P. Jouppi 书写的“Improving Direct-Mapped Cache Performance by Addition of a Small Fully-Associative Cache and Prefetch Buffers”，其中提到的 Victim Cache，Stream Buffer[20]至今依然在活跃。

使 Direct Mapped 方式逐步退出历史舞台的部分原因是 CPU Core 主频的增加使得 Direct Mapped 方式所带来的 Load-Use Latency 在相对缩小，更为重要的是呈平方级别增加的主存储器容量使得 Cache 容量相对在缩小。

从 Cache Miss Rate 的层面考虑，一个采用 Direct Mapped 方式容量为 N 的 Cache 其 Miss Rate 与采用 2-Way Set-Associative 方式容量为 N/2 的 Cache 几乎相同。这个 Observation 被 John 和 David 称为 2:1 Cache Rule of Thumb[7]。这意味着采用 Direct Mapped 方式的 Cache，所需要的 Cache 容量相对较大。

近些年 L1 Cache 与主存储器容量间的比值不但没有缩小而是越来越大。L1 Cache 的大小已经很少发生质的变化了，从 Pentium 的 16/32KB L1 Cache 到 Sandy Bridge 的 64KB L1 Cache，Intel 用了足足二十多年的时间。在这二十多年中，主存储器容量何止扩大了两千倍。相同的故事也发生在 L2 与 L3 Cache 中。这使得在采用 Direct Mapped 方式时，Cache 的 Miss Ratio 逐步提高，也使得 N-Way Set-Associative 方式闪亮登场。

采用 Set-Associative 方式时，Cache 被分解为 S 个 Sets，其中每一个 Set 中有 N 个 Ways。根据 N 的不同，Cache 可以分为 Fully-Associative, N-Ways Set-Associative 或者是 Direct Mapped。在 Cache 的总容量不变的情况下，即 S×N 的值为一个常数 M 时，N 越大，则 S 越小，反之亦然。8-Way Set-Associative Cache 的组成结构如图 2-4 所示。

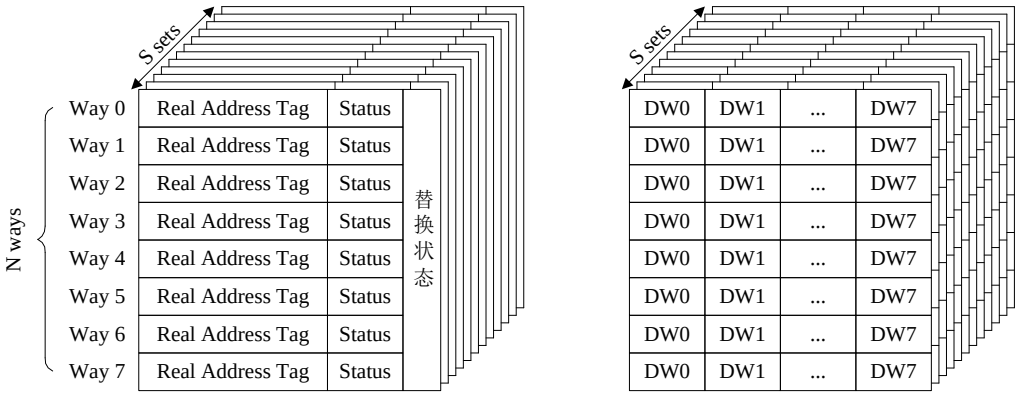

图 2-4 8-Way Set-Associative Cache 的组成结构

如图 2-4 所示，在 Cache Block 中 Real Address Tag 字段与数据字段分离，因为这两类字段分别存在不同类型的存储器中。在同一个 Set 中，Real Address Tag 阵列多使用 CAM(Content Addressable Memory)存放，以利于并行查找 PS(Parallel Search)方式的实现，在设计中也可以根据需要使用串行查找方式 SS(Sequential Search)。与 PS 方式相比，SS 方式使用的物理资源较少，但是当使用的 Ways 数较大时，采用这个方式的查找速度较慢。在现代微架构中，数据字段组成的阵列一般使用多端口，多 Bank 的 SRAM。下文主要讨论 PS 的实现方式。

对于一个 Cache，其总大小由存放 Tag 阵列和 SRAM 阵列组成，在参数 N 较低时也可以采用 RAM Tag，本篇仅讨论 CAM Tag，在功耗日益敏感的今天，Highly Associative Cache 倾向于使用 CAM Tag。在许多微架构中，如 Nehalem 微架构的 L1 Cache 为 32KB[12]，这是的大小是指 SRAM 阵列，没有包括 Tag 阵列。Tag 阵列占用的 Die Size 不容忽略。

在说明 Set-Associative 方式和 Tag 阵列之前，我们进一步讨论 NS 这个参数。不同的处理器采用了不同的 Cache 映射方式，如 Fully-Associative, N-Ways Set-Associative 或者是 Direct Mapped。如果使用 NS 参数进行描述，这三类方式本质上都是 N-Ways Set-Associative 方式，只是选用了不同的 NS 参数而已。

在使用 N-Ways Set-Associative 方式时，Cache 首先被分解为多个 Set。当 S 参数等于 1 时，即所有 Cache Block 使用一个 Set 进行管理时，这种方式即为 Fully-Associative；N 参数为 1 时的管理方式为 Direct Mapped；当 NS 参数不为 1 时，使用的方式为 N-Ways Set-Associative 方式。在图 2-4 中，N 为 8，这种方式被称为 8-Way Set-Associative。

诸多研究结果表明，随着 N 的不断增大，Cache 的 Miss Ratio 在逐步降低[7][23]。这并不意味着设计者可以使用更大的参数 N。在很多情况下，使用更大的参数 N 并不会显著降低 Miss Ratio，就单级 Cache 而言，8-Way Set-Associative 与 Fully-Associative 方式从 Miss Ratio 层面上看效果相当[7]。许多微架构使用的 16-Way 或者更高的 32-Way Set-Associative，并不是单纯为了降低 Miss Ratio。

随着 Way 数的增加，即便 Cache 所使用的 Data 阵列保持不变，Cache 使用的总资源以及 Tag 比较所需要的时间也在逐步增加。在一个实际的微架构中，貌似巨大无比的 CPU Die 放不下几片 Cache，在实际设计中一个 Die 所提供的容量已经被利用的无以复加，很多貌似优秀的设计被不得已割舍。这也使得在诸多条件制约之下，参数 N 并不是越大越好。这个参数的选择是资源权衡的结果。

当 S 参数为 1 时，N 等于 M，此时 Tag 阵列的总 Entry 数目依然为 M，但是对于 CAM 这样的存储器，单独的一个 M 大小的数据单元所耗费的 Die 资源，略微超过 S 个 N 大小的数据单元。随着 N 的提高，Tag 阵列所消耗的比较时间将逐步增加，所需要的功耗也在逐级增大。在进一步讨论这些细节知识前，我们需要了解 CAM 的基本组成结构，如图 2-5 所示。

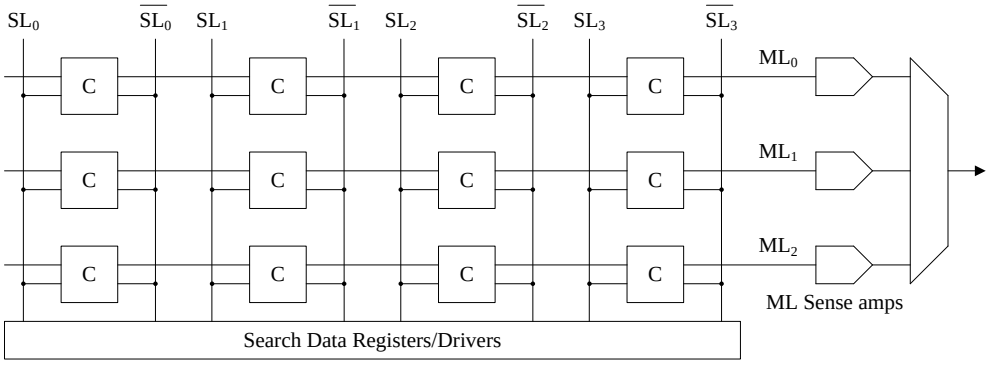

图 2-5 CAM 的基本组成结构[25]

图 2-5 中所示的 CAM 中有 3 个 Word，每一个 Word 由 4 个 Bits 组成，其中每一个 Bit 对应一个 CAM Cell。其中每一个 Word 对应一条横向的 ML(Match Line)，由 $ML_{0\sim2}$ 组成。在一个 CAM 内，所有 Word 的所有 Bits 将同时进行查找。在一列中，Bits 分别与两个 SL(Serach Line) 对应，包括 $SL_{0\sim3}$ 和~$SL_{0\sim3}$#。其中一个 Bit 对应一个 CAM Cell。

使用 CAM 进行查找时，需要首先将 Search Word 放入 Search Data Register/Drivers 中，之后这个 Search Word 分解为若干个 Bits，通过 SL 或者~SL 发送到所有 CAM Cell 中。其中每一个 CAM Cell 将 Hit/Miss 信息传递给各自的 ML。所有 ML 信息将最后统一在一起，得出最后的 Hit/Miss 结论，同时也将给出在 CAM 的哪个地址命中的信息[①]。由此可以发现，由于 CAM 使用并行查找方式，其查找效率明显操作 SRAM。

[①] 在 Cache 的设计中，地址信息并不是必要的。在 Routing Table 的查找中多使用了地址信息。

这也使得 CAM 得到了广泛应用。但是我们依然无法从这些描述中，获得随着 CAM 包含的 Word 数目增加，比较时间将逐步增加，所需要的功耗也在逐级增大这个结论，也因此无法解释 Cache 设计中为什么不能使用过多的 Way。为此我们需要进一步去微观地了解 CAM Cell 的结构和 Hit/Miss 识别机制。在门级电路的实现中，一个 CAM Cell 的设计可以使用两种方式，一个是基于 NOR，另一个是基于 NAND，如图 2-6 所示。

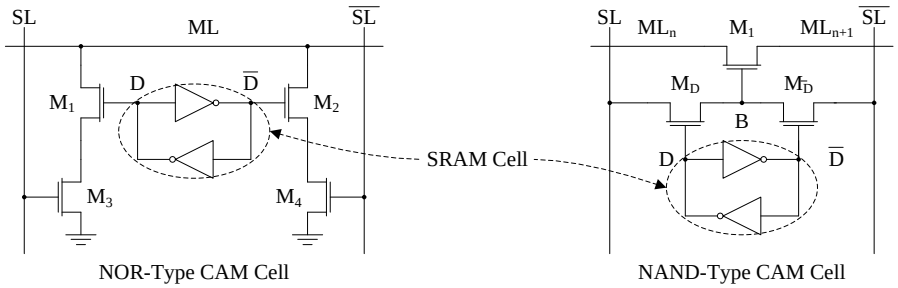

图 2-6 NOR 和 NAND CAM Cell[25]

在上图中可以发现 NOR-Type CAM Cell 和 NAND-Type CACM Cell 分别由 4 和 3 个 MOSGET 加上一个 SRAM Cell 组成，一个 SRAM Cell 通常由 6 个 Transistor 组成[①]。由此可以发现一个基本的 NOR-Type CAM Cell 由 10 个，NAND-Type CACM Cell 由 9 个 Transistor 组成。

这不是 NOR/NAND CAM Cell 的细节，NOR CAM Cell 有 9 个 Transistor 的实现方式，NAND CAM Cell 有 10 个 Transistor 的实现方式，这些由实现过程中的取舍决定。NOR 和 NAND CAM 的主要区别是 Word 的查找策略，NOR CAM Cell 采用并行查找方式，NAND CAM Cell 使用级联方式进行查找，如图 2-7 所示。

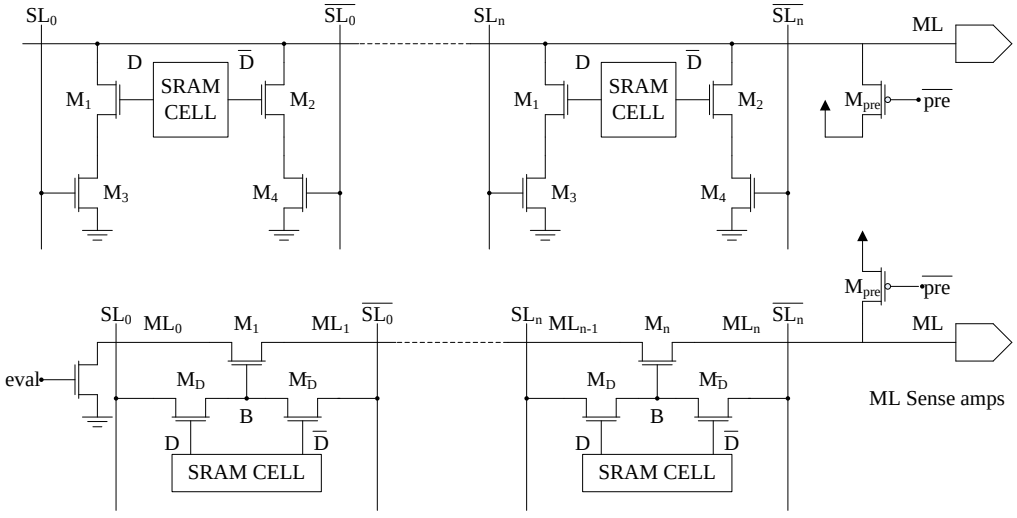

图 2-7 NOR/NAND CAM Word 的查找策略[25]

为节约篇幅，我省略如何对这些晶体管进行 Precharge，Discharge，如何 Evaluate，并最终确定一个 Word 是 Hit 还是 Miss，读者可以进一步阅读[25]获得详细信息。较为重要的内容是从图 2-7 中可以发现，当使用 NAND CAM 时，Word 的匹配使用级联方式，ML_n 需要得到 ML_{n-1} 的信息后才能继续进行，采用 NOR CAM 是全并行查找。仅从这些描述上似乎可以发现，NOR CAM 在没有明显提高 Transistor 数目的前提之下，从本质上提高了匹配效率，貌似 NAND CAM 并没有太多优点。

[①] SRAM Cell 还可以使用 8 个 Transistors 的实现方式。

事实并非如此。在使用 NOR CAM 时，随着 Word 包含 Bit 数目的增加，ML 上的负载数也随之增加。这些过多输入的与操作将带来延时和功耗。在图 2-7 所示的 N-Input 与操作采用了 One-Stage 实现方式，需要使用 Sense Amplifier 降低延时，但是这个 Sense Amplifier 无论在工作或者处于 Idle 状态时的功耗都较大。当然在 n 较大时，Multi-Stage n-input AND Gate 所产生的延时更不能接受。

即便使用了 Sense Amplifier，也并不能解决全部问题。当时钟频率较高时，这种并联方式所产生的延时很难满足系统需求。使用 NAND CAM 方式是一推一关系，没有这种负载要求。在 Cache 的 Tag 中除了需要保存 Real Address 之外还有许多状态信息，这也加大了 AND Gate 的负担。

CAM 除了横向判断一个 Word 是 Hit 还是 Miss 之外，还有另外一个重要问题，就是 Search Data Register/Driver 的驱动能力问题，这个问题进一步引申就是 N-Way Set-Associative 中 N 究竟多大才合理的问题。一个门能够驱动多少个负载和许多因素相关，频率，电流强度和连接介质等。对于一个运行在 3.3GHz 的 L1 Cache 而言，一个 Cycle 仅有 300ps，在这么短的延时内，门级电路做不了太多事情。

我们可能有很多方法提高驱动能力，Buffering the fan-out，使用金属线介质或者更多级的驱动器，缩短走线距离等等。在所有追求极限的设计中，这些方法不是带来了过高的延时，就是提高了功耗。一个简单的方式是采用流水方式，使用更多的节拍，但是你可知我为了节省这一拍延时经历了多少努力。这一切使得在 N-Ways Set-Associative 方式中 N 的选择是一个痛苦的折中，也使得在现代微架构的 L1 Cache 中 N 不会太大。

尽管有这些困难，绝大多数现代微架构还是选择了 N-Ways Set-Associative 方式，只是对参数 N 进行了折中。在这种方式下，当 CPU 使用地址 r(i)进行存储器访问时，首先使用函数 f 寻找合适的 Set，即 s(i) = f(r(i))，然后在将访问的地址的高字段与选中 Set 的 Real Address Tag 阵列进行联合比较，如果在 Tag 阵列中没有命中，表示 Cache Miss；如果命中则进一步检查 Cache Block 状态信息，并将数据最终读出或者写入。

现代微架构多使用 2-Ways，4-Ways，8-Ways 或者 16-Ways Set-Associative 方式组成 Cache 的基本结构。Ways 的数目多为 2 的幂，采用这种组成方式便于硬件实现。然而依然有例外存在，在有些处理器中可能出现 10-Ways 或者其他非 2 幂的 Ways。

出现这种现象的主要原因是这个 Way 并非对等。在一个处理器系统中，微架构和外部设备，如显卡和各类 PCIe 设备，都可以进行存储器访问。这些存储器访问并不类同。在多数情况下，微架构经由 LSQ，FLC，MLCs，之后通过 LLC，最终与主存储器进行数据交换。外部设备进行 DMA 访问时，直接面对的是 LLC 和主存储器，并对 L1 Cache 和 MLC 产生简介影响。微架构与外部设备访问主存储器存在的差异，决定了在有些处理器中，LLC Way 的构成并不一定是 2 的幂，而是由若干 2 的幂之和组成，如 10-Ways Set-Associative 可能是由一个 8-Ways 和一个 2-Ways 组成。

无论是软件还是硬件设计师都欣赏同构的规整，和由此带来的便利与精彩。但是在更多情况下，事物存在的差异性，使得严格的同构并不能发挥最大的功效，更多时候需要使用异构使同构最终成为可能。

在 Cache 中，不对等 Way 的产生除了因为访问路线并不一致之外，还有一个原因是为了降低 Cache Miss Rate，有些微架构进行 Way 选择使用了不同的算法。如 Skewed-Associative Cache[26]可以使用不用的 Hash 算法，f_0 和 f_1 分别映射一个 Set 内的两个 Way，采用这种方法在没有增加 Set 的 Ways 数目的情况下，有效降低了 Cache Miss Rate。[26]的结论是在 Cache 总大小相同时，2-Way Skewed- Associative Cache 的 Hit Ratio 与 4-Way Associative Cache 相当，其 Hit Time 与 Direct Mapped 方式接近。但是在 Cache 容量较大时，f_0 和 f_1 的映射成本也随之加大，从而在一定程度上增加了 Cache 的访问时间。

在历史上还出现过其他的 Cache 组成结构，如 Hash-rehash Cache，Column-Associative Cache 等，本篇对此不再一一介绍。值得留意的是 Parallel Multicolumn Cache[27]，这种 Cache 的实现要点是综合了 Direct Mapped Cache 和 N-Ways Set-Associative 方式，在访问 Cache 时首先使用 Direct Mapped 策略以获得最短的检索时间，在 DM 方式没有命中后，再访问 N-Ways 方式组成的模块。

上述这些方式基本是围绕着 Direct Mapped 进行优化，目前鲜有现代微架构继续使用这些方法，但是在一个广义 Cache 的设计中，如果仅存在一级 Cache，这些方法依然有广泛活跃着。这也是本节在此提及这些算法的主要原因。

在体系结构领域，针对 Cache 的 Way 的处理上，还有很多算法，但是在结合整个 Cache 层次结构的复杂性之后，具有实用价值的算法并不多。在目前已实现的微架构中，使用的最多的方式依然是 N-Ways Set-Associative。

在某些场景下，Miss Penalty 无法忍受，比如 TLB Miss，无论是采用纯软件还是 Hardware Assistance 的方法，Miss Penalty 的代价都过于昂贵。这使得架构师最终选择了 Fully Associative 实现方式。在现代微架构中，TLB 的设计需要对 Hit Time 和 Miss Rate 的设计进行折中，TLB 因此分为两级，L1 和 L2。L1-TLB 的实现侧重于 Hit Time 参数，较小一些，多使用 Fully Associative 方式，对其的要求是 Extremely Fast；L2 TLB 的实现需要进一步考虑 Miss Rate，通常较大一些，多使用 N-Way Associative 方式。

2.3 Why Index-Aware

在 N-Ways Set-Associative 方式的 Cache 中，CPU 如何选用函数 f 映射 Cache 中的 Set 是一个值得讨论的话题。其中最常用的算法是 Bit Selection。如图 2-3 所示，CPU 使用 Bits 12~6 选择一个合适的 Set。此时 f(r(i)) = Bits 12~6。

这是一种最快，最简洁的实现方式，使用这种方法带来的最大质疑莫过于 Set 的选择不够随机。历史上曾经有人试图使用某些 Pseudo-Random 算法作为函数 f，但是需要明确的是在 Set Selection 中，严格意义上的 Random 算法并不可取。

一是因为在 Silicon Design 中，很难在较短时间内产生一个随机数，即便使用最常用的 LFSR(Linear Feedback Shift Register)机制也至少需要一拍的延时，而且也并不是真正随机的。二是因为多数程序具有 Spatial Locality 特性，依然在有规律地使用 Cache，采用严格意义的 Random 很容易破坏这种规律性。

在许多实现中，Set Selection 时选用的 Pseudo-Random 算法等效于 Hash 算法，这些 Hash 算法多基于 XOR-Mapping 机制，需要几个 XOR 门级电路即可实现。诸多研究表明[23][28][29]，这种算法在处理 Cache Conflict Miss 时优于 Bit Selection。

在已知的实现中，追求 Hit Time 的 L1 Cache 很少使用这类 Hash 机制，但是这些方法依然出现在一些处理器的 MLC 和 LLC 设计中，特别是在容量较大的 LLC 层面。在 MLC 层面上，多数微架构使用的 Set Selection 的实现依然是简单而且有效的 Bit Selection 方式。

Bit Selection 方式所带来的最大问题是在选择 Set 时，经常发生碰撞。这种碰撞降低了 Cache 的整体利用率，这使得系统软件层面在使用物理内存时需要关注这个碰撞，也因此产生了与物理内存分配相关的一系列 Index-Aware 算法。

操作系统多使用分页机制管理物理内存。使用这种机制时，物理内存被分为若干个 4KB[①] 大小的页面。当应用程序需要使用内存时，操作系统将从空闲内存池中选用一个未用物理页面(Available Page Frame)。选取未用物理页面的过程因操作系统而异。

[①] 在多数操作系统中，4KB 是最常用的页面大小。本篇以此为例说明 Index-Aware 算法的必要性。

有些操作系统在选用这个物理页面时，并没有采用特别的算法，对此无所作为。在很多时候，这种无为而治反而会带来某种随机性，无心插柳柳成荫。对于 Cache 使用而言，这种无所作为很难带来哪怕是相对的随机。

精心编制的程序与随机性本是水火难容，而且这些程序一直在努力追求着时间局部性和空间局部性，最大化地利用着 Cache。这使一系列 Index-Aware 类的 Memory 分配算法得以引入。在介绍这些 Memory 分配算法之前，我们首先介绍采用分页机制后，一个进程如何访问 Cache，其示意如图 2-8 所示。

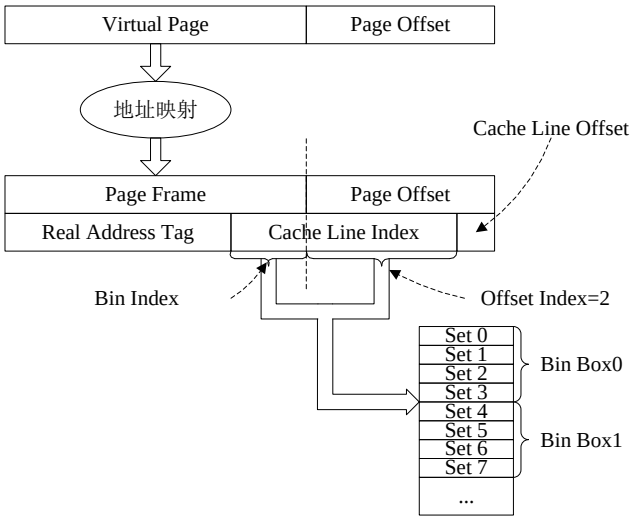

图 2-8 分页机制下 Cache 的使用方式[30]

在多数情况下，操作系统以 4KB 为单位将 Memory 分解为多个页面。如上图所示，这个 4KB 的页边界将 Cache Line Index 分解成两个部分，其中在 Page Frame 中的部分被称为 Bin Index，在 Page Off 中的部分被本篇称为 Offset Index。以此进行分析，Memory 分配算法也被分为两大类，一类是 Bin Index Aware，另一类是 Offset Index Aware Memory 分配算法。

我们首先简要介绍 Bin Index 的优化。根据 Bin Index 的不同，Cache 被分为 Large Cache 和 Small Cache 两类。当一个 Cache 的容量除以 Way 数大于实际使用的物理页面时，这种 Cache 被称为 Large Cache，反之被称为 Small Cache。

在许多处理器系统的实现中，L2 和 L3 一般都属于 Large Cache，L1 Cache 需要视情况而定。如 Sandy Bridge L1 Data Cache 的大小为 32KB，8-Ways 结构[5]，两者之商为 4KB，不大于 4KB 页面，该 Cache 即为 Small Cache。例如 Opteron 的 L1 Data Cache 为 64KB，2-Ways 结构[6]，两者之商为 32KB，大于 4KB 页面，该 Cache 即为 Large Cache。

需要注意的是 4KB 只是在多数操作系统中常用的物理页面大小，有些操作系统可以使用更大的页面，如 8KB，16KB 等。这使得 Small Cache 和 Big Cache 的划分不仅与微架构相关，而且与操作系统的具体实现相关。一个物理页面大小的使用与许多因素相关，页面越大，碎片问题也越发严重，但是与此相关的 TLB Miss Rate 也越低。在上文提到的，在 Superpages 中存在的 Allocation，Relocation，Promotion，Pollution 和 Fragmentation Control 等若干问题，随着页面大小的增加，其解决难度也在逐步增大。

另外需要注意的是，这里的“物理页面”指实际使用的物理页面。在 1GB 大小的 Superpage 面前，所有 Cache 都应该属于 Small Cache，但是只有极少有应用真正这样使用这个 1GB 页面，在多数情况下，1GB 大小的 Superpage 仍然被分解为更小的物理页面，可能是 4KB，8KB，或者是其他尺寸。

Bin-Index 优化算法的实现要点是，不同的物理页面在使用 Cache 时，尽量均匀分配到不同的 Bin Box 中。如图 2-8 所示，在一个 Bin 中通常有若干个 Set，Set 中还有若干个数据单元，当所有 Set 的所有数据单元都被占用时，如果有新的物理页面依然要使用这个 Bin Box 时，将在一定程度上引发 Cache Contention，即便在 Cache 中仍有其他未用的 Block。

对于一个进程而言，产生 Cache Contention 原因是，一个进程使用的虚拟地址空间虽然连续，但是在进行虚实映射时，内存分配器如果没有进行 Bin-Index 的优化手段，将“随机”选取一个物理页面与之对应，这个“随机”不但不是“随机”的，而且有非常强的规律性，从而造成了一些原本不该出现的 Cache Contention。

[30]列举了几个常用的 Bin-Index 的优化算法，如 Page Coloring，Bin Hooping，Best Bin 和 Hierarchical。有些算法已经在 FreeBSD 和 Solaris 中得到了实现，在 Linux 系统中，目前尚未使用这些 Bin-Index 算法。

但是我们不能得出因为 Linux 系统没有使用这类算法而导致性能低下这一结论。上文所述的所有 Bin-Index 算法都有其优点和缺点。而且从某种意义上说，不使用 Bin-Index 也是一种 Bin-Index 算法，同时在许多微架构的 Cache 设计中，由于 Virtual Cache 和 Hash 算法的使用，使得 Bin-Index 算法并不会取得很好的效果。

Page Coloring 算法最为简单，其主要原理是利用了 Virtual Cache 无需染色的原理，因为在多数情况下地址连续的 Virtual Page 很少在 Cache 中冲突，此时在分配物理页面时，其部分低位可以直接使用虚拟地址的低位，从而在一定程度上避免了 Cache Contention。如果进一步考虑多进程情况，使用这种算法时还可以将之前的结果与进程的 PID 参数再次进行 XOR-Mapping 操作；进一步考虑多内核情况，可以将再之前的结果与内核的 Logical Processor ID 进行 XOR-Mapping 操作。

对于使用了 Virtual Cache 的微架构，并不意味着 Bin-Index 算法不再适用。因为在多数情况下，Virtual Cache 仅在需要进一步降低 Hit Time 的 L1 Cache 中使用[①]，在 L2 或者更高层的 Cache 中，很少再使用这种技术。

在[30]中出现的 Bin Hooping，Best Bin 和 Hierarchical 算法也并不复杂，这些算法都是利用 Temporal Locality 原则，其详细实现参见[30]，这些算法并不复杂，没有逐行翻译的必要。在微架构和操作系统层面的设计与实现中，使用的多数算法都不复杂。

在一个操作系统实现中，Memory Allocator 是一个非常重要的组成部件，其设计异常复杂。有一些人在努力寻求最优的，通用的分配管理原则，虽然最优和通用几乎很难划等号。通用原则有通用原则的适用场合，专用的定制原则也有其存在的必要，没有一个绝对的准则。通常在一个 Memory Allocator 的设计中需要关注自身的运行效率，如 Footprint，False Sharing，Alignment 和 TLB 的优化等一系列问题，也包含 Cache-Index 的优化问题。

许多操作系统实现了 Cache-Index 的优化，包括 Bin-Index 和 Offset-Index。一些操作系统统筹考虑 Bin 和 Offset 的 Index，并将其合并为 Cache-Index，其中最著名算法的是 Hoard 和 Slab。近些年也出现了一些较新的 Cache-Index Aware 算法，如 CLFMalloc 和 CIF(Cache-Index Friendly)[31]。

这些算法都在关注如何寻求合适的策略以避免 Cache Index 的冲突，尽量使物理页面映射到 Cache 的不同 Set 中。需要读者进一步考虑的是，即便为某个应用找到了这些最优的映射方式，这些方式是否能在更广阔的应用领域发挥更大的效率。所有这些最优可能都有其合适的场景，很难有放之四海而皆准的策略。

为此我们首先简要回顾 Cache 自身的编码方式。下文以一个 2-Way Set-Associative，Cache Block 长度为 64B，总大小为 64KB 的 Cache 说明其内部的编码和组成方式。在采用这种方式时，该 Cache 的 Set 数目为 512。

[①] TLB Translation 处于 L1 Cache 访问的 Critical Path 中，Virtual Cache 可以提高转换效率，但依然是一个权衡。

这种结构的 Cache 与 Opteron 的 L1 Data Cache 类似，如图 2-9 所示。由上文所述，Cache 由两部分组成，一个是 Tag 阵列，另一个是数据阵列。在 Opteron 的 L1 Cache 中，Tag 阵列由 512 个 Set 组成，每个 Set 的 Entry 数目为 2，数据阵列与此一一对应。

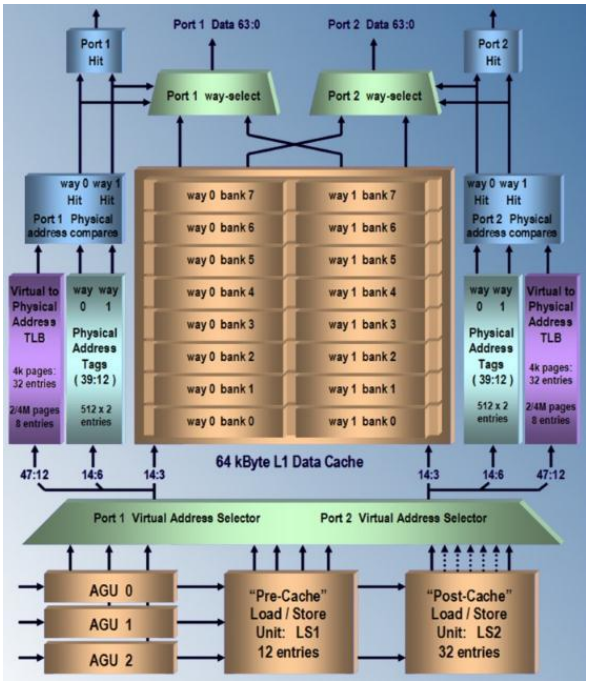

图 2-9 Opteron 的 L1 Data Cache[6]

我们首先讨论 Cache 的 Data 阵列。在现代处理器微架构中，从逻辑上看 Data 阵列首先被划分为多个 Set，在每一个 Set 中含有多个 Way，而且每一个 Way 由多个 Bank 组成。但是从物理实现上看，Data 阵列本质上是一块 SRAM，使用连续的物理地址统一编码。从 Silicon Design 的角度上看，Cache 的 Data 阵列具有地址，其编码方式如图 2-10 所示。

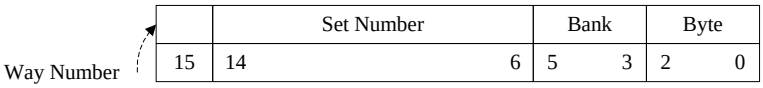

图 2-10 Cache 的 Data 阵列地址编码方式

对于 64KB 大小的 Cache，一共需要 16 位地址进行编码。为简化起见，我们忽略这个地址的最后 6 位，仅讨论 Set Number 和 Way Number。一个 Cache 的物理地址整体连续，然后被 Way Number 划分成为多个物理地址连续的子块。

CPU 访问 Cache 时，首先使用 Set Number 访问子块，在 2-Ways Set- Associative 结构中，将有两个子块被同时选中，之后这些数据将同时进入一个或者 Way-Select 部件。在这种情况下这个 Way-Select 部件的数据通路为 2 选 1 结构。

在现代处理器中，为提高 Cache 的数据带宽，通常设置多个 Way-Select 部件，以组成 Multi-Port Cache 结构。相应的，AGU 也必须具备产生多套地址的能力，分别抵达不同的 Tag 阵列。从而在微架构中需要设置多个 AGU，并为 Cache 设置多个 Tag 阵列以支持 Multi-Ports Cache 结构。多端口 Cache 的实现代价较大，多数处理器仅在 L1 层面实现多端口，其他层面依然使用着 Single-Port 结构。

图 2-9 中所示的 Cache 结构使用了 2-Ports 结构，每多使用一个 Port 就需要额外复制一个 Tag 阵列，也由此带来了不小的同步开销，而且过多的 Port 更易产生 Bank Conflict。在现代微架构的实现中，L1 Cache 层面多使用 2 个 Port，很少更多的 Port。同时为了支持 Cache 的流水操作，Opteron 微架构设置了 3 个 AGU 部件，个数超过了 2 个 Port 所需要的。其中更深层次的原因是 x86 处理器 EA 的计算在某些情况下过于复杂。

在 Cache 的组成结构中，还有一个细节需要额外关注。当 CPU 对一个物理地址进行访问没有在 Cache 中命中时，通常意味着一个 Cache Block 的替换。Cache Block 的替换算法对 Cache Hit 乃至整个 Cache 的设计至关重要。

2.4 Cache Block 的替换算法

在处理器系统处于正常的运行状态时，各级 Cache 处于饱和状态。由于 Cache 的容量远小于主存储器，Cache Miss 时有发生。一次 Cache Miss 不仅意味着处理器需要从主存储器中获取数据，而且需要将 Cache 的某一个 Block 替换出去。

不同的微架构使用了不同的 Cache Block 替换算法，本篇仅关注采用 Set-Associative 方式的 Cache Block 替换算法。在讲述这些替换算法之前，需要了解 Cache Block 的状态。如图 2-4 所示，在 Tag 阵列中，除了具有地址信息之外，还含有 Cache Block 的状态信息。不同的 Cache 一致性策略使用的 Cache 状态信息并不相同，如 Illinois Protocol[32]协议使用的相关的状态信息，该协议也被称为 MESI 协议。

在 MESI 协议中，一个 Cache Block 通常含有 MESI 这四个状态位，如果考虑多级 Cache 层次结构的存在，MESI 这些状态位的表现形式更为复杂一些。在有些微架构中，Cache Block 中还含有一个 L(Lock)位，当该位有效时，该 Block 不得被替换。L 位的存在，可以方便地将微架构中的 Cache 模拟成为 SRAM，供用户定制使用。使用这种方式需要慎重。

在很多情况下，定制使用后的 Cache 优化结果可能不如 CPU 自身的管理机制。有时一种优化手段可能会在局部中发挥巨大作用，可是应用到全局后有时不但不会加分，反而带来了相当大的系统惩罚。这并不是这些优化手段的问题，只是使用者需要知道更高层面的权衡与取舍。不谋万世者，不足谋一时；不谋全局者，不足谋一域。

在 Cache Block 中，除了有 MESIL 这些状态位之外，还有一些特殊的状态位，这些状态位与 Cache Block 的更换策略相关。微架构进行 Cache Block 更换时需要根据这些状态位判断在同一个 Set 中 Cache Block 的使用情况，之后选择合适的算法进行 Cache Block 更换。常用的 Replacement 算法有 MRU(Most Recently Used)，FIFO，RR(Round Robin)，Random，LRU(Least Recently Used)和 PLRU(Pseudo LRU)算法。

所有页面替换算法与 Belady's Algorithm 算法相比都不是最优的。Belady 算法可以对将来进行无限制的预测，并以此决定替换未来最长时间内不使用的数据。这种理想情况被称作最优算法，Belady's Algorithm 算法只有理论意义，因为精确预测一个 Cache Block 在处理器系统中未来的存活时间没有实际的可操作性，这种算法并没有实用价值。这个算法是为不完美的缓存算法树立一个完美标准。

在以上可实现的不完美算法中，RR，FIFO 和 Random 并没有考虑 Cache Block 使用的历史信息。而 Temporal 和 Spatial Locality 需要依赖这些历史信息，这使得某些微架构没有选用这些算法，而使用 LRU 类算法，这不意味着 RR，FIFO 和 Random 没有优点。

理论和 Benchmark 结果[23][35][37]多次验证，在 Miss Ratio 的考核中，LRU 类优于 MRU，FIFO 和 RR 类算法。这也并不意味着 LRU 是实现中较优的 Cache 替换算法。事实上，在很多场景下 LRU 算法的表现非常糟糕。考虑 4-Way Set-Associative 方式的 Cache，在一个连续访问序列{a, b, c, d, e}命中到同一个 Set 时，Cache Miss Ratio 非常高。

在这种场景下，LRU 并不比 RR，FIFO 算法强出多少，甚至会明显弱于 Random 实现方式。事实上我们总能找到某个特定的场景证明 LRU 弱于 RR，FIFO 和上文中提及的任何一种简单的算法。我们也可以很容易地找到优于 LRU 实现的页面替换算法，诸如 2Q，LRFU，LRU-K 和 Clock-Pro 算法等。

这些算法在分布式存储，Web 应用与文件系统中得到了广泛的应用。Cache 与主存储器的访问差异，低于主存储器与外部存储器的访问差异。这使得针对主存储器的页面替换算法有更多的回旋空间，使得在狭义 Cache 中得到广泛应用的 LRU/PLRU 算法失去了用武之地。PLRU 算法实现相对较为简单这个优点，在这些领域体现得并不明显，不足以掩饰其劣势。

LRU 算法并没有利用访问次数这个重要信息，在处理 File Scanning 这种 Weak Locality 时力不从心。而且在循环访问比广义 Cache 稍大一些的数据对象时，Miss Rate 较高[42]。LRU 算法有几种派生实现方式，如 LRFU 和 LRU-K。

LRFU 算法是 LRU 和 LFU(Least Frequently Used)，LFU 算法的实现要点是优先替换访问次数最少的数据。LRU-K 算法[38]记录页面的访问次数，K 为最大值。首先从访问次数为 1 的页面中根据 LRU 算法进行替换操作，没有访问次数为 1 的页面则继续查找为 2 的页面直到 K，当 K 等于 1 时，该算法与 LRU 等效，在实现中 LRU-2 算法较为常用。

LRU-K 算法使用多个 Priority Queue，算法复杂度为 $O(\log_2 N)$ [39]，而 LRU，FIFO 这类算法复杂度为 O(1)，采用这种算法时的 Overhead 略大，多个 Queue 使用的空间相互独立，浪费的空间较多。2Q 算法[39]的设计初衷是在保持 LRU-2 效果不变的前提下减少 Overhead 并合理地使用空间。2Q 算法有两种实现方式，Simplified 2Q 和 Full Version。

Simplified 2Q 使用了 A1 和 Am 两个队列，其中 A1 使用 FIFO 算法，Am 使用 LRU 算法进行替换操作。A1 负责管理 Cold 数据，Am 负责管理 Hot 数据，其中在 A1 的数据可以升级到 Am，但是不能进行反向操作。

如果访问的数据 p 在 Am 中命中时将其放回 Rear[①]；如果在 A1 中命中，将其移除并放入到 Am 中。如果 p 没有在 A1 或者 Am 中命中时，率先使用这些 Queue 中的空余空间，将其放入到 A1 的 Rear；如果没有空余空间，则检查 A1 的容量是否超过 Threshold 参数，超过则从 A1 的 Front 移除旧数据，将 p 放入 A1 的 Rear；如果没有超过则从 Am 的 Front 移除旧数据，将 p 放入 A1 的 Rear。

在这种实现中，合理设置 Threshold 参数至关重要。这个参数过小还是过大，都无法合理平衡 A1 和 Am 的负载。这个参数在 Access Pattern 发生变化时很难确定，很难合理地使用这些空间。LRU-2 算法存在同样的问题。这是 Full Version 2Q 算法要解决的问题。

Full Version 2Q 将 A1 分解为 A1in 和 A1out 两个 Queue，其中 Kin 为 A1in 的阈值，Kout 为 Aout 的阈值。此外在 A1in 和 A1out 中不再保存数据，而是数据指针，使用这种方法 Am 可以使用所有 Slot，在一定程度上解决了 Adaptive 的问题。

如果访问的数据 x 在 Am 中命中时将其放回 Rear；如果在 A1out 命中，则需要为 x 申请数据缓冲，即 reclaimfor(x)，之后将 x 放入 Am 的 Rear；如果 x 在 A1in 中命中，不需要做任何操作；如果 x 没有在任何 queue 中命中，则 reclaimfor(x)并将其放入 A1in 的 Rear。

如果 A1in，A1out 或者 Am 具有空闲 Slot，reclaimfor(x)优先使用这个 Slot，否则在 Am，Ain 和 Aout 中查询。如果|Ain|大于 Kin 时，则首先从 Ain 的 Front 处移除 identifier y，然后判断|Aout|是否大于 Kout，如果大于则淘汰 Aout 的 Front 以容纳 y，否则直接容纳 y。如果 Ain 和 Aout 没有超过阈值，则淘汰 Am 的 Front。

这些算法都基于 LRU 算法，而 LRU 算法通常使用 Link List 方式实现，在访问命中时，数据从 Link List 的 Front 取出后放回 Rear，发生 Replacement 时，淘汰 Front 数据并将新数据放入 Rear。这个过程并不复杂，但是遍历 Link List 的时间依然不能忽略。

[①] 此处对原文进行了修改。[39]中使用 Front 不是 Rear，我习惯从 Front 移除数据，新数据加入到 Rear。

Clock 算法可以有效减少这种遍历时间，而后出现的 Clock-Pro 算法的提出使 FIFO 类页面替换算法受到了更多的关注。传统的 FIFO 算法与 LRU 算法存在共同的缺点就是没有使用访问次数这个信息，不适于处理 Weak Locality 的 Access Pattern。

Second Chance 算法对传统的 FIFO 算法略微进行了修改，在一定程度上可以处理 Weak Locality 的 Access Pattern。Second Chance 算法多采用 Queue 方式实现，为 Queue 中每一个 Entry 设置一个 Reference Bit。访问命中时，将 Reference Bit 设置为 1。进行页面替换时查找 Front 指针，如果其 Reference Bit 为 1 时，清除该 Entry 的 Reference Bit，并将其放入 Rear 后继续查找直到某个 Entry 的 Reference Bit 为 0 后进行替换操作，并将新的数据放入 Rear 并清除 Reference Bit。

Clock 算法是针对 LRU 算法开销较大的一种改进方式，在 Second Change 算法的基础之上提出，属于 FIFO 类算法。Clock 算法不需要从 Front 移除 Entry 再添加到 Rear 的操作，而采用 Circular List 方式实现，将 Second Change 使用的 Front 和 Rear 合并为 Hand 指针即可。

这些算法各有优缺点，其存在的目的是为了迎接 LIRS(Low Inter-Reference Recency Set)[40]算法的横空出世。从纯算法的角度上看，LIRS 根本上解决了 LRU 算法在 File Scanning，Loop-Like Accesses 和 Accesses with Distinct Frequencies 这类 Access Pattern 面前的不足，较为完美地解决了 Weak Locality 的数据访问。其算法效率为 O(1)，其实现依然略为复杂，但在 I/O 存储领域，略微的计算复杂度在带来的巨大优势面前何足道哉。

在 LIRS 算法中使用了 IRR(Inter-Reference Recency)和 Recency 这两个参数。其中 IRR 指一个页面最近两次的访问间隔；Recency 指页面最近一次访问至当前时间内有多少页面曾经被访问过。在 IRR 和 Recency 参数中不包含重复的页面数，因为其他页面的重复对计算当前页面的优先权没有太多影响。IRR 和 Recency 参数的计算示例如图 2-11 所示。

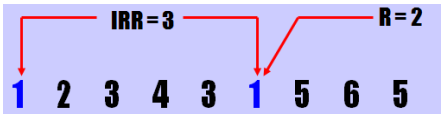

图 2-11 IRR 和 Recency 参数的计算示例[41]

其中页面 1 的最近一次访问间隔中，只有三个不重复的页面 2，3 和 4，所以 IRR 为 3；页面 1 最后一次访问至当前时间内有 2 个不重复的页面，所以 Recency 为 2。我们考虑一个更加复杂的访问序列，并获得图 2-12 所示的 IRR 和 Recency 参数。

V time / Blocks	1	2	3	4	5	6	7	8	9	10	R	IRR
A	X					X		X			1	1
B			X		X						3	1
C				X							4	inf
D		X					X				2	3
E									X		0	inf

图 2-12 多个页面的 IRR 和 Recency 参数的计算[41]

这组访问序列为{A，D，B，C，B，A，D，A，E}，最后的结果是各个页面在第 10 拍时所获得的 IRR 和 Recency 参数。以页面 D 为例，最后一次访问是第 7 拍，Recency 为 2；第 2~7 中有 3 个不重复的页面，IRR 为 3。其中 IRR 参数为 Infinite 表示在指定的时间间隔之内，没有对该页面进行过两次访问，所以无法计算其 IRR 参数。LIRS 算法首先替换 IRR 最大的页面，其中 Infinite 为最大值；当 IRR 相同时，替换 Recency 最大的页面。

IRR 在一定程度上可以反映页面的访问频率，基于一个页面当前的 IRR 越大，将来的 IRR 会更大的思想；Recency 参数相当与 LRU。在进行替换时，IRR 优先于 Recency，从而降低了最近一次数据访问的优先级。有些数据虽然是最近访问的却不一定常用，可能在一次访问后很长时间不会再次使用。如果 Recency 优先于 IRR，这些仅用一次的数据停留时间相对较长。

在一个随机访问序列中，并在一个相对较短的时间内精确计算出 IRR 和 Recency 参数并不容易。但是我们不需要精确计算 IRR 和 Recency 这两个参数。很多时候知道一个结果就已经足够了。在 LIRS 算法中比较 IRR 参数时，只要有一个是 Infinite，就不需要比较其他结果。假如有多个 infinite，比如 C 和 D，此时我们需要进一步比较 C 和 D 的 Recency，但是我们只需要关心 $C_R>D_R$ 这个结果，并不关心 C 是 4 还是 5。

LIRS 算法的实现没有要求精确计算 IRR 和 Recency 参数，而是给出了一个基于 LIRS Stack 的近似实现。LIRS 算法根据 IRR 参数的不同，将页面分为 LIR(Low IRR)和 HIR(High IRR)两类，并尽量使得 LIR 页面更多的在 Cache 中命中，并优先替换在 Cache 中的 HIR 页面。

LIRS Stack 包含一个 LRU Stack，LRU Stack 大小固定由 Cache 决定，存放 Cache 中的有效页面，在淘汰 Cache 中的有效页面时使用 LRU 算法，用以判断 Recency 的大小；包含一个 LIRS Stack S，其中保存 Recency 不超过 R_{MAX}[①]的 LIR 和 HIR 页面，其中 HIR 页面可能并不在 Cache 中，依然使用 LRU 算法，其长度可变，用于判断 IRR 的大小；包含一个队列 Q 维护在 Cache 中的 HIR 页面，以加快这类页面的索引速度，在需要 Free 页面时，首先淘汰这类页面。淘汰操作将会引发一系列连锁反应。我们以图 2-13 为例进一步说明。

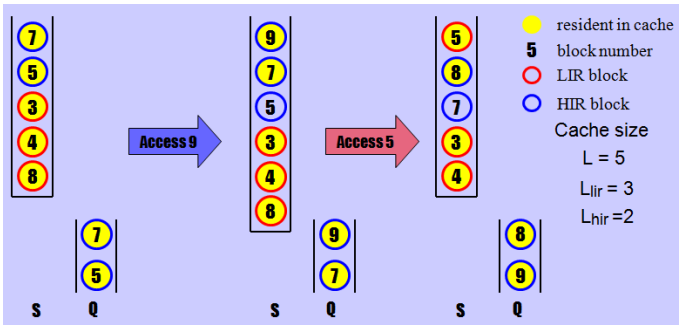

图 2-13 访问不在 Cache 命中的 HIR 页面[41]

我们仅讨论图 2-13 中 Access5 这种情况，此时 Stack S 中存放页面{9，7，5，3，4，8}，Q 中存放{9，7}。Stack S 的{9，7，3，4，8}在 Cache 中，{3，4，8}为 LIR 页面，{9，7，5}为 HIR 页面。其中 Cache 的大小为 5，3 个存放 LIR 页面，2 个存放 HIR 页面。

对页面 5 的访问并没有在 Cache 中命中，此时需要一个 Free 页面进行页面替换。LIRS 算法首先淘汰在 Q 中页面 7，同时将这页面在 S 中的状态更改为不在 Cache 命中；之后页面 8 从 S 落到 Q 中，状态从 LIR 迁移到 HIR，但是这个页面仍在 Cache 中，需要重新压栈；页面 5 没有在 Cache 中命中，但是在 S 中命中，需要将其移出后重新压栈，状态改变为在 Cache 中命中。本篇不再介绍 LIRS 算法的实现细节，对此有兴趣的读者可以参照[40][41]。

而后出现的 Clock-Pro 算法是 LIRS 思想在 Clock 算法中的体现。Clock-Pro 和 LIRS 都为 LIRS 类算法。其中 Clock-Pro 算法实现开销更小，适用于操作系统的 Virtual Memory Management，并得到了广泛的应用，Linux 和 NetBSD 使用了该算法；LIRS 算法适用于 I/O 存储领域中，mySQL 和 Apache Derby 使用了该算法。LIRS 算法较为完美地解决了 Weak Access Locality Access Pattern 的处理。在 LIRS 算法出现之后，还有许多页面替换算法，这些后继算法的陆续出现，一次又一次证明了尚未出现更好的算法在这些领域上超越 LIRS 算法。

① R_{MAX} 为 Recency 的最大值。

LIRS 和 Clock-Pro 算法在这个领域的地位相当于 Two-Level Adaptive Branch Prediction 在 Branch Prediction 中的地位。在详细研读[40][42]的细节之后，发现更多的是作者的实践过程。LIRS 算法类不是空想而得，是在试错了 99 条路之后的发现。这是创新的必由之路。

在掌握必要的基础知识后，也许我们最应该做的并不是研读他人的书籍和论文，更多的是去实践。经历了这些艰辛的实践过程，才会有真正的自信。这个自信不是盲从的排他，是能够容纳更多的声音，尽管发出这个声音的人你是如此厌恶。

与微架构设计相比，在操作系统和应用层面可以有更多的资源和更多的时间，使用更优的页面替换算法。虽然在操作系统和应用层面对资源和时间依然敏感，但是在这个层面上使用的再少的资源和再短的时间放到微架构中都是无比巨大。在微架构的设计中，很多在操作系统和应用层面适用的算法是不能考虑的。

假设一个 CPU 的主频为 3.3GHz，在每一个 Cycle 只有 300ps 的情况之下，很多在操作系统层面可以使用的优秀算法不会有充足的时间运行。虽然 LRU 算法在 Simplicity 和 Adaptability 上依然有其优势，在微架构的设计中依然没有得到广泛的应用。即便是 LRU 算法，在 Cache 的 Ways Number 较大的情况之下也并不容易快速实现。当 Way Number 大于 4 后，LRU 算法使用的 Link Lists 方式所带来的延时是 Silicon Design 不能考虑的实现方式。更糟糕的是，随着 Way Number 的增加，LRU 算法需要使用更多的状态位。

下文讨论的 Cache Block 替换算法针对 N-Way Set Associative 组织方式。在这种情况下，Cache 由多个 Set 组成，存储器访问命中其他 Set 时，不会影响当前 Set 的页面更换策略，所谓的替换操作是以 Set 为单位进行的。为简化起见，假设下文中出现的所有存储器访问都是针对同一个 Set，不再考虑访问其他 Set 的情况。

通常情况，在 N-Way Set Associative 的 Cache 中，快速实现 Full LRU 最多需要 N×(N-1)/2 个不相互冗余的状态位，理论上的最小值是 Floor(LOG_2(N!))个状态位[23]。因此当 Way Number 大于 4 之后，所需要的状态位不是硬件能够轻易负担的，所需要的计算时间不是微架构能够忍受的。这使得更多的微架构选用了 PLRU 算法进行 Cache Block 的 Replacement。

与 LRU 算法相比，PLRU 算法使用了更少的存储空间，查找需要替换页面的时间也较短，而且从 Miss Rate 指标的考量上与 LRU 算法较为类似，在某些特殊场景中甚至会优于 LRU 算法[36]，从而在微架构的设计中得到了大规模的应用。

PLRU 算法有两种实现方式，分别为 MRU-Based 和 Tree-Based 方式。MRU-Based PLRU 的实现方式是为每一个 Cache Block 设置一个 MRU Bit，存储器访问命中时，该位将设置为 1，表示当前 Cache Block 最近进行过访问。当因为 Cache Miss 而进行 Replacement 时，将寻找为 0 的 MRU Bit，在将其替换的同时，设置其 MRU Bit 为 1。

采用这种方式需要避免同一个 Set 所有 Cache Block 的 MRU Bit 同时为 1 而带来的死锁。考虑一个 4-Way Set Associative 的 Cache，当在同一个 Set 只有一个 Cache Block MRU Bit 不为 1 时，如果 CPU 对这个 Cache Block 访问并命中时，则将该 Cache Block 的 MRU Bit 设置为 1，同时将其他所有 Cache Block 的 MRU Bit 设置为 0，如图 2-14 所示。

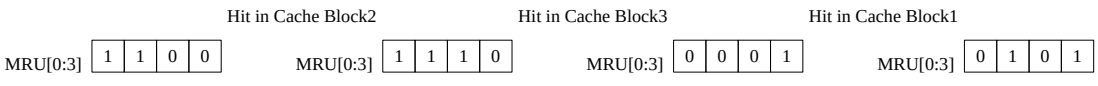

图 2-14 MRU-Based PLRU 算法

在上图中，假设 MRU[0:3]的初始值为{1, 1, 0, 0}，当一次存储器访问命中 Cache Block2 时，MRU[0:3]将迁移为{1, 1, 1, 0}；下一次访问命中 Cache Block3 时，MRU[0:3]的第 3 位置 1，为了避免 MRU[0:3]所有位都为 1 而出现死锁，此时其他位反转为 0，即 MRU[0:3]迁移为{0, 0, 0, 1}；再次命中 Cache Block1 时，将迁移为{0, 1, 0, 1}。

有些量化分析结果[36]认为 MRU-Based 实现方式在 Cache Miss Ratio 的比较上，略优于 Tree-Based PLRU 方式。但是从实现的角度上考虑，使用 MRU-Based 实现时，每一个 Set 都需要增加一个额外的 Bit。这并不是问题关键，重要的是 MRU-Based 实现在搜索为第一个为 0 的 MRU Bit 时需要较大的开销，也无法避免为了防止 MRU Bits 死锁而进行反转开销。

本篇所重点讨论的是 Tree-Based PLRU 实现方式。下文将以一个 4-Way Set Associative 的 Cache 说明 PLRU 的使用方式，使用更多 Way 的方式可依此类推。在 4-Way 情况之下，实现 PLRU 算法需要设置 3 个状态位 B[0~2]字段，分别与 4 个 Way 对应；同理在 8-Way 情况下，需要 7 个状态位 B[0~7]；而采用 N-Way Set Associative 需要 N-1 个这样的状态位，是一个线性增长。Tree-Based PLRU 使用的替换规则如图 2-15 所示。

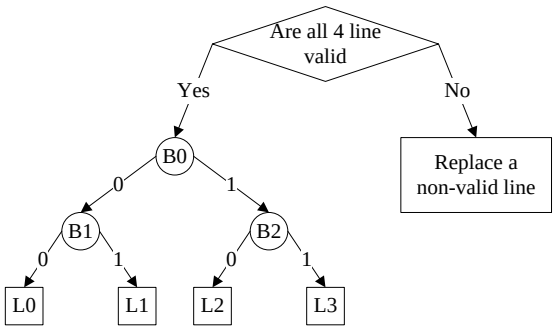

图 2-15 Tree-Based PLRU 替换算法

在 Cache Set 初始化结束后，B0~2 位都为 0，此时在 Set 中的 Cache Block 的状态为 Invalid。当处理器访问 Cache 时，优先替换状态为 Invalid 的 Cache Block。只有在当前 Set 中，所有 Cache Block 的状态位都不为 Invalid 时，Cache 控制逻辑才会使用 PLRU 算法对 Cache Block 进行替换操作。

在 Tree-Based PLRU 实现中，搜索过程基于 Binary Search Tree，在 N 较大时，其搜索效率明显高于 MRU-Based PLRU。在这种实现中，当所有 Cache Block 的状态不为 Invalid 时，将首先判断 B0 的状态，之后决定继续判断 B1 或者 B2。如果 B0 为 0，则继续判断 B1 的状态，而忽略 B2 的状态；如果 B0 为 1，则继续判断 B2 的状态，而忽略 B1 的状态。

举例说明，如果 B0 为 0 而且 B1 为 0 时，则淘汰 L0；否则淘汰 L1。如果 B0 为 1 而且 B2 为 0，则淘汰 L2；否则淘汰 L3。淘汰合适的 Cache Block 后，B0~B2 状态将被更新。值得注意的是，除了发生 Cache Allocate 导致的 Replacement 之外，在 Cache Hit 时，B0~B2 的状态同样需要更新。Cache Set 替换状态的更新规则如表 2-1 所示。

表 2-1 PLRU Bits 的更新规则

Current Access	New State of the PLRU Bits		
	B0	B1	B2
L0	1	1	No Change
L1	1	0	No Change
L2	0	No Change	1
L3	0	No Change	0

以上更新规则比较容易记忆。从图 2-15 中可以发现在替换 L0 时，需要 B0 和 B1 为 0，与此对应的更新规则就是对 B0 和 B1 取反，而 B2 保持不变；同理替换 L3 时，需要 B0 和 B2 为 1，与此对应的更新规则就是对 B0 和 B2 取反，而 B1 保持不变。

依照以上规则，我们简单举一个实例说明 PLRU 算法的替换和状态迁移规则。假设连续三次的存储器访问分别命中了同一个 Set 的不同 Way，如顺序访问 Way 3，0 和 2。其 B0~2 的状态迁移如图 2-16 所示。

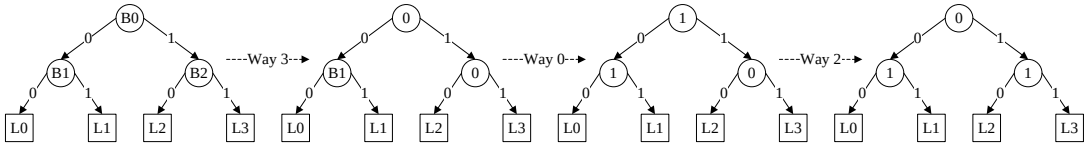

图 2-16 Way3, 0 和 1 访问序列

由以上访问序列可以发现，当 CPU 访问 Way 3，0 和 2 之后，B0~B2 的状态最后将为 0b011，此时如果 Cache Block 需要进行 Replacement 时，将优先替换 Way 1。这个结果与期望相符，对此有兴趣的读者，可以构造出一些其他访问序列，使得最终结果与 LRU 算法预期不符。这是正常现象，毕竟 PLRU 算法是 Pseudo 的。

根据以上说明，我们讨论与 Tree-Based PLRU 算法的相关的基本原则。由上文的描述可以发现当 Cache 收到一个存储器访问序列后，Cache Set 的替换状态将根据 PLRU 算法进行状态迁移，我们假设这些存储器访问序列都是针对一个 Set，而且存储器访问使用的地址两两不同。这是因为重复的地址访问不会影响到 Cache Set 的替换状态，二是因为如果访问序列是完全随机的，几乎没有办法讨论 Cache 的替换算法。满足这一要求，而且最容易构造的序列是，忽略一个地址的 Offset 字段，Index 保持不变，其上地址顺序变化的存储器访问序列。

为了进一步描述 Cache Block 的替换算法，我们引入 Evict(k)和 Fill(k)这两个参数，其中参数 k 指 Way Number。Evict 指经过多少次存储器访问后才会将 Cache Set 中未知的数据完全清除，在一个指定的时间，Cache Set 中包含的数据是无法确定的，但是经过 Evict(k)次存储器访问可以将这些未知数据全部清除。而在经过 Fill(k)次存储器访问后，可以确定在 Cache Set 中存在那些访问数据，Evict 和 Fill 参数的关系如图 2-17 所示。

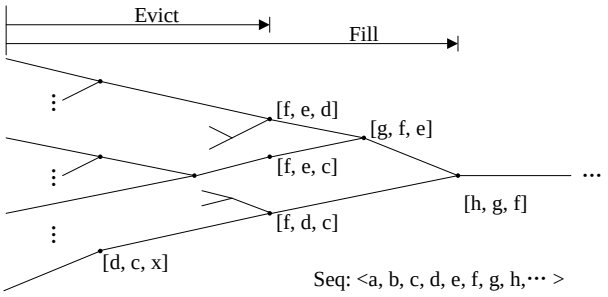

图 2-17 Fill(k)与 Evict(k)参数[37]

Fill(k)和 Evict(k)参数的计算需要分多种情况分别进行讨论。我们首先讨论参数 Evict(k)。如果在一个 Cache Set 内的所有 Way 的状态都为 Invalid，这种情况几乎不用讨论，那么连续 K 次存储器访问，一定可以将该 Set 内的所有 Way Evict。如果所有 Way 的状态都是 Valid，这种情况也较为容易，同样需要 K 次存储器访问 Evict 所有 Way。

而如果在一个 Cache Set 内，有些 Way 为 Invalid 而有些不是，这种情况略微复杂一些。而且一次存储器访问可以在 Cache 中 Hit 也可能 Miss，此时 Evict 和 Fill 参数的计算方法更为复杂一些。我们使用 $Evict_m$ 和 $Fill_m$ 表示存储器访问在 Cache 中 Miss 的情况，而使用 $Evict_{hm}$ 和 $Fill_{hm}$ 表示其他情况。

我们首先讨论每一次存储器访问都出现 Cache Miss 的情况，此时如果在 Cache Set 内具有 Invalid 的 Cache Block，并不会使用 PLRU 标准替换流程，而直接使用状态为 Invalid 的 Cache Block。此时将一个 Set 内所有 Way Evict 所需要的存储器访问次数如公式 2-1 所示。

$$\mathrm{Evict_m} = \begin{cases} 2k - \sqrt{2k} : k = 2^{2i+1}, i \in N \\ 2k - \frac{3}{2}\sqrt{K} : otherwise \end{cases}$$ 公式 2-1[37]

当 k 等于 8 时，$\mathrm{Evict_m}$ 为 12。这也是在 PowerPC E500 手册中，Evict 大小为 32KB 的 L1 Dcache 需要操作 48KB 内存区域的原因。使用这一方法时需要注意两个实现细节，一个是 Interrupts must be disabled，另一个是 The 48-Kbyte region chosen is not being used by the system—that is, that snoops do not occur to this region.[43]。

如果进一步考虑在一个存储器访问序列中，在 Cache Set 中，不但具有 Invalid 的 Cache Block，而且不是每一次存储器访问都会出现 Miss 操作而引发 Replacement 操作，因为可能某次访问了出现 Cache Hit 时，公式 2-1 进一步演化为公式 2-2。

$$\mathrm{Evict_{hm}} = \frac{K}{2}\log_2 K + 1$$ 公式 2-2[37]

在这种情况下，对于 8-Way Set Associative 的 Cache，将一个 Set 所有 Way Evict 所需要的存储器访问次数为 13。在 Cache Block 替换算法中，MLS(Minimum Life-Span)参数也值得关注，该参数表示在一个 Way 在 Cache Set 中的最小生命周期。如果一次存储器操作命中了一个 Way，这个 Way 至少需要 MLS 次 Cache 替换操作后，才能够从当前 Set 中替换出去。对于 Tree-Based PLRU 算法，MLS 的计算如公式 2-3 所示。

$$MLS(k) = \log_2 k + 1$$ 公式 2-3[37]

由以上公式，可以发现对于一个 8-Ways Set Associative 的 Cache，一个最近访问的 Block，其生命周期至少为 4，即一个刚刚 Hit，或者因为 Miss 而 Refill 的 Cache Block，至少需要 4 次 Cache Block 替换操作后才能被 Evict。

MLS 参数可以帮助分析 Cache 的 Hit Rate，对于一个已知算法的 Cache，总可以利用某些规则，极大提高 Hit Rate，只是在进行这些优化时，需要注意更高层次的细节。与 Cache Block 替换算法相关的 $\mathrm{Fill_m}$ 和 $\mathrm{Fill_{hm}}$ 参数的计算如公式 2-4 所示。

$$\begin{cases} \mathrm{Fill_m}(k) = 2k - 1 \\ \mathrm{Fill_{hm}}(k) = \frac{k}{2}\log_2 k + k - 1 \end{cases}$$ 公式 2-4[37]

除了 PLRU 算法之外，文章[37]对 FIFO，MRU 和 LRU 进行了详细的理论推导，这些证明过程并不复杂，也谈不上数学意义上的完美，但是通过这篇文章提出的 Fill 和 Evict 算法和相关参数，依然可以从 Qualitative Research 的角度上论证一个替换算法自身的 Beautiful，特别是对于一个纯粹的 Cache 替换算法，在没有考虑多级 Cache 间的耦合，和较为复杂的多处理器间的 Cache 一致性等因素时的分析。

如何选用 Cache Block 的 Replacement 算法是一个 Trade-Off 过程，没有什么算法一定是最优的。在 Niagara 和 MIPS 微架构的实现中甚至使用了 Rand 算法，这个算法实现过程非常简单，最自然的想法是使用 LFSR 近似出一组随机序列，与 Cache Set 中设置替换状态相比，LFSR 使用的资源较少，而且确定需要 Replacement 的 Cache Block 的过程非常快速。

虽然很多评测结果都可以证明 Random 算法不如 PLRU，但是这个算法使用了较小系统资源，而且系统开销较小。利用这些节省下来的资源，微架构可以做其他的优化。因而从更高层次的 Trade-Off 看，Random 算法并不很差。

在体系结构领域很少有放之四海而皆准的真理。PLRU 算法之后，也有许多针对其不足的改进和增强，这些想法可能依然是 Trade Off。但是不要轻易否定这些结果，在没有较为准确的量化分析结果之前，不能去想当然。想当然与直觉并不等同。人类历史上，许多伟大的革新源自某个人的直觉。这些革新在出现的瞬间甚至会与当时的常识相悖。

在一个技术进入到稳定发展阶段，很难有质的提高。更多时候，在等待着变化，也许是使用习惯的改变，也许是新技术的横空出世。在许多情况下，CS 和 EE 学科的发展进步并不完全依靠自身的螺旋上升发生的质变，更多的时候也许在耐心等待着其他各个领域的水涨船高。Cache 替换算法如此，体系结构如此，人类更高层面的进步亦如此。

近期随着 MLP 的崭露头角，更多的人重新开始关注 Cache Block 的替换算法，出现了一系列 MLP-Aware 的替换算法，如 LIN(Linear) Policy[44]，LIP(LRU Insertion Policy)，BIP(Bimodal Insertion Policy)和 DIP(Dynamic Insertion Policy)[45]。

其中 LIN 算法根据将 Cache Miss 分为 Multiple Cache Miss 和 Isolated Miss。其中 Isolated Miss 出现的主要场景是 Pointer-Chasing Load 操作，而 Multiple Cache Miss 发生在对一个 Array 的操作中。Multiple Cache Miss 可以并行操作，Amortize 所有开销，对性能的影响相对较小；而 Isolated Miss 是一个独立操作对性能影响较大，传统 Cache Block 的替换算法并没有过多考虑这些因素，从而影响了性能。LIN 算法即是为了解决这些问题而引入[44]。

LIP，BIP 和 DIP 针对 Memory-Intensive 的应用。在某些 Memory-Intensive 应用中，可能存在某段超出 Cache 容量的数据区域，而且会按照一定的周期循环使用。如果采用传统的 LRU 算法，最新访问的 Cache Block 将为 MRU，在超过 MLS 个 Cycle 后才可能被替换，而将不应该替换的 Cache Block 淘汰，从而造成了某种程度的 Cache Trashing。

采用 LIP 算法时，则将这样的 Incoming Cache Block 设置为 LRU，以避免 Cache Trashing，这种思想谈不上新颖，重要的是简洁快速的实现。BIP 是 LIP 的改进。DIP 是对 BIP 和传统的 LRU 算法进行加权处理，以实现 Adaptive Insertion Policies[45]。

除了在 Memory-Intensive 的应用层面之外，相对在不断提高的 DDR 延时，也改变着 Cache Block Replacement 算法的设计。现代处理器的设计对不断提高的 DDR 延时在本质上束手无策，只有引入更多的 Cache 层次，采用容量更大的 LLC，在满足访问延时的同时获得更高的 Coverage 与不断增长的 DDR 容量和访问延时匹配。与此对应，产生了一些用于多级 Cache 结构的 Cache Block 替换算法，如 Bypass and Insertion Algorithms for Exclusive LLC[46]。

这些算法的出现与日益增加的主存储器容量与访问延时相关，也是当前 Cache Block 替换算法的研究热点。这为 Cache Hierarchy 的设计提出了新的挑战，使得原本已经非常难以构思，难以设计难以验证的 Cache 层次结构，愈加复杂。

2.5 指令 Cache

在一个处理器系统中，指令 Cache 与数据 Cache 的组成方式和使用规则有所不同。在现代处理器系统中，在 L1 Cache 层面，指令 Cache 与数据 Cache 通常分离，而在其后的 Cache 层次中，指令与数据混合存放，在多数情况下 L1 指令 Cache 是只读的，因此 Cache Block 中包含的状态较少一些，一致性处理相对较为简单。

与指令 Cache 相比，数据 Cache 的设计与实现复杂得多。在此回顾指令 Cache 的主要原因是，在之后的篇章中，我会专注于介绍数据 Cache。在目前已知的微架构中，x86 体系结构的指令 Cache 和指令流水线的设计最为复杂。所以本节只介绍 x86 处理器中指令 Cache 和与其相关的设计。这些复杂度与 x86 处理器不断挑战着处理器运行极限直接相关，此外由 x86 的 Backword-Compatibility 而继承的变长 CISC 指令也需要对此负责。这些变长指令对于设计者是一个不小的灾难。在这场灾难中，x86 架构久病成良医促成了一个又一个的发现。

通常情况下，工业界很少有大的成果能够领先于学术界。更多的情况是理论上较为成熟的技术在若干年之后被 Industry 采纳。Intel 的伟大之处在于在微架构的很多领域中领先于理论界。更加令人深思的是，Intel 的很多发现其起源是为了解决自身的并不完美。

Intel 的每一代 Tock，几乎都是从 Branch Predictor 的设计与优化开始，并基于此重新构建指令流水与 Cache Hierarchy 结构。在微架构中功耗与性能的权衡主要发生的领域也集中于此。在处理器系统中最大的功耗损失莫过于把一些已经执行完毕的指令抛弃和一些不必要重复的操作。在指令执行阶段，Misprediction 将刷新指令流水线，将丢弃很多 In-Flight 的指令，对于 x86 处理器，丢弃一条最长可达 256b 的指令，是一个不小的损失。而 Cache Hierarchy 是微架构耗费资源最多的组成部件。一个处理器将广义和狭义 Cache 去掉之后，所剩无几。

在 Intel x86 处理器的 Tick-Tock 的不断运行过程中，Branch Predictor 的设计依然在不断发展，指令流水线的设计也随之改变。虽然指令流水线并不是本篇所关注的重点，但仍有必要在此介绍一些与指令 Cache 直接相关的内容。在 x86 微架构中，一条指令流水线通常被分为 Front-End 和 Back-End 两大部分。在 Front-End 与 Back-End 之间使用 DQ(Decoder Queue)连接，其结构如图 2-18 所示。

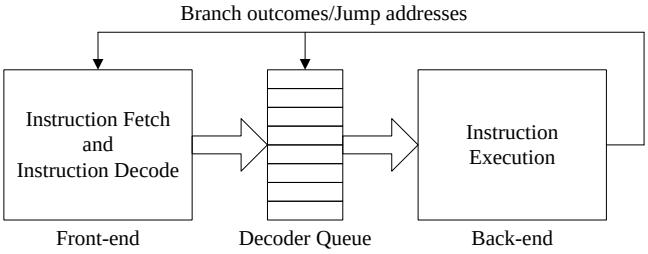

图 2-18 Front-end 与 Back-end 之间的关系[67]

Front-End 负责指令的 Fetches 和 Decodes，将指令发送给 Decoder Queue，相当于 Producer；而 Back-End 在条件允许的情况在从 Decoder Queue 中获得指令并执行，相当于 Consumer。指令在 Back-End 中执行完毕后，需要将结果反馈给 Decoder Queue，进一步处理 Depedency 和资源冲突，同时还需要为 Front-End 中的 Branch Predictor 反馈 Branch 指令的最终执行结果。Branch Predictor 使用这些反馈确认是否发生了 Misprediction。

Front-End，Decoder Queue 和 Back-End 之间需要协调工作，以保证整个流水线的顺利运转。理想的情况是 Front-End 可以将指令源源不断地发送给 Decoder Queue，而 Back-End 可以顺利地从 Decoder Queue 获得指令，采用这种方式似乎只要不断提高 Front End 送至 Decoder Queue 的指令条数(Instruction Block)，就可以不断的提高 ILP。

但是这个 Block 数目并不能无限制提高，因为在 DQ 中的指令流已经被 Branch 指令切割成为若干个子指令流，物理地址连续的指令流从程序执行的角度上看并不连续，为此现代处理器多设置了强大的 Branch Predictor 猜测程序即将使用的子块。

但是对于一个指令流水线过长的微架构，一次 Misprediction 所带来的 Penalty 仍然足以击溃指令流水的正常运行。基于这些考虑，Intel 在 Pentium IV 处理器中，率先使用了 Trace Cache 机制以最大限度的使指令流水线获得事实上连续的指令流。

从本质上看，Trace Cache 机制是一个机器的自学习加修正策略，试图让机器尽可能的分析本身难以琢磨的指令流，捕获其运行规律，使得指令流水获得的指令尽量连续。从整个微架构的发展历史上看，少有这样级别机器智能的成功案例。为什么偏要赋予仅能识别 0 和 1 这两个数字的机器如此重任。从历史的进程上看，从微架构中流传下来更多的是简单精炼的设计。偏执的 Pentium IV 使用了 Trace Cache。

这个后来被 John 和 David 称为“likely a one-time innovation”[7]的技术，在当时依然受到热捧，“Trace cache: a low latency approach to high bandwidth instruction fetching”这篇文章被公开索引了五百多次。

当时依然有很多人质疑使用 Trace Cache 提高的性能与付出的代价严重不成比例，但是有更多的学者在大书特书这个主题，根据这些文章提供的模型和相关的 Quantitative Analysis，不难得出 Trace Cache 是一个伟大发明的结论，直到高频低能的 Pentium IV 被 AMD 的 Opteron 彻底击败。尽信量化分析结果不如没有。

Trace Cache 的组成结构和使用方法不在本书的讨论范围内，对此部分有兴趣的读者可以参考[67]获得详细信息。在 Intel 后继发布的微架构中已不见 Trace Cache 的踪迹，但是依然不能全盘否定 Pentium IV 构架使用的指令预取，其详细过程如图 2-19 所示。

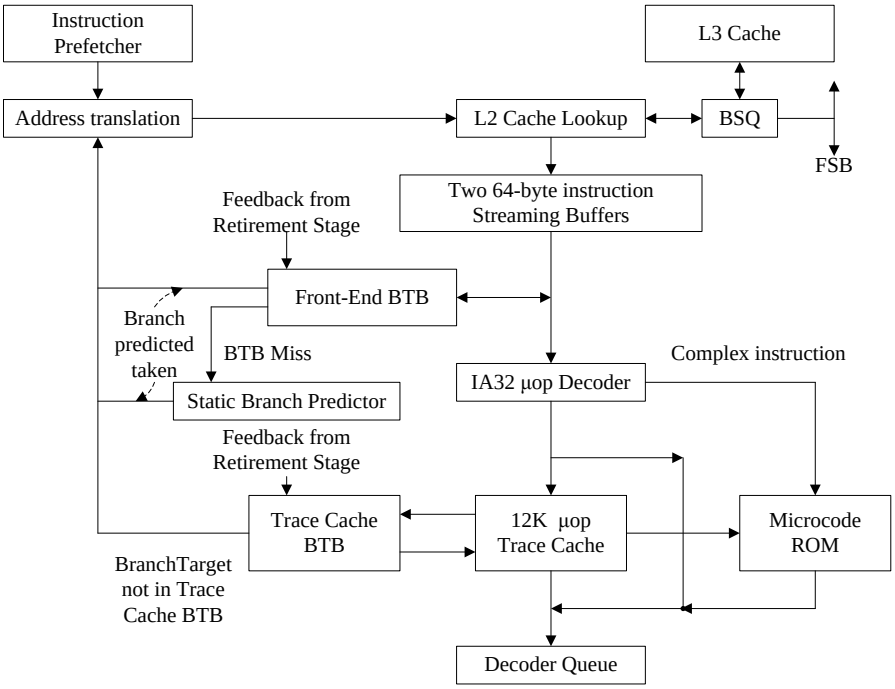

图 2-19 Pentium IV 架构的 Front-End[68]

在 x86 架构中，FLC 被分解为指令与数据 Cache， MLCs 和 LLC 同时存放这指令与数据。这种方式是当代绝大多数微架构采用的 Cache 组成方式，也被称为 Harvard Architecuture，更为准确的称呼应该是 Modified Harvard Architecuture。

在 Pentium IV 架构中，指令将从 L2 Cache 首先到达 Instruction Stream Buffers，为简化起见，本节不讨论所预取的指令在 L2 Cache Miss 的情况。只要 Instruction Stream Buffers 不满，预读地址有效，控制逻辑就会将指令从 L2 Cache Lookup 单元源源不断地传递下去。

如果在 Instruction Stream Buffer 中的指令为 Conditional Branch 时，该指令将首先被送至 Front-End BTB 进行查找，如上文所述，在一个程序执行过程中使用的指令流并不连续而是被 Conditional Branch 分割成为若干个子块，因此需要对这类指令进行特别的处理。

这些指令将被保存在 Front-End BTB 中，这个 Front-End BTB 也是一种广义 Cache，其组成结构和使用方式与狭义 Cache 类似。因为在 Front-End BTB 中只有 4096 个 Entry，所以 Miss 时有发生，在 Miss 时，需要根据相应的替换算法淘汰一个 Entry，并重新创建一个 Entry，但是并不会改变指令的预读路径，同时 Conditional Branch 还需要转交给 Static Branch Predictor 做进一步处理，本节不关心这种情况。

如果 Hit，需要检查 BTB 预测的结果是 Not Taken 还是 Taken。如果是 Not Taken，BTB 不会通知指令预读单元改变预读路径；如果是 Taken，则将 BTB 预测的 Target Address 送至指令预取单元，最终传递到 L2 Cache Lookup 单元，将 Target Address 指向的指令数据向下传递给 Instruction Stream Buffers。

当 Conditional Branch 从指令单元中 Retire 时，会将 Commit 的结果传递给 Front-End BTB 和 Trace Buffer BTB。如果 Conditional Branch 最终的执行结果与 BTB 的预测结果相同时皆大欢喜。不同时即为 Misprediction，会带来一系列严厉的系统惩罚，不再详细描述这个过程。

Instruction Stream Buffers 会将 CISC 指令继续传递给 IA μop Decoder。Decoder 将绝大部分的 CISC 指令翻译成为 μops 并送入 Trace Cache，同时按序送入 Decoder Queue。此时在 Trace Cache 中保存的是 μops，而不再是原始的 CISC 指令。部分 CISC 指令，如会被送入到 Microcode ROM，进行查表译码，这些 CISC 指令较为复杂，通常会被分解为 5 条以上的 μops，设立 Microcode ROM 是 x86 架构的无奈，我们忽略这个无奈。

译码后得到的 μops 将依次进入 Trace Cache。其中 Conditional Branch 译码后得到的 μops 需要进行额外处理，因为 Trace Cache 的实现要点是记录这些 Conditional Branch 连接而成的指令流而不是物理地址连续的指令流。

Pentium IV 为 Trace Cache 设立了专用的 BTB，Trace Cache BTB，并且希望其命中率最好是 100%，这样 Decoder Queue 中的 μops 可以全部来自 Trace Cache 中已有的已经完成译码的指令集合。这只是一个理想情况，依然存在着 Branch Target 没有在这个 BTB 中命中的情况，此时依然需要回退到 Address Translation 部件，进行指令预取。Trace Cache BTB 依然会监听 Conditional Branch 最后的执行情况，并做进一步处理。

Pentium IV 微架构在 Front-End 所做的各种努力，并没有改变其高频低能的结局。这一个几乎给 Intel 带来浩劫的微架构最终被抛弃，以色列 IDC 团队的 Core Architecuture 拯救了 Intel。Intel 启动了 Tick-Tock 计划，更加优秀的微架构 Nehalem 横空出世，很快是 Sandy Bridge。

Nehalem 没有保留 Trace Cache，Sandy Bridge 微架构也没有。但是 Pentium IV 的 Front-End 并没有轻易地被抛弃，我们依然可以在 Nehalem 和 Sandy Bridge 微架构中找到源自 Pentium IV 的设计，其 Front-End 的组成结构如图 2-20 所示。

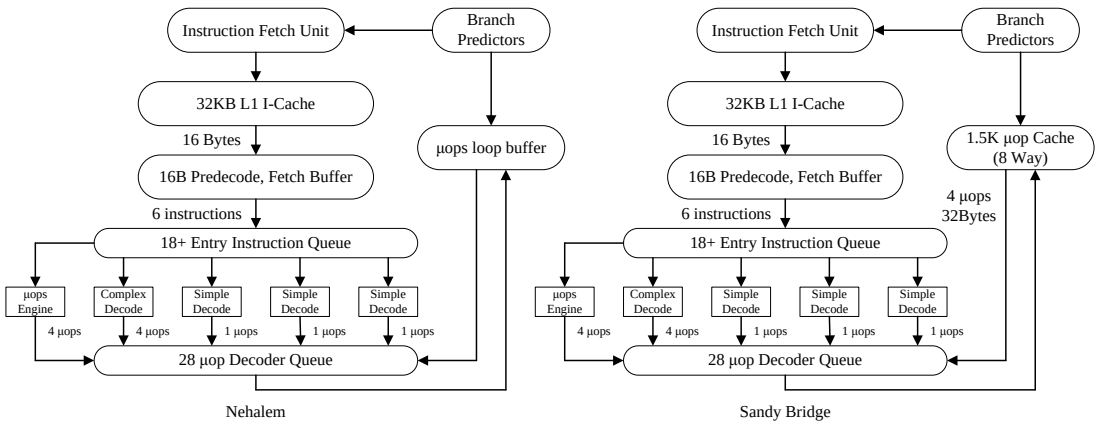

图 2-20 Nehalem 和 Sandy Bridge 的 Front End[69]

Nehalem 的 Front-End 源自 Core Archtecture，与 Pentium IV 有较大差异。Pentium IV 巨大的失败阴影左右了 Intel 后续微架构的设计。Pentium IV 流水线的多数设计没有被继承，除了一个源自 Trace Cache 的附带品。x86 架构的 CISC 指令过于冗长，与其他微架构相比译码过程也复杂很多。在 Trace Cache 中保留经过已经完成译码的 μops 是一个不错的设想。

Merom 微架构继承了这个设想，使用 Instruction Loop Buffer 保留这些 μops，在 Nehalem 中这个部件被称为 μops Loop Buffer，最后 Sandy Bridge 使用一个 1.5KB 的 μops Cache 保留这些已经完成译码的 μops[1]。

这个 Decoded μops Cache 也被 Sandy Bridge 微架构称之为 L0 Cache[1]。Sandy Bridge 微架构采用 L0 指令 Cache 除了可以保留珍贵的译码结果之外，更多的是基于 Power/Energy 的考虑。如果在一个程序的执行过程中，检测到 Loop 后，将直接从 μops Cache 中获得指令，暂时关闭并不需要的部件以节约功耗。

历经 Nehalem 和 Sandy Bridge 两轮 Tock 之后，x86 处理器在 Branch Predictor 部件的设计中虽然仍有调整，但是日趋稳定。x86 微架构引入 Two-Level Adaptive Branch Prediction[47] 机制之后，BTB 的变化更加细微。除了 μops Cache 之外，Nehalem 微架构 Branch Predictor 使用的 Gshare Predictor，Indirect Branch Target Array 和 Renamed Return Stack Buffer 这几个重要部件被 Sandy Bridge 架构继承[69]。

与 Pentium IV 相比，Core，Nehalem 和 Sandy Bridge 微架构简化了 Front-End 的设计，也使我失去了进一步使用文字叙述的动力。但是与其他微架构，如 Power 和 MIPS 相比，不论是 Nehalem 还是 Sandy Bridge，在 Front-End 的设计中依然投入了大量的资源。对于 x86 体系结构，也许在放弃了 Backword-compatibility 之后，才能真正地改变 Front-End 的设计。也许到了那一天，Front-End 这个专用术语也会随之消失。

2.6 Cache Never Block

在一个微架构中，有两条值得重点关注的流水线，一个是指令流水线。另一个是 Cache Controller 使用的流水线，下文将其简称为 Cache 流水线。这两条流水线的实现对于微架构的性能至关重要。指令流水线的设计与 Cache 流水线相关，反之亦然。

1967 年 6 月，来自 IBM 的 ROBERT MACRO TOMASULO 先生发明了最后以自己名字命令的算法[48]，这个算法最终使得 Alpha 处理器，MIPS 处理器，Power 处理器，x86 处理器，ARM 系列处理器，所有采用 OOO 技术的处理器成为可能。1997 年 Eckert-Mauchly Award 正式授予 TOMASULO 先生, for the ingenious Tomasulo's algorithm, which enabled out-of-order execution processors to be implemented.

2008 年 4 月 3 日，TOMASULO 先生永远离开我们。他的算法也历经了多轮改进。即便在针对 ILP 的优化因为 Memory Stall 而处境艰难，TOMASULO 算法也并不过时。虽然本篇文章的重点并不在 Superscalar 和 OOO，仍然建议所有读者务必能够清晰地理解 TOMASULO 算法和 Superscalar 指令流水的细节。为节约篇幅，本篇不会对这些知识做进一步的说明。

而在 Cache 流水线中，Non-Blocking 几乎等同于 TOMASULO 算法在指令流水中的地位。在现代处理器中，几乎所有 Cache Hierarchy 的设计都采用了 Non-Blocking 策略。Non-Blocking Cache 实现为 Superscalar 处理器能够进一步发展提供了可能。

假设在一个 Superscalar 处理器中，一个时钟周期能够发射 n 条指令。这要求该处理器需要设置多个执行部件，提供充分的并行性。假设在这个处理器中，某类功能部件 x 所能提供的带宽为 BW_X，此处的带宽是借用存储器的概念，如果该功能部件需要 4 拍才能完成一次操作，那么该功能部件为指令流水线所提供的带宽为 1/4。

在一个应用的执行过程中使用这类功能部件所占的百分比为 f_X。那么只有当 BW_X/f_X 不小于 n 时，Superscalar 处理器才能够充分的并行，否则该功能部件必将成为瓶颈。因此在微架构的设计中，通常并行设置多个执行较慢的功能部件以提高 BW_X 参数，当然还有一个方法是缩小 f_X 参数。如何缩小 f_X 参数并不是微架构的关注领域，因为在一个给定的应用中，从微架构设计的角度上看，f_X 参数没有太大的变化空间。

我们可以将 BW_X/f_X 公式扩展到存储器读写指令。假设 BW_S 为提供给指令流水线的存储器访问带宽，而 f_M 为存储器读写指令所占的比例，那么只有在 BW_S/f_M 不小于 n 时，存储器访问单元才不会成为指令流水线的瓶颈。

为此我们建立一个基本的存储器访问模型。为简化起见，假设在一个 Superscalar 处理器中，存储器结构的最顶层为 L1 Cache，其中 L1 Cache 由指令和数据 Cache 两部分组成，并使用 Write Back 方式进行回写，L1 Cache 通过 L1-L2 Bus 与 L2 Cache 连接。L2 Cache 与主存储器系统直接连接。处理器在访问存储器时，首先通过 L1 Cache 之后再经过 L2 Cache，最后到达主存储器系统。该模型也可以进一步扩展到 L3 Cache 和更多的 Cache 层次，但是为了简化起见，我们仅讨论 L1 和 L2 Cache 的情况。在这个前提之下，我们讨论与 Cache 相关的延时与带宽，重点关注 Cache Hierarchy 为指令流水提供的有效带宽，即 BW_S 参数。

当微架构进行存储器访问时，将首先访问 L1 Cache，此时有 Hit 与 Miss 两种情况。如果为 Hit，L1 Cache 将直接提供数据；如果为 Miss，L1 Cache 将产生一个 Miss 请求，并通过 L1-L2 Bus 从 L2 Cache 中获得数据。

通常情况下 Cache Miss 会引发 Cache Block 的 Replacement，如果被替换的 Cache Block 为 Dirty 时，还需要向 L1-L2 Bus 提交 Writeback 请求，此时 L1 Cache Controller 将向 L2 Cache 发送两类数据请求，一个是 Cache Miss Request，一个是 Write Back Request。为了提高 Miss Request 的处理效率，在绝大多数微架构中首先向 L2 Cache 发送 Cache Miss Request，之后再发送 Write Back Request。

由上文的分析可以发现，BW_S 参数不能通过简单的计算迅速得出，必须要考虑整个 Cache Hierarchy 的实现方式。BW_S 参数与 L1-L2 Bus 的实现方式，与是否采用了 Non-Blocking Cache 的设计密切相关。

在现代微架构中很少再继续使用 Blocking Cache 实现方式，但是我们依然需要分析为什么不能使用这种技术。在使用 Blocking Cache 的微架构中，存储器访问如果在 L1 Cache 中 Hit 时，可以直接获得数据，不会影响指令流水。

但是出现 Cache Miss 时，由于 Cache Blocking 的原因，L1-L2 Bus 将被封锁，直到从其下的存储器子系统中获得数据。由于采用这种技术 L1-L2 Bus 一次仅可处理一个 L1 Cache Miss，如果出现多个 Cache Miss 的情况时，这些 Miss 请求将逐一排队等待上一个 Miss 获得数据。这种做法严重降低了 BW_S 参数，从而极大影响了 Superscalar 处理器的并行度。

从以上说明可以发现 Blocking Cache 的主要问题是处理 Cache Miss 时，使用了停等模型，L1 Cache 最多仅能处理一个 Pending 的 Miss 请求。而 L1-L2 Bus 一次也只能处理一种数据请求，或者是 Miss 请求，或者是 Writeback 请求。

在否定 Blocking Cache 之前，我们需要对 BW_S 参数做进一步的分析。假设在一段程序中一共进行了 H+M 次存储器请求，其中 H 是 L1 Cache 的 Hit 次数，而 M 为 Miss 次数。由于本次模型规定存储器访问不得 Bypass L1 Cache，所以在 L1 Cache 中 Hit 和 Miss 次数之和即为存储器访问请求的总和。

在现代处理器中 L1 Cache 通常设置两个 Ports，但是为了简化模型，我们假设在 L1 Cache 中只含有一个 Port。假设 L1 Cache 在 Hit 的情况下，一个 Cycle 即可将数据读出，此时处理 H 个 Hit 操作，L1 Cache 一共需要使用 H 个 Cycle，如果所有存储器访问都能命中 L1 Cache，BW_S 参数为 1。但是 L1 Cache 不可能永远 Hit，因此在计算 BW_S 参数时，必须要考虑 M 个 Cache Miss 的情况。

L1 Cache 处理一次 Miss 所需的时间为 T_M+B，而 L1-L2 Bus 在处理一次 Miss 所需的时间为[T_M+B(1+d)]。其中 T_M 是指 L1 Cache 发出 Miss 请求之后，L2 Cache 可以提供有效数据 Block 的时延；B 指 L2 Cache 通过 L1-L2 Bus 向 L1 Cache 提交一个数据 Block 的时间；d 指需要进行 Writeback 请求的比例。

在 Blocking Cache 中，由于 Hit 和 Miss 操作不能流水处理，因此 L1 Cache 和 L1-L2 Bus 在处理 H+M 次存储器请求时，一共需要(H+M[T_M+B])和 M×[T_M+B(1+d)]个 Cycle。在这样一个处理器系统中，BW_S 参数为 L1 Cache 和 L1-L2 Bus 所提供带宽的最小值，如公式 2-5 所示。

$$\mathrm{Min}\left[\frac{H+M}{H+M\times[T_M+B]},\frac{H+M}{M\times[T_M+(1+d)\times B]}\right]=\mathrm{Min}\left[\frac{1}{1+m\times[T_M+B-1]},\frac{1}{m\times[T_M+(1+d)\times B]}\right]$$

公式 2-5[49]

其中 m 为 Miss Ratio，即 M/(H+M)。由公式 2-5 可以发现，当 m 增加时 BW_S 参数将等比例降低。考虑 m 等于 0.05，T_M 等于 10，B 等于 1，d 等于 0 这个理想状态时，BW_S 参数也仅为 0.67。此时如果 f_M 参数为 0.4，BW_S 与 f_M 参数的比值为 1.67。由此可以得出在这种模型下，一个 Superscalar 处理器不管内部具有多少可以并行执行的模块，但是一次发射的指令都不能超过两条，否则存储器读写必将成为瓶颈，最终阻塞整个流水线。这一发现使 Superscalar 处理器因为 Cache Blocking 的原因几乎无法继续发展。

因此必须有效的提高 BW_S 参数。提高 BW_S 参数有两个有效途径，一个是增加命中 L1 Cache 后的带宽，也由此带来了多端口 Cache 的概念。正如上文的讨论结果，Cache 上每增加一个端口，需要耗费巨大的开销，而且 Cache 容量越大，这类开销越大。在现代处理器中，L1 Cache 多由两个端口组成，如图 2-9 所示。为了进一步提高并行度，Superscalar 处理器还可以使用更多的端口，但是这将使用更多的资源。

在对 Cost/Performance 进行 Trade-Off 之后，Cache 设计引入了 Multi-Bank 的概念。一些量化分析的结果证明，与单纯的 Multi-Port Cache 实现相比，Multi-Bank 与更少的 Port 组合在经过一些简单的消除 Bank Conflict 的优化之后，可以获得更高的 Cost/Performance 比[50]。这使得 MBMP(Multi-Banks and Multi-Ports)结构在现代微架构的 Cache 设计中得到了广泛应用。Opteron 的 L1 Data Cache 使用了 8-Banks 2-Ports 的组成结构[6]。

这些优化依然只是提高了 Cache Hit 时的 Cache 带宽。如公式 2-5 所示，计算 Cache 的总带宽需要考虑 Cache Miss 时的情况。因此即便我们将 Hit 时的 Cache 带宽提高到 Infinite，依然不能解决全部问题，必须要考虑 1/(m×[T_M+(1+d)×B])这部分带宽。

当 m 和 T_M 参数都非常小，而且 B 等于 1 时，这部分带宽并不值得仔细计算。但是事实并非如此，随着计算机主频的不断攀升，T_M 参数的相对值一直在不断提高；而主存储器的快速增加，L1 Cache 相对变得更小，使得 m 参数进一步提高。这使得 Miss 时使用的带宽备受关注。最自然的想法是使用流水方式将多个 Outstanding Cache Miss 并行处理，使得多个 Miss 可以 Amortize T_M 参数，从而最终有效提高 BW_S 参数。

这一构想使 David Kroft 在 1981 年正式提出了 Lockup-Free Cache(Non-Blocking Cache)[51]这个重要的概念。Kroft 建议设置一组 MSHR(Miss Information/Status Holding Registers)暂存 Miss 请求。其中每一个 MSHR 与一个 Outstanding Miss 相对应。当 MSHR 的个数为 N 时，该 Cache 结构即为 Non-Blocking(N) Cache。Non-Blocking(N) Cache 可以并行处理 N 个 Outstanding Cache Miss，而且在处理 Cache Miss 的同时可以并行处理 Cache Hit，修正了 Blocking Cache 存在的诸多问题，使得 Superscalar 处理器得以继续。

David Kroft，这个毕生都没有发表过几篇文章的，也许是普通的再也不能普通的工程师或是学者，让众多拥有几十篇，甚至几百篇的教授和 Fellow 汗颜。他在 Retrospective 中这样评价着这个算法。

How does one begin to describe the dreams, thoughts and fears that surround a discovery of a different view of some old concepts or the employment of old accepted methodology to new avenues? It is probably best to start the account by describing the field of Computer Architecture, in particular, the area of hierarchical memory design, that was prevalent around and before the time the ideas came to light.

...

As indicated at the beginning, a discovery is just a different look at some old concepts or the use of some old concepts for new mechanisms. Mine was the latter [52].

从今天对 Cache 的认知上看，David Kroft 的建议顺理成章，几乎每一个人都可以想到。而在 David Kroft 提出这个设想的那个年代，Miss Blocking 几乎是理所应当的，可以极大简化 Cache 的设计。David Kroft 所以能够提出这个建议更深层次的原因，是他具有向已知的结论挑战的勇气。

我目睹过一些填鸭式的魔鬼训练营。那里逼迫着诸多才智之士强记着几乎不会出错的结论，强化着比拼编码速度的各类实现。直接使用结论比从学习基础的原理并推导出结论快得多。按部就班是过于漫长的，是无法速成的。这样做似乎是一条捷径。只是伏久者，飞必高；开先者，谢独早。世上没有捷径，起初的捷径必为将来付出惨痛的代价。

依靠速成得来的这些结论和已知实现虽然可以被信手拈来，用于构建各类设计，却更加容易使人忽略一些更加基础的知识。最重要的是这些结论会很容易在心中生根发芽。在经过多次反复后，这些才智之士不会没有挑战结论的勇气，只是失去了挑战结论的想法。诸恶之恶，莫过于此。以中国的人口基数，却远没有获得与此对应的成就，在 Cache Memory，处理器，计算机科学，在更多的领域。生于斯，长于斯，愧于斯。

David Kroft 提出了 Non-Blocking Cache 的同时，打开了潘多拉魔盒，这个领域奋战着的 Elite 万劫不复。在 Lockup-Free Cache 出现后的不长时间内，经历了许多修改，使得 CPU Core-L1 Bus，L1-L2 Bus，其后的 Bus 设计进一步复杂化。

David Kroft 提出的算法只有 4 页纸的描述，更多的是春秋笔法，以至于很难根据这个论文，实现出可用的 Non-Blocking Cache 结构。这些已不再重要。David Kroft 当时的想法基于当时的认知，很难突破当时的认知，绝非完美，可能只适用于 Statically-Scheduled 架构，依然指明了前进之路。其后 Dynamically-Scheduled 架构进一步改进了 Lockup-Free Cache 结构。这也是现代微架构在进行 Cache 设计时常用的方法。

对于 Dynamically-Scheduled 的处理器，Non-Blocking Cache 有多种实现方法，Address Stack，Load Queue，Address-Reorder Buffer 或者其他方法。本篇仅介绍 Address Stack 实现机制，该机制的实现较为直观，其基本组成结构如图 2-21 所示。

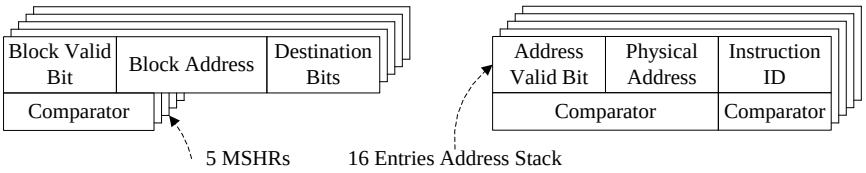

图 2-21 基于 Address Stack 的 Non-Blocking Cache[53]

以上 Non-Blocking Cache 的实现中，包含 5 个 MSHRs，Address Stack 的 Entries 数目为 16。在这种实现中，如果微架构访问存储器出现 Cache Miss 时，需要首先判断本次访问所需要的数据是否正在被预取到 Cache Block 中，该功能由 MSHRs 寄存器组及相应控制逻辑实现；当数据从存储器子系统中返回时，需要解决所有依赖这个数据的 Pending Cache Miss，该功能由 Address Stack 及相应的控制逻辑实现。

每个 MSHR 至少包含 3 个数据单元，Block Valid Bit，Block Address 和 Destination Bits。其中 Block Valid Bit 有效表示地址为 Block Address 的 Cache Block 发生过 Fetch，从 IC Design 的角度上看，Cache Block 是有地址的，上文已经对此进行过描述。

当 Cache Miss 时，Cache 流水线将首先并行查找 MSHRs 寄存器组的 Block Address，判断即将访问的 Cache Block 是否正在被 Fetch。如果在 MSHRs 寄存器组中命中，本次 Cache Miss 即为 Secondary Cache Miss，否则为 Primary Cache Miss。

对于 Primary Cache Miss，需要在 MSHRs 寄存器组中获得一个空闲 Entry，并填写相应的 Block Address，设置 Block Valid Bit，同时向其下的存储器系统发送 Fetch 请求；如果当前 MSHRs 寄存器组中没有空闲 Entry，之后微架构将不能继续 issue 存储器请求，Cache 流水线将被 Stall，直到之前发出的 Fetch 请求返回，释放 MSHRs 寄存器组中的对应 Entry。

Secondary Miss 的处理相对较为复杂。此时需要视不同情况讨论。不同的微架构采用了不同的策略处理 Secondary Miss。在 Kroft 的建议中，Cache Block 的管理按照 Word 进行分组。因此只有 Secondary Miss 访问的 Word 并与 Pending Miss 请求完全一致时，Cache 流水线才会 Stall，这种 MSHRs 也被称为 Implicitly Addressed MSHRs[54]。与此对应的概念还有 Explicitly Addressed MSHRs，采用这种组成方式，可以解决对同一地址进行访问时，不会 Stall Cache 流水线，其详细实现见[54]。

一个很容易想到的，更加高效的方法是将 Primary Miss 和其后若干个 Secondary Miss 合并为一个 Fetch 请求。但是这种 Merge 是有条件的，需要更多的 Buffer 实现。这种方式不仅带来延时，更为 Memory Consistency 制造了不小的麻烦。

当 Cache Miss 引发的 Fetch 请求最终返回，并从其下的存储器系统中获得 Cache Block 之后。Cache 流水线将并行检查 MSHRs 寄存器组，并获得对应的 Destination Bits，并依此判断，所获得的数据是发向指令 Cache，数据 Cache，还是某个定点或者浮点寄存器。然而在 Dynamically-Scheduled 架构中，为了维护 Cache Consistency，问题更加复杂。

在 Dynamically-Scheduled 架构中，由于 Load Speculation 的存在，存储器读操作在微架构出现 Mispredicted 的转移指令，或者 Exception 时，需要抛弃之前的 Speculative Load 请求，此时可以采用 Address Stack 处理这种情况。

Address Stack 为 Fully-Associative Buffer，由 Address Valid Bit, Physical Address, Instruction ID 三个字段组成。当微架构发射存储器读写指令时，首先计算 Physical Address，之后并行查找 Address Stack，获得空余的 Entry。如果 Address Stack 中没有空余的 Entry，将产生 Structural Hazard，已发射的存储器指令不断重试，直到获得空余的 Entry；如果有空余，则设置该 Entry 的 Address Valid Bit，填写 Physical Address，并向其下存储器系统发送 Fetch 请求。

当 Fetch 的 Cache Block 返回时，会并行查找 Address Stack，如果 Match，返回的数据将进入相应的 Cache，可能还会传递给某个寄存器。指令流水线的设计者可以决定是在存储器读写指令在 Complete 还是在最后的 Commit 阶段时，才真正释放 Address Stack 的 Entry。前者实现较为复杂，后者较为简单。

在一切都是“正常”的情况下，返回的 Cache Block 并行查找 Address Stack 似乎是冗余的。但是考虑 Misprediction 转移和 Exception 处理之后，我们不能得出这样的结论。当发生这两种情况时，在刷新微架构状态时，所有 Speculative Store/Load 指令也需要被刷新。

使用 Address Stack 方法设计 Non-Blocking Cache 时，处理这些异常需要清除相应 Entry 的 Address Valid Bit。当数据返回时，将无法在 Address Stack 中找到合适的 Entry。对于存储器读请求，数据将不会写入 Physical Register，对于存储器写请求，数据不会真正写入，以避免各类潜在的错误。

由以上描述，可以发现这种 Non-Blocking 设计由两部分组成，一个是 Cache Miss 时使用的 Address Queue，和处理数据返回使用的 Address Stack。不同的微架构使用的 Non-Blocking Cache 设计方法大同小异，要点是设置专门的 Buffer 处理多个 Cache Miss，采用流水方式，使多个 Cache Miss 请求最终可以 Amortize T_M 参数，最终有效提高 BW_S 参数。

Non-Blocking Cache 的概念不难理解，重要的是实现。总线技术的不断发展，使得这些实现愈发困难。在现代微架构中，用于 Cache 间互联的总线，大多被分解为若干个子总线，由 Data，Request，Ack 和 Snoop 四条子总线组成，而在这四条子总线中还会继续分级，以进一步扩展带宽。这些变化使得 Non-Blocking Cache 的话题离学术界渐行渐远。

在系统总线设计中出现的这些变化极大增加了 Cache 流水线的设计难度。我们进一步考虑 Cache 的层次结构从 L1，L3 到 EDRAM，进一步考虑处理器系统从 CMP，SMP 到 ccNUMA，进一步考虑 Memory Consistency 从 Strict consistency，Sequential Consistency 到 Weak Consistency，进一步考虑各类细节和不断攀升的 Cache 运行频率，会使最具睿智的一群人为之魂系梦牵。他们明白世上再无任何数字逻辑能够如此忘情。

Cache 流水线与协议的复杂，触发了如何解决 State-Explosion 的问题，这使得 Cache 的 Verification 也成为了一个专门的学问，引发了无尽的讨论。每在 Cache 流水线面前，心中想的总是成千上万的服务器日夜不息的忙碌，持续追求完美的设计在最高的工艺上不断进行的深度定制。

面对着这一切，任凭谁都显得那样的微不足道。

第3章 Coherency and Consistency

本章出现的 Coherency 指 Cache Coherency，Consistency 指 Memory Consistency。许多工程师经常混淆这两个概念，没有建立足够准确的 Memory Consistency 概念。Consistency 与 Coherency 之间有一定的联系，所关注的对象并不等同。

Memory Consistency 的实现需要考虑处理器系统 Cache Coherency 使用的协议。除了狭义 Cache 之外，在处理器系统中存在的广义 Cache 依然会对 Memory Consistency 模型产生重大影响。Memory Consistency 和 Cache Coherency 有一定的联系，但是并不对等。这两部分内容相对较为复杂，可以独立成篇。有些学者认为 Cache Coherency 是 Memory Consistency 的一部分[55]，更为准确的说是 Memory Coherency 的一部分。

我们首先给出 Memory Coherency 的定义。Memory Coherency 指处理器系统保证对其存储器子系统访问 Correctness。我们并不关注对处理器私有空间的存储器访问，仅考虑共享空间的这种情况，即便在这种情况下定义 Correctness 依然很困难。

在一个 Distributed System 中，共享存储器空间可能分布在不同的位置，由于广义和狭义 Cache 的存在，这些数据单元存在多个副本；在 Distributed System 中，不同处理器访问存储器子系统可以并发进行，使得 Memory Coherency 层面的 Correctness 并不容易保证。

我们假设在一个 Distributed System 中含有 n 个处理器分别为 P_1~P_n，P_i 中有 S_i 个存储器操作，此时从全局上看可能的存储器访问序列有$(S_1+S_2+…+S_n)! / (S1! \times S_2 ! \times…\times S_n)$种组合[56]。为保证 Memory Coherency 的 Correctness，需要按照某种规则选出合适的组合。这个规则被称为 Memory Consistency Model，也决定了处理器存储器访问的 Correctness。这个规则需要在 Correctness 的前提下，保证操作友好度的同时，保证多处理器存储器访问较高的并行度。

在不同规则定义之下，Correctness 的含义并不相同，这个 Correctness 是有条件的。在传统的单处理器环境下，Correctness 指每次存储器读操作所获得的结果是 Most Recent 写入的结果。在 Distributed System 中，单处理器环境下定义的 Correctness，因为多个处理器并发的存储器访问而很难保证。在这种环境下，即便定义什么是 Most Recent 也很困难。

在一个 Distributed System 中，最容易想到的是使用一个 Global Time Scale 决定存储器访问次序，从而判断 Most Recent，这种 Memory Consistency Model 即为 Strict Consistency，也被称为 Atomic Consistency。Global Time Scale 不容易以较小的代价实现，退而求其次采用每一个处理器的 Local Time Scale 确定 Most Recent 的方法被称为 Sequential Consistency[56]。

与 Sequential Consistency 要求不同处理器的写操作对于所有处理器具有一致的 Order 不同，Causal Consistency 要求具有 Inter-Process Order 的写操作具有一致的 Order，是 Sequential Consistency 的一种弱化形式。Processor Consistency 进一步弱化，要求来自同一个处理器的写操作具有一致的 Order 即可。Slow Memory 是最弱化的模型，仅要求同一个处理器对同一地址的写操作具有一致的 Order[56]。

以上这些 Consistency Model 针对存储器读写指令展开，还有一类目前使用更为广阔的 Model。这些 Model 需要使用 Synchronization 指令，这类指令也被称为 Barrier 指令。在这种模型之下，存储器访问指令被分为 Data 和 Synchronization 指令两大类。其中 Synchronization 指令能够 Issue 的必要条件是之前的 Data 指令执行完毕，其他指令在 Synchronization 指令执行完毕前不能进行 Issue。在 Synchronization 指令之间的存储器访问需要依照处理器的约束，可以 Reordered 也可以 Overlapped。

在这种 Model 中，Data 指令的 Order 并没有受到关注，所有规则仅针对 Synchronization 指令起作用，也因此产生了 Weak Consistency，Release Consistency 和 Entry Consistency[55][56] 三个主要模型。这些模型将在下文做进一步的说明。

对于不支持 Network Partitions 的 Distributed System，可以在实现 Strict Data Consistency 的同时实现 Availability[①]。而对于一个较大规模的 Distributed System 中，Network Partitions 是一个先决条件。例如在一个大型系统中，所使用的 Web 服务器和数据库系统已经分布在世界上的很多角落，Network Partitions 已经是一个事实。

Consistency，Availability 和 Partition-Tolerance 三者不可兼得[57]，这使得在一个 Network Partitions 的 Distributed System 中，必须在 Consistency 和 Availability 之间进行 Trade-Off，也引出了 Eventually Consistent 模型[58]。这种模型是另一种 Weak Consistency Model，基于一个数据在较长时间内没有发生更新操作，所有数据副本将最终一致的假设。

Eventually Consistent 在 DNS(Domain Name System)系统中得到了较为广泛的应用，也是 Distributed Storage 领域的用武之地。这些内容在 Cloud 崭露头角之后迅速成为热点，却不是本篇重点。我依然相信在 Cloud 相关领域工作的人必然可以在处理器存储器子系统的精彩中获得进一步前进的动力，可能是源动力。

这些内容超出了本书的覆盖范围，我们需要对 Cache Coherency 做进一步说明。从上文中的描述可以发现 Memory Consistency 关注对多个地址进行的存储器访问序列；Cache Coherency 单纯一些，关注同一个地址多个数据备份的一致性。不难发现 Cache Coherency 是 Memory Coherency 的基础。

Cache Coherency 要求写操作必须最终广播到参与 Cache Coherency 的全部处理器中，即 Write Propagation；同时要求参与 Cache Coherency 的处理器所观察到的对同一个地址的写操作，必须按照相同的顺序进行，即 Write Serialization。

Write Propagation 有 Invalidate-Based 和 Update-Based 两种实现策略。Invalidate-Based 策略的实现首先是确定一次存储器访问是否在本地 Cache Hit，如果 Hit 而且当前 Cache Block 状态为广义的 Exclusive/Ownership，不需要做进一步的操作；否则或者在 Cache Miss 时需要获得所访问地址的 Exclusive/Ownership。此时进行存储器访问的 CPU 向参与 Coherency 的所有 CPU 发送 RFO(Read for Ownership)广播报文，这些 CPU 需要监听 RFO 报文并作出回应。

如果 RFO 报文所携带的地址命中了其他 CPU 的 Cache Block，需要进一步观察这个 Cache Block 所处的状态，如果这个 Cache Block 没有被修改，则可以直接 Invalidate；否则需要向发出请求的 CPU 回应当前 Cache Block 的内容，在多数情况下，被修改的 Cache Block 只有一个数据副本。这种方法在 Share-Bus 的处理器系统中得到了最广泛的应用，如果存储器访问连续命中本地 Cache，命中的 Cache Block 多处于 Exclusive 状态，不需要使用 RFO 报文，因此不会频繁地向处理器系统发出广播操作，适合 Write-Back 方式。

Update-based 策略的实现通常使用 Central Directory 维护 Cache Block 的 Ownership，在 Cache Block Miss 时，需要 Write Update 其他 CPU Cache Block 存在的副本，可以视网络拓扑结构同时进行多个副本的同步，即便如此所带来的 Bus Traffic 仍较严重，适用于使用 Directory 进行一致性操作的大型系统。如果进一步考虑实现细节中的各类 Race Condition，完成这种方式的设计并不容易。

除了 Invalidate 和 Update-Based 策略之外，Cache Coherency 可以使用 Read Snarfing 策略。在这种实现方式中，可以在一定程度上避免再次 Read 被 Write-Invalidate 的 Cache Block 时，引发的 Miss。当一个 CPU 读取一个 Data Block 时，这个读回应除了需要发给这个 CPU 之外，还需要更新其他 CPU 刚刚 Invalidate 的 Cache Block。在实现中，其他 CPU 可以监控这个读回应的地址与数据信息，主动更新刚刚 Invalidate 的数据拷贝[59]。

在参考文献[59]的模式中，使用 Read Snarfing 策略可以减少 36~60%的 Bus Traffic。但是这种方式的实现较为复杂，目前尚不知在商业处理器是否采用过这样的实现方式。在学术领域，Wisconsin Multicube 模型机曾经使用过 Read Snarfing 策略[61]。

[①] Availbility 指来自任何一个处理器的读写请求一定可以获得 Response。

Write Serialization 的实现需要使用 Cache Coherent Protocol 和 Bus Transaction。类似 RFO 这样的广播报文必不可少。在使用 Share Bus 和 Ring-Bus 互连时，较易实现 Write Serialization。Directory 方式在对同一个地址 Cache Block 的并发写时需要使用额外的逻辑处理 ACK Conflict，这些逻辑大多设置在 Home Agent/Node 中。

3.1 Cache Coherency

Cache Coherency 产生的原因是在一个处理器系统中，不同的 Cache 和 Memory 可能具有同一个数据的多个副本，在仅有一个数据副本的处理器系统中不存在 Coherency 问题。维护 Cache Coherency 的关键在于跟踪每一个 Cache Block 的状态，并根据 CPU Core 的读写操作及总线上的相应 Transaction，更新 Cache Block 的状态，借此维护 Cache Coherency。Cache Coherency 可以使用软件或者硬件方式保证。

在使用软件方式维护时，CPU 需要提供专门的显式操作 Cache 的指令，包括 Cache Block Copy，Move，Eviction 和 Invalidate 等指令，多数微架构都提供了这样的指令，如 PowerPC 处理器设置的 dcbt，dcbf，dcba 等指令[43]。

程序员可以使用这些指令，维护处理器系统的 Cache Coherency。在进行 DMA 操作之前，可以将数据区域与主存储器通过软件指令保证 Cache Coherency，进行 DMA 操作时，不需要硬件来维护 Cache Coherency。在某种情况下，使用这种方式可以提高数据传送效率。

软件维护 Cache Coherency 的优点是硬件开销小，缺点在多数情况下对性能有较大影响，而且需要程序员的介入。多数情况下 Cache Coherency 由硬件维护。不同的处理器使用不同的 Cache Coherency Protocol 实现 Cache Coherency。这些 Protocol 维护一个有限状态机 FSM(Finite State Machine)，根据存储器读写指令或者 Bus Transaction，进行状态迁移和相应的 Cache Block 操作，隐式保证 Cache Coherency，不需要程序员的介入。

根据 Cache Coherency Protocol 维护 Cache Block 状态方法的不同，处理器可以使用 Bus Snooping 和 Directory 这两大类机制。这两类机制的主要区别在于 Directory 机制全局统一管理不同 Cache 的状态；而在 Bus Snooping 机制中，每个 Cache 分别管理自身 Cache Block 的状态，并通过 Interconnection 进行不同 Cache 间的状态同步。

无论采用哪种机制，Coherency Protocol 所要求的 Cache Block 操作都可以通过 Invalidate，Update 或者 Read Snarfing 策略完成。Update 和 Read Snarfing 策略需要更多的总线操作，对带宽和延迟都有很大影响，多数 Cache Coherency Protocol 采用了 Invalidate 策略。

最为经典的总线监听协议 Write-Once[60]由 James Goodman 于 1983 年提出，是在 x86，ARM 和 Power 处理器中大行其道的 MESI Protocol(也叫 Illinois Protocol)的前身和一种变体，或者说是一种具体实现。

Write-Once Protocol 的实现关键在于其使用的特殊的 Cache 回写机制，即 Write Once。Write-Once 是 Write-Back 和 Write-Through 的综合。当使用这种机制时，对一个 Cache Block 进行第一次回写时，采用 Write-Through 策略将数据同时回写到 Cache 和主存储器，之后的写操作采用 Write-Back，只回写到 Cache 而不回写到主存储器。

这种设计有利于在带宽和 Coherency Protocol 的复杂度之间取得均衡。Write-Back 机制能够有效节约带宽，但是由于主存储器中并没有最新的数据副本，增加了维护 Cache Coherency 的开销。Write-Through 则相反。有关 Write-Back 和 Write-Through 的详细信息可以继续阅读本篇的后续章节。

通过之前的描述可以发现，在任何一种 Cache Coherency Protocol 中，每个 Cache Block 都有自己的一个状态字段。而维护 Cache Coherency 的关键在于维护每个 Cache Block 的状态域。Cache Controller 通常使用一个状态机来维护这些状态域。

使用 Write-Once 回写机制实现 Cache Coherency 时，每一个 CPU Core 中的 Cache Block 需要设置 4 个状态位，用以识别当前 Cache Block 的状态，处于这些状态的 Cache Block 在收到不同的输入后，将进行状态迁移，以保证 Cache Coherency。

- Invalid 位。表示当前 Cache Block 不含有有效数据，是个无效 Cache Block。
- Valid 位。表示当前 Cache Block 的数据为最新，在主存储器中拥有该 Cache Block 的数据副本。在 Eviction 时不需要回写主存储器。其他 CPU Core 的 Cache 中可能含有该 Cache Block 的数据。在这个 Cache Block 中的数据可能与其他 CPU Core 中的 Cache 共享，各个共享数据副本都为最新的值，即与主存储器同步。在进行存储器读操作后，Cache Block 可能处于该状态。
- Reserved 位有效表明该 Cache Block 中的数据是最新的，在主存储器中拥有该 Cache Block 的数据副本。在 Eviction 时不需要回写主存储器。其他 CPU Core 的 Cache 中不能含有该 Cache Block 的数据。这是由第一次对该数据进行写操作所达到的状态，读者可以简单回忆 write-once 的写回机制。
- Dirty 位。表示该 Cache Block 中的数据是最新的，而且只有该 Cache Block 拥有这个最新的数据副本，而且与主存储器不同步。主存储器和其他 CPU Core 的 Cache 中没有这个数据副本。这是通过对该数据反复进行写操作所达到的状态，读者可以再次回忆 Write-Once 的回写机制。

根据以上这几种状态，我们简要分析基于 Write-Once 协议的 Cache Coherency Protocol，其 FSM 描述如图 3-1 所示，其中右图是使用 Write-Once 协议的处理器系统示意图。在该处理器系统中含有两个处理器，而且只有一级 Cache，分别为 C0 和 C1，通过共享总线方式与主存储器进行连接。

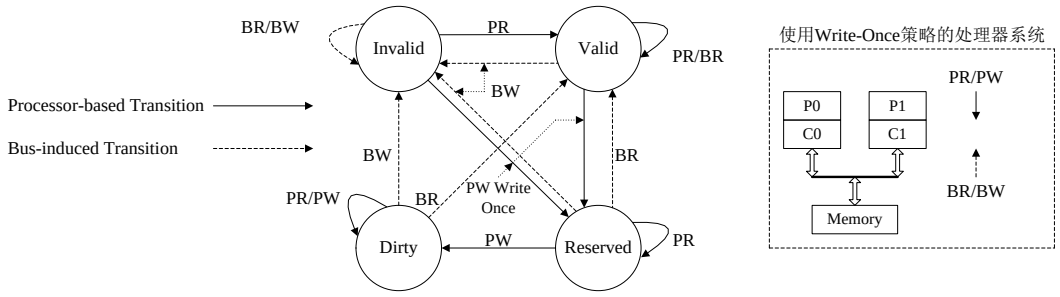

图 3-1 Write-Once Transition Diagram[①]

在学习基于 Write-Once 策略的 Cache Coherency Protocol 时，需要将状态机迁移与处理器互联拓扑进行对照。在 Write-Once 状态机中含有四个状态，分别为 Invalid，Valid，Dirty 和 Reserved。Cache Block 处于这些状态时，可以接收到四种输入请求。

这四种输入请求也是不同处理器的 Cache Controller 所接收到四种读写操作，即状态机中的输入信号，包括总线读 BR(Bus Read)，总线写 BW(Bus Write)，处理器读 PR(Processor Read)和处理器写 PW(Processor Write)。当 Cache Controller 监听到对某个 Cache Block 的操作的时候，将根据当前 Cache Block 的状态进行迁移。

总线读 BR 表示 Cache Controller 监听到了来自总线的读操作。该操作的产生原因是处理器读取的数据不在其本地的 Cache 中，需要查找主存储器或者其他处理器的 Cache。如果此时另外一个处理器的 Cache 含有该数据最新的副本，该处理器的 Cache Controller 将提供这个最新的副本，即 Data Refill，当该数据副本为 Dirty 时，需要将数据回写到主存储器，并将

[①] 图 3-1 来自 **Error! Reference source not found.**Yale N. Patt 的讲义。

自身的状态迁移为 Valid，表示这个数据被多个 Cache 共享。

总线写 BW 表示某个 CPU 的 Cache Block 需要回写主存储器。如果总线写 BW 是第一次对 Cache Block 进行写操作时产生，此时使用的是 Write-Through 策略，而不是 Write-Back。特别注意的是，Cache Block 因为 Replacement 而回写到主存储器时不会产生总线写 BW。

如果某个处理器的 Cache Controller 监听到了总线写 BW，并且在自身 Cache 中含有数据的一个副本。此时该 Cache Block 处于 Valid，Reserved 或者 Dirty 状态，首先需要向发起总线写的处理器提供这些数据，即进行 Data Refill 操作，之后 Invalidate 这个 Cache Block 中的副本，并将状态迁移到 Invalid。

处理器读 PR 是处理器读取本地 Cache 时产生的 Transaction。如果本地 Cache 没有命中，或者处于 Invalidate 状态，将通过 Data Refill 操作获得相应的数据，此时该 Cache Controller 将发出总线读 BR，从其他 CPU Core 的 Cache 中获得数据副本，之后将状态迁移到 Valid。如果此时 Cache Block 为 Valid，Reserved 或者 Dirty 状态时为 Cache 命中，维持原状态不变。

处理器写 PW 是处理器向本地 Cache 写数据时产生的 Transaction。在 Write-Once 策略使用的特殊回写机制中，如果处理器是第一次对该数据进行写操作，该 Cache Block 状态将迁移为 Reserved，表明只有当前 Cache Block 和主存储器拥有数据副本。

此时 Cache Controller 将使用总线写 BW Invalidate 其他处理器中的数据副本。如果是对该数据的后续写操作，则只写回 Cache，从而使状态迁移为 Dirty，表明只有当前本地 Cache 含有该数据副本，此时不会产生 BW 信号。另外需要注意的是，如果某个数据因为某种原因被替换到主存储器，之后又被再次读入 Cache 时，对它的写操作也相当于第一次，相当于维护对同一个数据的新的 Cache Coherency 周期。

除了从 Cache Controller 的角度分析这些状态迁移之外，还可以从处理器的角度认识 Write-Once 机制。处理器访问 Cache 时，可能会出现 Read Miss，Read Hit，Write Miss 和 Write Hit 这四种情况。我们分别讨论这四种情况。

如果发生 Read Hit，命中的 Cache Block 一定处于 Valid，Reserved 或者 Dirty 状态。Read Hit 不会改变该 Cache Block 的状态。如果发生 Read Miss，则该 Cache Block 的当前状态或者是 Invalid 或者不在本地 Cache 中。此时 Cache Controller 会产生一个总线读 BR，从其他处理器的 Cache 获得所需的数据，该数据会同时存在于主存储器中。此时将 Cache Block 状态将迁移为 Valid，表示此时 Cache Block 中的数据和主存储器一致，并且其他处理器的 Cache 可能拥有该数据的副本。

如果发生 Write Hit，则根据该写操作是第一次还是后续写，将状态分别迁移到 Reserved 或者 Dirty 状态。如果是第一次写操作，进行 Write-Through 操作的同时也会产生总线写 BW，以保证本地 Cache 中的数据是除主存储器之外唯一的副本。如果发生 Write Miss，则该 Cache Block 的当前状态或者是 Invalid 或者不在本地 Cache 中，这次写操作一定是第一次写，因此将迁移为 Reserved。

Write-Once 协议的实现要点在于第一次写操作采用 Write-Through，并通过 Write-Through 产生的总线写 BW，Invalidate 其他处理器 Cache 中的相应数据副本，使本地 Cache 中的数据成为除了主存储器以外的唯一副本，从而维护 Cache Coherency。

MESI Protocol，也被称为 Illinois Protocol[65]，是大多数 SMP 处理器维护 Cache Coherency 采用的策略，其实现与 Write-Once Protocol 大同小异，虽然这种 Protocol 出现得相对较晚，却得到了更广泛的应用。目前主流的处理器，如 x86，Power，MIPS 和 ARM 处理器，均使用类 MESI 协议维护 Cache Coherency。

MESI Protocol 的得名源于该协议使用的 Modified，Exclusive，Shared 和 Invalid 这四个状态，这些状态对应 Write-Once 协议使用的 Dirty, Reserved, Valid, Invalid 四个状态，所表达的含义也大致相同，但是仍然有些微小区别。标准的 MESI Protocol 还有一些变种，如 MOESI

和 MESIF 等一系列协议。

值得额外关注的是，一个 Cache Block 除了要使用 MESI 这些基本的状态位之外，还包含许多辅助状态位，共同维护 Cache Coherency。考虑多级 Cache 间的联系和各种 Race Condition 的处理情况，MESI Protocol 的实现比想象中难出许多。在标准 Illinois Protocol 中定义了以下四种状态。

- M(Exclusive-Modified)有效表示当前 Cache Block 中包含的数据与主存储器不一致，而且仅在当前 Cache 中有正确的副本。
- E(Exclusive-Unmodified)有效表示当前 Cache Block 中的数据在当前 Cache 及主存储器中一致。M 和 E 状态都表示 Exclusive 状态，一个 Dirty，一个 Clean。
- S(Shared-Unmodified)有效表示当前 Cache Block 中的数据至少在当前 Cache 及主存储器中有效，在其他处理器的 Cache 中也可能含有正确的副本。
- I(Invalid)：当前 Cache Block 无效，不包含有效数据。

Illinois protocol 和 Write-Once 协议的最大不同在于回写机制。Illinois Protocol 在写命中时的策略只有 Write-Back，而 Write-Once 区分第一次写和后续写。在使用 Write-Once 协议时，第一次写时采用 Write-Through 策略非常重要，正是通过第一次 Write-Through 产生的总线写 BW Invalidate 其他副本从而维护 Cache Coherency。

在使用 Write-Back 策略的 Illinois Protocol 中，一个处理器进行 Cache 写操作时，将主动广播一个 Invalidate Transaction，也被称为 RFO(Request For Ownership)。Illinois Protocol 根据 Write Miss 与 Write Hit 两种情况，将 Write-Once 协议使用的总线写 BW，分解为 BRI(Bus Read with Invalidate)和 BI(Bus Invalidate Observed)两类 Invalidate 信号。下面着重分析状态机中与 BRI 和 BI 相关的状态迁移，其余情况和 Write-Once 协议的处理较为类似。Illinois Protocol 的 FSM 描述如图 3-2 所示。

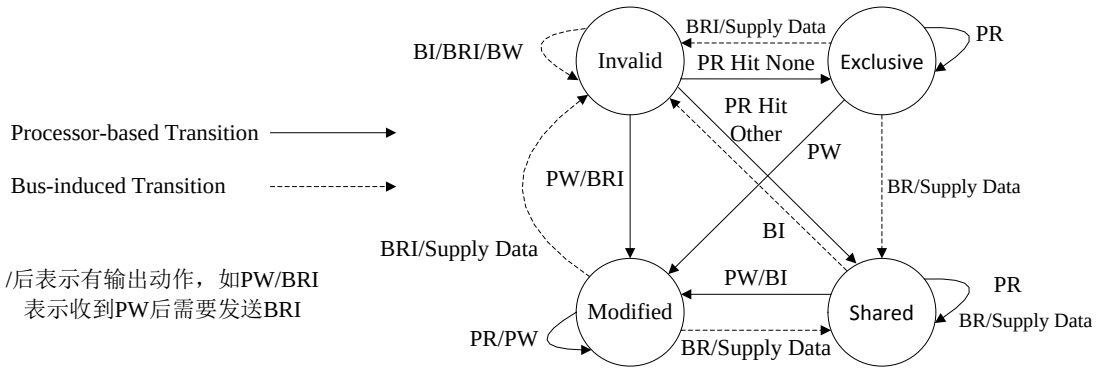

图 3-2 Illinois Protocol Transition Diagram[①]

当 Cache Block 处于 Invalid 状态时，如果 Cache Controller 收到来自总线的信号时，如 BR，BRI 和 BI，将保持原状态不变；如果 Cache Controller 收到处理器写 PW 时，将出现 Write Miss，此时 Cache Controller 会广播一个 BRI 信号，表示需要从其他处理器中读取该数据，进行 Data Refill，并且 Invalidate 其他所有副本，这也是该操作也被称为 Bus Read with Invalidate 的原因，之后该 Cache Block 的状态迁移为 Modified，表示拥有唯一最新的副本。

当 Cache Block 处于 Shared 状态时，如果收到处理器写 PW 信号，会广播一个 BI 信号，表示本地 Cache 有最新的副本，但是在进行写操作前需要 Invalidate 其他副本，之后该 Cache Block 的状态将迁移为 Modified。如果处于 Shared 状态的 Cache Block 收到来自总线的 BRI 或者 BI Transaction 时，将 Invalidate 本地 Cache 副本，并且状态迁移为 Invalid；如果收到总

[①] 图 3-2 来自 **Error! Reference source not found.**Yale N. Patt 的讲义。

线读时 BRI 需要向请求段提供该数据的副本以做 Data Refill。

当 Cache Block 处于 Exclusive 状态时，在处理器系统中的所有 Cache 中只有本地 Cache 拥有数据副本，因此不可能收到来自总线的请求 BI；如果收到来自总线的请求 BRI，需要向请求方提供该数据的副本以做 Data Refill，并且迁移至 Invalid 状态。

当 Cache Block 处于 Modified 状态时，在处理器系统中的所有 Cache 中只有本地 Cache 拥有数据副本，因此也不可能收到来自总线的请求 BI；如果收到来自总线的请求 BRI，需要向请求方提供该数据的副本以做 Data Refill，并且迁移至 Invalid 状态。

MESI 协议有一个重要的变种，即 MOESI。DEC Alpha21264，AMD x86，RMI Raza 系列处理器，ARM Cortex A5 和 SUN UltraSPARC 处理器使用了这种协议。基于 MOESI 协议的 FSM 描述如图 3-3 所示，该模型基于 AMD 处理器使用的 MOESI 协议。不同的处理器在实现 MOESI 协议时，状态机的转换原理类似，但是在处理上仍有细微区别。

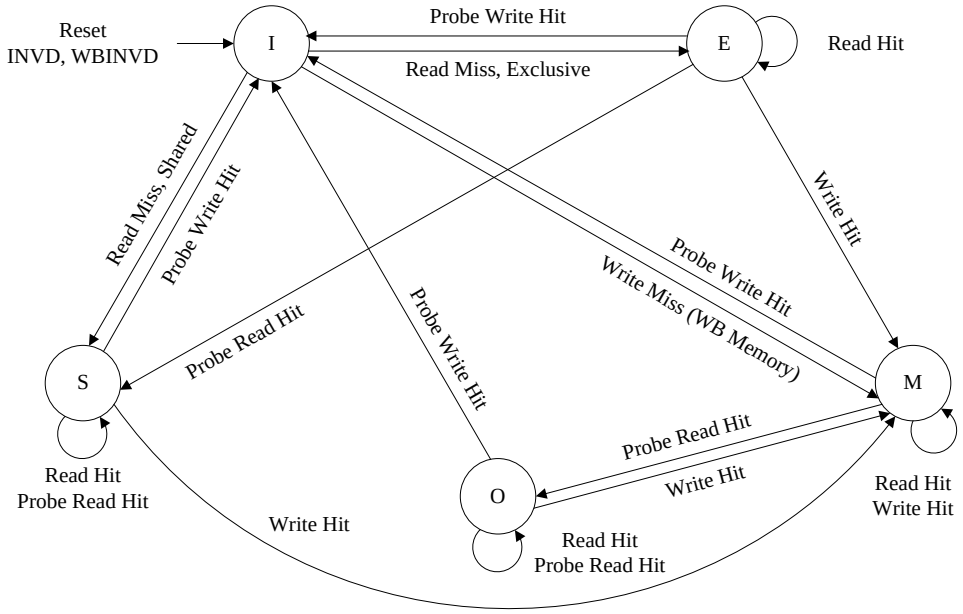

图 3-3 MOESI Protocol Transition Diagram[62]

MOESI 协议引入了一个 O(Owned)状态，并在 MESI 协议的基础上，进行了重新定义了 S 状态，而 E，M 和 I 状态和 MESI 协议的对应状态相同。

- O 位。O 位为 1 表示在当前 Cache 行中包含的数据是当前处理器系统最新的数据拷贝，而且在其他 CPU 中可能具有该 Cache 行的副本，此时其他 CPU 的 Cache 行状态为 S。如果主存储器的数据在多个 CPU 的 Cache 中都具有副本时，有且仅有一个 CPU 的 Cache 行状态为 O，其他 CPU 的 Cache 行状态只能为 S。与 MESI 协议中的 S 状态不同，状态为 O 的 Cache 行中的数据与主存储器中的数据可以不一致。
- S 位。在 MOESI 协议中，S 状态发生了微小变化。当 Cache Block 状态为 S 时，其包含的数据并不一定与存储器一致。不存在状态为 O 的副本时，Cache Block 中的数据与存储器一致；存在状态为 O 的副本时，Cache Block 中的数据与存储器不一致。

在 MOESI 模型中，Probe Read 表示主设备从其他 CPU 中获取数据拷贝的目的是为了读取数据；Probe Write 表示主设备从其他 CPU 中获取数据拷贝的目的是为了写入数据；Read Hit 和 Write Hit 表示当前访问在本地 Cache 中获得数据副本；Read Miss 和 Write Miss 表示当前访问没有在本地 Cache 中获得数据副本；Probe Read Hit 和 Probe Write Hit 表示当前访问在

其他 CPU 的 Cache 中获得数据副本。

我并不喜欢图 3-3 的 FSM，更倾向使用图 3-1 和图 3-2 中的描述，下一版将统一使用 BR/BW 和 PR/PW 模型。MESI Protocol 广泛应用与 Bus-Snooping 结构，CMP 间的 Cache Coherency 常使用 Directory Protocol。

与 Bus-Snooping 相比，Directory Protocol 所带来的 Latency 较长，但是 Bus Traffic 较少，下一版会主要基于 Stanford FLASH 和 DASH[63][64]去书写这部分内容。Intel Sandy Bridge EP 系列处理器也使用了这种结构，目前没有公开详细实现。AMD 的 Magny-Cours 中也使用了这种方式进行 Cache Coherency，但是也没有公开最新的 Bulldozer 处理器的实现。还有一个适用与 Ring-Bus 的 Token Protocol 值得关注，准备下一版书写。目前暂时如此。

3.2 Memory Consistency 1

3.3 Memory Consistency 2

3.4 Memory Consistency 3

3.5 Memory Consistency 4

第4章 Cache 的层次结构

我第一次接触存储器瓶颈这个话题是在上世纪九十年代，距今已接近二十年。至今这个问题非但没有缓和的趋势，却愈演愈烈，进一步发展为 Memory Wall。在这些问题没有得到解决之前，片面的发展多核，尤其是片面提高在一个 CMP 中的 CPU Core 数目几乎没有太大意义，除非你所针对的应用是风花雪月的科学计算。在越来越多的应用领域中，在一个 CMP 中提供的多个处理器内核很难全部发挥作用，造成了不容忽视的资源浪费。这与很多因素相关，我们不要贸然尝试如何去解决这些问题，需要了解这些问题因何而来。

4.1 Cache 层次结构的引入

我经常尝试一些方法，试图去解决在存储器子系统存在的瓶颈问题，总会陷入长考。在我们所处的这个领域，这个时代，在不断涌现出一些新的变化。这些变化会我们之前历千辛万苦学得一些知识和理念荡然无存。这些变化不会依赖你的喜好而改变。

1975 年，Moor 将其定律修正为“晶体管的数目每 18 月增加一倍，性能也将提高一倍”。这个定律在维持了相当长一段时间的正确性之后逐步失效。首先体现在性能提高一倍上，使用过多核编程的人不敢轻易地说用 3 个处理器一定能到达单处理器乘 2 的效果，不用说 2 个核。更多的领域和应用证明并正在不断证明“18 个月性能提高一倍”不断着发生错误。

晶体管的数目每 18 个月增加 1 倍也遭到了前所未有的挑战。Intel 的 Tick-Tick 计划努力维护着摩尔定律最后的领地，也无法改变最后结果。从已知的几乎全部领域看，技术的发展是初期爆发而后逐步平缓，从更长的时间段上分析将呈对数增长。摩尔定律不会例外只会在一段时期适用。这段时光将是属于 Intel 和半导体界最美好的岁月。

在不远的将来，相同的 Die 所容纳的晶体管数目将是有限的。在时光飞舞中，这个将来距离我们这个年代很近，因为永远尚不足够遥远。摩尔定律的失效近在咫尺，这个失效率先对目前云集在处理器领域的人群产生影响。

传统意义的多核时代已经逐步结束。在半导体技术所依赖的基础领域发生重大的更新与变革之前，使用更多的晶体管资源扩大处理器数量并不可取。在设计多核处理器公司中工作的一些朋友经常和我提起，今年或者明年他们就可以集成更多的处理器内核向 Intel 发起挑战。我想和他们说以相同的 Die，Intel 可以集成更多那样的处理器内核，集成 256，512 或者更多的处理器内核对 Intel 都不是太大的难题。只是不知这样做的目的何在，很多应用甚至无法发挥 4 个或者 8 个内核的性能。

即便不谈这些性能问题，我们讨论功耗问题。有很多许多工程师在这个领域中孜孜以求，默默耕耘着，针对功耗进行各类优化。很多工程师在做这件事情时，忽略了一些在其他领域中的基本常识，甚至是能量守恒定律。

忽略这些常识的结果是所做努力最后灰飞烟灭。在处理器领域，完成一件事情可能由多个任务组成，各类降低功耗手段的出发点是在完成这件事情时，统筹考虑多个任务的执行过程，避免重复性的工作。

从这个角度上看，微架构在 Misprediction 时，产生的重复执行操作是最大的功耗来源，还有在多个任务执行过程中，与当前任务执行无关的功能部件的开启等，这些都是具体的优化细节。围绕着处理器功耗优化，有许多需要作出努力的工作细节，需要更多的人来参与。只是再多的人也注定不够。截止到今天，处理器设计的出发点还是为了能够解决更多的应用，希望拥有更多 Feature 来怀抱天下。

这种设计思路从处理器诞生的第一天开始，那个年代距今已经较为遥远。处理器及其相关应用领域的发展与革新是二战之后人类文明进步重要的源推动力。其他领域在可望不可及过程中产生的无限向往，使得几乎剩余的所有应用迅速向这一领域靠拢。也因此产生了通用处理器，使处理器更加聪明，更加通用。

处理器系统正是这样逐步通用化。Intel 是产生这种通用处理器的最典型和最庞大的厂商。而目前发生和正在发生的诸多事实表明这种通用化已经举步维艰，诸如 Memory Wall，功耗居高不下这样的问题已经触及通用处理器是否能够继续发展的底线。

越来越多的领域提出了应用为王这样的口号，本质上这是一个使处理器不再继续通用的口号。更多的领域需要属于自己的定制，不再完全继续依托通用处理器。在这些领域，通用处理器仅作为一个基本组成模块，更多的是适合这个领域的定制逻辑。

处理器领域在经历了爆发式增长，快速增长后回归自然，使得处理器不再神秘，使得一些电子类产品的使用习惯正在进一步回归。苹果近期辉煌源自于此。Jobs 再次来到苹果后只做了一件事情，使 MP3 更像 MP3，电话更像电话，平板更像平板。苹果的成功在于 Jobs 使一个没有受到这些来自处理器领域的专业或者半专业人的影响之下，正常的使用这些设备。这不是创新，是回归自然。任何人，任何群体，任何公司都无法阻挡这股天然的力量。

在此之前，任何一款电子类产品都自觉或者不自觉的受到了来自处理器领域的影响。许多设备，甚至是许多嵌入式设备亦无法幸免。原本就应该属于应用领域的设计并没有按照自身领域的特点设计，被来自通用处理器领域的想法左右。在过去的岁月中，那片领域曾经多次证明引入通用处理器观点后获得了巨大的成功。这种成功在某种程度上桎梏了观念，使更多的人容易忘记在应用领域中所真正应该的关注。

这些固有的观念无法阻挡专用化时代的脚步。专用化和定制化已经出现在通用处理器领域中，比如在加密算法实现领域使用的专用引擎，用于图像处理的 GPU 等等。近期会出现更多的加速引擎，会进一步弱化通用处理器。可以说这种专用化和定制化时代已经来临，通用处理器所覆盖的范围会越来越小。

从这个角度上说，Intel 之忧不在功耗性能比没有超过 ARM 的 Cortex 系列处理器，而是离通用处理器日益远去的各类应用，继续坚持使用更加通用的处理器，以包含更多的应用会遭遇比 Memory Wall 更加牢不可破的 Wall，迎接几乎必败的格局。如果 ARM 架构不是进一步为应用预留空间，去采用通用方法去左右这些应用，通用处理器所面临的困境将如期而至。

在通用处理器领域中，ARM 架构的成功是一个莫大的讽刺。在其成功背后，最主要的原因是在使用 ARM 的许多应用中，ARM 微架构不是如想象中那样重要。我们可以简单列举出在使用 ARM 构架中最伟大的几类电子产品，其中哪怕有一个是因为单纯使用 ARM 架构而伟大。ARM 公司的高明之处是轻架构而重应用，由环绕其总线组成的 Ecosystem 势不可挡，率先开启了专用化时代，ARM 微架构不过是一个载体。

进一步定制化不意味着通用处理器将很快退出历史舞台。对于某类应用，定制辅以通用处理器的组合将长期存在，这种场景之下，定制的与应用紧密结合的部分成为主角。较为理想的情况是 95%的定制与 5%的通用处理器。这有助于解决存储器 Wall 和功耗问题。在这种情况下，处理器将不会频繁访问主存储器，即使访问也是在 5%的基础上，也不用担心功耗问题，因为只有 5%的权重。

这样的结果是不是将全部负担转移到了定制化部分。不用为此做太多担心，我反对使用通用处理器所做的某些应用，是基于处理器过于通用而产生了过多的不需要的存储器访问，引入了并不需要的功耗。贴近应用的定制无法做到不进行数据传递，也无法避免不消耗能量，但是能够将这种影响限制到最小的范围之内。附着在处理器系统上的操作系统，也会因为这些变化而改变。从通用性的角度上看，x86 处理器和 Windows 操作属于一类产品，只是一个在处理器领域，一个在操作系统领域。这两个产品处于同一个困境中。

在不久的将来也许会出现适用于通用部分的操作系统，这个操作系统与传统操作系统，如 Windows 和 Linux，这些操作系统所做的工作较为类似，却在不断弱化，Linux 进入到 2.6 内核之后，再无实质变化，这是通用操作系统所面临的同一个问题。另一个部分操作系统将与定制相关，也被称为应用类操作系统，这些操作系统将有很多种类。

有人会问，既然你认为定制化的时代已经开启，为什么花如此气力在通用处理器的 Cache 中。因为他山之石可以攻玉。通用处理器的发展借鉴了很多领域的精华，在定制即将兴起的时代，不能忽略通用处理器的核心。定制化领域依然会从狭义广义 Cache 的设计思想中受益。Cache 的核心是数据缓冲组织结构，通路连接方法，管理策略和各类算法，这些内容较为基础。世间万物，千变万变，基础内容的变化较为缓慢。

使用定制会缓解通用处理器做遇到的各类瓶颈，而不是消除。主存储器的容量和延时的不断增加依然是客观存在的事实。通用处理器领域的解决方法是引入更多级别的 Cache。多核处理器的进一步发展使得多级 Cache 间的组成结构异常复杂。

在单处理器系统中，Cache Memory 由多个层次组成。这些层次相互关联，相互依托，而为参天大树。底部由主存储器与外部存储器组成庞大的根系，其上由各类 Cache 和各类用于连接的 Buffer，形成茂密的枝干。这些参天大树更通过各种类型的网络，或松或紧连接在一起，若小为林，夫广为森，煌煌立于天地，善能用之，攻坚强者莫之能胜。在一个 ccNUMA 处理器系统中，典型的 Cache Memory 层次结构如图 4-1 所示。

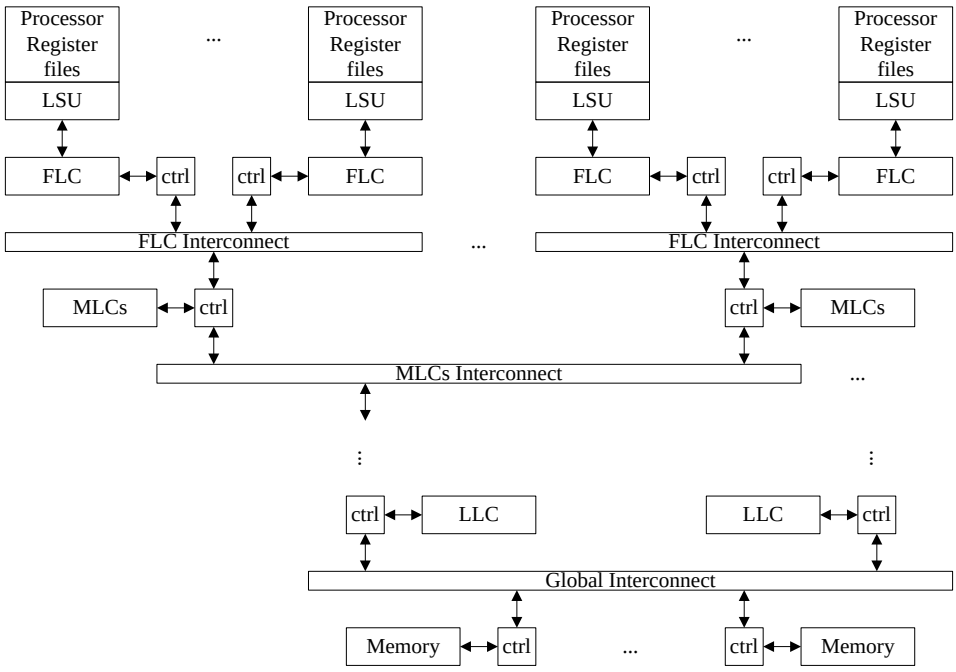

图 4-1 Cache Memory 的层次结构[66]

不断增加的主存储器容量需要更大的 Cache 减少 Miss Rate，使用一个很大的 L1 Cache 无法胜任这个工作，主存储器不仅在变大，访问延时也在逐步提高，进一步加大了指令流水线间的差距。单独一级 Cache 无法掩盖这个差距。提高 Cache 的访问延时与进一步扩大容量是一个 Trade-Off。而且 Cache 容量越大，其访问延时也越大。

这些现状使得在现代处理器系统中引入多级 Cache 成为必然。在没有从根本上解决存储器子系统存在的这些问题时，Cache 级数会逐步提高，目前大多数中高端处理器都已经包含了 L3 cache，大规模地使用 L4 Cache 并非遥不可及。

在 Cache Memory 层次结构的最顶端为微架构中包含的系统寄存器和各类 Buffer，如与 Data Cache 连接的 LSQ 和与 Instruction Cache 连接的 I-Cache Line Filling 部件中的 Buffer。FLC(First-Level Cache)指与微架构直接进行连接的 Cache。在许多微架构中，FLC 指 L1 Cache，而在有些微架构中，在 L1 Cache 之上，存在一级 L0 Cache，此时这个 L0 Cache 即为 FLC。

LLC 指在处理器系统中与主存储器直接相连的 Cache。在不同的处理器系统中，LLC 所指对象迥异，有的处理器使用 L2 Cache 连接主存储器，此时 L2 Cache 为 LLC；有的使用 L3 Cache，此时 L3 Cache 为 LLC。有些处理器系统中，使用 L1 Cache 连接其上的微架构，同时连接其下的主存储器，此时 L1 Cache 为 FLC 的同时也为 LLC。

MLC 其上与 FLC 相连，其下与 LLC 相连。在一个复杂的处理器系统中，Cache Memory 层次结构由 L0~L3 Cache 和 EDRAM 组成，其中 L0 Cache 与微架构直接相连，而 EDRAM 与主存储器相连。此时 L0 Cache 为 FLC，EDRAM 为 LLC，而 L1~L3 Cache 为 MLCs。

在处理器系统中，不同级别的 Cache 所使用的设计原则并不类同。每一级 Cache 都有各自的主要任务，并不是简单的容量与延时的匹配关系。假设一个处理器系统中含有 L1 和 L2 两级 Cache。多数程序具有的 Temporal 和 Spatial Locality 使得 L1 Cache 的 Hit Rate 非常高，即便 L1 Cache 只有 4KB 或者 32KB。这使得在 L2 Cache 层面并不会看到过多的 Miss[70]，这使得 L2 Cache 层面的优化重点并不是匹配容量与延时，而且 L1 Cache 的 Miss Rate 越低，L2 Cache 的延时越无关紧要[70]。

这使得在绝大多数微架构中，对于 L1 Cache 的优化专注于提高 Hit Time，在现代处理器中其范围在 3~5 个 Cycle 之间，为了避免在这个 Critical Path 上虚实地址转换的开销，很多微架构，如 Cortex A8/9 和 Opteron，直接使用了 Virtual Cache。在 L2 Cache 层面，其 Cycle 与容量比大于 L1 Cache 的对应比值。这些设计方法使得 Cache Hierarchy 的设计在与直接使用一级 Cache 的整体 Miss Ratio 等效时，提高了 Hit Time 这个重要的参数。

在不同的微架构中，L1 与 L2 Cache 的关系可以是 Inclusive 也可以是 Exclusive 或者其他类型，如果是 Exclusive 关系，计算 L2 Cache 的有效容量需要累加 L1 Cache 的容量。在 L2 Cache 层面中，具有更多的时间，可以实现更多的 Way Associatively 进一步降低 Miss Ratio。

在现代处理器中，这个 L1 和 L2 Cache 通常是私有的。LLC 的实现需要考虑多核共享的问题，除了继续关注 Hit Time，Miss Ratio 和 Miss Penalty 这些基本参数之外，需要重点考虑的是 Scalability，由多核处理器引发的 Bus Traffic 等一系列问题。

在一个大型处理器系统中，不同种类的 Cache 需要通过 Cache Coherent Networking 进行连接，组成一个复杂的 ccNUMA 处理器系统。这个 Cache Coherent Networking 可以采用多种拓扑结构，可以使用 MESH，n-Cube，也可以是简单的树状结构，环形结构。在整个 ccNUMA 处理器系统中，这个 Cache Coherent Networking 的搭建异常复杂。

比这个 Networking 更难实现的是 Cache 间的一致性协议，这个一致性策略可以使用纯硬件，也可以使用软件 Message 方式实现。Cache 每多引入一级，这个一致性协议愈难实现愈难验证。其间的耦合关系设计复杂度已经超出许多人的想象。

Cache Hierarchy 还与主存储器间有强的耦合关系，延时与带宽是两者间永恒的话题。在这个话题之后的功耗亦值得密切关注。在主存储器之下是外部存储器系统，包括 SSD(Solid State Disk)，磁盘阵列和其他低速存储器介质。外部存储器系统的复杂程度不但没有因为相对较低的速度而减弱，而且这种低速使得设计者有着更大的回旋余地，可以设计出远超过 Cache 层次结构使用的的复杂算法。如上文提及的 LIRS 和 Clock-Pro 页面替换算法。在这个领域没有什么算法是万能的，所有算法都有自身的适用范围。

在讲述这些复杂的设计之前，我们需要从更加基本的内容开始，需要回顾存储器读写指令的执行过程，在微架构中使用的 LSU 部件是 Cache Hierarchy 设计的最顶端。在多数微架构中，LSU，AGU 和 ALU 协调工作实现存储器读写指令的发射与执行。

4.2 存储器读写指令的发射与执行

在 CPU Core 中，一条存储器指令首先经过取值，译码，Dispatch 等一系列操作后，率先到达 LSU(Load/Store Unit)部件。LSU 部件可以理解为存储器子系统的最高层，在该部件中包含 Load Queue 与 Store Queue。其中 Load Queue 与 Store Queue 之间有着强烈的耦合关系，因此许多处理器系统将其合称为 LSQ。在多数处理器的存储器子系统中，LSU 是最顶层，也是指令流水线与 Cache 流水线的联系点。

LSU 部件是指令流水线的一个执行部件，其上接收来自 CPU 的存储器指令，其下连接着存储器子系统。其主要功能是将来自 CPU 的存储器请求发送到存储器子系统，并处理其下存储器子系统的应答数据和消息。在许多微架构中，引入了 AGU 计算有效地址以加速存储器指令的执行，使用 ALU 的部分流水处理 LSU 中的数据。而从逻辑功能上看，AGU 和 ALU 所做的工作依然属于 LSU。

在一个现代处理器系统中，LSU 部件实现的功能较为类同。LSU 部件首先要根据处理器系统要求的 Memory Consistency 模型确定 Ordering，如果前一条尚未执行完毕存储器指令与当前指令间存在 Ordering 的依赖关系，当前指令不得被 Schedule，此时将 Stall 指令流水线，从来带来较为严重的系统惩罚；LSU 需要处理存储器指令间的依赖关系，如对同一个物理地址的多次读写访问，并针对这种 Race Condition 进行特殊优化；LSU 需要准备 Cache Hierarchy 使用的地址，包括有效地址的计算和虚实地址转换，最终将地址按照 L1 Cache 的要求将其分别送入到 Tag 和状态阵列。

L1 Cache 层面需要区分是存储器读和存储器写指令。无论是存储器读还是写指令穿越 LSU 部件的过程均较为复杂。为缩短篇幅，下文重点关注存储器读指令的执行。存储器读操作获得物理地址后将进行 Cache Block 的状态检查，是 Miss 还是 Hit，如果 Cache Hit，则进行数据访问，在获得所需数据后更新在 LSU 中的状态信息。

如果在 L1 Cache 中 Miss，情况略微复杂一些。在现代处理器系统中，Non-Blocking Cache 基本上是一个常识般的需求，为此首先需要在 MSHR 中分配空余 Entry，之后通过 L1 Cache Controller 向其下 Memory Hierarchy 发送 Probe 请求。

在 L1 Cache Controller 中，大多使用 Split Transaction 发送这个 Probe 请求，之后经过一系列复杂的操作，这个操作涉及多核之间的 Cache 一致性，不同的一致性协议对此的处理不尽相同。在获取最终数据之后，回送 Reply 消息。LSU 在收到这个 Reply 消息后将数据传递给指令流水线，并释放 MSHR 中的对应 Entry。

以上这些操作可以并行执行，如使用 Virtual Cache 方式可以直接使用虚拟地址，而无需进行虚实地址转换，可以将数据访问与 Tag 译码部件重叠，更有一系列预测机制进一步缩短数据 Cache 访问的延时。

存储器写的过程较存储器读复杂一些，在 L1 Cache Hit 时，会因为状态位的迁移而带来一系列的 Bus Traffic；如果在 L1 Cache Miss，要首先取得对访问数据的 Ownership 或者 Exclusive。获得 Ownership 的步骤与存储器读发生 Miss 时类似，但是只有在存储器写 Commitment 时，才能将数据真正写入到 Cache 中。

在微架构的设计中，缩短 Cache 的访问延时与提高带宽是 Cache 流水设计的一个永恒话题。这些话题可以空对空的讨论，搭建各类模型得出最后的量化分析报告。但是这些报告面对着 ccNUMA 处理器多级 Cache 设计的复杂度时，显得如此苍白无力。

下文以 AMD 的 Opteron 微架构为主，进一步说明存储器读写的执行过程。我们从 Cache Hit 时的 Load-Use Latency 这个重要参数的分析开始，进一步介绍 Cache Hierarchy 性能的评价标准，最后详细说明 Opteron 微架构的 LSU 部件的工作原理。

在多数微架构的数据手册中，Load-Use Latency 的计算是从指令进入 LSU 之后开始计算，以指令从 Cache Hierarchy 中获得最终数据作为结束。在不同的场景，如其下的 Cache Hit 或者 Miss 时，所得到的 Load-Use Latency 参数并不相同。

在 Cache Hit 时，AMD Opteron 微架构凭借着 Virtual Cache，使得其 L1 Cache 的 Load-Use Latency 仅为 3 个 Cycle[6]，这是一个非常快的实现。在 Intel 近期发布的 Sandy Bridge 中，L1 Cache 的 Load-Use Latency 也仅为 4 个 Cycle[69]。这个参数不能完全反映在一个处理器系统中 Cache Hierarchy 的整体 Hit Time 参数和存储器平均访问时间，远不能仅凭这一个参数得出 Opteron 的 Cache Hierarchy 结构优于 Sandy Bridge 微架构的结论。

从许多 Benchmark 的指标上看，Sandy Bridge 微架构在 Cache Hierarchy 中的表现远胜于 Opteron，Virginia STREAM 的测试结果表明 Opteron 在 Copy，Scale，And 和 Traid 测试指标上优于 Pentium4，却落后于使用 Nehalem Xeon 的 Apple Mac Pro 2009，如表 4-1 所示。

表 4-1 STREAM "standard" results[71]

Data	Machine ID	ncpus	COPY	SCALE	ADD	TRAID
2000.12.23	Generic_Pentium4-1400	1	1437.2	1431.6	1587.7	1575.4
2005.02.04	AMD_Opteron_848	1	4456.0	4503.6	4326.3	4401.4
2009.10.23	Apple_Mac_Pro_2009	1	8427.6	8054.4	8817.4	8953.4

Copy，Scale，And 和 Traid 是 Virginia STREAM 定义的 4 个操作。其中 Copy 操作与 a(i) = b(i) 等效；Scale 与 a(i) = q×b(i)等效；Add 与 a(i) = b(i) + c(i)等效；而 Traid 与 a(i) = b(i) + q×c(i)等效。由以上操作可以发现，STREAM 的测试结果不仅和 Cache 层次结构相关，而且与主存储器带宽和浮点运算能力相关。这一测试指标并不能作为 Opteron 和 Nehalem 微架构 Cache 层次结构孰优孰劣的依据。与存储器测试相关的另一个工具是 lmbench，不再细述。

这些 Benchmark 程序在相当程度上反映了 Cache 和存储器子系统的性能。但是在一个实际项目的考核中依然只能作为参考。对于这个设计，最好的指标一定从自身的应用程序中得出的。很多人会对这个说法质疑，因为许多应用程序的书写异常糟糕，没有用到很多的优化手段。但是面对着由数千万行组成，积重难返的应用程序，更多的人剩下的只有无奈。

在 Intel 的 Nehalem 和 Sandy Bridge 微架构中，FLC 访问延时均为 4 个 Cycle[12][69]，一个可能的原因是，Intel x86 处理器在 Pentium IV 后并没有采用 VIPT(Virtually Indexed and Physically Tagged)方式[12][68]组织 L1 Cache，而采用了 PIPT(Physically Indexed and Physically Tagged)方式[①]。Opteron 微架构使用 VIPT 方式，访问 L1 Cache 时可以直接使用 Virtual Address，从而可以节省一个时钟周期。但是 Opteron 的总线位数只有为 64 位。AMD 最新的 Bulldozer 微架构，将总线位数提高为 128b 后，Load-Use Latency 提高为 4。

确定存储器平均访问时间除了需要考虑 Hit Time 参数之外，还需要考虑 Miss Penalty 参数，Virtual Cache 并不能解决所有问题。当 L1 Cache Miss 时，Opteron 微架构访问 PIPT 结构的 L2 Cache 时，依然需要使用 TLB，TLB 译码过程最终不会省略。

采用 Virtual Cache 还会带来 Cache Synonym/Alias 问题，需要设置 RTB(Reverse Translation Buffer)解决这一问题。使用这种方法除了要使用额外的硬件资源，而且在多核系统中还引入了 TLB Consistency 的问题。在一个微架构的设计中，Virtual Cache 和 Physical Cache 间的选择依然是一个 Trade-Off。

在抛开 Virtual Cache 的讨论后，我们专注讨论 LSU 部件。在现代处理器系统中，为了提高存储器读写指令的执行效率，一个微架构通常设置多个 LSU 部件，这些 LSU 不是简单的对等关系，而是有相互的依赖关系。

[①] 尚无可靠详细资料说明 Sandy Bridge 的 L1 Cache 是 VIPT 还是 PIPT。

简单对等相当于在一个微架构中设置了多个功能一致，异步执行的 LSU 部件。为了保证处理器系统 Memory Consistency Model 的正常运行，这些异步操作需要在运行的某个阶段进行同步处理，从而带来了较大的系统复杂度。这不意味着不采用这种对等设计可以避免这些同步问题，只是相对容易处理。

在多数微架构中，多个 LSU 部件间通常是各司其职，并非完全对等。Opteron 微架构的 LSU 部件也是采用了非对等的两个 LSU 部件。在介绍这些内容之前，我们需要了解 Opteron 微架构的基本组成结构和流水线示意，如图 4-2 所示。

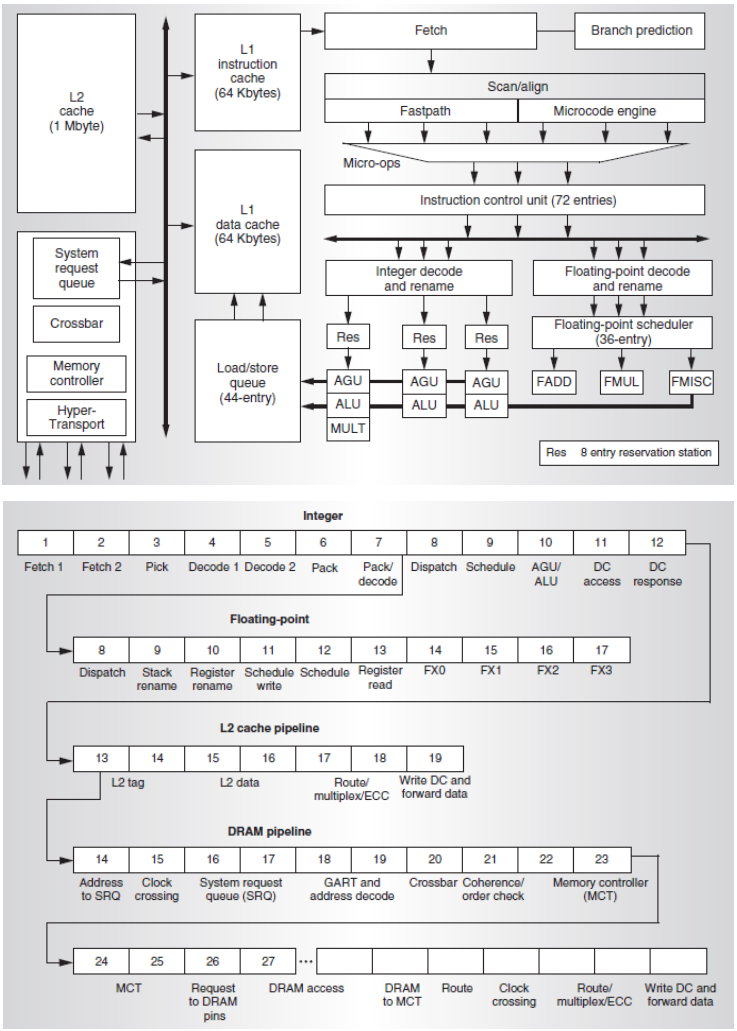

图 4-2 Opteron 微架构的基本组成结构和指令流水线[72]

在 Opteron 微架构中，存储器指令通过 Fetch，decode，Dispatch 阶段，并在所有 Operands 准备就绪后，经由 Scheduler launch 到对应的执行部件。上图中的 Res 部件全称为 Reservation Station，这是 Scheduler 的一个重要组成部件。其中存储器读写指令需要使用 Integer Scheduler，在这个 Scheduler 中包含一个 AGU 和一个 ALU。

Opteron 微架构与存储器指令执行相关的部件由 Integer Scheduler，Load/Store Queue，L1 Cache 和其下的存储器子系统组成。Load/Store Queue 即为上文多次提及的 LSQ，是 Opteron LSU 部件的重要组成部分。在 LSU 部件中除了具有 LSQ 之外，还有许多异常复杂的控制逻辑和与指令流水线进行数据交换的数据通路，是一个微架构的存储器子系统设计的起始点，是存储器指令的苦难发源地。

在 x86 处理器系统中，存储器指令大体可以分为 F(reg, reg)[①]，F(reg, mem)和 F(mem, reg)三大类。在这些指令中，第 1 个 Operand 为目标操作数也可以为源操作数，第 2 个 Operand 只能为源操作数。与 LSU 部件直接相关的指令为 F(reg, mem)和 F(mem, reg)。后一类指令的处理相对较为复杂，该类指令需要首先进行存储器读，进行某种运算后，再次进行存储器写，即 Read-Modify-Write μop。在 Opteron 微架构中，浮点和 SSE 指令的存储器读写依然需要通过 Integer 指令流水，为简化起见，下文不再讨论浮点和 SSE 指令的存储器读写指令。

一条存储器指令的执行需要使用 AGU，ALU 和 LSU 三大部件。AGU，ALU 将和 LSU 部件协调流水作业，最终完成一次存储器读写操作。L1 Cache 的 Load-Use Latency 参数的计算是从存储器指令进入 LSQ 开始计算。为了实现指令流水的并发高速运转，在 AGU 和 ALU 中也具备多级流水结构。存储器指令在通过 ALU，AGU 与 LSU 部件的过程中存在相互依赖关系，如图 4-3 所示。

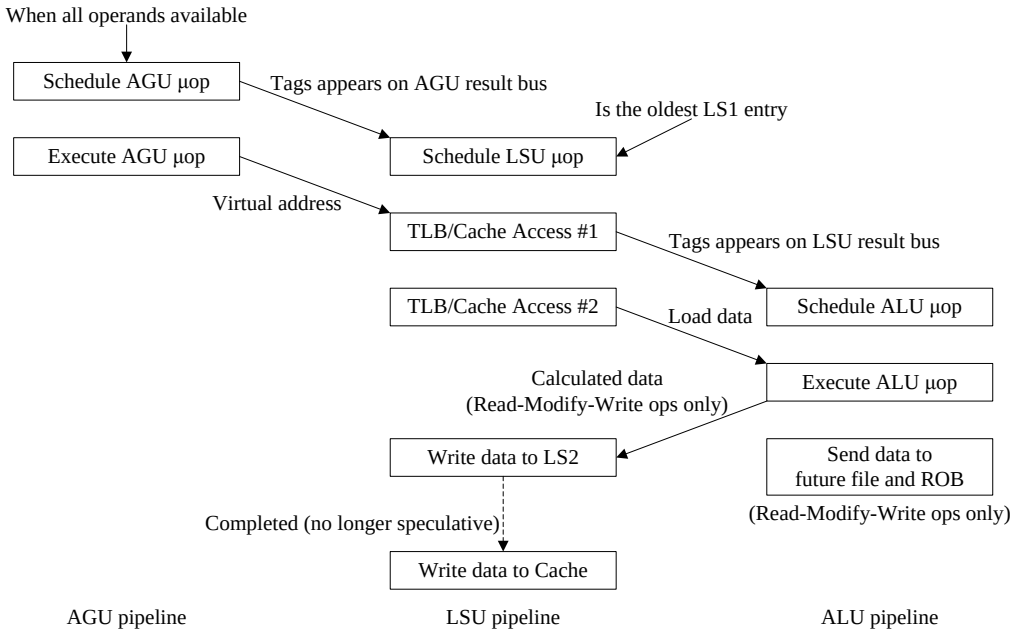

图 4-3 AGU/ALU 与 LSU 流水线间的依赖关系

我们分别讨论 F(reg, mem)指令和 F(mem, reg)指令的执行过程。在 RS(Reservation Station)等待的 F(reg, mem)指令，当所需的 Operands Available 后首先 Launch 到 AGU 部件，并在地址计算完毕后，将结果发送到 LSU 部件。但是存储器访问 μop 是同时进入到 AGU 和 LSU 部分中的，只有 LSU 需要等待 AGU 的计算结果。

在 LSU 部件从存储器子系统中获得最终数据之后，将其送给 ALU，并最终完成指令的执行。F(mem, reg)指令的执行过程与此相近，只是需要在 LSQ 中经历两次等待过程，第 1 次是等待 ALU 计算的结果，第 2 次等待 Store μop 的 commit 将结果最终写入 Cache。F(mem, reg)进行存储器读时的操作与 F(reg, mem)一致。

AGU 和 ALU Pipeline 的执行过程较易理解，本篇对此不做进一步的说明。Opteron LSU 的设计使用 Tag 和 Data 总线分离的方式。Scheduler 在监听 Result Bus 时，发现对应的 Tag 有效时即可启动 LSU 和 ALU 的 μop，不必等待 Data 总线，通常情况下 Tag bus 上的数据先于 Result Bus 一个 cycle，以 Overlap 不同的流水阶段。在 Opteron 微架构中，1 个 Cycle，可以 Launch 一个 ALU 和一个 AGU 微操作。如图 4-3 所示，这两个操作显然没有对应关系。

[①] 许多论文书籍认为 F(reg, reg)指令不属于存储器指令。

LSU 部件负责与其下的存储器子系统进行数据交互，也是图 4-3 所示三大部件中最为复杂的一个部件。在 Opteron 微架构中，LSU 由两个部件组成，分别是 LS1 和 LS2。其中 LS1 中含有 12 个 Entry，LS2 中含有 32 个 Entry，共 44 个 Entry[6]。

LS1 和 LS2 也被分别称为 Pre-Cache 和 Post-Cache Load/Store Unit，绝大多人会认为 LS1 用于访问 L1 Cache，而 LS2 用于访问 L2 Cache。这一说法源自 AMD 的软件优化手册[73]，这似乎是一个权威说法，也在某种程度上善意地误导着读者。这一说法非但并不准确，在某种程度上甚至是错误的。Opteron LSU 的结构示意如图 4-4 所示。

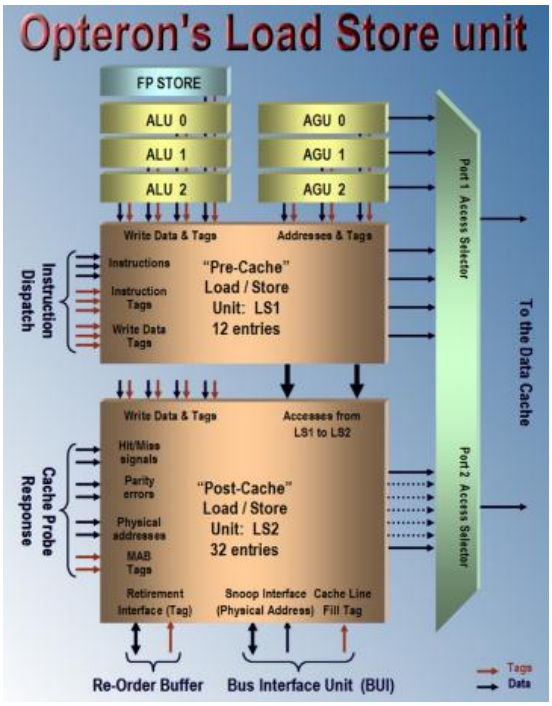

图 4-4 Opteron 的 LSU 的示意图[6]

在 Opteron 微架构中，一条存储器读写指令在被 Dispatch 到 Integer Scheduler 的同时，也会被 Dispatch 到 LS1 的相应 Entry 中等待。LS1 通过监听 AGU 的 Tag 总线获得必要的信息后，开始真正意义上的执行。这些必要的信息包括，LS1 所需要的地址是否能够在下一拍有效，将使用哪一个 AGU 产生的地址。LS1 使用 Tag 总线的信息虽然不能用于 L1 Cache 的 Probe 操作，却可以实现 AGU 和 LSU 的流水执行。

Opteron 微架构分离 Tag 和 Data 总线是利用流水进行加速的技巧，将 AGU 的执行分解为两个步骤，以最大化 AGU 与 LSU 流水部件间的 Overlap。除了 AGU 部件分离的 Tag 和 Data 总线，LSU 和 ALU 也进行了这样的总线分离。

在 LS1 中的指令在下一拍从 Data 总线中获得地址后执行，由上文所述，Opteron 微架构的 L1 Cache 具有两个端口，当所访问的数据间不发生冲突的前提下，LS1 率先处理最先进入队列的两条存储器读写指令，并将其转交给 L1 Cache Controller 做进一步处理。

无论是存储器读还是存储器写操作，L1 Cache Controller 都需要首先进行 Probe 操作。读操作发起的 Cache Read Probe 操作将带回数据。写操作发起的 Cache Probe 操作，也被称之为 Probe-before-Write。进行这个 Probe 操作之前，需要准备好待写的 Cache Block，Probe 操作返回时将会带回数据，而当且仅当存储器写指令获得最终数据而且进行 Commit 操作之后，才能将数据真正写入。由于写入的数据在多数情况不能占满一个完整的 Cache Block，此时需要和 Probe 操作带回的数据进行 Merge 后进行写入操作。

写操作的实现细节比上文描述的过程复杂得多。即便我们抛离了与 Store Ordering 相关的诸多和 Memory Consistency 的概念和各类 Cache Write 策略，没有考虑在 Cache Write Hit 时使用的 Write-Through，Write-Back 和 Write-Once 策略，没有考虑 Cache Write Miss 时使用的 Write-Allocate, Write-Validate 和 Write-Around 策略。

我们首先考虑在进行 Write Probe 操作之前，在 Cache 流水线准备好一个未使用的 Cache Block，到从存储器子系统获得数据后真正写入数据的这段延时。我们首先关注这个刚刚准备好的 Cache Block 在此时使用的状态信息，似乎 MOESIF 这几个状态都无法准确描述此时的这条 Cache Block 所处的状态。

MOESIF 这些状态位仅是 Cache Controller 所使用的简单得不能再简单的状态位，仅是一个 Stable 状态，是 Cache Block 诸多状态中一个子集。一个 Cache Block 中还含有其他状态位，更为复杂的处理 Race Condition 的 Transient 状态位。在 Cache 与 Cache 间的总线中包含着诸多 Coherence Message 和各类 Bus Transaction。如果进一步考虑多级 Cache 的组成结构后，Cache Controller FSM 使用的状态位和流程迁移的复杂程度令人叹为观止。

其次我们需要考虑存储器写操作在等待最终 Commitment 时所带来的延时。如上文所述，如果存储器读指令的访问结构在 LSQ 中命中时，微架构将不会读取 L1 或者更高层次的 Cache，从而充分利用了这个延时。这个延时可以带来的另外一个好处，由于后续的存储器读指令可能需要读取相同的 Cache Block，此时也要进行 Probe Cache 操作。在这个延时中，这两个 Probe 操作可以合并，从而在一定程度上降低了总线的 Traffic。这些优化手段提高了存储器读写指令的效率，也带来了较大的系统复杂度。

在 Opteron LS1 中的存储器读或者写操作，如果没有及时完成，例如存储器写指令没有及时地进行 Commitment，将导致存储器写操作无法在 LS1 中完成，此外如果存储器读没有在 L1 Cache 中命中也无法及时完成时，这些在 LS1 中的指令都将进入 LS2，从而为后续的存储器指令预留空间。这些预留可以为更多可能在 L1 Cache Hit 的存储器操作并发执行。从这个角度分析可以发现 LS1 主要用于 Cache Hit 的处理。

在 LS2 中的所有存储器读写指令，包括 Load，Store 和 Load-Store 指令将继续监听 Cache Probe 的结果，当发生 Cache Miss 时，需要将存储器读写请求转发给 BUI(Bus Interface Unit) 部件，BUI 部件将根据需要从 L2 Cache 或者主存储器中获得最终数据。即 LS2 用于处理 Cache Miss 时，需要较长时间的存储器读写指令。

在 Opteron 微架构中，采用了 Non-Blocking Cache 的实现方式，为每一条 Miss 存储器请求添加了一个 MAB Tag(Missed Address Buffer Tag)之后再发送给 BUI 部件，同时为这些 Miss 请求在 LS2 中设置了 Fill Tag。当 Probe 操作的数据从 L2 Cache Controller 返回时，其 MAB Tag 将与在 LS2 中的 Fill Tag 进行比较，进一步确认数据是否有效，并将 LS2 中的存储器指令的状态从 Miss 更新为 Hit。

Store 指令将在 LS2 中停留更长的时间，直到该指令从流水线中按序退出后，才能将数据写入到存储器子系统中。当发生 Misprediction 或者 Exception，微架构可以丢弃在 LS2 中的暂存的 Store 指令。只要 Store 指令没有离开 LSU，都可以丢弃，当然这种丢弃是条件的，并不是随意丢弃。

在 Opteron 微架构中，L1 和 L2 Cache Controller 将与 LSU 共同完成一次存储器读写操作。在 L1 和 L2 Controller 中含有各类 Buffer，和连接这些 Buffer 的通路。一次存储器访问指令，在通过指令流水后，将首先到达 LSU，并由 LSU 将其请求转发至 L1 和 L2 Cache Controller，并由这些 Cache Controller 完成剩余的工作。

Cache Controller 需要在保证 Memory Consistency 的前提下，将数据重新传递给 LSU，完成一次存储器读写的全过程。在一个微架构中，Cache Controller 是一个较为复杂的功能，由其管理的 Cache 流水线是整个微架构的精华。

4.3 Cache Controller 的基本组成部件

在一个处理器系统中，存储器子系统是一个被动部件，由来自处理器的存储器读写指令和外部设备发起的 DMA 操作触发。虽然在存储器子系统中并无易事，DMA 操作依然相对较易处理。在多数设计中，一个设备的 DMA 操作最先看到的是 LLC，之后在于其他 Cache 进行一致性操作。通常处理器系统的 LLC 控制器将首先处理这些 DMA 操作。

随着通用处理器集成了更多的智能外部设备，这些智能设备已经直接参与到处理器系统的 Cache Coherency 中，这些设备可以直接访问原来属于处理器系统的存储器子系统。最典型的设备就是 Graphic Controller。从 Sandy Bridge 微架构开始，x86 处理器内部集成了 Graphic Controller，Graphic Controller 可以与处理器共享存储器子系统。这使得 Nvidia 无法继续参与这个内部集成，以获得许多显而易见的优点时，义无反顾地投奔 ARM 阵营。

本篇不再关注这些与外部设备相关的 Cache 一致性，也不再讲述与 DMA 操作相关的细节，重点关注来自指令流水线的存储器指令的执行，以及在这些存储器指令的执行过程中，通过 Cache Controller 时所进行的对应操作。

从实现的角度上看，任何一个复杂的处理器系统都是由数据缓冲，连接这些数据缓冲的通路和控制逻辑组成。其中与 Cache Controller 相关的通路，数据缓冲和控制逻辑是最重要的组成部件，也是处理器系统的数据通路的主体。

这些 Cache Controller 及其数据缓冲通过横向和纵向连接，组成一个基本的 CMP 系统，而后这些 CMP 通过各类拓扑结构进一步连接，形成复杂的 ccNUMA 处理器系统，如图 4-1 所示。本篇所重点关注的是，在一个 CMP 处理器系统之内的 Cache Controller 组成与结构，并简单介绍 ccNUMA 处理器系统的互连方式。

在一个 ccNUMA 处理器系统中，Cache Controller 由 FLC/MLCs/LLC Cache Controller，DMA Controller 和 Directory Controller，共同组成。其中 FLC Cache Controller 与 LSU 和指令流水线直接相连，管理第 1 级分离的指令与数据 Cache；MLCs Cache Controller 可能由多级 MLC Controller 组成，上接 FLC 下接 LLC Controller，管理中间层次的 Cache。在多数 CMP 处理器中，FLC 与 MLCs Controller 在一个 CPU Core 之内，属于私有 Cache，并且与 CMP 处理器中的其他处理器的 FLC/MLCs 保持一致。

LLC Controller 管理 CMP 处理器的最后一级 Cache。在一个 CMP 处理器中，LLC 通常由多个 CPU Core 共享，其组成结构可以是集中共享，也可以分解为多个 Distributed LLC。Directory Controller 是实现 ccNUMA 结构的重要环节，该 Controller 通常设置在 Home Agent/Node 中，并与主存储器控制器直接关联在一起。当一个数据请求在 LLC 中 Miss 后，将到达 Home Agent，之后在进行较为复杂的 CMP 间的 Cache Coherency/Memory Consistency 处理。

我们首先讨论在一个 CMP 系统内，FLC，MLCs，LLC 的连接拓扑结构，即 Cache 通路互连结构。在这个通路设计中，需要重点关注的依然是 Throughput 和 Latency，同时实现处理器系统所约定的 Memory Consistency 模型。

在一个 CPU Core 内，FLC 大多为 Private Cache，当然我们需要忽略 SMT 这类典型的共享 FLC 的拓扑结构。MLCs 可以被多个 CPU Core Share，也可以为 Private。如在 Intel 的 Nehalem 微架构[12]和 Sandy Bridge 微架构[69]中，MLCs 为 Private。最初 Power 系列处理器对多 CPU Core 共享 MLCs 情有独钟，Power4 和 Power5 都采用了这种架构[75][76]，而后出现的 Power6 和 Power7 放弃了 Share MLCs 这种方式，也采用了 Private MLCs 的方式[77][78]。

AMD 最新的 Bulldozer 微架构采用了两个 CPU Core 共享一个 MLC(L2 Cache)的方式，而且出人意料的使用了 NI/NE (Non-inclusive and Non-Exclusive)和 Write-Through 组成结构[74]，与之前微架构 Cache 的组成方式相比，唯一坚持的只有 VIPT。

在有些 CMP 中，LLC 为多个 Core 共享，也有部分 CMP 将 LLC 作为 Victim Cache，如 Power6 和 Power7[77][78]。为增加 LLC 的带宽和 Scalability，一些 CMP 采用了 Distributed LLC 方式。我们首先讲述 CPU Core 与 LLC 之间的连接关系。在一个 CMP 系统中，多个 CPU Core 与 LLC 通常使用 Share Bus，Ring 或者 Crossbar 三种方式进行连接，如图 4-5 所示。

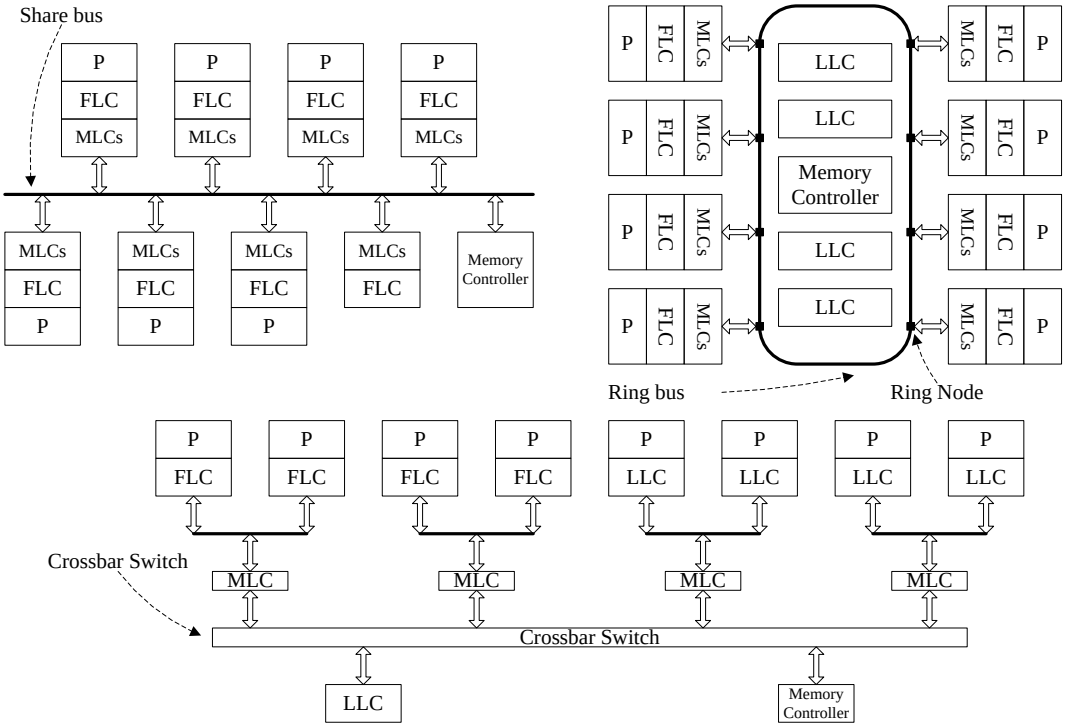

图 4-5 CPU Core 与 Cache Hierarchy 的连接通路

其中 Share Bus 结构最为直观，在处理 Memory Consistency 时也最为便捷，所存在的问题也显而易见，即 Share Bus 所能提供的最大带宽有限，很难挂接更多的 CPU Core。多个 CPU Core 在争用同一条总线时，不仅降低了有效带宽，也进一步加大了 Latency。

目前许多 CMP 处理器使用了 Ring Bus 的组成结构，如 Intel 的 Sandy Bridge[69]，IBM 的 Power4[75]，Power5[76]，Cell 处理器[79]和 XLP832[80]。与 Share Bus 相比，Ring Bus 能够提供的带宽相对较大，处理 Memory Consistency 也相对较为容易。

当使用 Ring Bus 时，每一个 CPU Core 通过 Ring Node 与 Ring Bus 互连，而每一个 Ring Node 与其相邻的两个 Node 采用点到点的连接方式。在一个实现中 Ring Bus 可以是单向的，也可以是双向的。由于 Coherency 的要求，Ring Bus 通常由多个 Sub-Ring 组成，分别处理数据报文与 Coherent Message 报文。其设计难点主要集中在 Ring Bus 的 Ordering 处理和 Ring-Based 的 Token Coherence 模型。在某种程度上说，Share Bus 与 Ring Bus 是一致的，尤其是在 Snoop Message 的处理器中。

与 Share Bus 和 Ring Bus 两种连接方式相比，使用 Crossbar Switch 方式可以提供较大的物理带宽，而且可以获得较小的 Latency，然而在 Memory Consistency 层面需要付出更大的努力。在有些处理器中，联合使用了 Crossbar 和 Ring 结构。如 Power4 使用 Crossbar 连接两个 L1 Cache 和 L2 Cache，而使用 Ring 连接 L2 Cache，L3 Cache，L3 Directory 和存储器控制器[75]。采用 Crossbar Switch 的一个重要的微架构是 UltraSPARC 系列处理器，目前 UltraSPARC T1 和 T2(Niagara 和 Niagara 2)[81]采用了这种结构。在 Niagara 2 处理器中含有 8 个 CPU Core，每一个 Core 中含有 8 个 Thread。

T2 微架构的 Crossbar 采用 Non-Blocking，Pipelined 结构，上接 8 个 L1 Cache，下与 8 个 Bank 的 L2 Cache 相连，可以同时处理 8 个 Load/Store 数据请求和 8 个 Data Return 请求。L1 Cache 采用 Write-Through 方式，L2 Cache 采用 Write-Back，Write-Allocate 方式。使用 Crossbar 方式并不易处理 Cache Coherent， T2 微架构专门设置了 L2 Directory[81]，占用了较多的 Die Size，以至于 T2 包括之后的 SPARC64 VIIIfx(Venus) L3 Cache 的容量较小，这对 BLAS 和其他用于科学计算的实现并没有太大影响。采用 Venus 微架构的 K Computer 集成了 68,544 个 CPU Core，LINPACK 的最终 Benchmark 结果达到了 8.162 petaflops，跃居 TOP500 处理器之首[82]。

这些 CMP 处理器可以进一步通过互连网络，组成更为复杂的 ccNUMA 处理器系统，更为复杂的 Supercomputer 处理器系统，如 K Computer，Tianhe-1A，Jaguar 和 Roadrunner 等。此时的连接通路已在处理器芯片之外，这是 Infiniband，QPI 和 HT 这些连接方式的用武之地。连接上万个 PE(Processor Element)的 Interconnection 令人惊叹。这些 Interconnection 所使用的数据通路，通信协议和状态机组成了一个夺目的奇观。

对 Performance，Performance，Performance 的渴望使得 Supercomputer 的设计无所不用其极。每一个子设计都异常重要，任何一个疏忽都会极大降低一个系统整体的 Stability。而对 Scalability 的无限追求更加增大了系统的设计复杂度。这一切极大降低了 Supercomputer 的 Programmability。

每念及此都会放弃画出使用几个 CMP，组成 2S，4S 和 8S 系统的连接通路图，这张连接通路图甚至没有一个 CMP 内部使用的 Cache Hierarchy 的连接关系复杂。而且这些连接通路仅是 Cache Hierarchy 设计的一部分，在 FLC，MLCs 和 LLC 之间还存在各类的 Buffer。经过多番考量，我决定首先介绍 LSU，FLC 与 MLCs 之间存在的 Buffer，如图 4-6 所示。

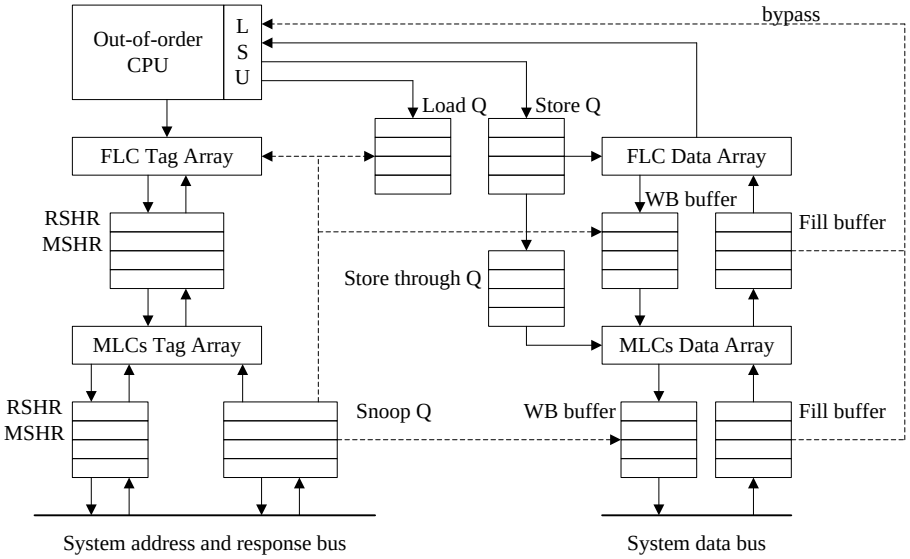

图 4-6 CPU 与数据 Cache 的连接关系

这张图依然不能反映 CPU 与其下 Cache Hierarchy 间的关系，一个实际的 CPU Core 与其下的存储器子系统间的连接异常复杂。不同的处理器架构其存储器子系统的实现也有较大的差别。但是对于一个存储器子系统而言，其所担负的主要任务依然明晰。

存储器子系统的首要任务是将所访问的数据经由各级 Cache，最后传递到距离 CPU Core 最近的一级缓冲，即进行数据传送；另外一个任务是使用合适的机制管理与这些数据相关的状态信息，包括 Cache 的 Tag，MSHR 和其他复杂状态信息；最后可能也是需要额外关注的是，存储器子系统需要考虑本系统所使用的 Consistency Model 和 Coherence Protocol。

在一个存储器子系统中，依然存在若干级子系统。其中每一个子系统大体由数据单元包括 Data Array 和相关 Buffer，Cache 控制器和连接通路这三大部分组成。这个子系统与其上和其下的子系统通过各类 Buffer 进行连接，协调有序地完成存储器子系统的三大任务。

上文已经多次提及 LSQ 和 MSHR，而图 4-6 中的 RSHR(Request Status Handling Register)与 MSHR 并不等同，MSHR 的主要功能是处理来自 CPU Core 访问引发的 Cache Miss 请求，而 RSHR 主要处理来自存储器控制器的 Coherence 请求。在有些微架构中并没有设置 RSHR，而采用了其他类似的部件实现。

Fill Buffer 和 MSHR 协调工作暂存来自其下 Cache Hierarchy 的数据，在其中可能只保存了部分 Cache Block；Store through Queue 主要用于 Defer Write 和 Write Combining 请求。Writeback Buffer 暂存 Evicted Cache Line，其主要目的是 Defer Writeback 这个总线 Transaction，保证其下的 Cache Hierarchy 可以优先处理 Cache Miss 请求。

上文提及的 Buffer 绝非 FLC/MLCs Controller 使用的全部 Buffer，还有许多 Buffer 与 Cache 的预读，Cache Block 的替换状态，以及 Cache Read/Write/Evict Policy 相关。不同的微架构采用了不同的实现策略。在这些不同的实现策略中所关注的重点依然是 Bandwidth，Latency 和 Consistency。

为实现以上任务，Cache Controller 需要处理几大类数据请求和消息。首先是 Data Transfer 请求，即进行数据块的搬移；其次是 Data Transfer Replies，这个 Reply 报文可以带数据如存储器读请求使用的应答报文，也可以不带数据；之后是 State Inquiry/Update Requests 和 State Inquiry/Update Replies，该类报文为保证数据缓冲间的状态信息的 Consistency。

在一个实际的处理器系统中，进一步考虑到多级 Cache Controller 和外部设备，这几大类数据请求与消息会进一步分为更多的子类，以维护 Cache 协议与状态机的正常运行。不同的处理器系统使用了不同的 Cache 协议与状态机，使用了不同的组成结构，进一步加大了 Cache Controller 的设计难度。在 ccNUMA 处理器系统中，包含许多与 Coherence 相关的 Data 和 Message。这些内容与 Cache Memory 的控制逻辑和协议状态机相关。

另外一个与控制逻辑和协议状态机直接相关的是 Cache Block 状态，MOESIF 只是其中的部分状态，还有许多用于 Cache 层次结构互连，用于 Consistency 层面的 Base 状态，还有一些 Transient 状态用于处理 Cache Block 状态切换时出现的 Race Condition 条件。

在不同的 Cache Controller 中，如 FLC/MLCs/LLC 和 Directory Cache Controller，这些状态即便名称类似，其定义依然有所差异。这些状态位之间相互关联也相互影响，进一步提高了 Cache Controller 的设计复杂度。

我们暂时忽略 Cache Controller 使用的其他状态位，重点关注其中一个较为特殊的状态位，Inclusion Property[83]。Inclusion Property 的发明者 Wen-Hann Wang 先生最后加入了 Intel，影响了 Intel 从 P6 直到 Sandy Bridge 微架构的多级 Cache Hierarchy 设计。Wen-Hann 先生的这一发明使得多级 Cache Hierarchy 间的组成结构除了 Inclusive，Exclusive 之外，多了另外一个选择 NI/NE 结构。

4.4 To be inclusive or not to be

无数经典的体系结构书籍专注于介绍 Inclusive。这使得我所接触的毕业生和工程师很少有 Exclusive 和 NI/NE Cache 的概念，包括几年前的自己。一些甚至是来自处理器厂商的工程师也对此知之甚少。也许我们早已熟悉了 Inclusice 这种 Cache Hierarchy 结构，认为 CPU Core 访问 L1 Cache 中 Miss 后查找 L2 Cache，L2 Cache Miss 后继续查找其下的层次。而现代 CMP 处理器在多级 Cache 的设计中更多的使用着 Exclusive 或者 NI/NE 的结构，纯粹的 Inclusive 结构在具有 3 级或者以上的 Cache 层次中并不多见。

在本节中，我们引入 Inner 和 Outer Cache 的概念。Inner Cache 是指在微架构之内的 Cache，如 Sandy Bridge 微架构中含有 L1 和 L2 两级 Cache，而 Outer Cache 指在微架构之外的 Cache，如 LLC。在有些简单的微架构中，Inner Cache 只有一级，即 L1 Cache，Outer Cache 通常由多个微架构共享，多数情况下也仅有一级。

在有些处理器系统中，Inner Cache 由多个层次组成，如 Sandy Bridge 微架构中，包括 L1 和 L2 Cache，这两级 Cache 都属于私有 Cache，即 Inner Cache。此时 Inclusive 和 Exclusive 概念首先出现在 Inner Cache 中，之后才是 Inner Cache 和 Outer Cache 之间的联系。为便于理解，如果本节约定 Inner Cache 和 Outer Cache 只有一级。

Inclusive Cache 的概念最为直观，也最容易理解，采用这种结构时，Inner Cache 是 Outer Cache 的一个子集，在 Inner Cache 中出现的 Cache Block 在 Outer Cache 中一定具有副本；采用 Exclusive Cache 结构时，在 Inner Cache 中出现的 Cache Block 在 Outer Cache 中一定没有副本；NI/NE 是一种折中方式，Inner Cache 和 Outer Cache 间没有直接关联，但是在实现时需要设置一些特殊的状态位表明各自的状态。

Inclusive Cache 层次结构较为明晰，并在单 CPU Core 的环境下得到了广泛的应用。但是在多核环境中，由于 Inner Cache 和 Outer Cache 间存在的 Inclusive 关联，使得采用多级 Cache 层次结构的处理器很难再使用这种方式。如果在一个处理器系统中，每一级 Cache 都要包含其上 Cache 的副本，不仅是一种空间浪费，更增加了 Cache Coherency 的复杂度。

即便只考虑 2 级 Cache 结构，严格的 Inclusive 也不容易实现。为简化起见，我们假设在一个 CMP 中含有两个 CPU Core。在每一个 Core 中，L1 为 Inner Cache，而 L2 为 Outer Cache，Inner Cache 和 Outer Cache 使用 Write-Back 策略，Core 间使用 MESI 协议进行一致性处理。

我们首先讨论 L1 Cache Block，在这个 Cache Block 中至少需要设置 Modified，Exclusive，Shared 和 Invalid 这 4 种 Stable 状态。由于 Strict Inclusive 的原因，还有部分负担留给了 L2 Cache Block。L2 Cache Block 除了 MESI 这些状态位之外，为了保持和 L1 Cache 之间的 Inclusive，还需要一些额外的状态，如 Shared with L1 Cache，Owned by L2 Cache，L1 Modified and L2 stale 等一些与 L1 Cache 相关的状态。

如果在一个 CMP 中，仅含有两级 Cache，增加的状态位仍在可接受的范围内，但是如果考虑到 L3 Cache 的存在，L3 Cache 除了自身需要的状态之外，还受到 Inclusive 的 L1 和 L2 Cache 的影响，与两级 Cache 相比，将出现更多的组合，会出现诸如 L1 Modified，L2 Unchanged and Shared with L3 这样的复杂组合。

有人质疑采用 Inclusive 结构，浪费了过多的 Cache 资源，因为在上级中存在的数据在其下具有副本，从而浪费了一些空间。空间浪费与设计复杂度间是一个 Trade-Off，在某些场景之下需要重点关注空间浪费，有的需要关注设计复杂度。

通常只有 Outer Cache 的容量小于 Inner Cache 的 4 倍或者 8 倍时，空间浪费的因素才会彰显，此时采用 Exclusive Cache 几乎是不二的选择。而当 Outer Cache 较大时，采用 Inclusive Cache 不仅设计复杂度降低，这个 Inclusive Outer Cache 还可以作为 Snoop Filer，极大降低了进行 Cache Coherency 时，对 Inner Cache 的数据访问干扰。

采用 Inclusive Cache 的另一个优点是可以将在 Inner Cache 中的未经改写的 Cache Block 直接 Silent Eviction。这些是 Inclusive Cache 的优点。但是从设计的角度上看，采用 Inclusive Cache 带给工程师最大的困惑是严格的 Inclusive 导致的 Cache Hierarchy 间的紧耦合，这些耦合提高了 Cache 层次结构的复杂度。

首先考虑 Outer Cache 的 Eviction 过程，由于 Inner Cache 可能含有数据副本，因此也需要进行同步处理。采用 Backward Invalidation 可以简单地解决这个问题，由于 Snoop Filter 的存在，使得这一操作更加准确。不过我们依然要考虑很多细节问题，假如在 Inner Cache 中的副本是已改写过的，在 Invalidation 前需要进行数据回写。

另外一个需要重点关注的是在 Inclusive Cache 设计中存在的 Race Condition。无论是采用何种 Cache 结构，这些临界条件都需要进行特别处理，但是在 Inclusive Cache 中，Inner Cache 与 Outer Cache 的深度耦合加大了这些 Race Condition 的解决难度。

我们首先考虑几种典型的 Race Condition。假设 Outer Cache 由于一个 CPU Core 存储器操作的 Cache Miss 而需要进行 Outer Cache 的 Eviction 操作。由于 Inclusive 的原因，此时需要 Backward Invalidate 其他 CPU Core Inner Cache Block，而在此同时，这个 CPU Core 正在改写同一个 Cache Block。同理假设一个 CPU Core 正在改写一个 Inner Cache Block，而另外一个 CPU Core 正在从 Outer Cache Block 中读取同样的数据，也将出现 Race Condition。

这些 Race Condition 并不是不可处理，我们可以构造出很多模型解决这些问题。这些模型实现的相同点是在 Inner Cache 和 Outer Cache 的总线操作间设置一些同步点，并在 Cache Block 中设置一些辅助状态，这些方法加大了 Cache Hierarchy 协议与状态机的实现难度。解决这些单个 Race Condition 问题，也许并不困难，只是诸多问题的叠加产生了压垮骆驼的最后一根稻草。如果进一步考虑 CMP 间 Cache 的一致性将是系统架构师的噩梦。

这并不是 Inclusive Cache 带来的最大问题。这个噩梦同样出现在 Exclusive 和 NI/NE 结构的 Cache 层次结构设计中，只有每天都在做噩梦还是两天做一个的区别。顶级产品间的较量无他，只是诸多才智之士的舍命相搏。

Inclusive Cache 绝非一无是处，如上文提及的 Snoop Filter，Inclusive LLC 是一个天然的 Snoop Filter，因为在这类 LLC 中拥有这个 CMP 中所有 Cache 的数据副本，在实现中只需要在 LLC 中加入一组状态字段即可实现这个 Filter，不需要额外资源，Nehalem 微架构使用了这种实现方式[12]。

在电源管理领域，使用纯粹的 Inclusive Cache 时，由于 Outer Cache 含有 Inner Cache 的全部信息，因此在 CPU Core 进入节电状态时，Inner Cache 几乎可以全部进入低功耗状态，仅仅维护状态信息，不需要对其内部进行 Snoop 操作。这是采用 NI/NE 和 Exclusive Cache 无法具备的特性。

采用 Exclusive Cache 是 Pure Inclusive Cache 的另一个极端，从设计实现的角度上看，依然是紧耦合结构。在这种 Cache 组成结构中，1 个 Cache Block 可以存在于 Inner Cache 也可以存在于 Outer Cache 中，但是不能同时存在于这两种 Cache 之中。与 Inclusive Cache 相比，Inner 和 Outer Cache 之间避免了 Cache Block 重叠而产生的浪费，从 CPU Core 的角度上看，采用 Exclusive Cache 结构相当于提供了一个容量更大的 Cache，在某种程度上提高了 Cache Hierarchy 的整体 Hit Ratio。

在一个设计实现中，Cache 的 Hit Ratio 和许多因素相关。首先是程序员如何充分发挥任务的 Temporal Locality 和 Spatial Locality，这与微架构的设计没有直接联系。而无论采用何种实现方式，对于同一个应用，提供容量更大的 Cache，有助于提高 Cache 的 Hit Ratio。

由上文的讨论我们可以轻易发现，在使用相同的资源的前提下，使用 Exclusive Cache 结构可以获得更大的 Cache 容量，此时 Cache 的有效容量是 Inner Cache+Outer Cache。这是采用 Inclusive Cache 或者 NI/NE 无法做到的，也是 Exclusive Cache 结构的最大优点。这仅是事实的一部分。

假设在一个微架构中，含有两级 Cache，分别是 Inner 和 Outer，并采用 Exclusive 结构。此时一次存储器读请求在 Inner Miss 且在 Outer Hit 时，在 Inner 和 Outer 中的 Cache Block 将相互交换，即 Inner Cache 中 Evict 的 Cache Block 占用 Outer Cache Hit 的 Cache Block，而 Outer Cache Hit 的 Cache Block 将占用 Inner Cache Evict 的 Cache Block。

如果存储器读请求在 Inner 和 Outer Cache 中全部 Miss 时，来自其下 Memory Hierarchy 的数据将直接进入 Inner Cache。因为 Exclusive 的原因，来自其下 Memory Hierarchy 的数据不会同时进入到 Outer Cache。

在这个过程中，Inner Cache 可能会发生 Cache Block 的 Eviction 操作，此时 Eviction 的 Cache Block 由 Outer Cache 接收，此时 Outer Cache 也可能会继续出现 Eviction 操作。这些因为 Inner 或者 Outer Cache 的 Eviction 操作而淘汰的 Cache Block，也被称为 Victim Cache Block 或者 Victim Cache Line。

在采用 Exclusive Cache 的微架构中，需要首先考虑 Victim Cache Block 的处理。当 CPU Core 进行读操作时，如果在 Inner Cache 中 Miss，需要从 Outer Cache 或者其下的 Memory Hierarchy 中获得数据，这个数据直接进入到 Inner Cache。此时 Inner Cache 需要首先进行 Eviction 操作，将某个 Cache Block 淘汰。这个 Victim Cache Block 需要填充到某个数据缓冲中。可以是 Outer Cache 作为 Victim Cache。即淘汰的 Cache Block 进入 Outer Cache，当然采用这种方法，可能继续引发 Outer Cache 的 Eviction 操作，从而导致连锁反应。

在采用 Exclusive Cache 结构的处理器系统中，Outer Cache 经常 Hit 的 Cache Block 也是 Inner Cache 经常 Evict 的 Cache Block[84]。这与 Wen-Hann 有关 NI/NE Cache 结构的 Accidentally Inclusive 的结论一致，在 NI/NE Cache 结构中，虽然 Inner Cache 与 Outer Cache 彼此独立工作，但是根据统计在多数时间，在 Inner Cache 中 Hit 的 Cache Block 也存在于 Outer Cache。这不是设计的需要，而是一个 Accident，Wen-Hann 将其称为 Accidentally Hit[85]。

[84]和[85]的结果是对同一现象的两个不同角度的观察，这一现象由 Inner Cache 和 Outer Cache 的相互关联引发。对于使用 Exclusive Cache 结构，需要使用某类缓冲存放淘汰的 Cache Block，如 Victim Replication[88]，Adaptive Selective Replication[84]等一系列方式，当然理论界还有更多的奇思妙想。这些内容进入到了比较专业的领域，并不是本篇的重点，本篇仅介绍 AMD K7 系列的 Athlon 和 Duron 微架构的实现方式。

在 Athlon 微架构中，L1 Cache 的大小为 128KB，分别为 64KB Data 和 64KB Instruction Cache，运行频率与 CPU Core Clock 相同，在 Hit 时的 Load-Use Latency 为 3 个 Clock Cycle。L2 Cache 大小为 512KB，运行频率为 CPU Core Clock 的一半[86][①]。L1 与 L2 Cache 之间的比值为 4，这使得 Exclusive Cache 结构成为必然的选择。在 L1 Cache 与 L2 Cache 之间，Athlon 微架构设置了专用的 Buffer，暂存从 L1 Cache 中淘汰的 Cache Block，这个 Buffer 也被称之为 Victim Buffer。L1 Cache，L2 Cache 与 Victim Buffer 的组成结构如所示。

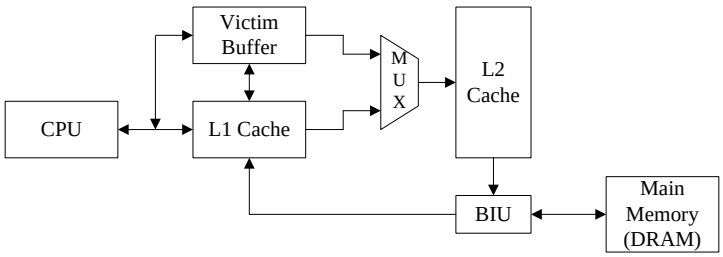

图 4-7 L1 Cache，L2 Cache 与 Victim Buffer 的组成结构[89][②]

在 Athlon 微架构中，Victim Buffer 由 8 个 64B 大小的 Entry 组成。而 L1 Cache，Victim Buffer 和 L2 Cache 之间也是严格的 Exclusive 关系。在 Athlon 微架构盛行的年代，L1 Cache 容量为 128KB 是一个很大的数字，这个数字放到今天也并不小。这使得 CPU Core 访问的多数数据在 L1 Cache 中命中，L1 Cache 和 L2 Cache 间的总线并不繁忙，Victim Buffer 暂存的 Cache Block 可以在总线 Idle 时与 L2 Cache 进行同步。Victim Buffer 很少会因为所有都 Entry 被占用而成为系统瓶颈[86]。

① 本文介绍的 Athlon 处理器采用的 L2 Cache 为 Full-Speed On-Die L2 Cache。
② 原图来自[89]，并有所改动。笔者并不认同该论文的部分内容，仅引用该图。

当 CPU Core 读取的数据在 L1 Cache Miss，而在 Victim Buffer Hit 时，数据将从 Victim Buffer 中传递给 CPU Core 和 L1 Cache，从 L1 Cache 中 Evict 的 Cache Block 将送至 Victim Cache，无需 L2 Cache 的参与。即便数据访问在 L1 Cache 和 Victim Buffer 全部 Miss 时，Athlon 微架构 L2 Cache 的 Load-Use Latency 也仅为 11 个 Clock Cycle，包括 L1 Miss 所使用的 3 个 Cycle。

在 Victim Buffer 为满，CPU Core 访问的数据在 L1 Cache 中 Miss 且在 L2 Cache 中 Hit 的场景中，Victim Buffer 暂存的 Cache Block 需要使用 8 个 Cycle 刷新到 L2 Cache 中；之后 L2 Cache 需要 2 个 Cycles 的 Turnaround 周期将命中的 Cache Block 提交给 L1 Cache，同时将在 L1 Cache 中 Evict 的 Cache Block 送往 Victim Buffer；最后 L2 Cache 还需要 2 个额外的 Turnaround 周期完成整个操作。此时 L2 Cache 的 Load-Use Latency 也仅为 20 个 Cycle。

Athlon 微架构的 Cache Hierarchy 结构在与当时同类处理器的较量中赢得了先机。而在多数应用场景中，微架构间的较量首先发生在 Cache Hierarchy 中。而后的 K8 微架构进一步优化了 Cache Hierarchy 结构。

AMD 在 Magny Cours 微架构中，L3 Cache 作为 L2 Cache 的 Victim[87]。AMD 对 Exclusive Cache 情有独钟，基于 K7，K8 和 K10 的一系列微架构均使用了这一结构。在当时 AMD 凭借着 Cache Hierarchy 结构上这些貌似微弱的领先优势，迎来了其历史上风光无限的时代。Intel 直到 Nehalem 微架构之后才真正超越了 AMD，并开始在 Cache Hierarchy 领域上的一骑绝尘。

AMD 最新的 Bulldozer 微架构在 Cache Hierarchy 层面上做出了较为激进的改革，虽然在 L1 Cache 层面依然使用 VIPT 方式，大小为 16KB，却也不再坚持之前微架构 3 个 Cycle 的 Load-to-use Latency，而是扩大到 4 个 Cycle。Latency 的提高使得 Bulldozer 在 L1 Cache 层面使用 128 位总线带宽成为可能，提高了总线的带宽。

L2 Cache 由两个 CPU Core 共享，最大可达 2MB，这使得 Exclusive Cache 的组成方式失去意义，不出意外 Bulldozer 微架构采用了 NI/NE 方式。最令人意外的是该微架构 L2 Cache 的 Load-to-use Latency 达到了 18~20 个 Cycle，远高于 Nehalem 和 Sandy Bridge 微架构的 10 个 Cycle[74][12][69]。但是与 Nehalem 和 Sandy Bridge 相比，Bulldozer 微架构提高了可并发的 Outstanding Cache Miss 总线请求，L1 Cache Miss 的可并发总线请求为 8 个，L2 Cache Miss 的可并发总线请求为 23 个。

Bulldozer 微架构这些设计必然经过详尽的 Qualitative Research 和 Quantitative Analysis，在没有获得精确的性能数据之前，无法进一步诠释 Bulldozer 微架构的优劣。事实上即便你拿到这些数据，又能够说明多少问题。这些数据很难真正地做到公正全面，在很多情况之下依然是一个片面的结果。此时可以肯定的是，AMD 依然不断前进摸索，在工艺落后 Intel 的事实中正在寻求新的变化，这个公司值得尊敬。

Bulldozer 微架构放弃了 Exclusive Cache 结构必定基于其深层次的考虑。即便是通过简单的理论分析，Exclusive Cache 也并非完美。Exclusive Cache 是 Pure Inclusive Cache 的另一个极端表现方式，Inner Cache 与 Outer Cache 间依然紧耦合联系在一起，这造成了各级 Cache 间频繁的数据交换，尤其是 Inner Cache 和 Victim Cache 之间的数据颠簸。在 CMP 组成的 ccNUMA 处理器系统中这种颠簸更加凸显。

首先在 Outer Cache 作为 Inner Cache 的 Victim 时，Outer Cache 仍需要监控发向 Inner Cache 的 Request 和 Reply 等信息，加大了 Outer Cache Controller 的设计难度。其次在一个 CMP 处理器系统中，某个 CPU Core 发起的 Snooping 操作，必须要同时 Probe 其他 CPU Core 的 Outer Cache Tag 和 Inner Cache Tag。

对 Inner Cache Tag 的 Probe 操作必将影响当前 CPU Core 对 Inner Cache 的访问延时，而这个延时恰是单个 CPU Core 设计所重点关注的内容，处于关键路径。如果每次 Probe 操作都直接访问 L1 Cache 的 Tag，将影响 CPU Core 对 L1 Cache 的访问，可能会 Stall 指令流水线的执行，带来严厉的系统惩罚。

通常情况下，处理器可以使用两种方法解决这类问题。一个是设置专用的 Snoop Filter，处理来自其他 CPU 的 Snoop Transaction，减少对 Inner Cache 不必要的 Probe，对于 Exclusive Cache 设置 Snoop Filter 需要额外的逻辑，而 Inclusive Cache 较易实现 Snoop Filter。另一种方法是复制 Inner Cache 的 Tag，实现 CPU Core 访问 Inner Cache Tag 与 Snoop 的并行操作。

这两种方法都会带来额外的硬件开销，从而加大了 Cache Controller 的设计难度。不仅如此，在 Cache Hierarchy 的设计中每加入一个 Buffer 都要细致考虑 Memory Consistency 层面的问题，各类复杂的 Race Condition 处理和由此带来的 Transient State。这一切使本身已经极为复杂的 Cache Controller，更加难以设计。

NI/NE Cache 是 Exclusive Cache 与 Inclusive Cache 的折衷。Intel 从 P6 处理器开始一直到目前最新的 Sandy Bridge 处理器，一直使用这种结构。Intel 也曾经尝试过 Exclusive 和 Inclusive Cache 结构，最终坚持选择了 NI/NE 结构，也开始了 Intel x86 在 Memory Hierarchy 领域的领先。但是我们不能依此得出 NI/NE 结构是最优的结构，也不能认为这个结构是一个很古老的设计而应该淘汰，在没有得到较为全面的量化结果之前，很难做出孰优孰劣的判断。即便有这些结果也不能贸然作出结论。

在使用 NI/NE Cache 结构时，Inner Cache 与 Outer Cache 的部分将内容重叠，与 Inclusive 结构相比，Cache 容量利用率相对较高，但是仍然不及 Exclusive Cache 结构，因为 Accidentally Hit 的原因，NI/NE Cache 容量利用率与 Inclusive Cache 相比，提高得较为有限。单纯从这个角度出现，在设计中并没有使用 NI/NE Cache 结构的强大动力。

此外采用 NI/NE Cache 方式时，在 Outer Cache 中不保证包含 Inner Cache 中的全部信息，因此其他 CPU Core 的 Snoop Transaction 仍然需要 Probe Inner Cache，这使得 NI/NE Cache 方式依然要复制 Inner Cache Tag，或者加入一个 Snoop Filer。

从以上两方面分析，我们很难得出使用 NI/NE 结构的优点。NI/NE Cache 的支持者显然会反对这些说法，他们会提出很多 NI/NE Cache 的优点。NI/NE Cache 可以在 Outer Cache 中加入 IB(Inclusive Bit)和 CD(Clean/Dirty)位，即可克服 Inclusive 和 Exclusive Cache 存在的诸多缺点，并带来许多优点，如消减 Outer Cache 的 Conflict Miss，充分利用 Inner Cache 与 Outer Cache 间总线带宽，Write Allocate on Inner Cache without Outer Cache interaction，减少不必要的 RFO(Read for ownership)操作等。

我无从辩驳这些确实存在的优点，但是更加关心这些优点从何而来。从一个工程师的角度上看，NI/NE Cache 带来的最大优点莫过于简化了 Cache Hierarchy 的设计。与使用 Inclusive 和 Exclusive Cache 结构相比，采用这种方式使得 Inner Cache 和 Outer Cache 间的耦合度得到了较大的降低，也因此降低了 Cache Hierarchy 的设计难度。

耦合度的降低有助于 Inner 和 Outer Cache Controller 设计团队在一定程度上的各自为政。这种各自为政的结果不仅仅提高了 Cache Controller 的效率，更重要的是提高了设计人员的工作效率。但是这种各自为政只是在一定程度上的，Inner 是 Outer Cache 的 Inner 这个事实决定了 Inner Cache 和 Outer Cache 无论采用何种方式进行互联，依然存在大的耦合度。

NI/NE Cache 结构并不是 Intel x86 处理器的全部。Intel 近期发布的 Nehalem 和 Sandy Bridge 处理器，在使用 NI/NE Cache 的同时，也使用了 Inclusive Cache。Nehalem EP 处理器含有 4 个 CPU Core，其中每一个 CPU Core 含有独立的 L1 和 L2 Cache，其中 L1 和 L2 Cache 为 Inner Cache；而所有 CPU 共享同一个 L3 Cache，这个 L3 Cache 也被称为 LLC 或者 Outer Cache。

其中 L1 Cache 由 32KB 的指令 Cache 与 32KB 的数据 Cache 组成，采用 NI/NE 结构；L2 Cache 的大小为 256KB，采用 NI/NE 结构；L3 Cache 的大小为 8MB，采用 Inclusive 结构，即该 Cache 中包含所有 CPU Core L1 和 L2 Cache 的数据副本。Inclusive LLC 也是一个天然的 Snoop filter。在 LLC 中的每一个 Cache Block 中都含有一个由 4 位组成的 Valid Vector 字段，用来表示 LLC 中包含的副本是否存在于 4 个 CPU Core 的 Inner Cache 中[12]。

当 Valid Vector[i]为 1 时，表示第 i 个 CPU Core 的 Inner Cache 中可能含有 LLC 中的 Cache Block 副本，因为 NI/NE 结构的缘故，Valid Vector[i]为 1 并不保证 Inner Cache 中是 L1 还是 L2 Cache 中含有数据副本，仍然需要进一步的 Probe 操作；当 Valid Vector[i]为 0 时表示，第 i 个 CPU Core 的 Inner Cache 中一定不含有 LLC 中的 Cache Block 副本[12]。

LLC 的 Valid Vector 字段可以简化由 Nehalem EP/EX 处理器组成的 ccNUMA 处理器系统中的 Cache Coherency。因为 LLC 可以代表一个 Nehalem EP/EX 处理器中的所有 Cache，当其他 Socket 进行 Snoop Cache 时，仅需首先访问 LLC 即可，而不必每一次都需要 Probe 所有 Inner 和 Outer Cache，从而简化了 Cache Hierarchy 的设计。

Sandy Bridge 处理器的 Snoop Filter 的设计与 Nehalem EP/EX 处理器类似，只是进一步扩展了 Nehalem EP/EX 处理器的 Valid Vector 字段，以支持内部集成的 GPU[69]。采用 Exclusive Cache 结构的 Magny Cours 在 6MB 的 L3 Cache 中划出了 1MB 的 Probe Filter Directory 作为 Snoop filter[87]，而且提高了 Cache Controller 的设计复杂度。

AMD 在其工艺落后于 Intel，在相同的 Die Size 只能容纳更少晶体管数目的劣势下，使用规模庞大 Probe Filter Directory 有利于多个 CMP 系统间 Cache 一致性的实现，尤其在 4 个或者更多 Socket 的场景。但是即便是在这个场景下，AMD 仅依靠 Probe Filter Directory 并不足以超过 Intel。Sandy Bridge 在 LLC 层面的实现几乎独步天下，不仅运行在 Clock Frequency，而且其 Load-Use Latency 仅为 26~31 个 Cycles[69]。

除此之外 Sandy Bridge 在 LLC 的实现中使用了 Distributed 的方式，将一个 LLC 分解为多个 Slice，其中每一个 CPU Core 对应一个 Slice，CPU Core 经过 Hash 结果访问各自的 Slice。这种 Partitioning Cache Slice 降低了 Cache Coherency 的设计难度，进一步提高了 LLC 的总线带宽，提高了 LLC 的 Scalability，避免了潜在的 Cache Contention[69]。AMD 的 Magny Cours 也支持这种 Cache 组织方式[87]。除此之外，NI/NE with Inclusive Cache 还可以使用一些手段进一步优化，如[90]中介绍的 TLA(Temporal Locality Aware)算法。

无论是 Intel 采用的 NI/NE Inner Cache 加 Inclusive Outer Cache 的结构，还是 AMD 采用的 Exclusive Cache 结构[①]，在 Cache Hierarchy 的设计中，只有耦合程度相对较低的区别。如果从数字逻辑的设计角度上看，这些设计都是耦合的不能再耦合的设计。

在很多场景中，一个完美的设计通常从少数人开始，这也意味着设计的强耦合，一体化的设计有助于整体效率的提高，但也很容易扼杀子团队的创造热情。一个将完美作为习惯的架构师最终可以左右一个设计，创造出一个又一个属于他的完美设计。这样产生的完美，不可继承，不可复制，等待着粉碎后的重建。这样的完美本身是一场悲剧，这些悲剧使得这些完美愈显珍贵。

不同群体对完美的不同认知使得这些悲剧几乎发生在每朝每代。这些完美的人不可轻易复制，使得一个大型设计通常选用多数人基于这个时代赋予认知后的完美，属于多数人的完美。这使得架构师在设计产品时，若只考虑技术层面，而不考虑设计者间的联系，并不称职。这也使得一个优秀的架构师在历经时光痕迹，岁月沧桑后，做出了很多不情愿也不知对错的中庸选择。这些中庸可能是大智慧。而总有一些人愿意去挑战这些中庸。

4.5 Beyond MOESIF

再次回顾与 Cache Coherency 相关的 MOESIF 状态位，却不知从何说起。MOESIF 这些状态位似曾相识，已物是人非。在 CMP 处理器系统中使用的多级 Cache 层次结构和 CMP 间的 Cache Coherency，改变了 MOEIF 这些状态位的原始形态。

[①] Magny Cours 微架构并不是严格意义的 Exclusive Cache，L3 Cache 需要检查 True Sharing 状态[87]。

在一个由多个 CMP 组成的 ccNUMA 处理器系统中，Cache Coherency 包含两方面内容，首先是 Intra-CMP Coherence，其次是 Inter-CMP Coherence。其中 Intra-CMP Coherence 指一个 CMP 内部的 Cache Coherency，而 Inter-CMP Coherence 指 CMP 间的 Cache Coherency，两者之间的关系如图 4-8 所示。

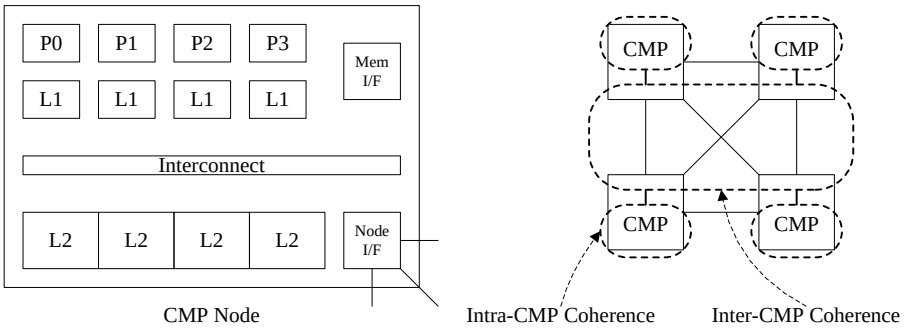

图 4-8 Intra-CMP Coherence 和 Inter-CMP Coherence 之间的关系[91]

在一个 ccNUMA 处理器系统中，Intra-CMP Coherence 需要与 Inter-CMP Coherence 协调工作，完成 Cache 的全局 Coherency。不同的 CMP 处理器采用了不同的 Intra-CMP Coherence 策略，可以是 Share Bus，Ring Bus 或者 Directory，这些策略各有利弊。Inter-CMP Coherence 甚至可以通过软件交换 Message 的方式实现，而为了提高软件的 Programmability，多数系统使用了硬件实现这些 Message 交换。在一个 ccNUMA 处理器系统中使用的 CMP 超过 4 个时，多使用 Directory 结构。

在串行总线替代并行总线的大趋势下，ccNUMA 处理器的数据以及 Coherence Message 通过 Packet 的方式进行传递，会涉及在 Internet 中出现的 Router，NI(Networking Interface)，Flow Control，QoS，Routing Algorithm 等概念。在 ccNUMA 处理器系统中使用的 Interconnection 不但不比 Internet 简单而且复杂得多。K Computer 使用的 6D Mesh/Torus Interconnect[92]结构目前尚无用于 Internet 的可能。

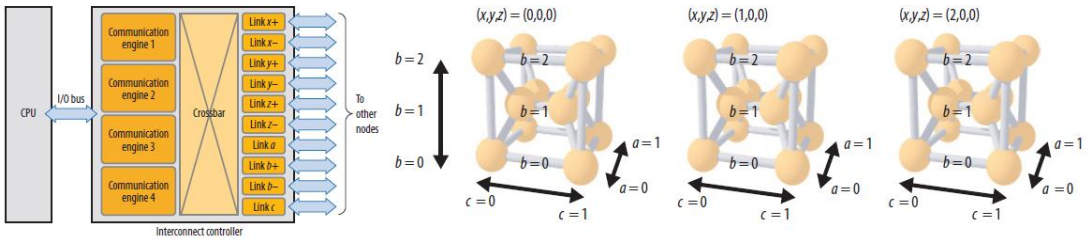

图 4-9 6D Mesh/Torus Interconnect[92]

Supercomputer 使用的 Interconnect 超出了本篇的讨论范围，但是仅从 Cache 的 Coherency 层面进行分析，Inter-CMP Coherence 的设计没有难于 Intra-CMP Coherence，Cache Coherency 依然是越接近 CPU Core 越复杂，Cache 间的互联总线也是越接近 CPU Core 越复杂。这是本节重点介绍 Intra-CMP Coherence 的主要原因。

Intra-CMP Coherence 的复杂程度超过了初学者的想象。其中各级 Cache 之间的关系，及为了处理这些关系而使用的 Cache Block 状态和总线协议均较为复杂。仅是其中使用的 Cache Coherency Protocol 也复杂到了需要使用专门的语言才能将其简约地进行描述。这个语言即 SLICC(Specification Language for Implementing Cache Coherence)，这个语言是有志于深入了解 Cache Coherency Protocol 所需要了解的基础知识。

对于多数人而言，学习 Cache Coherency Protocol 最好的工具是 Simulator 和使用 SLICC 语言书写的这些源代码。虽然在模拟舱中很难学会开飞机，但是如果连模拟舱都没有呆过，很难有人让你开真飞机。学术领域提供了这样的模拟舱学习 Cache Coherency Protocol。

与 CMP 处理器相关的模拟器主要有 SESC，Simics，M5 和 GEMS。Simics 最初由 Virtutech 开发。当时的 Virtutech 和飞思卡尔在多核处理器上进行了一些合作，虽然 Simics 是商业产品，我们当时却有机会获得无需付费的 License。如获至宝。Intel 后来收购了 Virtutech。

M5 和 GEMS 主要用于教学与科研，是一个免费而且代码公开的模拟器。M5 侧重 CPU Model，ISAs 等方面；GEMS 最重要的组成部件是 Cache Coherency Protocol 和 Ruby Memory Hierarchy。M5 和 GEMS 具有很强的互补性，也正是因为这个原因，这两个 Simulator 逐步融合为 GEM5 Simulator[93]。

GEM5 是我目光所及范围内，由脚本语言书写的最复杂的系统。GEM5 吸纳了 M5 和 GEMS 的主要优点，支持 Alpha，ARM，SPARC 和 x86 处理器，支持 Functional 和 Timing Simulation，提供 FS(Full-System)和 SE(Syscall Emulation)两种方式。即便是 Android 这样复杂的系统也可以运行在 GEM5 Simulator 之上。在 GEM5 Simulator 中包含许多内容，本节重点关注使用 SLICC 语言实现的 Cache Coherence Protocol。

GEM5 Simulator 的源代码可在 http://www.gem5.org/Download 中下载。在这些源代码中，我们重点关注./src/mem/protocol 目录。这个目录包含 GEM5 支持的所有 Cache Coherence Protocol，包括 MI_example，MOESI_hammer，MOESI_CMP_token，MOESI_CMP_directory 和 MESI_CMP_directory，由.slicc 和.sm 两类文件组成。其中 xyz.slicc 文件包含实现 xyz protocol 所需要的所有.sm 文件，而在.sm 文件中包含各级 Cache Controller 的具体实现。

在每一个.sm 文件中，首先包含一个 machine(L1Cache, "MSI Directory L1 Cache CMP")，用以指出当前 Cache Controller 的名称和采用的协议。其后是由 Network Ports，States，Events，Ruby Structure，Trigger Events，Actions 和 Transitions 组成的基本模块。

- Network Ports 通过 MessageBuffer 原语定义，描述 Cache 使用的“From”和“to”数据通路，使用的 virtual_network 编号等一系列信息。
- States 指 Cache Block 使用的状态位，由 Base 和 Transient States 组成。这些状态与 Cache Coherence Protocol 相关，超越了常用的 MOESIF 这些状态位。多级 Cache 层次结构赋予了 MOESIF 这些基本状态位以新的表现形式。
- Events 将引起 Cache Block 的状态迁移，包括 Load，Ifetch，ACK，NAK 等一系列事件。这些 Event 可以由外部也可以由内部产生。
- Ruby Structures 用于定义当前程序内部使用的 Variables，Structures 和 Functions，也可以引用当前程序之外的 Ruby Structures。
- Trigger Events 定义 Input 和 Output 端口。每一个端口将和一个 MessageBuffer 绑定。Input 端口可以设置执行代码，当前 Components Wakeup 时，这段代码将首先检查端口中是否有 Message，如果有则使用 peek 函数其存放到一个内部变量 in_msg 中，之后可以对这个 in_msg 进行分析，并做相应的操作，如 Trigger 某个 Event。
- Actions 描述 Cache Block 进行状态迁移时所需要进行的操作，如 enqueue 和 dequeue 操作等等。
- Transitions 由 Starting State，Triggering Event，Final State 组成和一系列 Actions 组成。如果没有 Final State 参数，说明完成 Actions 后，将停留在 Starting State。

对于某一个具体的 Cache Coherence Protocol 而言，这些.sm 文件累加在一起也并不多。其中最复杂的 MOESI_CMP_directory Protocol，其总代码也仅有 5,565 行。只是这些.sm 文件相互关联，相互依托，需要首先理解的是 Cache 的数据通路，Cache 间总线的各类 Transaction，基本的 Cache 一致性模型，CPU 的 LSU 部件和许多与体系结构相关的基础知识。

在掌握与 Cache 相关的基础知识之后，阅读并理解这些源代码不会成为太大的障碍。建议读者从最简单的 MI_example Protocol 开始直到较为复杂的 MOESI_CMP_directory Protocol。下文将重点讨论 GEM5 的 MOESI_CMP_directory protocol，我并不是要挑战 GEM5 中最难的 Protocol，而是 GEM5 的网站提供了一些基本的 State Transition 的 FSM(Finite-State Machine)转换图，省去了许多工作。

我们首先列出 MOESI_CMP_directory protocol 使用的 Coherence Messages，这些 Messages 由两部分组成，一个是 Coherence Request，另一部分是 Coherence Response，其详细描述如表 4-2 与表 4-3 所示。

表 4-2 MOESI_CMP_directory protocol 使用的 Coherence Request[94]

Message	Description
GETX	Request for exclusive access.
GETS	Request for shared permissions to satisfy a CPU's load or IFetch.
PUTX	Request for writeback of cache block.
PUTO	Request for writeback of cache block in owned state.
PUTO_SHARERS	Request for writeback of cache block in owned state but other sharers of the block exist.
PUTS	Request for writeback of cache block in shared state.
WB_ACK	Positive writeback ack
WB_ACK_DATA	Positive writeback ack with data
WB_NACK	Negative writeback ack
INV	Invalidation request. This can be triggered by the coherence protocol itself, or by the next cache level/directory to enforce inclusion or to trigger a writeback for a DMA access so that the latest copy of data is obtained.
DMA_READ	DMA Read Request
DMA_WRITE	DMA Write Request

在 GETX，GETS，PUTO 等往往包含一些前缀，其中 L1_前缀指来自 L1 Cache 的请求，而 Fwd_前缀指来自其他 CMP 的请求，而 Own_前缀指来自当前 CMP 的请求。这些 Request 比较基本，需要认真理解。

表 4-3 MOESI_CMP_directory protocol 使用的 Coherence Response[94]

Message	Description
ACK	Acknowledgment, responder doesn't have a copy
DATA	Acknowledgment, responder has a data copy
DATA_EXCLUSIVE	Data, no processor has a copy
UNBLOCK	Message to unblock next cache level/directory for blocking protocols.
UNBLOCK_EXCLUSIVE	Unblock, we're in E/M
WRITEBACK_CLEAN_DATA	Clean writeback with data
WRITEBACK_CLEAN_ACK	Clean writeback without data
WRITEBACK_DIRTY_DATA	Dirty writeback with data
DMA_ACK	Ack that a DMA write completed

不同的 Cache Controller 使用不同的 Coherence Request 和 Response Message。对于 L1 Cache 其上的请求来自 CPU Core，也被称为 Sequencer，并可将请求发至 L2 Controller，Response 可以来自 CPU Core 也可以是 L2 Controller。L1 Cache Controller 还需要处理来自 CPU Core 的 Load，Store 和 IFetch 请求，除此之外其内部还可能产生 L1_Replacement 请求。

使局势更加错综复杂的是这些 Request 和 Response 使用了不同的 Virtual Network，因为 Request 和 Response 之间的同步而产生的 Race Condition 并不容易难以解决。这也决定了在 L1 或者 L2 Cache Block 中并不是只有 MOESIF 这样简单的状态，每一个 Cache Controller 都使用着不同的状态位。这些状态位及其迁移组成了一个本不是采用 FSM 能够描述清楚的 FSM。

在 MOESI_CMP_directory Protocol 的 L1 Cache Controller 中包含了 7 个 Stable 的状态位。

- I。当前 Cache Block 不含有有效数据。与传统定义相同。
- S。当前 Cache Block 用于 1 个或者多个数据副本，与传统定义相同。Read_Only。
- O。当前 Cache Block 含有有效数据，与传统意义的 O 状态位相同。Read_Only。
- M。仅是当前 Cache Block 含有有效数据，与传统意义的 E 状态位相同。Read_Only。
- M_W。仅是当前 Cache Block 含有有效数据，与传统意义的 E 状态位相同。Cache Block 处于该状态时，禁止 Replacement 和 DMA 对其的访问，经过一段延时后，将自动转换为 M 状态。Store:Hit 时，该状态将迁移到 MM_W 状态。Read_Only。
- MM。与传统意义的 M 状态位相同。Read_Write。
- MM_W。与传统意义的 M 状态位相同。Cache Block 处于该状态时，禁止 Replacement 和 DMA 对其的访问，经过一段延时后，将自动转换为 MM 状态。Read_Write。

由这些状态组成的 FSM 如图 4-10 所示。

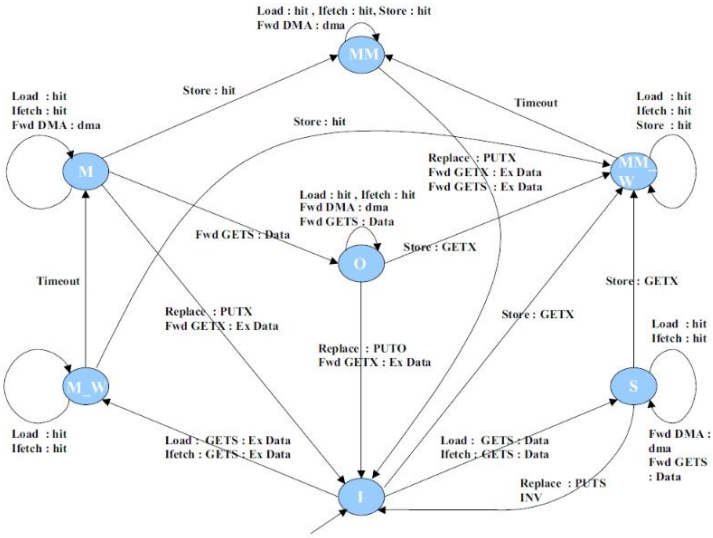

图 4-10 L1 Cache Controller 的 FSM[95]

值得留意的是 MM_W 到 MM，M_W 到 M_W 状态的迁移。MM_W 状态由 Store:GETX 和 Store:Hit 触发，由 S，I，O 和 M_W 状态迁移而来，为减少 DMA 操作对 Cache 状态机的影响，同时为了避免很快替换最近改写的 Cache Block，FSM 要求最近改写的 Cache Block 在 MM_W 状态停留一段时间后，才能进入 MM 状态。在 M_W 设置的 Timeout 是同一个道理。

在 MOESI_CMP_directory Protocol 中，L2 Cache Controller 的设计难度更加复杂，因为该 Controller 需要兼顾 L1 Cache Controller 和其下的 Directory Controller。L2 Cache Controller 共设置了 14 个 Stable 状态，处理 CMP_directory Protocol。这些状态共分为 4 组，其含义与简要说明如表 4-4 所示。

表 4-4 L2 Cache Controller 使用的 Stable 状态[95]

Intra-Chip Inclusion	Inter-Chip Exclusion	State	Description
Not in any L1 or L2(1)	May be present	NP/I	The cache block at this chip is invalid.
Not in L2, but in L1 or more L1s(2)	May be present at other chips	ILS	The cache block is not present at L2 on this chip. It is shared locally by L1 nodes in this chip.
		ILO	The cache block is not present at L2 on this chip. Some L1 node in this chip is an owner of this cache block.
		ILOS	The cache block is not present at L2 on this chip. Some L1 node in this chip is an owner of this cache block. There are also L1 sharers of this cache block in this chip.
	Not present at any other chip	ILX	The cache block is not present at L2 on this chip. It is held in exclusive mode by some L1 node in this chip.
		ILOX	The cache block is not present at L2 on this chip. It is held exclusively by this chip and some L1 node in this chip is an owner of the block.
		ILOSX	The cache block is not present at L2 on this chip. It is held exclusively by this chip. Some L1 node in this chip is an owner of the block. There are also L1 sharers of this cache block in this chip.
In L2, but not in any L1(3)	May be present	S	The cache block is not present at L1 on this chip. It is held in shared mode at L2 on this chip and is also potentially shared across chips.
		O	The cache block is not present at L1 on this chip. It is held in owned mode at L2 on this chip. It is also potentially shared across chips.
	Not present	M	The cache block is not present at L1 on this chip. It is present at L2 on this chip and is potentially modified.
Both in L2, and 1 or more L1s(4)	May be present	SLS	The cache block is present at L2 in shared mode on this chip. There exist local L1 sharers of the block on this chip. It is also potentially shared across chips.
		OLS	The cache block is present at L2 in owned mode on this chip. There exist local L1 sharers of the block on this chip. It is also potentially shared across chips.
	Not present	OLSX	The cache block is present at L2 in owned mode on this chip. There exist local L1 sharers of the block on this chip. It is held exclusively by this chip.

这些 Stable 状态位分为两大部分，4 个小组，一部分用于反映 L2 Cache 的 Coherence 状态，另一部分用于 CMP 内部的多个 L1 Cache。

- 第 1 组是{NP, I}，表示在本 CMP 中 Cache Block 无效，可能其他 CMP 具有数据副本。
- 第 2 组是{ILX, ILOX, ILOSX, ILS, ILO, ILOS}。其中{ILX, ILOX, ILOSX}确定当前 CMP 是否对一个 Cache Block 具有排他的 Ownership，即 Exclusive Ownership，其他 CMP 不会有

数据副本，当然这并不保证当前 CMP 某个 CPU Core 也具备这种 Exclusive Ownership；{ ILS, ILO, ILOS}，保证在当前 CMP 中的 L2 Cache 没有数据副本，但是并不保证其他 CMP 具有数据副本。

- 第 3 组为{S, O, M}，这些状态是对于 L2 Cache 而言，传统的 S, O 和 M 状态位，此时当前 CMP 中的 L1 Cache 不含有数据副本。
- 第 4 组为{ SLS, OLS, OLSX}与第 3 组类似，只是在当前 CMP 的 L1 Cache 中存在副本。

第 1, 3 和 4 组状态位组成的 FSM，即 Intra-Chip Inclusion FSM 如图 4-11 所示。

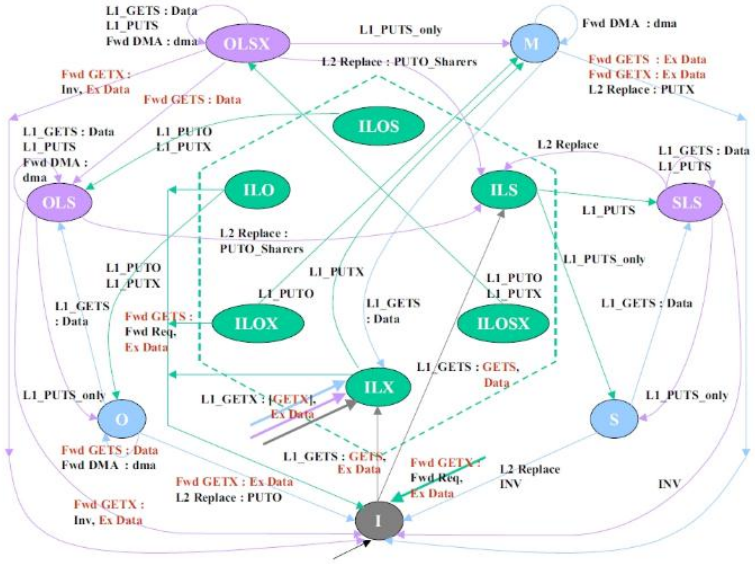

图 4-11 Intra-Chip Inclusion FSM[95]

{ILX, ILOX, ILOSX, ILS, ILO, ILOS}这些状态位属于 L2 Cache Block，却与 L1 Cache 有直接的联系。由这些状态组成的 FSM 如图 4-12 所示。

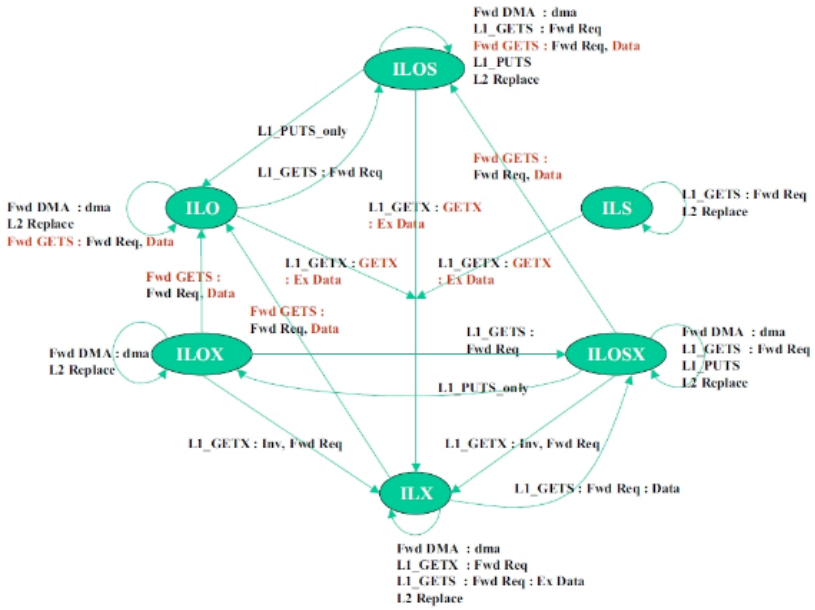

图 4-12 与 L1 Cache Block 相关的 FSM[95]

在与 L1 Cache Block 相关的状态位中，{ILX, ILOX, ILOSX}状态位可以反映当前 CMP 是否 Exclusive Owned Cache Block，此时进行状态迁移时，不用向 CMP Directory 发送 GETX 请求。当然 Cache Block 处于 M，OLSX 时，也不需要向 CMP Directory 发送 GETX 请求；否则都需要发送 GETX 请求。如何实现这个 GETX 请求与 CMP 处理器间使用的网络拓扑结构相关，可以是广播，也可以是指定的 CMP。

无论采用哪一种种 CMP 间的一致性，在 Directory Controller 中所使用的状态位都不及 L1 或者 L2 Cache Controller 复杂。在 CMP_directory Protocol 中的 Directory Controller 设置了 4 个状态位，如下所示。

- M 位有效时表示 Cache Block 仅在当前 CMP 中具有有效数据副本，可能与主存储器不一致，也可能一致，即包含传统的 E。
- O 位有效时表示 Cache Block 在当前 CMP 中具有有效数据副本，而且在其他 CMP 中含有数据副本。数据副本与主存储器不一致。
- S 位有效时表示 Cache Block 在当前 CMP 中具有有效数据副本，而且在其他 CMP 中含有数据副本。数据副本与主存储器可能一直，也可能不一致。
- I 位有效表示 Cache Block 无效。

由这些状态位组成的 FSM 可能是最简单的，如图 4-13 所示。

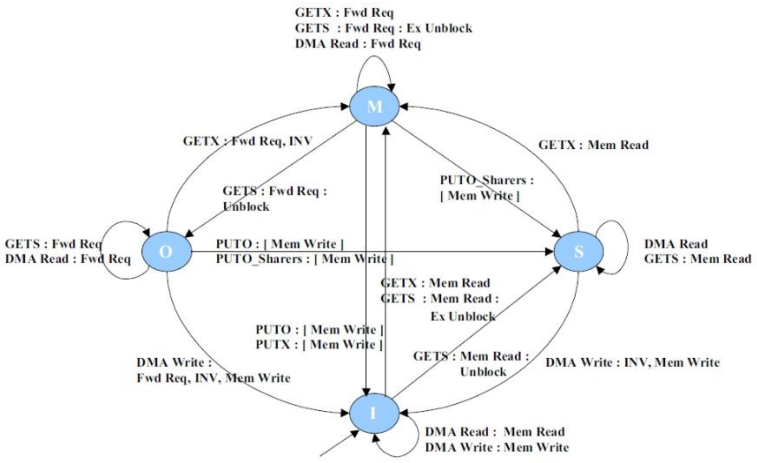

图 4-13 Directory Controller 使用的 FSM[95]

我一直盼望整个 Cache Controller 的 FSM 只有图 4-10，图 4-11，图 4-12 和图 4-13 所示的内容这样简单。倘若真的如此，我们即便再多引入一级 Cache 结构也并不会太复杂，依然存在单个个体就能够理解整个 Cache Hierarchy 的可能。

事实上在图 4-10 中，O 不能直接迁移到 I；在图 4-11 中 O 不能直接迁移到 I；在图 4-12 中，ILO 不能直接迁移到 ILX；在图 4-13 中，O 也不能直接迁移到 I。在整个 Cache Hierarchy 的设计中，合理有效解决因为 Memory Consistency 引发的 Race Condition 贯彻始终，也引入了过多的所谓“Safe State”，这些 Safe State 被称为 Transient State。我们以图 4-10 中简单到不能再简单的 I 到 MM_W 的状态迁移说明这些 Transient State 的作用。

L1 Cache 与 Sequencer 直接相连，当 CPU Core 进行一次存储器 Store 操作，将引发一系列复杂的操作，这些操作不仅与采用的 Cache Coherence Protocol 相关，与采用的 Write Policy 和使用的 Memory Consistency 模型也有直接关系。为简化起见，我们选用 Weekly Ordered Memory Model，Write Policy 为 Write-Back Write-Allocate，Coherence Protocol 为本节所重点描述的 MOESI_CMP_directory。

即便在这种情况之下，我依然会省去更多的细节，忽略绝大多数状态信息和绝大多数总线 Transaction，仅书写 Store 操作中的一种状态转移情况。我希望可以在一个较少的篇幅内完成最基本的说明，给予对这部分内容有兴趣的读者一个入门级别的描述。

在 CPU Core 中，一个 Store 操作，引起的结果异常深远，这个 Store 操作首先到达 L1 Cache Controller，之后从一个 Stable State 进入到一个 Safe State，之后穿越当前 CMP 处理器系统的 L2/Directory Cache Controller，到达和这个 Store 操作相关的其他 CMP 处理器系统后，再逐级回溯到当前 CMP 处理器的 L1 Cache Controller 后，再次进入到另一个 Safe State，最终小心翼翼地完成整个操作后进入到最终的 Stable State。对于 MOESI_CMP_directory，I 状态转移到 MM_W 状态，需要经历 IM，OM 两个 Transient State 状态，其过程如图 4-14 所示。

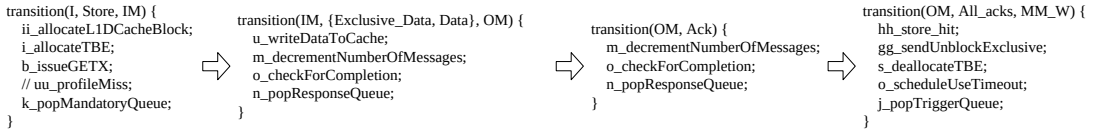

图 4-14 状态 I 至 MM_W 的迁移过程

本次 Store 操作很幸运，因为在其 Miss 时，恰好有一个状态位为 I 的未用 Cache Block，这种情况甚至比在 Cache 中 Hit 容易处理。首先 I 状态将迁移到 IM 状态，在迁移的过程中，完成 Cache Block 的分配，发送 GETX 操作，获得当前 Cache Block 的控制权。

在 IM 状态时，如果收到 Exclusive_Data 或者 Data 后，将迁移到 OM 状态，在迁移的过程中，将进行数据 Merge 并写入 Cache Block 中。为了维护 Store 的 Global View 的统一，此时虽然数据已经写入到 Cache Block，依然不能通知 CPU Core 当前 Store 操作完成，必须要等待 CMP 处理器系统中所有 CPU Core 的 ACK。一次 GETX 操作可能会产生多个 ACK。当收齐所有 ACK 后，L1 Cache Controller 将 Trigger All_acks 这个重要的 Message。

GEM5 Simulator 依然简化了 IM 状态的处理，因为 Data/Exclusive_Data 和 ACK 可能是异步的，即便收到了所有的 ACK，Data/Exclusive_Data 也未必到达，因为 ACK 与 Data 的传递可能使用了不同的路径。更为重要的是什么叫 All_acks，不同的 Protocol 采用的方法并不相同。

Intel 的 MESIF Protocol[34]引入了一个 F(Forward)状态。在 ccNUMA 处理器系统中，可能在多个处理器的 Cache 中存在相同的数据副本，在这些数据副本中，只有一个 Cache Block 的状态为 F，其他 Cache Block 的状态为 S。

当一个数据请求方读取这个数据副本时，只有状态为 F 的 Cache 行，可以将数据副本转发给数据请求方，状态位为 S 的 Cache 不能转发数据副本。数据请求方从状态位为 F 的 Cache Block 中收到 ACK 后，即认为收齐了所有 ACK。这种方法可以减少 Bus Traffic，其他 ccNUMA 处理器会采用其他方法避免这种 Bus Traffic，谈不上是伟大的发明。

在 OM 状态收到 All_acks 后，最终迁移到 MM_W 状态，此时首先通知 CPU Core Store 操作执行完毕，然后宣布当前 Cache Block 处于 Exclusive 状态，之后设置 Timeout，在经过一段延时之后，MM_W 状态将自动迁移到 MM 状态。

在 MOESI_CMP_directory Protocol 的 L1 Cache Controller 中与 IM 和 OM 类似的 Transient State 还有 SM，IS，SI，OI，MI 和 II，共计 8 个；L2 Cache Controller 中有 50 个这样的兄弟；Directory Controller 中有 15 个这样的姐妹。所有这些状态累积在一起形成的 FSM 不应该用 2 维方式进行描述，至少需要 3 维。

我曾试图用 2 维 FSM 描述 MESI_CMP_directory Protocol，最后发现无法找到尺寸合适的纸张。也许使用 PDA(Pushdown Automation)是一个不错的想法，有时间我会进行这方面的尝试。更有人明知我看不懂和 Quantum 相关的任何内容，依然愚弄我，向我推荐了 QFA(Quantum Finite Automata)。

这些 SLICC 描述可以方便地转换为 HTML 文件，通过点击鼠标就可发现 State 之间的迁移，如 http://www.cis.upenn.edu/~arraghav/protocols/VI_directory/L1Cache.html 所示，只是在状态过多时，使用这样的方法并不方便。

过多的状态增加了 Cache Coherence Protocol 的设计复杂度，即便是 GEM5 的实现也并不容易完全掌握。每次看到莘辛学子们强读 GEM5，有种在刀尖上行走的感觉，为他们骄傲的同时祝他们好运。GEM5 不是 Cache Hierarchy 的全部，更不是 Cache Coherence Protocol 的全部。GEM 没有实现最基础的 atomic 操作，没有实现 Memory Barrier，没有连接 Cache 间的 Buffer，也没有更多的 Cache 层次结构，如 L3 Cache，没有太多的实现细节。

GEM5 并没有提供有效的 Verification 策略。而在某种意义上说，Cache Verification 的难度甚至超过 Design。彻底验证哪怕是最简单的 Cache Coherence Protocol 也并不是如想象中简单，不仅在于 Verification 的过程中，会遇到牛毛般匪夷所思的几角旮旯，还在于 Verification 策略本身的完备。

Design 和 Verification 间存在着密切联系，存在着诸多选择。这使得在一个实际的 Cache Hierarchy 设计中，取舍愈发艰难。在你面前有太多的选择，无法证明哪一个更为有效，在你面前也没有什么 Formal Proof Methods。

在人类的智慧尚没有解决交换舞伴这样平凡的 NP Hard 的前提下，可能最完美最有效的策略只有 O(N!)级别的穷举。这些平凡的问题一直等待着不平凡的解读。解决这些问题的人的不朽将远远超越计算机科学这个并不古老的学科。这一切羞辱着天下人的智慧，令世界上最快的 K Computer 不堪重负。更多的人去依赖没有那么可靠的 Quantitative Approach。在这些 Approach 中，谁去验证了所使用的 Models，Theories 和 hypotheses。

计算机科学在这些不完美中云中漫步。几乎没有人知道准确的方向。有些技巧是没有人传授，因为没有人曾经尝试过。这使得这个星球最顶级的 Elite，甘愿穷其一生去做猎物，等待猎人的枪声。没有人欣赏这种方法。更多的是饥饿着的愚钝着的执着。

Stay Hungry，Stay Foolish。

4.6 Cache Write Policy

如果有可能，我愿意放弃使用 Write 操作，Write 操作不是在进行单纯地进行写，而是将 Read 建立的千辛万苦异常小心地毁于一旦，是 Cache Hierarchy 设计苦难的发源地。没有写操作在 Cache Block 中就不会有这么多状态，更没有复杂的 Memory Consistency 模型。

很多人都不喜欢写操作，这并不能阻挡它的真实存在。绝大多数人痛恨写操作，也不愿意去研究如何提高写操作的效率，写虽然非常讨厌，但是很少在速度上拖累大家，只是写太快了，带来了许多副作用经常影响大家。不允许写操作使用 Cache 貌似是个方法，却几乎没有程序员能够做到这一点。过分的限制写操作也并非良策，凡事总有适度。

这使得在 Cache Hierarchy 设计中读写操作得到了区别对待。读的资质差了一些，有很多问题亟待解决，最 Critical 的是如何提高 Load-Use Latency；对于写，只要不给大家添乱就已经足够，Cache Bus 的 Bandwidth 能少用点就少用点，能降低一点 Bus Traffic 就算一点，实在不能少用，就趁着别人不用的时候再用。从总线带宽的角度上看，Load 比 Store 重要一些，在进行 Cache 优化时，更多的人关心读的效率。

在一个程序的执行过程中不可能不使用 Write 操作。如何在保证 Memory Consistency 的前提下，有效降低 Write 操作对 Performance 的影响，如何减少 Write Traffic，是设计的重中之重。在一个处理器系统中，Write 操作需要分两种情况分别讨论，一个是 Write Hit，另一个是 Write Miss。Write Hit 指在一次 Write 操作在进行 Probe 的过程中在当前 CMP 的 Cache Hierarchy 中命中，而 Write Miss 指没有命中。

我们首先讨论 Write Hit，从直觉上看 Write Hit 相对较为容易，与此相关的有些概念几乎是常识。Write Hit 时常用的处理方法有两种，一个是 Write Through，另一个是 Write Back[①]。相信绝大部分读者都明白这两种方法。Write Through 方法指进行写操作时，数据同时进入当前 Cache，和其下的一级 Cache 或者是主存储器；Write Back 方法指进行写操作时，数据将直接写入当前 Cache，而不会继续传递，当发生 Cache Block Replace 时，被改写的数据才会更新到其下的 Cache 或者主存储器中。

很多人认为 Write-Back 在降低 Write Traffic 上优于 Write-Though 策略，只是 Write-Back 的实现难于 Write-Though，所以有些低端处理器使用了 Write-Though 策略，多数高端处理器采用 Write-Back 策略。

这种说法可能并不完全正确，也必将引发无尽的讨论。在没有拿到一个微架构的全部设计，在没有对这个微架构进行系统研究与分析之前，并不能得出其应该选用 Write-Through 或者是 Write-Back 策略的结论。即便拿到了所有资料，进行了较为系统的 Qualitative Research 和 Quantitative Analysis 之后，你也很难得出有说服力的结论。

在目前已知的高端处理器中，SUN 的 Niagara 和 Niagara 2 微架构的 L1 Cache 使用 Write-Through 策略[81]，AMD 最新的 Bulldozer 微架构也在 L1 Cache 中使用 Write-Through 策略[74]。Intel 的 Nehalem 和 Sandy Bridge 微架构使用 Write-Back 策略。Write-Through 和 Write-Back 策略各有其优点和不足，孰优孰劣很难说清。

Norman P. Jouppi 列举了 Write-Through 策略的 3 大优点，一是有利于提高 CPU Core 至 L1 Cache 的总线带宽；二是 Store 操作可以集成在指令流水线中；三是 Error-Tolerance [96]。这篇文章发表与 1993 年，但是绝大部分知识依然是 Cache Write Policies 的基础，值得借鉴。

其中前两条优点针对 Direct Mapped 方式，非本节的重点，读者可参考[96]获得细节。制约 Cache 容量进一步增大的原因除了成本之外，还有 Hit Time。如上文已经讨论过的，随着 Cache 容量的增加，Hit Time 也随之增大。而 Cache 很难做到相对较大，这使得选择 Direct Mapped 方式时需要挑战 John 和 David 的 2:1 Cache Rule of Thumb，目前在多处处理器系统中都不在使用 Direct Mapped 方式。

对这些 Rule of Thumb 的挑战都非一蹴而就。在某方面技术取得变革时，首先要 Challenge 的恰是之前的 Rule of Thumb。善战者择时而动，而善战者只能攻城掠地，建立一个与旧世界类同的新世界，不会带来真正的改变。

革新者远在星辰之外，经得住大悲大喜，大耻大辱，忍受得住 Stay Hungry Stay Foolish 所带来的孤单寂寞。是几千年前孟子说过的“富贵不能淫，贫贱不能移，威武不能屈。居天下之广居，立天下之正位，行天下之大道。得志，与民由之，不得志，独行其道”。几千年祖辈的箴言真正习得是西方世界。

忽略以上这段文字。我们继续讨论 Norman P. Jouppi 提及的使用 Write-Through 策略的第三个优点，Error-Tolerance。CMOS 工艺在不断接近着物理极限，进一步缩短了门级延时，也增加了晶体管的集成度，使得原本偶尔出现的 Hard Failures 和 Soft Errors 变得更加濒繁。这一切影响了 SRAM Cells 的稳定性，导致在基于 SRAM Cells 的 Cache Memory 中，Hard 和 Soft Defects 不容忽视。

采用 Write-Back 策略的 Cache 很难继续忍受 Single Bit 的 Defects，被迫加入复杂的 ECC 校验；使用 Write-Though 策略，Cache 因为仍有数据副本的存在只需加入 Parity Bit。与 ECC 相比，Parity Bit 所带来的 Overhead 较小。这不是采用 ECC 校验的全部问题，在一个设计中，为了减少 Overhead，需要至少每 32 位或者更多位的参与产生一次 ECC 结果。这些 Overhead 的减少不利于实现 Byte，Word 的 Store 操作，因为在进行这些操作时，需要首先需要读取 32 位数据和 ECC 校验，之后再 Merge and Write 新的数据并写入新的 ECC 校验值。

[①] 还有一种是上文提及的 Write Once，Write Once 是 Write Through 和 Write Back 的联合实现。

除此之外使用 Write-Though 策略可以极大降低 Cache Coherence 的实现难度，没有 Dirty 位使得多数设计更为流畅。但是你很难设想在一个 ccNUMA 处理器系统中，Cache 的所有层次结构都使用 Write-Though 策略。由此带来的各类 Bus Traffic 将何等壮观。

采用 Write-Though 策略最大的缺点是给其他 Cache 层次带来的 Write Traffic，而这恰恰是在一个 Cache Hierarchy 设计过程中，要努力避免的。为了降低这些 Write Traffic，几乎所有采用 Write-Though 策略的高端处理器都使用了 WCC(Write Coalescing Cache)或者其他类型的 Store-Though Queue，本节仅关注 WCC。

假设一个微架构的 L1 Cache 采用 Write-Through 方式。WCC 的作用是缓冲或者 Coalescing 来自 L1 Cache 的 Store 操作，以减少对 L2 Cache 的 Write Traffic。在使用 WCC 时，来自 L1 Cache 的 Store 操作将首先检查 WCC 中是否含有与其地址相同的 Entry，如果有则将数据与此 Entry 中的数据进行 Coalescing；如果没有 Store 结果将存入 WCC 的空闲 Entry 中；如果 WCC 中没有空闲 Entry 则进行 Write Through 操作。WCC 的大小决定了 Write Traffic 减少的程度。WCC 的引入也带来了一系列 Memory Consistency 问题。

采用 Write-Back 策略增加了 Cache Coherency 设计难度，也有效降低了 Write Traffic，避免了一系列 Write-Through 所带来的问题。下文以 GEM5 中的 MOESI_CMP_directory 为例简要 Write-Back 策略的实现方法。

如果 Write Hit 的 Cache Block 处于 Exclusive[①]状态时，数据可以直接写入，并通知 CPU Core Write 操作完成。Write Hit 的 Cache Block 处于 Shared 或者 Owner 状态时的处理较为复杂。即便是采用 Write-Through 策略，这种情况的处理依然较为复杂，只是 Cache 间的状态转换依然简单很多。本节仅介绍命中了 Shared 状态这种情况。

在 L1 Cache 层面，这个 Write Hit 命中的 Cache Block，其状态将会从 S 状态迁移到 MM_W 状态，之后经过一段延时迁移到 MM 状态。在 MOESI_CMP_directory Protocol 中，如果 L2 Cache Block 的状态为 ILS，ILO，ILOS，SLS，OLS 和 OLSX 时，L1 Cache 中必定含有对应的状态为 S 的 Block。由于 Accidentally Inclusive 的原因，L1 Cache Block 多数时候在 L2 Cache 中具有副本，因此必须要考虑 L1 Cache Block 进行状态迁移时对 L2 Cache Controller 的影响。不仅如此还需要考虑 CMP 间 Cache Coherency 使用的 Directory。

我们首先考虑 L1 Cache Block 的从 S 开始的状态迁移过程。当 CPU Core 的 Cache Hit 到某个状态位 S 的 L1 Cache Block 后便开始了一次长途旅行。在这个 Clock Block 的 S 状态将经由 SM，OM，最终到达 MM_W 和 MM 状态，如图 4-15 所示。

```
transition(S, Store, SM) {
    i_allocateTBE;
    b_issueGETX;
    // uu_profileMiss;
    k_popMandatoryQueue;
}

transition(SM, Ack) {
    m_decrementNumberOfMessages;
    o_checkForCompletion;
    n_popResponseQueue;
}

transition(SM, {Data, Exclusive_Data}, OM) {
        // v_writeDataToCacheVerify;
        m_decrementNumberOfMessages;
        o_checkForCompletion;
        n_popResponseQueue;
}
```

图 4-15 S 状态转移到 MM_W 状态的源代码

CPU Core 进行 Store 操作，Cache Hit 一个状态为 S 的 Block 时，首先迁移到 SM 状态，之后在收到 Data 或者 Exclusive Data 之后进入到 OM 状态，OM 状态到 MM_W 状态的迁移过程与图 4-14 的过程类似。本节重点关注 b_issueGETX 请求，这个请求被称为 Read for Exclusive。

在一个 ccNUMA 处理器系统中，b_issueGETX 请求虽然复杂依然有机可循，首先在 Intra-CMP 中尝试并获得对访问 Cache Block 的 Exclusive 权限。如果没有获得 Exclusive 权限，则将这个请求转发给 Directory Controller，并由 Home Agent/Node 经由 CMP 间的互联网络发送给其他 CMP 直到获得 Exclusive 权限。

[①] M，M_W，MM 和 MM_W 状态都属于 Exclusive 状态。

在 MOESI_CMP_directory Protocol 中，Cache Block 的状态为 S 表示在当前 ccNUMA 处理器系统中至少还存在一个数据副本，因此 b_issueGETX 请求将首先在 Intra-CMP 中进行，并通过 requestIntraChipL1Network_out 发送至 L2 Cache Controller。

L2 Cache Controller 通过 L1requestNetwork_in 获得 b_issueGETX 请求，如果在 L2 Cache 中包含访问的数据副本，b_issueGETX 请求将转换为 L1_GETX 请求。在 L2 Cache 中包含 L1 Cache Block 为 S 状态数据副本的情况有很多，还有一些 L2 Cache Block 包含数据副本，但是在当前 CMP 中其他 L1 Cache 中也包含数据副本的情况。这些状态转移的复杂程度如果我能够用语言简单描述，就不会有 SLICC 这种专用语言的存在价值。

如果在 Intra-CMP 中可以处理 b_issueGETX 请求，并不算太复杂，如果不能，而且 L2 Cache Block 的状态为 SLS 时，需要进一步做 Inter-CMP Coherence 的处理，此时 b_issueGETX 请求将转换为 a_issueGETX，并通过 globalRequestNetwork_out 继续发送至 Directory Controller。此时 Directory Controller 通过 requestQueue_in 获得该请求，并将其转换为 GETX 请求。Directory 中具有 4 个 Stable 状态 M，O，S 和 I，不同的状态对于 GETX 请求的处理不尽相同。

如果处于 O 状态，此时 Intra-CMP 为该 Cache Block 的 Owner，此时仅需要向其他 Inter-CMP 发送 g_sendInvalidations 请求，并可将 Exclusive_Data 转发至上层，L1 Cache Block 的状态也因此迁移为 OM。

如果在 Directory 中没有记录哪些 CMP 中具有状态为 S 的数据拷贝时，g_sendInvalidations 请求将广播到所有参与 Cache Coherence 的 CMP 处理器中，这种方式带来的 Bus Traffic 非常严重，也是这个原因在 Directory 中多设置了 Bit Vectors，用来记录哪些 CMP 中具有状态为 S 的数据副本。

在这种情况下 g_sendInvalidations 请求仅需发送到指定的 CMP，而不需要进行广播。因为 Memory Consistence 的原因，发起请求的 CMP 必须要等待这些指定的 CMP 返回所有的 ACK 后，才能进一步完成 CPU Core 的 Store 操作。除了在 Directory 中设置了 Bit Vectors 之外，Intel 的 MESIF 中的 F 状态也可以在某种程度上避免因为过多的 ACK 带来的 Bus Traffic。

通常情况在 CMP 间的连接拓扑不会使用 Share Bus 方式，对于 Request/ACK 这种数据请求模式，采用 Share Bus 方式最大的好处是具有一个天然的全局同步点，这个同步点在严重影响了总线带宽的前提下，极大了降低了设计难度。

采用 Share Bus 方式时，随着总线上节点数目的增加，冲突的概率也以 O(N!)级别的算法复杂度进一步增长。这使得一个可用的 CMP 间的互联方式极少采用 Share-Bus 方式。采用其他方式，无论是 Ring-Bus 或者更加复杂的拓扑结构，都会涉及到多个数据通路，这使得从这些 CMP 返回的 ACK 并没有什么顺序，Home Agent/Node 不能接收到一个 ACK 就向上转发一次，而是收集后统一转发。

当 L1 Cache Block 收到 All_acks 后，将从 OM 状态转移为 MM_W 状态，完成最后的操作，其过程与图 4-14 所示相同。为简略起见，本节不再介绍在 Directory 中是 M，S 和 I 状态的情况。而专注与 Write Miss 的处理。

Norman P. Jouppi 从两个方面讨论 Write Miss 的处理策略。首先是否为每一个 Miss 的请求重新准备一个 Cache Block，因此产生了两种方法，Write-Allocate 或者 No-Write Allocate。如果使用 Write-Allocate 方法，Cache Controller 为 Miss 请求在当前 Cache 中分配一个新的 Cache Block，否则不进行分配；

其次是否需要从底层 Cache 中获取数据，因此产生了两种处理方法，Fetch-on-Write 或者 No-Fetch-on-Write 方法，如果采用 Fetch-on-Write 方法，Cache Controller 将从其下 Cache 层次结构中 Fetch 已经写入的数据，并进行 Merge 操作后统一的进行写操作，否则不进行 Fetch。还有一种方法是 Write-Before-Hit，这种实现方法多出现在 Direct Mapped Write Through Cache 中，本节对此不再关注。

Write-Allocate 与 Fetch-on-Write 方法没有必然联系，但是这两种方法经常被混淆，以至于后来没有更正的必要与余地，通常意义上微架构提到的 Write-Allocate 策略是 Write-Allocate 与 Fetch-on-Write 的组合；实际上 Write-Allocate 也可以与 No-Fetch-on-Write 混合使用，该方法也被称为 Write-Validate，这种方法很少使用；No-Write Allocate 与 No-Fetch-on-Write 的组合被称为 Write-Around，如表 4-5 所示。

表 4-5 Write-Allocate 与 Fetch-on-Write 的组合[①]

	Fetch-on-Write	No-Fetch-on-Write
Write-Allocate	Write-Allocate	Write-Validate
No-Write Allocate	N.A.	Write-Around

我们假设一个处理器系统的 Memory Hierarchy 包含 L1，L2 Cache 和主存储器，而 Cache Miss 发生在 L1 Cache。并在这种场景下，分析 Write-Allocate，Write-Validate 和 Write-Around 这三种方法的实现。

当 Write L1 Cache Miss 而且使用 Write-Allocate 方法时，L1 Cache Controller 将分配一个新的 Cache Block，之后与 Fetch 的数据进行合并，然后写入到 L1 Cache Block 中。这种方法较为通用，但是带来了比较大的 Bus Traffic。

新分配一个 Cache Block 往往意味着 Replace 一个旧的 Cache Block，如果这个 Cache Block 中含有 Dirty 数据，Cache Controller 并不能将其 Silent Eviction，而是需要进行 Write-Back；其次 Fetch 操作本身也会带来不小的 Bus Traffic。

Write-Around 策略是 No-Write Allocate 和 No-Fetch-on-Write 的策略组合。使用这种方法时，数据将写入到 L2 Cache 或者主存储器，并不会 Touch L1 Cache。当 Cache Miss 时，这种方法并不会影响 Memory Consistency。但是这种方法在 Cache Hit 时，通常也会 Around 到下一级缓冲，这将对 Memory Consistency 带来深远的影响。

虽然我能构造出很多策略解决这些问题，但是这些方法都不容易实现。读者可以很自然的想到一种方法，就是在 Cache Hit 时不进行这个 Around 操作，对此有兴趣的读者可以进一步构想在这种情况之下，如何在保证 Memory Consistency 的情况较为完善的处理 Cache Hit 和 Miss 两种情况。本节对此不再进一步说明。

Write-Validate 策略是 No-Fetch-on-Write 和 Write-Allocate 的策略组合。使用这种方法时，L1 Cache Controller 首先将将分配一个新的 Cache Block，但是并不会向 Fetch 其下的 Memory Hierarchy 中的数据。来自 CPU Core 的数据将直接写入新分配的 L1 Cache Block 中，使用这种方法带来的 Bus Traffic 非常小。

但是来自 CPU Core 的数据不会是 Cache Block 对界操作，可能是 Byte，Word 或者是 DWord，L1 Cache 必须要根据访问粒度设置使能位，这个使能位可以是 By Byte，Word 或者是 Dword，也因此带来的较大的 Overhead。当进行 Cache Write 操作时，除了要写入的数据的使能位置为有效外，其他所有位都将置为无效。在使用这种方法时，有效数据可能分别存在与 L1 Cache 和 L2 Cache，这为 Memory Consistence 的实现带来了不小的困难。

No-Write Allocate 和 Fetch-on-Write 的策略组合没有实际用途。从其下 Memory Hierarchy Fetch 而来的数据因为没有存放位置，在与 CPU Core 的数据进行合并后，依然需要发送到其下的 Memory Hierarchy。几乎没有什么设计会进行这种不必要的两次总线操作。

在 Write Miss 的处理方法中，Write-Allocate 和 Write-Around 策略较为常用，Write-Validate 策略并不常用。Write Miss 的处理方法与 Write Hit 存在一定的依赖关系。Write-Around 需要与 Write-Through 策略混合使用，而 Write-Allocate 适用于 Write-Through 和 Write-Back 策略。

[①] 表 4-5 源自[96]，并有所改动。

在一个 ccNUMA 处理器系统中，与 Write 操作相关的处理更为复杂。本节介绍的 Hit Miss 实现策略各有其优点，所有这些策略所重点考虑的依然是如何降低因为 Write 而带来的 Bus Traffic 和 Memory Consistency。

4.7 Case Study on Sandy Bridge Cache Load

这一节是我准备最后书写的内容，在此之前最后一章的书写早已完成。待到结束，总在回想动笔时的艰辛。这些艰辛使我选择一个 Case Study 作为结尾，因为这样做最为容易。这些 Case 实际存在的，不以你的喜好而改变。缺点与优点都在你面前，你无需改变，只需要简单的去按照事实去陈述。

Sandy Bridge 是 Nehalem 微架构之后的 Tock，并在 Nehalem 的基础上作出的较大的改动。本节重点关注 Cache Hierarchy 上的改动。鉴于篇幅，鉴于没有太多公开资料，我并不能在这里展现 Sandy Bridge 微架构的全貌，即便只限于 Cache Hierarchy 层面。这一遗憾给予我最大的帮助是可以迅速完成本节。

Intel 并没有公开 Sandy Bridge 的细节，没有太多可供检索的参考资料。David Kanter 在 Realworldtech 上发表的文章[69][98]较为详尽，但这并不是来自官方，鉴于没有太多公开资料，本篇仍然使用了 David Kanter 的文章。虽然我知道真正可作为检索的资料是 Intel 发布的 Intel 64 and IA-32 Architectures Optimization Reference Manual 中的第 2.1.1 节[97]和 Intel 在 2010 年 IDF 上公开的视频 http://www.intel.com/idf/audio_sessions.htm[1]。

Sandy Bridge 包含两层含义。首先是 Sandy Bridge 微架构，即 Core 部分，由指令流水，L1 Cache 和 L2 Cache 组成，如图 4-16 所示。其中指令流水的 Scheduler 部件，和 Load，Store 部件需要重点关注，最值得关注的是 L1 和 L2 Cache 的组成结构。

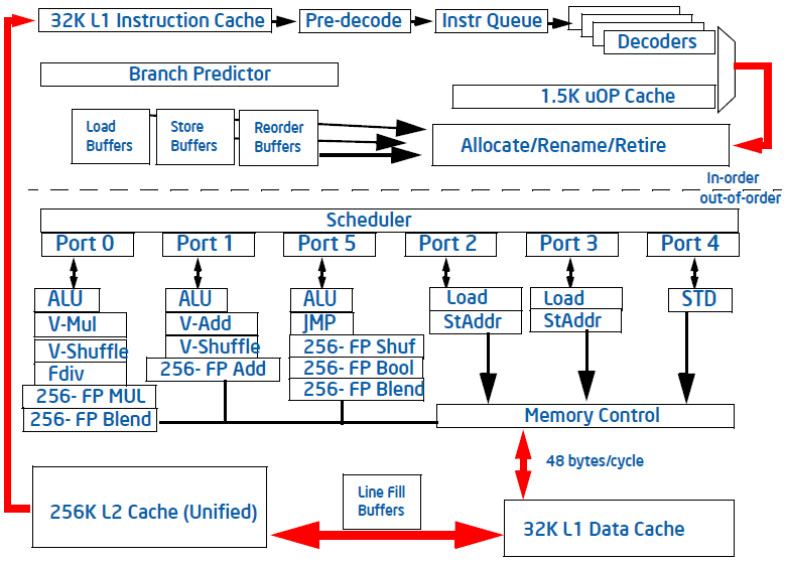

图 4-16 Sandy Bridge 微架构示意图[97]

从 CPU Core 的角度上看，Sandy Bridge 与 Nehalem 相比，并没有太多质的变化。最值得关注的是 Sandy Bridge 增加的 L0 Instruction Cache 和 PRF(Physical Register File)。L0 Instruction Cache 也被称为 Decoded μops Cache，这是 Sandy Bridge 在指令流水中相对于 Nehalem 微架构的重大改进。PRF 替换了 Nehalem 微架构使用的 CRRF(Centralized Retirement Register File)。PRF 不是什么新技术，只是 Intel 实现的晚了些。

在 Core 和 Nehalem 微架构中，每一个 μops 包含 Opcode 和 Operand。这些 μops 在经过指令流水执行时需要经过若干 Buffer，有些 Buffer 虽然只需要 Opcode，但是也必须要同时容纳 Operand，因而带来了不必要的硬件开销。在 Core 微架构时代，Operand 最大为 80b，Nehalem 为 128b，到了 Sandy Bridge 微架构，Operand 最大为 256b。

如果 Sandy Bridge 不使用 PRF，支持 AVX(Advanced Vector Extension)的代价会变得无法承受，因为有些 AVX 指令的 Operand 过长。AVX 的出现不仅影响了指令流水线的设计，也同时影响了 Sandy Bridge 的 Memory 子系统的设计。我们首先关注指令执行部件中的 Memory Cluster，Memory Cluster 即为 LSU，其结构如图 4-17 所示。

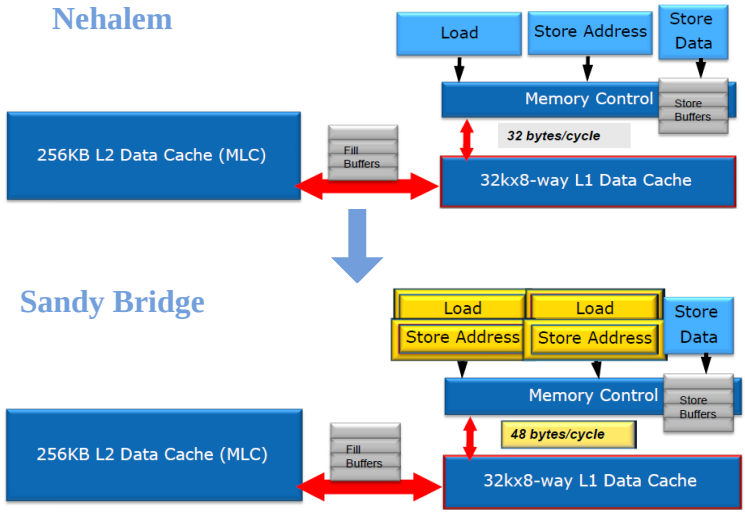

图 4-17 Nehalem 和 Sandy Bridge 的 LSU[1]

与 Nehalem 微架构在一个 Cycle 中只能执行一条 128b 的 Load 和 Store 指令[1][12]相比，Sandy Bridge 微架构在一个 Cycle 中可以执行一条 128b 的 Load 和一条 128b 的 Store 指令，或者两条 Load 指令，进一步提高了 Load 指令的执行效率，在微架构设计中，通常会优先考虑 Load 指令，而不是 Store 指令。如果将 Store 指令提高为两条，其中因为 Memory Consistency 引发的同步并不容易处理。也因为这个原因，Sandy Bridge 设置了两条 Load 通路，LSU 与 L1 Data Cache 间的总线宽度也从 Nehalem 微架构的 2×128b 提高到 3×128b。

合并 Load 和 Store Address 部件在情理之中，因为 Load 操作和 Store Probe 操作有相近之处，在现代处理器中，Store 操作的第一步通常是 Read for Ownership/Exclusive，首先需要读取数据后，再做进一步的处理。

在 Sandy Bridge 中，FLC 和 MLC 的组成结构与 Nehalem 微架构类同。最大的改动显而易见，是在 L1 Cache 之上多加了一个读端口。单凭这一句话就够工程师忙碌很长时间。在 Cache Memory 层面任何一个小的改动，对于工程师都是一场噩梦。

其中 FLC 由指令 Cache 与数据 Cache 组成，由两个 Thread 共享；MLC 为微架构内部的私有 Cache。L1 指令和数据 Cache 的大小均为 32KB，MLC 的大小为 256KB。FLC 和 MLC 的关系为 NI/NE，组成结构为 8-Way Set-Associative，Cache Block 为 64B，MPMB，Non-Blocking，Write-Allocate，Write Back 和 Write-Invalidate。Cache Coherence Protocol 为 MESI。

这些仅是 Sandy Bridge 微架构，即 Core 层面的内容。Sandy Bridge 的另一层含义是 Sandy Bridge 处理器。Sandy Bridge 处理器以 Sandy Bridge 微架构为基础，包括用于笔记本和台式机的 Sandy Bridge 处理器，和 Server 使用的 Sandy Bridge EP 处理器。但是 Sandy Bridge 和 Sandy Bridge EP 在 Uncore 部分的设计略有不同，本节重点讲述 Sandy Bridge EP 处理器。

Sandy Bridge EP 处理器由 CPU Core，iMC 控制器(Home Agent)[97]，Cache Box[1][98]，PCIe Agent，QPI Agent 和 LLC(L3 Cache)组成，由 Ring Bus(Ring-Based 的 Interconnect)连接在一起，并在其内部集成 Graphics Controller，其组成结构如图 4-18 所示。

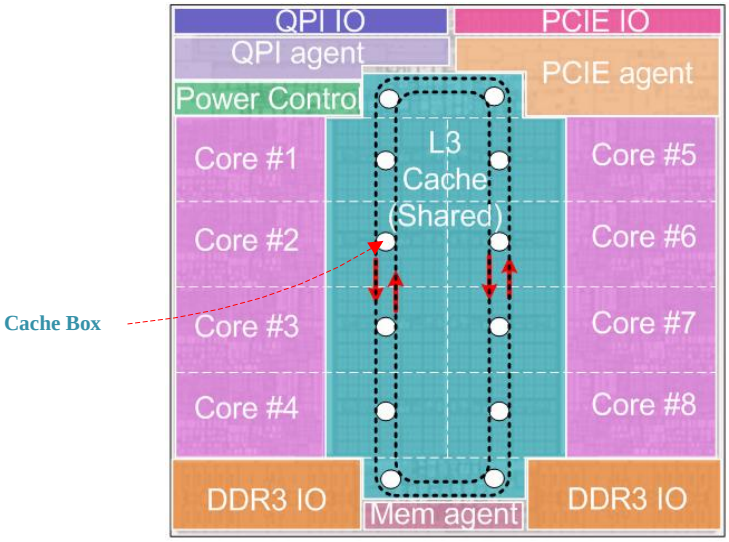

图 4-18 Sandy Bridge EP 处理器的组成结构[1][98]

其中 Cache Box 是 Core 与 Uncore 部分的连接纽带。如图 4-18 所示，Cache Box 提供了三个接口，与 CPU Core，LLC 和 Ring Bus 的接口。Cache Box 的主要功能是维护 Sandy Bridge EP 处理器中 CPU Core 与 Core 间的 Memory Consistency，并将来自 CPU Core 的数据请求发送到合适的 LLC Slice 或者其他设备中[1]。

Sandy Bridge EP 处理器的 LLC 采用 Distributed 方式，每一个 CPU Core 都有一个对应的 LLC Slice，每个 Slice 的大小可以是 0.5/1/1.5/2MB，可以使用 4/8/12/16-Way Associated 方式。这不意味着每一个 CPU Core 都有一个私有 Slice。

来自 CPU Core 的数据访问在经过 Cache Box 时，首先进行 Hush，并通过 Ring Bus 转发到合适的 Cache Slice。但是从逻辑层面上看这些 Slice 组成一个 LLC。Sandy Bridge EP 处理器的 LLC 与 Core 内 Cache 的关系是 Inclusive，与 Nehalem 的 L3 Cache 类同。这意味着空间的浪费，也意味着天然的 Snoop Filter[98]。

所有 CPU Core，LLC Slice，QPI Agent，iMC，PCIe Agent，GT(Graphics uniT)通过 Ring Bus 连接在一起[1][98]。Ring Bus 是 Sandy Bridge EP 处理器的设计核心，也意味着 GT 可以方便的与 CPU Core 进行 Cache Coherence 操作。这在一定程度上决定了 Sandy Bridge 处理器横空出世后，基于 PCIe 总线的 Nvidia Graphics Unit 黯然离场。

Sandy Bridge EP 处理器的 Ring Bus，采用 Fully Pipelined 方式实现，其工作频率与 CPU Core 相同，并由四个 Sub-Ring Bus 组成，分别是 Data Ring，Request Ring，Acknowledge Ring 和 Snoop Ring[1]，其中 Data Ring 的数据宽度为 256 位。这些 Sub-Ring Bus 协调工作，共同完成 Ring Bus 上的各类总线 Transaction，如 Request，Data，Snoop 和 Response。采用 4 个 Sub-Ring Bus 可以在最大程度上使不同种类的 Transaction 并发执行。

Sandy Bridge EP 处理器的这些 Sub-Ring Bus 谈不上是什么创新，所有使用了 Ring Bus 结构的现代处理器都需要这么做。由于 Dual Ring 的存在，Sub-Ring 中通常含有两条总线，可能只有 Snoop Ring 除外，所以在 Sandy Bridge 的 Ring Bus 至少由 7 条 Bus 组成[①]。

[①] 这些说法仅是猜测。Snoop Ring 有两条总线，至少我现在想不出什么简单的方法确保 Memory Consistency。

在 Ring Bus 上，还有两个重要的 Agent，一个是 Memory Agent，另一个是 QPI Agent。其中 Memory Agent 用来管理主存储器，包括 iMC，而 QPI Agent 用于管理 QPI 链路，并进行与其他 Sandy Bridge EP 处理器互联，组成较为复杂的 ccNUMA 处理器系统。

以上是对 Sandy Bridge EP 处理器与 Memory Hierarchy 结构的简单介绍，下文将以此为基础进一步说明 Sandy Bridge EP 处理器如何进行 Load 操作。

剩余的内容需要等待 Intel 公开 Sandy Bridge EP 使用的 Transaction Flow，估计会在 Sandy Bridge EP 正式发布时公开。Sandy Bridge EP 的正式发布推迟到了 2012 年 Q1，那时我会重新书写本节。整篇文章需要更改的地方还有很多。

第5章 Data Prefetch

处理器与存储器子系统运行速度的失配，使得存储器层次结构多次引起关注，处理器系统使用了更大规模的 Cache。在很多处理器系统中，LLC 的大小已达十几兆字节。随着工艺的提高，使用更大规模的 Cache 容量，并非遥不可及。只是 Cache 容量依然远不能与主存储器容量增加的速度相比。在某些应用中，即便将现有的 Cache 容量提高一倍也于事无补。

存储器访问在最后一级 Cache 中 Miss 后，指令流水可能会被迫 Stall，有些执行部件甚至要为此等待几百个 Cycle，极大降低了处理器的整体运行效率。在这种情况之下，使用再精巧的指令流水线设计也无能为力。

这一切使得更多的人重新考虑存储器子系统的延时处理。各种想法层出不穷，如更加充分利用 Non-Blocking Cache 流水线，容纳上千条指令的 OOO 指令流水，Runahead 执行，Prefetch 等等。这些想法并非天方夜谭，具有理论基础与量化数据作为支撑。这些想法不是绝对的真理，可能只是 Trade-Off。在这些想法中，目前使用最多的，最为成功的是 Prefetch。

5.1 数据预读

Prefetch 指在处理器进行运算时，提前通知存储器子系统将运算所需要的数据准备好，当处理器需要这些数据时，可以直接从这些预读缓冲中，通常指 Cache，获得这些数据，不必再次读取存储器，从而实现了存储器访问与运算并行，隐藏了存储器的访问延时。Prefetch 的实现可以采用两种方式，HB(Hardware-Based)和 SD(Software-Directed)。这两种方法各有利弊，我们首先以图 5-1 为基础模型讨论采用 SD 方式的数据预读。

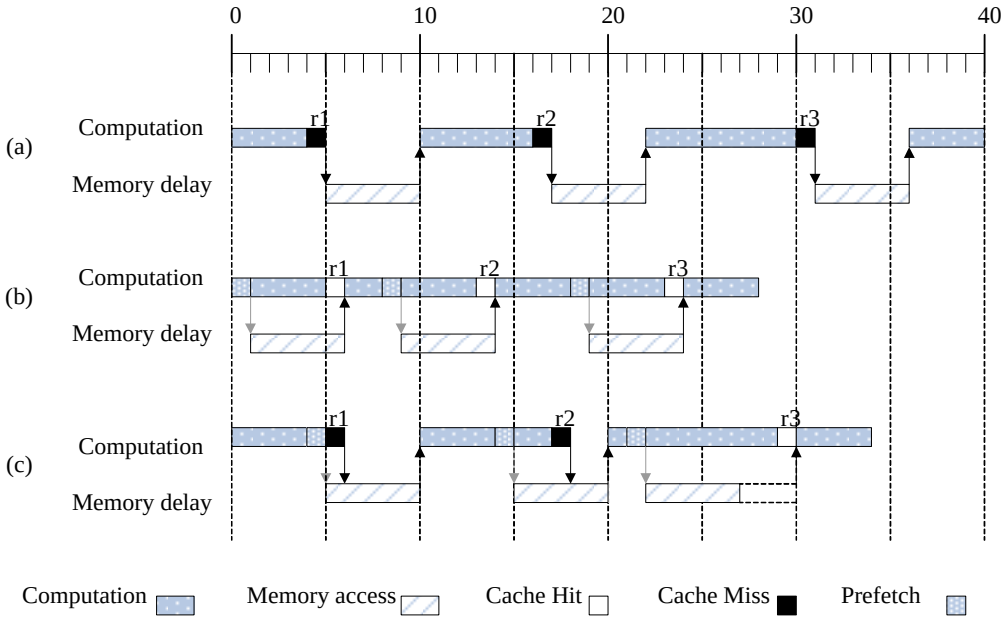

图 5-1 数据预读机制示意[99]

其中实例 a 没有使用预读机制；实例 b 是一个采用预读机制的理想情况；实例 c 是一个采用预读机制的次理想情况。我们假设处理器执行图 5-1 所示的任务需要经历四个阶段，每个阶段都由处理器执行运算指令和存储指令组成。

在其中处理器的一次存储器访问需要 5 个时钟周期。在第一个阶段处理器执行 4 个时钟周期后需要访问存储器；在第二个阶段处理器执行 6 个时钟周期后需要访问存储器；在第三个阶段处理器执行 8 个时钟周期后需要访问存储器；在第四个阶段处理器执行 4 个时钟周期后完成整个任务。

实例 a 没有使用预读机制，在运算过程中，在进行存储器访问，将不可避免的出现 Cache Miss。执行上述任务共需 40 个时钟周期。使用预读机制可以有效缩短整个执行过程。在实例 b 中在执行过程中，会提前进行预读操作，虽然这些预读操作也会占用一个时钟周期，但是这些预读操作是值得的。合理使用这些数据预读，完成同样的任务 CPU 仅需要 28 个时钟周期，从而极大提高了程序的执行效率。

这种情况是非常理想的，处理器在执行整个任务时，从始至终是连贯的，处理器执行和存储器访问完全并行，然而这种理想情况并不多见。在一个任务的执行过程中，并不容易确定最佳的预读时机；其次采用预读所获得数据并不一定能够被及时利用，因为在程序执行过程中可能会出现各种各样的分支选择，有时预读的数据并没有被及时使用。

在实例 c 中，预读机制没有完全发挥作用，所以处理器在执行任务时，Cache Miss 仍会发生，减低了整个任务的执行效率。即便这样，实例 c 也比完全没有使用预读的实例 a 的任务执行效率还是要高一些。在实例 c 中，执行完毕该任务共需要 34 个时钟周期。当然我们还可以轻松出采用预读使图 5-1 中的实例执行的更加缓慢。

图 5-1 中的实例可以使用硬件预读的方式。但是无论采用什么方式，都需要注意预读的数据需要及时有效，而且在产生尽可能小的 Overhead 的基础上供微架构使用。在实例 c 的 r1 和 r2 中，预读操作过晚，因此指令流水依然会 Stall，从而影响执行效率。

在 r3 中，预读操作过早，虽然数据可以提前进入某个 Cache Block，但是这意味着过早预读的数据可能会将某个将要使用的 Cache Block 替换出去，因此 CPU Core 可能会重新读取这个被替换出去的 Cache Block，从而造成了 Cache Pollution。除此之外每一个 Cache Block 有自己的 MLS，过早预读的数据，有可能被其他存储器访问替换出去，当 CPU Core 需要使用时，该数据无法在 Cache 中命中。

因此在进行数据预读时，需要首先重点关注时机，不能过早也不能过晚。如果考虑多处理器系统，无论是采用 HB 或者 SD 方式，做到恰到好处都是巨大的挑战。除了预读时机之外，需要进一步考虑，预读的数据放置到 Cache Hierarchy 的哪一级，L1，L2 还是 LLC，所预读的数据是私有数据还是共享数据。需要进一步考虑预读数据的 Granularity，是 By Word, Byte，Cache Block，还是多个 Cache Block；需要进一步考虑是否采用 HB 和 SD 的混合方式。这一切增加了 Prefetch 的实现难度。

这也造成了在某些情况下，采用预读机制反而会降低效率。什么时候采用预读机制，关系到处理器系统结构的各个环节，需要结合软硬件资源统筹考虑，并不能一概而论。处理器提供了必备的软件和硬件资源以实现预读，如何“合理”使用预读机制是系统程序员考虑的一个细节问题。数据预读可以使用软件预读或者硬件预读两种方式实现，下文将详细介绍这两种实现方式。

软件和硬件预读策略所追求的指标依然是 Coverage，Accuracy 和 Timeliness[100]。Coverage 指 CPU Core 需要的数据有多少是从 Prefetcher 中获得，而不是访问存储器子系统；Accuracy 指 Prefetched Cache 中有多少数据是 CPU Core 真正需要的；Timeliness 指预读的数据是否能够恰到好处的到达，不能太早也不能太晚。

在已知的软件或者硬件预读策略主要针对以上三个参数展开，这些策略的底线是预读所使用的开销不大于于不使用预读机制时 Cache Miss 的开销。在许多情况下，采用预读策略不仅不会提高程序的执行效率，甚至会极大影响程序的正常执行，带来严重的系统惩罚，最终结果不如放弃这些预读机制。

我们需要对预读算法进行定性分析。假设 Prefetch Ratio 参数指由于 Prefetch 而读取的 Cache Block 总数在所有存储器访问的 Cache Block 中所占的比率；Transfer Ratio 指 Prefetch Ratio 和 Miss Ratio 之和。

Access Ratio 指所有 Cache 的访问次数与 Prefetch Lookup 之间的比值。所有 Cache 的访问次数是 Actual 和 Prefetch Lookup 之和。其中 Prefetch Lookup 指由 Prefech 算法决定当前 Cache Block 是否应该替换，是否应该 Prefetch 新的 Cache Block 而引发的 Cache 访问，是由 Cache Controller 主动发起的 Cache 访问操作；Actual Lookup 指 Cache Controller 之外的访问操作，如 CPU Core 或者外部设备对 Cache 的访问操作。Access Ratio 的值大于 1。

在此基础之上，我们进一步引入参数 D，P 和 A。其中参数 D 为 Demand Miss 所带来的 Penalty，Demand Miss 指没有采用预读而产生的 Cache Miss 开销；参数 P 为预读的代价，包括数据读入，因为读入新的数据而 Replacement 旧的数据，等各类因为预读导致的数据传递的开销；参数 A 为因为预读干扰了程序对 Cache 正常使用而带来的惩罚。在这种情况下，一个有效的预读算法需要满足公式 5-1。

$$\mathrm{D} \times \mathrm{Miss\ Ratio(Demand)} > \mathrm{D} \times \mathrm{Miss\ Ratio(Prefetch)} + \mathrm{P} \times \mathrm{Prefetch\ Ratio} + \mathrm{A} \times (\mathrm{Access\ Ratio} - 1) \quad \text{公式 5-1[23]}$$

如果出现 Miss Ratio(Prefetch)大于 Miss Ratio(Demand)的情况，即便 P，A，Prefetch Ratio 参数为 0，上述公式也无法成立。这种情况是使用预读机制所造成的最糟糕结果。此时预读造成 Cache Pollution，使得 Cache Miss Ratio 反而低于与没有使用预读的情况

硬件还是软件预读机制都会造成这种情况。与硬件预读相比，软件预读更加灵活一些。但是在很多情况之下，我并不喜欢使用编译器强行加入的预读处理，倾向根据微架构和应用的具体要求，书写这些预读代码。有时由编译器增加的预读代码除了进一步污染指令 Cache 之外，不会带来更多帮助。这不是否认编译器的努力，而是提醒读者需要因地制宜。

5.2 软件预读

软件预读机制由来已久，首先实现预读指令的处理器是 Motorola 的 88110 处理器，这颗处理器首先实现了 Touch Load 指令，这条指令是 PowerPC 处理器 dcbt 指令[4]的前身。后来绝大多数处理器都采用这类指令进行软件预读，Intel 在 i486 处理器中使用 Dummy Read 指令，这条指令也是后来 x86 处理器中 PREFETCHh[5]指令的雏形。

使用软件预读指令可以在处理器真正需要数据之前，向存储器预先发出读请求，这个预读请求不需要等待数据真正到达存储器之后，就可以执行完毕，以实现存储器访问与处理器运算同步进行，从而提高了任务的整体执行效率。

除了专有指令外，普通的读指令也可以用作预读，如 Non-Blocking 的 Load 指令。这个读指令与 Prefetch 指令最大的区别是，这些指令不仅将数据引入 Cache 层次结构，而且会将结果写入某个寄存器，这类指令也被称为 Binding Prefetch。与此对应，在微架构中专门设置的 Prefetch 指令被称为 Non-Binding Prefetch 指令。

Prefetch 指令需要采用 Non-Blocking，Non-Exception-Generating 方式实现。Non-Blocking 较易理解，因为在一个使用 Blocking Cache 的微架构中，没有使用 Prefetch 指令的任何必要。在微架构中，一个简单实现 Prefetch 指令的做法是借用 Non-Blocking load 指令，并将结果传递给 Nobody 寄存器，较为复杂的实现是预读数据的同时，引入一些 Hint，如微架构将如何使用预读的数据，是写还是读，这些信息有助于多核处理器的一致性处理。

Non-Exception-Generating 指在 Prefetch 时不得引发 Exception，包括 Page Fault 和其他各类的 Memory Exception。在一些微架构中如果 Prefech 引发了 Exception，获得的数据将被丢弃。此外 Exception 还会带来较大的 Overhead，对 Memory Consistency 的实现制造障碍。

软件预读指令可以由编译器自动加入，但是在很多场景，更加有效的方式是由程序员主动加入预读指令。这些预读指令在进行大规模向量运算时，可以发挥巨大的作用。在这一场景中，通常含有大规模的有规律的 Loop Iteration。这类程序通常需要访问处理较大规模的数据，从而在一定程度上破坏了程序的 Temporal Locality 和 Spatial Locality，这使得数据预读成为提高系统效率的有效手段。我们考虑图 5-2 中的实例。

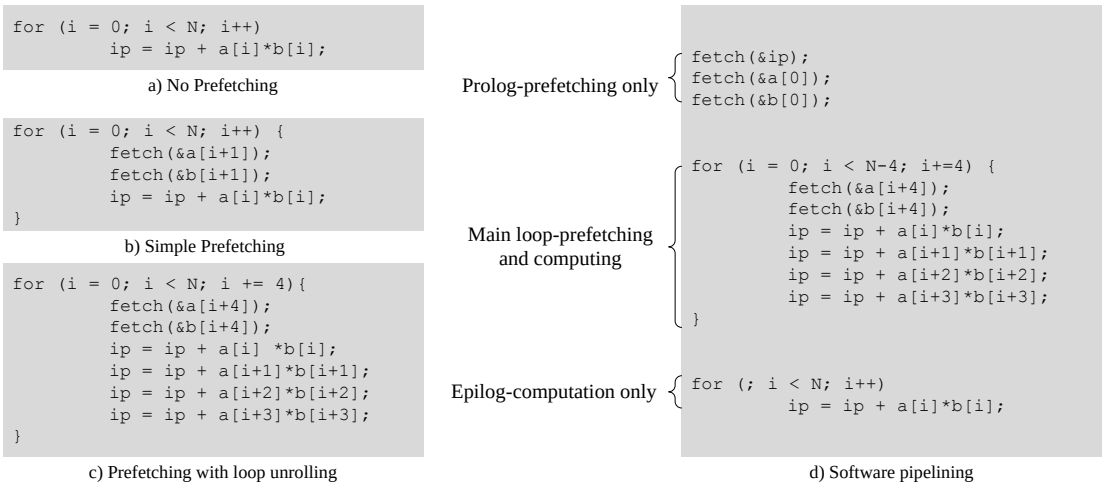

图 5-2 利用软件预读指令进行程序优化[99]

这个例子在进行向量运算时被经常使用，这段源代码的作用是将 int 类型的数组 a 和数组 b 的每一项进行相乘，然后赋值给 ip，其中数组 a 和 b 的基地址 Cache Block 对界。我们假设 N 为一个较大的常数且能够被 4 整除，此外微架构的 Cache Block 为 32 字节，并在此基础上考虑图 5-2 中的几个实例。

在实例 a 中没有使用预读机制进行优化。这段程序在执行时，a[i]和 b[i]中的数据不会在处理器的 Cache 中命中，而且在顺序访问向量 a 和 b 的数据单元时，每次跨越 Cache Block 都会因为 Compulsory Misses 向存储器子系统发送读请求，从而 stall 微架构的指令流水，降低了程序的执行效率。

实例 b 在对变量 ip 赋值之前，首先对数组 a 和 b 进行预读，当对变量 ip 赋值时，数组 a 和 b 中的数据可能已经在 Cache 中，从而在一定程度上提高了代码的执行效率。这段代码并不完美。因为在绝大多数微架构中，预读以 Cache Block 为单位进行，对 a[0], a[1], a[2], a[3] 进行预读时都是对同一个 Cache Block 进行预读。因此这段代码对同一个 Cache Block 进行了多次预读，从而影响了执行效率。

实例 c 使用 Loop Unrolling 技术，将循环体内的赋值操作进一步展开为 4 个子步骤，从而避免了实例 b 中存在的多次预读。在现代处理器中，Branch Prediction 较为完善，此处出现的 Loop Unrolling 并不会降低循环转移的开销，其主要目的是提高 Cache Block 的利用率，以减少预取次数。

实例 d 是在 c 基础上的继续优化，借用流水线设计的思想，将一次计算，分解为 Prolog，Main loop 和 Epilog 三个阶段。其中 Prolog 是建立流水时的准备工作，Main Loop 是预读与计算的并行阶段，而 Epilog 是最后的结尾工作。

以上这些方法较为通用，有些编译器会自动将实例 a 转化为实例 d。但是这些优化方式仍然忽略了一个细节，由于存储器的访问延时，预读的数据可能不会在计算需要时及时达到，指令流水线依然会 Stall。为此预读指令需要进一步考虑存储器延时与计算所需时间之间的关系，保证预读的数据在计算需要时准时到达。

为此我们需要对 Prefetch Distance 参数做进一步分析，该参数简称为 δ，其计算公式为 δ = Ceiling(L/S) [99]。其中 L 为平均存储器访问延时，而 S 为一个 Loop Iteration 中计算部分使用的最短执行时间。

假设在实例 d 中，平均存储器访问延时为 100 个时钟周期，而一个 Loop Iteration 中的计算使用的最短执行时间为 45 个时钟周期时，δ 参数的值为 3。这一结果表明每次预读指令需要在 3 倍于 Loop Iteration 中的计算时间之前执行，才能保证软件流水可以顺利进行，不会因为预读的数据尚未到达而被迫等待。使用 Prefetch Distance 参数可以进一步优化实例 d，如图 5-3 所示。

Prolog-prefetching only

```
fetch( &ip);
for (i = 0; i < 12; i += 4){
        fetch(&a[i]);
        fetch(&b[i]);
}
```

Main loop-prefetching and computing

```
for (i = 0; i < N-12; i += 4){
        fetch(&a[i+12]);
        fetch(&b[i+12]);
        ip = ip + a[i] *b[i];
        ip = ip + a[i+1]*b[i+1];
        ip = ip + a[i+2]*b[i+2];
        ip = ip + a[i+3]*b[i+3];
}
```

Epilog-computation only

```
for ( ; i < N; i++)
        ip = ip + a[i]*b[i];
```

图 5-3 考虑 Prefetch Distance 后的程序优化[99]

这些优化并不是软件预读的终点，还有很多利用某些 Cache 深层次特性做进一步优化的可能。这些优化都是具有一定的针对性，需要对处理器体系结构有着较为深刻的理解。在很多情况下软件预读机制有较为明显的缺点，首先是 Code Expansion 的问题，软件预读优化增加了代码长度，在一定程度上容易造成 L1 Instruction Cache 的 Pollution，其次是预读指令本身的所带来的 Overhead。采用硬件预读机制可以有效避免这两种缺陷，这使得更多的人开始重新关注硬件预读机制。

5.3 硬件预读

采用硬件预读的优点是不需要软件进行干预，不会扩大代码的尺寸，不需要浪费一条预读指令来进行预读，而且可以利用任务实际运行时的信息(Run Time Information)进行预测，这些是硬件预读的优点。

硬件预读的缺点是预读结果有时并不准确，有时预读的数据并不是程序执行所需要的，比较容易出现 Cache Pollution 的问题。更重要的是，采用硬件预读机制需要使用较多的系统资源。在很多情况下，耗费的这些资源与取得的效果并不成比例。

硬件预读机制的历史比软件预读更为久远，在 IBM 370/168 处理器系统中就已经支持硬件预读机制。大多数硬件预读仅支持存储器到 Cache 的预读，并在程序执行过程中，利用数据的局部性原理进行硬件预读。

最为简单的硬件预读机制是 OBL(One Block Lookahead)机制，这种方式虽然简单，但是在许多情况下效率并不低于许多复杂的实现，也是许多处理器采用的方式。OBL 机制有许多具体的实现方式，如 Always Prefetch，Prefetch-on-Miss 和 Tagged Prefetch[23]。

在使用 Always Prefetch OBL 实现方式时，当一段程序访问数据块 b 时，只要数据块 b+1 没有在 Cache 中 Hit，就对数据块 b+1 进行预读。这种方式的缺点是可能程序访问数据块 b 之后，将很长时间不使用数据块 b+1，从而带来较为严重的 Cache Pollution。使用这种方式时的 Access Ratio 为 2。

在使用 Prefetch-on-Miss OBL 实现方式时，当程序对数据块 b 进行读取出现 Cache Miss 时，首先将数据块 b 从存储器更新到 Cache 中，同时预读数据块 b+1 至 Cache 中；如果数据块 b+1 已经在 Cache 中，将不进行预读。使用这种方式时的 Access Ratio 为 1+Miss Ratio。

Always Prefetch 和 Prefetch-on-Miss OBL 方式没有利用之前的历史信息，在某些应用中，容易造成 Cache Pollution。Tagged Prefetch 是 Prefetch-on-Miss 实现方式的一种改进，其实现相对较为复杂，也使用了额外的硬件资源。

在使用 Tagged Prefetch OBL 实现方式时，需要为每一个 Cache Block 设置一个 Tag 位，该位在复位或者当前 Cache Block 被替换时设置为 0。如果当前 Cache Block 是因为 Prefetch 的原因从其下的存储器子系统中获得时，该位依然保持为 0。

当前 Cache Block 在预读后第一次使用，或者是 Demand-Fetched 时，Tag 位将从 0 转换为 1，此时如果其后的数据块不在 Cache Block 时将进行预读[23]。这种方式与 Prefetch-on-Miss 的最大区别在于访问已经 Prefetch 到 Cache 中数据的处理。

当程序访问已经预读到 Cache 的 Block 时，在使用 Prefetch-on-Miss 方式时，不会继续预读下一个 Cache Block，而使用 Tagged Prefetch 方式时，会继续预读下一个 Cache Block，从而减少了 Demand-Fetched 的概率，其实现示意如图 5-4。

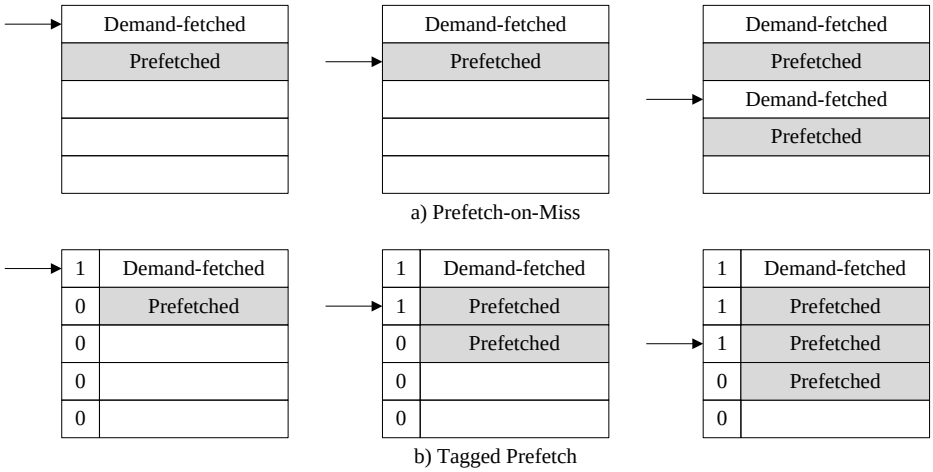

图 5-4 Prefetch-on-Miss 与 Tagged Prefetch 实现方式的比较[99]

从上图可以发现，对于一个顺序访问的 Access Patern，使用 Prefetch-on-Miss 方式，每次访问过一个 Prefetched Cache Block 后，都会出现一次 Cache Miss；而是用 Tagged Prefetch 时仅会出现一次 Cache Miss。

但是仅用这一种访问模型，并不能证明 Tagged Prefetch 一定由于 Prefetch-on-Miss 方式。Alan J. Smith [23]根据 Miss Ratio，Access Ratio 和 Transfer Ratio 三个参数对以上实现方式进行了较为细致的对比。从 Access Ratio 参数的上看，Always prefetch 实现方式大于后两种方式。

与 Prefetch-on-miss 方式相比，Tagged prefetch 实现方式在 Access Ratio 和 Transfer Ratio 没有明显提高的前提下，降低了 50%~90%的 Miss Ratio [23]。但是我们依然不能得出 Tagged prefetch 一定优于 Prefetch-on-miss 方式的结论。与其他方式相比，Tagged Prefetch 方式每一个 Cache Block 多使用了一个 Tag 位，依然是某种程度的 Trade-off。

Tagged Prefetch 实现有许多衍生机制，比如可以将数据块 b+1，b+2，…，b+k 预读到 Cache 中。其中 k 为 Prefetch 的深度，当 k 为 1 时，即为标准的 Tagged Prefetch。更有甚者提出了一种 Adaptive Sequential Prefetching 实现方式，此时 k 可以根据任务执行的 Run Time 信息进行调整，可以为正，也可以为负。

以上这些硬件预读算法都有其局限性，特别是在处理 Strided Array 相关的计算时，为此也产生了一系列可以利用 Stride 信息的硬件预读实现，如 Lookahead Data Prefetching 实现[101]。该实现的组成结构如图 5-5 所示。

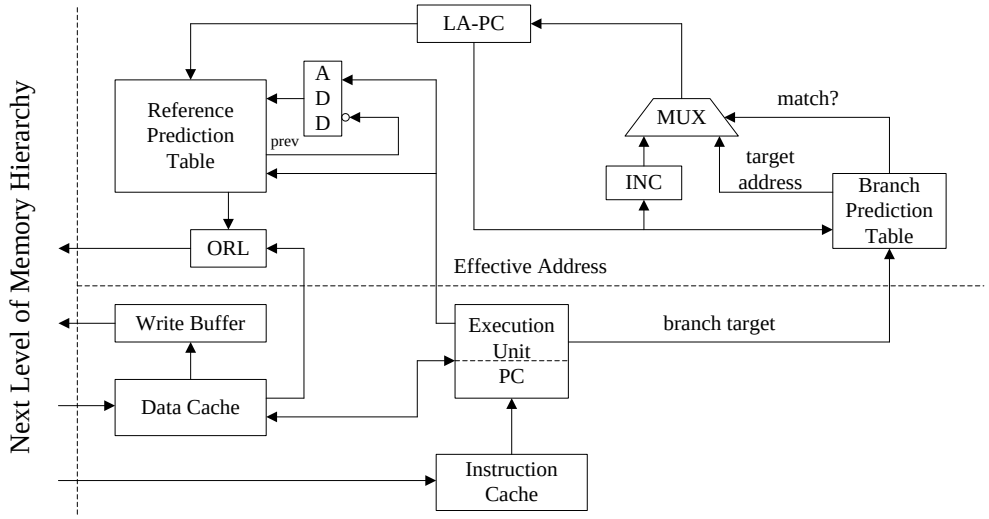

图 5-5 Lookahead data prefetching 实现的组成结构[103]

假设在一个 3-Nested Loop Iterations 中，某条存储器访问指令 m_i 需要陆续访问 a_1, a_2 和 a_3。当$(a_2 - a_1) = \Delta \neq 0$ 时，需要对 m_i 进行预读，Δ 参数即为预读的 Stride。第一次预读地址 $A_3 = a2 + \Delta$，其中 A_3 为预测值，如果预测与实际的 a_3 相同，则继续预测，直到 $A_n \neq a_n$。采用这种实现方法，需要使用历史地址信息和最后一次检测成功的 Δ 参数，为此在硬件上需要设置一个 RPT(Reference Prediction Table)，RPT 的组成结构与 Cache 类似，如图 5-6 所示。

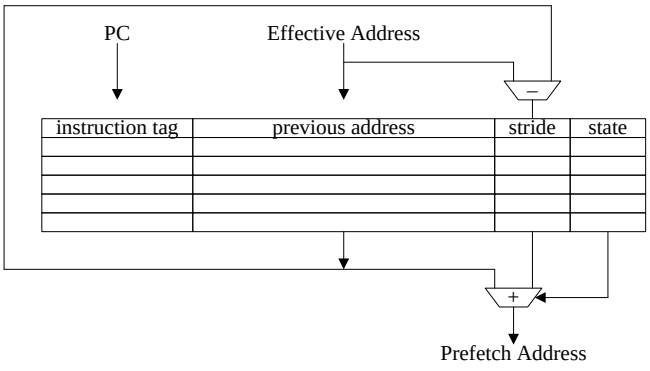

图 5-6 RPT 的组成结构

RPT 由微架构的 PC 进行索引。当指令 m_i 第一次执行时，将从 RPT 中分配一个空闲 Entry，填写相应的 Instruction Tag，Previous Address，并将 state 设置为 initial 状态。当指令 m_i 第二次执行时，并在 RPT 中命中时，将根据当前的 EA 与 Previous Address 计算 Atride 参数后填入当前 Entry，并将 State 设置为 Transient 状态。

此时如果地址(Effective Adderss+Stride)所指向的数据没有在 Cache 中命中，进行 Tantative Prefetch 操作。当指令 m_i 第三次执行时，在 RPT 中命中，而且 A_3 与实际的 a_3 相同时，表示发生了一次 Correct stride Prediction，此时继续进行下一个地址的预读，同时将 State 改写为 Steady。在 RPT 中，State 的状态迁移如图 5-7 所示。

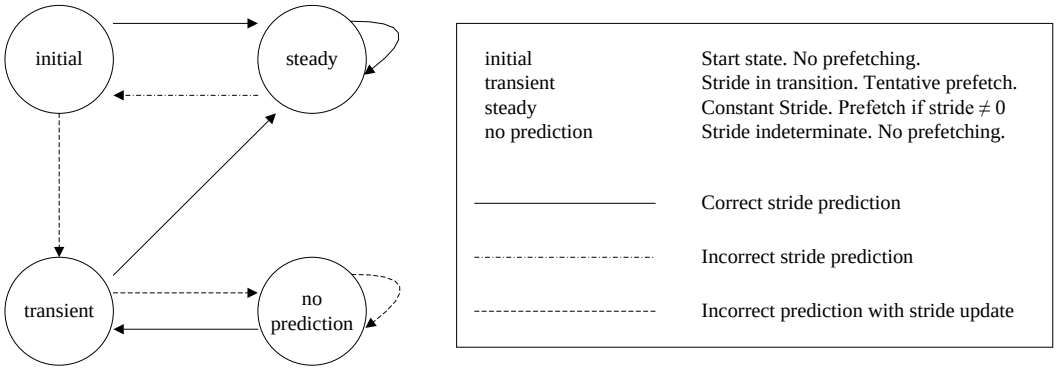

图 5-7 RPT state 的状态迁移

根据图 5-7 的状态迁移关系，我们考察以下实例，如图 5-8 所示。其中左图为一个 3-Nested Loop Iterations，并对数据 a 进行赋值操作，其中数组 a，b 和 c 使用的 Stride 参数并不相同。但是在一下程序中，数据 a，b 和 c 使用的 Stride 参数依然具有强烈的规律性，在 RPT 中分别保存着这些规律，从而在一定程度上提高了预读的准确性。

```
float a[100][100], b[100][100], [100][100];

…

for ( i = 0; i < 100; i++)
        for ( j = 0; j < 100; j++)
                for ( k = 0; k < 100; k++)
                        a[i][j] += b[i][k] * c[k][j];
```

instruction tag	previous address	stride	state
ld b[i][k]	20,000	0	initial
ld c[k][j]	30,000	0	initial
ld a[i][j]	10,000	0	initial

after first iteration

instruction tag	previous address	stride	state
ld b[i][k]	20,004	4	transient
ld c[k][j]	30,400	400	transient
ld a[i][j]	10,000	0	steady

after second iteration

instruction tag	previous address	stride	state
ld b[i][k]	20,008	4	steady
ld c[k][j]	30,800	400	steady
ld a[i][j]	10,000	0	steady

图 5-8 Matrix multiply 程序的硬件预读

假设数组 a, b 和 c 的基地址分别以 10,000, 20,000 和 30,000 对界。在第一次进行运算时，通过计算可以在 RPT 表中记录相应的 Previoud Address，数组 a，b 和 c 的 Stride 参数为初始值 0，而 State 为初始状态 Initial。

在第一次 Iteration 之后，RPT 表中的数组 b 和 c 的 Stride 分别为 4 和 400(Current Address 与 Previous Address 之差)，State 改变为 Transient，并开始预读之后的 Cache Block，而通过计算数组 a 的 Stride 为 0，与之前的值相同，State 改变为 Steady，即不进行预读；在第二次 Iteration 之后，RPT 中的数组 b, c 和 a 发现 Stride 没有再次发生变化时，State 改变为 Steady，开始稳定地进行预读。

在第一重循环 k 执行完毕后，由于 k 的变化，将使 RPT 的数组 c 进入 Initial 状态，重新进入准备阶段；第二重循环 j 执行完毕后，由于 j 的变化，将使 RPT 的数组 a 和 c 进入 Initial 状态，重新进入准备阶段；第三重循环 i 执行完毕后，由于 i 的变化，将使 RPT 的数组 a 和 b 进入 Initial 状态，重新进入准备阶段。

周而复始，直到三重循环完全执行完毕。

采用这种硬件预读方法，可以有效解决在 Loop Iterations 中数据的 Stride 问题。在进一步考虑了 Prefetch Distance，即 δ 参数的基础上，Lookahead data prefetching 算法可以在此基础上继续优化，可以设置一个 LA-PC(Lookahead Program Count)。此时预读的地址 Prefetch Address 等于 Effective Addess + (Stride × δ)，LA-PC 与 PC 的差值即为 δ。

在某些情况下，基于 RPT 的预读机制并不能理想地处理 Triangle-Shaped Loop，这种 Loop 访问 Stride 值的计算不但与自身有关，而且与相邻的 Loop 直接相关。采用 Correlated Reference Prediction 预读机制[102]可以有效解决这一问题。

该机制的实现要点是除了关注在一个 Loop 内的数据访问轨迹之外，还关心相邻的 Loop，以实现对 Triangle-Shaped Loop 的预读。为此在图 5-6 中需要加入另外一组 Prev Address 和 Stride 参数，对此有兴趣的读者可参阅[102]以获得更详细的信息。

无论是软件还是硬件 Prefetch 的实现方式，都不可避免地出现 Prefetch 得来的数据并没有被及时使用，从而会在一定程度上一定程度上的重复，这种重复会进一步提高系统功耗，对于有些功耗敏感的应用，需要慎重使用 Prefetch 机制。Prefetch 机制除了对系统有较大影响之外，还会引发一定程度的 Cache Pollution。这使得 Stream buffer[20]机制因此引入。

5.4 Stream Buffer

Stream Buffer 是一种广义 Cache，主要功能是避免因为预读而造成的 Cache Pollution 问题。当采用该机制时，处理器可以将预读的数据序列放入 Stream Buffer 中而不是放入 Cache，如果处理器使用的数据没有在 Cache 中命中，将首先在 Stream Buffer 中查找，采用这种方法可以消除预读对 Cache 的污染，但是也因此增加了系统设计的复杂性。Stream Buffer 的组成结构如图 5-9 所示。

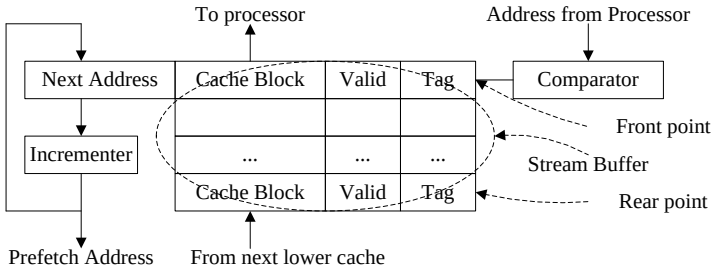

图 5-9 Stream Buffer 的组成结构[104][①]

在一个 Stream Buffer 中，由多个 Entry 组成，在这个 Entry 中可以存放一个或者多个 Cache Block，也包含若干个状态位。Stream Buffer 的每一个 Entry 由 Cache Block，Valid 位和与此对应的地址 Tag 组成。其中 Valid 位表示当前 Cache Block 中的数据是否有效，而地址 Tag 用来进行地址比较。Stream Buffer 的使用方法与 FIFO 类似，从 Front 指针处开始使用，新的数据将填入 Rear 指针的位置。

出现 Cache Miss 时，微架构首先在 Stream Buffer 的 Front 开始寻找数据，如果命中，该数据才预读进入 Cache，从而不会造成 Cache Pollution，同时预读进行 Cache 的数据将从 Stream Buffer 的头部移除。随后微架构根据 Prefetch Address 从其下 Cache Hierarchy 中获得 Cache Block，并填写 Rear 指针对应 Entry 的 Tag 信息，数据返回时将填写相应的 Cache Block，

[①] 与[20]中的 Stream Buffer 示意图相比，[104]中的图片更为直观一些。

并将 Valid 位置为有效。

如果数据在 Stream Buffer 中 Miss，而且系统中只有一个 Stream Buffer，该 Stream Buffer 将被刷新，并试图建立新的预读序列。显然在多数情况下，设立一个 Stream Buffer 并不合理，在一个实际的应用中，一个任务经常会访问多个 Stride 不同的数据序列，如图 5-8 所示。为此在现代微架构中，一般设置多个 Stream Buffer，即 Multi-Way Stream Buffers，其组成结构如图 5-10 所示。

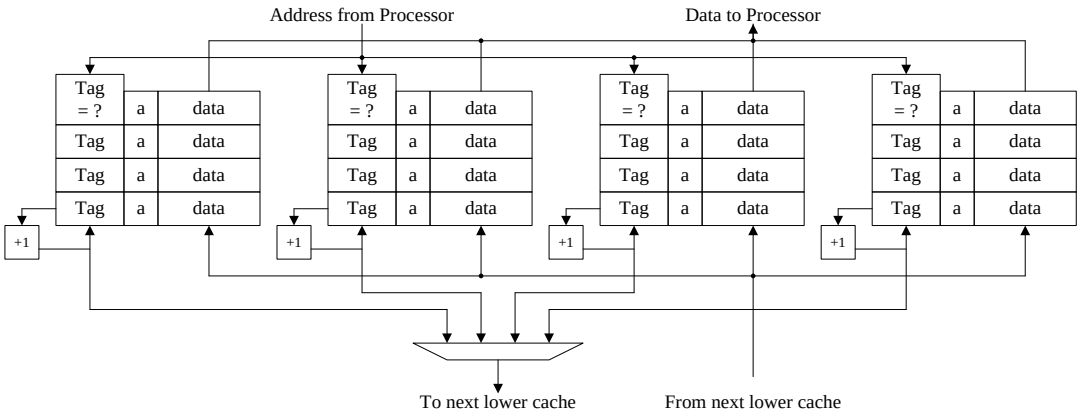

图 5-10 Multi-way stream Buffer 的组成结构[20]

当出现 Stream Buffer Miss 时，将使用某种替换算法，LRU 或者 PLRU，替换其中的一个 Stream Buffer，以装填新的访问序列。当使用这种结构时，如果一个任务需要访问 Stride 不同的几种数据序列时，可以使用不同的 Stream Buffer，从而有效提高了 Stream Buffer 的利用率。在一个微架构的具体实现中还可以将 Stream Buffer 与 Lookahead Data Prefetching 方式联合使用，其结构示意如图 5-11 所示。

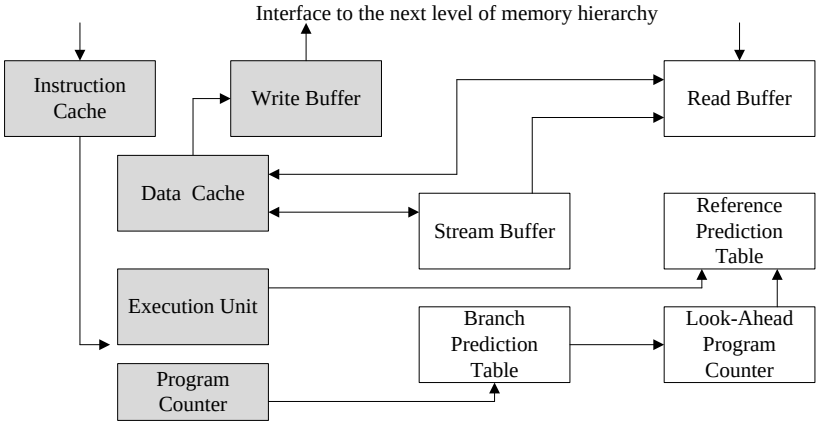

图 5-11 联合使用 Stream Buffer 与 Lookahead data prefetching 方式[20]

即便是使用这种硬件预读方式，也无法彻底解决因为预读带来的 Cache Pollution 问题，很难解决预读数据的及时有效等一系列。硬件预读机制不断的发展演变过程，与程序的分支预测有某些相近之处，其本质都是硬件自学习数据访问轨迹的过程。

各类 Stride Prefeching，Distance Prefeching 和 Global Histrory Buffer Prefeching 算法，其本质均是如此，没有必要对此再一一进行介绍。很多从 Qualitative Research 看起来非常不错的预读算法，其 Quantitative Analysis 的最终结果未必能够超过 OBL 算法。这些优化方法都有较强的针对性，在某类 Access Pattern 之下有较好的表现，而在其他情况之下并不适用。

在 Prefectch 这个领域，有时简单逻辑获得的效果并不弱于复杂逻辑。

这也引发了一个思考，是否应该把更多的硬件资源用于微架构的其他部分，而非用于硬件预读。一些简单的方法可能就是最优，比如 OBL 实现和最基本的 Stream Buffer。这一切依然是一个深层次的 Trade-Off 问题，没有优劣之分。

结束语

搁笔不意味着结束。许久之前，我与怀临先生聊过准备书写有关 Cache 的文字，这不是书写的目的，这篇文章与重然诺如邱山没有太多联系。心中想着《菜根谭》中的“宠辱不惊，闲看庭前花开花落；去留无意，漫随天外云卷云舒”，不知不觉完成了这些文字。

只是我依然尚不明了为何去写这些文字，不清楚如此惜寸阴，却花费了如此精力；不明了为何一直去在忽视，忍受着各种忽视去完成这篇文章。我奢求完成时可以发现少许原因。待到结束，却愈发模糊。

我们所处的年代与之前所有年代一样，总有些可以继承的事物。近些年我一直品读着这些事物，他们的尊严与智慧在历经时光磨砺后没有消失，而是加倍地尚显出来。这些可以被继承的事物并不是多数个体群体苛求的财富。

财富可以评价许多事物，就是不能评价生命为何高贵，就是不能让子孙后代去赖以自豪。堆积的财富终为土灰。卸任时留给美国政府最多财富的克林顿总统，在就职时曾说过一段话，When our founders boldly declared America's independence to the world and our purposes to the Almighty, they knew that America, to endure, would have to change. Not change for change's sake, but change to preserve America's ideals; life, liberty, the pursuit of happiness. Though we march to the music of our time, our mission is timeless.

使命没有高下之分，都是为尊严而战。尊严很贵，不能去乞讨，更没有人会给予你，只有赚足了本钱，一口气赢回来。这种本钱并不是财富。可以富可敌国依然无法赢得士人之心，古已有之。大人物有其使命，小个体有自己的追逐，没有高下之分。

圣经中有段话“And let us not be weary in well-doing, for in due season, we shall reap, if we faint not”，翻译成中文是“我们努力，不求回报，时候到了，就有收成”。我明白没有什么特别的目的驱使我完成这些文字。

真放肆不在饮酒高歌。兴之所至，无处不是乐土。我安于在这条轨迹中前行，只要前方有路，不在乎路途遥远。这次书写，比之前的完成的所有文章难出许多，每次在获得少许的进展后，发现的是更多的无知。我喜欢这种无知。近来多读《坛经》，以其中的一句话作为全文的结束。

世界虚空，能含万物色像。日月星宿、山河大地、泉源溪涧、草木丛林、恶人善人、恶法善法、天堂地狱、一切大海、须弥诸山，总在空中。

参考资料

[1] Opher Kahn and Bob Valentine [June 2010]. Intel Next Generation Microarchitecture Codename Sandy Bridge: New Processor Innovations. Intel IDF2010 San Francisco, CA. http://www.intel.com/idf/audio_sessions.htm

[2] T. Kilburn, D.B.G. Edwards, M.J. Lanigan, F.H. Sumner, IRE Trans [Apr. 1962]. One-Level Storage System. Electronic Computers April 1962.

[3] Freescale. E500 TLB Entries. http://forums.freescale.com/freescale/attachments/freescale/CWCFCOMM/2355/1/e500_tlb.pdf

[4] Freescale. EREF: A Programmer's Reference Manual for Freescale Embedded Processors . http://cache.freescale.com/files/32bit/doc/ref_manual/EREFRM.pdf

[5] Intel [May 2011]. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1, Section 4.10.1 Process-Context Identifiers (PCIDs). http://www.intel.com/Assets/pdf/manual/253668.pdf

[6] Hans de Vries [Sep. 2003]. Understanding the detailed Architecture of AMD's 64 bit Core. http://www.chip-architect.com/news/2003_09_21_Detailed_Architecture_of_AMDs_64bit_Core.html.

[7] David A. Patterson and John L. Hennessy [Jan. 2008]. Computer Architecture A Quantitative Approach, Fourth Edition. ISBN：13:978-0-12-370490-0 ISBN 10:0-12-370490-1. Original English language edition copyright by Elsevier Inc. Published by China Machine Press.

[8] J. Navarro [Apr. 2004]. Transparent operating system support for superpages. PhD thesis, Rice University, Houston, Texas.

[9] Juan E. Navarro [Apr. 2004]. Transparent operating system support for superpages.

[10] Adam G. Litke [Jun. 2007]. "Turning the Page" on Hugetlb Interfaces. Proceedings of the Linux Symposium Volume One. June 27th–30th, 2007.

[11] Narayanan Ganapathy and Curt Schimmel [1998]. General purpose operating system support for multiple page sizes. Proceeding ATEC '98 Proceedings of the annual conference on USENIX Annual Technical Conference.

[12] Michael E. Thomadakis, Ph.D [Jan. 2011].The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms. http://sc.tamu.edu/systems/eos/nehalem.pdf.

[13] Hughes; William Alexander, Ramagopal; Hebbalalu S., Meyer; Derrick R., Conor; Stephen M. Snoop resynchronization mechanism to preserve read ordering. US Patent 6,473,837.

[14] G. Reinman and B. Calder [Dec. 1998]. Predictive techniques for aggressive load speculation in 31st International Symposium on Microarchitecture.

[15] G. Reinman and B. Calder [May. 2000]. A comparative Survey of Load Speculation Architectures. Journal of Instruction-Level Parallelism.

[16] Digital Semiconductor [Jun. 1996]. Alpha 21064 and Alpha 21064A Microprocessors Hardware Reference Manual

[17] Compag Computer Corporation [Dec. 1998]. Alpha 21164 Microprocessor Hardware Reference Manual.

[18] Compag Computer Corporation [Jul. 1999]. Alpha 21264 Microprocessor Hardware Reference Manual.

[19] A. Moshovos, G.S. Sohi [Dec. 1997]. Streamlining inter-operation memory communication via data dependence prediction. In 30th International Symposium on Microarchitecture.

[20] Norman P. Jouppi [Jun. 1990]. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers, ACM SIGARCH Computer Architecture News, v.18 n.3a, p.364-373, June 1990.

[21] G. Tyson, T. M. Austin [Dec. 1997]. Improving the accuracy and performance of memory communication through renaming. In 30th Annual International Symposium on Microarchitecture, pages 218–227.

[22] Andrew Glew [Oct. 1998]. MLP yes! ILP no! ASPLOS Wild and Crazy Idea Session'98.

[23] Alan Jay Smith [Sep. 1982]. Cache Memories. ACM Computing Surveys Volume 14 Issue 3.

[24] JEDEC [Jul. 2010]. DDR3 SDRAM STANDARD.

[25] Kostas Pagiamtzis, Student Member, IEEE, and Ali Sheikholeslami, Senior Member, IEEE [Mar. 2006]. Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey. IEEE JOURNAL OF SOLID-STATE CIRCUITS, VOL. 41, NO. 3.

[26] André Seznec [May. 1993]. A case for two-way skewed-associative caches. ISCA '93 Proceedings of the 20th annual international symposium on computer architecture.

[27] Chenxi Zhang, Xiaodong Zhang and Yong Yan [Sep. 1997]. Two Fast and High-Associativity Cache Schemes, IEEE Micro, v.17 n.5, p.40-49.

[28] H. Vandierendonck and K. D. Bosschere [2008]. Constructing optimal XOR-functions to minimize cache conflict misses. In Proc. Int. Conf. on Architecture of Computing Systems (ARCS). Springer, pp. 261-272.

[29] Antonio González, Mateo Valero, Nigel Topham and Joan M. Parcerisa [Jul. 1997]. Eliminating cache conflict misses through XOR-based placement functions, Proceedings of the 11th international conference on Supercomputing, p.76-83, July 07-11, 1997, Vienna, Austria.

[30] R. E. Kessler, Mark D. Hill [Nov. 1992]. Page placement algorithms for large real-indexed caches, ACM Transactions on Computer Systems (TOCS), v.10 n.4, p.338-359.

[31] Yehuda Afek, Dave Dice and Adam Morrison [Jun. 2011]. Cache Index-Aware Memory Allocation. ISMM'11, San Jose, California, USA.

[32] Mark S. Papamarcos and Janak H. Patel [Jun. 1984]. A low-overhead coherence solution for multiprocessors with private cache memories.

[33] P. Sweazey and A. J. Smith [Jun. 1986]. A class of compatible cache consistency protocols and their support by the IEEE futurebus, Proceedings of the 13th annual international symposium on Computer architecture, p.414-423, Tokyo, Japan.

[34] Herbert H. J. Hum and James R. Goodman [Jul. 2005]. Forward State for use in Cache Coherency in a Multiprocessor System. US Patent No. 6,922,756 B2. Original Assignee: Intel Corporation. July 26, 2005.

[35] GURURAJ S. RAO [Jul. 1978]. Performance Analysis of Cache Memories. Journal of the ACM (JACM) Vol 25, No 3.

[36] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic [Apr. 2004]. Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite. Proc. 42nd ACM Southeast Regional Conference, 2004.

[37] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm [Nov. 2007]. Timing Predictability of Cache Replacement Policies. Real-Time Systems. Volume 37, Number 2.

[38] Elizabeth J. O'Neil, Patrick E. O'Neil and Gerhard Weikum [Jun. 1993]. The LRU-K Page Replacement Algorithm for Database Disk Buffering. Proc. ACM SIGMOD, Washington, D.C.,

Volume 22 Issue 2.
[39] Theodore Johnson and Dennis Shasha [Sep. 1994]. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. Proc. 20th VLDB Conf., Santiago, Chile, 1994.
[40] Song Jiang and Xiaodong Zhang [Jun. 2002]. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In Proc. ACM SIGMETRICS Conf., 2002.
[41] Song Jiang and Xiaodong Zhang [Jun. 2002]. The PPT of LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. www.ece.eng.wayne.edu/~sjiang/Projects/LIRS/sig02.ppt.
[42] Song Jiang, Feng Chen and Xiaodong Zhang [Apr. 2005]. CLOCK-Pro: an effective improvement of the CLOCK replacement. ATEC '05 Proceedings of the annual conference on USENIX Annual Technical Conference. USENIX Association Berkeley, CA, USA.
[43] Freescale [Apr. 2005]. PowerPC™ e500 Core Family Reference Manual Supports e500v1 and e500v2. Rev. 1, 4/2005. Pg. 384. 11-22.
[44] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu and Yale N. Patt. [May 2006]. A Case for MLP-Aware Cache Replacement, ACM SIGARCH Computer Architecture News, v.34 n.2, p.167-178.
[45] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely and Joel Emer [Jun. 2007]. Adaptive insertion policies for high performance caching, Proceedings of the 34th annual international symposium on Computer architecture, June 09-13, 2007, San Diego, California, USA.
[46] Jayesh Gaur, Mainak Chaudhuri and Sreenivas Subramoney [Jun. 2011]. Bypass and Insertion Algorithms for Exclusive Last-level Caches. ISCA'11, June 4-8, 2011, San Jose, California, USA.
[47] Tse-Yu Yeh and Yale N. Patt [May 1992]. Alternative implementations of two-level adaptive branch prediction. ISCA '92 Proceedings of the 19th annual international symposium on Computer architecture.
[48] TOMASULO, R. M [Jun. 1967]. An efficient algorithm for exploiting multiple arithmetic units. IBM J Res. Dev 11, 1 (Jan. 1967), 25-33.
[49] Gurindar S. Sohi and Manoj Franklin [Sep. 1990]. High-bandwidth data memory systems for superscalar processors, Proceedings of the fourth international conference on Architectural support for programming languages and operating systems, p.53-62, Santa Clara, California, United States。
[50] Toni Juan, Juan J. Navarro and Olivier Temam [Jul. 1997]. Data caches for superscalar processors, Proceedings of the 11th international conference on Supercomputing, p.60-67, July 07-11, 1997, Vienna, Austria.
[51] David Kroft [May 1981]. Lockup-free instruction fetch/prefetch cache organization. Proceedings of the 8th annual symposium on Computer Architecture, p.81-87, May 12-14, 1981, Minneapolis, Minnesota, United States.
[52] David Kroft [1998]. Retrospective: lockup-free instruction fetch/prefetch cache organization. Published in: Proceeding ISCA '98 25 years of the international symposia on Computer architecture.
[53] Keith I. Farkas, Paul Chow, Norman P. Jouppi and Zvonko Vranesic [Jun. 1997]. Memory-system design considerations for dynamically-scheduled processors, Proceedings of the 24th annual international symposium on Computer architecture, p.133-143, June 01-04,

1997, Denver, Colorado, United States.

[54] Keith I. Farkas and Norman P. Jouppi [Apr. 1994]. Complexity/performance tradeoffs with non-blocking loads, Proceedings of the 21ST annual international symposium on Computer architecture, p.211-222, April 18-21, 1994, Chicago, Illinois, United States.

[55] Sarita V. Adve and Kourosh Gharachorloo [Sep. 1995]. Shared Memory Consistency Models: A Tutorial. Western Research Laboratory Research Report 95/7, Digital Equipment Corporation Palo Alto, California 94301-1616.

[56] Ajay D. Kshemkalyani and Mukesh Singhal [Mar. 2011]. Distributed Computing: Principles, Algorithms, and Systems, Section 12.2 Memory consistency models. Cambridge University Press; Reissue edition. ISBN-10: 0521189845. ISBN-13: 978-0521189842

[57] Seth Gilbert and Nancy Lynch [Jun. 2002]. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, ACM SIGACT News, v.33 n.2.

[58] Werner Vogels [Jan. 2009]. Eventually consistent. Communications of the ACM, v.52 n.1.

[59] Fredrik Dahlgren [May 1995]. Boosting the performance of hybrid snooping cache protocols. Proceeding ISCA '95 Proceedings of the 22nd annual international symposium on computer architecture.

[60] James R. Goodman [Jun. 1983]. Using Cache Memory to Reduce Processor-Memory Traffic. Proceedings of the 10th Annual International Symposium on Computer Architecture, pp 124-131, 1983.

[61] James R. Goodman and Philip J. Woest [May 1988]. The Wisconsin multicube: a new large-scale cache-coherent multiprocessor. Proceeding ISCA '88 Proceedings of the 15th Annual International Symposium on Computer architecture.

[62] AMD [May 2011]. AMD64 Technology--AMD64 Architecture Programmer’s Manual Volume 2: System Programming, Section 7.3 Memory Coherency and Protocol. Revision 3.18.

[63] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy [Apr. 1994]. The Stanford FLASH multiprocessor. Proceeding ISCA '94 Proceedings of the 21st annual international symposium on Computer architecture.

[64] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta and John Hennessy [Jun. 1990]. The directory-based cache coherence protocol for the DASH multiprocessor. Proceeding ISCA '90 Proceedings of the 17th annual international symposium on Computer Architecture. ACM New York, NY, USA.

[65] Mark S. Papamarcos and Janak H. Patel [Jun. 1984]. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. Proceeding ISCA '84 Proceedings of the 11th annual international symposium on Computer architecture.

[66] Amin Firoozshahian [Dec 2008]. Smart Memories: A Reconfigurable Memory System Architecture. PhD thesis, Stanford University.

[67] Eric Rotenberg, Steve Bennett and James E. Smith [Dec. 1996]. Trace cache: a low latency approach to high bandwidth instruction fetching, Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture, p.24-35, December, 1996, Paris, France.

[68] Tom Shanley [Jul. 2004]. Chapter 38 Pentium 4 core description, pg 901, Chapter 40. The Pentium 4 Caches, the unabridged Pentium 4 IA32 processor genealogy. Mindshare. Addison-Wesley. ISBN: 0-321-24656-X.

[69] David Kanter [Sep. 2010]. Intel's Sandy Bridge Microarchitecture. http://www.realworldtech.com/page.cfm?ArticleID=RWT091810191937&p=1

[70] Steven Przybylski, John Hennessy, and Mark Horowitz [May 1989]. Characteristics of Performance-Optimal Multi-level Cache Hierarchies. In Proc. of the 16th Annual International Symposium on Computer Architecture.

[71] STREAM "standard" results. http://www.cs.virginia.edu/stream/standard/Bandwidth.html

[72] Chetana N. Keltcher, Kevin J. McGrath, Ardsher Ahmed and Pat Conway [Mar. 2003]. The AMD Opteron Processor for Multiprocessor Servers. IEEE Micro, vol. 23, no. 2, pp. 66-76.

[73] AMD [Sep. 2005]. Software Optimization Guide for AMD Athlon™ 64 and AMD Opteron™ Processors. Revision 3.06.

[74] David Kanter [Aug. 2010]. AMD's Bulldozer Microarchitecture. http://www.realworldtech.com/page.cfm?ArticleID=RWT082610181333&p=8

[75] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy [Jan 2002]. Power4 System Microarchitecture. IBM Journal of Research and Development, 46(1), Jan 2002.

[76] B. Sinharoy, R. Kalla, J. Tendler, R. Eickemeyer, and J. Joyner [Jul. 2005]. Power5 System Microarchitecture. IBM Journal of Research and Development, 49(4), July 2005.

[77] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz and M. T. Vaden [Nov. 2007], IBM POWER6 microarchitecture, IBM Journal of Research and Development, v.51 n.6, p.639-662, November 2007.

[78] Ron Kalla, Balaram Sinharoy, William J. Starke and Michael Floyd [Mar. 2010]. Power7: IBM's Next-Generation Server Processor, IEEE Micro, v.30 n.2, p.7-15, March 2010.

[79] J. Kahl, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy [2005]. Introduction to the Cell Multiprocessor. IBM Journal of Research and Development, 49(4), 2005.

[80] Tom R. Halfhill [Jul. 2010]. NetLogic Broadens XLP Family Multithreading and Four-Way Issue with One to Eight CPU Cores. Microprocessor Report.

[81] Kunle Olukotun, Lance Hammond and James Laudon [2007]. Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency, Morgan and Claypool Publishers, 2007.

[82] RIKEN Fujitsu Limited [Jun. 2011]. Supercomputer "K computer" Takes First Place in World. http://www.fujitsu.com/global/news/pr/archives/month/2011/20110620-02.html

[83] Jean-Loup Baer and Wen-Hann Wang [May 1988]. On the inclusion properties for multi-level cache hierarchies, Proceedings of the 15th Annual International Symposium on Computer architecture, p.73-80, May 30-June 02, 1988, Honolulu, Hawaii, United States.

[84] Bradford M. Beckmann, Michael R. Marty and David A. Wood [Dec. 2006]. ASR: Adaptive Selective Replication for CMP Caches, Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, p.443-454, December 09-13, 2006.

[85] Wen-Hann Wang [1989]. Multilevel Cache Hierarchies. Ph.D. Dissertation. University of Washington. AAI9013828.

[86] AMD [Jun. 2000]. AMD Athlon™ Processor and AMD Duron™ Processor with full-speed on-die L2 cache. June 19, 20

[87] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes [Apr. 2010]. Cache hierarchy and memory subsystem of the AMD opteron processor. IEEE Micro, vol. 30, no. 2, pp. 16-29, Apr. 2010.

[88] Michael Zhang, Krste Asanovic [Jun. 2005]. Victim Replication: Maximizing Capacity while

Hiding Wire Delay in Tiled Chip Multiprocessors, Proceedings of the 32nd annual international symposium on Computer Architecture, p.336-345, June 04-08, 2005.

[89] Ying Zheng, Brain T. Davis, Matthew Jordan [Mar. 2004]. Performance evaluation of exclusive cache hierarchies, in: ISPASS '04: Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software, IEEE Computer Society, Washington, DC, USA, 2004, pp. 89–96.

[90] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr. and Joel Emer [Dec. 2010]. Achieving Non-Inclusive Cache Performance with Inclusive Caches: Temporal Locality Aware (TLA) Cache Management Policies. Proceeding MICRO '43 Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture.

[91] Michael R. Marty, Jesse D. Bingham, Mark D. Hill, Alan J. Hu, Milo M. K. Martin and David A. Wood [Feb. 2005]. Improving Multiple-CMP Systems Using Token Coherence, Proceedings of the 11th International Symposium on High-Performance Computer Architecture, p.328-339, February 12-16, 2005.

[92] Yuichiro Ajima, Shinji Sumimoto and Toshiyuki Shimizu [Nov. 2009]. Tofu: A 6D Mesh/Torus Interconnect for Exascale Computers, Computer, v.42 n.11, p.36-40, November 2009.

[93] http://www.gem5.org/dist/tutorials/isca_pres_2011.pdf.

[94] GEM5 source code ./src/mem/protocol/MOESI_CMP_directory-L1cache.sm

[95] http://gem5.org/Cache_Coherence_Protocols.

[96] Norman P. Jouppi [May 1993]. Cache write policies and performance, Proceedings of the 20th annual international symposium on Computer architecture, p.191-201, May 16-19, 1993, San Diego, California, United States.

[97] Intel [June 2011]. Intel 64 and IA-32 Architectures Optimization Reference Manual. Section 2.1.1 Intel microarchitecture code name Sandy Bridge Pipeline Overview.

[98] David Kanter [Jul. 2011]. Sandy Bridge for Servers. http://realworldtech.com/page.cfm?ArticleID=RWT072811020122&p=1

[99] Steven P. Vanderwiel and David J. Lilja [Jun. 2000]. Data prefetch mechanisms, ACM Computing Surveys (CSUR), v.32 n.2, p.174-199.

[100] Doug Joseph and Dirk Grunwald [May 1997]. Prefetching using Markov predictors. ISCA '97 Proceedings of the 24th annual international symposium on Computer architecture.

[101] Tien-Fu Chen and Jean-Loup Baer [Apr. 1994]. A performance study of software and hardware data prefetching schemes, Proceedings of the 21ST annual international symposium on Computer architecture, p.223-232, April 18-21, 1994, Chicago, Illinois, United States.

[102] Tien-Fu Chen and Jean-Loup Baer [May 1995]. Effective Hardware-Based Data Prefetching for High-Performance Processors, IEEE Transactions on Computers, v.44 n.5, p.609-623.

[103] G. S. Manku, M. R. Prasad, and D. A. Patterson [Dec. 1997]. A new voting based hardware data prefetch scheme. Proc. of IEEE Int. Conf. High-Performance Computing, pp.100 - 105, 1997.

[104] S. Palacharla and R. E. Kessler [Apr. 1994]. Evaluating stream buffers as a secondary cache replacement, Proceedings of the 21ST annual international symposium on Computer architecture, p.24-33, April 18-21, 1994, Chicago, Illinois, United States.