彎曲評論

科技 · 人物 · 潮流

弯曲评论论文集(中)

(TekTalk Lecture Notes)

[谨于此文集献给以邓稼先(1924/6/25/--1986/7/29)/等为代表的,为国家和民族利益鞠躬尽瘁的科技知识分子]

弯曲评论编辑部

2012年6月25日

目录(中)

大数据从“小”做起——中小企业Big Data解决之道	吴朱华
下一代数据中心的虚拟接入技术–VN-Tag和VEPA	libing
从半空看虚拟化	Roy
浅谈 数据中心 。 高压直流	KPang
网络处理器的讨论	弯曲评论
浅谈CPU Cache Memory	王齐, Xi Yang, Yuhao Zhu
再谈Intel CPU微结构--过去,现在和将来	陈怀临
关于城域网的思考	理客
浅谈思科核心路由器CRS-1	陈怀临
浅谈思科QuantumFlow处理器及其战略	陈怀临

大数据从“小”做起——　　Big Data解决之道

任何一个时代或者模式的兴起，都离不开与之相关的Killer App，比如，C/S时代的SAP ERP，互联网 1.0 时代的门户，以及互联网 2.0时代的搜索和SNS等，那么在当今云计算这个时代有那些Killer App呢？当然首当其冲的肯定是以VMware 和Amazon EC2为代表的虚拟化和相关IaaS服务，除此之外，新近崛起的大数据绝对也是云计算的Killer App之一，并且不仅类似百度、阿里以及腾讯这样的互联网巨头有相关的应用需求，而且根据我个人平时与客户接触，发现有很多普通中小企业，特别是中型的互联网和物联网企业，在这方面的场景也有很多。本文将首先给大家介绍一下在我眼中的大数据，以及大数据的意义和特点，再给大家聊聊大数据的常见处理流程，之后将会和大家分享一下我是如何帮助一些中小企业实施大数据相关的解决方案，也就是大数据如何从“小”做起。

什么是大数据？

过去计算机产生的数据，较简单，基本上都是一笔笔事务，总量虽大，但是都是整体增长幅度都还是可控的，比如传统的金融企业，经常使用几台大型机就管理其所有的业务数据，而最近几年，由于以平板、智能手机和传感器为代表的智能设备越来越多，同时这些设备的生成的数据更是远远地超过我们的想象。据美国著名咨询公司IDC的统计，全球数字信息在未来几年将呈现惊人增长，预计到2020年总量将是现在的44倍。据另外一份数据显示，全球 90% 的数据都是在过去两年中生成的，并且每年以50%的速度进行增长，每天，遍布世界各个角落的传感器、移动设备、在线交易和社交网络产生上PB级别的数据；每个月，全球网友发布了 10多 亿条 Twitter 信息和300多 亿条 Facebook 信息。那么这些大数据的存在有什么价值和意义呢？

大数据的意义

我个人和一些朋友一直觉得大数据就好比一口油井，因为里面蕴含着非常丰富的价值，如果企业能有效利用其内部存储的海量数据，那么将会改善其自身的产品和服务，从而提升客户和受众的体验，从而在大数据时代获取竞争优势，并且随着本身分析和挖掘技术不断地提升，可以在之前的基础上提供新的决策模式，从而支持管理者进行快速和精确地决策，这样能够超越对手，抢占市场先机，独领风骚。
下面通过几个行业来和大家举例讲解一下大数据有那些意义和作用？

互联网企业

我们有一些客户，他们主要是做网络舆情或者网络广告，他们明天都会处理和收集TB级别日志或者网页，结构化和非结构化都有，他们就是通过分析这些数据来给其客户提供价值，比如分析一下一个男性护肤品广告是在世界杯期间投放好，还是在亚洲杯那段时间播出好？还有，在电子商务方面，国外eBay的分析平台每天处理的数据量高达100PB，超

过了纳斯达克交易所每天的数据处理量。为了准确分析用户的购物行为，eBay定义了超过500种类型的数据，对顾客的行为进行跟踪分析，并且通过这些分析促进eBay自身的业务创新和利润增长。

智能电网

我们有一个合作伙伴，他们是做智能电网相关的解决方案。对那些电网而言，如果无法准确预估实际电力的使用情况，将会使电网要求电厂发出过量的电力，虽然这些过量电力可以通过某种模式进行保存，但是大量的电力浪费已不可避免。而通过他们智能电网的解决方案，每隔一刻钟会采集一个省几千万用户的用电数据，之后他们会根据这些数据来精确分析用户的用电模型，最后通过这个用电模型来优化电力生产，从而有效地减少电力资源的浪费。

车联网

在车联网方面，我们也有一个客户，他们在一个城市有几十万台基于Android的终端，而这些终端会每隔一段时间会发送具体位置的GPS消息给后端的数据集群，接着这些集群会分析一下这些海量的GPS信息，分析出那些路段在什么时候比较堵，之后将这些非常有价值的信息不断地推送给客户，从而帮助用户减少在路上所消耗的时间。

医疗行业

在这个方面，大数据的用例有很多。首先，通过分析大量的病例信息，将有效地帮助医生治病；其次，假设在一个病人身体的多个节点加入探针设备，而且每个探针每天会采集GB级别关于人体细胞和血液运行状态的数据，之后计算集群可以根据这些数据来来进行分析，这样能更精确地判断病因，从而让医生对病人进行更具针对性地治疗。

机器学习

在这方面，最出名的例子莫过于最近很火的Siri，它后台有一个庞大的HBase集群来对类似语言这样的文本数据进行分析和管理，从而使Siri变成一位越来越老练的个人助手，为iPhone 4S的用户提供了日期提醒、天气预报和饭店建议等服务。除此之外，还有IBM的Watson，它通过一个基于Hadoop UIMA框架的集群来挖掘海量的文本信息来实现一定程度的人工智能，并在美国著名知识问答节目Jeopardy中战胜多位出色的人类选手。

国家安全

这方面最出名的例子，莫过于美国的联邦情报局（CIA）。在过去10年中，他们通过无人侦察机收集了大量阿富汗那边地理相关的视频资料，之后通过分析这些海量视频资料，来对极具危害性的恐怖组织团伙进行定位。

大数据的特点

大数据，不仅有“大”这个特点，除此之外，它还有很多其他特色，在这方面，业界各个厂商都有自己独特的见解，但是总体而言，我觉得可以用“4V+1C”来概括，“4V+1C分别代表了Variety（多样化）、Volume（海量）、Velocity（快速）、Vitality（灵活）以及Complexity（复杂）这五个单词。

Variety（多样化）

大数据一般包括以事务为代表的结构化数据、以网页为代表的半结构化数据和以视频和语音信息为代表的非结构化等多类数据，并且它们处理和分析方式区别很大。

Volume（海量）
通过各种智能设备产生了大量的数据，PB级别可谓是常态，我接触的一些客户每天量都在几十GB，几百GB左右，我估计国内大型互联网企业的每天数据量已经接近TB级别。
Velocity（快速）
要求快速处理，因为有些数据存在时效性，比如电商的数据，假如今天数据的分析结果要等到明天才能得到，那么将会使电商很难做类似补货这样的决策，从而导致这些数据失去了分析的意义。
Vitality（灵活）
因为在互联网时代，和以往相比，企业的业务需求更新的频率加快了很多，那么相关大数据的分析和处理模型必须快速地适应。
Complexity（复杂）
虽然传统的BI已经很复杂了，但是由于前面4个V的存在，使得针对大数据的处理和分析更艰巨，并且过去那套基于关系型数据库的BI开始有点不合时宜了，同时也需要根据不同的业务场景，采取不同处理方式和工具。

大数据的常见处理流程

前面已经跟大家讲了处理大数据的必要性和特点，那么接着将谈到如何处理大数据，特别是常见的流程。具体的大数据处理方法其实有很多，但是根据长时间的实践，我总结了一个基本的大数据处理流程（图1），并且这个流程应该能够对大家理顺大数据的处理有所帮助。

图1. 大数据的常见处理流程

整个处理流程可以概括为四步，分别是采集、导入和预处理、统计和分析以及挖掘
采集
利用多个的数据库来接收发自客户端（Web、App或者传感器形式等）的数据，并且用户可以通过这些数据库来进行简单的查询和处理工作，比如，电商会使用传统的关系型数据库MySQL和Oracle等来存储每一笔事务数据，除此之外，Redis和MongoDB这样的NoSQL数据库也常用于数据的采集。
在采集部分，主要特点和挑战方面是并发数高，因为同时有可能会有成千上万的用户来进行访问和操作，比如著名用于购买火车票的12306站点和淘宝，它们并发的访问量在峰值时达到上百万，所以需要在采集端部署大量数据库才能支撑，并且如何在这些数据库之间进行负载均衡和分片的确是需要深入地思考和设计。
导入/预处理
虽然有采集端本身会有很多数据库，但是如果要对这些海量数据进行有效地分析，还是应该将这些来自前端的数据导入到一个集中的大型分布式数据库或者分布式存储集群，并且可以在导入基础上做一些简单的清洗和预处理工作，也有一些用户会在导入时候使用来自Twitter的Storm来对数据进行流式计算，来满足部分业务的实时计算需求。

在特点和挑战方面，主要是导入数据量大，每秒导入量经常达到百兆，甚至GB级别。

统计/分析

统计与分析主要利用分布式数据库或者分布式计算集群来对存储于其内的海量数据进行普通的分析和分类汇总等，以满足大多数常见的分析需求，在这方面，一些实时性需求会用到EMC 的GreenPlum、Oracle的Exadata以及基于MySQL的列式存储Infobright等，而一些批处理或者基于半结构化的需求可以使用Hadoop。

统计与分析这部分，主要特点和挑战方面是分析涉及的数据量大，其对系统资源，特别是I/O会有极大地占用。

挖掘

与前面统计和分析不同的是，数据挖掘一般没有什么预先设定好的主题，主要是在现有数据上面进行基于各种算法的计算，从而起到预测（Predict）的效果，这样实现一些高级别数据分析的需求，比较典型算法有用于聚类的K-Means、用于统计学习的SVM和用于分类的Naive Bayes，主要使用的工具有Hadoop的Mahout等。

在特点和挑战方面，主要是挖掘的算法复杂，并且计算涉及的数据量和计算量都很大，还有，常用数据挖掘算法库以单线程为主。

如何从"小"做起？

由于我平时与中小企业的接触非常频繁，虽然技术方案与实际的问题相关，很难在一篇文章当中详尽地道来。除了上面那个基本处理流程之外，我下面会将一些基本的从"小"做起的思路给大家阐述一下：

1. 认识自己的不足，主要是在技术、人力和财力等方面是不仅无法与Google和Amazon这样国外巨头比肩，而且与国内三大互联网BAT（百度、阿里巴巴和腾讯）也是无法比肩的，所以需要深刻认识；
2. 明确分析自己的需求，下面是几个常见的需求选项：
    a. 数据类型，是结构化，半结构化，还是非结构化为主；
    b. 数据大小，内部数据级别是TB级别、PB级别或者PB以上级别；
    c. 读写量级，比如每小时写入的数据达到GB级别，或者每天写入达到TB级别等；
    d. 读写比例，是写为主，还是以读为主；
    e. 并发数，大致的每秒并发数；
    f. 一致性，只接受强一致性，或者可以接受最终一致性和弱一致性；
    g. 延迟度，最高能容忍的延迟度是多少，是10毫秒，100毫秒，还是可以1秒
    h. 分析的复杂度，需不需要引入较复杂的数据挖掘算法等。
3. 要灵活使用现有的工具，首先，推荐使用一些开源或者是可以承受的商业软件，虽然个人并不排斥自建，但是一定要有具体的商业价值，并且最好是在现有工具上的画龙点睛，而不是从头开始构建；其次，工具方面应与具体的场景相关，在不同的场景要使用不同的工具。
4. 尽量不要走平台思路，应以具体的应用和场景为主，因为建一个平台有很多附

加的成本和设计，比如，Amazon的云平台是通过至少五年时间构建而成。特别是项目的初期，不建议走平台这个方向，而是应脚踏实地以具体的商业场景为主。

5. 找准切入点，最好是找到一个技术难度小，并且有一定的商业价值的场景来做大数据技术落地的试点，并不断地进行测试和迭代来验证，而不是一味求复杂，求大，这样比较容易说服企业管理层来进行长期地投入和支持；

最后，想和大家说一下，“罗马不是一天建成的”，无论是Google的用于大数据处理的基础设施，还是我们国内淘宝的“云梯”都是一步步通过不断积累和实践而成，所以我们这些中小企业应该贯彻“大处着眼、小处着手”的方针来持续地验证和推进。还有，我们人云科技将于今年上半年发布用于海量结构化数据处理的YunTable，由于其性能指标非常出色，并且已经有正式运行的大型集群，所以请各位朋友敬请期待。

下一代数据中心的虚拟接入技术–VN-Tag　VEPA

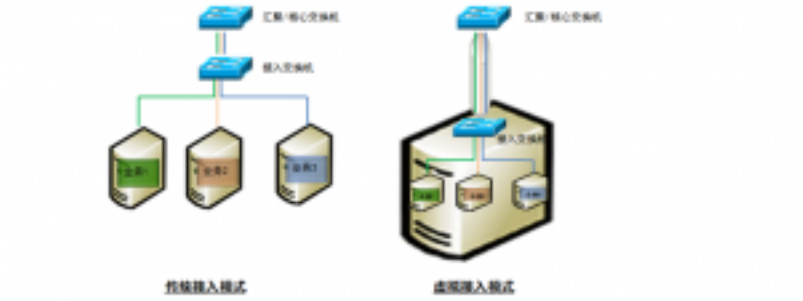
数据中心的虚拟接入是新一代数据中心的重点课题，各方已经争夺的如火如荼。目前网络上的中文资料还不多，根据自己的经验写了一点对虚拟接入的理解，意在丢砖，引出真正的大佬。

一、为什么虚拟化数据中心需要一台新的交换机

随着虚拟化技术的成熟和x86 CPU性能的发展，越来越多的数据中心开始向虚拟化转型。虚拟化架构能够在以下几方面对传统数据中心进行优化：

- 提高物理服务器CPU利用率；
- 提高数据中心能耗效率；
- 提高数据中心高可用性；
- 加快业务的部署速度

正是由于这些不可替代的优点，虚拟化技术正成为数据中心未来发展的方向。然而一个问题的解决，往往伴随着另一些问题的诞生，数据网络便是其中之一。随着越来越多的服务器被改造成虚拟化平台，数据中心内部的物理网口越来越少，以往十台数据库系统就需要十个以太网口，而现在，这十个系统可能是驻留在一台物理服务器内的十个虚拟机，共享一条上联网线。



这种模式显然是不合适的，多个虚拟机收发的数据全部挤在一个出口上，单个操作系统和网络端口之间不再是一一对应的关系，从网管人员的角度来说，原来针对端口的策略都无法部署，增加了管理的复杂程度。

其次，目前的主流虚拟平台上，都没有独立网管界面，一旦出现问题网管人员与服务器维护人员又要陷入无止尽的扯皮中。当初虚拟化技术推行的一大障碍就是责任界定不清晰，现在这个问题再次阻碍了虚拟化的进一步普及。

二不管的虚拟接入

接入层的概念不再仅仅针对物理端口，而是延伸到服务器内部，为不同虚拟机之间的流量交换提供服务，将虚拟机同网络端口重新关联起来。

二、仅仅在服务器内部实现简单交换是不能的

既然虚拟机需要完整的数据网络服务，为什么在软件里不加上呢？

没错，很多人已经为此做了很多工作。作为X86平台虚拟化的领导厂商，VMWare早已经在其vsphere平台内置了虚拟交换机vswitch，甚至更进一步，实现了分布式虚拟交互机VDS（vnetwork distributed switch），为一个数据中心内提供一个统一的网络接入平台，当虚拟机发生vmotion时，所有端口上的策略都将随着虚拟机移动。

VMWare干得貌似不错，实际上在当下大多数情况下也能够满足要求了。但如果谈到大规模数据中心精细化管理，内置在虚拟化平台上的软件交换机还有很多问题没有解决。首先，目前的vswitch至多只是一个简单的二层交换机，没有QoS、没有二层安全策略、没有流量镜像，不是说VMWare没有能力实现这些功能，但一直以来这些功能好像都被忽略了；其次，网管人员仍然没有独立的管理介面，同一台物理服务器上不同虚机的流量在离开服务器网卡后仍然混杂在一起，对于上联交换机来说，多个虚拟机的流量仍然共存在一个端口上。

虚拟平台上的软件交换机虽然能够提供基本的二层服务，但是由于这个交换机的管理范围被限制在物理服务器网卡之下，它没法在整个数据中心提供针对虚拟机的端到端服务，只有一个整合了虚拟化软件、物理服务器网卡和上联交换机的解决方案才能彻底解决所有的问题。

这个方案涉及范围如此之广，决定这又是一个只有业界大佬才能参与的游戏。

三、谁在开发新型交换机？

HP，Cisco。

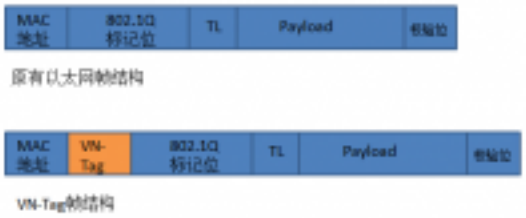
一个是PC服务器王者，近年开始在网络领域攻城略地，势头异常凶猛；一个是网络大佬，借着虚拟化浪潮推出服务器产品，顽强地挤进这片红海。

针对前文所说的问题，两家抛出了各自的解决方案，目的都是重整虚拟服务器同数据网络之间那条薄弱的管道，将以往交换机上强大的功能延伸进虚拟化的世界，从而掌握下一代数据中心网络的话语权。

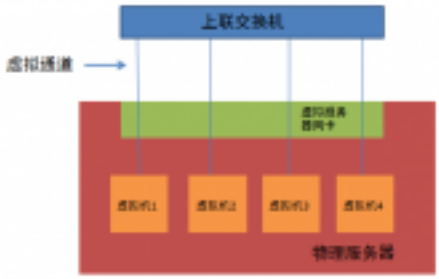
Cisco和VN-TAG

虚拟化平台软件如VMWare ESX部署之后，会模拟出一整套硬件资源，包括CPU、硬盘、显卡，以及网卡，虚拟机运行在物理服务器的内存中，通过这个模拟网卡对外交换数据，实际上这个网卡并不存在，我们将其定义为一个虚拟网络接口VIF（Virtual Interface）。VN-tag是由Cisco和VMWare共同提出的一项标准，其核心思想是在标准以太网帧中增加一段专用的标记—VN-Tag，用以区分不同的VIF，从而识别特定虚拟机的流量。

VN-Tag添加在目的和源MAC地址之后，在这个标签中定义了一种新的地址类型，用以表示一个虚拟机的VIF，每个虚拟机的VIF是唯一的。一个以太帧的VN-Tag中包含一对这样新地址dvif_id和svif_id，用以表示这个帧从何而来，到何处去。当数据帧从虚拟机流出后，就被加上一个VN-Tag标签，当多个虚拟机共用一条物理上联链路的时候，基于VN-Tag的源地址dvif_id就能区分不同的流量，形成对应的虚拟通道，类似传统网络中在一条Trunk链路中承载多条VLAN。只要物理服务器的上联交换机能够识别VN-Tag，就能够在交换机中直接看到不同的VIF，这一下就把对虚拟机网络管理的范围从服务器内部转移到上联网络设备上。



思科针对VN-Tag推出了名为Palo的虚拟服务器网卡，Palo卡为不同的虚拟机分配并打上VN-Tag标签，上联交换机与服务器之间虽然只有一条网线，但通过VN-Tag上联交换机能区分不同虚拟机产生的流量，并在物理交换机上生成对应的虚拟接口VEth，和虚拟机的VIF一一对应，好像把虚拟机的VIF和物理交换机的VEth直接对接起来，全部交换工作都在上联交换机上进行，即使是同一个物理服务器内部的不同虚拟机之间的流量交换，也通过上联交换机转发。这样的做法虽然增加了网卡I/O，但通过VN-Tag，将网络的工作重新交回到网络设备。而且，考虑到万兆接入的普及，服务器的对外网络带宽不再是瓶颈，此外，利用Cisco Nexus 2000这种远端板卡设备，网管人员还能够直接在一个界面中管理数百台虚拟机，每个虚拟机就好象在传统的接入环境中一样，直接连接到一个交换机网络端口。



目前，思科推出的UCS服务器已经能够支持VN-tag，当Palo卡正确安装之后，会对上层操作系统虚拟出多个虚拟通道，每个通道对应一个VIF，在VMWare EXS/ESXi软件中可以将虚拟机绕过vswitch，直接连接到这些通道上，而在UCS管理界面上则能够看到对应的虚拟机，使网管人员能够直接对这些端口进行操作。

Cisco同VMWare已经将向IEEE提出基于VN-Tag的802.1Qbh草案，作为下一代数据中心虚拟接入的基础。

HP和VEPA

Cisco提出的VN-Tag，在IT业界引起的震动远远大于在客户那得到的关注，如果802.1Qbh成为唯一的标准，Cisco等于再一次制定了游戏规则，那些刚刚在交换机市场上屯下重兵的厂商，在未来数据中心市场上将追赶得异常痛苦。此外，VN-Tag是交换机加网卡的一揽子方案，还能够帮助Cisco快速切入服务器市场，对其他人来说是要多不爽有多不爽。

很容易猜到，这其中最不爽的就是HP，在交换机和服务器领域跟Cisco明刀明枪地干上之后，被这样摆上一道，换谁也不可能无动于衷。HP的应对很直接，推出一个类似的方案，替代VN-Tag。

HP的办法称为VEPA（Virtual Ethernet Port Aggregator），其目的是在部署了虚拟化环境的服务器上实现同VN-tag类似的效果，但VEPA采取了一条截然不同的思路来搭建整个方案。

简单来说，VEPA的核心机制就是两条：修改生成树协议、重用Q-in-Q。

VEPA的目标也是要将虚拟机之间的交换行为从服务器内部移出到上联交换机上，当两个处于同一服务器内的虚拟机要交换数据时，从虚拟机A出来的数据帧首先会经过服务器网卡送往上联交换机，上联交换机通过查看帧头中带的MAC地址（虚拟机MAC地址）发现目的主机在同一台物理服务器中，因此又将这个帧送回原服务器，完成寻址转发。整个数据流好象一个发卡一样在上联交换机上绕了一圈，因此这个行为又称作"发卡弯"。

虽然"发卡弯"实现了对虚拟机的数据转发，但这个行为违反了生成树协议的一项重要原则，即数据帧不能发往收到这个帧的端口，而目前虚拟接入环境基本是一个大二层，因此，在接入层，不可能使用路由来实现这个功能，这就造成了VEPA的机制与生成

树协议之间的矛盾。

但是VEPA没有vPC，在接入层还是要跑生成树。HP的办法就是重写生成树协议，或者说在下联端口上强制进行反射数据帧的行为（Reflective Relay）。这个方式看似粗暴，但一劳永逸地解决了生成树协议和VEPA机制的冲突，只要考虑周全，不失为一步妙棋。

除了将虚拟机的数据交换转移到物理服务器上之外，VN-Tag还做了一项重要的工作，就是通过dvif_id和svif_id这对新定义的地址对不同虚机流量进行区分。HP在这里的搞法同样简单直接，VEPA使用Q-in-Q在基本的802.1q标记外增加了一层表示不同虚拟机的定义，这样在VLAN之外，VEPA还能够通过Q-in-Q区分不同的虚拟机，只要服务器网卡能够给数据帧打上Q-in-Q标记，上联交换机能够处理Q-in-Q帧，基本就可以将不同的虚拟机流量区分开来，并进行处理。

至此，VEPA看起来已近能够实现同VN-Tag类似的功能，因此HP也将VEPA形成草案，作为802.1Qbg的基础提交至IEEE。不得不说，VEPA是个非常聪明的设计，不管是对生成树行为的修改，还是利用Q-in-Q都不是什么不得了的创新，目前的交换机厂商只要把软件稍微改改，就能够快速推出支持802.1Qbg的产品，重新搭上数据中心这班快车，追上之前被Cisco甩下的距离。

VN-Tag和VEPA

自从Cisco祭出VN-Tag大旗后，各种争议就没停过，直到HP推出VEPA，这场口水仗达到高潮，随着2011年，802.1Qbh和802.1Qbg标准化进程的加快，围绕虚拟接入下一代标准的争夺将进入一个新的阶段。

这也不难理解，随着数据中心内虚拟机数量的不断增加，越来越多的物理网口转化为虚拟的VIF，如果一家网络厂商没法提供相应的接入解决方案，它的饼会越来越小，活得非常难受。

VN-Tag就是Cisco试图一统下一个十年数据中心的努力，HP虽然同思科正面开战时间不长，但从VEPA来看，其手法相当老辣。由于VEPA没有对以太网数据结构提出任何修改，实现成本非常低，以往被思科扫到大门之外的厂商，一下子见到了曙光，前仆后继地投靠过来，Juniper、IBM、Qlogic、Brocade等等都毫不掩饰对VEPA的期待，Extreme甚至表示，已近着手修改OS以保证对VEPA的支持。待各方站队结束，大家发现Cisco虽然有强大的盟友VMWare，但另外一边几乎集结了当今网络界的所有主流厂商，舆论也逐渐重视VEPA的优点，甚至Cisco自己也不得不松嘴说会考虑对802.1Qbg的支持。

戏演到这里，很多人幸灾乐祸地等着看Cisco怎么低头。但有一个问题，VEPA这么完美，为啥Cisco之前没有采用类似的思路？仅仅为构建一个封闭的体系架构吗？我认为不是。

回答这个问题前，我们首先要弄清楚另一个问题。以VMWare ESX/ESXi为例，由于ESX/ESXi自带的vswitch只是模拟了一台二层交换机，当一台物理服务器上两个处于不同VLAN的虚拟机之间需要交换数据时，vswitch是无能为力的。只能将数据送到上联物理交换机上，由物理交换机完成VLAN间的三层转发。听起来是不是很熟悉？这和之前提到的VN-Tag与VEPA的机制很相似，如果现有的虚拟化环境已经能够将数据交换的行为转移到上联交换机，为啥还要大费周折地提出一个新标准呢？

这是因为，当下的这种方案是利用VLAN来隔离不同虚拟机，通过TRUNK将对应多个虚拟机的VLAN送到物理交换机上。这种方式打破了数据中心内对VLAN的使用惯例，比如，网管人员通常会把负责同一业务的多台服务器放在一个VLAN内，如果VLAN标签都被用来隔离虚拟机了，则没法按照传统方式来区分不同业务，解决了一个问题，带来另外的问题，这是绝对行不通的。

现在，我们可以回答之前的问题了，新一代的虚拟接入方案是要在不影响802.1Q等原有网络行为的前提下，完成对虚拟机的接入、区分和管理。有人会说，用PVLAN不可以吗?但我们怎么保证PVLAN没有其他的用处呢？出于这样的思路，Cisco没有利用现有的任何技术，提出了一个全新的实现方案，正因为VN-Tag从出生起就“干干净净”，同谁都没有瓜葛，因此VN-Tag携带的信息就能够在整个数据中心内自由的传递，从而快速为用户搭建起一个清晰、完整的虚拟接入平台，所谓“磨刀不误砍柴工”。

HP充分利用了现有条件，VEPA的整个架构看上去简洁、高效，但是对生成树协议改动和利用Q-in-Q无疑会影响到现网的行为。生成树协议的效率和问题一直是个老大难，但无数聪明绝顶的高手琢磨了这么多年，协议的变动仍然不大，说明对这种基本协议的修改不是一蹴而就的，往往迁一发而动全局，现有的模式是各方协调、妥协的结果。VEPA要在短时间内拿出一个完美的方案，所需花费的精力也许并不比重新提一套方案少。

除了协议本身之外，摆在HP和VEPA面前还有两个难题，首当其冲就是VMWare的支持。VEPA虽然对交换机硬件改动不大，但要真正跑起来，还需要虚拟化平台软件的支持，虚拟网卡和虚拟交换机得主动把所有数据帧扔到上联交换机上，后面的故事才能续上。可是VMWare还是Cisco在VN-Tag上最大的盟友，虽然Cisco已经表示会支持802.1Qbg，但会有多及时就难说了。

时间也就是VEPA的第二个困难。目前，思科的UCS服务器已经能够提供端到端的VN-Tag部署。而HP的Virtual Connect解决方案仅实现了Q-in-Q的多链路，对“发夹弯”的支持并不好，也没有VMWare的支持，说白了，VEPA还只是图纸上的设计，没有实际产品支撑。此外，虚拟接入只是下一代数据中心组成之一，FCoE、THRILL等都非常重要，针对这些技术，HP仍拿不出成型的产品，相反，Cisco在所有领域几乎都布局完毕，留给HP的时间不多了。

这场针对数据中心接入的争夺，在2011年必将愈演愈烈，Cisco携全线产品势在必得，而HP的VEPA评价聪明的设计，得到业界广泛支持，故事结局如何，还待静观其变。

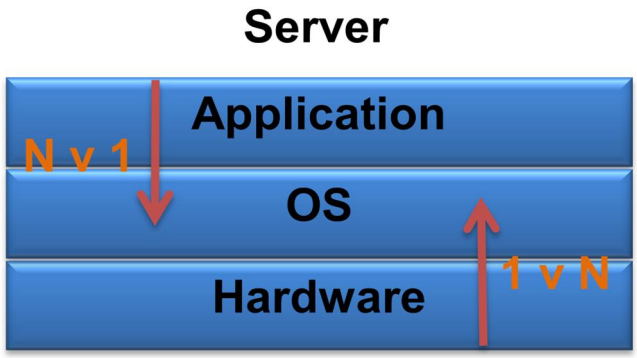
从半空看虚拟化

题解

打个比方，云计算是天上的云，虚拟化技术就是地上的湖泊水塘。以前也整理过一些对虚拟化和云计算技术的学习理解，感觉那时候就是站在云端，最多算是从云层中探个头出来瞅瞅下面，对虚拟化技术的理解似是而非。现在较之前有了一定进步，但也仍处在半空之中，期望有天真能落到水面上扑腾两下，但看来这辈子希望不大。在此向那些真正的泳者们致敬。

引言

书归正传，服务器虚拟化技术粗略的可分为一虚多和多虚一两大类。从下面这张图理下这两类技术的方向，注意这里提到的 Server 是针对云计算的 X86 服务器系统。

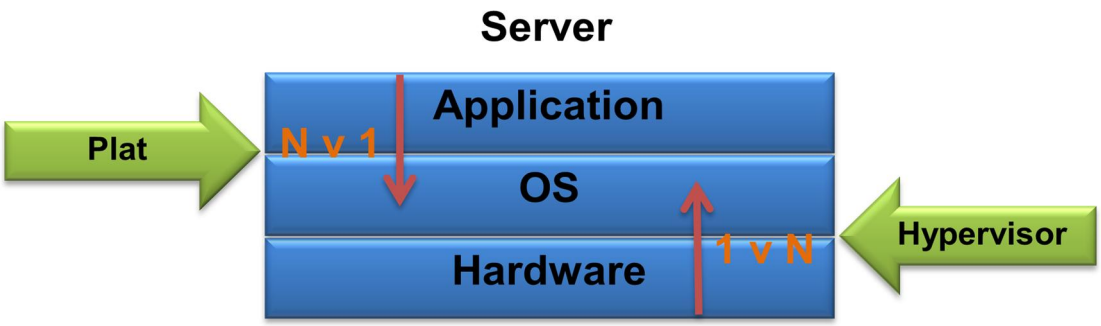


云计算的根本是服务器，服务器可简单分为硬件资源、操作系统和应用程序三个层面。一虚多技术主要面对硬件层面与操作系统层面，总的来说就是在一台物理服务器上虚拟出多套逻辑资源来运行多套操作系统。需注意前半句物理资源虚拟化只是手段不是重点，真正的目的是要搞多套能够同时运行且互不干扰的虚拟操作系统出来。这块技术是当前云计算 IaaS 的基础，也是本文后面要介绍的重点部分。

而多虚一技术主要针对应用程序层面，操作系统层面也有如 Cluster 和 Load Balance 等技术手段，但其重点是要在应用层面上将底层资源整合，达到对外统一服务的目的。上亿的用户在使用搜索引擎或浏览新闻的时候，实际的任务是由后端成千上万的服务器完成，但对我们这些使用者来说，看到的就是那几个应用页面。

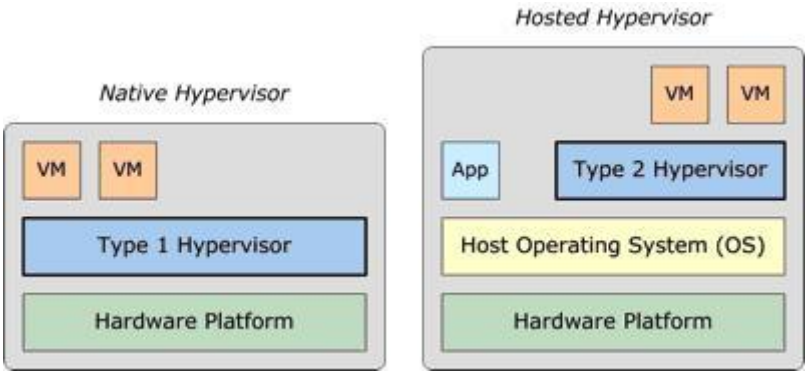
当我们想要在尽量少改动甚至不改动原有对象的条件下，对整体结构做调整时，可以在其中间插入新的一个对象，专门来处理结构调整与任务调度。虚拟化技术很好的阐述了

这种设计思路，如下图中 Plat 位置对应的 Hadoop 技术和 Hypervisor 位置对应的 VMware ESX/ESXi、Xen、Linux KVM 以及 Microsoft Hyper-V 等技术。



相对来说，底层的一虚多技术更加收敛一些，上层的多虚一技术则要更加开放，毕竟主流的硬件和操作系统就那么几种，而应用软件数不胜数。因此从简单的入手，本文主要分析的是基于硬件到操作系统层面的一虚多技术，对应云计算中的 IaaS 部分。

传统上有两种类型的 Hypervisor 应用如下图所示，本文主要针对的是 Type1，但个人理解这两种类型从技术上讲区别并不是很大，Hypervisor 不可能脱离 OS（Operating System 操作系统）独立去进行硬件的处理， Type1 中的 Hypervisor 也包含了一个最简单的 OS 内核，无非就是在这个内核上不能跑其他的应用罢了。因此可以将 Type1 的 Hypervisor 理解为 Mini OS+VMM(Virtual Machine Manager)，而 Type2 的 Hypervisor 理解为单独的 VMM。最新的主流 Hypervisor 产品之一 KVM 已经打破了这种简单划分，我们无法将其明确的定位为 Type1 或 Type2，因此以后对虚拟化技术应该会有更加先进准确的分类方式出现。后面的文章中会再详细的分析当前主流 Hypervisor 产品中 OS 与 VMM 之间的关系架构。



正文以介绍服务器相关虚拟化技术为主，按照硬件、Hypervisor 和 OS 三个层面自底向上展开，其中会针对典型的技术与产品进行分析讨论。

本文的主要目的是帮助那些在云上或者云中的同学们往下降一降。但毕竟作者现在也还是在半空中，因此很多内容无法讲细讲透，甚至可能会有一些理解误区，仅供参考，欢迎指正。

虚拟化技术介绍

虚拟化管理平台

讲虚拟化技术之前，先提两句虚拟化管理平台。

当通过虚拟化技术将物理资源模拟为逻辑资源后，还需要一个平台将其统一管理起来，并通过接口提供给终端用户使用，这个管理平台也是 IaaS 必不可少的另一支柱。从用户角度出发，申请虚拟机时最基本的四个要素是 CPU、内存、硬盘和带宽，其中 CPU 和内存代表了虚拟机的计算能力，属于单台硬件服务器一虚多分离出来的子集（你不可能要求一个 2 核的虚拟机使用的是两台物理服务器上各自分出来的一个核，同样也不可能同时使用多台物理服务上的物理内存来构建单台虚拟机的虚拟内存）；硬盘则代表数据存储能力，这个可以通过 NAS/SAN 等技术连接到磁盘阵列上来无限扩充；带宽则是网络服务能力的表现，同时网络还需要为不同租户的虚拟机之间提供隔离控制能力，为相同租户的虚拟机之间提供通道互访能力，而这些虚拟机可能在相同或不同的物理服务器上。

虚拟化管理平台并不具备什么先进的技术内容，但是其在推动云计算发展进程中占据着重要地位。如果说虚拟化技术是树根，则虚拟化管理平台可以看作树干，云计算这棵大树想要枝繁叶茂则二者缺一不可。虚拟化管理平台通过对硬件资源进行统筹管理，形成统一的资源池供用户使用。用户可以定制并管理自己的虚拟机和软件程序，不需要经过管理员和设备厂商的中间传递，其需求能够更加准确简便的得到实现。

再多废话两句，以 Apple 的例子来说明一下平台为王的现状，Apple Phone 的具体技术其实都算不上先进，无论是触屏还是模块化应用都是很早就出现的技术，真正帮助其王者归来的是 Apple Store。通过这种新的市场模式，手机成为了一个平台，人们看重的不再是其硬件功能本身，而是聚焦于其上的应用程序。在传统的手机制造销售中，程序的开发者与实际用户之间隔着厂商这堵墙，对开发者而言厂商让编什么就编什么，而使用者则只能是厂商给什么就用什么。一款手机产品的成功与否只能依赖于厂商对用户使用喜好的猜测和硬件与价格等低级竞争手段。Apple Store 则打破了那堵墙，用户可以直面开发者去选择喜爱的程序，开发者也可以根据下载量和排行榜更准确的摸清用户喜好，从而进行有针对性的开发，并直接获益。从目前来看唯一会对 Apple 造成发展制约的就是其硬件了，当满世界人们都举着 Apple Phone 时，审美疲劳会导致大家倾向于选择更新鲜更与众不同的款式。如果 Apple 将来的 IOS 可安装于任意手机硬件之上，那么现在

如日中天的 Apple 帝国也将如 Microsoft 般延续出属于自己的不朽王朝。另外，今日的 IOS 与 Android 之争与 90 年代的 Windows 与 Linux 之争是多么相似，同样从那场纷争可以看出，IOS 今后的主要对手不会是 Android，而只有 Microsoft 的 WP。

硬件层面虚拟化

首先来说硬件，一虚多的根本目的就是把一台物理服务器虚拟成多台虚拟服务器来使用。就服务器的 CPU、内存、存储和网卡等核心物理资源来说，真正的虚拟化难点在于 CPU。因此下面以 CPU 虚拟化技术介绍为主，还会简单说下北桥芯片和网卡相关的虚拟化技术。

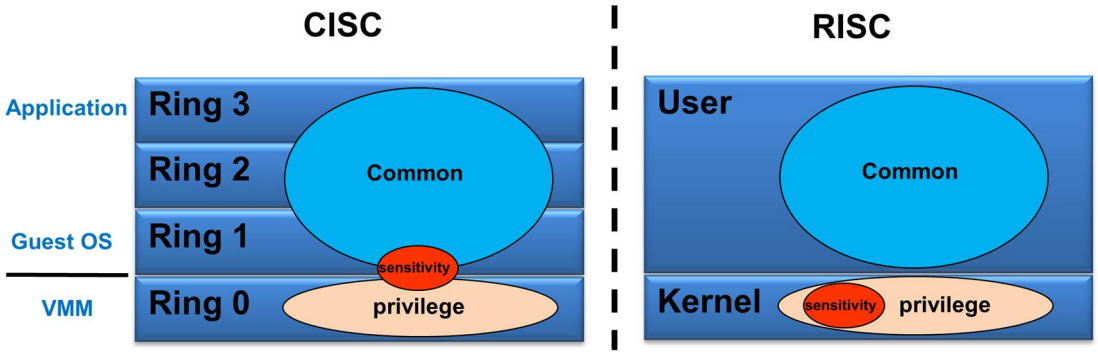
CPU 虚拟化技术

现代计算机的 CPU 技术有个核心特点，就是指令分级运行，这样做的目的是为了避免用户应用程序层面的错误导致整个系统的崩溃。需要注意这里“指令分级”中的级别专门指 CPU 指令运行级别，而和我们操作系统里面的进程运行优先级没有关系。不同类型的 CPU 会分成不同的级别，如 IBM PowerPC 和 SUN SPARC 分为 Core 与 User 两个级别，MIPS 多了个 Supervisor 共三个级别。本文针对的 X86 系统则分为 Ring0-Ring3 共 4 个级别，而在这里我们不需要考虑那么多，只需关注核心级别（Ring0）和用户级别（Ring1-Ring3）两个层面即可。

对于非虚拟化的普通操作系统而言，应用程序和系统下发的普通指令都运行于如 Ring1 到 Ring3 的用户级别中，只有特权指令会运行在如 Ring0 的核心级别中。而当进行虚拟化之后，实质上出现了两层操作系统，底层 OS（VMM 或 Hypervisor，下文都以 VMM 简称）负责虚拟机（VM）到硬件资源的调用和协调，所有的业务应用都运行在上层 OS（Guest OS）中。

在非 X86 系统中经典的虚拟化做法是令 VMM 拥有超级特权，其指令都运行在类似 Ring0 的核心级别；令 Guest OS 运行在类似 Ring1 到 Ring3 的用户级别，取消其特权指令在核心级别直接运行的权利。当 Guest OS 运行到特权指令时，这些指令产生异常并被 VMM 捕获到，VMM 会在核心级别中模拟执行，然后再将运行结果返回给 Guest OS。这种经典的虚拟化技术被称为特权解除和陷入模拟，以 IBM 的 Power 系列为代表。

我们将操作系统中会涉及系统底层公共资源调用的一些运行指令称为敏感指令，显然这些敏感指令在虚拟化的结构中也需要陷入到 Ring0 的核心级别执行，否则会导致不同 Guest OS 之间的资源调用冲突。大型服务器如 PowerPC 和 SPARC 运行的 RISC 指令集中，所有的敏感指令都属于特权指令，因此可以采用上面说的特权解除和陷入模拟技术完美的进行虚拟化实现。但对于 X86 的 CISC 指令集而言，存在 17 条非特权指令的敏感指令，这些指令被 Guest OS 在 Ring1 级别执行时，会被直接执行，无法产生异常从而陷入 Ring0 处理，也就导致无法采用经典技术进行虚拟化，因此下文将介绍的一系列方案都是为了解决此问题而设计的。



在上述问题中，涉及到三个主要对象，Guest OS、VMM 和硬件 CPU 的指令集，其中 VMM 是新插入的对象，修改起来很方便，但 OS 和 CPU 改起来就难一些了。解决方案的思路也由此分为三个方向：

1、 只变动 VMM。好处是兼容性最强，OS 和 CPU 都不用动，但效率肯定是最低的。这种方案也被称为 CPU Full-Virtualization。
2、 改动 Guest OS。好处是效率较高，但缺点是 Windows 肯定不愿意干，只能在 Linux 上做些文章，而且使用特制的 OS，会带来一些可扩展性方面的隐患。这种方案也被称为 CPU Para-Virtualization。
3、 改动 CPU 指令集。这个改动就只有 Intel/AMD 能做了，好处是对 Guest OS 可以不需变动，兼容 Linux 和 Windows，VMM 的使用效率也较高。缺点也有，就是增加了一些虚拟化指令和结构，导致对 CPU 的利用率下降，在部分应用场景下的性能表现不如前面的 CPU Para-Virtualization 方案。这种方案也被称为硬件辅助虚拟化技术 HVM（Hardware-assisted Virtualization Machine）。随着 Intel/AMD 的服务器 CPU 全部更新换代对其提供支持，HVM 已经成为当前虚拟化技术应用的主流。

CPU Full-Virtualization

首先说 CPU Full-Virtualization，这种思路又被细化分为三种主要方案 Emulation、Scan-and-Patch 和 Binary Translation。其中 Emulation 是根本解决方案，而 Scan-and-Patch 和 Binary Translation 可以理解为是 Emulation 在 X86 体系上使用的扩展实现方案。CPU Full-Virtualization 由于实现较为简单，早在上世纪末就已经出现，是最早期的 X86 虚拟化技术。

基本的 Emulation 主要应用在跨平台进行虚拟化模拟，Guest OS 与底层系统平台不同，尤其是指令集区别很大的场景，比如在 X86 系统上模拟 PowerPC 或 ARM 系统。其主要思路就是 VMM 将 Guest OS 指令进行读取，模拟出此指令的执行效果返回，周而复始，逐条执行，不区分用户指令和敏感指令，由于每条指令都被通过模拟陷入到 Ring0 了，因此也就可以解决之前的敏感指令问题。代表产品就是 Linux 上的 QEMU 和 Bochs，其中 QEMU 可以通过开源加速器软件 KQEMU 来提升处理速度。它们作为开发工具，在嵌入式平台的开发工作中应用较多。

Scan-and-Patch 主要思路是将 Guest OS 的每个指令段在执行前先扫描一遍，找出敏感指令，在 VMM 中生成对应的补丁指令，同时将敏感指令替换为跳转指令，指向生成的补丁指令。这样当指令段执行到此跳转时会由 VMM 运行补丁指令模拟出结果返回给 Guest OS，然后再顺序继续执行。代表产品是 Oracle 的开源虚拟化系统 VirtualBox，目前主要应用于在主机上进行虚拟机的模拟，服务器使用较少。

Binary Translation 主要思路是将 Guest OS 的指令段在执行之前进行整段翻译，将其中的敏感指令替换为 Ring0 中执行的对应特权指令，然后在执行的同时翻译下一节指令段，交叉处理。代表产品为 VMware Workstation、Microsoft Virtual PC 主机虚拟化工具，以及早期 VMware 的 ESX/GSX 系列服务器虚拟化系统，目前的服务器上已经很少使用了。

CPU Full-Virtualization 受性能影响，在服务器上目前被逐渐淘汰。主要代表产品如 VirtualBox 和 VMware Workstation 大都应用于主机虚拟化的一些开发测试环境中。有个了解即可，可不必对其技术进行深究。只有 QEMU 作为基础虚拟化技术工具，在其他的虚拟化产品中被广泛实用。

CPU Para-Virtualization

Para-Virtualization 有半虚拟化、类虚拟化和超虚拟化等多种译法，这也可以看出人们对此技术褒贬不一的矛盾心理。此技术以 Xen 和 Hyper-V 为代表，但 VMware 的 ESX Server 和 Linux 的 KVM 两种当前主流虚拟化产品也都支持 Para-Virtualization，另外还有基于 Linux 的 Lguest 和基于 Windows 的 CoLinux 等各具特色的小型产品也都通过部分 Para-Virtualization 技术来实现操作系统的虚拟化。Para-Virtualization 技术实际上是一类技术总称，下面先要谈的是 CPU 的 Para-Virtualization（CPU PV）。

CPU PV 技术实现的主要原理如下，首先 VMM 公布其一些称为 Hypercalls 的接口函数出来，然后在 Guest OS 中增加根据这些接口函数修改内核中的代码以替代有问题的 17 条敏感指令执行系统调用操作。修改后的指令调用通常被称为 Hypercalls，Guest OS 可以通过 Hypercalls 直接调用 VMM 进行系统指令执行，相比较前面提到的陷入模拟方式极大的提升了处理效率。

然而 CPU PV 修改操作系统内核代码的方式带来了对 Guest OS 的很多使用限制，如只有 Hyper-V 可以支持 Para-Virtualization 方式的 Windows Server 作为 Guest OS，另外由于 KVM/Xen/VMware VMI/Hyper-V 各自 Hypercalls 代码进入 Linux 内核版本不同，因此采用 Linux 作为 Guest OS 时也必须关注各个发行版的 Linux 内核版本情况。KVM 是 2.6.20，VMI 是 2.6.22，Xen 是 2.6.23，Hyper-V 是 2.6.32。

CPU PV 方式由于对 Guest OS 的限制，应用范围并不很广，但由于其技术上的系统调用效率提升，仍然被部分开发与使用者所看好，在某些特定场景中也存在一定需求。另外由于 2002 年 Xen 既有采用 CPU PV 的产品面世，目前部分大型的数据中心服务器已经进行了部署，基于技术成熟度和利旧的考虑，短时间内仍然会拥有一定的生存竞争能力。

硬件辅助虚拟化技术 HVM

前面的 Full-Virtualization 和 Para-Virtualization 两种方案是在 VMM 和 Guest OS 的软件层面解决 CPU 的敏感指令问题，而最后这个方案则是从硬件出发，由 CPU 自己搞定。当前 X86 的 CPU 厂商就是 Intel 和 AMD 两家，个人觉得从虚拟化技术发展上来看，AMD 更多的属于跟随者的角色。Intel 推出的 CPU 虚拟化技术是在 Xeon CPU 上的 VT-x，和 Itanium CPU 上的 VT-i，AMD 的类似技术则是 AMD-V。下面以 VT-x 为例对 HVM 技术进行剖析，其他两种类似也就不多说了。

VT-x 在 CPU 操作方面有两个主要特性，一是增加了 VMCS（Virtual-Machine Control Structure）数据结构和 13 条专门针对 VM 的处理指令，用于提升 VM 切换时的处理效率，二是通过引入 Root/Non-Root 操作模式解决前面遇到的 Guest OS 敏感指令无法陷入问题。

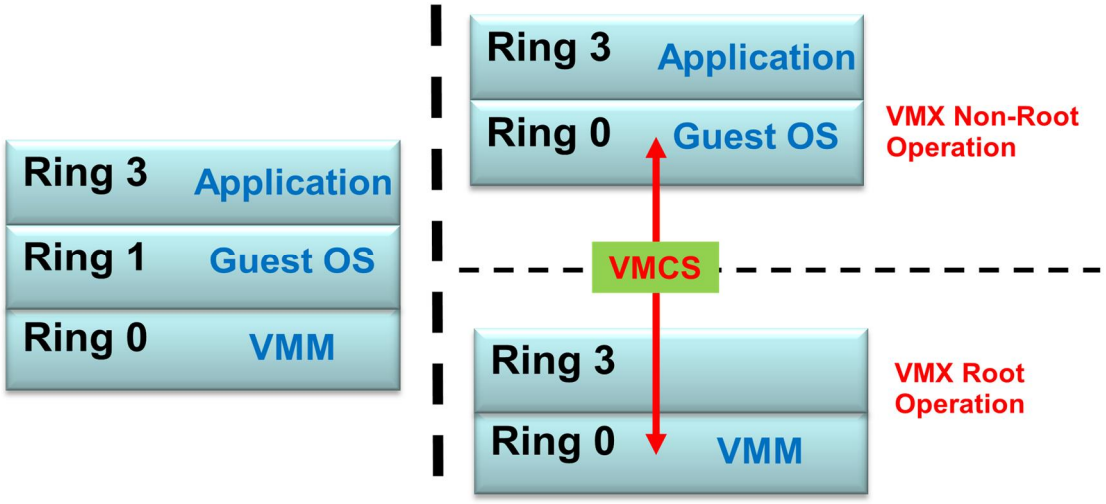
13 条新指令中包括 5 条用于 VMCS 维护，4 于 VMX 管理，2 条用于 VMX 对 TLB（Translation Look aside Buffer）的管理，2 条用于 Guest OS 调用。详见下表，说的啥大家应该也都能看明白，犯下懒就不翻译了。

Behavior	Instruction	Description
VMCS-maintenance	VMPTRLD	This instruction takes a single 64-bit source operand that is in memory. It makes the referenced VMCS active and current, loading the current-VMCS pointer with this operand and establishes the

		current VMCS based on the contents of VMCS-data area in the referenced VMCS region. Because this makes the referenced VMCS active, a logical processor may start maintaining on the processor some of the VMCS data for the VMCS.
	VMPTRST	This instruction takes a single 64-bit destination operand that is in memory. The current-VMCS pointer is stored into the destination operand.
	VMCLEAR	This instruction takes a single 64-bit operand that is in memory. The instruction sets the launch state of the VMCS referenced by the operand to “clear”, renders that VMCS inactive, and ensures that data for the VMCS have been written to the VMCS-data area in the referenced VMCS region. If the operand is the same as the current-VMCS pointer, that pointer is made invalid.
	VMREAD	This instruction reads a component from the VMCS (the encoding of that field is given in a register operand) and stores it into a destination operand that may be a register or in memory.
	VMWRITE	This instruction writes a component to the VMCS (the encoding of that field is given in a register operand) from a source operand that may be a register or in memory.
VMX management	VMLAUNCH	This instruction launches a virtual machine managed by the VMCS. A VM entry occurs, transferring control to the VM.
	VMRESUME	This instruction resumes a virtual machine managed by the VMCS. A VM entry occurs, transferring control to the VM.
	VMXOFF	This instruction causes the processor to leave VMX operation.
	VMXON	This instruction takes a single 64-bit source operand that is in memory. It causes a logical processor to enter VMX root operation and to use the memory referenced by the operand to support VMX operation.
VMX-specific	INVEPT	This instruction invalidates entries in the TLBs and paging-structure caches that were derived from extended page tables (EPT).

TLB-management	INVVPID	This instruction invalidates entries in the TLBs and paging-structure caches based on a Virtual-Processor Identifier (VPID).
Guest-available	VMCALL	This instruction allows software in VMX non-root operation to call the VMM for service. A VM exit occurs, transferring control to the VMM.
	VMFUNC	This instruction allows software in VMX non-root operation to invoke a VM function (processor functionality enabled and configured by software in VMX root operation) without a VM exit.

Root/Non-Root 操作模式将原有的 CPU 操作区分为 VMM 所在的 Root 操作与 VM 的 Non-Root 操作，每个操作都拥有 Ring0-Ring3 的所有级别。通过 VMCS 数据结构指向 Guest OS 到 VMM 的操作切换。



VMM 在初始化的时候会在物理内存中为每个 VM 开辟一段空间用于存储对应的一个 VMCS 数据结构，可以理解一个 VMCS 就对应一个 vCPU。VM 切换时，VMM 可以通过 VMCS 指针方便的在不同 VMCS 间进行跳转，有效的提升了 CPU 使用效率。

VT-x 技术还包含以下三个重要内容：

1、 EPT（Extended Page Table）：由 VMM 控制的一种新页表结构，主要目的为了在保证多个 VM 访问物理内存相互隔离的同时，提升大块内存的读写效率。

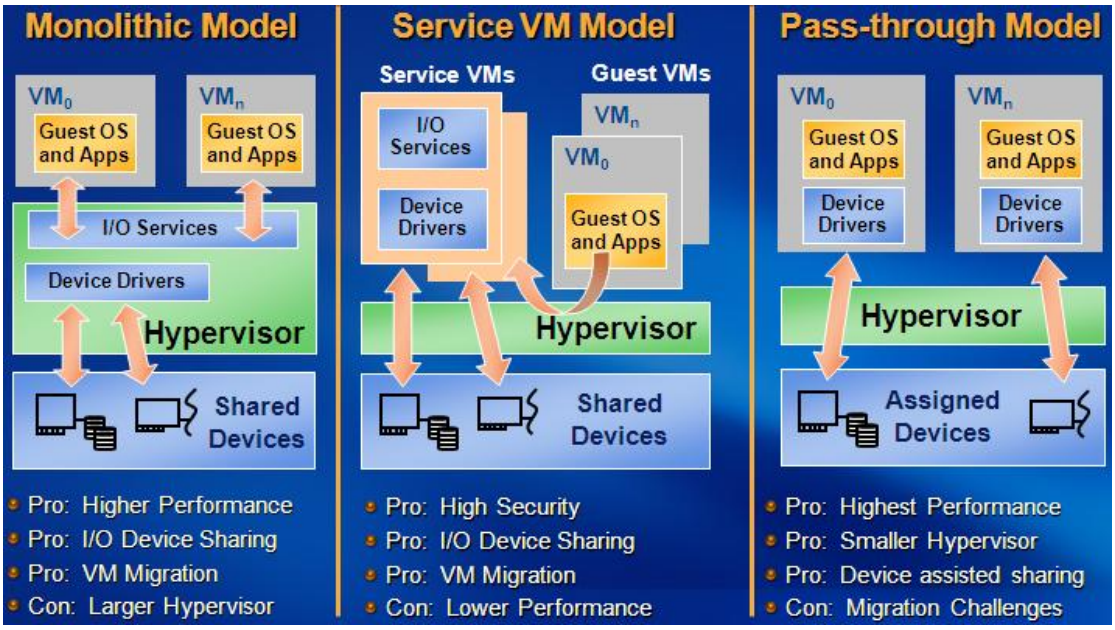
2、 FlexMigration：灵活迁移技术，用于 VM 在使用不同型号 CPU 的物理服务器间进行迁移，促成了类似 vMotion 的 VM 迁移技术的发展。

3、 FlexPriority：灵活优先级技术，用于优化 CPU 中断处理过程中对 TPR（Task-Priority Register）的使用，提高中断处理效率。

VT-x 和 AMD-V 等技术的出现，解决了前面两种纯软件方案进行 X86 虚拟化时，CPU Full-Virtualization 性能低和 Para-Virtualization 的 Guest OS 兼容性差问题。随着服务器 CPU 两三年一换代的更新速度，当前的主流 X86 服务器已经都可以支持 VT-x/AMD-V 等技术，因此 HVM 成为当前云计算 IaaS 服务器虚拟化的主流。主要的几款 VMM 产品 Xen/VMware ESXi/KVM/Hyper-V 都已经能够支持 HVM 功能。

北桥芯片 Chipset 虚拟化技术

北桥在服务器中主要是用来连接 CPU、内存、AGP 和南桥的各种 I/O 设备，处理 DMA 与中断请求，北桥芯片的虚拟化技术是为了支持 I/O 虚拟化技术。先介绍下常见的 I/O 虚拟化方案，有以下三种。



第一种将 I/O 服务和设备驱动都直接装载在 Hypervisor 的 Mini OS 上，优点是性能较高，缺点是 Hypervisor 会变得很大，代表产品就是 VMware 的 ESX 和 ESXi，最新的 ESXi 5.0 安装包简化后也达到 290M，另外一个缺点就是对新设备的驱动加载不够方便，有时需要升级整个 Hypervisor，比如 ESXi 直到 5.0 版本才能识别出 Qlogic 驱动的 CNA 网卡。如果希望采用此种方案，一定要先确认好 Hypervisor 的支持驱动列表再进行安装。

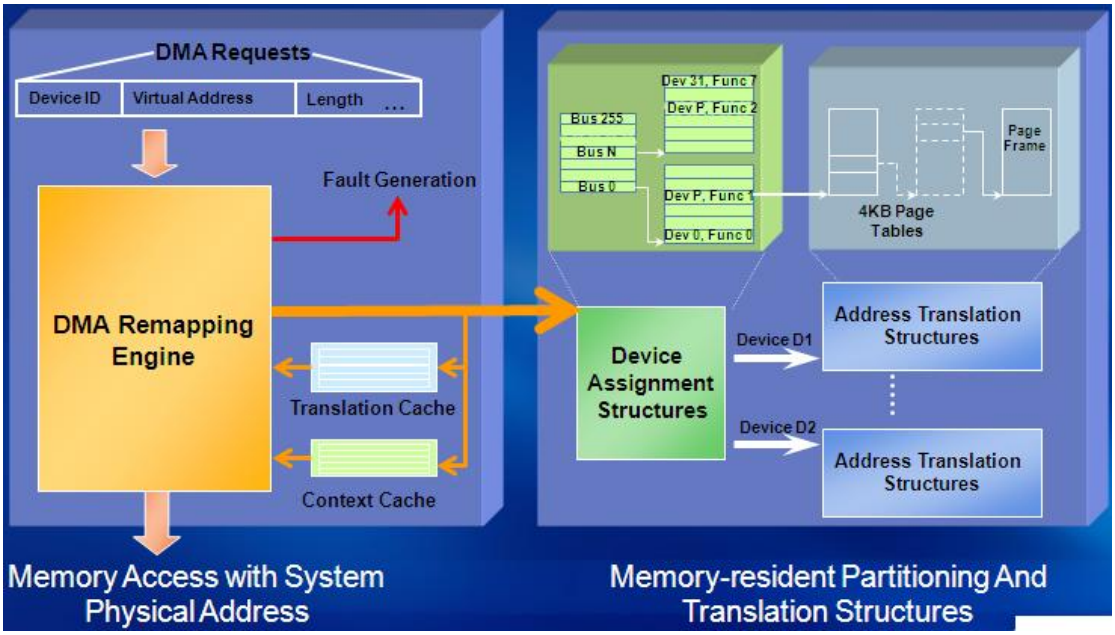
第二种是通过加载一个服务系统（如 Domain0）来减小 Hypervisor 的负担，所有的产品驱动都安装在服务系统上，这样便于其他外设硬件的管理和增加变化，同时 Hypervisor 越小意味着越安全，降低了故障的风险。缺点是资源调用需要经过服务系统，性能会稍低。典型代表就是 Xen 和 Hyper-V。KVM 有些类似，但不完全一样，因为其 Hypervisor 和服务系统使用了同一套 Linux 内核，而 Xen/Hyper-V 的 Hypervisor 各使用了一套最简化的新内核，与服务系统不同，下文会有更清楚的介绍。

另外这里还有个概念需要说明，前面两种方式在一些公开资料中也被称为 Full-Virtualization（FV）和 Para-Virtualization（PV），但个人认为这个是 I/O 的虚拟化，而 FV 和 PV 的更主要区别是在于 CPU 的虚拟化技术。CPU 的 FV/PV 和 I/O 的 FV/PV 并不是紧密联系的，例如早期的 VMware ESX 使用 CPU FV + I/O FV，早期的 Xen 使用 CPU PV + I/O PV，但在 HVM 技术出现之后，其各自主流结构已经变成 VMware ESXi 使用 VT-x + I/O FV，Xen 使用 VT-x + I/O PV 了，所以目前的 Xen 也都没有必须使用修改过内核的 Guest OS 的限制了。

第三种则是从硬件层面已经将各个设备进行了划分，分配给不同的 VM 使用，由每个 VM 自行维护其使用的 I/O 和设备驱动。Hypervisor 只提供通道，不再对 I/O 和设备驱动进行管理。这样做的好处是可以达到最高的性能和最小的代码量（Hypervisor），但缺点是设备非完全共享且对 VM 迁移提出了一定挑战。这种方案只有在对 I/O 操作要求较高的特定场景下使用，不具备普适性。部署时 I/O 设备支持逻辑划分技术如 SR IOV（Single Root I/O Virtualization）等即可，对 Hypervisor 没有什么特殊要求，前面说的几款主流产品都提供了此种工作方式。

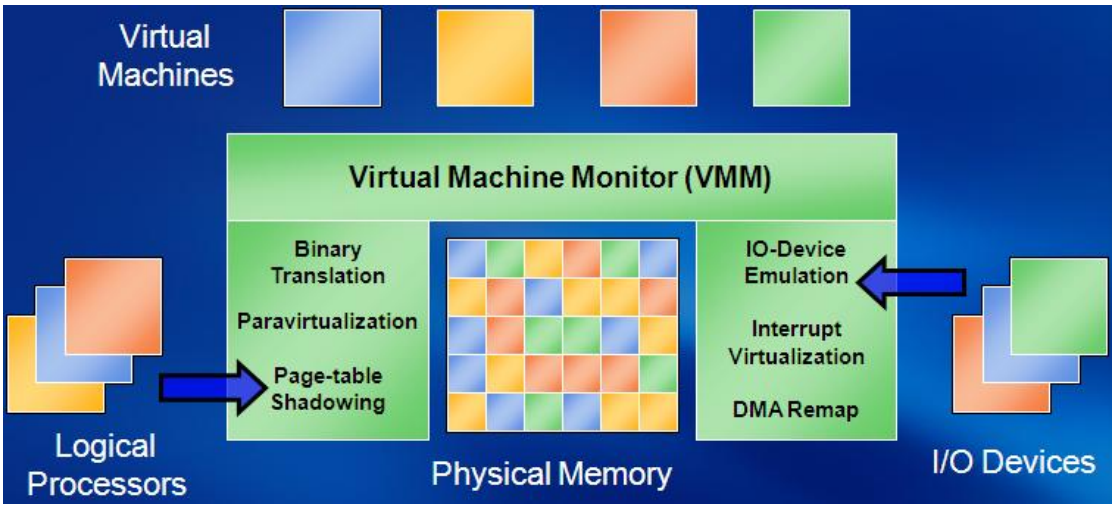
Intel 的 VT-d 技术作为一种平台结构在硬件层面可以有效的对上面三种方案全部提供支持。其根本目的就是为了多个 VM 在对虚拟设备进行 I/O 操作时，能够对其 DMA 和中断处理进行隔离保护与提升性能。

VT-d 的主要内容就是 DMA Remapping，用下面这张网上摘来的截图帮助有兴趣的同学进行理解，作者水平有限，细节就不解释了。

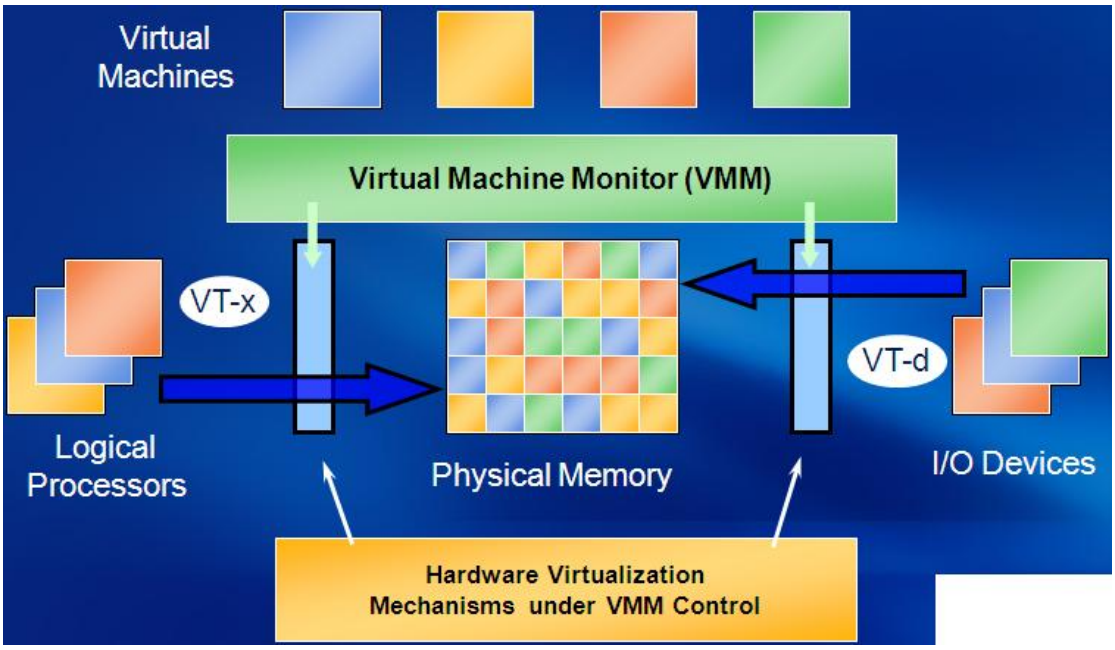


VT-x 和 VT-d 等技术内容很多，细说起来可以整理很多内容，本文深度有限，就不做详细探讨了，最后再用两张图来比较 VT-x 和 VT-d 出现前后的虚拟化结构区别。

软件虚拟化结构



HVM 结构



（上述四幅截图均来自于 Microsoft WinHEC 2006 大会宣讲胶片资料）

网卡虚拟化技术

服务器内部如 CPU、内存、芯片等设备虚拟化后都是希望让不同的 VM 能够达到资源共享，使用隔离的效果，但网络方面就有所区别了，除隔离外还得考虑一个 VM 互联互通的问题。

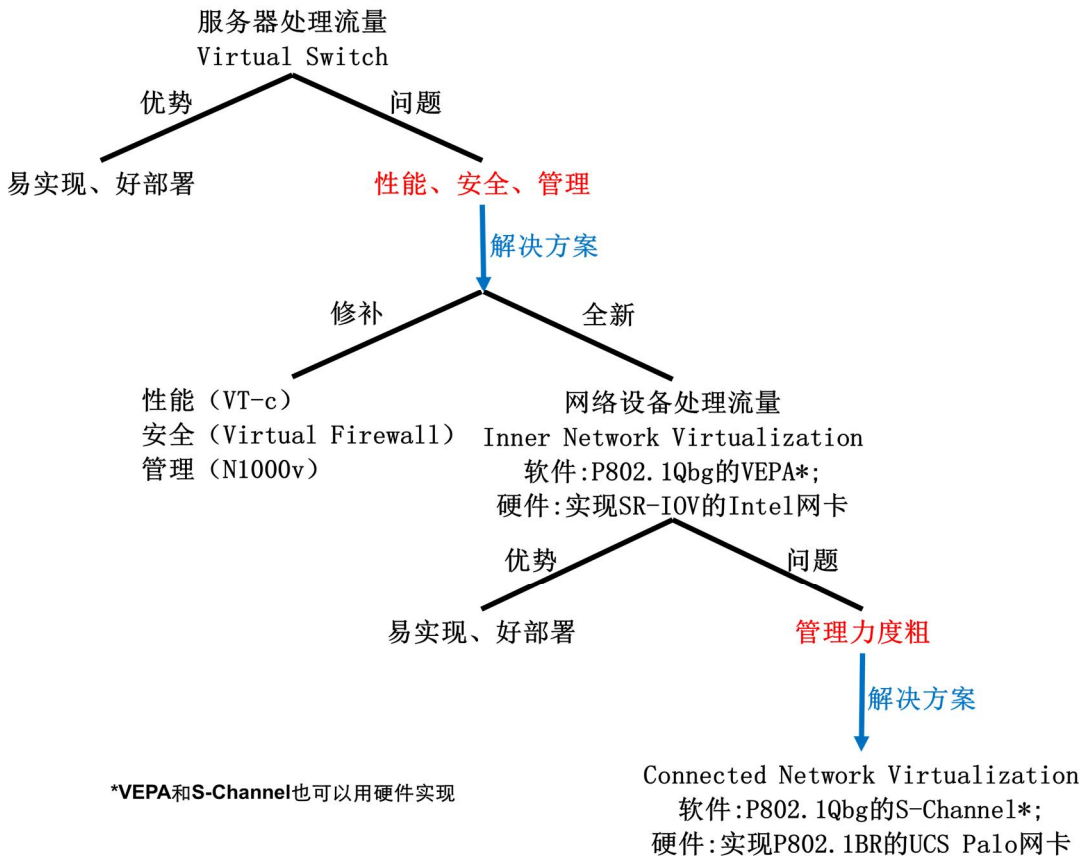
每个 VM 发出的流量会有两种去向，类型一流量是去往物理服务器外面的世界，这部分流量好办，按照其他设备的思路进行虚拟化隔离使用，给每个 VM 一个专用通道，共用物理网卡资源即可。VM 还有可能发出访问本物理服务器内部其他 VM 的类型二流量，

这部分流量目前的主流处理方式是在 Hypervisor 建立一个 Virtual Switch，内部软件交换，不需要走到服务器外面去。另外一个将要实现的思路是物理服务器内部仍然按照流量类型一的方式去处理，将流量隔离后都扔到服务器外面，由外界的网络设备确认流量终点位置，如果目的地址属于同一物理服务器的其他 VM，则通过相同的路径返回给服务器即可。

Virtual Switch 思路有两个主要问题，一是性能很差，二是对 VM 来说类型一与类型二流量通常是会同时存在的，而 Virtual Switch 在处理类型二流量的时候会存在较大的安全性问题。但由于其易于实现，因而 Virtual Switch 成为了当前使用的主流技术，几款主要的 Hypervisor 都有自己的 Virtual Switch，再如 IEEE P802.1Qbg 定义了 VEB（Virtual Ethernet Bridge），Cisco 也推出独立 Virtual Switch 产品 N1000v。同时为了解决前面所说的性能与安全问题，其他厂商也陆续做出了一些新的东西对其修修补补，像 Intel 的 VT-c 使用 VMDq（Virtual Machine Device Queues）技术通过网卡芯片硬件处理一些 Virtual Switch 的工作，这样可以有效提高 VM 数据交换性能，部分安全厂商则推出了 Virtual Firewall 产品来提高 Hypervisor 内部层面的数据转发安全性。

网络设备处理的 VM 互访流量思路从服务器来看也分为两种处理方式，一是只在服务器内部对 VM 的流量进行区分识别，从物理网卡扔出去时就是普通报文，外部网络设备根据 MAC/IP 再查表转发，不了解服务器内部的任何 VM 网络信息，作者给起个名字叫做 Inner Network Virtualization（INV）。其技术代表，软件的是 P802.1Qbg 中的 VEPA（Virtual Ethernet Port Aggregation），硬件的是 PCI-SIG 组织推的 SR-IOV（Single Root I/O Virtualization）标准。而第二种处理方式就是服务器将 VM 发到外面的流量都加个 Tag，外部网络设备根据此 Tag 识别来自不同 VM（的 vNIC）的流量进行细化处理。作者再给起个名字叫做 Connected Network Virtualization（CNV）。（只为了描述方便，作者可不是什么命名控）其技术代表，软件的是 P802.1Qbg 的 S-Channel（Multichannel），硬件的就是 Cisco UCS 服务器中 Palo 网卡支持的 P802.1BR 技术。由于 CNV 明显是 INV 的扩展，也可以说是为了解决 INV 控制力度太粗的问题而出的方案，所以从技术产品上来看，S-Channel 是 VEPA 的超集，而 Palo 网卡实现 P802.1BR 时也用到了 SR-IOV 的内容。

通过下面的图表，希望能够帮助读者更清晰的理解上述文字内容。



有些跑题，真正和本章主题“网卡硬件虚拟化技术”相关的内容其实主要就是 VT-c 和 SR-IOV 两项，一个是 Intel 实现的，一个是 Intel 主导的，但研究不深这里也不再展开了，有兴趣的同学请自行查阅相关资料吧。

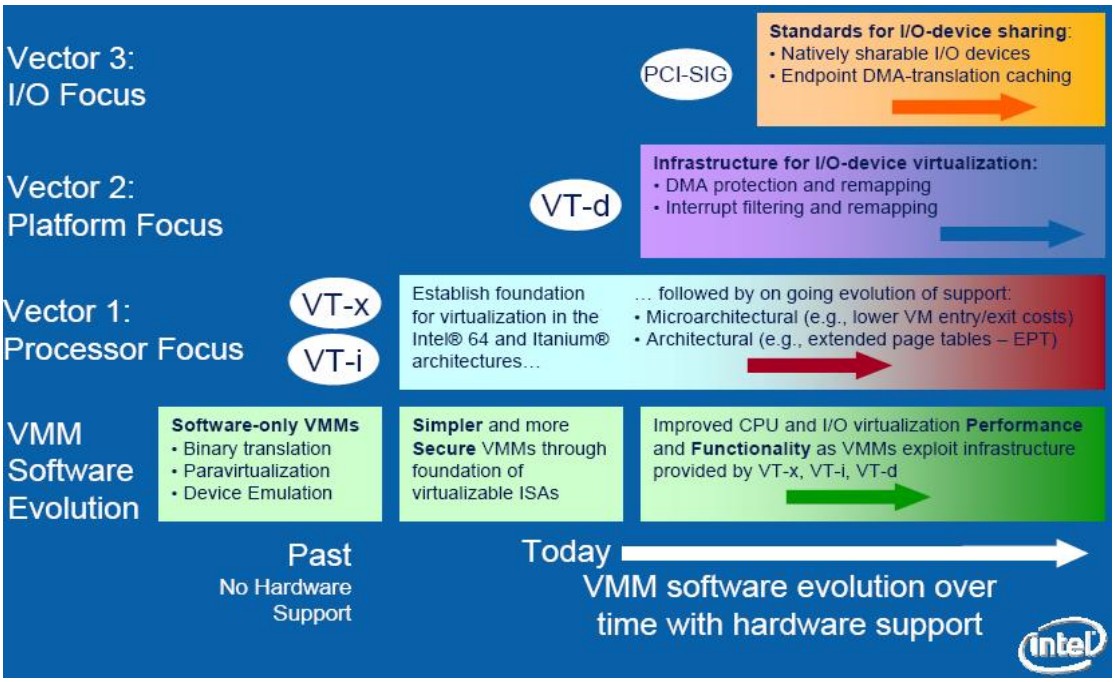
硬件虚拟化技术小结

前面章节从服务器硬件的角度对服务器一虚多技术进行了一些分析，其中两个关键方向就是 CPU 和 I/O 的虚拟化。CPU 虚拟化经历了早期的 CPU FV/PV 软件虚拟化技术发展，目前已经进入了硬件辅助技术 HVM 为主的阶段。而 I/O 虚拟化目前仍然由 I/O FV/PV 的软件虚拟化技术主导，SR IOV 等硬件辅助技术刚刚起步，存在很多使用上的局限，但预计在今后几年内随着技术的发展完善，也会逐步替代软件方式成为技术主流。

这种技术发展过程其实在网络技术方面更加常见，一个新的数据传输特性做出来后往往是先由软件实现，然后用各种方式优化改进效率，但最终总是由于性能需求推动，通过 ASIC、Fabric 或 NP 等硬件手段来实现大规模商用。

X86 系统 HVM 的最大推手和赢家就是 Intel，当然对 DELL、HP 等服务器集成商，Microsoft、Redhat 等系统厂商，VMware、Citrix 等 VMM 厂商以及 Google、Amazon 等云计算服务提供商来说，做大虚拟化这块蛋糕符合大家的利益，属于共赢的局面，因此可以看到所

有人都对 HVM 技术拍手称好。下面以一张来自 Intel 的胶片截图将其几个主要技术汇总作为本段的结尾。



Hypervisor 虚拟化

前面从硬件的角度来分析了虚拟化技术的构成，下面来看看 Hypervisor，主要针对 Type1 Hypervisor，介绍几款主流产品的构成与区别。

开源与社区

首先来说几句题外话。查看国外软件的时候，经常会看到 XX 软件开源、XX 开源软件由 XX 社区维护，XX 项目隶属于 XX 基金会等内容。感觉整个软件业的生态环境和国内有巨大差别，有必要先介绍一些背景。

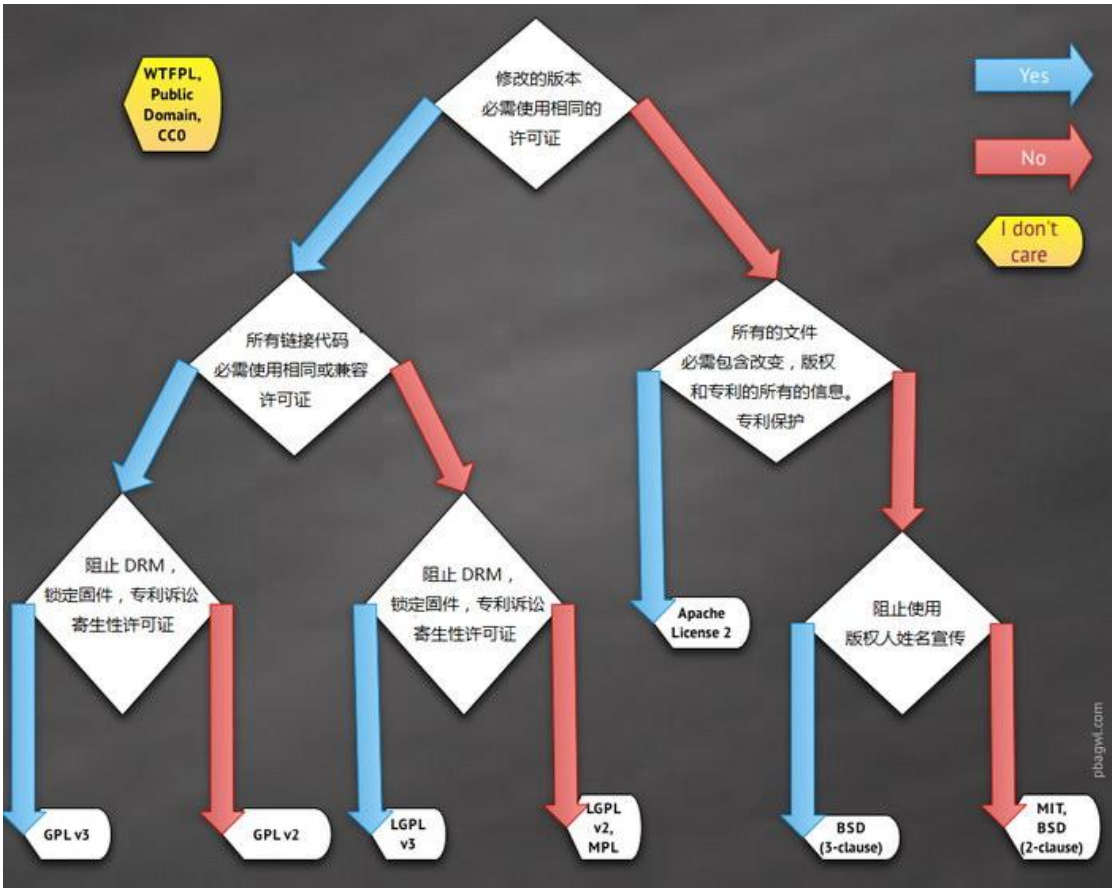
开源指开放源码 Open Source，最早是在 1998 年 2 月份美国加州 Palo Alto 一个会议上由 Christine Peterson 提出的。早先的自由软件中 Free 一词同时拥有自由和免费的含混意思，容易使人产生误解，而 Open 一词可以更好的对自由软件进行阐释。为了改善自由软件的生态环境，Open Source 通过制定完善的开发使用规范使其更容易被个人与企业所接受，更具有生命力。 同年 2 月底开源计划组织 OSI(Open Source Initiative)成立，对开源理念推广提供支持，至今此组织仍然是开源软件的中坚力量。

开源对作者来说主要是为了使软件能够得到更广阔的发展前景，同时通过更多的人参与进来对源码进行修改补充，也可以使作品更加完善。开源作为一种软件开发方式已经成为大势所趋，可以说当前的软件行业没有哪家公司会站出来说和开源一点儿关系都没有。从编程语言 JAVA、C、PHP 到操作系统 Linux、Android；从 HTTP 服务器 Apache 到数据库 MySQL；从浏览器 Firefox 到 Blog 系统 Wordpress；从多虚一分布式技术 Hadoop 到后面要介绍的一虚多 Hypervisor 系统 Xen 和 KVM，这些都是开源软件中大家耳熟能详的重量级产品。

开源软件最大的特点就是制定了很多规范用来确定使用者的权利和义务。这些规范一般都会被放置在代码的最前面或单独成章使人一目了然，我们通常称其为软件使用许可证。在 GNU 网站上记录了几乎所有的开源软件许可证，达到上百个之多，但我们平时主要会用到的就是 BSD（Berkeley Software Distribution）、MIT（Massachusetts Institute of Technology）、Apache、GPL（GNU General Public License）和 LGPL（GNU Lesser General Public License）几个。其中 BSD/MIT 和 Apache 相对宽松一些，重点要求保证作者对原始程序的版权，要求每个修改版本中都必须保留原始程序的作者信息。GPL 是当前应用最广的许可证，由 FSF（Free Software Foundation）发布，其赋予接受者对软件程序自由使用、修改、复制和发布改进的权利，并通过 Copyleft（针对 Copyright）规则，要求修改继承遵循 GPL 的软件得到的所有演绎版本都仍然必须遵循 GPL。鼎鼎大名的 Linux 就是使用 GPL，帮助我们可以免费使用所有版本的 Linux OS。当然企业和个人在遵循 GPL 开放源码和免费使用的基础上，仍可以对服务进行适当收费，这也是当前很多 Linux 厂商的存活之道。LGPL 对 GPL 进行了一定放宽，当发布程序对遵循 LGPL 的软件进行调用、

连接而非包含时，允许封闭源码。LGPL 最早是针对类库进行的设计，因此早期也曾被称为 Library General Public License。

对于主要开源许可证的使用选择可以通过下面这张网上截图来查看。



其中 WTFPL（What The Fuck Public License）具体内容如下，从名字就能看出其 NB 之处。

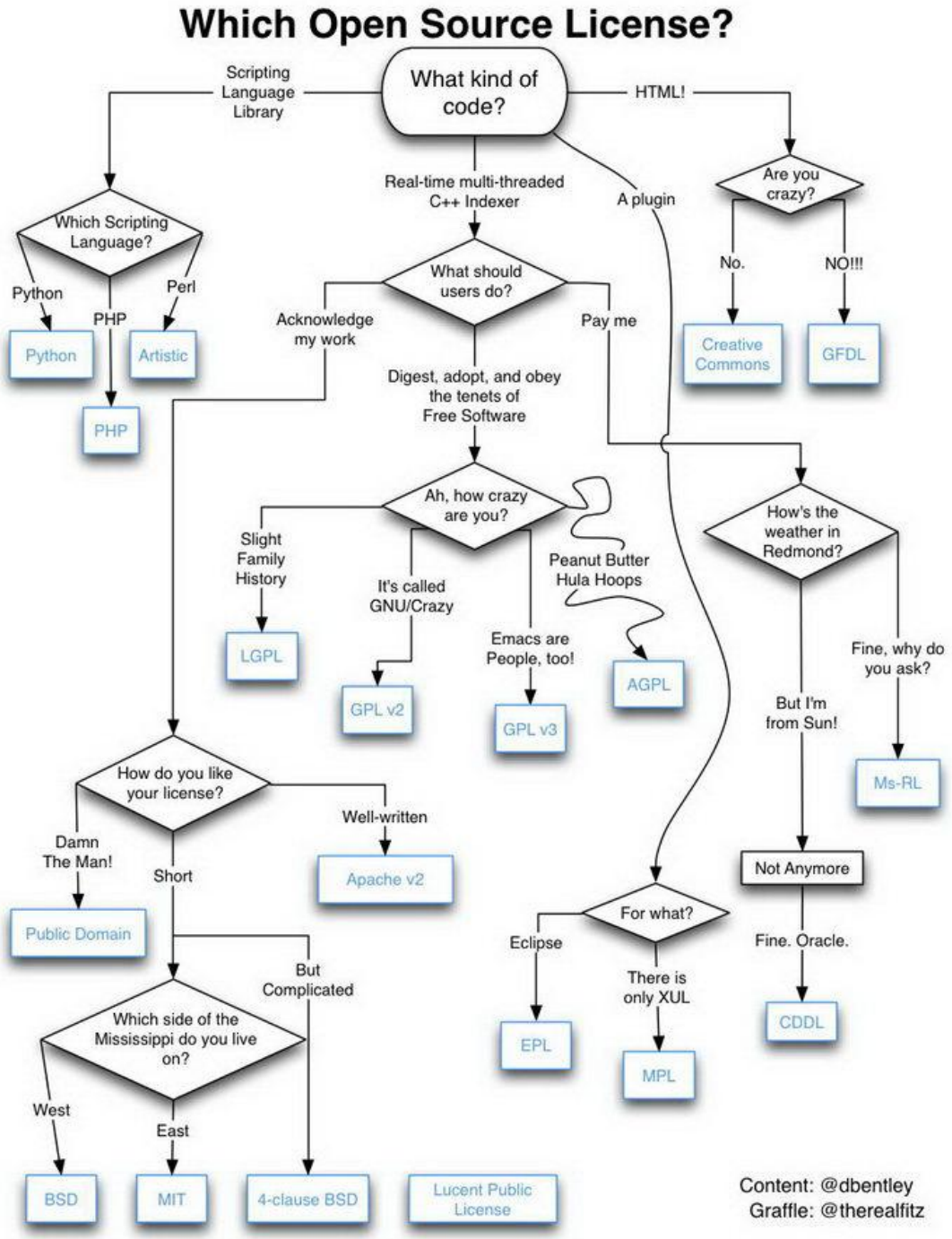
```
DO WHAT THE FUCK YOU WANT TO PUBLIC LICENSE

                    Version 2, December 2004


 Copyright (C) 2004 Sam Hocevar <sam@hocevar.net>


 Everyone is permitted to copy and distribute verbatim or modified copies
of this license document, and changing it is allowed as long as the name
is changed.
```

DO WHAT THE FUCK YOU WANT TO PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. You just DO WHAT THE FUCK YOU WANT TO.

另外还有张截图更详细也更调侃一些。此处关于开源软件的理解描述大部分源自于开源中国社区（www.oschina.net），有兴趣的同学可以自行登陆查阅详细内容。



关于开源软件再补充下其与免费软件的关系，开源软件不一定就免费，而免费软件也不一定会开源，一切均取决于开发者的意愿并通过许可证 License 进行控制。

那么开源软件如何发布和管理呢，这就要说到社区了。社区有多种多样的组织形式，其主要作用有两个，一是作为代码仓库为开发者们提供修改程序管理版本的平台，二是作为发布窗口为使用者们提供获取软件程序及源码的平台。

从服务的开源软件对象类型可以将社区简单分为三大类。首先是服务无数小型软件的社区，此类社区为开源软件提供最基本的版本管理与发布平台，典型代表就是SourceForge.net，也被称为SF.net。其作为全球最大的开源软件开发平台和代码仓库，为近30万软件项目和几百万注册用户提供服务。SourceForge既包含如Wiki百科使用的Media Wiki这种知名开源程序，也包含了大量个人开发及目前已经停止开发的软件项目。类似的社区还有采用分布式版本控制系统Git的GitHub和采用Mercurial的BitBucket；由Microsoft维护专注于Windows周边程序的CodePlex；以及由Google提供服务资源的Google code（最初专注于Google产品及其周边的开源软件，目前已经成为更加开放的软件托管平台，代表软件为Android）。

第二类社区专门服务于某个大型开源软件，典型代表如Linux、Mozilla、OpenBSD、OpenOffice、Eclipse、Xen和OpenStack等社区。此类社区还要对其服务的开源软件进行推广、维护和提供法律保护等工作，比第一类社区做得更多，也需要更大的投入，往往会以基金会的形式运作以确保持续运营的能力。

第三类社区为多个大型开源软件项目提供管理运维服务，对项目运作及版本发布使用进行更严格的监管和控制。如果将第一类社区比作大型集市，将第二类社区比作专卖店，那么第三类社区就是精品百货商场了。典型代表目前看来好像也就ASF(Apache Software Foundation)一家。ASF是由第二类社区发展来的，最初只是针对开源软件Apache HTTP Server的维护，后来在启动并行项目研究时，子项目由于发展得很好就被提升为新的顶级项目，同时很多优秀的项目也加入到ASF，由ASF统一管理，将其项目家族打造成今天的庞然大物。现今ASF拥有93个直接管理的顶级项目，同时还托管着如OpenOffice等其他开源社区。其知名顶级项目有Apache HTTP Server、Tomcat、Ant、Struts、Lucene、Nutch、Logging、Geronimo、XML、Perl、Hadoop、Hbase等。在自建维护社区有顾虑的情况下，将项目托管给ASF管理和运作已经成为大型开源软件的主要选择之一。ASF管理的开源项目全都遵循许可证Apache License2.0。

最后再简单介绍下这些社区的运作资金来源。支出方面虽然部分工作人员可以是自愿的投入，但是必要的损耗仍然是避免不了的，如服务器、网络等物理资源及法律支持和日常维护的人员开销。从收入方面来看主要是三个方面，首先是广告收入，通过网站上的广告位出售获取资金，这块是小头。然后是项目收入，通过特定License授权开源软件给第三方企业进行商业化修改包装销售来获取部分收入，ASF就允许其管理项目以此方式受益，但明显这种行为只能适合于部分易于包装为产品的项目，而且个人觉得与开源软件的初衷有悖。最后一项实际上开源软件的最大头收入是来自于募捐，也就是我们看到社区经常被命名为XX基金会，其背后经常会看到一个或多个主要的金主，当然这些财神们也会对开源软件的发展产生一些影响和利益纠葛，置于背后的博弈就不是我这篇文章里面要分析的了。

关于基金会募捐再多提两个有趣的案例。一个是 2006 年时，OpenBSD 的管理人员向部分厂商发出了对 OpenSSH 产品项目的捐款请求，用于 OpenSSH 的开发和维护工作。这些厂商在其自身产品中都集成和使用了 OpenSSH，并因此产生了部分获益，对象包括 HP、IBM、Redhat 和 Cisco 等大公司。最终的募捐结果没有找到，但据说收到的最快回应是来自于 Mozilla 基金会捐出的 1 万美元，因为 FireFox 浏览器中也使用 OpenSSH 的部分功能。另一个案例也与 Mozilla 有关，由于开源基金会大多定位于公益性的非盈利组织，在 M 国是免税的。而 2007 财年发布的公告中，Mozilla 从 Google 收入了一笔 6600 万美元，来源于其在 FireFox 浏览器中默认集成的 Google 搜索引擎。此收入已经占到了 Mozilla 当年收入总和的 88%，因此被 M 国政府税务局进行审查，考虑要免除其非盈利组织的的免税待遇，并要求为 2007 年补税 10 万美元。几经波折之后，Mozilla 还是最终逃过了一劫。

上述两个案例均来自于网上资料，作者没有任何的主观倾向性与褒贬意图，只是摘录来和读者分享，如引起任何人的不适，请望谅解，如有不正确之处请知会作者，会立即进行修改删除。

Hypervisor 与操作系统内核

回归正题，继续讲技术，当前有四种主流的 Hypervisor 产品，VMware ESXi、Xen、KVM 和 Hyper-V。其中 Hyper-V 未开源， VMware 部分开源（从 VMware 网站上可以自由下载到包括 ESX/ESXi/vCenter 等所有发售产品的部分源码），Xen 和 KVM 则完全开源。下面将从操作系统内核的角度分析下这几款产品的差异。

操作系统与内核

先介绍一些背景技术。从批处理到分时处理，到实时处理再到分布式处理，从单进程到多进程处理，推动操作系统出现与发展的根本需求都来源于对多工作任务的排序处理。其核心作用就是在不同应用程序运行操作硬件资源时，进行统一安排调度。

在最早的计算机时代，根本没有操作系统的事，应用程序直接去调用操作硬件进行任务处理，一个任务结束，后面人再上去安装运行新的任务。随着时间的发展和硬件性能的提升，大家觉的可以安排些自动化的工作，就不用操作人员天天站在设备前面排队了。于是最早的操作系统诞生，其只需要进行任务分时调度即可，反正所有的工作都是排队顺序执行。此时的应用和硬件也就那么几种，仍然由应用程序编写时通过代码直接操作硬件就够了。但是随着硬件和应用的发展，到了今天估计没有人能数清计算机到底包含多少种应用程序和多少种硬件组成单元（包括不同品牌），在应用程序中直接集成硬件操作指令也就成为不可能完成的任务。因此现代的操作系统就必须提供任务调度外的另一项必要功能，对应用程序代码和硬件操作命令之间进行转换，简单来说就是在应用程序和硬件之间做一个翻译器，帮助两边进行交流。

由于服务对象众多，这种翻译往往不是一步到位的，首先需要由各个硬件设备厂商提供驱动 Driver，将最基本的硬件操作指令翻译给操作系统，然后操作系统需要将这些操作指令抽象为应用程序可以理解的高级对象（如文件系统等），最后由编译接口 API（如 C 库等）将这些抽象的对象元素提供给应用程序调用。我们通常将执行硬件抽象动作的部分称为操作系统的内核 Kernel，同时内核还要负责任务调度排序处理的执行，因而其成为了不同操作系统之间的根本区别之处。

当然完整的操作系统还必须提供人机交互通道（命令行或者图形方式）以及一些其他的系统服务。虚拟化技术应用场景中，监控管理 VM 运行的 VMM 可以理解为一个应用程序，也必须通过内核才能去操作硬件，因此前面也说过 Type1 的 Hypervisor 可以理解为 Mini OS+VMM。程序设计的原则永远是简单为美，这个 Mini OS 更需要小而稳定，必要的最简人机交互通道和系统服务大家能做的都差不多，而 VMM 需要提供的基本功能也同样类似，那么 Hypervisor 真正的差异化设计思路就剩下内核了。

内核分类与 Hypervisor

内核最基本的任务有以下几项：

1、 进程管理：对应用程序运行时在内存中存放的代码执行进行管理。比如执行的先后顺序，硬件操作冲突了怎么办（多核或多进程情况下），应用程序间互相通信调用怎么办（IPC）等等。说白了都是由最开始的多任务批处理扩充演变而来的。

2、 内存管理：由于进程（应用程序）的代码都被放到内存里面执行，那么对内存的地址分配就需要统一管理了，需要将物理内存的空间抽象为虚拟内存元素（如 Page 或 Segment）方便应用程序使用。另外既然虚拟化了就可以将硬盘上的空间以链接形式扩充为虚拟内存，提供更高的内存的可用性（如休眠时将内存都拷贝到硬盘上用于快速恢复）。需要注意，由于内核也是一段代码要在内存中运行，因此通常将所有的虚拟内存地址划分为专门跑内核代码的内核空间（Kernel Space）和应用程序代码的用户空间（User Space），应用程序不能直接访问和操作内核空间干扰内核代码的运行已经成为当前内核设计的基本准则。下文的内核分类会涉及到这两个代码空间的划分。

3、 I/O 设备管理：就是前面讲的硬件抽象等设备管理工作。是内核最重要的工作之一。

4、 系统调用：操作系统通过 C 库等 API 提供给应用程序的一组专用代码，用于改变 CPU 模式，执行访问设备和内存的一系列基础动作。包含 close、open、read、write 和 wait。

操作系统内核目前有很多分类，最根本的是单内核与微内核类别，其他的都可以看作是由这两种内核设计思路衍生而来。单内核 Monolithic kernel，内核的所有功能代码都运行在同一个内核空间内，包括模块化设计，所有的模块也是运行于同一空间。此种方式设计起来会比较复杂，任何一个功能模块的问题都会导致整个系统的崩溃，当然优势是性能较高。典型代表是 Linux 内核。

四大 Hypervisor 中的 KVM 就是直接使用 Linux 内核，可以作为程序安装在任何一个开放的 Linux OS 上，更像传统中 Type2 类型的 Hypervisor，底层的开放 Linux OS 上可以安装任何驱动和应用程序，灵活但导致系统整体稳定性稍差。下文还会对 KVM 进行进一步分析。

另一类被称为微内核 Microkernel，其设计思路是类似 Client-Server 的分层模式，只有上述四项内核任务中最基本的部分代码会运行于内核空间作为 Server（注意四项任务一个都不能少），非关键部分和其他的操作系统相关内容（如 Network，File System 等）都运行于用户空间中作为 Client。这样的好处是任何一个 Client 有问题，都能够避免整个 OS 的崩溃。从理论上来说，虽然内核空间内的代码尽量简化，能够远小于单内核设计，但由于处于不同空间，服务调用时就需要更多的指针从而导致更大的内核代码总量，性能效率也会变低（更多次上下文切换）。Mach 和 QNU 是微内核的两个典型代表之作，目前更多的被应用于对稳定性有更高要求的一些特定领域中，如航天和医疗等。单内核与微内核孰优孰劣的争执由来已久，本文不做深入讨论。

微内核作为一种通过层次化将基本内核功能尽量简化的思路，实际上没有明确界定哪些功能代码就一定应该是最基本的，需要放在内核空间中，而哪些代码就要作为 Client 放在用户空间中。因此有了第三类内核，处于单内核与微内核之间，对设计难易程度、工作效率与稳定性的妥协产物，混合内核 Hybrid kernel。此类内核相比较微内核将更多的东西放入了内核空间中，但仍选择了一些非关键代码放置于用户空间中。由于归类划分完全由设计者自行定义，因此混合内核在封闭操作系统中很受欢迎，如鼎鼎大名的 Windows 和 Mac OS X。

VMware 的官方公开资料较少的提及其 Hypervisor ESX/ESXi 的具体内核技术结构，从种种蛛丝马迹中，作者主观的推断其应属于微内核或混合内核结构（这两者并没有明确区分界定），其微内核 VMkernel 有很大可能是从斯坦福大学的 SimOS 变化而来。下文也会对其进行更详细的分析。

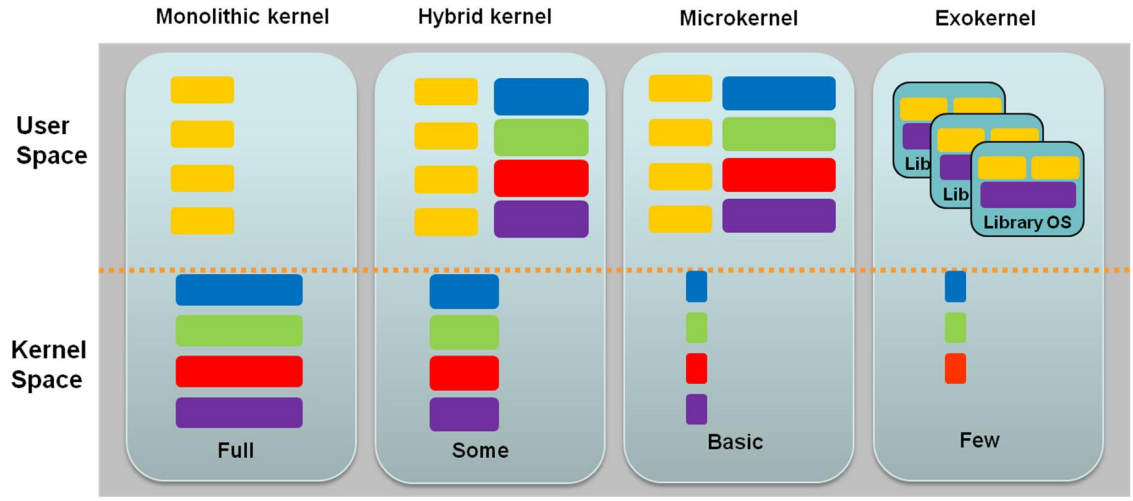
除了上述主流内核外，还有其他的很多类型，如微微内核 Nano kernel：将设备管理的中断和计时器等基本的控制工作都扔给设备驱动程序去做，以达到内核空间代码比微内核更简的目的；巨内核 Megalithic Kernel：将所有的操作系统相关内容甚至包括应用软件程序都写到内核空间中，以达到最快的运行效率目的，当然这也意味着安装修改软件时就只能重编整个内核了。

这里还有一款需要重点介绍的内核被称为外内核 Exokernel（也叫垂直结构操作系统），比微内核更加极端一些，外内核将硬件抽象工作也从内核空间中移除，丢回给用户空间的应用程序去完成，只确保应用程序访问资源时，资源是空闲可用的。当然为了处理硬件抽象工作以及维护设备驱动和 API，外内核往往需要另外一个 Library 操作系统进行硬件设备管理，而且这个 Library OS 的内核代码需要有所修改，以便与 Exokernel 协作处理任务。

感觉 Exokernel 的思路有些回归到计算机操作系统原始状态的意思，以处理任务为主，不负责维护硬件操作。这样做有三个明显的好处：一是内核空间代码量更小内核更稳定；二是应用程序可以直接做硬件抽象工作，软件开发人员对硬件使用拥有更高的自由度和处理效率；三是外内核之上理论上讲可以同时运行多个不同类型的 Library 操作系统如 Linux 和 Windows 进行基本的硬件抽象和管理工作，同时通过不同的 API 供不同的类型的应用程序使用。换而言之，我们可以做出个巨大的操作系统，这个操作系统以很小的 Exokernel 在内核空间内负责协调工作，然后在用户空间内运行一大堆各种操作系统作为其 Library OS，并跑上基于各种操作系统 API 的应用程序，如 Apache 的 WEB Server 和 Windows 的 SQL Server 等等。有人该说了，这不就是理想的服务器虚拟化么。Bingo，准确的说这个也许是将来的虚拟化结构，但是 Exokernel 现在还是处于试验阶段的技术，没有办法十全十美的实现上述模型。另外真这么搞的话需要 Library OS 都得改内核以适应 Exokernel，也是个不大不小的麻烦。典型的 Exokernel 如 University of Cambridge 的 Nemesis 和 MIT 的 ExOS。

OK，理想的虚拟化现在还实现不了，那么我们先搞搞折中的办法，Library OS 不好直接用来跑应用，可以先弄一个出来专门做做硬件设备管理，再通过类似 QEMU 等模拟器走主流路线去虚拟其他的 Guest OS 来运行应用程序。这就是 Xen 和 Hyper-V 的思路，由此我们看到其 Hypervisor 必须和一个叫做 Domain 0 或者 Parent Partition 的 Library OS 一起运行为 Guest OS 提供虚拟化服务。又因为这个 Library OS 必须是内核增加了相关代码的 Modified OS，导致一直迟迟未能进入 Linux 内核的 Xen 与 Redhat 等 Linux 集成商们之间不断演绎着悲欢离合，而 Microsoft 干脆将 Hyper-V 与改了内核的 Server 2008 捆在一起安装发售。下文还将详细介绍这两款主要的 Hypervisor 产品。

通过下面这张图再将前面四种主要内核的关系汇总一下。



OS	Linux	Windows	Mach	Nemesis
Hypervisor	KVM	ESX/ESXi		XEN Hyper-V

从代码量来说，这四种内核的内核空间代码是逐渐减少的，但就整个内核的代码总量而言，通常由大到小的顺序是 Hybrid kernel、Microkernel 和 Monolithic kernel，而 Exokernel 由于必须与 Library OS 共生，因此不好进行排位。

主流 Hypervisor 产品介绍

按照发布时间的早晚，下文依次详细介绍 VMware ESX/ESXi、Xen、KVM 和 Hyper-V 技术。

VMware ESX/ESXi

首先分析 VMware 的 ESX/ESXi，无论公司还是产品都是当前 Hypervisor 界当之无愧的重量级元老。先八卦一下 VMware 的精彩历史。

1998 年 VMware 在 M 国的 Palo Alto, California 由五位高人创建，其中的关键人物 Mendel Rosenblum 是 Stanford 的副教授，当时 SimOS 项目的主负责人；Edouard Bugnion 是他的在读博士学生，当时 SimOS 项目的主要研究员；Diane Greene 是 Rosenblum 的妻子（两人相识于 University of California, Berkeley）。

1999 年 5 月，VMware 发布其第一款产品 VMware Workstation。2001 年发布 GSX Server（Type2）和 ESX Server（Type1）两款 Hypervisor 产品正式进入服务器虚拟化市场。2003 年 VMware 发布 Virtual Center（后来的 vCenter）产品，集成 vMotion 和 VSMP 等技术。这些产品一直发展到现在已经成为虚拟化技术的典型代表。

2004 年，VMware 以$625 million 被 EMC 收购，截止目前仍以其全资子公司的形式独立运作。

2005 年 Bugnion 从 VMware 的 CTO 职位上辞职，参与创建了 Nuova System，此公司 2008 年被 Cisco 收购，而后这个哥们在 Cisco 一直干到 VP 的职位，于 2011 年辞职回 Stanford 继续完成其博士学业去了。

2006 年 6 月，VMware 收购了私有公司 Akimbi Systems，开始其成长为 IT 大鳄的吞噬之旅。

2007 年 8 月，EMC 将 VMware 的 10%股份拿出来在纽交所上市，发行当天从 29$上涨到收市时的 51$。

2008 年 5 月，VMware 收购了以色列的 Start-up 公司 B-Hive Network，并基于其场地设备和人力等资源在以色列建立了第一个海外研发中心。

2008 年 7 月，VMware 创始人之一 Greene 在 CEO 的位子上被董事会突然辞退， EMC 云计算部门主管 Paul Maritz，一位曾在微软工作 14 年的老兵取而代之。

2008 年 9 月 10 日，另一位重要创始人 Rosenblum，Greene 的丈夫，也从 VMware 首席科学家的职位上辞职，与妻子一同回归校园，专心于研究事业。上述人员变动导致 VMware 当时股价剧烈波动，市值骤降接近 25%。

紧跟着 2008 年 9 月 16 号，VMware 宣布与 Cisco 在数据中心市场进行深度合作，将其 N1000v 虚拟软件交换机集成到 VMware 架构中作为可选产品一同销售。

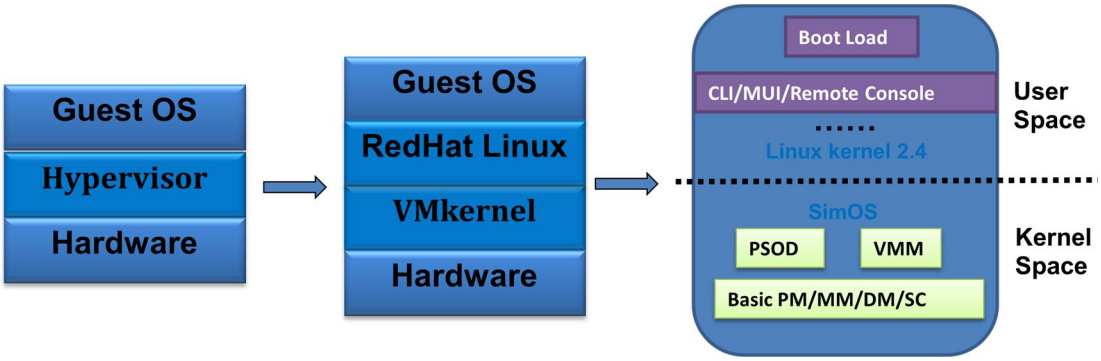
随后几年内，VMware 陆续收购了多家 PaaS 相关软件公司，并与 2011 年 4 月推出自己的 PaaS 开源服务平台系统 Cloud Foundry。

紧跟着在 2011 年 4、5、6 三个月，VMware 又连续出手收购了 3 家 SaaS 相关软件公司，其下一步动作昭然若揭。。。

从 VMware 的发展史中，看到了又一个典型的硅谷公司成长案例，和早年的 Cisco 起家颇有些相似之处。技术与资本之间错综复杂的关系总是使人滋味无穷，估计再过个几年高科技企业创业的题材应该会掀起一阵影视热潮，诸位有志于向编剧导演发面发展的同志们现在可以适当关注并抓紧搜集素材了。

好了，小憩之后让我们转回到技术，讲下 ESX/ESXi 是怎么一回事。ESX 名称来自于“ Elastic Sky X”，而 ESXi 则是从 ESX3.5 版本开始改的名，最早是 ESX 3i，后来就直接叫做 ESXi *.* 了。ESXi 比较 ESX 最大的变化是对 Hypervisor 进行了简化，去掉了服务控制台等功能，导致产品在体积缩小的同时从操作角度来看更加封闭，只能使用其 vCenter 或 Client 软件进行管理操作。最新的 ESXi5.0 安装包的大小为 290MB，远远小于早期的 ESX 版本。

本文开篇的 Hypervisor 典型分类其实就是从 VMware 的 ESX 和 GSX 产品分类而来的。GSX 为代表的 Type2 Hypervisor 由于其性能局限，目前的应用已经很少，不做细说。ESX/ESXi 一路发展至今，在服务器虚拟化市场已经独占鳌头。从结构上讲，VMware ESX 一直给人以紧密结合的一个整体的印象，由于其各种系统服务（包括系统启动和管理操作等）都是采用源于 Linux 2.4 内核的 Redhat 发布版进行修改后提供的，导致大家很容易误解其采用的也是 Linux 内核进行整个系统硬件的管理。但是其实不然，它还另有一颗不同的心 VMkernel。VMware 的官方发布资料一直有意无意的对其内核结构进行模糊宣传，但是 VMkernel 作为其不开源内核肯定和 Linux 无关是毋容置疑的，否则其必须得遵循 GPL 也完全开源。而事实上 VMware 只是对其产品中修改使用的各种 Linux 服务组件进行了开源，并没有将整个 ESX 完全开源。还有从 ESX 支持的所有硬件驱动必须集成在产品版本中发布，而不能进行自动安装修改这一表现推断，硬件抽象这一内核基本工作应该是由 VMkernel 完成的，而非其中的 Linux 内核。因此推断其结构应该是由 VMkernel 作为微内核，而 Linux 内核运行在用户空间，作为一个 Client 进程，管理整个系统其他服务特性（如 CLI、Boot Load 等）。这是典型的微内核或混合内核结构，区别就在于 VMkernel 在内核空间中集合的代码多少而定。再参考前面 VMware 创始人的经历，SimOS 的修改版本作为 VMkernel 微内核的推断基本上就十之八九成立了。注意这里说的是推断，只有等 VMware 的官方资料公布后才能确认对错。



多说一句，SimOS 项目最早研究的是在 MIPS 上运行 IRIX 系统，目前在 Stanford 网站上已经搜索不到，其衍生出的 SimOS-PPC 成为 IBM AIX 系统模拟的一个内部项目，在 AIX 4.3 License 下对外发布。还有一个衍生产品 SimBCM 用于模拟 Broadcom BCM1250 作为完全开源软件在 GPL 下发布。国内有些研究龙芯的兄弟在 2002 年左右就开始接触 SimOS，并研究通过其对 Linux/MIPS 进行模拟，有兴趣的可以去网上搜搜当时的一些文章，不过这两年也都销声匿迹了。

最后再介绍下 VMware 的最新服务器虚拟化产品家族，让大家对其盈利模式有所了解。首先是 ESXi，去年 8 月底发布的版本已经到了 5.0，免费使用。（早先的 ESX 版本都是收费的，但从 ESXi 开始全部免费使用）。单独的 ESXi 可以通过 vSphere Client 软件进行管理，也是免费的，但同时只能一次性管理一个物理 ESXi Server，不嫌累的话可以在管理 PC 上开多个进程同时管理多台物理 ESXi Server。当然不管谁真想用虚拟化，肯定不

会就只有一两台物理 Server，那么就需要 vCenter 软件对多台服务器进行集中管理了，vCenter 可以提供 convert、vMotion、HA 和 DRS 等 VM 扩展管理功能，而这些扩展功能是大型数据中心 VM 管理所必须的。vCenter 有 60 天的免费试用期，依靠 License 进行控制。这样做的好处是可以避免各种试用和测试等场景的 License 控制管理麻烦。同时只要是个正规点儿的企业使用 VM，谁也忍受不了 60 天就将整个系统重新折腾一遍，肯定要乖乖掏银子。

Xen

第二个 Hypervisor 界老牌重量级选手就是 Xen 了。Xen 最早诞生于剑桥大学电脑实验室的研究项目，2003 年 10 月 2 号由研究项目领导人 Ian Pratt 创建的 XenSource Inc.发布了第一个 Xen 的发行版本。2007 年 Citrix 收购了 XenSource，将 Xen 产品全部置于旗下，并将 Xen 的开源项目转移到 Xen.org 进行维护。同一时期 Citrix 推动成立了 Xen Project Advisory Board（简称 Xen AB，早期成员有 Citrix, IBM, Intel, Hewlett-Packard, Novell, Red Hat, Sun Microsystems 和 Oracle）对 Xen 进行代码管理和市场推广。2009 年，Citrix 宣布将其虚拟化管理平台 XenServer 也在 Xen.org 上全部开源，命名为 XCP（Xen Cloud Platform）。

目前，Xen.org 上面共包含 6 个相关项目，主要项目 Xen Hypervisor 和 XCP；孵化项目 Xen ARM Project（与三星合作在 ARM 上利用 Xen 达到虚拟化目标）和 XCI - Xen Hypervisor for Client Devices（目标减小 Xen Hypervisor 的代码规模，类似于 ESXi 之于 ESX）；归档项目 HXen - Hosted Xen（在 Windows 上和 Mac 上运行 Xen 的 Type2 类主机虚拟化）和 Project Satori（在 Hyper-V 上运行带有 Xen 代码 Para Virtualization Linux 作为 Guest OS）。

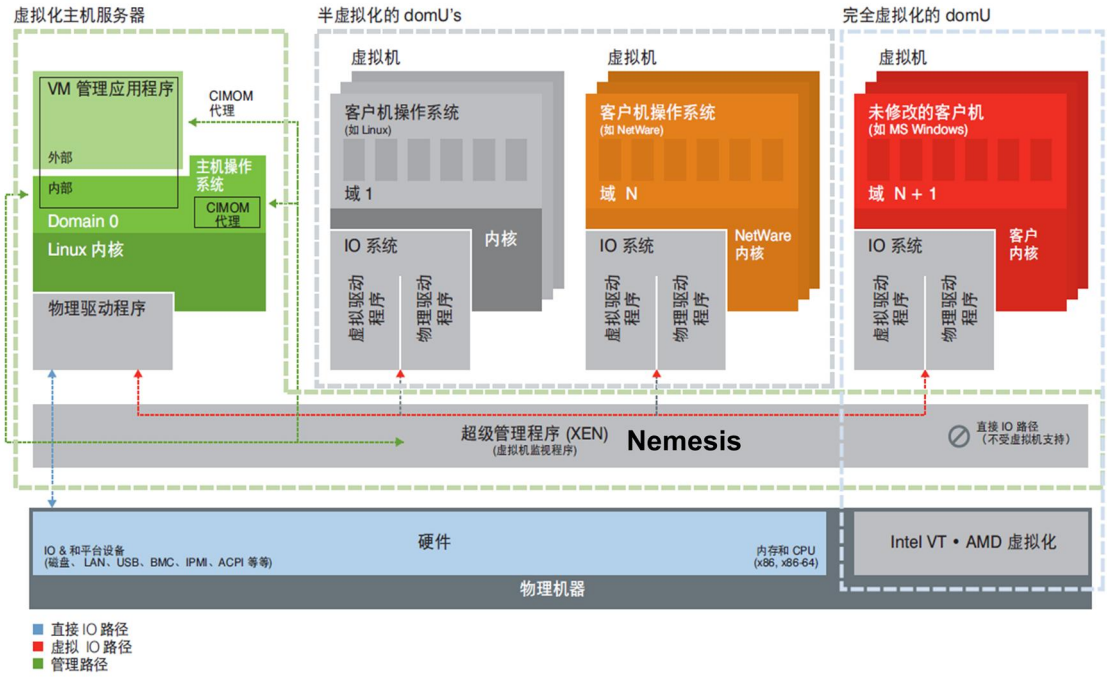
再八一八 Xen 现在东家 Citrix 的背景，这家由 IBM 工程师在 1989 年成立的公司，最初定位于操作系统（当时的 OS/2，Win32 等）多用户使用的软件市场，然而命运坎坷，从 1989 年到 1995 年连年亏损，甚至在 1991 年之前几乎没有任何收入。但从 1991 年开始至 1993 年 Citrix 融到了多笔风投，并收购了北电的一款软件产品 Netware Access Server，随后开始走上了上坡路，并于 1995 年上市。从 1997 年至今 Citrix 共收购了 27 家公司并将其产品融入到自己的产品体系中，如当前其主要产品 GotoMeeting、NetScaler、EdgeSight 和 Xen 等都是靠收购整合而来的。现如今 Citrix 已经发展成为超过 6000 人的全球性企业，2011 年总资产达到$3.75B，其中 2010 财年收入$327M。个人看完后感觉很励志，创业在于坚持，不要怕没收入，不要怕技术方向走错，山穷水尽疑无路，柳暗花明又一村。当然对 Citrix 来说得到 Microsoft 的贵人相助也是很重要的，更细节的内容有兴趣的同学自己去网上查看吧。

回来说 Xen Hypervisor 的实现，在前面已经介绍了部分内容，这里再对其进行一下汇总。首先从内核结构来说，Xen Hypervisor 采用的是 Exokernel 结构，其 kernel space 运行的

内核代码推断为剑桥大学电脑实验室 90 年代研究的 Nemesis 内核演变而来。Nemesis 在 99 年 4 月发布第二个版本之后就渺无声息了，时间上也与 Xen 的研究发布时期比较匹配。

然后是管理驱动的 Library OS，Xen 称其为 Domain0，可选的操作系统很显然是开源的 Linux 最为合适。为了和 Exokernel 配合进行系统调用等动作，必须对 Linux 内核代码进行适当修改才能作为 Domain0 系统。和 VMware 的 Linux 系统类似，Domain0 同样要负责 Bootload 和 CLI 等系统管理工作。Xen 与 Linux 的关系可谓一波三折，从诞生开始，开源的 Xen 就一直在受到市场的追捧，早期的 Amazon EC2、RackSpace 等大型 IaaS 运营商都是采用 Xen Hypervisor 来搭建其虚拟化平台，这些有实力的厂商自行修改和维护 Linux 内核代码，使 Xen 可以顺畅运行，同时推动了 Linux 集成厂商对 Xen 的支持，到 2009 年时，Suse、Redhat、Ubuntu、Debian、Gentoo 和 Arch 等发行版都集成 Xen domain0 的代码。但是随着 KVM 的雄起，以及 Xen Domain0 代码始终未能进入 Linux kernel，2009 年 KVM 的东家 Redhat 在其最新的 Enterprise 6 版本中去掉了支持 Xen Domain0 的代码（其另一款完全开源的知名产品 Fedora 则始终未支持 Xen Domain0），其盟友 Ubuntu 新的 8.10 版本也不再支持作为 Xen Domain0。这些变化导致 Xen 的前景一度不被人们所看好，甚至很多人开始预言 Xen 的末日来临，其地位很快会被 KVM 所取代。然而以 Citrix 为主导的 Xen 盟军们开始发力，2010 年 7 月 Linux kernel 2.6.31 发布版修改了部分代码，可以使用 PVOps 工具很简单的使其支持 Xen Domain0。2011 年 3 月 Xen Domain0 代码终于得偿所愿，进入 Linux kernel 2.6.37 发布版本。目前如 Redhat 和 Ubuntu 这些厂商们又需要在其还未发布的新版本中重新考虑对 Xen Domain0 的支持了。

最后再说下 Guest OS，Xen 早期的当家技术就是 Para Virtualization 的 Guest OS，虽然需要修改内核代码，但是相比较使用 QEMU 之类的模拟器技术，更高效的系统资源调用表现始终令人称赞。如 Amazon EC2 等 IaaS 厂商也大量采用这种 Para Virtualization Guest OS 来搭建自己的虚拟化服务器平台。而后随着硬件辅助虚拟化技术 HVM 的盛行，Xen 也开始推广其采用 QEMU 模拟器的硬件辅助技术以同时支持 Windows 系统作为 Guest OS。由于 Xen PV 代码早就进入了 Linux kernel 2.6.23 版本，因此基本上目前所有集成商的 Linux 发行版本已经都能够作为 Guest OS 支持 Para Virtualization。下面这张源自 XEN 发布资料的图可以很清楚的看出其各个部件的基本结构关系。



KVM

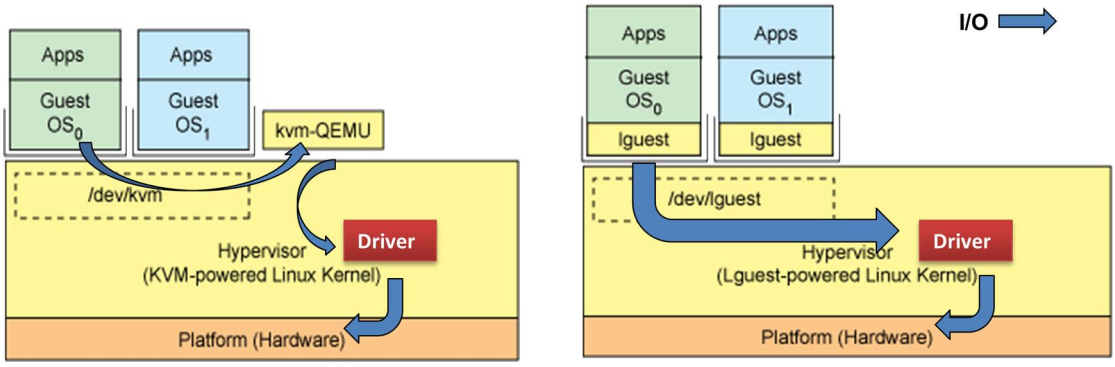
下面介绍一下使用 Linux 内核的 Hypervisor 技术 KVM（Kernel-based Virtual Machine）。这个可是最近几年比较火的明星选手，在 Redhat 等厂商的鼓吹之下，大有要在开源 Hypervisor 中取 Xen 而代之的意思。

首先说为什么会有 KVM 的出现。VMware 虽然部分开源，但是由于其 VMkernel 不公开，也没法做修改和二次开发，只有开源的 Xen 可以被广泛的开发者们折腾。而 Xen 虽然在 Linux（Domain0）上运行，但是采用的是独立的外内核 Nemesis，这点始终令 Linux 的狂信们不爽，而且也导致其无法进入早期的 Linux 内核，只能是在集成商修改过内核的 Linux 发行版中使用。（Linux 再开放也不会希望在其内核中再运行另一个内核，至于去年 Xen 成功进入 Linux 内核只能说是金主们博弈的结果，从技术上讲未必能有几人赞同）基于上述原因，Linux 的信徒决定要搞出一款真正基于 Linux 内核的开放性 Hypervisor 出来，于是有了 KVM。

KVM 是由一家以色列的 startup 公司 Qumranet 发布，2007 年 2 月进入 Linux kernel 2.6.20 版本后开始受到大家的关注，2008 年 9 月 Qumranet 被 Redhat 以$107M 收购。Qumranet 的创始人好像有以色列军方背景，应该也非常人。而后在 Redhat 的大力推动下，KVM 在 Hypervisor 技术领域的影响力扩张一发不可收拾。

从技术实现上看，KVM 必须安装在一个底层的 Linux 操作系统之上，通常也称之为宿主操作系统。KVM 包含了一个内核模块帮助宿主 Linux OS 成为一个 Hypervisor，同时使用

修改过的 QEMU 模拟器 KVM QEMU 来为其上的 Guest OS 提供进程空间，使每个 Guest OS 启动后，都会成为宿主 Linux OS 上的一个独立进程，并可以对其执行各种进程调度。与传统 Linux 的区别是，这些 Guest OS 运行在一个新的 Guest 状态，独立于标准的 Kernel 和 User 状态。Guest OS 的 I/O 请求都会先被映射到 KVM QEMU 独立进程中进行模拟处理，然后再下发给宿主 Linux OS 的 Driver 进行硬件操作。这个 KVM QEMU 进程有些类似 I/O PV 中的 Domain0 的作用，区别是 Driver 被安装在宿主 OS 上。当然我们可以不要这个 QEMU，直接将一些调用和模拟放到 Guest OS 上来做，这同样需要修改 Guest OS 内核，增加一个执行层次。这也是另一款基于 Linux 的开源 Hypervisor 软件 Lguest 的思路，如下图可以明显看出其实现的区别比较。



Lguest（也称 Lhype）是澳大利亚 IBM 的工程师 Rusty Russell 开发和维护的，在 2007 年就已经进入 Linux 2.6.23 版本内核，但由于其需要修改 Guest OS 的内核，同 CPU PV 一样，目前也只有 Linux 的版本支持。Lguest 最大的优势就是简单，Hypervisor 代码被简化到只有 5000 行左右，是 KVM 的 1/10，号称是最小的 Hypervisor，因而也被一些开发者所喜好。在 Guest OS 的 Lguest 代码瘦层中还做了很多其他虚拟化增强工作，如系统服务和中断处理等。

KVM 也曾有限的支持 I/O PV 技术（如 VirtIO），可以对部分修改过的 Guest OS 的 I/O 请求不经 QEMU 模拟直接下发到硬件，但最新的 KVM 社区介绍中已经摒弃掉了这部分内容，将其定位为一套运行在支持硬件虚拟化技术（Intel VT 和 AMD-V）的 X86 平台上的 Full Virtualization 解决方案。

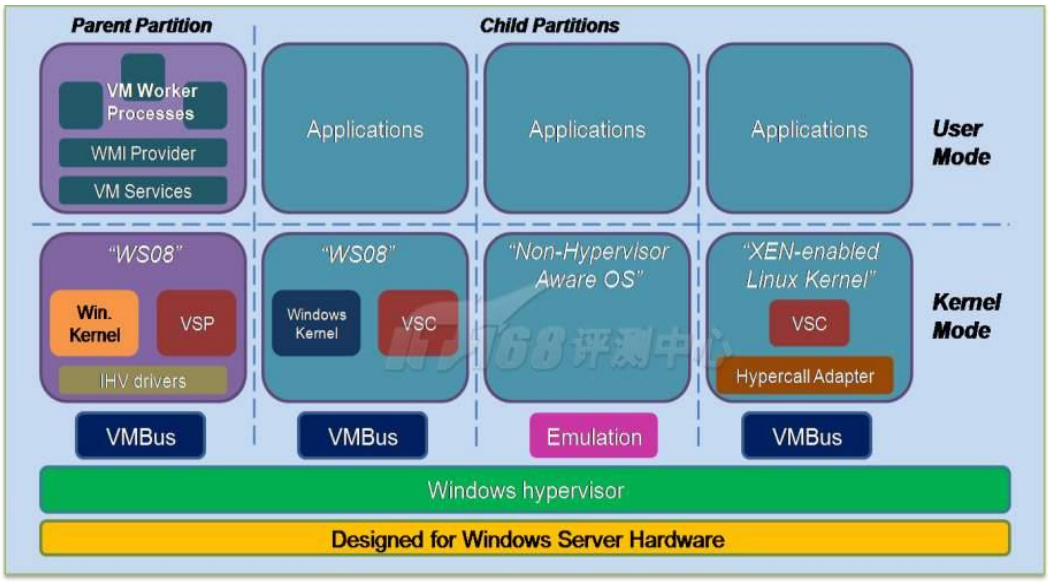
在当前的服务器市场，Linux 占据了绝对的优势，Windows 只是在中低端服务器有些份额，而像 AIX 等系统则只能在专用的高端服务器上有所斩获，开放性不够限制了其市场份额的发展。对高科技企业尤其是互联网公司来说，设备硬件的标准化（如 X86）可以降低采购成本和厂商捆绑风险，而软件开发能力又是其起家的根本，因此越大的公司越喜欢由自己的团队专门为业务应用系统定制开发和维护自己的底层平台，像 Linux 这种开源又免费的系统成为了最佳选择。得益于 Linux 的不断前进，KVM 同样在虚拟化方面拥有着光明的前景。如前所说，最大的 Linux 集成商 Redhat 已经在其 Enterprise 6.0 发布版本中取消了作为 Xen Domain0 的代码集成支持（Fedora 则始终没支持过 XEN），专心

推广其主导的 KVM 技术，导致很多使用 Redhat/Fedora 作为底层 OS 的用户也只得转投 KVM 的怀抱。同时 Ubuntu、Suse 和 Gentoo 等厂商也陆续在发行版中直接集成支持 KVM 虚拟化组件，推动了 KVM 更大范围的普及。显然在今后很长的一段时间内 KVM 和 Xen 还会在 Linux Server 平台上继续上演着夺面双雄这出大戏。

Hyper-V

最后来说说 Microsoft 的 Hyper-V。与其在个人终端市场所向披靡的境况截然相反，Windows 平台在服务器市场始终都未能成大气候。至于其原因业内分析众多，作者就不再献丑了。2008 年 7 月，Microsoft 推出了 Hyper-V 参与到服务器 Hypervisor 的战团中，虽然无论是名气还是市场份额都无法与前三款产品相比，但毕竟也在业内占据了一席之地，依靠背后 Windows 家族底蕴，在市场上也是小有斩获。

Hyper-V 有两种安装方式，一种是在现有的 Windows Server2008 系统上部署的安装包，另一种是包含了 Server2008 的完整安装包 Hyper-V Server，目前 Microsoft 推荐更多的是后一种安装方式。无论哪种方式，都需要一个 Server2008 作为 Parent Partition 与 Hyper-V 一起运行，管理硬件驱动程序。很明显 Hyper-V 也是以 Exokernel 方式运行，而 Server2008 则扮演着 Library OS 的角色，同时传说中 Hyper-V 500KB 的代码大小，也几乎只有 Exokernel 能够搞定。Hyper-V 与 Xen 架构大同小异，这里就不再多介绍了，贴张网上搞到的截图给大家简单看看。另外 Microsoft 的不开放也导致对其分析的无从下手。



Detailed architecture of Windows Server virtualization

在 Hyper-V 之上的 Guest OS 肯定是能跑几乎所有的 Windows 系统，但是 Linux 就有些麻烦了。目前经过 Windows 认证可以跑的就是 Suse10 SP3、Redhat Enterprise5 和 CentOS5.2

及其后续版本的几个发行版。而采用 Para Virtualization 方式时，理论上讲，由于 Hyper-V 的代码被合入了 Linux kernel 2.6.32 版本，因此采用其及后续内核版本的 Linux 发行版都可以作为 Guest OS 在 Hyper-V 上运行。另外通过 Xen 的 Satori 项目，还可以将修改过内核的支持 Xen PV 的操作系统在 Hyper-V 上作为 Guest OS 运行。

最后再提一下 Hyper-V 的限制，这可能也是其当前无法挤进主流 Hypervisor 的原因之一。基于 Windows Server2008（R2）版本的 Hyper-V 在 USB/声卡/光驱/显卡都存在着诸多使用限制和问题，同时在非 R2 版本上不支持迁移，总之给人以不够成熟稳定的感觉，与微软第一 OS 厂商的市场地位实不副名。只能期待着其基于下一代 Windows8 平台的 Hyper-V 能够解决上述问题限制，令人耳目一新。

OS 虚拟化技术

在上述虚拟化场景中 Guest OS 的运行可以有两种选择，一是无意识虚拟化，就是说安装后 Guest OS 仍然认为自己是工作在一台独立的物理服务器上。二是有意识虚拟化，既通过对 OS 内核的一些变动，使 Guest OS 了解自己是个虚拟化操作系统，可以执行一些虚拟化相关指令和系统调用。例如前面各种 PV 技术或 Lguest 技术中提到的 Modified OS 等。

无意识虚拟化 Guest OS 明显较有意识虚拟化 Guest OS 具有更广泛的应用场景，但二者都对整个计算机运行体系提出了更多的要求，导致了复杂的 Hypervisor 的出现。那么我们换个思路，看看能不能省些事，抛开 Hypervisor 搞下虚拟化。

前面的虚拟化场景尽量都要求 Guest OS 能够为 Windows 或 Linux 等任意 OS，这样可以更灵活的在物理服务器上进行虚拟机部署。但是从实际项目来说，我们完全可以要求在同一台物理服务器上部署的 Guest OS 都采用相同的操作系统，甚至是同一种发布版本的 OS，将不同类型的 Guest OS 分布部署在不同的物理服务器之上，尤其是在大型的数据中心项目中，这种方式更加有利于整体规划。例如在一台物理服务器上只部署多个 Redhat Linux 作为 Guest OS，而另一台物理服务器只部署 Windows Server 的 Guest OS。那么这时我们除了传统虚拟化方案外就有了新的选择，OS Level Virtualization（OS LV）技术。

OS LV 技术的思路是在操作系统之上，将其进程和资源进行隔离处理，同时提供给不同的用户使用，让不同的用户感觉自己是在操作一个独立的 VM，但其实各种调用和处理都是由底层 OS 完成的，只有程序进程的执行空间是相互独立的。这种技术提供的 VM 通常被形容为“容器 Container”，个人觉得这个词用得很贴切。有点儿类似 Windows Server 系统的多用户同时登陆操作，但 OS LV 更进一步，每个用户都拥有独立的文件系统与进程空间，相互隔离操作互不干扰。目前的 OS LV 还只有基于 Linux OS 的产品，如

Open VZ、Linux-VServer 和 FreeVPS，都能够支持 CPU、内存、I/O、存储和网络的配额独立分配，其中 Open VZ 还能提供如 vMotion 的 VM 迁移功能。相信 Microsoft 应该也有相似的 Windows Server 平台技术在开发中，估计发布也就是这两年的事情。

OS LV 的优点和缺点同样突出，优点是 Guest OS 的应用程序执行过程中不需要像其他虚拟化技术那样多经过 Guest OS 和 Hypervisor 两层的翻译调用，效率可以很高。据 Open VZ 称单物理服务器可以支持多达 300-500 个 VPS（就是 VM，其实这个数值可以简单理解为一个操作系统能够支持的进程总量），这是当前主流 Hypervisor 虚拟化技术所无法想象的。缺点则是目前能够支持的 Guest OS 操作系统有限，只有 Linux，而且还需要注意 Guest OS 和底层 OS 的版本需要尽量一致，导致其部署适用范围较窄。另外就是安全性与稳定性方面的顾虑，由于资源共享，单个 VPS 的进程故障是否真的如其宣传的不会影响到其他 VPS 甚至底层 OS，还需要更多更广泛的应用案例来验证。

其他虚拟化技术

再多介绍几个虚拟化技术，从实现上看类似于 OS LV 或 Type1 Hypervisor 模型，这些技术由于种种明显的限制目前大多应用在研究试验中，都没有知名商用产品流传于世。

首先是 UML（User-Mode Linux），同样是在 Linux 上运行 Linux，但是作为 Guest OS 的 Linux 需要像 Lguset 一样做些内核方面的改动，因此适用范围更窄，但其特点是可以在 Guest OS 上继续以 UML 运行更上层的 Guest OS。

然后是 CoLinux，专门用于在 Windows 上运行 Linux 的技术，可以使它们共享底层硬件资源，同样需要修改 Linux Guest OS 的内核，但每个 Windows 上只能运行一个 Guest OS。

其次是 WINE（Wine Is Not an Emulator），从名称就可以看出其非同一般的设计思路。WINE 是在 Linux 系统上运行 Windows 系统的一种方案，其没有对硬件操作层面做指令模拟，而是在 API 层面对 Guest Windows OS 的指令通过动态链接库 DLL 的形式翻译执行。很显然庞大的 Windows API 指令会对其运行造成严重的负担。

最后是 Cygwin，相信这个熟悉的同学更多一些，它是由 Redhat 开发的软件工具，用于在 Windows 上模拟 Linux 环境，以便调试类 Unix 软件，更类似于一种编程开发用环境。

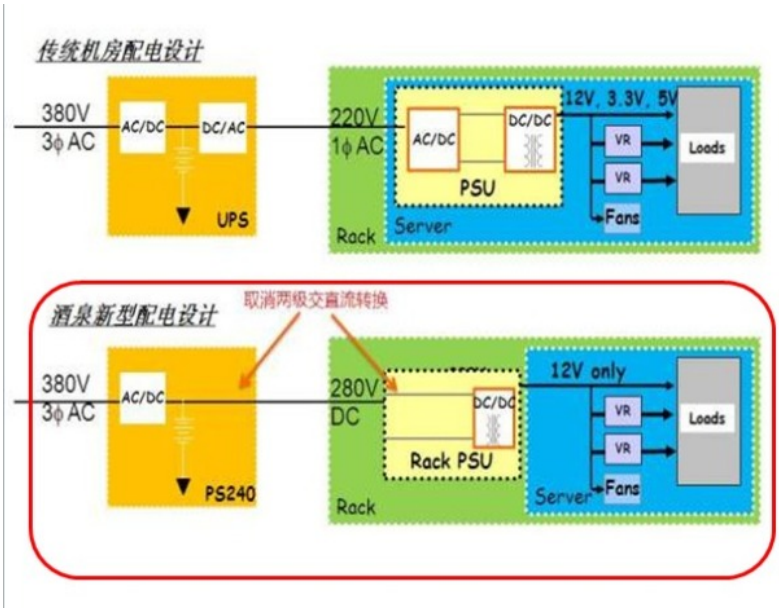
结束语与声明

服务器虚拟化技术就介绍这么多了，其实还有很多如 QEMU、VirtualBox 等重要的技术产品都可以拿出来再深入讲讲，但是个人能力有限就不多献丑了。本文的套路还是横向平铺，纵向深入实在是力有不及。

如果希望更好的理解服务器虚拟化技术，就需要对现代计算机技术的方方面面都有所涉猎，软的硬的最好都要看看。无可否认，计算机技术的出现已经对人类种群的发展产生了根本性的影响，再丰富的想象力也已跟不上时代日新月异的变化，无数几十甚至十几年前的梦想现在都已实现，今后还会迎来更多。身处在这个波澜壮阔的时代，身为一名技术人员，永远追不上技术更新的脚步，永远处于饥渴的学习状态，但任何不经意的发现都可能改变整个技术生态的发展，进而改变人类的历史。悲哉？幸哉？

浅谈 数据中心．高压直流

作者: @KPang 支付宝- 网络架构师
编辑: @陈怀临, 弯曲评论



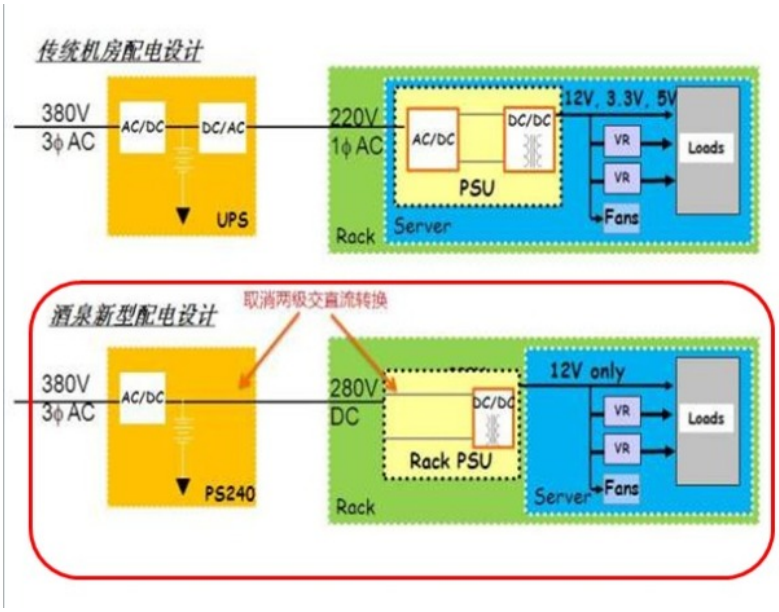
前言

本人只是普通的网络工程师，本文来源于日常学习积累笔记，仅供大家当成科普，文中尽可能量化数据，少用"很多""极大"类词汇，同时由于是笔记，文字均有出处，凡是雷同全因照抄，幸运的是不涉及版权纠纷。

一、传统数据中心采用UPS供电的缺点

传统的数据中心大都是由UPS实现掉电保护， 通常所有IT负载都要经过UPS供电，假定实际运行UPS的平均效率为90%（虽然目前UPS最高效率是可以达到95%以上，但UPS的效率和负载率有关，随着负载率的提升，效率才会变提升，因此正常情况下20%-40%负载率很难会达到最优的效率点。根据在运行UPS的实际测试数据，绝大多数情况下的效率不高于90%），即每100度电，经过UPS这个环节就白白损耗掉10%。不仅如此，由于 UPS自身散也需要空调交换带走热量，按数据中心典型PUE为1.8来算，那么UPS环节带来的总能耗达18%，很不节能。交流系统一般运行在低负荷下（通常在30%），实际运行效率较低，直流系统通过模块休眠可以提高负荷率，实际运行的效率较高。

二、采用240V高压直流的特点：



从上图的对比，我们可以清晰地看到：

1、减少变化级数，整体效率更高；

2、电池直挂在输出母线上，相当于提供另外一路备份，可靠性更高；

3、兼容现有绝大多数IT设备的高频开关电源，用电设备几乎不用任何更改，推广容易；

4、拓扑非常简单，可靠性提高；

5、高压直流系统为模块化热插拔设计，运维简单方便，减低运维成本

6、易于扩容

高压直流可靠性，扩展性，管理性，以及谐波等优势明显，随着数据中心技术的发展以及降低运营成本和节能减排的需求，高压直流技术不管是在节能、投资成本、可靠性以及运维便捷性等方面较传统的UPS都有明显优势，随着高压直流供电方案在大型的互联网数据中心等场合的越来越广泛应用，将逐步成为未来数据中心供电的趋势

【注1】： 上图"酒泉新型配电设计"是服务器和交换机为12VDC Input，服务器主板和交换机电源输入都是定制的。文章后面解释标准220VAC 的IT设备可以直接接240VDC。

【注2】：酒泉是一个曾经存在二年的项目组，酒泉取自火箭发射基地之意。

三、 为什么叫高压直流

高压直流并非按我国电力高低压区分标准划分，而是相对于广泛应用于通信领域的现有直流技术来称呼的，240VDC是传统48VDC的五倍电压。

四、 直流电压选择标准

用于IT/IP机房的高压直流供电，欧洲研究机构提出并实验的为直流350~400V标准，我国专家、学者曾提出以300~350V标准，我国最后制定并执行的是 并执行的是240V标准。

【注】：欧洲336V高压直流请参见附件。

五、 高压直流制定为 240V的原则依据

（1）直流技术应用于IT系统设备的供电后，设备运行可靠性应比原来的交流UPS 供电技术的可靠性大大提高

（2）直流电压标准替代原有交流UPS 供电技术，总体上对原有受电设备不作任何改造，为已建有交流UPS供电的系统升级改造成直流供电提供方便

（3）直流电压标准应满足IT设备运行及IT设备配套的相关设备的使用(如有设备配套的相关设备的使用传统直流48V或交流220V的设备），使所有设备能够统一应用一个直流供电技术标准

（4）直流电压标准应充分应用目前的整个产业链，减少现有产业链新流程

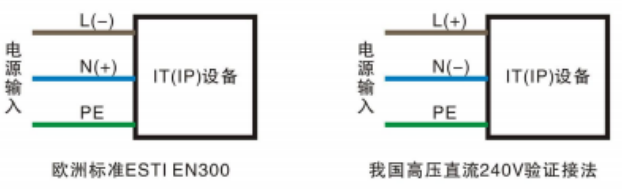
（5）直流电压标准应安全与节能兼顾（目前无论从理论还是实践上，均已证明IT设备采用高压直流供电技术，相比传统的交流UPS供电技术,既安全 又节能。

基于以上要求，我国最终制定了新型高压直流为240V电压标准

六、220VAC IT设备可用240VDC原理

1、 目前在网IT设备受电标准均为交流220V为标准，以L、N、PE三线制。若采用高压 240V 直流供电技术可直接应用，原有火线L和零线N转变为直流电源系统中的正、负极（其对应关系后述），原有PE保护地转变为直流电源系统中的直流保护地。

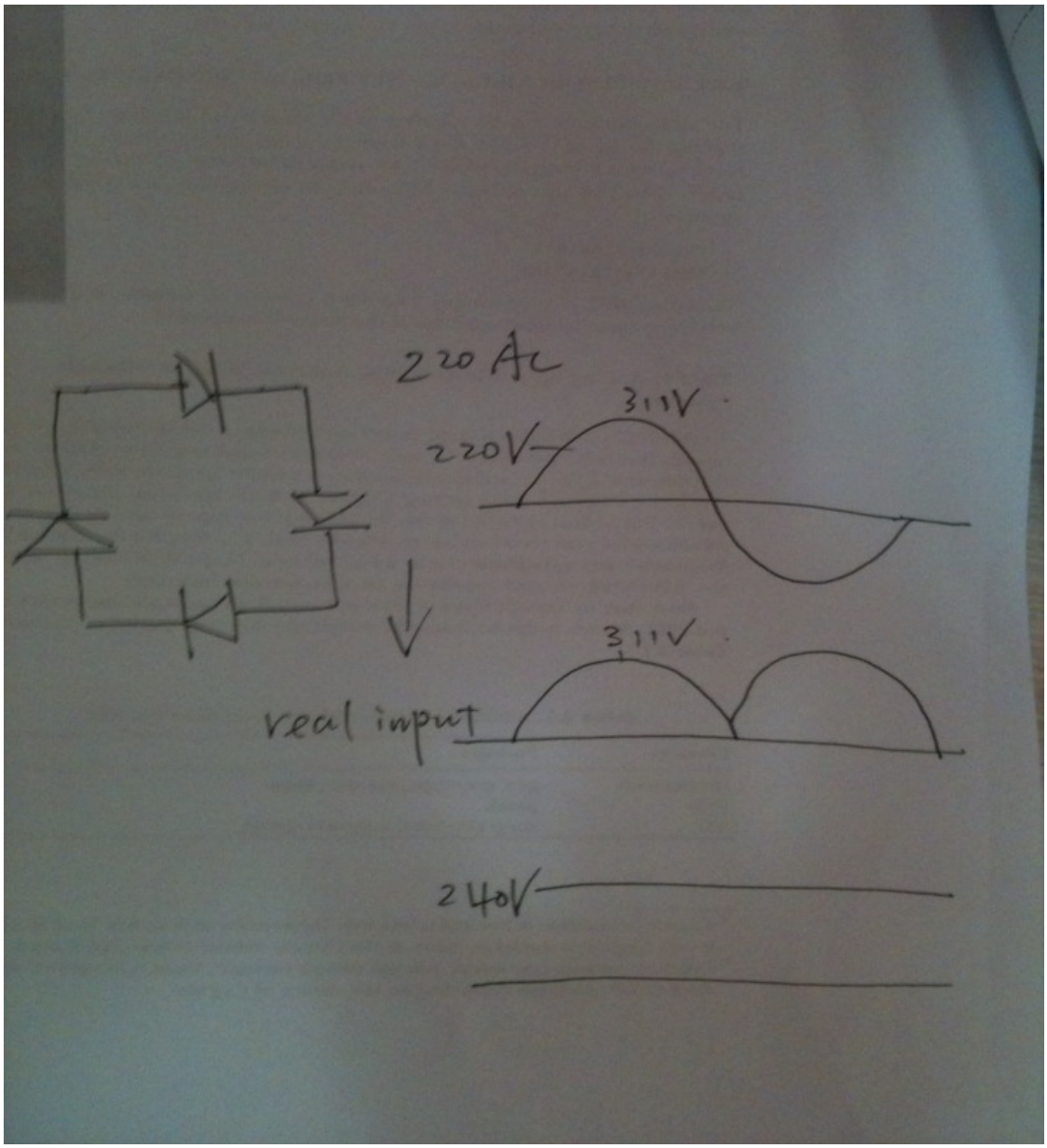
IT设备受电接口标准对应关系



【注】：为什么我国高压直流240V验证接法与现行欧洲标准 ESTI EN300 ESTI EN300在L、N上不同，主要是我国部分设备在原220V交流供电技术上，为节约成本，设计制造时，设备内部采用的整流仅为半波整流。对于全桥整流的 对于全桥整流的IT设备，直流的正负极与原有交流 设备，直流的正负极与原有交流L、N不存在对应关系。

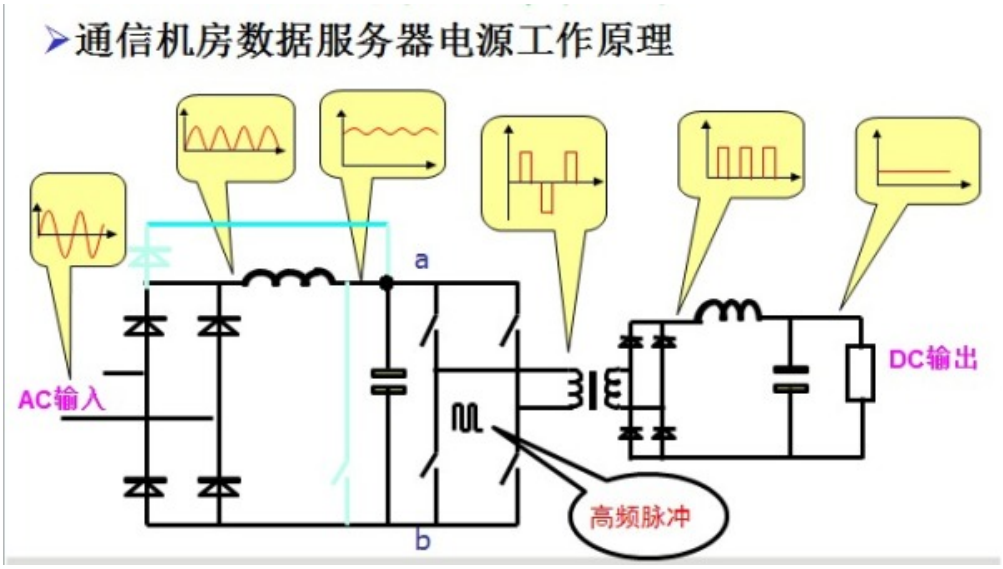
2、 UPS交流电压的供电范围是176—264VAC，对应的整流后的最低直流电压Uab=1.25*176V=220V，最高直流电压Uab=1.4*264=369.6V,工作效率最高的电压点是220VAC，对应的直流电压是Uab=1.25*220=275V，显然如果我们直接将直流电接到计算机服务器也是完全可以工作的，只要直流电压在220V—369V之间，选取2V一节120节电池，其均充电压是2.35*120=282VDC，浮充电压是2.25*120=270VDC。

结论：由于服务器电源是将UPS提供的交流电源先要整流成直流电，然后再将直流再变换成12V、5V或3.3V等低压直流给服务使用，而全桥二极管整流电路对直流电可以直接输入，只要直流电压能够达到220V以上就可以使用！



【注】：此照片拍自对我们科普时高压直流技术的推动者在讲解为什么220VAC可以直接用240VDC。

七、 HVDC为什么可以替代UPS？



八、行业使用情况及建设成本

1、中国电信全网用240V直流电源系统数已达到252个，直流总容量超过10万安培，功率为24000KW,相当于交流UPS30000KVA的供电能力。供电品质提升：可用度可由"六个9"向"十个9"逼近；带载率大幅提升；系统节电率在20%左右；可节约建设投资20%左右

2、中国几大互联网公司都已投入使用

3、短期建设成本无明显优势，但随着行业的普及使用，相关模块的量产会逐步降低相应成本。

4、哪些品牌服务器支持12VDC输入：DELL、中兴、浪潮

哪些品牌交换机支持12VDC输入：锐捷定制交换机

九、-48VDC来由：

使用早期的通讯网是电话网，话机的供电是由局端供电的，为了尽可能提高用户到局端的的距离选择了-48V，这个电压不能太高，因为36V是安全电压，采用太高的电压不安全，但电压也不能太低，太低的电压会因传输线的压降导致到用户端的电压过低而使设备无法工作。

因此选择一个不高也不低的电压作为电源系统的供电电压，（传说主要原因还是为了兼容早期设备，降低成本而考虑）-48V电源系统只是中国和大部分国家采用的通信电源标准并非所有国家都使用这个标准，如有的国家用-60V和-24V

十、对人体的安全性

本人亲自单手摸过，而且专家解释女的比男的更安全。

十一、此领域的人正在玩什么？

市电直供+HVDC

参考文献：

1、酒泉项目立项文档：朱华、Andy2(厉建宇)。
2、《数据中心的高压直流之路》 李典林、朱华：http://vdisk.weibo.com/s/50stJ
3、中恒培训材料
4、互联网信息

资料下载：

欧洲336V高压直流： http://vdisk.weibo.com/s/4-YbR
240V直流电源系统标准：http://vdisk.weibo.com/s/2zUpP
数据中心高效交流配电与直流配电的量化比较：http://vdisk.weibo.com/s/4--cB
交流电源的高压直流直供可行性分析：http://vdisk.weibo.com/s/50sn2
数据中心的高压直流之路：http://vdisk.weibo.com/s/50stJ

感谢我的师长Andy li及我的朋友朱华、李典林。感激酒泉的日子，如果还有一次失败，我仍这样选择。

网络处理器的大讨论

【编者注：近日，EZChip的100G　NP4的文章引起了广泛的评论。Of the 129个评论中，评论内容蕴含信息量之丰富，技术含量之深刻，业界动态之八卦，可以说是对NP，ASIC和多核讨论的精华。参与讨论的都是各大公司的技术骨干。现把相应的讨论整理如下，以方便读者阅读和查询。WP系统有点菜，不能查询评论的内容。在这次大讨论中涉及的公司与产品如下：EZChip　NP4；Xelerated: X11。另外，弯曲评论在发布的另外一篇文章Packet Processing at 100 Gbps and Beyond—Challenges and Perspectives中也对高性能线卡线速交换对各方面的要求作出了系统的阐述。有兴趣的读者可以进行一些阅读。陈首席在参与讨论大战中曾经抛出的MEM的Bandwidth必须是交换的6x的一说其实就是来自与这篇文献。在关于NPU，ASIC的比较上，其实工业界的产品是一个Hybrid的方式。特别是在高端系统上。对于中低端产品，NPU或者多核用在Pizza Box的架构系统上，问题是不大的。Again，任何一个技术和产品是为了来定位一个市场，服务于一个市场的。任何一个技术和产品都有其相应的问题。但是只要能够在相应的时间区域内，相应的地域，相应的市场，能确定商业上的成功，这就是一个好的设计。A good analogy is：天下美女多的很；但最适合你的才是最好的。当然，不同的时间，不同的阶段，这个最适合的可以不一样。。。在整理过程中，编者删除了一下没有内容的评论。】

1. 小小牛皮U 于 2010-02-01 6:23 pm edit

1月28日San Jose的seminar上，有一个问题是EZChip的speaker拒绝谈论的，那就是NP4在100G线速情况下的operations per packet. 这个很关键，决定了NP4是不是credible的100G NPU。啥业务都不能实现的NPU还不如选择博通的交换芯片。

另一个问题是RLDRAM。NP4不支持SRAM，据说是为了降低proposal的BOM cost。但是我实在是搞不明白怎么用RLDRAM去实现Statistics（包括meter）的功能。就算像其宣称的使用了什么cache技术，性能上也让我不能相信可以达到150Mpps。

还有一个疑问点是interlaken LA。这是未来高速查找接口的方向没错，但是目前没有量产的TCAM芯片啊。起码留个标准的NSE做备选吧？等个Interlaken LA的TCAM起码要到6月份吧，还仅NL一家，对大厂来说成本风险都太高了。

对于NP4的TM，也有一个问题。包指针不知道是存在哪里的？RLDRAM还是内部SRAM。如果是片内的话成本太高了，RLDRAM的话效率太低了。

5. 西门青云 于 2010-02-01 7:53 pm edit

关于SRAM和RLDRAM的比较
http://www.esmchina.com/
ART_8800069495_1400_0_3305_0_f48d9e43.HTM
貌似processor中NP4是第一个采用RLDRAM的。

6. 小小牛皮U 于 2010-02-01 7:59 pm edit

不是第一个，n年前X公司的40G NPU X11就采用了。但是貌似他们下一代的100G

processor已经转到DDR阵营了
7. 陈怀临 于 2010-02-01 8:31 pm edit
华为的Solar 2.0的100G也是指100单工。
8. atst 于 2010-02-01 9:14 pm edit
看了sram和rldram的比较，结论是rldram很适合高密度的包处理和大容量的表查找啊，对于统计数据来说，它的数据量不会太大，用rldram也没有问题。只是对于TM这种需要延迟的场合来说，qdrsram可能会更好。综合来看，选择rldram是高密度包，大容量表，低成本 的选择。
9. 小小牛皮U 于 2010-02-01 10:29 pm edit
拜托，RLDRAM做计数器延时太长了。
100G系统的计数器需要高速率短数据的存取操作，操作与操作之间的延时是有要求的。比如1次计数器累加是1次读+1次写。DRAM的Trc限制了访问速率，做不了高速系统。SRAM的2次操作之间只需要1个时钟周期就OK了。这就是为什么几乎所有计数器都是用SRAM实现的。DRAM一般用来实现线性表，hash表，以及包缓存——高密度，低成本的意义就在此。
而在DRAM现有的应用中，DDR表现就足够好了，而成本比RLDRAM低的多。看看现在DRAM市场有几家做RLDRAM的就知道了。
EZchip推出RLDRAM好像主要目的在于用它实现计数器。毕竟从成本上SRAM要贵的多。但是我实在搞不清楚怎么实现的。
10. 小小牛皮U 于 2010-02-01 10:39 pm edit
To 陈首席：Solar2.0还只是一大块画饼。
100G NPU目前样片的确实只有EZchip一家。但是我非常不看好他的实际工作性能。EZ的名气现在是不小，但是做的产品实在不敢恭维。当年NP2就号称20G了，但是到了NP3c才刚刚做到10G性能。现在怎么一下就整出个单片100G的NPU了？华为自研芯片也没这么快的进展啊。
我倒是看好另一家startup Xelerated，陈首席有空看一下。相较于EZchip的RISC架构，我更看好它的dataflow架构在高性能NPU市场上的应用。
11. MNSR 于 2010-02-01 11:03 pm edit
这里从首席和创始人开始，高手如云，Juniper的系列芯片，尤其是100G，有没有高手可以揭密一下？
12. system 于 2010-02-01 11:09 pm edit
关键这种芯片一年能卖出去几个，大厂都自己搞了（HUAWEI/ ZTE），小厂做不了。
13. aaa 于 2010-02-02 6:15 am edit
没有qdr接口，tcam只用于acl，不知道packet buffer，stats/policing，ip路由的性能能达到多少，不过可以想象会和吹嘘的100G差的不是一点。
14. CC 于 2010-02-02 7:34 am edit
楼上，既然TCAM能做ACL，那什么样的查找还能实现不了？
15. CCC 于 2010-02-02 8:30 am edit
to小小牛皮：1，从它的说明上看，人家明确说了TM和包查找都是DDR3实现，不是RLDRAM，也不是片内；2 interlaken是MAC接口；能否给介绍下做stats时，150M的

数据是怎么得到的？
16. kevin 于 2010-02-02 9:01 am edit
to 小小牛皮U
Xelerated什么时候成startup了。。。创立了有10来年了吧。。。从x10算也有6，7年了
17. 小小牛皮U 于 2010-02-02 9:10 am edit
答CCC：
1. 所有的processor都会采用DDR3实现TM Buffer，这点不会错的。但是我感兴趣的是packet pointer，那个实现会有不同。实现算法会直接影响到实际的TM性能。
2. NP4采用serial LA接口，即interlaken LA接600M(也许是700M)的TCAM的。请查实。业界还没有这样的TCAM。
3. 100GE的系统，对应到就是150Mpps。每包仅进行一次包统计不过分吧，这是最简单的统计需求吧，对应就是150M operations per second。如果要同时进行包长统计，就会要求300M的计数器性能。再算上Meter/Policing，会到600M。如果是某些复杂的场合，可能会要求更高。但是对最简单的业务，150M也是最低要求——一般情况下会要求300M。

18. 陈怀临 于 2010-02-02 9:11 am edit
QuantumFlow的DRAM是RLDRAM。
20. IE 于 2010-02-02 9:45 am edit
MNSR，trio也是类似的
21. MNSR 于 2010-02-02 10:43 am edit
IE: trio是NP还是ASIC还是什么其他架构？
22. aaa 于 2010-02-02 4:44 pm edit
CC 于 2010-02-02 7:34 am 楼上，既然TCAM能做ACL，那什么样的查找还能实现不了？
如果我又要acl又要路由呢？bcm的maybe可以用acl同时做，但不知道ezchip可不可以多个acl并行查找
23. 小小牛皮U 于 2010-02-02 5:58 pm edit
to 首席：我可算不上专业，仅知皮毛而已，多谢夸奖了。
to aaa：不知道NP4 prefer什么样的IPv4/IPv6路由实现？以前的NP2/NP3或者没有TCAM接口（NP2），或者即使有也不好用（NP3，理由不详述），所以是prefer它内部的一个算法引擎的。但是这个算法引擎性能是不能满足线速转发要求的，这个极大的制约了EZChip的芯片性能。——来自Cisco 内部未经证实不可靠不负责的消息称，之所以独家供货给C的NP3c能够达到10G线速，一个很重要的原因就是C帮EZChip在NP3c上解决了TCAM 接口的问题。当然还有其他的一些原因，不想多说了。
另外还有一个很大很大的话题：NPU（牛皮U:)）发展到更高的速率，难点到底在哪里？为什么ASIC总是很轻易的能够在线速能力上有所突破，而NPU就这么难？而在Router上NPU又是不可替代的。
曾和一些参与了HW 40G NPU开发的SE聊过这个，有些思路，但是还不很清晰，而且这

个话题实在是太大了，超出我的能力范围了，不敢随便大放厥词。
24. 小小牛皮U 于 2010-02-02 6:02 pm edit
to Kevin：没错，Xelerated成立很久了。不过到目前规模还很小，而且又没IPO，我是把它看成startup的。不过我对它的架构更有信心，在高端NPU的市场上，它的架构是能够保证高性能(20G/40G/100G)线速的。目前国内似乎它比EZchip更占优势
25. 黑猫 于 2010-02-02 8:51 pm edit
NP4集成了tm，故牺牲了np处理能力、代码空间和IP查表能力,故它定位是低端低成本。高端芯片是将部件分开，做成不同的芯片。君不见首席发的asr9k的性能测试报告是用1500字节长包，因为短包性能是远不如mx960.
26. atst 于 2010-02-02 9:16 pm edit
to 小小牛皮：请注意，NP4的brief_datasheet有明确的说明，interlaken接口是三个可选MAC接口，不是tcam接口；不知道你的 150M怎么得到的，按照线速打满，每包操作一次，都是最小包长64BYTE，应该是100G/64/8=195M，而说明上说它的RLDRAM的时钟为 533MHZ，RLDRAM的特点是写操作不需要等待，那么533的时钟既意味着533M操作；
to aaa：TCAM查找当然都是并行的，不分种类的。不过它的说明文档说它的路由查找一般用DDR实现，除非你有特别需求采用TCAM。另外它号称自己的二叉树查找为线速查找；
27. 杰克 于 2010-02-02 9:24 pm edit
号称自己的二叉树查找为线速查找；
那说明它的二叉树层数固定。
28. bbb 于 2010-02-02 10:04 pm edit
NP4 RLDRAM采用的是SIO的，RL=4延时不会低于250M的QDR II，用SRAM最大才72Mb估计计数器的数量也可能不够用了
29. 小小牛皮U 于 2010-02-02 10:11 pm edit
to atst：关于NP4的TCAM接口，我这里不想再继续下去了。它确实是Interlaken LA，或者你叫他Serial LA interface也可。无需争论，事实如此。
关于100GE的包速率，你的计算有误，Ethernet包64B在线路上可不是只有64B。当然精确值是150Mpps不到一点，但是一般都是这么说的。
RLDRAM的问题我不是很擅长，但是很愿意和各位高人详细讨论一下，所谓答疑解惑，皆为吾师。
30. ilovebgp4 于 2010-02-02 10:23 pm edit
To atst
不能用100G/64/8,以太网帧之间有12bytes的IFG和8bytes的Preamble + SOF，因此一个以太网帧最少要占用64+12+8=84bytes
100G/84/8=148,809,523.8pps,约等于150Mpps
31. bbb 于 2010-02-02 10:47 pm edit
以250M SRAM为例写延时WL=0 读延时RL=2 533M RLDRAM WL=5 RL=4，一次计数器操作为先读再写，写操作为post，WL的影响可以忽略，关键是RL，533M的RLDRAM延时要小于250M的SRAM，使用的 RLDRAM为SIO的也就是读写数据线是分

开的与SRAM一致也就是相当于QDR了，唯一有影响的就是RLDRAM连续对同一bank进行操作有个 tRC=4，这个就看RP了，冲突概率1/64吧.算下来和250M SRAM差不多了
32. 小小牛皮U 于 2010-02-02 10:53 pm edit
我对RLDRAM的理解：
RLDRAM和传统的DRAM的区别主要是指由物理上的4 bank变成了8 bank，以及时钟的上下沿均可传输数据，所以极大的缩短了tRC。这也是其Reduced Latency命名的由来。但是即使如此，tRC还是存在的，这是DRAM的本质决定的，只要有tRC，就会影响带宽利用率。
RLDRAM由于采用了8 bank，所以可以采用Round Robin机制，保证顺序访问不同bank时tRC可以忽略。但是对于随机访问，必须考虑tRC，因为连续2个操作是可能会落到同一个bank上的。
至于所谓的simultaneously R&W，是指R&W在不同Bank上的。因此对statistics这样的R/W ratio of 1:1的访问并不适用。
33. 小小牛皮U 于 2010-02-02 11:56 pm edit
to bbb：对statistics的读写操作，是sequentially的。因此必须考虑R/W之间的tRC，我的理解对否？
然后是R/W和R/W之间的tRC。对于statistics这样的随机访问，这也是无解的。绝对不是RP的问题。系统设计时你就得考虑到，你不可能和 customer说sometimes wire speed。如果这样的话，WL也是不能忽略的。这样整个带宽的利用率就是20%多——这个计算可能有误，不太会算这个，请指正。
1次计数器操作对应到2个operation，因此533MHz的RLDRAM只能做到70MHz左右的计数器。
而且我理解的目前RLDRAM最高到400Mhz吧？
我认为statistics不可能用RLDRAM实现的，这条路走不通。
34. bbb 于 2010-02-03 12:41 am edit
对statistics的读写操作，是sequentially的。因此必须考虑R/W之间的tRC，我的理解对否？
因为是先读再写，重点是什么时候能读回来，写操作是post并不需要颗粒的reply，写的延时是不需要关心的。要考虑tRC的就只有连续的两次 statistics的读写操作落在同一个bank上，进一步说就是连续的两个statistics读操作落在了同一个bank。
35. bbb 于 2010-02-03 12:42 am edit
RLDRAM与DRAM的区别主要就是在tRC，DRAM2 tRC在50多ns，RLDRAM做到了4个tck，能不能做statistics重点是在SIO还是CIO，statistics操作决定了总线上会是频繁的读写互换操作，使用SIO要好很多
36. 数通人 于 2010-02-03 1:24 am edit
http://www.cypress.com/?docID=9310,这个是cypress的一个SRAM vs RLDRAM的PPT，说的很清楚了。在1:1读写比例下，对于4和8字的突发长度，SIO RLDRAM II可以达到100%的总线使用率，在最极端情况下是50%利用率。所以用在NP的统计上应该是没问题的。况且通过预先设计，可以让容易冲突的表项位于不同的bank来减少冲突的概率。

37. 小强 于 2010-02-26 1:21 am edit
to 数通人：其实对于RLDRAM能否做counter和meter的问题，我想没有人说他不能做，但这里面的问题是到底能做到什么样的一个效率。EZ号称双工50G，最短以太网报文75Mpps，我们关注的是用RLDRAM一条流在最恶劣的情况下能够做几次counter，又能做几次meter 呢？meter需要更多的周期。据我所知一般情况下，一条复杂流需要2次meter+4次counter，我想知道的时候EZ用RLDRAM能否做到这样的需求呢？当然EZ可以说这种情况下他不线速，这也是他们芯片的一贯作风。等到满足他所有线速条件的时候，要么发现这个业务已经简单到毫无竞争力了，要么就发现时间已经过去4，5年了，要么就是用多片级联来完成本该一片就能完成的工作。（不是攻击，只是看到了太多这样的例子了，从NP2到NP3,衷心希望 NP4这次不会让大家失望，但是谁又能保证呢？我相信他们自己都不知道，目前看，做不到的可能性更大）
从技术角度看的话，能不能请各位大侠根据你们的理解计算一下RLDRAM在EZ上可以做到的counter和meter的能力到底是多少？没必要争论能不能做，而是要定量的分析它能做到什么程度。作为系统设计和芯片选型来讲，一个系统工程师关注的是它能给产品带来什么样的竞争力而不是一个芯片能不能做某个功能
还有就是说用RLDRAM所谓的避免冲突的问题，计数器怎么避免冲突呢？他是完全随机的，你无法预测到下一个报文到底要访问哪个BANK的计数器。如果使用HASH等方法的话，也一样无法保证散列的很开，能否请大侠解释一下用什么样的"预先设计"来避免冲突呢？就算能，我相信也会给设计带来非常非常的难度吧。而使用RLDRAM做统计的唯一一个好处就是多支持一些计数器罢了，但是据我所知X的芯片有至少3种方案来支持百万级以上的计数器。而且设计简单灵活。个人认为用RLDRAM做计数器就算能做也是得不偿失
再说说X的芯片，至少在城域和路由器的领域里面，我非常推崇，5个人认为比EZ更具优势和活力。那么说说X的计数器，在下一代产品里面他的计数器可以做到非常灵活，而且没有任何限制（因为使用QDR SRAM）。如果用户要支持百万级counter和meter的话，X芯片已有至少3种解决方案，其中也包括使用DRAM的方案。由用户选择到底在乎成本还是设计难度。但是不得不说的是，X的最复杂的那套方案也要比EZ目前的方案设计起来更简单。在这里不方便说太多，有兴趣的同学我们可以私下讨论：）
呵呵，没有个人感情倾向，不要拍我啊。我就是觉得NPU本身没有好坏之分，设计也没有好坏之分，所有的比较都应该放在某个环境下进行比较。我们选用NPU的时候都要放在系统设计和产品需求的角度评价，单纯说能不能做是没有太多意义的：）
38. 理客 于 2010-02-26 8:52 am edit
小强很强，问一下：EZCHIP如此不济，为什么C/J都选它，是因为便宜还是别的什么原因？
39. 阿尔吉侬 于 2010-02-26 7:07 pm edit
NPU的counter设计并不是一个难点，EZ不可能连accounting的线速都做不到。
典型的read-modify-write对外存的首要要求是带宽满足。
400MHz RLDRAM， 理论上就能做到400M次的accounting 操作 （这里不考虑总线宽度限制，可多片并联）. 即使考虑到r/w带宽效率，对150Mpps的业务也能对付。而且EZ似乎有两个RLDRAM接口，带宽又可以double了。

至于bank切换问题，ASIC设计一般是把相关的counter放在相邻地址来改善性能。因为同一个包总会关联到多个计数器，可以在一定程度上改善带宽效率。
至于metering，ACL相关的对latency就比较敏感了。EZ真把它放在外存了？我个人不太相信。既不现实，也没必要。这部分占的资源又不大，为什么不用内存？
40. Gary 于 2010-02-26 9:07 pm edit
Huawei的SE认为X比E好啊，看来EZCHIP的Marketing在Huawei没有做好工作.
41. 数通人 于 2010-02-27 12:52 am edit
to小强
阿尔吉侬兄已经讲的很清楚了，SIO接口RLDRAM就是为了统计设计的，对于统计的RMW模式，带宽利用率是100%，和SRAM是没有差别的。
NPU的设计在我看来，一大难点还在QOS。受限于NL的320bit最大字长，做基于IPv6的MQC的时候，光是源目的地址就至少要用去256+8，留给其他fields的bits非常少，无法做复杂的policy matching。假如做IPv6地址压缩，会浪费非常多的表项。还有就是policy class的deny jump问题也是无法解决，25个class，每组3个deny就有3^25的可能，能让ASR1K死循环到没响应，而基于纯软件的7200则没有这种问题。NL基于功耗考虑，短时间也不可能再增加TCAM字长，或者增加内部BANK。反正TCAM这个东东是限制多多，让人又爱又恨啊。
42. 小强 于 2010-02-27 4:35 am edit
to 理客:C和J选用EZ已经是过去的事情了。J已经不用EZ了，除了他们准备自研芯片外，还有一点很重要的就是他们用了EZ后，产品就没有怎么大卖过，饱受诟病。对于C用EZ，前面我记得小小牛皮U已经提到了一点，C承诺帮EZ做了很多改进，而这些是不准开放给其他客户的。而且有非常可靠的消息说C对EZ相当的不满意，说出这话的是非常高的领导。别问我是谁的，您可以通过其他渠道去多了解下。EZ是上市公司，又是美国公司，有些公司用他们的芯片当然可以理解，难道能拿这个东西说事？要看用它的产品有没有成功大卖的，给客户带来什么了。不仅仅C和J，据我所知国内的H和Z以及H3C也有个别产品用过，但是请问哪个公司的产品大卖了？打开他们的单板你会发现上面有4pcs，6pcs甚至8pcs的NPx，这样的产品能大卖才怪。而据我所知X芯片作出的产品现在都是大卖吧。不要说谁用过，要想想用了都是什么样的结果。
To Gary，恰恰相反，EZ作出的产品性能这么不好，仍然有很多厂商在关注他，仍然有很多同学喜欢他，如各位。就证明他的市场做的很好，如果不是靠市场的话，我想结果完全不是这样了。
To 阿尔吉农，meter和ACL是两个概念，meter当然是用RAM做的（不管是SRAM还是所谓的RLDRAM），ACL里面存的是是否做meter以及meter ID，真正的meter实现是要靠令牌桶的，也就是RFC2697，RFC2698已经MEF10.1所描述的机制，这是需要用RAM做的。请查证，如果还认为不是的话，我们可以再讨论。还有就是阿尔吉侬大侠说一个周期做一次counter？太夸张了吧？连QDR SRAM都无法做到，还需要2个周期呢，小小RLDRAM能一个周期？请给点专业的理论分析好吗？我学习一下。还有就是再给我培训下一次meter要多少个周期：)
To 数通人&阿尔吉侬，我的意思是请不要再去说什么能不能做，如阿尔吉侬数哦的400M应付150M没问题之类的描述。我一再说，能不能做根本不是问题，请从业务角度

出发，请问2次meter+4次counterEZ如何保证线速?请给个计算公式好吗？我相信大家这么推崇EZ，最EZ的性能计算应该了如指掌吧？非常乐意和大家讨论
至于EZ有两个RLDRAM接口的问题，请问各位大侠知道X有几个统计接口吗？知道X的灵活度和设计理念吗？我觉得如果大家只知道一个的话，那就不太公平了，比较下才有好坏。
to all，我上次说EZ用RLDRAM做统计，应该是会选出说支持大量counter和meter（也就是所谓的statics），并且成本低。但是我相信EZ一定用了很多空间换时间的手段，请不要迷信RLDRAM的效率，用起来才知道，如果 RLDRAM能用来直接做statistic，各大芯片商早就，说X不用大家可能觉得瑞典人想不到，那请问H的自研芯片用RLDRAM做统计了吗？以前 X11可以连RLDRAM，H和Z有用RLDRAM做统计了吗？为什么都用QDR SRAM（甚至DDR SRAM都无法满足要求，效率太低）。所以，EZ如果用RLDRAM就一定是做了很多限制在里面，用的时候会告诉你们的，这是他们一贯的做法。曾经有一个总工级的人物说过，他也不知道EZ为啥用RLDRAM做counter和meter而且完全不知道如何保证线速（应该说是一个正在看EZ的客户，这总改有点说服力吧？难道堂堂总工会想不到阿尔吉侬所谓的400M完成150M？没这么简单的）。如果EZ用了我说的空间换时间的话，那请各位再想想，他所谓的成本优势还有什么？一个576M的RLDRAM和一个36M的QDR SRAM比，成本可想而知。而且再加上设计的难度。设计成本，维护成本不算吗？
to 数通人，你说的ACL问题，的确是这样的，但是现在IPv6的应用并不广，而且IPv6的ACL规则也没有什么定论，也没有说一定要用IP的全字段做 ACL，这还是一个很久远的事情呢。而且ACL的实现方式本来就有很多。有的是L2-L4的ACL有的是用L2或L3或L4的ACL。我相信就算用 IPv6的IP全字段的话，也可以用L2或L3或L4的ACL。这些需求的明确需要很长的时间，而且客户需求之上，客户会在性能，功能，成本之间做权衡的。而且TCAM达到640bit或者480bit以后也会是一种确实的
不是要与大家为敌啊，是想讨论，多向各位大侠学习学习，请多指教
43. 小强 于 2010-02-27 4:50 am edit
原来说两个接口问题的是阿尔吉侬不是数通人，不好意思啊。
还有一个阿尔吉侬说的meter要用内存的问题，呵呵，请您查实。首先我觉得您可能搞错了meter的概念吧？我想问问EZ的内存有多大？客户对meter的需求又有多大？怎么可能用内存？而且meter还就是用外存来实现的。
对于ACL，EZ有个Tree算法，但不是证明他用内存完成的，还是那个问题，他的内存有多大？你的ACL需求是多少？好好计算下再说。说到Tree算法，又有好多要说啊。用过EZ芯片的人都知道Tree不用则已，用则线速必失（我说的是在做ACL等大Key值和IP等大规格表象的时候）。以前NP2和 NP3是不能外接TCAM的（不能说不能接，是接了就有一个10G口不能用，共享PIN的），他们一直号称他们的Tree很牛，“忽悠”了多少客户。最后客户一用，脸就长了。才发现根本不是那么回事。所以就优化再优化，压缩再压缩，最后还是要多片级联来完成本该一片能完成的工作（因为一开始宣称的老好了！）。如果他的Tree很好的话，我想问问他们为什么在NP4的时候增加了TCAM接口，别增加不就得了吗？所以阿尔吉侬大侠，请重新审视一下你的内存说法吧。不知道您的不现实也没必要是基于什么得出的结论啊？您用EZ设计了几款产品？用X设计了几款产品呢？

还是那句话，不要定性的说，咱们都专业点，拿数据说话，说出来能做多少，怎么做的，效率多少，这样的讨论才有意思，否则就变成争吵了，就没劲了
45. 理客 于 2010-02-27 8:26 am edit
小强的专业很强，佩服。那是不是可以这样理解：C/J选择EZCHIP是个错误？能否再分析一些为什么C/J这么强，怎么犯了这么大的错误？
46. 陈怀临 于 2010-02-27 9:16 am edit
线速 与 DRAM带宽的问题：
我的理解是，一个芯片或系统的Packet Mem的带宽应该是 6x的Traffic的Rate，系统才能确保线速。
否则，好像有问题。。。这也是为什么NP里往往有多个Memory Controller。。。。
47. 数通人 于 2010-02-27 9:37 am edit
to小强
不好意思，现在玩的是20G的接入设备，我们的NPU用的还是RLDRAM做统计，所以也没觉得非要QDR SRAM不可。不过经你一提醒，在100G的环境下，确实统计也会成为一个瓶颈。可能真得要SRAM或者很多个DRAM控制器来实现。暂时没机会玩 NP4，手里也没有它的DS，所以也不敢妄下结论。
EZchip的好卖除了会做市场外，也许也和Ran Giladi的那本书有些关系吧，开放了构架信息，也同化了TX们的思想。至少暂时EZ的100G是独此一家sample，HX还得再等等，后面还有 TPACK加入战团。至于HW和J弃用EZ，除了性能上的考虑，个人觉得也有EZ和思科走的实在太近了（主要是9K)，怕EZ变成下一个 Greenfield的考虑吧。
IPv6的QoS的确不是什么很久以后的问题了，ASR1K的DDTS库里已经有相应的CFD了，客户对QoS的需求总会超出程序员的想象，policy-class能配啥，客户都敢配。而且IPv6的next header机制也不那么好玩。
48. 小强 于 2010-02-27 6:55 pm edit
to 陈首席：我可不强，可能是说的“嗓门”大了点，给大家带来不好的感觉请多原谅啊。
对于陈首席说的packet mem要是packet rate的六倍以上，我不太确定明白了你的意思，您指得packet mem是包存储还是只表项存储还是专指最近大家讨论的counter和meter啊？如果是包存储的话，这个好像不太对，由于包存储一般只有一次读和写，至于微码处理所得到的各种信息都是存在内存里最后统一修改报文的，所以不需要6倍；如果是表项存储的话，那么6倍显然是不够的，因为线速的评估是根据最长流程来设计的，一般最长流程对外存表项的查找需要15次以上，而且还要根据数据线的宽度来计算周期，如返回值是64bit的表项一个周期，返回值是 128bit的表项要两个周期，以此类推等等，所以基本上算下来，存储表项的外存的读写效率应该在26x包速率左右（最复杂流程）；如果是针对meter 和counter的话，6倍应该也不够，拿QDR SRAM来讲，一个counter需要2个周期，一个meter需要4个周期，最复杂的2个meter+4个counter为例的话，需要16个周期左右。当然有些芯片厂商会做一些优化，如X，他们不需要这么多。
我不知道首席说的跟我说的是不是一个问题，写了点个人意见，见笑了
49. 小强 于 2010-02-27 7:07 pm edit
to 理客：我觉得最好不要这样讨论，没意思。咱们都用数据说话行吗？不过我倒是可以针对您的问题说几句。

C和J是不是犯了很大的错误我不能评价，谁会承认自己错了呢？但是错不错，只有里面的人自己心里清楚。今年有个同事跟我说，他回家想给他爸买瓶茅台，但是发现很多茅台都很贵，后来发先茅台醇挺便宜，他就买了（他以为这也是茅台），可是回家别人都说这不是茅台，但是他会告诉所有人他买错了吗？他还是坚持说其实还是不错的，口感还可以，等等。就好象卡片机里本来佳能的非常好，但是由于索尼的外观让你觉得很好，你就买了，后来发现性能不行，就连存储卡都不能和别人共享的时候，你会承认你买错了吗？你依然会说，其实外观还是很时尚的。您问的这个问题太不专业了。
上面是用不专业的方法来回答你的问题，下面举几个例子。
C和J是否失败，我不多说了，因为我说多了，您也查证不了，我相信您的渠道不够多，否则我前面已经说了，C公司的非常高层领导已经很愤怒与E了，您为什么不去考证下？还来问我这个问题？我说几个国内的吧。我说了国内有几个公司的几款产品用过E了，但是据我所知，现在已经有至少3个产品主动考虑更换到X了，虽然有的由于进度问题，需求迫切程度问题等更换速度不是很快，可能直接切换到下一代也就是HX。但是不能不说现在很多已经无法忍受E的芯片了。我想您的能力应该可以验证到这些信息吧？您用在这质问我的时间去多调查下岂不是对大家更有帮助？
还有啊，C和J是很强大？但是请看看趋势吧，好吗？强大是因为E变得强大吗？J已经在走下坡路了，C的强大目前看毋庸置疑，但是主要是他的标准主导，产品前瞻性，客户忠诚度等等吧。而且C根本靠的就不是商用NPU打天下，他们有自己的芯片。所以就算E选错了，对他们根本没有任何创伤。我始终相信H和Z会慢慢强大起来，C曾经说过H是他们21世纪唯一的竞争对手，请不要再说C和J有多强，不要妄自菲薄
50. 陈怀临 于 2010-02-27 7:18 pm edit
咔咔咔，小强强好样的！：）就是要给客客hard time…
是的，ezchip会淡出c和j.edge上ez确实不够了。但enterprise products还是很大的。。。
51. 小强 于 2010-02-27 7:26 pm edit
to 数通人：原来是个误会啊，这里面谈论的都是100G芯片，所以我没想到您再说20G，不好意思，语言上有些犀利请见谅啊。其实我还是那句话，我没说过EZ 用RLDRAM做不了100G的meter和counter，只是觉得他可能做不了多少次，会在复杂业务的时候成为线速瓶颈，我也希望有大牛给大家解惑一下，说明EZ的统计不是瓶颈，大家学习下：）
其实EZ的架构和以前的Rainer以及2800的架构都有比较相似的地方，各大厂商接受起来比较容易，而且EZ有了自己的改进，所以开始更容易被人接受，尤其是底层工程师。而X的芯片是流水线式的，连包缓存都不需要，恰恰是他的颠覆性的架构给他带来了只要业务排开必线速的巨大优势，同时也恰恰是这个颠覆性的架构不太容易说服客户去接受他。加之公司小，没有上市，等等所谓公司级风险一直被对手打压，客户也必然会受影响。而且在07年的确经历过一段最困难的阶段，加之那个时候他们的中国销售团队刚刚成立，很多东西不系统，让EZ钻了不少空子。可是事情就是这样的，你可以在别人虚弱的时候钻空子，但是你钻了空子只是争取了一个证明自己的机会，当你把握不住的话，大家的眼睛是雪亮的，你一再让你的客户，尤其是当年选你芯片的人在仕途上折戟沉沙，那你必然要被替换掉。太多的就不多说了：）

说到HX的进度问题，这可能是EZ的NP4现在在市场上没有完全败下来的唯一理由吧。其实根据可靠不能再可靠的消息是HX会在E的一到两个月后 sample，这个时间并不算长，很容易被接受，只是X以前的进度延了太多次了，导致现在客户连这个最可靠的消息也不愿意轻易相信了，所以才会同时关注下 E，尤其是在商务上，各个公司不可能只看一家，否则商务太被动。但是胜败会在HX出来后，各家公司选型结果出来后有所定论的。其实也有很多公司怕X成为下一个Greenfield，这是E一直在客户那边打压X的唯一一个把柄，因为X小，而在产品方面E基本找不出X任何缺点。尽管曾经有X的员工去了E，但是仍然找不出缺点，从来都是从商务上进行攻击。不过如果X公司的人看到您的帖子后，以后也可以在客户那边打压一下E了，呵呵，你内X公司提供了一个思路啊，他也有可能成为Greenfield啊：)
至于ACL，您说的是对的，客户的确啥都敢用，只要有人能做出来，这也正是为什么各个厂家都在追求新特性的原因，这个我同意。我还是那句话，最终的运营商在性能，功能，成本上做权衡的。而且针对IPv6的ACL，各大TCAM厂商一定会出480bit或640bits的TCAM的。
52. 小强 于 2010-02-27 7:42 pm edit
to 首席：多谢您的权威性总结。没错，enterprise prudct是很大。而且企业网对线速没啥严格的要求，要求的是业务全。所以EZ的低性能在企业网上看不太出来，或者说可以勉强接受。
因为您的这个帖子中有一句话"NP4芯片的定位是在城域网（MAN）为主要市场，具体产品方向为Carrier Ethernet Switch和Router等"，据我所知，城域以太网和企业网在各个公司都是两条独立的产品线。所以我主要是不认同他在城域以太网交换机和路由器可以跟 X相提并论。
对于企业网，只能说X和E各有千秋吧，其实E能做的，X都能做。以前的X11有一个弱点就是要么所有业务线速，要么所有业务都不线速。这就导致了有的业务客户可以接受不线速，但是有的业务必须线速的需求无法满足，而NP3刚好可以，因为他压根就不能保证所有也无权线速。但是X即将推出的HX和AX已经解决了这个问题，他可以根据业务的不同类型来决定业务是线速还是不线速，要换回还是不换回，不仅仅如此，他们还有专门的调度机制来调度保证线速的业务和不保证线速的业务，充分的保证了资源利用率。并且使得产品形态更合理。而且据我所知某超大公司的某企业网交换机已经评估过了HX和AX，并且决定等待HX出来了。所以说在企业网级别，H和E的一场斗争就将展开。而且我依然看好H，因为实在不知道E哪方面可以赢X，或者哪位大侠知道可以给小弟指点下：)
说到X相对于E的弱点，有一个就是X只能解析处理报文的钱256字节，而E可以全解析。但是在城域网和企业网里，256字节足够了。这也不算是啥缺点
这里我还说到了一个AX，据我所知AX是针对PON和接入交换机设计的产品，因为以前X在这个市场上没有关注，现在他们的AX横空出世，准备和各个接入市场的对手们较量一番，当然其中包括很多很多对手，NPU领域的对手也包括E。但是有于AX真的很强大，所以很多以太网交换机以及OTN的设备都可以用 AX，很多厂商都在关注。AX尚且如此，更何况HX。
我关注X和E的芯片可以追溯到4年前了，我一直觉得他们都很不错，但是非要比较下的话，我认为X完胜E，当然话说的有点死，但是至少在我看到的领域，我设计的产品的领

域里面看X完胜
55. 理客 于 2010-02-27 10:04 pm edit
小强同学，感谢回复，我是来学习的，对您的发言都能理解，我拿不出具体技术分析数据。没有任何challenge您的意思，您误会了，因为这是我的外行，我也基本上没有更多的渠道。您太谦虚了
希望EZCHIP方面也有强人来challenge一下我们的小强同学，还是已经被PK得此时无声胜有声了:)
C在某种程度上在向下走，H确实也在猛追，鹿死谁手，拭目以待
56. Multithreaded 于 2010-02-27 10:47 pm edit
没想到现在还有NPU的粉丝，在这里讨论X和E的关系。刺激一下小强同学！
X的产品随好，但有多少人会在流水线上用汇编编程哪? 玩好一个100多级的流水线可不是一件容易的事。
从市场的角度，E和X都没戏。C&J都有自己的100G的芯片，只有H&Z的一点市场。最有潜力的是Intel IXP-2400类似的东西，在无线的接入层上。
57. 小强 于 2010-02-28 3:34 am edit
to 首席：我是志愿军：)
58. 小强 于 2010-02-28 3:46 am edit
to Multithreaded：NPU自然有他有魅力的地方，就连不会唱歌的人唱出五音不全的歌都有数不尽的"歌迷"和粉丝，更何况还有很多可取之处的NPU呢？
还有就是您说的100多级流水，此言一出，就知道你完全是个外行，至少对X的流水线结构完全是个外行，外到十万八千里了，请问您的100多级流水是指什么啊？小强不才可以跟您讨论下，呵呵
从市场上看的话，您说对了一半吧，C早就有自己的芯片，这谁都知道，可是他的NPU不是很强，当年收购Greenfield也是出于此原因，可惜 Greenfield实在不行，他现在还在寻求和商用NPU的合作，X和E都有很大的机会，只是E先走了一步，但是我也说过现在里面对E也很不满意，所以 X也是机会很大，现在鹿死谁手还不清楚，当然里面还有很多政治因素了。当然我也说过，C不会让NPU占他太大的比重的，但是NPU一定要有，NPU相对于 ASIC的优势不用我絮叨吧？大家都心里清楚。敢问现在哪家大公司不用NPU？商用ASIC和FPGA都无法满足业务的灵活性和新需求的爆发式增长。对于 J的自研芯片，我们只能拭目以待了。
我再补充您两句，不仅仅C和J，H和Z也在做自己的NPU，而且不久您可能就会听到更多的消息，但是这又能说明什么呢？能说明X和E没机会了吗？NPU做出来，一版成功的从未有过，万事需要时间的积累的。
有一点我赞同，现在NPU的市场被压缩的很厉害了，正所谓适者生存，E和X也在积极转型中。
您所谓的2400在无线接入，没错，还有LSI（Agere）的芯片，他们是走介入路线的，接入的量永远是很大的，这是必然。但是并不是说只有他们有优势。您研究过E和X在接入市场上的动作吗？
我前面的帖子已经说过了，尤其是X现在正在面向接入市场，尤其是PON，现在PON都在考虑转向NPU，现在的问题在于NPU和PON MAC的合作，新的战场已经硝烟弥漫了。您

的X和E都没戏的结论似乎停留在2年前了吧。感觉不能说您说的不对，但是的确觉得说的差强人意了。
59. 小强 于 2010-02-28 3:48 am edit
to 理客：其实我也是想学习的，就是想大家都用各自知道的专业知识共同讨论共同进步。：）不吵不相识。马克思和恩格斯当年也是吵到差点动刀子，但是人家是无产阶级战士，创造出了最牛逼的哲学。争吵出哲学：）
60. Multithreaded 于 2010-02-28 8:20 am edit
In general a pipeline consists of a numer of pipeline stages. X's product may have several pipelines and each pipeline has more than 100 pipeline stages. A user needs to program these pipeline stages to make it work.
10 years ago, there was a fight between pipelineed archtecture vs. RISC in NPU. The programming difficulty of the pipelined architecture makes X's product hard to understand and maintain.
When you discuss NPU, please think both ways HW and SW. Every NPU company can make HW, be it 10G/40G/100G, even though it is very challenging. The real issues come from SW. I.e. how to program this type of monster. That is the exact problem Intel faces right now. Who knows how to program multi-core?
就知道你完全是个外行的评论是否下得太早了？
61. 理客 于 2010-02-28 8:25 am edit
04年曾建议过NP+ASIC，可能当时技术发展还不够，当然，老大们估计也早就考虑到了，无需小人物的大建议。
目前多核江山也是一片火，把多核和NP组合成一个类似山寨的通用方案，也许能做点事，对于更有专业追求的公司，还可以在此基础上加上自己的专用ASIC，就更完美了。
当然具体到系统架构，涉及到成本、功耗和散热等诸多问题，就不是那么简单了
J的自研NPU指的是TRIO还是另有所指？
62. Multithreaded 于 2010-02-28 8:32 am edit
>我再补充您两句，不仅仅C和J，H和Z也在做自己的NPU，而且不久您可能就会听到更多的消息，但是这又能说明什么呢？能说明X和E没机会了吗？NPU做出来，一版成功的从未有过，万事需要时间的积累的。
So what? I knew the story of H brought their first generation NPU in 弯曲。 The real difficuly is from SW. In this case, H can not make the OPEN64 compiler work 
63. 陈怀临 于 2010-02-28 8:32 am edit
小强倒是没必要说MT是个外行。。。：-）。弯曲评论的水比较深，IEEE的张Fellow也在这里混呢。没准MT是个什么公司的大牌，或者某个学校的兽兽。。。：-）
64. 陈怀临 于 2010-02-28 8:49 am edit
从技术的角度，我相对同意MT的看法。我也大概知道你是谁。是，我回来后与你吃饭，见面。
是，硬件微结构一定，其实事情相当稳定下来了。软件的变化太大了。。。
据说Intel fail掉的那颗GPU，就是（埋怨）软件做不了优化：-）。

65. 老刘 于 2010-02-28 4:05 pm edit
MT口气有点大，说话有点绝对，这也难怪别人说你外行。
NP短期内不会消失嘚。不是每家OEM都能像Cisco,HW那样牛逼。请记住，华为海思过年那阵license了Xtensa处理器。接下来你们也知道干啥了。
回理客，多核和NP集成嘚SoC有了。LSI ACP34xx….
66. 陈怀临 于 2010-02-28 4:44 pm edit
小强估计是广州军区的，或者就是广州军区做大辽办事处的。。。
据我不愿透露来源的消息透露：–)
华为最新的NE，CX的线卡 ：：= Dune Fabric + X11 NP。
While,
ME 线卡 ：：= Dune Fabric + EZ
再晒晒？ 好，大元宵的，不晒有损人品：-）。
LPUF-20/21 card
=1×20G X11 + 1 FAP20v
LPUF-40 card
=2×20G X11 + 2FAP20v
67. 小强 于 2010-02-28 6:26 pm edit
to 首席：我真是志愿军，哈哈
您晒的还少了点，不仅仅如此，HW里面还有产品用X11而且是出货量最大的，您给漏掉了，应该再调查下：）
ME已经有意换掉EZ了
对于说MT是外行，我在这里道歉了，倒不是因为我觉得我的判断依据有错，而是觉得不该这么攻击他人，诚挚道歉
68. 小强 于 2010-02-28 6:39 pm edit
to MT: 我还是需要直接跟您说句对不起，我的攻击性言辞是不对的。
但是您摆了一堆英文上来，一我看不懂，二您觉得这是权威的吗？100多级编程，您知道是指什么吗？我相信您知道，否则也不会这么理直气壮哈。100多级编程其实就是100多条微码指令而已。只用100多条指令就可以完成最复杂的业务，请问编程还有比这容易的吗？所以您说因为100多级编程就导致了X很难理解和使用，这让我觉得您一开始是"外行"。
我来说下X对设计的时候要求最高的地方在哪里吧，是因为他的PB模式，因为每个PB有32条指令（HX是16条，就是为了解决这个问题的），32条指令后就可以进行查表（这是相当灵活的），但是有一个问题，当一开始设计的时候流程已经写死，加入写了30条指令，但时候来发现要在那次查表之前增加3条指令，30+3>32，这样就导致需要将流程重新排或者将流程顺移一个PB。这要求在一开始系统设计的时候要有一个NB的人来做总体把关。将以后最有可能出现需求增加或者目前需求不明确的地方，做多一点的指令预留，因为X11完成所有业务还是有资源预留的（HX和AX会有更多的资源富裕），这是X的 PB架构的唯一的需要多加注意的地方。而不是他的流水线的100多级流水。所以我说写这个文章的稍显"外行"
我举一个例子，以前有一个某国内超大型公司的一个产品从其他的NPU切换到X，开发工

程师一开始很抵触，害怕不会用，可是后来居然说，这是他们遇见的最好编程的NPU，一个人，3天就可以将IP最复杂流程编完并调试通过（当然他已经很深入的学习一段时间了）。所以后来给他提心需求的时候他都是欣然接受，而不是像平时大家想想的那种推脱，因为他说改起来太容易了（不过说句实话，我也觉得他有点自残，呵呵）所以别说100多级流水，听起来怪吓人的，您说100 多条指令就直观了
P.S.我再强调一下，因为我发现可能是我写的比较多，每次我写的信息都被跟贴的同学忽略了很多。如果前期设计不当，X的维护可能会在极端情况下，让人觉得不是那么爽。但是只要你前期设计做得好，万事OK。而且HX和AX已经改进了，每个PB有16条指令就很好的解决了指令碎片问题
69. 小强 于 2010-02-28 6:43 pm edit
to 理客：现在很多公司尤其是我们国内的那个大鱼两年前就已经用了X11+ASIC的做法了，难道这是您4年前的大建议起作用了？：）哦，这么说来的话首席晒的还是不全，还差了好几个呢
而且现在这条大鱼的其他产品的新班子也有X+ASIC的做法。当然不是X自己搞不定，加ASIC是因为要降成本。一定有同学问难道X+ASIC会比单独的X便宜吗？嘿嘿，还就是便宜，至于其中原委我就不道破天机了，想知道的人自然会通过他的渠道去了解的：）
70. 小强 于 2010-02-28 6:50 pm edit
再 to首席和MT：没错，很多NPU代码的优化是相当困难的，当年H在2800上优化代码做了大半年。这就是血的教训。而E公司的芯片由于性能的问题，几乎没有一个设备上用了之后不花大力气优化的，这也是他的弊端。而我似乎没有听说过X11需要怎么大规模优化的？X11一个流程下来20G线速的情况下，全流程最多640条指令（其实不是MT说的100多条，所以我才会觉得那个结论有点道听途说），640条指令能需要怎么优化呢？所以X更趋紧与ASIC，他只需要不难的几条微码就可以实现业务的灵活了。这也是我推崇他的地方。
没错ASIC不需要对转发编程，只需要驱动，但是我说了ASIC的弊端大家都知道，业务不能修改，灵活度低无法应对需求的爆发式增长，大公司一定要用NPU，至于是用自己的还是商用的，那就看谁NB了。自己的好当然用自己的，但是前提是自己的要好
所以从大家关心的软件角度出发的话，我又要再次推崇一下X了，呵呵
71. 小强 于 2010-02-28 6:52 pm edit
to MT，你的信息有点慢了，虽然我不知道您是谁，看首席都要和你吃饭，您应该是个人物，但是我说的不是他的第一代NPU。多说无益了，您再调查下。跟软件无关。
我还是那句话，NPU比ASIC是有优势的，短期内没有消失的理由，无外乎就是用谁的而已
72. 小强 于 2010-02-28 6:55 pm edit
to 老刘：您虽然短短几语，但是我fully agree
73. test 于 2010-02-28 7:46 pm edit
to 66楼 陈首席,
FAB20v 应该是FAP20v吧!
74. 理客 于 2010-02-28 8:00 pm edit
老刘：LSI ACP34xx….大体能力和前景如何？

小强：我不是这个专业，04年瞎扯的几次，应该没人理
NP的代码空间目前看看不到好的解决方法，唯一商用的方案就是加ASIC。
因为IP太灵活变化太多太快，这个对要求设计做许多更多巧妙的后向考虑的问题，不是一般的难解决
J号称的的programmable ASIC如何？
75. cs-cisco 于 2010-02-28 8:05 pm edit
说思科走下坡路的话未免过于唐突了吧，我相信鲑鱼会跳过瀑布去产卵的。呵呵。
76. 小强 于 2010-02-28 11:06 pm edit
to 理客：现在新一代的NPU在指令空间上已经有很大飞跃了，如X的下一代的最长指令空间增长了近两倍。所以指令空间不是问题。当然是否够用还要看路由器把最复杂的业务放进来的评估结果，目前做过评估的都没有问题
可编程ASIC只是一个噱头，既然是ASIC他能做的就是尽量增加点灵活性而已。我还是看好NPU在灵活性上的优势，只要NPU能够很好的解决他的劣势。
77. 理客 于 2010-02-28 11:17 pm edit
小强：如果没有ASIC完成基本的业务，全用NP完成，对于一个全业务的智能路由器而非功能相对单一的L3设备，恐怕再来两倍的空间也未必够，两种方案：
1、把ASIC的功能尽可能多的集成到NP中去，简单的调用而不必用很多微码，可能目前已经有在这样做了，这个不知道在技术上有何限制，肯定有问题，否则早就发展得很快了
2、license控制，这个也会很烦
J的programmable有噱头的成分，但J的做IP的ASIC能力，在业界可以算第一吗？
78. Multithreaded 于 2010-03-01 12:00 am edit
>MT口气有点大，说话有点绝对，这也难怪别人说你外行。
NP短期内不会消失嘚。不是每家OEM都能像Cisco,HW那样牛逼。请记住，华为海思过年那阵license了Xtensa处理器。接下来你们也知道干啥了。
这也正是独立的NPU公司做不大的原因。 每家都有自己的NPU，有谁会真心地用别人的NPU哪？
我曾经为NPU事业贡献过"青春"，真不希望大家被MARKETING的人给吆呼了。用NPU做那些L2/L3的东西没有什么高深的，都十几年过去了，应该不难掌握。
79. Multithreaded 于 2010-03-01 12:14 am edit
> 我来说下X对设计的时候要求最高的地方在哪里吧，是因为他的PB模式，因为每个PB有32条指令（HX是16条，就是为了解决这个问题的），32条指令后就可以进行查表（这是相当灵活的），但是有一个问题，当一开始设计的时候流程已经写死，加入写了30条指令，但时候来发现要在那次查表之前增加3条指令，30+3>32，这样就导致需要将流程重新排或者将流程顺移一个PB。这要求在一开始系统设计的时候要有一个NB的人来做总体把关。将以后最有可能出现需求增加或者目前需求不明确的地方，做多一点的指令预留，因为X11完成所有业务还是有资源预留的（HX和AX会有更多的资源富裕），这是X的 PB架构的唯一的需要多加注意的地方。而不是他的流水线的100多级流水。所以我说写这个文章的稍显"外行"
这个正是我说的编程难点。 如果那位大牛离开了公司，代码如何维护？一级放不下，岂不是要长江后浪推前浪的来个大改动？

请回顾一下，软件维护所占的工作量要远远大于开发的工作量。每级预留提前量的做法是不得已而以， 不是一个软件开发的好方法。
我说100多级就是这个PB. 也许最新的X产品有改进。但本质上这种流水线的编程方式不是一个好地编程模型。
80. 小强 于 2010-03-01 12:29 am edit
to 理客：这里面有一个问题您可能忽略了。那就是全业务职能路由器是否需要所有的业务都线速。
分别讨论下：（我这里就以X的HX举例了，当然也希望比我更熟悉EZ的兄弟们站出来用EZ举例哈，很想多学习下，期待大侠指教）
1、目前我看到的一些接入层面的路由器，他们把全业务摆进HX是基本没有问题。当然X11是放不下的，这也就是首席在66楼晒的几个产品之一了。
2、如果您说的是更高端的话，那么要考虑是否要求所有也无必须权线速
2.1、需要全线速，那么ASIC+HX或者2×HX是很好的选择
2.2、不需要全线速，如最复杂业务可以不线速，那么1×HX就足够了，因为我前面的帖子说过了HX可以支持部分业务转圈（也就是可以用更多的转发资源，多转一圈就会多用一倍的资源，如指令空间，各种引擎等等），不仅仅支持转圈，还支持线速业务和不线速业务之间的调度，非常灵活，所以1×HX就足够了
其实前面说的都是基于1片搞定所有业务线速的前提的，其实当需求拿来的时候，任何设计都可以灵活考虑，咱们是高新技术行业吗，呵呵，一成不变哪受得了，否则不就是MT所说的没什么高深了吗，哈哈
其实现在有很多设备商在对运营商宣称不需要所有业务都线速的理念了，在这种情况下，业务的灵活度是最重要的，NPU的灵活优势一览无余
总而言之，具体问题具体分析
81. Multithreaded 于 2010-03-01 12:33 am edit
>> 再to首席和MT：没错，很多NPU代码的优化是相当困难的，当年H在2800上优化代码做了大半年。这就是血的教训。而E公司的芯片由于性能的问题，几乎没有一个设备上用了之后不花大力气优化的，这也是他的弊端。而我似乎没有听说过X11需要怎么大规模优化的？X11一个流程下来20G线速的情况下，全流程最多640条指令（其实不是MT说的100多条，所以我才会觉得那个结论有点道听途说），640条指令能需要怎么优化呢？所以X更趋紧与ASIC，他只需要不难的几条微码就可以实现业务的灵活了。这也是我推崇他的地方。
H用了IXP2800里面有8000条指令还叫不够用。640条能做啥？这十几倍的差距令人琢磨？
1). How to do IPv4 forwrding in general in X?
2). 是否难得操作都用TCAM？
3). Is X instruction a VLIW type?
In general NPU optimization is hard. pipelined based optimizations are even harder. That is the reason that avoiding pipeline programming as much as possible.
In a word, there is always a way to manually tune NPU to its peak

performance. However one has to consider how to maintain the peak performance when anew change request comes in the future.
82. 小强 于 2010-03-01 12:38 am edit
to MT：您作为一个为NPU事业奉献过青春的人，现在应该是个牛人了，这点我深信不疑。那您一定不难了解到现在都谁用X11了吧，您能顺便了解下他们谁在大改或者整天优化代码吗？因为我一个人说很难有说服力，您调查下咱们再讨论：）。
对于您说的牛人离职，这是不怕，据我所知有一个大用大卖X11单板的产品，当年负责架构设计，芯片选型，产品前期开发的牛人经过几年的洗礼，全都各奔前程了，但是他们依然是业界最牛的XXX产品。从未大改过。其实流水线架构，只要初期留的好，后面基本不用改，所以不怕大牛离职，只要大牛开始在设计就OK了
您说的这个流水线的编程方法，从X11出现的那天起，就不停的被人challenge过，天天的问，总是怕这那，可是怕的人最后都没用，用了的人都说好。这也就是说怕的人丧失了一个机会（有点绝对哈，相对的），而选了更头疼的东西。我相信MT应该是属于没有用过X的大牛吧。如果您用过的话，我想您不会这么说了，我不才，用过一些。
其实所谓的编程困难都是理论上的一种畏惧感，设计起来其实没有。如何避免和解决这个问题，我前面已经说过解决方法了，MT大牛是不是觉得我的那个方法不好？或者您觉得那个方法很难理解和被使用呢？我遇见过很多人，都为产品前期的设计如痴如醉，那是最锻炼一个人的时候，其实没有那么可怕了。用一下，给别人一个机会的同时也给了自己一个机会：），请不要停留在理论阶段，理论是把双刃剑啊
83. 小强 于 2010-03-01 12:47 am edit
to MT 81楼：看来您真不是内行，我有点小崩溃，难道所有的东西我都要解释清清楚楚吗？您也觉得640条指令很奇怪是吗？我相信所有人都会很奇怪，可是我咋就没想到您会认为这是总指令空间呢？
我虽然没有明说，可是行家一看应该知道我在说什么吧？可是我说了全流程或者最复杂业务等等了啊，我还以为您真了解流水线机制呢。我说的640条是单流程最大指令，不是整个的指令空间…..
还有就是，X的指令有merge功能，我不需要再培训下了吧？就是可以一条指令merge四个不同的操作，真正的操作数平局下来是2倍多于640的。前提这是20G线速，如果10G线速的话，指令空间更多。
X的指令是VLIM
对于您说的NPU总是宣称最高性能的问题的话，我觉得很多NPU是这样的，但是出X外，X在这点是他独步江湖的地方。当然别太离谱的需求，太离谱可以有N种方式，我在给理客的回帖中写了，您可以帮我review一下，欢迎讨论
P.S. 能不用英文吗？看起来费劲啊
84. 小强 于 2010-03-01 12:53 am edit
再to81楼MT：一个general IPv4如何放到X11里面，这个问题，您调查下行吗？我相信您的实力。这都多少人做过的了，咋还怀疑呢？这不是新片子啊。
HX的能力较X11有了大幅提升，并且功能有了突飞猛进的提升，我想您调查过X11后，就不用再问HX了吧
复杂业务都放在TCAM里？您的这个构想我真是从来没敢想过啊，TCAM是要连在

NSE接口上的（目前ASIC和NPU的主流连TCAM的接口），NSE 也是有频率的，有performance的，保证线速情况下，一个报文能访问几次？你可以计算下，TCAM分288bit和320bit两种，320bit的效率更高。20G线速下，现在返回值为160bit的，最多就能返回8次，320的最多返回4次。复杂的都放TCAM里这个构想太 crazy了，这样的系统能线速我就服了，我的确没想过....
85. 小强 于 2010-03-01 1:02 am edit
再toMT：不好意思，光顾回您的技术贴了，一时忽略了您的中心思想，您到底想说啥呢？是NPU赶紧滚出历史舞台还是您觉得我对X和E的比较有觉得不对的地方呢？如果是NPU要消失，那您觉得什么东西来替代呢？短期内的哈，我相信无穷远的将来一切皆有可能
86. 小强 于 2010-03-01 1:16 am edit
to MT, 想了想，我可能言辞有点激烈，再道个歉。我说话也有点决，哪有什么产品会不优化呢，不管用啥都会优化下的。只是时间长短吧。
由于HX支持了区分业务转圈，以后就没有您说的那个问题了，对于X11的话，我相信有人优化过，但是没有您说的那么严重吧。相对于其他的芯片来讲，无论是多核还是以前的2800，rainer，2400等都应该好很多。
87. Multithreaded 于 2010-03-01 1:22 am edit
小强同学，能不能不扣帽子？你有喜欢X的癖好，我可以有不喜欢X的自由。
88. 小强 于 2010-03-01 1:44 am edit
to MT,其实我不是有喜欢X的癖好，我是想通过这样的讨论多学习东西啊，我又不是来给谁做广告的。我就是想听听各位大牛说说他们的想法，以后我去跟别人聊天的时候也可以把学来的东西举为己用，装把“博学”啊
我多少有点被虐倾向，就是喜欢有牛人给我指点迷津，让我多学点，说不定哪天有人说X好的时候，我说不定又站在对立面了呢。只要大家说的有道理就行啊：）
何必为了喜欢和不喜欢做那种无畏的争论呢
89. Multithreaded 于 2010-03-01 1:55 am edit
抛块砖，引玉把。
1。 NPU市场长不大。两年前大概是$100M多一点. Intel占了半壁江山，谢谢H&Z:-)
2. Intel已经退出了这市场，这块小蛋糕够不够Cavium，RMI，LSI, X, E等分哪？有人会活下来，但我不推荐你买他们的股票。
3. 10G以下的NPU设计没有什么难点，中国的H和Z应该有能力搞定。难点是HW and SW co-design, especially HW design impacts on SW programming and performance.
4。40G/100G的NPU想收到钱还远着那。
5。Intel/AMD/IBM的多核会把NPU的设计吸进来。不过小强同学可能等不及了。但是要想到你今天写的程序明天如何移植到多核来。
90. 小强 于 2010-03-01 2:13 am edit
to MT: 这个帖子看起来还是有一定功力的：）
目前业界有能力做出40G/100G NPU的（做得好的），目前只有X和E吧，所以如果以后还是$100M的话，那么我强烈建议买这两个公司的股票，因为这两个公司真的很小，

$100M每年的话够把他们养的肥头大耳了。当然现在是多大的利润了我也不是很清楚了，可能没有这么大了，需要调查下。
对于20G的NPU的话，现在市场足以让X和E活到他们的40G和100G赢得利润的时候。
您说的远着呢有多远呢？我相信最多两年之内一定会有收益的。
10G以下的NPU目前我觉得H和Z不一定搞吧？他们的胃口应该不仅与此，所以对于X和E的未来的确有点担心，那就是当H和Z有了高性能NPU怎么办？但是那需要相当的时间，至少Z可能需要很久
您说到的这些，其实NPU的厂商早就看到了，他们已经在酝酿着更多的东西了。
NPU长不大是指多大呢？其实长多大不重要，重要的时候能不能养活做它的人。我觉得NPU有着他长期存在的理由。
隐约感觉你很推崇多核啊（可能感觉错了，不过觉得您好像和intel的渊源很深啊…），多核提出有一段日子了吧，怪不得您不喜欢流水线架构，这本来就是两个相对对立的概念，我倒没有不喜欢多核，我喜欢强者，但是目前我更喜欢流水线：）偶已经不编代码很多年了，如何移植到多核不是我考虑的了，但是移植那一天的到来说不定要很久哦
不过说句实话，您的这个命题对我来讲有点不那么手拿把掐了，反正我是不知道哪天NPU会死，也许明天，也许很久都不会
91. Multithreaded 于 2010-03-01 2:23 am edit
我想问的是
1。 5-tuble的classification是否放到TCAM里做了？
2。 longest prefix matching要花几个PB要多少指令？
一般来说IXP2800的一个engine可以有8000条指令，它对应的是X的一条Pipeline，完成所说的复杂业务. 640即使成以4已比8000少很多。
大概X把有些"难"的操作交给co-processor去做了。
92. Panabit 于 2010-03-01 2:28 am edit
据我所知，IBM两年前就在研发多核网络处理器（现在应该出来了吧），吸收了很多NPU方面的内容进去。NPU应该还会存在一段时间，不会轻易退出。
93. haha 于 2010-03-01 4:47 am edit
这里真热闹，x和e是2只癞蛤蟆，x后腿长一点，e是前腿长一点。x说我的后腿长，最漂亮；e说我的前腿长，也很英俊。他们都没有看见过青蛙。
94. haha 于 2010-03-01 4:56 am edit
本来以为能学习到东西的，就看见口水仗
95. 陈怀临 于 2010-03-01 6:40 am edit
不会吧？我还想把这篇文章的评论单独整理成一篇文章呢。例如：关于NP的讨论–X和E的聊天。
96. 小强 于 2010-03-01 7:19 am edit
to all,既然牛人们都纷纷站出来总结发言了，咱也就不多说啥了，只是偶也想学习，至少我还从大家的讨论中多少学到了些东西，只是haha同学的总结实在是太好了，好的让我这种没学问的人看了有点无地自容了，那我们就静待有学问的人出来写点东西让我们这种无知的人多学习点东西了，也很期待haha多教大家点哈。
只是到时候haha别也就是吐口水的本事就好。

101. Multithreaded 于 2010-03-01 12:25 pm edit
聊点专业的把。
据说H&Z在攻40G的NPU, 我个人的感觉是
1。 先把10G拿下，锻炼锻炼队伍。 然后上可供40G，下可把X&E etc. 在接入层扫地出门。
2。 40G/100G的东西，用一下X&E的产品，搞个Demo.什么的能向科技部交差就行。现在企业的数据中心和TOP 500 Super Computers主要的还是10G的天下，还要有3-5年的时间才会大量用到40G/100G.
3。 为什么不做10G的NIC? $1,000多美金一片，比10G的NPU还贵,量也大得多。 要学C进入Server市场，NIC这一关是否得过?
102. Multithreaded 于 2010-03-01 1:28 pm edit
〉〉隐约感觉你很推崇多核啊（可能感觉错了，不过觉得您好像和intel的渊源很深啊…），多核提出有一段日子了吧，怪不得您不喜欢流水线架构，这本来就是两个相对对立的概念，我倒没有不喜欢多核，我喜欢强者，但是目前我更喜欢流水线：)
其实我非常喜欢流水线架构， 现在考虑的问题就是多核架构下的流水线并行模型。比如10G的TCP/HTTP协议如何用流水线实现？ 请注意我的一个pipeline stage可不是32条微码。
我不欣赏的是把硬件流水线的微码编程方式强加给软件人员。 Verilog/VHDL适用的流水线编程模型可以在L2/L3上用用，但不应该当成是唯一的或者最佳的选择。它对软件维护的杀伤力是蛮大的。想想看，C 的QuantumFlow NPU为什么不用类X的流水线架构？
其实我们俩的分岐是在对流水线的grainularity的掌握上。让我们求同存异，今后不要乱扣外行的帽子就行了。 因为这世界上本身就没有外行，每个人可能只比其他人早接触了一点而已。
103. kkk 于 2010-03-01 8:42 pm edit
说到counter，不知道关注的人多不，的确遇到过某芯片的counter不太精确，千兆的环境，很快的速率读寄存器的counter然后累加的值就不对（寄存器是读清的），1w个包就会老差一两个，但是一次读就没问题，呵呵
104. kkk 于 2010-03-01 8:43 pm edit
最后他们的工程师确认是芯片的bug
105. haha 于 2010-03-02 8:14 pm edit
to kkk, 某芯片是x?我从Hw了解到曾经发生过这样的事情。
to 小强，我费尽9牛之力，想从网上搞到一点点x或e的资料，实在很少，都不够唾沫星子。
106. 楼上楼下，电灯电话 于 2010-03-02 8:33 pm edit
很有意思的讨论，存在就是合理的，无论X或E。这年代剩者为王，看谁能笑道最后。
107. 数通人 于 2010-03-03 2:26 am edit
要灵活就CPU，要快速就ASIC，这是亘古不变的原则。各家的NPU不过是走在CPU和ASIC之间的灰色道路上，无非有的偏硬，有的偏软而已。都有自己生存的空间，无非有的大点，有的小点而已。今天在finance.google翻了翻通讯芯片公司和国内网

游公司的业绩，老大BCM有44亿美刀的营收，却只有区区6千多万的净利。完美世界4.5亿的营收，净利则高达2亿3千万美刀。所以能理解钱老大为啥要改logo，为啥要收flipvideo了，还是老百姓的钱好骗啊。
108. 理客 于 2010-03-04 6:22 am edit
对于NPU，不要说全业务，就是基本的IP/MPLS L2/3业务，二层交换，QinQ,MPLS TE, VLL/PW/VPLS/VRF，组播，TRIPLE PLAY，QOS， ACL等等，还不算BRAS/IPV6/NAT等，要是没有ASIC做上面的基本业务，两个X11都很难应付，还不要说X11致命的一旦不能线速，就会性能大幅下跌到基本不能商用，而不是像传统模式的CPU/多核/NP一样，性能下降是一个比较平缓的曲线。要说用X11作为主转发引擎的协处理器，那是非常好的，但要是颠倒过来作为主力，那有点主次倒置，颠倒了。目前的商用系统基本主要是NP+ASIC模式，而不是纯NP的模式，已经说明的了纯NP在功能+ 性能的满足度上是有不能解决的问题的，哪个大供应商没有一些大牛，所以他们很清楚这些关键问题，不是很容易被别人忽悠
当然，实际的系统架构的设计还要考虑成本、X小公司的稳定性和前向兼容等问题，就更复杂了，所以目前的高端系统，门槛是非常高，整个系统包括CPU、多核、TCAM、查表器、NPU、ASIC、FPGA等多个核心硬件部件，把他们统一运行成一个商用系统，目前只有有数的几家大供应商可以负担得起，这不是一个纯技术问题
109. Multithreaded 于 2010-03-04 7:37 am edit
同意#108地分析。
NPU在10年之内没有成长当初预估10亿美金的一个市场，是由它深刻原因的。估计永远长不大了。
但1G以下的市场不用NPU, 有点说不过去。 在接入层，假定功耗和价格不是问题（将来不会是问题），为什么一片NPU或多核搞不定你说的业务哪？
110. mpc8240 于 2010-03-05 2:12 pm edit
国内的筒子推崇NPU的原因其实很简单，没机会体验过C/J/A的ASIC。ASIC做好了programability也是很强大滴。H牛气是牛气，动辄换NPU chip vendor，没有多年的继承，和C/J/A竞争也就只有靠性价比了。
111. 陈怀临 于 2010-03-05 3:58 pm edit
同学，这里H,Z等等的弟兄们很多滴。。。你被他们的口水淹死，可别怪首席在旁边乐。我这几天在幽州太忙。明天回去。我把这次NPU，ASIC等的大讨论整理成文。。。
我会从一个设计过一个网络处理器编译器的工程师的角度来讲一哈。
有设计和整个走过一趟编译器backend的人吗？：–）。咔咔咔咔，大家都安静了，估计。。。
112. Multithreaded 于 2010-03-05 5:58 pm edit
听说过Intel IXP -A 和 -C 两个C自动编译器吗 ：-）
113. Multithreaded 于 2010-03-05 6:05 pm edit
#100， 国内的筒子推崇NPU的原因其实很简单，没机会体验过C/J/A的ASIC。ASIC做好了programability也是很强大滴。
第一句好像有点道理，第二句不太准确。
ASIC can be configurable but not programmable.

114. 楼上楼下，电灯电话 于 2010-03-05 6:15 pm edit
#108 很精辟
115. 老刘 于 2010-03-05 6:27 pm edit
如果是GCC的backend，简单的说。没有你想象的那样复杂。
116. 老刘 于 2010-03-05 6:36 pm edit
什么ASIC不ASIC，思科的QFP ASIC实际上不就是一个NP。
117. 陈怀临 于 2010-03-05 6:47 pm edit
小刘，你还真牛了。你要是加过一个target，我还真不信了。什么叫做GCC的backend。我的意思是加一个ISA的target。你在想什么呢？如果你不是计算所的，或者原来Intel的那票人（MT你先不要乱讲：-）），我还真不信你懂我在说什么。。。你不要来个什么简单的说。你可以复杂的说一哈。。。
没想到，还有人觉得自己编译牛的。真是来劲。
118. 老刘 于 2010-03-05 7:01 pm edit
小弟不才，我也不牛。我就是说没有你说得那样的玄。
119. 陈怀临 于 2010-03-05 7:59 pm edit
你非要拿什么图灵奖的理论工作来比较，这就没法说了。。。
编译器难在对一个target的微结构的利用和优化上。例如，对性能的损失的控制。另外，对网络处理器的编译器，对特殊指令的设计和处理是最难的。。。。。。MT是个牛牛。我们让他多说说。。。
120. 老刘 于 2010-03-05 8:09 pm edit
都这么说了，我也就不是说了。
是牛人的站出来。你们讲，我来听。
121. 杰克 于 2010-03-05 10:38 pm edit
面过intel 做ixp编译器的人，后端的优化做了N多遍，经历最重要，讲里面有多高多难的技术，谈不上。在里面钻的多了，体会自然就来了。哪一行都是这样。
122. 理客 于 2010-03-05 11:07 pm edit
Multithreaded：我很同意你的idea，严格说，我属于外行，一块NPU的代码限制，我拿不出强强喜欢的具体数据，但是从到目前为止的实践上确实是搞不定，因为代码空间就是XXXK，除非通过license控制，但这种为了解决代码空间而做的license，在实际使用过程中涉及到如何做group features 、测试工作量增加、maintenance…等很多问题，说起来简单，做起来是比较trouble的。至于多核，是不存在代码空间问题的，除非把需要处理的业务代码全放到cache中，那就有和NP类似的问题，目前，好像还没有这样用多核的，有这样用过的大牛可以站出来show一下，IP从初始的单纯的 IPV4转发到现在的支持所有电信和非电信业务，即使是基本功能，也扩展了N倍，用C码实现所有这些功能，编译一下，应该也上M了吧，微码再优化，也还是需要很大的代码空间。总结来说，原因就是：
1、NP代码空间受限，CPU/多核不受限
2、目前路由器的主要业务对代码空间的需求已经很大
一些可以考虑的方案是把ASIC完成的IP的已经成熟固化的主要功能集成到NP中，作为一个特殊的指令调用，当然这时候成了一个CHIP里，谁主谁付，外外面看来就可以不那

么强烈的去争论了，不知道是否有这样的feasibility或者有人在实践，目前的商用都是NP+ASIC的分立模式，并且分立和融合，在scalability、consumption、flexibility、feasibility等方面比较起来也是各有利弊的，不好一概而论。
如果不不从技术细节，从市场效果来看，好像只有J在宣传或者实践programmable ASIC，J的产品在技术上不得不说应该是不错了，那么可以从侧面证明其programmable ASIC应该是做的不错的，当然，个人认为比其他的C/H/A等都强很多，毕竟J在这方面浸淫和投入了N年，这是他们的know how。当然我还是没有强强同学喜欢的数据，希望有大牛出来扔东西。个人粗浅的理解，J是把IP处理分解成合适的原子操作，这些操作又可以根据一定的规则做组合，但能否做具体变量的加减和条件处理，不知道J怎么做的，也许只是advertisement， not real。有一两个同学说在这里只看到争论，没看到有意义的技术（大意），这是超级的不符合事实，这个帖子，是除了很久以前关于多核CPU帖子外，少数几个涉及到很强的关于NP的前沿的技术实践的帖子，强强贡献最大，按这几天不露面，让喜欢强强的FANS，比如首席，有点寂寞:)
123. 小强 于 2010-03-06 7:21 am edit
好久没上来了：)
看到牛人们的讨论，觉得学到了好多啊
说下X：)
X11已经是上一代的事情了，X11针对的市场主要是城域以太网，包括PTN和以太网交换机。在实现这方面的功能时候一片X11足以线速，有市场成功案例做支撑，我想不用多说什么了
针对路由器市场，X11的确无法一个人完成全业务，要么两片，要么X11+ASIC，后者在市场已经有成功案例了
X的下一代产品近期就会面世。现在很多产品都在评估这款芯片，由于还没有拿到Demo所以现在都是纯理论分析，对于城域以太网，PTN，企业网，OTN以及接入市场来讲，一片达到全业务线速不在话下。现在的重点在路由器，路由器产品目前我知道的某知名大企业的主打产品评估下来是一片HX可以完成他们的全业务。路由器的确有非常变态的业务，在这种非常变态的业务出现的时候运营商不要求线速。而我在前面的很多个回复中都谈到了HX新增加了一个功能就是支持部分业务不线速，而不是X11所有业务要么权线速，要么全部线速的机制。这个机制的引入已经完全可以满足路由器的需求了。
当然如果要所有变态业务都线速的话，需要2片HX。
至于何为趋势，我的确知识浅薄，虚心向各位学习请教，不过据我所知C也在选择商用NPU芯片，而J也是在做自己的NPU，并不像大家认为的放弃NPU只做ASIC方案哦
124. 理客 于 2010-03-06 7:46 am edit
个人粗浅的胡言预测:HX再HHX也还是要+ASIC，除非它把之前ASIC完成的IP/MPLS基本业务seamless集成到其NPU中。NPU的重要性毋庸置疑并且无可替代，但NPU的在代码空间+性能的双赢是做不到的也是天然缺陷。在运营商L2/2.5/L3/MPLS/TE/VLL/PW/VRF /TDM/ATM/FRR/BFD/OAM…等等，你要敢说是变态业务，operator得把你P了，实事求是的说，运营商确实有变态的功能，但大部分是没有变态的，像上面的业务，都已经基本成了基本需求，这些业务的实现，我不看好HX在没有ASIC的情况下能摆平，即使摆平了，我们系统设计难道不留出 20%的余地？如果你不

做一定的预留，operator还是要P你:)都知道银行里银子多，可不好拿呀:)所以胡说的预测是：评估后的结果还是 NP+ASIC，希望到时候能有结果出来show一下看看到底鹿死谁手，花落谁家？
125. 理客 于 2010-03-06 8:13 am edit
看到强强的评估结果，如果是40G的降速到20G甚至10G来支持运营商要求的业务，这是可能的。但是这又和目前运营商已经要求100G/slot不符，所以还是不能单HX甚至双HX用在主流的线卡上，所以认为商用的主流产品还是HX+ASIC，但不排除部分低端产品，比如不要求100G/SLOT的产品用纯HX的方案
126. Multithreaded 于 2010-03-06 9:17 am edit
#118说的可能是GCC的移植工作，这玩意是难者不会，会者不难。难在搞稳定了。
#119说的是另一个高层次问题，修改GCC后端的关键算法，提高代码的生成质量。比如： Instruction scheduling. Let’s assume a non-blocking thread execution engine, given two memory reads, one is read from SRAM and the other is from DRAM, which one shuld be issued first? If two reads are both SRAM reads, which one should be issued first?
这里的难点是：
1。诸如此类的优化一般来说是NP-hard问题，解决好一个就可以PHD毕业了:-)
Frances E. Allen由于她对Fortran Compiler的贡献，拿了2006年的图林奖。
2。你写的优化是GCC上百种优化的一个，如何做到对其它优化没有破坏性的影响。比如，优化完了， 如何让.debug section 正确， make GDB work.
最难的是当硬件的人要加一条新的指令时，你得考虑到对程序员(language extension)，编译器(GCC tool chain)，和系统性能（performance evaluation)的影响，决定加还是不加？
Of course, no company could have done the above correctly yet. These are simply personal experiences, please take it as a grain of salts.
127. Multithreaded 于 2010-03-06 10:41 am edit
杰克说的对，实践出真知。 任何一个NP-hard问题，只要找到了启发式算法，也没什么可怕的。多做几个，也就家常便饭了。
不过第一个吃螃蟹的人，是要有勇气和智力的。
我就挺佩服#120的勇气的，敢于在COMPILER上挑战陈首席。
不过COMPILER何NETWORKING的研究方法好像不特一样。 Compiler optimizations more focus on algorithm design and network design more on entire system performance. 编译的人喜欢精雕细琢，网络的人要有大局观，喜欢一览众山小的感觉。
128. singlezhao 于 2010-03-06 10:24 pm edit
40G据说好象已经拿下了吧
129. 跳跳虎 于 2010-03-07 7:40 am edit
ASIC比较容易做到线速；NP可以比较灵活，根据不同的应用，互相配合应该是一个趋势。
做到高速焦点就是存储带宽的问题。TCAM，RLD等都太贵了，ddr首选。低速FC肯定首

选DDR。
看到各位牛牛的发言，受益匪浅。期待陈首席的整理。

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

Contributors List

贡献者名单以字母排序，希望更多的人加入。

Wang, Qi	V0.01 初始稿
Yang, Xi	V0.01 初始稿
Zhu, Yuhao	V0.01 初始稿

Contents

版本说明.....1
Contributors List.....2
序.....1
第 1 章 有关 Cache 的思考.....3
1.1 Cache 不可不察也.....4
1.2 伟大的变革.....6
1.3 让指令飞.....8
1.4 Crime and Punishment.....13
第 2 章 Cache 的基础知识.....17
2.1 Cache 的工作原理.....17
2.2 Cache 的组成结构.....19
2.3 Why Index-Aware.....24
2.4 Cache Block 的替换算法.....28
2.5 指令 Cache.....36
2.6 Cache Never Block.....40
第 3 章 Coherency and Consistency.....46
3.1 Cache Coherency.....48
3.2 Memory Consistency 1.....53
3.3 Memory Consistency 2.....53
3.4 Memory Consistency 3.....53
3.5 Memory Consistency 4.....53
第 4 章 Cache 的层次结构.....54
4.1 Cache 层次结构的引入.....54
4.2 存储器读写指令的发射与执行.....58
4.3 Cache Controller 的基本组成部件.....64
4.4 To be inclusive or not to be.....67
4.5 Beyond MOESIF.....73
4.6 Cache Write Policy.....82
4.7 Case Study on Sandy Bridge Cache Load.....87
第 5 章 Data Prefetch.....91
5.1 数据预读.....91
5.2 软件预读.....93
5.3 硬件预读.....95
5.4 Stream Buffer.....99
结束语.....102

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 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。

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

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]。

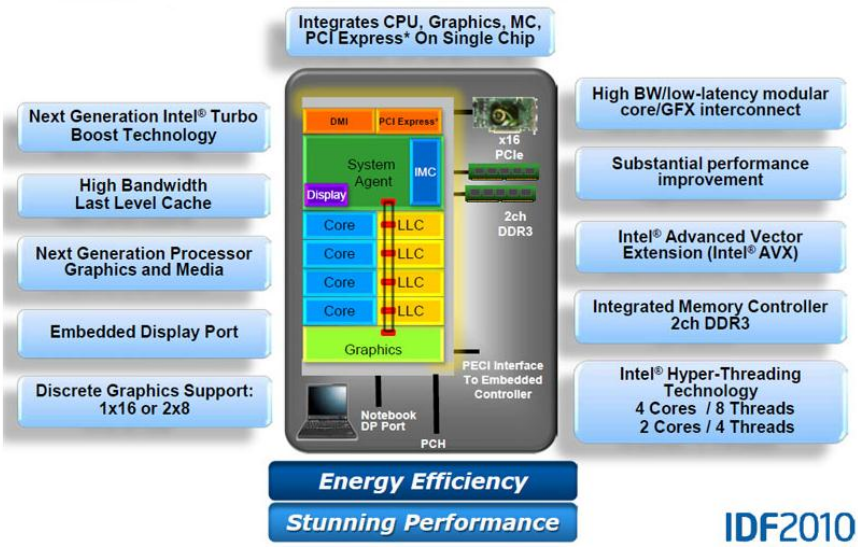


图 1-1 Sandy Bridge 的拓扑结构[1]

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示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。

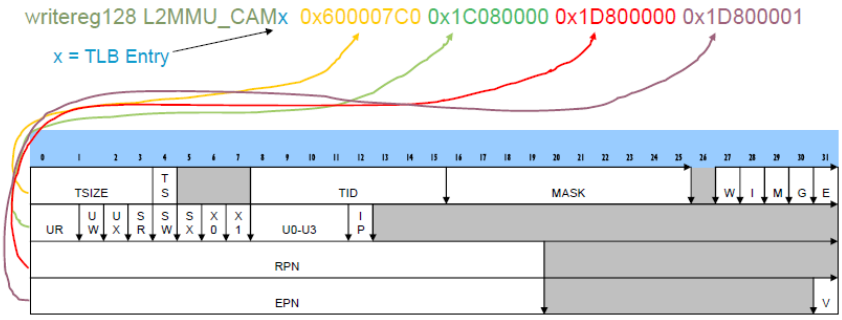


图 1-2 E500 TLB Entry 组成结构[3]

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，这个开销是难以接收的。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 仅为 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。

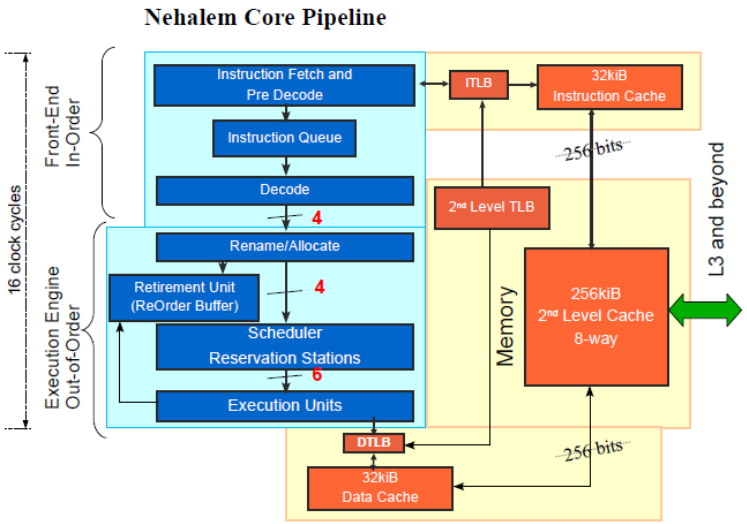


图 1-3 Nehalem 微架构 Pipeline 的组成结构[12]

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。

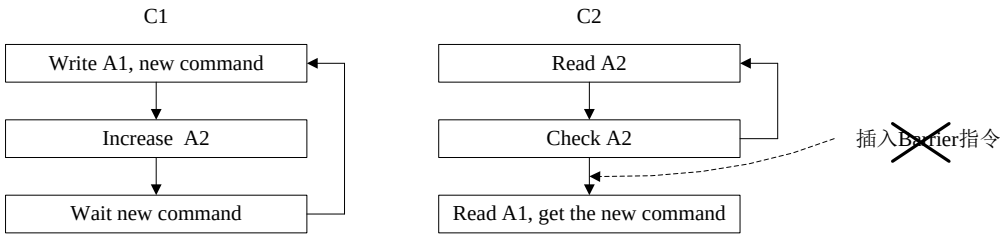


图 1-4 C1 与 C2 的数据交换

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。这切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 $O(N^2)$的复杂读，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。

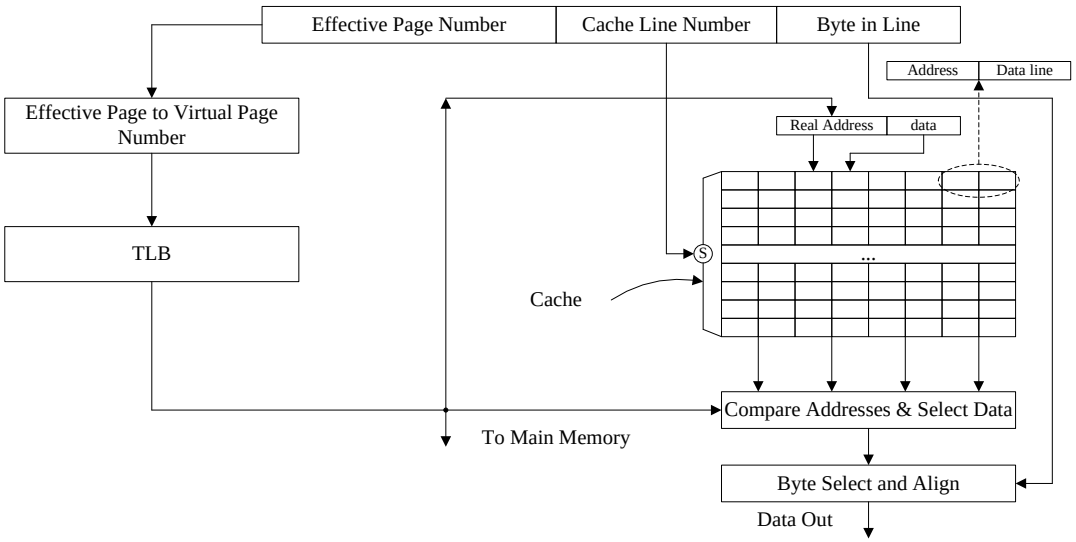


图 2-1 典型的 Cache 结构[①]

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。

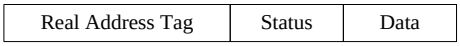


图 2-2 单个 Cache Block 的组成结构

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。

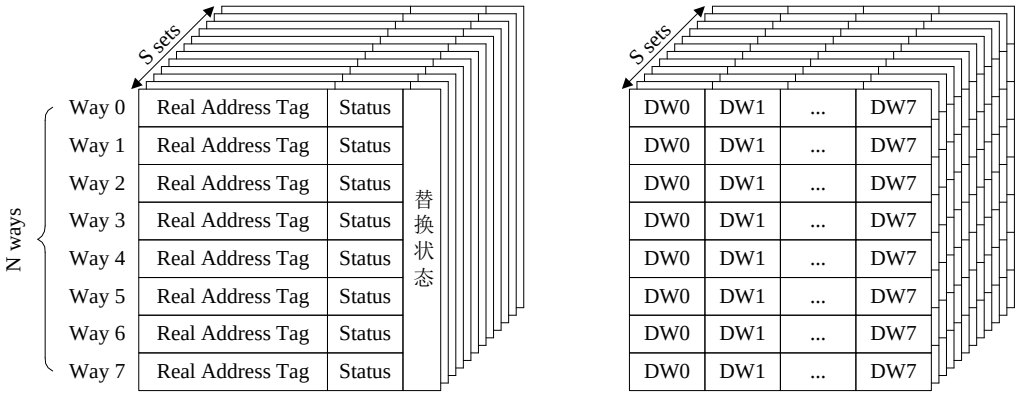


图 2-4 8-Way Set-Associative Cache 的组成结构

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。

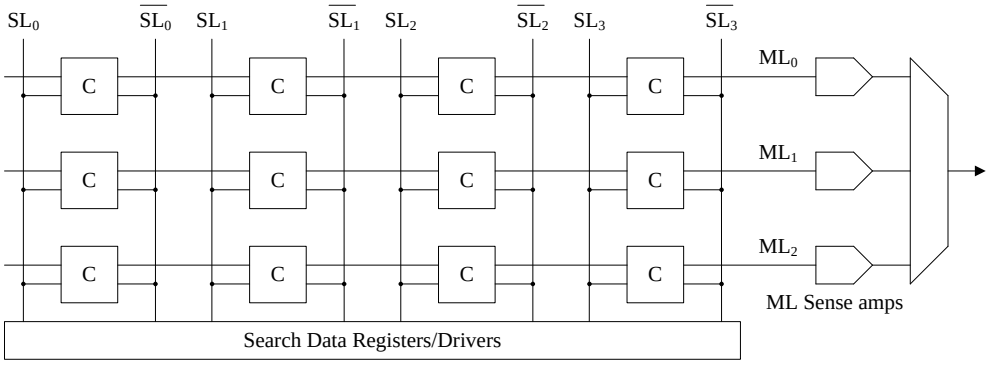


图 2-5 CAM 的基本组成结构[25]

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，之后这歌 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。

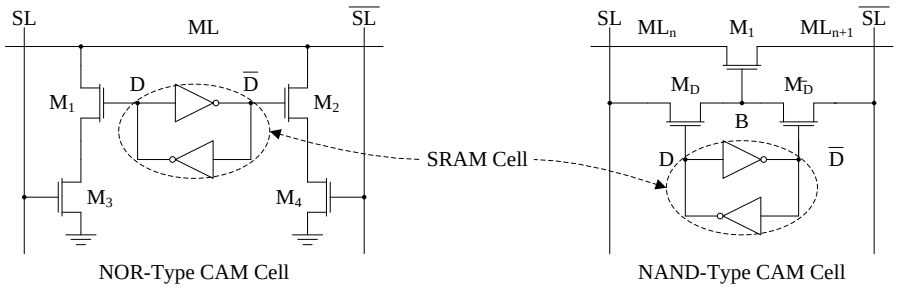


图 2-6 NOR 和 NAND CAM Cell[25]

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。

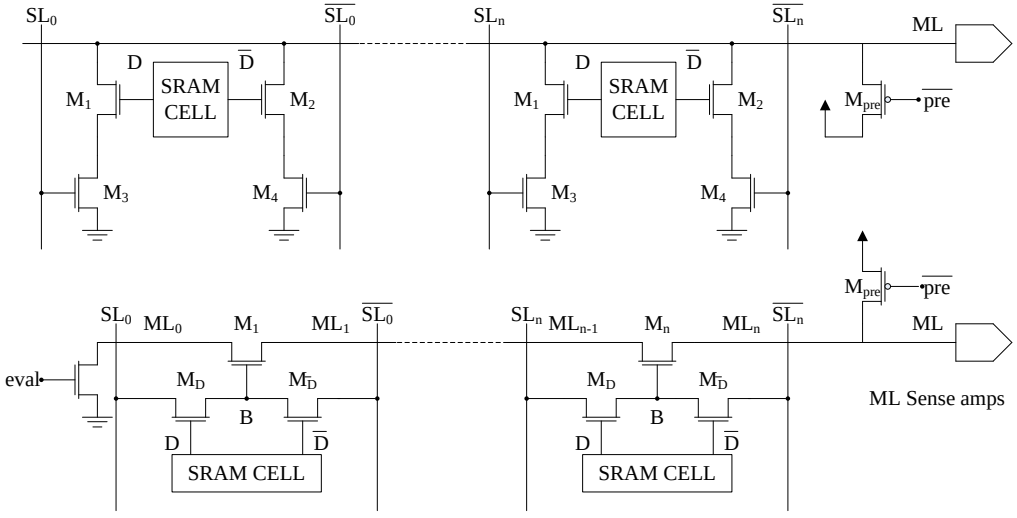


图 2-7 NOR/NAND CAM Word 的查找策略[25]

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 Hareware 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。

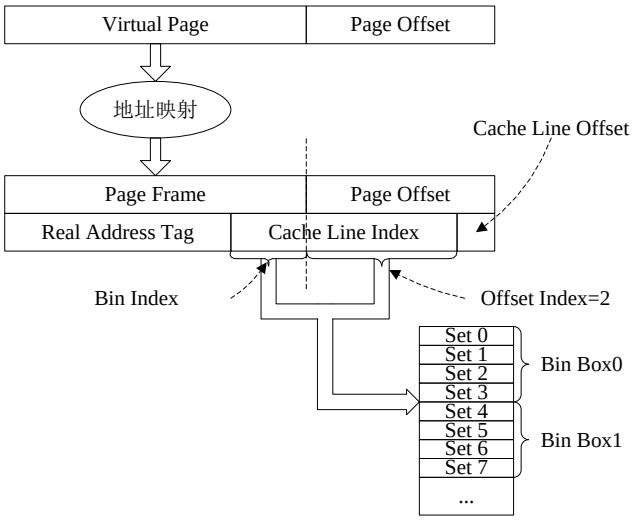


图 2-8 分页机制下 Cache 的使用方式[30]

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。

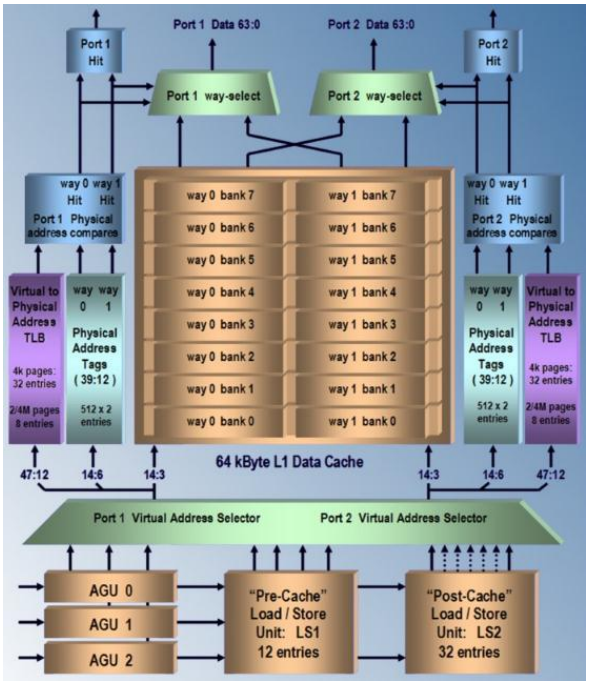


图 2-9 Opteron 的 L1 Data Cache[6]

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。

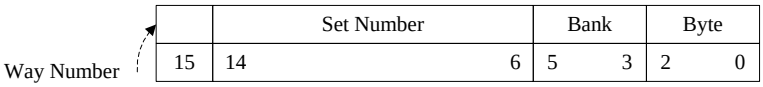


图 2-10 Cache 的 Data 阵列地址编码方式

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，Clock 和 Clock-Pro 算法等。

这些算法在分布存储，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。

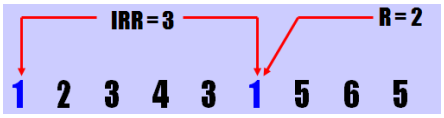


图 2-11 IRR 和 Recency 参数的计算示例[41]

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 的近似结果。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。

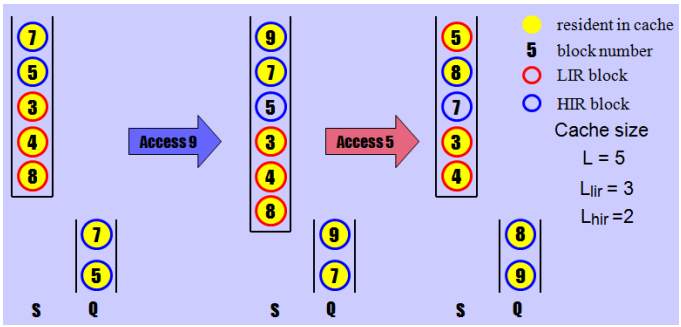


图 2-13 访问不在 Cache 命中的 HIR 页面[41]

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。

下文讨论的页面替换算法针对 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。

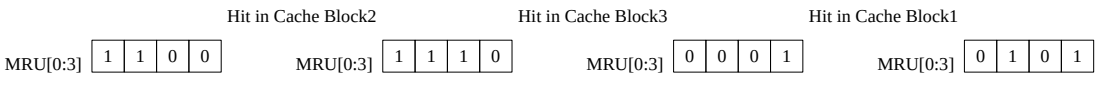


图 2-14 MRU-Based PLRU 算法

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。

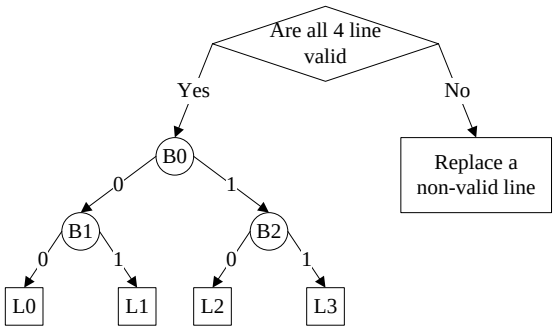


图 2-15 Tree-Based PLRU 替换算法

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。

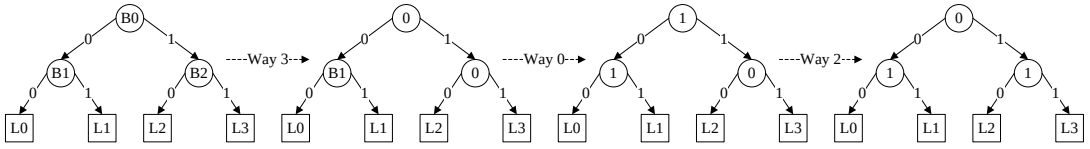


图 2-16 Way3, 0 和 1 访问序列

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。这是因为连续地址的访问不会影响到 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。

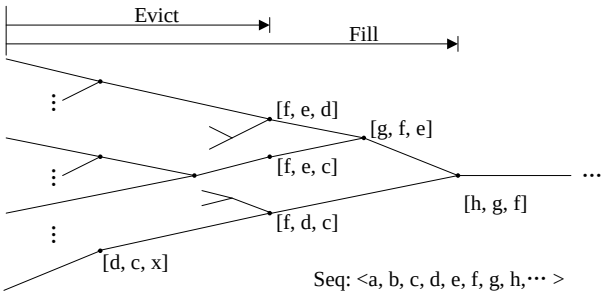


图 2-17 Fill(k)与 Evict(k)参数[37]

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。

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

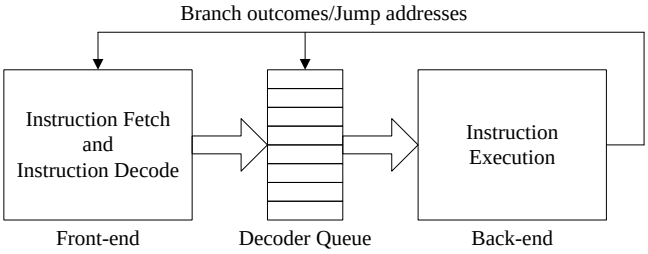


图 2-18 Front-end 与 Back-end 之间的关系[67]

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。

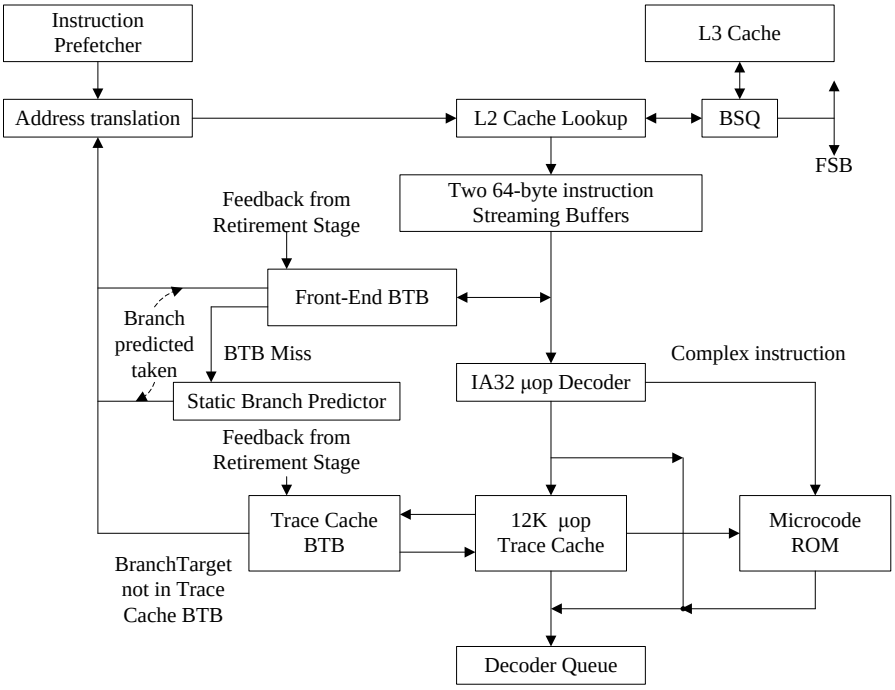


图 2-19 Pentium IV 架构的 Front-End[68]

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。

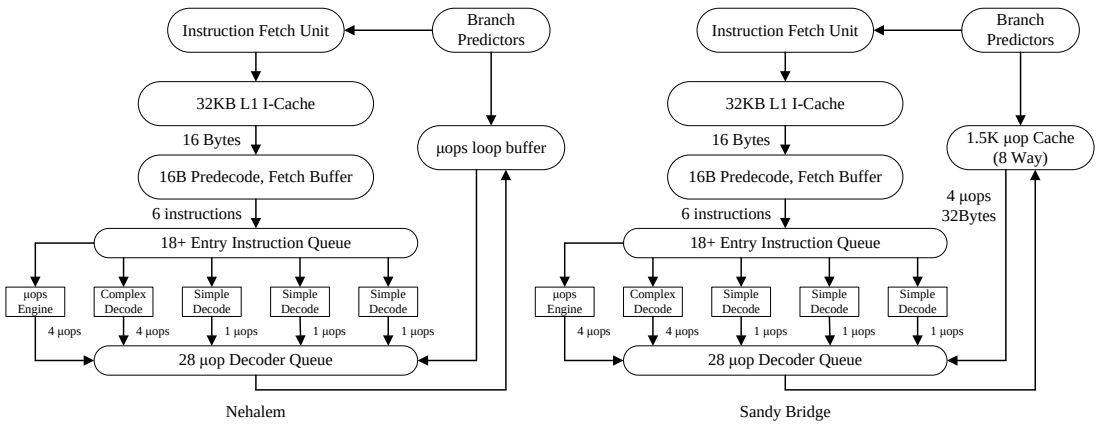


图 2-20 Nehalem 和 Sandy Bridge 的 Front End[69]

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。

$$\mathrm{Min}\left[\frac{H+M}{H+M\times[T_M+B]},\frac{H+M}{M\times[T_M+(1+d)\times B]}\right]=\mathrm{Min}\left[\frac{1}{1+m\times[T_M+B-1]},\frac{1}{m\times[T_M+(1+d)\times B]}\right] \quad \text{公式 2-5[49]}$$

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。

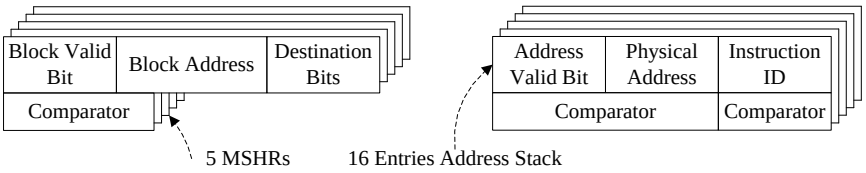


图 2-21 基于 Address Stack 的 Non-Blocking Cache[53]

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$(S_1+S_2+…+S_n)!$ / $(S1!×S_2!×…×S_n)$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。

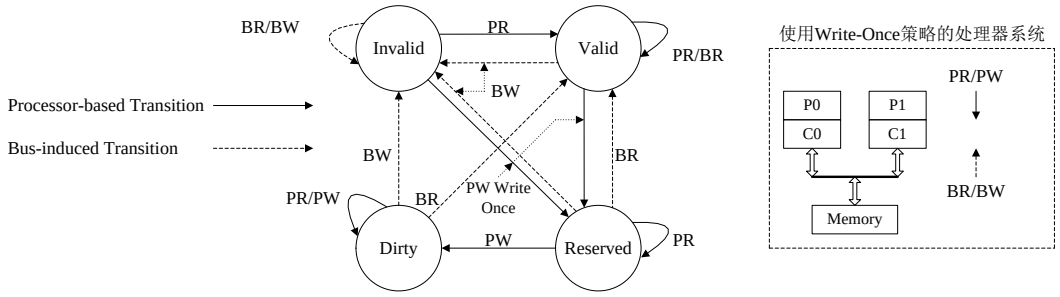


图 3-1 Write-Once Transition Diagram[①]

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。

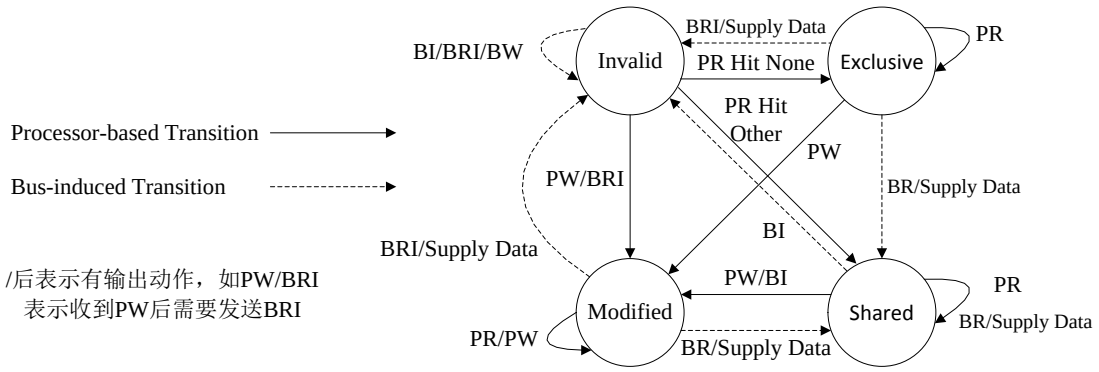


图 3-2 Illinois Protocol Transition Diagram[①]

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。

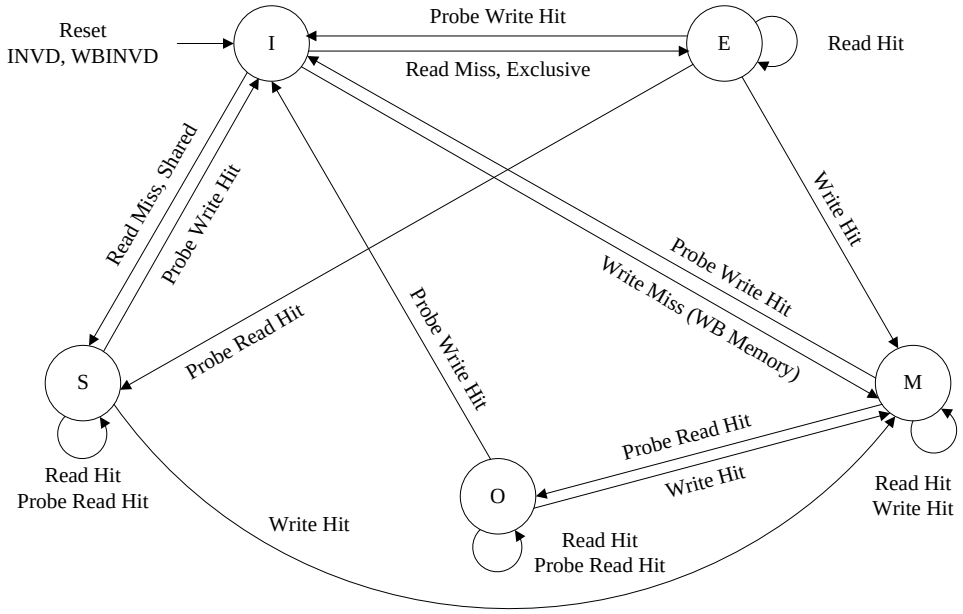


图 3-3 MOESI Protocol Transition Diagram[62]

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。

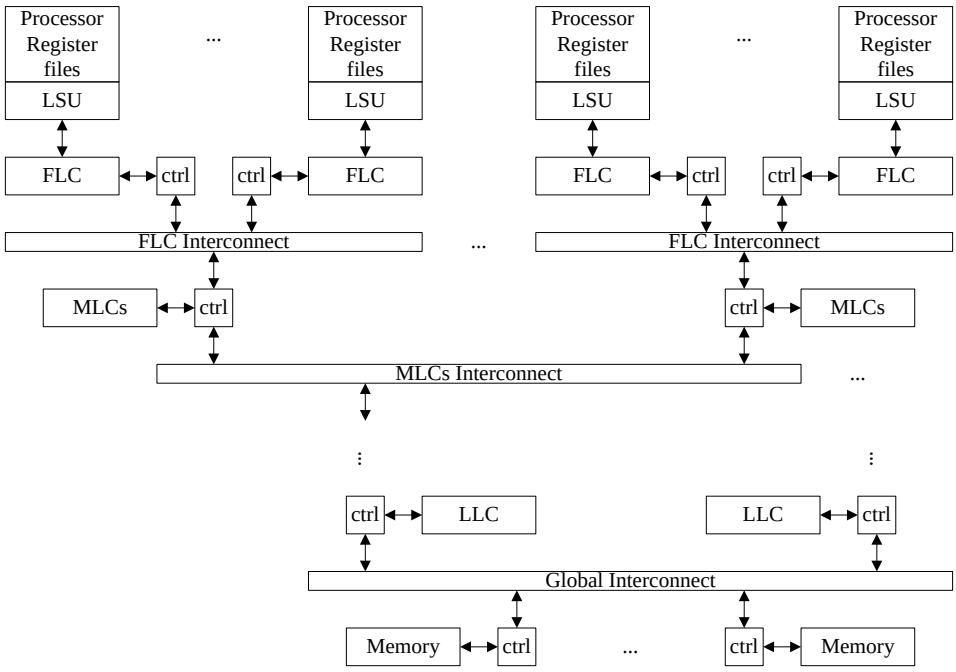


图 4-1 Cache Memory 的层次结构[66]

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。

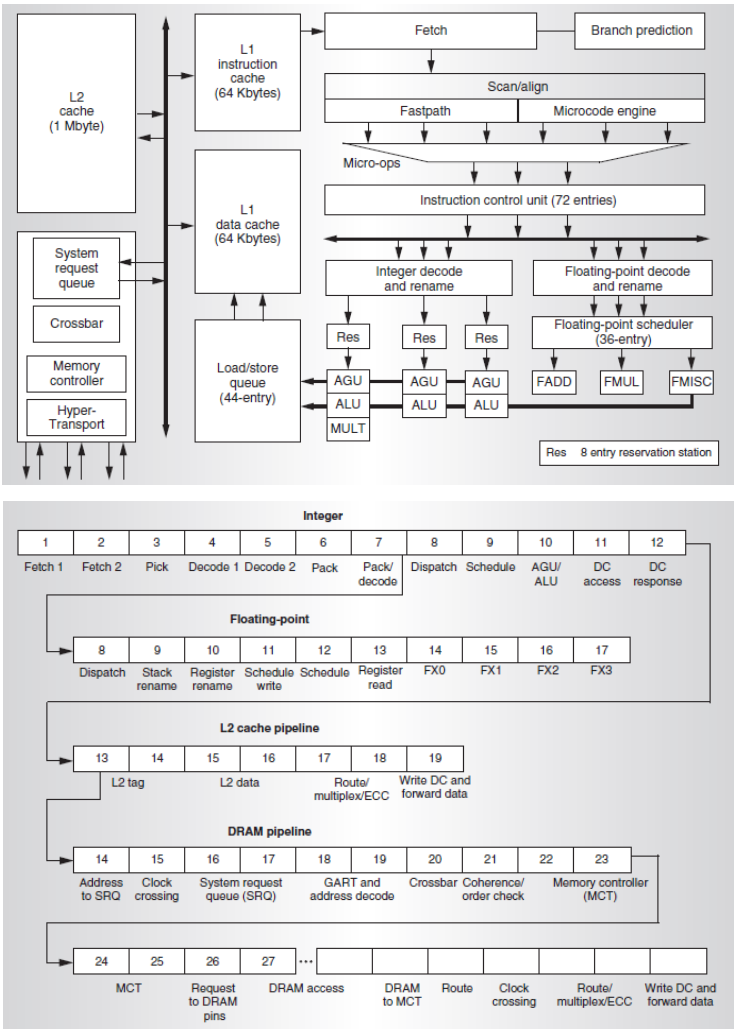


图 4-2 Opteron 微架构的基本组成结构和指令流水线[72]

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。

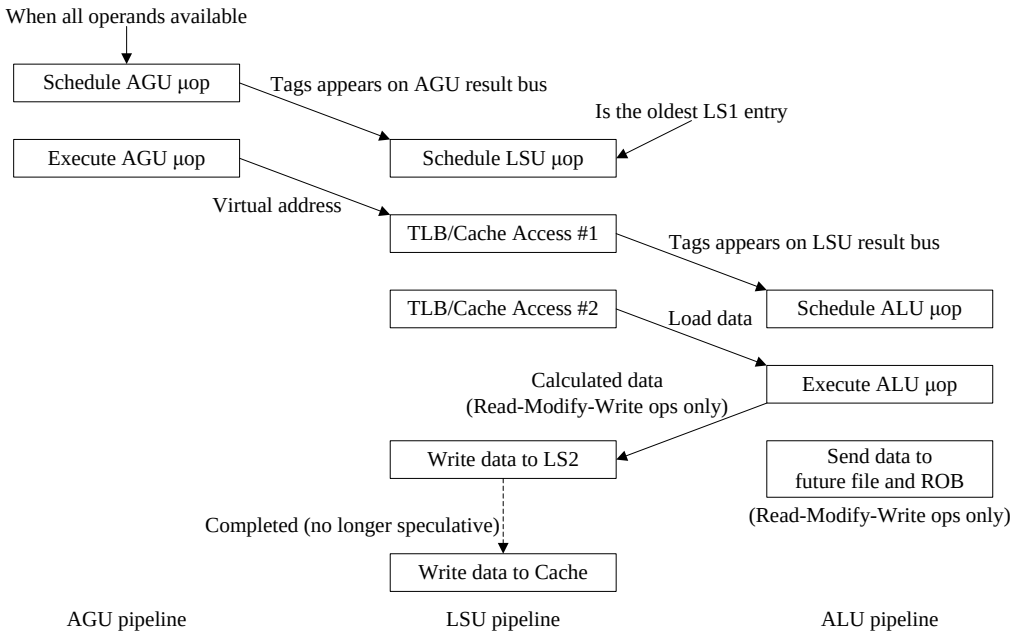


图 4-3 AGU/ALU 与 LSU 流水线间的依赖关系

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。

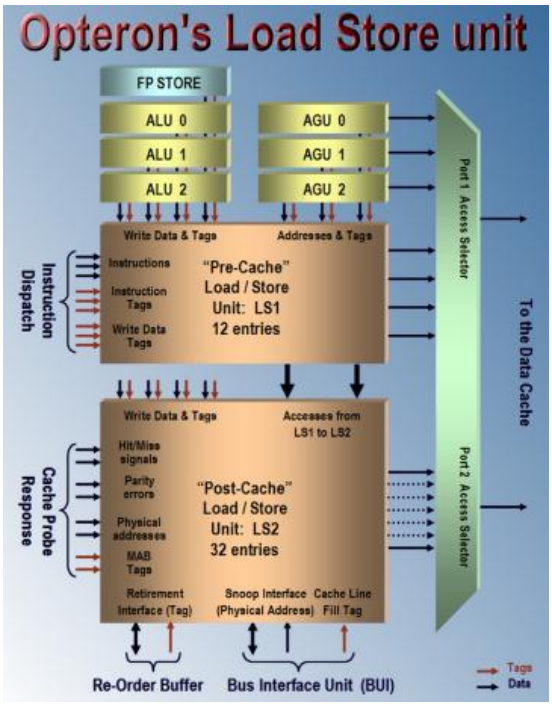


图 4-4 Opteron 的 LSU 的示意图[6]

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。

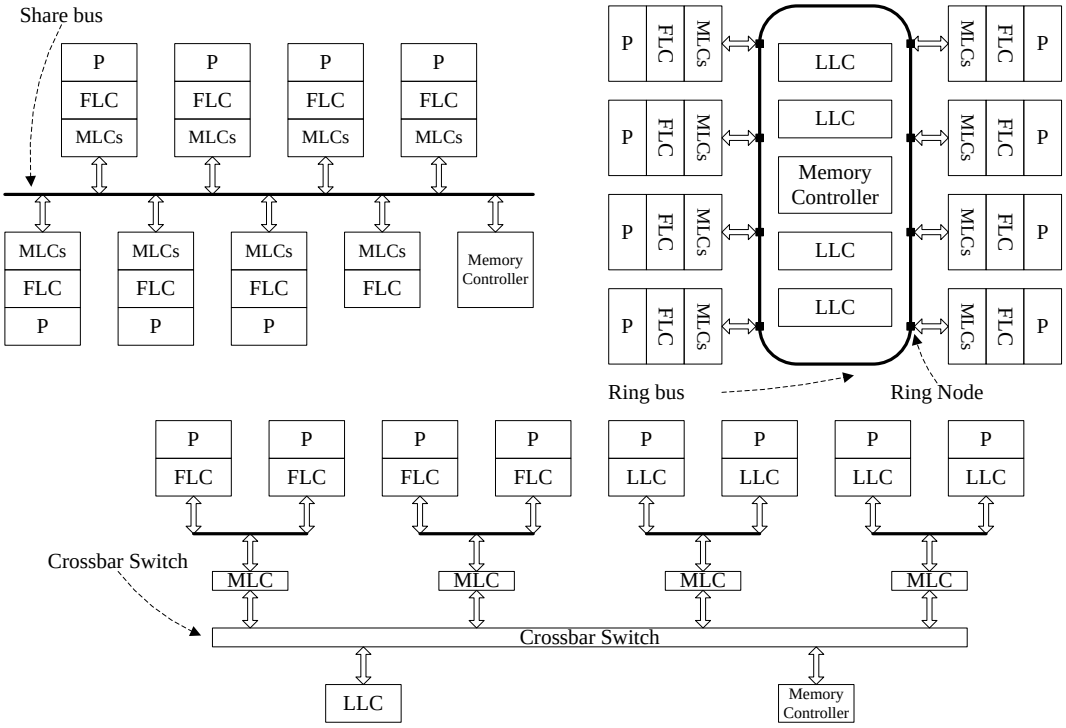


图 4-5 CPU Core 与 Cache Hierarchy 的连接通路

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。

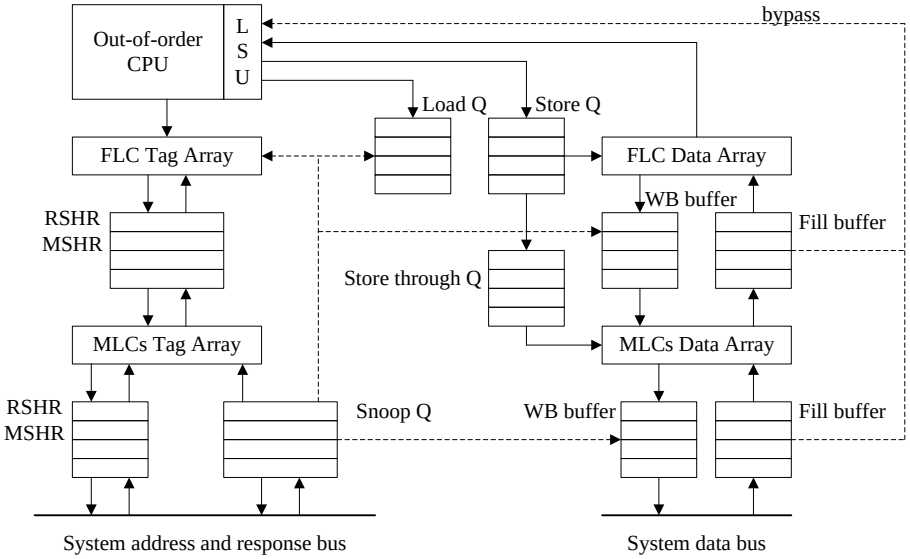


图 4-6 CPU 与数据 Cache 的连接关系

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。

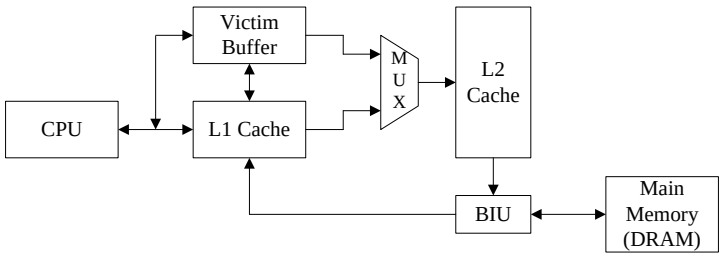


图 4-7 L1 Cache，L2 Cache 与 Victim Buffer 的组成结构[89][②]

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。

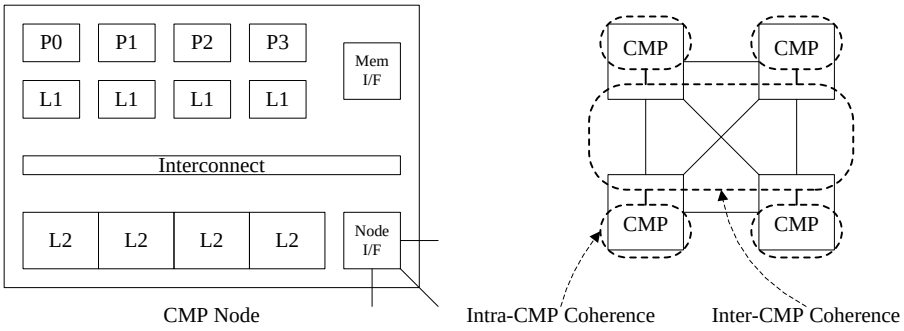


图 4-8 Intra-CMP Coherence 和 Inter-CMP Coherence 之间的关系[91]

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。

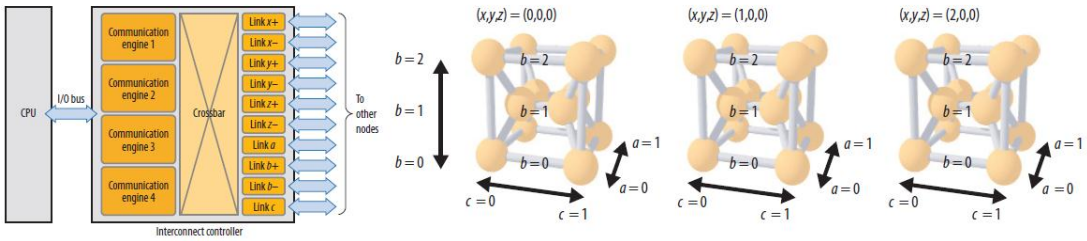


图 4-9 6D Mesh/Torus Interconnect[92]

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。

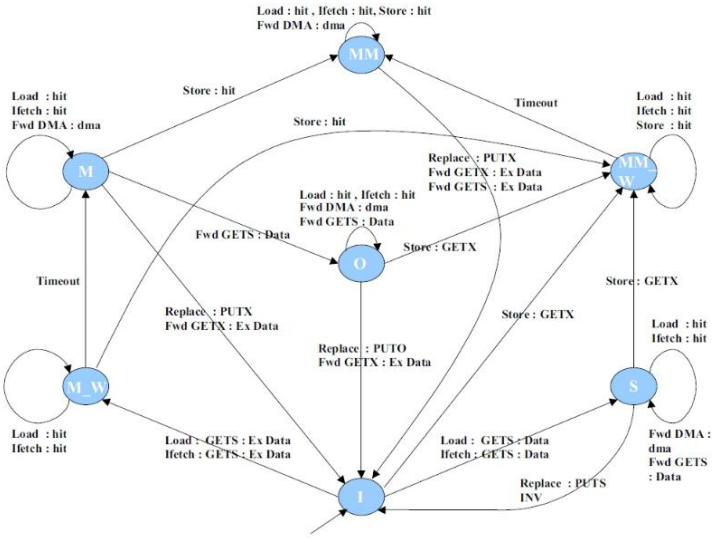


图 4-10 L1 Cache Controller 的 FSM[95]

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。

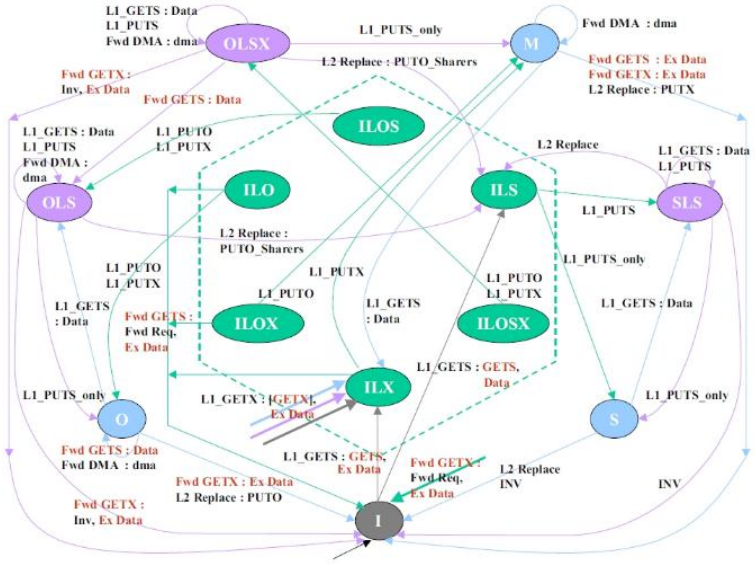


图 4-11 Intra-Chip Inclusion FSM[95]

{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。

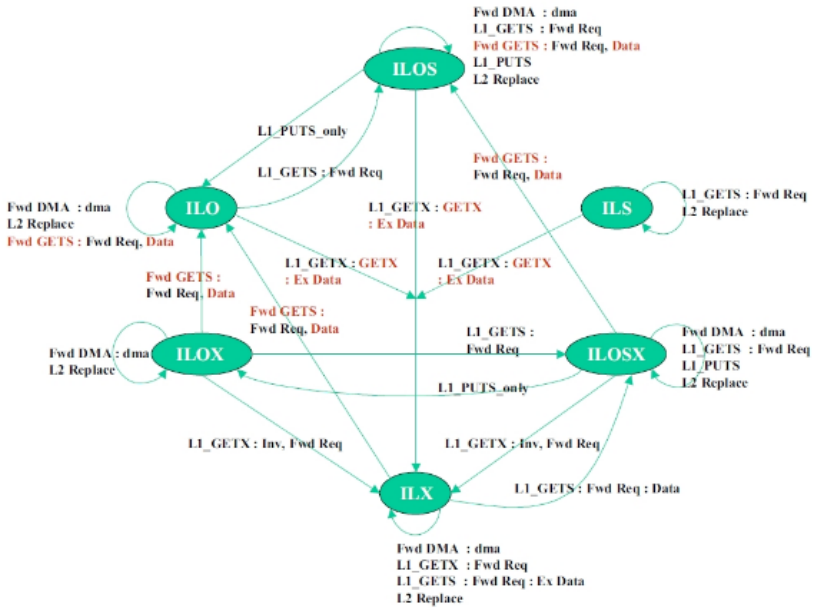


图 4-12 与 L1 Cache Block 相关的 FSM[95]

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。

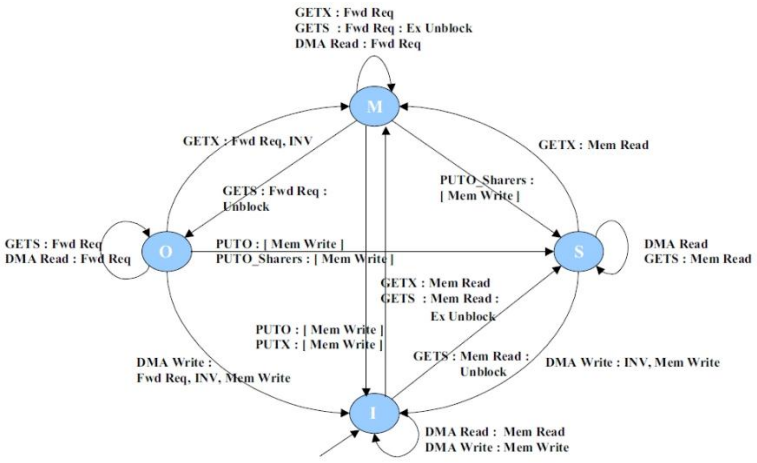


图 4-13 Directory Controller 使用的 FSM[95]

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。

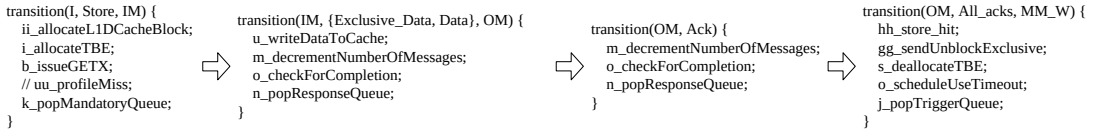


图 4-14 状态 I 至 MM_W 的迁移过程

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[96]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 [97]。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[97]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[97]，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][99]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[98]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。

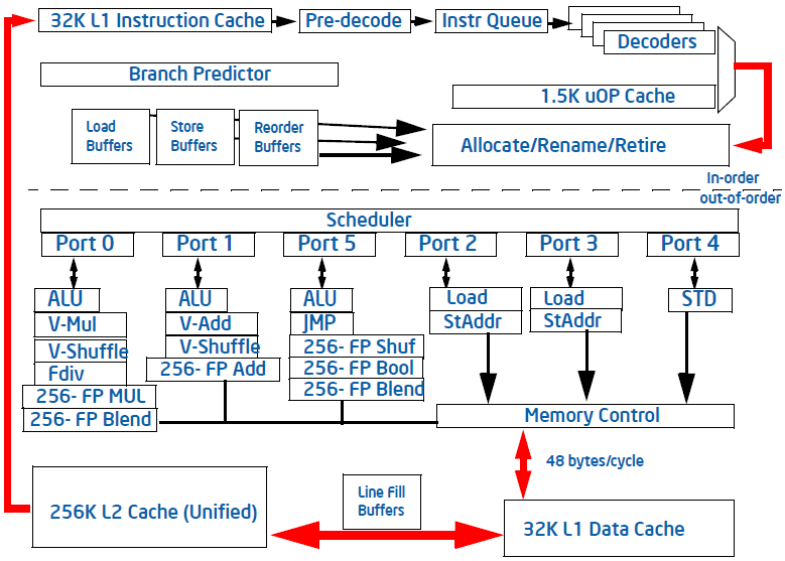


图 4-16 Sandy Bridge 微架构示意图[98]

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。

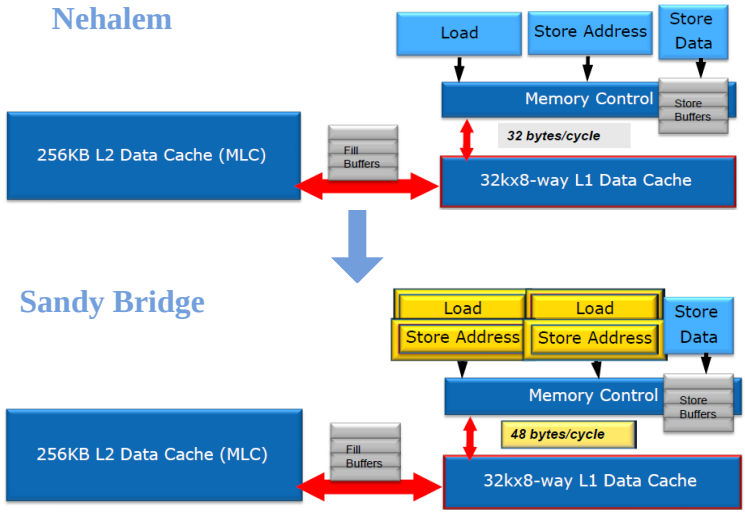


图 4-17 Nehalem 和 Sandy Bridge 的 LSU[1]

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 中可以执行两条 128b 的 Load 和一条 128b 的 Store 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)[98]，Cache Box[1][99]，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。

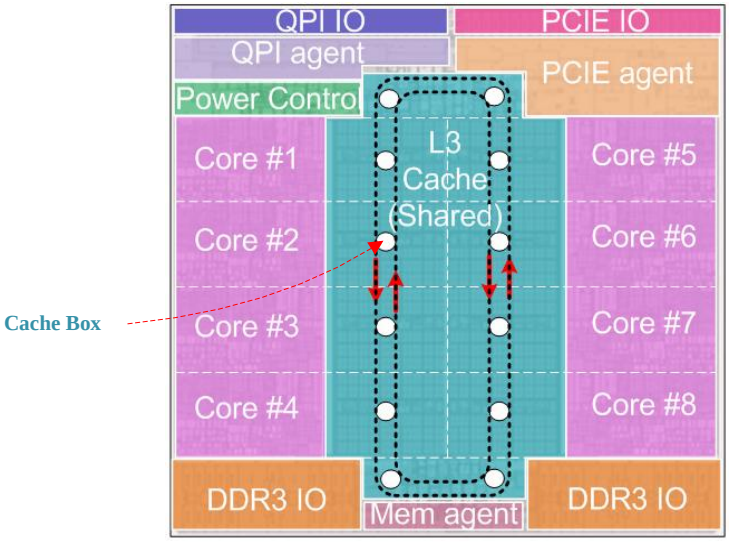


图 4-18 Sandy Bridge EP 处理器的组成结构[1][99]

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[99]。

所有 CPU Core，LLC Slice，QPI Agent，iMC，PCIe Agent，GT(Graphics uniT)通过 Ring Bus 连接在一起[1][99]。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。

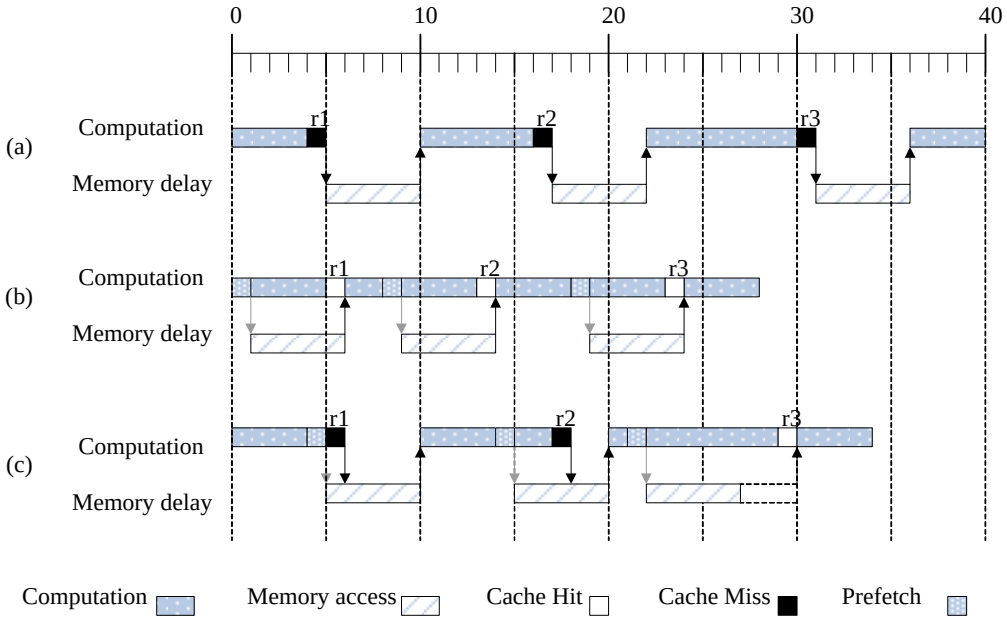


图 5-1 数据预读机制示意[100]

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[101]。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。

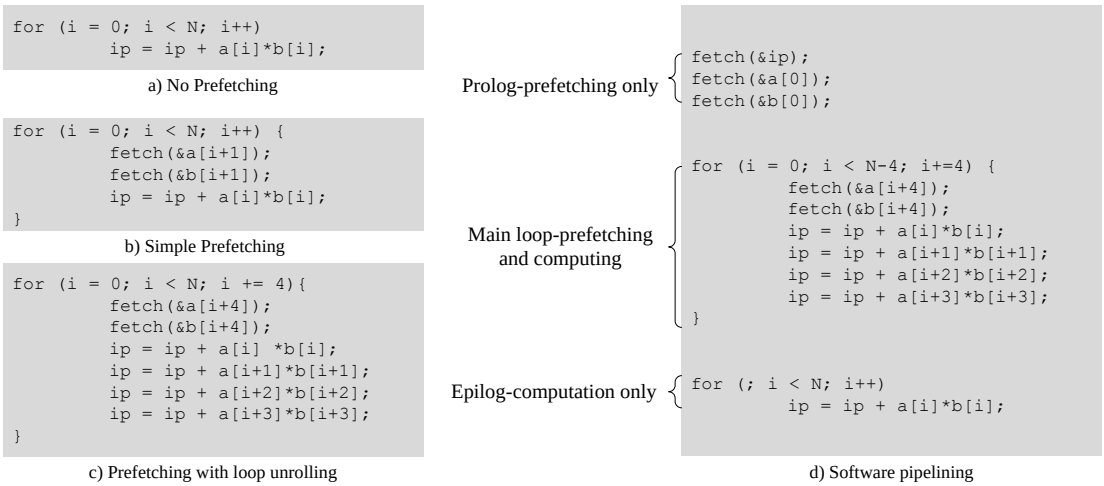


图 5-2 利用软件预读指令进行程序优化[100]

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) [100]。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[100]

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 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。

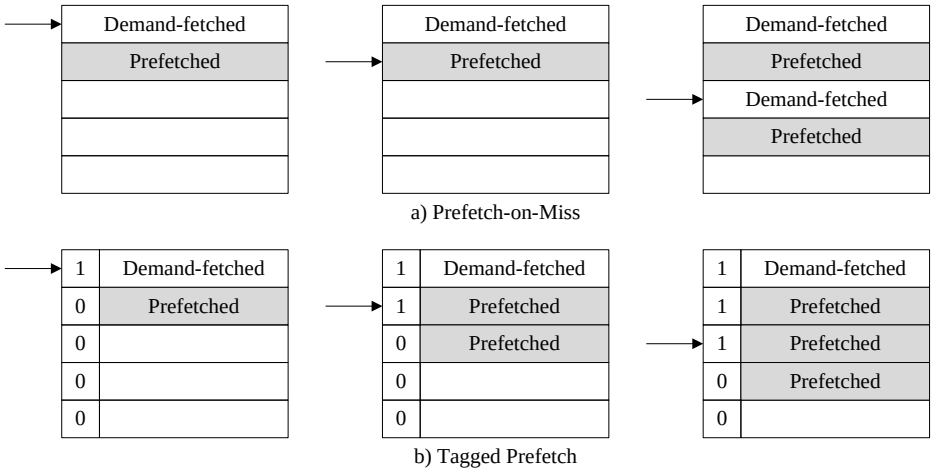


图 5-4 Prefetch-on-Miss 与 Tagged Prefetch 实现方式的比较[100]

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[102]。该实现的组成结构如图 5-5 所示。

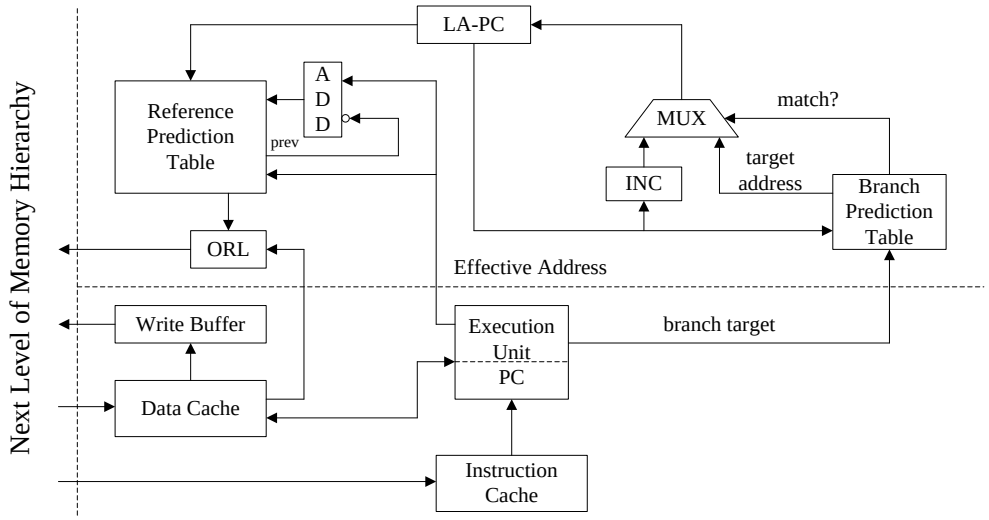


图 5-5 Lookahead data prefetching 实现的组成结构[104]

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。

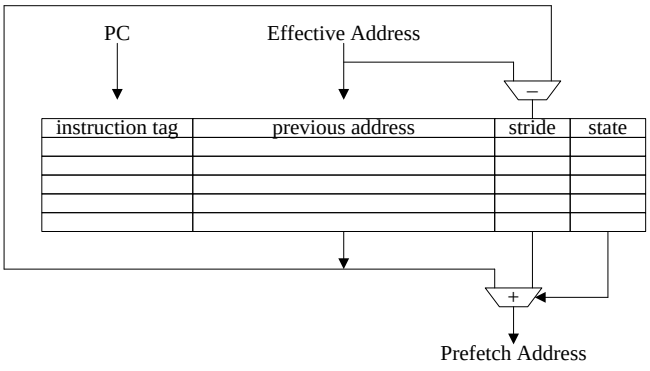


图 5-6 RPT 的组成结构

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。

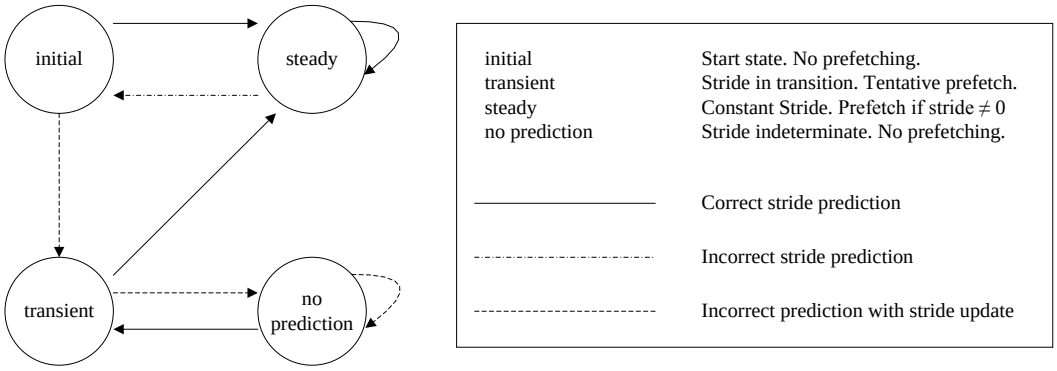


图 5-7 RPT state 的状态迁移

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[103]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[103]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。

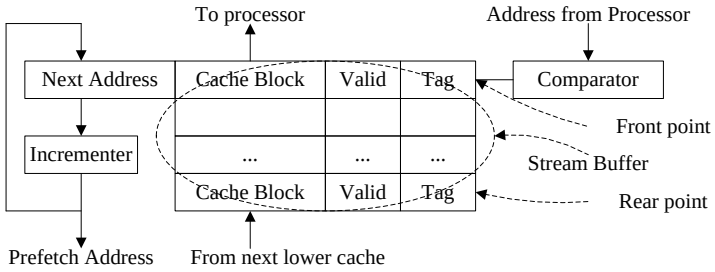


图 5-9 Stream Buffer 的组成结构[105][①]

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，[105]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。

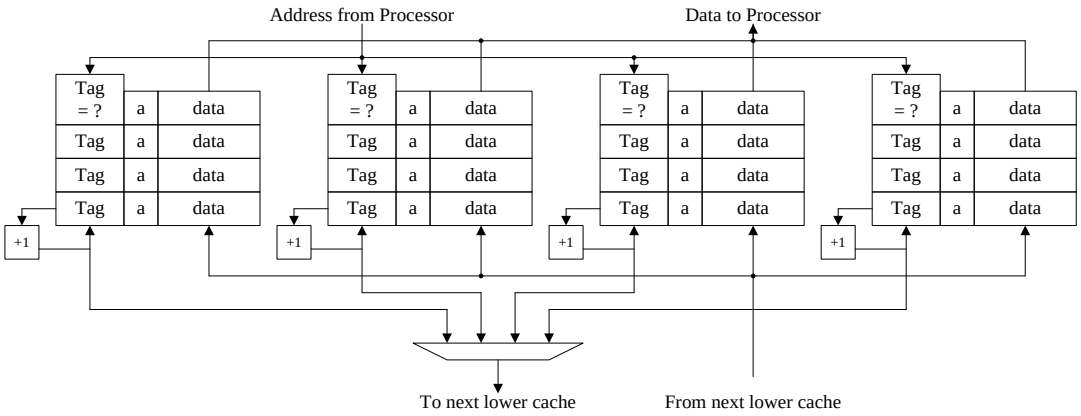


图 5-10 Multi-way stream Buffer 的组成结构[20]

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。

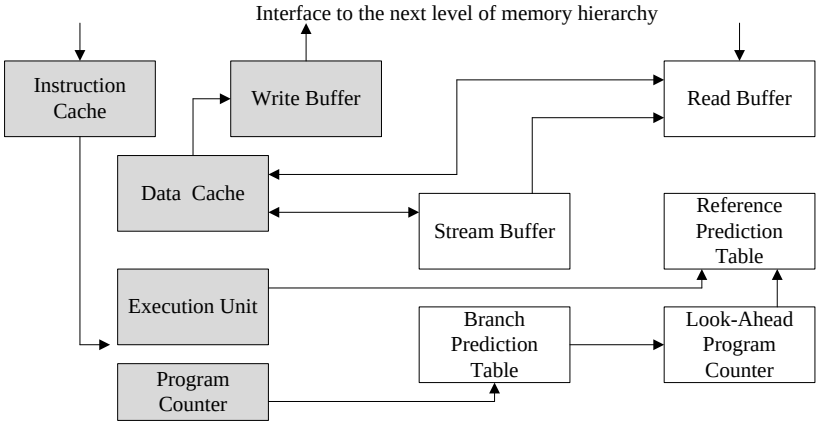


图 5-11 联合使用 Stream Buffer 与 Lookahead data prefetching 方式[20]

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 et al [Jul. 2005]. Forward State for use in Cache Coherency in a Multiprocessor System. U.S. Patent 6,922,756.

[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] 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.

[97] 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.

[98] Intel [June 2011]. Intel 64 and IA-32 Architectures Optimization Reference Manual. Section 2.1.1 Intel microarchitecture code name Sandy Bridge Pipeline Overview.

[99] David Kanter [Jul. 2011]. Sandy Bridge for Servers. http://realworldtech.com/page.cfm?ArticleID=RWT072811020122&p=1

[100] Steven P. Vanderwiel and David J. Lilja [Jun. 2000]. Data prefetch mechanisms, ACM Computing Surveys (CSUR), v.32 n.2, p.174-199.

[101] Doug Joseph and Dirk Grunwald [May 1997]. Prefetching using Markov predictors. ISCA '97 Proceedings of the 24th annual international symposium on Computer architecture.

[102] 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.

[103] 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.

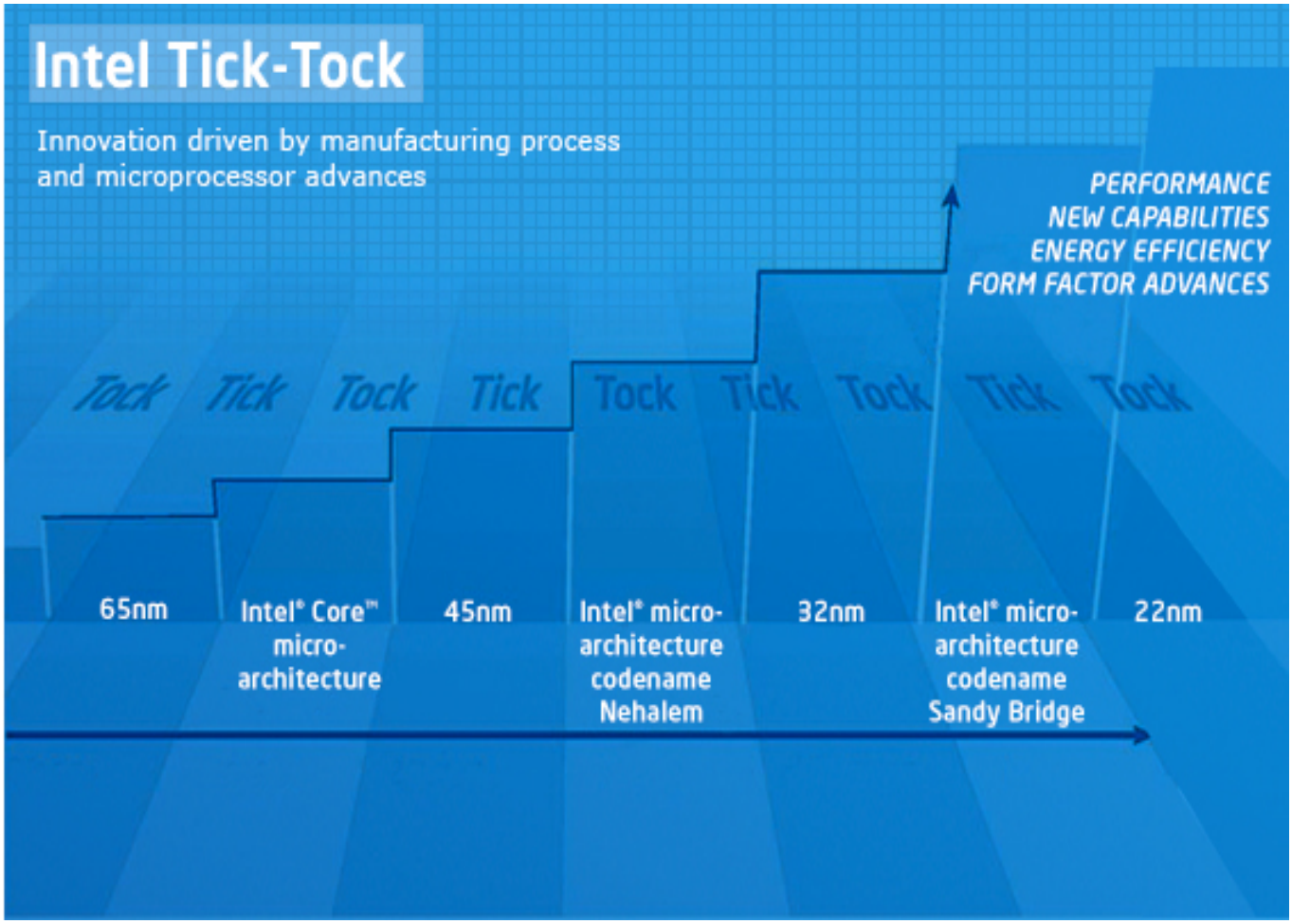
[104] G. S. Manku, M. R. Prasad, and D. A. Patterson [Dec. 1997]. A new voting based hardware data prefetch scheme. Proc. of IEEE Int. Conf. High-Performance Computing, pp.100 - 105, 1997.

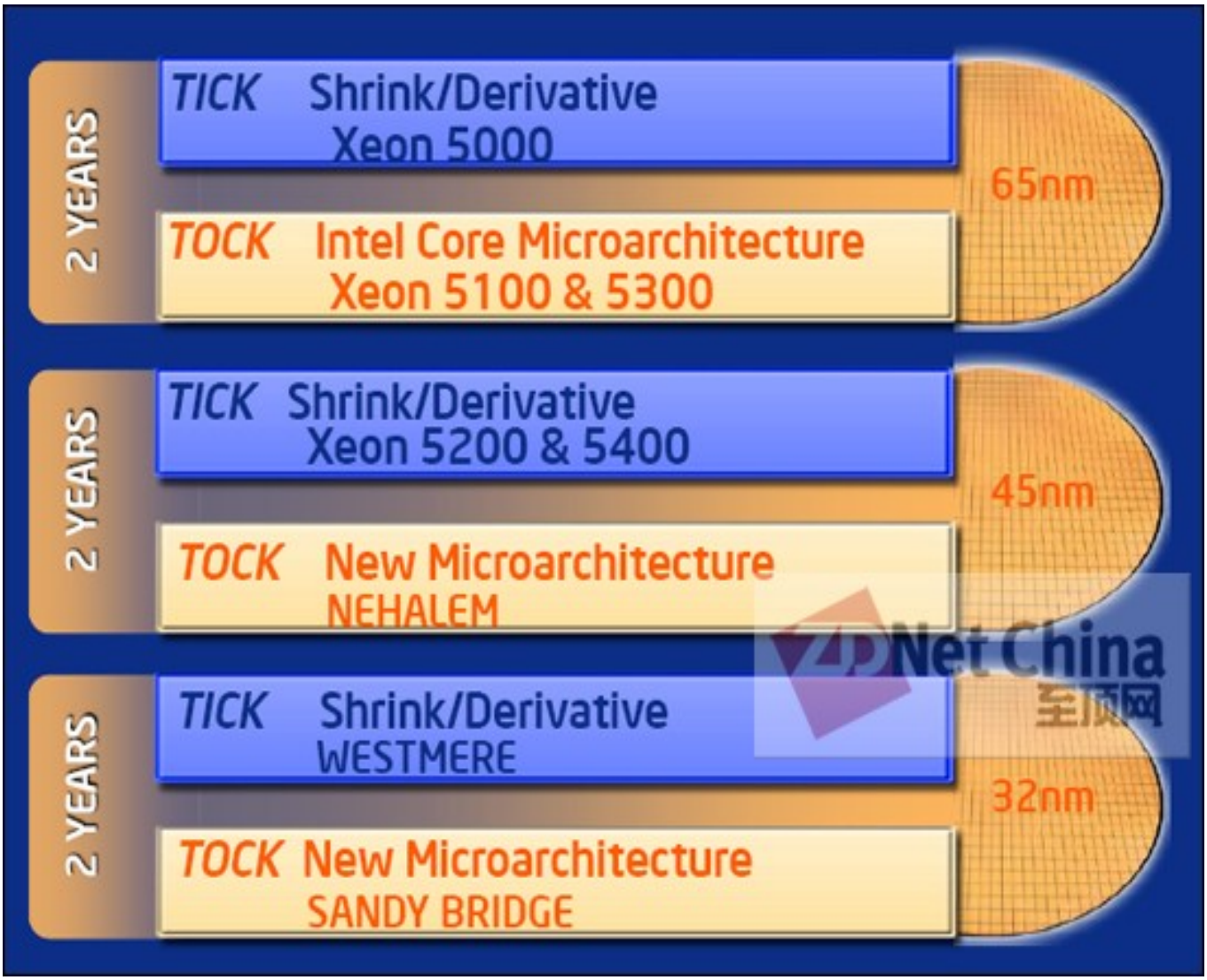
[105] S. Palacharla and R. E. Kessler [Apr. 1994]. Evaluating stream buffers as a secondary cache replacement, Proceedings of the 21ST annual international symposium on Computer

architecture, p.24-33, April 18-21, 1994, Chicago, Illinois, United States.

Intel CPU微结构–过去，现在和将来

【陈怀临注：时光如梭。上次写这篇文章是2008年10月。转眼2年过去。这次做一些修订和校注。】
Intel CPU的产品称呼比较混乱。这一点对于在Intel工作的员工也是如此。下面是笔者在这方面的一些经验。希望对读者同学们有所帮助。
首先，对非专业人士而言，接触的名称通常为CPU的产品品牌（Brand Name），而不是微结构（Micro-Architecture）名称。什么是微结构？就是你看不见的那些东西。例如流水线（Pipeline）的设计，局部总线（Local Bus），缓存（Cache）的设计，存储总线（Memory Bus）等。到目前为止，一般而言，Intel CPU微结构的系列为：i386, i486, P5, P6, Netburst, Pentium-M，Core（Merom 65nm和 Penryn 45nm）。Core微结构的65nm流片有：Merom，Concore，Allendale等；45nm的流片Penryn，Workfiled，Yorkfield等。Penryn是Merom的直系后代。Nehalem是基于Penryn 45纳米流程的直系后代。从Penryn 45nm工艺流程的基础上，Nehalem微结构横空出世。在Nehalem与Core微结构的继承性上，目前英文的wiki上是不精确的。Nehalem最大的特点就是FSB和（或）北桥的消失，QPI的引入和把在Core微结构抛弃（或曰暂停的）HyperThreading从新引入。 Nehalem之后的事情比较清爽：Nehalem, Westmere(Nehalem在32nm工艺下的Tick），Sandy Bridge，Ivy Bridge（Sandy Bridge在28nm 工艺下的Tick）和Haswell。通常，我们也可以说Nehalem和Westmere都属于Nehalem微结构。Sandy Bridge和Ivy Bridge都属于广义的Sandy Bridge微结构。这种关系就是Intel现在非常重要的Tick Tock的流程。如下图所示，为Intel的TT模型。


Intel Tick-Tock
Innovation driven by manufacturing process and microprocessor advances
PERFORMANCE
NEW CAPABILITIES
ENERGY EFFICIENCY
FORM FACTOR ADVANCES
Tock
Tick
Tock
Tick
Tock
Tick
Tock
Tick
Tock
65nm
Intel® Core™ micro-architecture
45nm
Intel® micro-architecture codename Nehalem
32nm
Intel® micro-architecture codename Sandy Bridge
22nm



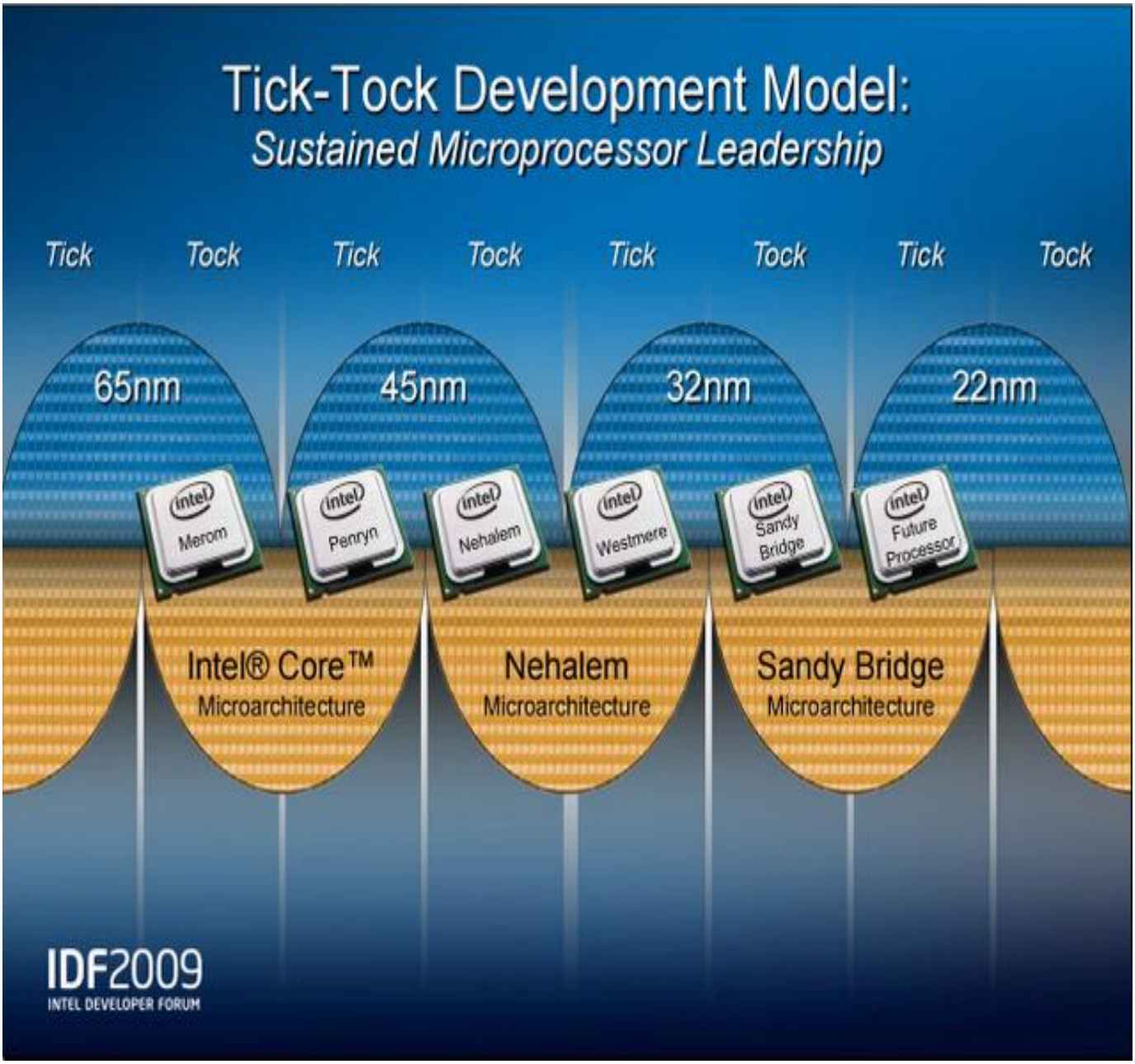
Tick其实就是Introduce一个新的工艺和各种PROCESS。Tock就是Introduce一个新的微结构。

工艺与结构的拆分对于Intel这些年来的成功是根本的。这里面的原因其实也很简单：如果一个芯片建立在一个新的工艺 AND又是一个新的微结构上。一旦出了问题，基本上整个公司就歇了一半了。CEO就要买豆腐撞死。没有Baseline的事情，最好不要去做。所以TT模式可以你确保，在一个已经酒精考验的PROCESS下，Tock一下，如果芯片有问题，那就基本上是新的结构出了问题。如果一个基于考验过的微结构在一个新的工艺下（Tick一下），如果芯片有问题，显然是工艺方面有bug。。。。总而言之，这样一个拿着billion美金烧出来的芯片，就可控了。。。

另外，TT模型的另外一个好处就是把工程师队伍有效的运作起来。例如，Core的Merom和Pennyn都是Intel以色列团队主导做的；然后Nehalem和Westmere是美国Oregon团队做的；然后Sandy Bridge和Ivy Bridge又是犹太人主导；再下面再是美国团队做。TT下去，循环往复。。

Intel在这个方面是从血的教训里学来的。当年的最新的NetBurst redesign Prescott在

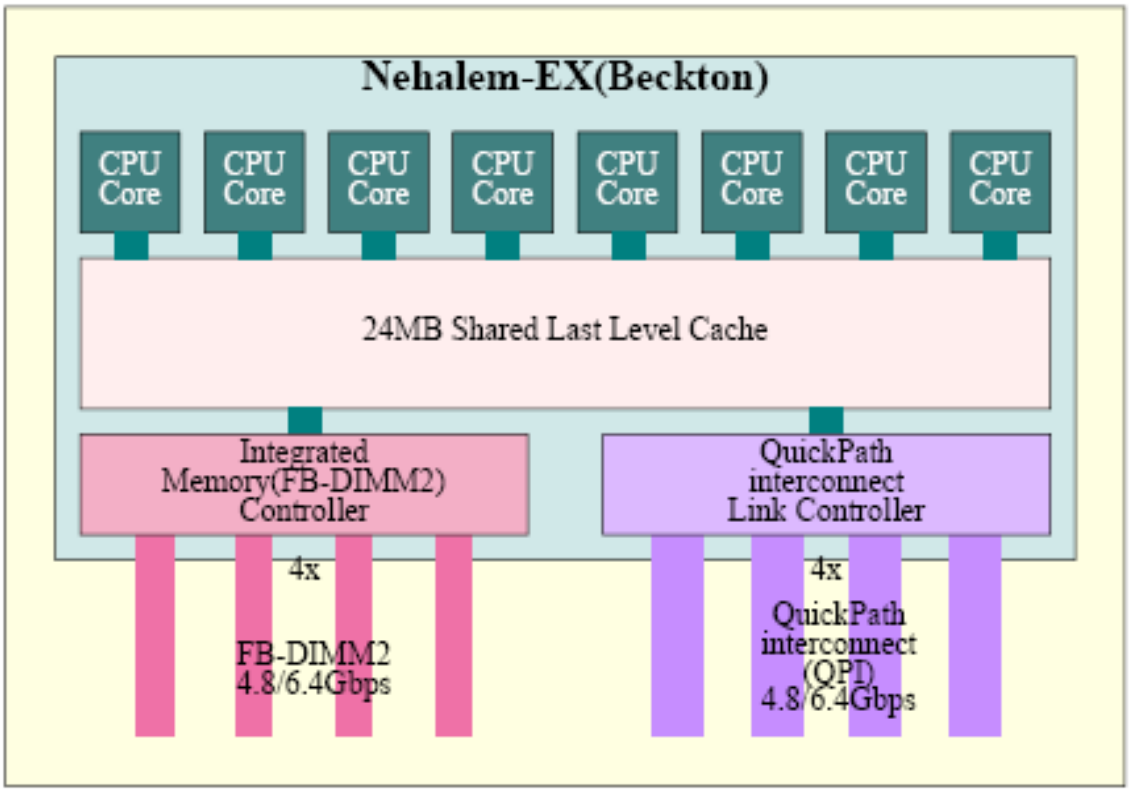
90nm就是Tick(90nm
新工艺与Tock（x86-64Extension）在一起了。一通混战之后，最后发现是Power Leakage问题无法解决，从而导致Intel彻底放弃NetBurst，从新捡起P6微结构。。。
所以，Intel的微结构变化可以简单概括为：
i386, i486, P5, P6, Netburst, Pentium-M，Core（Merom，Penryn），Nehalem(Nehalem, Westmere), Sandy Bridge(Sandy Bridge, Ivy Bridge), Haswell(Haswell, Rockwell)。
从下面这张Intel解释其Tick Tock的图中读者也可以清晰的认识到这些微结构的家族关系：

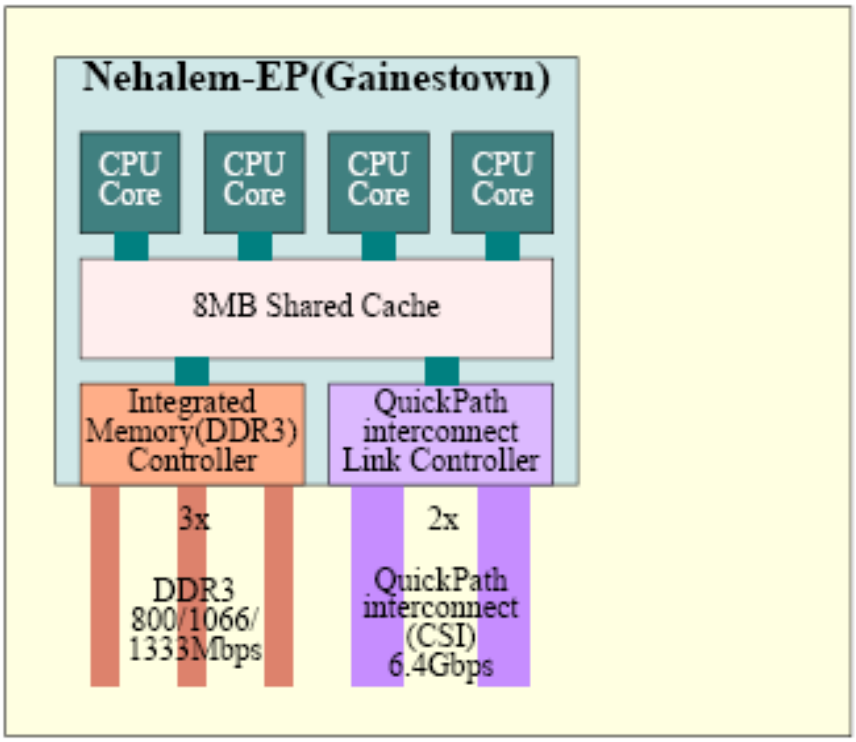


多个CPU产品可以来源于同一个微结构。其意思是一代产品。属于同一个微结构的多

款CPU基本上可以认为是同一类，或同一代产品。同一个微结构下的多款CPU的原因很多，例如，简单化的版本；某个特定市场的定制版本等。
同一个名称的CPU可以是来自不同的微结构。如Celeron CPU有P6微结构的，Netburst微结构的，Pentium-M微结构的和Core微结构的。如果不懂的话，您购买了一个Netburst微结构的Celeron，在性能价格比上就不好了。在同年代中，您当然应该选择Pentium-M的款式。对Intel的Xeon名称的CPU也一样，读者可以发现，Xeon可以来自不同的微结构技术。
因此，CPU的产品名称基本上没有用。一定要知道其来自哪个微结构。换言之，CPU名称与微结构的映射关系是M:N。
这些微结构名词与我们日常看到的广告上的"Inside Intel"的名称的关系如下：
(本文不讨论IA64体系结构。笔者认为IA64除了在科学计算方面，基本上没有任何意义了。)
<微结构名称>: {CPU品牌（Brand）}+
i386: 80386DX, 80386SX, 80376, 80386SL, 80386EX
i486: 80486DX, 80486SX, 80486DX2, 80486SL, 80486DX4
P5: Pentium, Pentium with MMX
P6: Pentium Pro, Pentium II, Celeron (Pentium II-based), Pentium III, Pentium II and III Xeon, Celeron (Pentium III Coppermine-based), Celeron (Pentium III Tualatin-based)
Netburst：(32位)Pentium 4, Xeon, Mobile Pentium 4-M, Pentium 4 EE, Pentium 4E, Pentium 4F，（64位)Pentium D, Pentium Extreme Edition, Xeon
Pentium-M：Pentium M, Celeron M, Intel Core, Dual-Core Xeon LV, Intel Pentium Dual-Core
Core：（64位）Xeon, Intel Core 2, Pentium Dual Core, Celeron M
Nehalem：Xeon， Core i7，Core i7 Extreme，Core i5。
Westmere：Xeon， Core i7，Core i7 Extreme，Core i5， Core i3，Pentium，Celeron
在基于Nehalem微结构下45nm工艺下流片的CPU有：

Nehalem(45nm)
桌面（Desktop）
Bloomfield (4核，8线程)
Lynnfield（4核，8线程）//存在4/4，无HyperThreading变种
Clarkfield（4核，8线程）
服务器（Server）
Nehalem-EX（8核，16线程）（Beckton）
Nehalem-EP（4核，8线程）(Gainestown) //存在4/4，2/2无HyperThreading变种
下图是Nehalem-EX和Nehalem-EP的结构比较图：



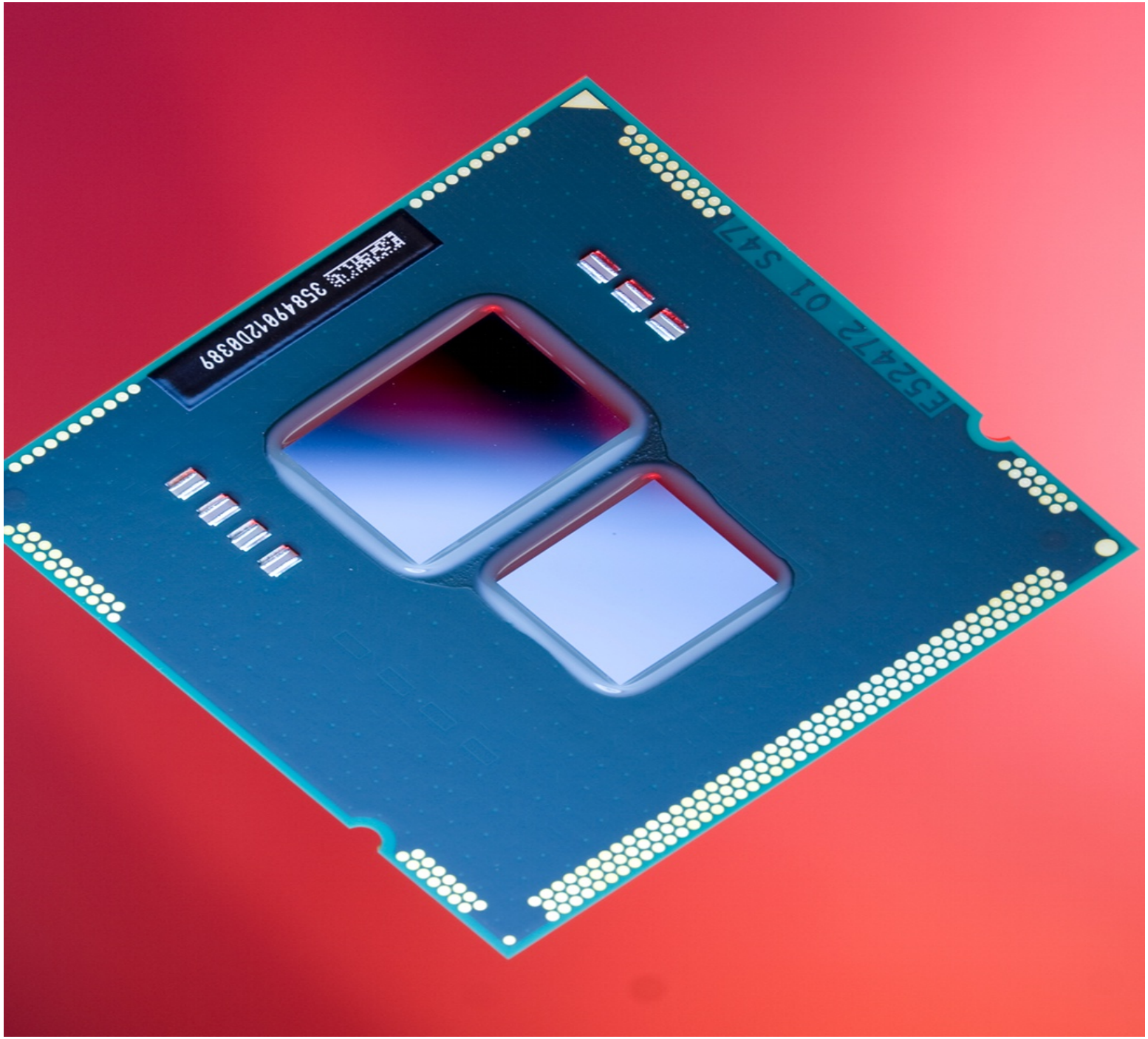


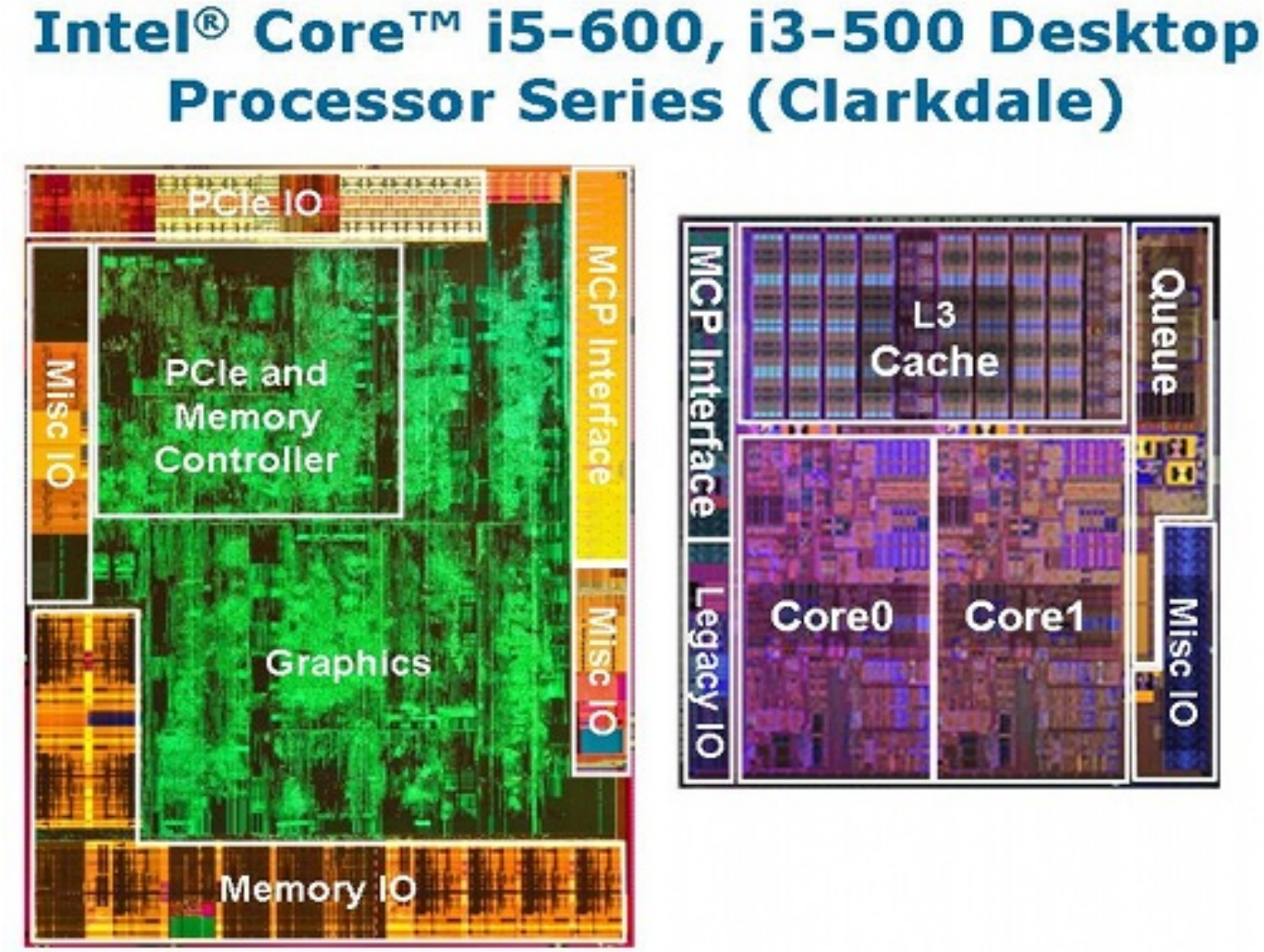
Westmere(32nm)

Westmere-EX (还没有出来。应该10个核，20个线程)

Westmere-EP（6核，12线程）（Gulftown）//存在4/8，4/4变种。XEON 5690为

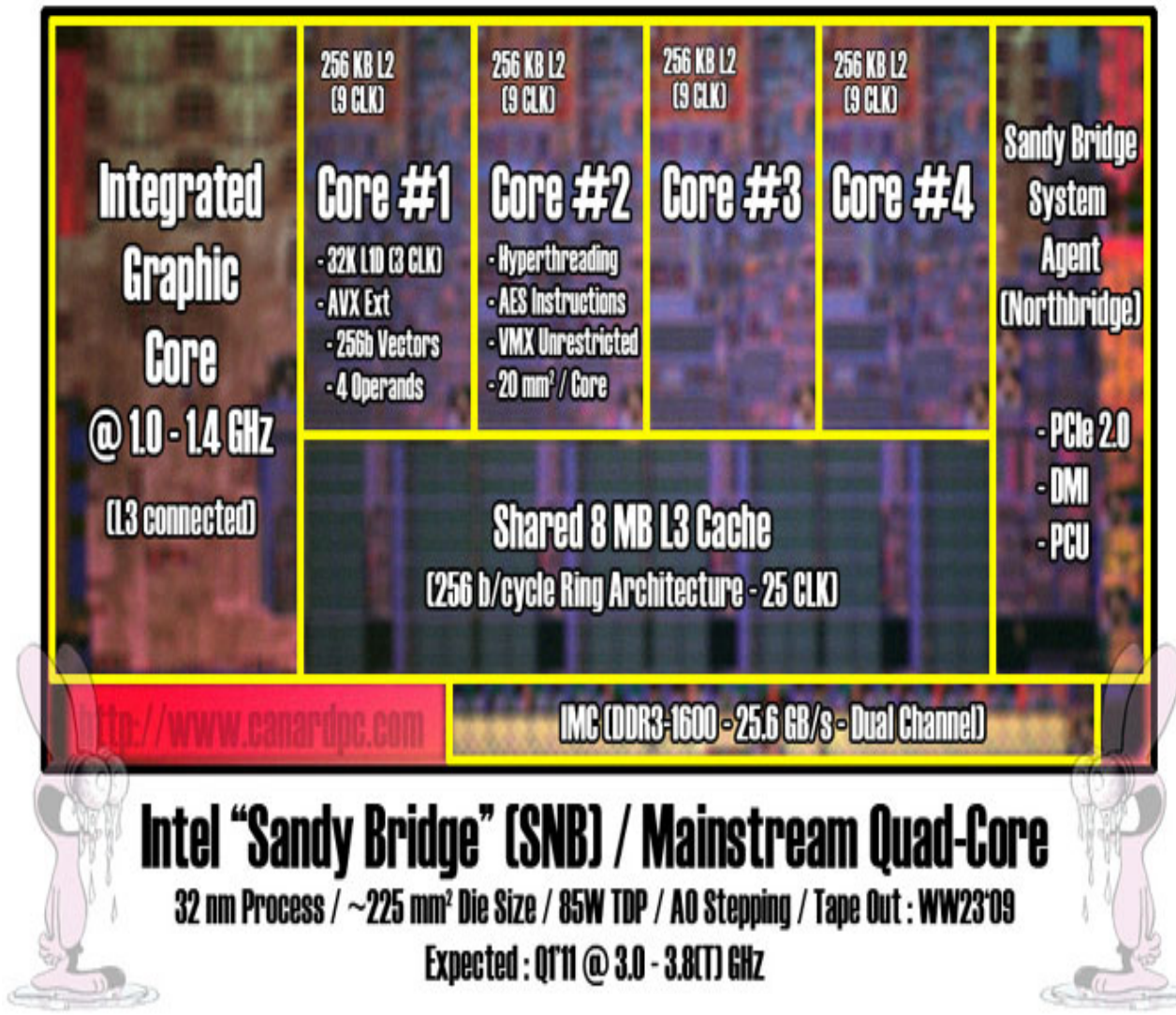
6/12.
Clarkdale（2核，4线程和加了一个45nm的iGFX）//其实是两个die。一个package。
Arrandale（2和，4线程和加了一个45nm的图形卡iGFX）。//其实是两个die。一个package。
下图所示为Westmere的Clarkdale的芯片图。一个Package，二个Die。CPU Die是32nm；Graphic芯片是45nm。换言之，就是混在一起完事交差。读者想想马上要出来的无缝的Sandy Bridge的GPU集成，就知道其中之区别了。


E52412 01 S47



Sandy Bridge估计是明年上市。首发（希望不要交钱或者给什么芯片协会黑金）是2核和4核的。之后会慢慢的温水煮青蛙，8核和10核的都往外涌。。。如果一旦Intel把Steve Jobs拿下，NVDA的GPU出局。天可怜目前6B的NVDA。。。估计最后下场与AMD会差不多。

下图所示为Sand Bridge的Die图。与Westmere + GPU的双芯片（Die)解决方案，我们可以看见，GPU Seamlessly进来了。。。



Intel "Sandy Bridge" (SNB) / Mainstream Quad-Core

32 nm Process / ~225 mm² Die Size / 85W TDP / A0 Stepping / Tape Out : WW23'09

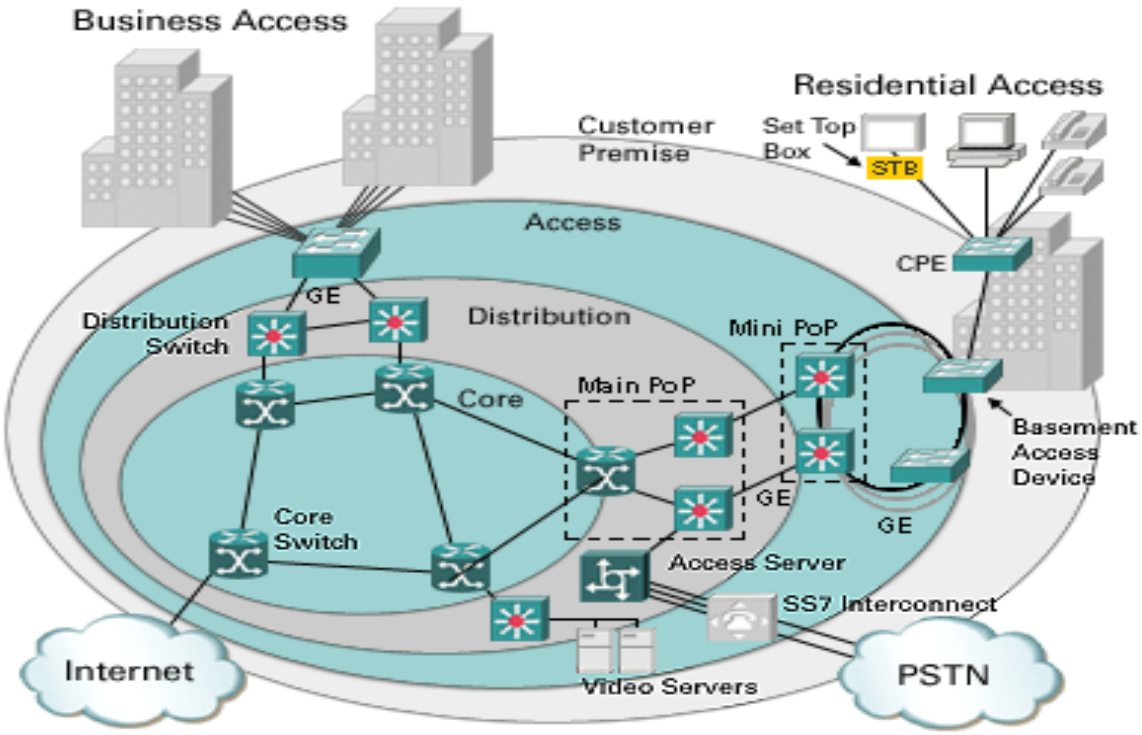
Expected : Q1'11 @ 3.0 - 3.8(T) GHz

彎曲評論
科技 · 人物 · 潮流

关于城域网的思考

（上）

作者：理克
totobeing@hotmail.com
编辑：陈怀临
huailin@tektalk.org



前言：

仍然是内外都忙，频繁出差，私事公事，身心俱疲，但也难以忘记曾经承诺给首席的关于ME的文章，因为最近偶然收到一个基层工程师关于城域网方案选型中why L3的议论，有一点点意思，所以想整理来借用一下，但直接发表显突兀，所以不得不罗嗦几句城域网的一些历史，因此有下面的文章，不想越写越多，感觉很罗嗦，很抱歉读者不能一下读到我刚才提到的文章，如果不喜欢读老太太说历史的内容，那么可以把精力放到对个人更有意义的事情上，作者非常支持读者

城域网目前已经成为承上（CORE）启下（access）的中心，所以成为运营商网络的建设核心，也因此成为IP网络技术的热点，在业务融合和TCO降低的驱动下的FMC更加速了城域网的重要地位，而POP设备的融合和service awareness的需求更增加了城域网的复杂性

城域网的概念大概在90年代才开始，是在IP发展开始普及的时候成为internet承载网络的必不可少的一部分，因为在没有IP之前的数据流量十分有限，并没有充分的业务需求，此时的数据业务要么限制在LAN，要么是低速的基于传统交换机或者DDN的X25/FR WAN网络，这成为制约人类文明发展的barrier，从二战到80年代，在以20世纪初为开始的科学理论和实践创新为基础，40年的经济和技术积累，使信息和知识得到极大的丰富，已经开出超出人脑手动能力的范围，计算机的发展从一个根本上解决的local的问题，但是如何让整个人类能快速的接受已经爆炸的信息，需要创新的工具作为知识传播的引擎，那么在终端计算机化的情况下，网络的IP和路由就成为人类文明发展的理所当然的物理engine，夸张点说像四大文明中的造纸术和印刷术。

所以LAN和低速的WAN必须要颠覆，此前的X25/FR对ACCESS/METRO/CORE的区分并不明显，从协议上来说，X25的的极端复杂当初是为了应付当时线路的不稳定，而线路质量得到改善后的FR，大幅简化但仍然比较繁琐，这都不是太适合城域网的需求，此时基于Ethernet的LAN是另外还一个极端：极其简单，但其广播的本质，低可靠性和管理性也是以传统运营商主建城域网不喜欢的地方，建立数据传输城域网在90年代末PC和internet爆发时高速发展的情况下，是必须的，但是并没有100% perfect的技术，所以21世纪初几年的城域网建设就出现了几种选择

1、 EOS：不是canon相机，是Ethernet over SDH/SONET，是MSTP（Multi-Service Transfer Platform，不是multiple spaning tree protocol）的主要组成部分，无论如何变化，基本就是在SDH上承载Ethernet，可以直接在SDH新建和升级扩容时支持，这是传统vendor极力推崇的方式，当然是利益问题，忽悠的要点无非是成本低，可靠性高，业务支

持强。当然事实并非完全如此，在MSTP下，有两种选择
（1）Ethernet L2/L3 Switch or Route rover MSTP，L3 Switch和路由器的主要区别是路由能力和IP业务能力，当时的IP其实没啥业务，一个internet，包括零售和转售，一个是企业专线，这里的区别无非是路由capacity和capability，尤其是指BGP、GRE、uRPF、组播、MPLS/L3VPN等，这里可能更多的还是在MPLS/L3VPN的支持能力上的区别更有意义。决定L2/3 SWITCH和路由器的选择的关键是TCO，显然，switch要比router便宜的多，便宜的原因有技术复杂度导致基于以太的L2/3芯片得以海量使用的原因，也有思科在路由器市场垄断能力更高的原因。当然，实际上，无论是选择switch还是router，当时全球来看，大部分还是思科的市场。但在交换机门槛较低的中国，华为、港湾的拼杀下的思科也很无奈。抛开政治恩怨不说，对于运营商，即使路由器城域网再好，但毕竟成本是不好难承受的，并且从HA和QOS来说，当时的switch和路由器也并无很大的本质区别，并且当时并没有电信级的语音和视频需求，不需要很高的HA和QOS，当时热门的视频会议，在中国还发展出基于传统SDH/E1的标准，历史上也是中华争抢自主知识产权的field之一，当然当时的视频会议更多的是基于WAN的，对城域的需求有限。说到OAM，可能路由器更复杂，至少对工程师的要求更高，这对运营商是必须要考虑的实际问题，所以综合考虑，在当时的ME，基本上没有路由器的事，L2/3 switch大行其道，当然，对于思科，也是乐见其成，受益匪浅，并且因此思科在传统L2的ME技术方案上达到了细致入微登峰造极的地步，思科的65系列交换机是其中最经典的产品，也许今日思科在ME产品的尴尬，也和65的过分成功有些干系.一些具体情况，在后面的文章中会略作细数

（2）　　MSTP：对于传统的vendor，在思科IP产品的垄断态势下，直接对抗效果不佳，所以MSTP成为中华风的天下，包括在标准上的努力和斗争，当然华为挟在SDH的强大份额，肯定是这个市场的最大受益者。MSTP通过把TDM/ATM/ETH三网合一，接入SDH，比如可以叫single SDH， multiple service来忽悠运营商，确实很好，这里面ETH的处理，就是EOS，MSTP设备把L2甚至可以做到L3（实际上好像基本没有人这样做，还是只做L2）集成到MSTP设备中，把所有业务传到POP点处理，这在当时的业务需求下，也没什么不好，为什么还要增加一层switch或者router呢？并且L2也可以做简单的本地交换，所以基于MSTP的城域网，至少是中国城域网的一个主流方案。增加SWITCH/ROUTE的方案似乎可以把POP能分布更边缘一些，可以忽悠future-proof，但还是面对MSTP的成本和现实优势，也不太好抵挡。但是MSTP对当时的港湾是很难受益的，肯定要打击这个方案，当港湾准备大力进入光网的时候，因为触动的任老板的核心利益，因此痛遭灭门，被彻底根除，一代通讯天才男哥，也因此流落到不停的寄人篱下的地方，令人扼腕
基于EOS的城域网，其实利用率是很低的，一个是GF的物理封装浪费带宽问题，比如一个100M的ETH，需要155M的VC进行封装，2.5G的SDH，只能支持2个GE封装，10G的SDH到现在为止都不便宜，在当时10G SDH成为北电的利润仓库的情况下，无论如何是不会很便宜的。另外一个是ATM和ETH是物理上独立的承载，没有multiplexing的，当然，在当时还没有成熟的TDM/ATM PW over MPLS/IP的情况下，也不可能multiplexing，所以整

体来看，基于MSTP的城域网成为主流也是历史的必然

2、 IP over WDM/fiber。毫无疑问，这是思科的始作俑，当然是基于自己地位和利益的考量，思科是清楚网络的发展趋势，所以根本不会去收购传统的光网络供应商，思科要的是near future、future和far future。无论从学术理论还是技术发展，IP over WDM无疑是正确的，但是无法让传统运营商放弃已经在光网上的投资，即使思科全部回购都未必能动摇运营商的决心，一个基层IP工程师的要求和薪水，和一个光网络工程师不是在一个level上，这对运营商即使拿到免费的IP网络都未必能承受的。这里主要的选择如下：

（1） IP over DWDM：城域的DWDM在当时是很难部署的，成本很高并且没有足够的业务买单，近几年才开始的城域DWDM还远没有普及，在当时是不可能的，所以思科虽然推出的当时IP+光的ONS15000系列，直到现在都没有市场，并且思科没有传统vendor的市场，所以这个方案既不符合传统运营商的利益，也没有自己的历史积累，所以对思科基本上没有成功的可能性

（2） IP over CWDM：这个可行性远高于DWDM，并且技术门槛也低，所以占领了一部分的城域市场，但大家都有份，思科并没有在这个上面获利很多，当时CWDM应该只有4个波，也就是只能有两个环，并且当时还没有10G的CWDM光模块，有也会很贵，所以CWDM的方案有一定的市场份额，但是并没有超出MSTP，毕竟MSTP可以说我的EOS是基于SDH的，有足够的可靠性，而CWDM+switch是没有这么高的可靠性的

（3） IP over （dark）fiber：不用说，IP独用，并且IP当时还不能做TDM/ATM仿真，传统运营商没法用，除非光纤足够丰富，但毕竟铺光纤也好，租光纤也会，都是要花不少银子的，所以这个方案也是很稀少的应用

现在看来是城域WDM化，TDM/ATM/ETH over MPLS/IP，N网合一，但是即使现在，许多运营商的方案还是让TDM/ATM自然消亡的migration方案，而不是立刻就迁移到IP，运营商有自己实际的考虑，从人力的能力到资源折旧的综合TCO，不是纯技术决策市场，需要smooth

总结一下这个阶段的城域网，比如大体可以在1998-2003年，主要是基于MSTP的EOS的城域网，主力产品是MSTP的SDH产品和L3交换机，受益厂家主要思科的交换机路由器产品，和以华为/AL为首的MSTP产品

ATM插曲

上面的大段关于城域网的论述，居然没有提到曾经试图辉煌到甚嚣尘上的ATM，是不应该的，所以下面也谈一下ATM的

在我大学毕业前，ATM就是现在的MPSL，everything over ATM是最fashion的数通网络技术，从核心网到城域网到desktop，基于ATM的网卡都有，你说得多贵？我当时刚搞懂PC机的基本装机和基本C语言的DOS编程，对计算机世界上业界大拿的忽悠文章，只能剪下来收藏，以便以后能读懂的时候再拿出来学习，因为这个报纸和其重量比，基本是免费的，估计卖废纸的价格能很大程度上能弥补购买的成本，当然，我并没有试过，后来这一大堆剪报不知道在哪次搬家中丢掉了，即使没有丢，再拿出来，也许看到的是：早就不流行

了，现在哪还有人穿呀。所以当时最喜欢的还是上面的IT普及教育级别的文章，ATM是其中大体不懂而被收藏的topic之一，因为工作关系后来接触到ATM技术，所以现在仍然记得当时ATM剪报的事

上面的大段关于城域网的论述，居然没有提到曾经试图辉煌到甚嚣尘上的ATM，是不应该的，所以下面也谈一下ATM。但是ATM已经是昨日黄花，IT是吃青春饭的，ATM已是门前冷落鞍马稀，估计现在的80/90后没人喜欢看这青年才俊眼里已经可以算是老掉牙的老太太了，世态炎凉，人情冷暖，尽在IT人生，所以本篇内容，不得不在技术描述中加点人生调料，妄图借一点小资情调打开一下人老珠黄的尴尬场面。此正是：

人老尽靠粉扑面，唏嘘长叹谁见？强作欢颜为接客，遥想当年perfect，扼腕！

在我大学毕业前，ATM就是现在的MPSL，everything over ATM是最fashion的数通网络技术，从核心网到城域网到desktop，基于ATM的网卡都有，你说得多贵？我当时刚搞懂PC机的基本装机和基本C语言的DOS编程，对计算机世界上业界大拿的忽悠文章，只能剪下来收藏，以便以后能读懂的时候再拿出来学习，因为这个报纸和其重量比，基本是免费的，估计卖废纸的价格能很大程度上能弥补购买的成本，当然，我并没有试过，后来这一大堆剪报不知道在哪次搬家中丢掉了，即使没有丢，再拿出来，也许看到的是：早就不流行了，现在哪还有人穿呀。所以当时最喜欢的还是上面的IT普及教育级别的文章，ATM是其中大体不懂而被收藏的topic之一，因为工作关系后来接触到ATM技术，所以现在仍然记得当时ATM剪报的事

在ATM之前，IP的承载网络有两大challenges：一个是WAN的速度和业务不能兼顾的问题，以E1/E3为主的X25/FR显然是低速并且复杂的协议，而STM-1/4的颗粒比较大，并且基于TDM的，不能提供业务层面的复用，也没有精细的业务QOS的保证；另外一个是EHT LAN，ETH是LAN王，但是没有HA/OAM/QOS的ETH是不会被运营商广泛用用到WAN上的，在城域网的应用主要是来做internet，提供低成本高带宽复用的能力，和ATM是不一样的。

ATM的目标定位是比较清楚的：提供高速的single ATM network，multiple services supporting，在业务上，ATM精细化的定义了CBR/rt-VBR//nrt-VBR/UBR等多种业务管道，同时对每种管道，又提供多种AAL，主要是AAL1/2/5，其中AAL1和2在实际应用中区别并不明显，所以主要就是CELL（AAL1/2）管道和FRAME(AAL5)管道，一般来讲分别对应TDM和数据类的业务。同时ATM定义的很好的OAM机制，包括F4/F5，一般来讲，基于VC的F5（不是负载分担设备的F5）比基于VP的F4更多一些。以上只是对ATM的概括介绍，但管中窥豹也能看到ATM是一个多么perfect的协议，所以大家把它应用到anywhere，everything，包括desktop，在当时似乎也无可厚非，并且也确实有部分实际的应用。为此，ATM定义了很多IWF，包括ATM-ETH, ATM-FR(FR over ATM)等等

现在大家都急着把ATM phase out，尤其是北电在意料中的倒下更是加速了大家抛弃ATM的信心和速度，在商业社会里，资本是冷血无情的代名词，不要跟他们谈感情，那只会受伤更深。但是在当时，进入21世纪的时候，DSLAM、3G和NGN(soft switch/carrier VOIP)开始emerging的时候，并没有好的承载网络可以选择，要么速度低，要么缺少

service-awareness QoS，为什么一定要HA和QOS，因为DSLAM/VoIP、3G里面的语音问题，在今天看来，语音那么一点点流量，给他一点足够的带宽不久够了吗？但是这是后生可畏的想法，在当时，甚至有立法要求必须保留传统语音交换作为VoIP的备份，并且当时虽然大家看到数据会成为主要的revenue，但是毕竟语音还是现实的revenue，语音业务是基础，怎敢随便用一个网络承载语音，所以最早的DSLAM是ATM DSLAM，3G的R4之前的版本语音和数据都是ATM，并且当时业界有ATM和router之争，大家当然会互相PK，忽悠客户，当时大体的结果是对于运营商的综合业务网络，基本都建立了ATM network，而数据网络，有实力的运营商会选择建立以太网络，包括EOS

从上面的performance看，北电选择ATM，抛弃路由器，也在情理之中，至少在revenue上还是赚了很多的。但是毕竟ATM已经过气了，所以ATM肯定是有问题的，从技术的观点看，ATM的完美和IP的简单是冲突的，这是违反IP网络的本质的，而IP是未来是毫无疑问的，所以ATM出局也是毫无疑问的。具体的表现在ATM的完美使其不得不复杂，复杂的结果就是严重影响了速度的提升，本质上是高速的成本太高，IP是属于平民化草根阶层的，ATM有点贵族气质了，草根喜欢大口喝肉，大口吃酒，这不是ATM能够忍受的人生，大家道不同，不足以谋，分手既然已是必然，又何必问佛诸多缘由，所以我们就此打住，顺其自然乃我道家真谛，强求合欢，快亦不快

对于起步阶段的中国厂商，不用说在ATM和IP之争的话语权，其中的权衡，连美洲的资深巨鳄都不能看清楚，我们也只能妄作分析，所以此时的国内厂商是两条腿走路，虽然不能把获取赌单的高额回报，但至少能获得生存资本，对于强权下的弱者来说，自然要有弱者的哲学，有毕其功于一役的时候，也要有compromising的时候，不可能全部都猛扑上去，也不可能总是compromise。但是大体上，在中国，似乎是ATM略占上风，长远来看，这是个错误的重点

ATM就先说到这里，在后面的FMC中还会有相当的篇幅考虑ATM迁移的问题，这个不得不处理的累赘也是运营商身上的肉，怎么割能尽量省钱还少伤筋动骨，不是小事

ATM本身是和X25/FR一样独立建网的，所以绝大多数运营商都有ATM城域网，后面城域网还要考虑ATM和MPLS比较等问题，但在这里先暂告一段落，因为笔者对ATM城域网建设的具体方案不是很熟，并且已经不fashion了，所以也就不去细追究了，有感兴趣的读者和补充相关材料

下面在正式隆重推出AL的新ME方案前，要拿一章介绍一些非主流的的ME技术和以太网在AL ME的发展，为AL的主流新ME清理一下门户。

大片之前的加片

小时候看电影的时候，是露天电影，但露天和加片无关，全国人民在看大片前都要加片，具体内容忘了，可能有新闻或者教育普及之类的，有时候，最恐怖的是加片比大片还长:)

在正式介绍AL新ME之前，我们先多历史总结一下，在第一章里，忘记了一个最一种广泛使用的城域网方案，看了一下印度鬼子视频的开头才想起来：SDH的城域网，除了支持传统的TDM业务，其实主要提供的是business业务，包括企业专线和转售，具体方式可以是直接的E2E SDH管道，也可以是在城域用SDH，在CORE用IP，下面简单罗列一下需求和技术

基本需求：

语音：需要好的QOS和HA，可以是TDM的SDH，或者后来支持TDM的ATM

数据：需要低成本的大带宽，那么ETH最好

那么再看一下基本技术：

SDH：提供的是没有弹性的硬管道，天生就是要支持语音业务没商量，但是数据业务时要复用的，这个SDH搞不定，成本好差和一个数量级

MSTP：从解决TDM/ATM/IP共传输上看，很好，这里的IP，如果单纯的提供EOS，而没有任何L2/L3的收敛，就和SDH没有区别了，甚至比SDH还差，所以一定要提供L2/L3功能，无论是内置到MSTP设备还是增加独立的SWITCH/ROUTER。MSTP是对SDH的成功优化和拓展。

ATM：ATM本身是基于数据的，能够提供复用，目的是同时完美承载TDM和IP，实际结果是都做了，但是都没有做完全。TDM并没有完全到ATM上，基于历史的legacy，只是新的语音业务NGN/3G切过来了，而原来的PBX和2G并没有全部且过来，许多运营商采取的是自然消亡到一定程度，还没死再切换的方式，比如VOIP就是随着ADSL来切换，ADSL足够多的时候，就全部切到NGN，2G切换的很少，无论如何，ATM支持语音业务是成功的；看数据业务，ATM不是那么成功，具体分析见上一章，对于固网的数据，更差一些，对于3G R4之前移动数据，应该说马马虎虎算是过得去

那么增长最迅速的是数据业务，21世纪初是ADSL泛滥，迅速淘汰窄带拨号的时期，华为在以超大容量超低成本的窄带接入几乎在一夜之间打垮思科在窄带接入市场后，还没转过神来，还试图推出宽窄一体的接入产品来过度一下的时候，ADSL以迅雷不及掩耳盗铃之势立马淘汰了这个产品，从成功到失败，都不到一步的距离，战场不相信眼泪，所以也来不及擦眼泪，就迅速投入到ADSL的广阔天地的斗争中去了。数据流量的迅猛发展，导致运营商必须想办法解决带宽问题，同时也不能一点QOS和HA不考虑，所以这个时期的掀起了城域网建设的高潮，但主要技术还是基于MSTP的EOS或者单独的传统以太城域网

DSLAM直到现在都是整个网络接入部分的核心，美其名曰MSAN(multiple service access node，大家可能是太喜欢微软了，所以什么都想MS)，应该有很多故事，笔者不是做这部分的，了解有限，在后面的seamless MPLS会多说几句。在这里，

我们可以提一下ATM DSLAM和IP DSLAM之争，很明显，这是男哥IP领域三大英明决策（全系列以太网L2/L3交换机、IP DSLAM和10G SWITCH/ROUTE）之一，前二者硕果甚丰，让华为苦不堪言，而最后一个，在开始让华为难受的时候，港湾折了，弹指一挥，5年美好时光烟消云散，一切都不复存在，只在记忆里。男哥英明，IP DSLAM的标准，好像在ATM DSLAM热了一年后就出来了，并迅速成为主流，但是这里，ATM网络这个绊脚石，在不少程度上阻碍了IP DSLAM在某些没有合适的城域网而有不错的ATM网络的国家和运营商的快速发展，导致的恶果是今天某些运营商在做ATM DLSAM迁移的时候有点痛苦。华为自从没了男哥，而宝哥重病，再无CTO，实践证明最强的小徐并无力替代男哥和宝哥，任老板能在多年没有CTO的情况下仍然领导华为披荆斩棘，不得不佩服任老板的彪悍和雄才大略，在让人对任老板离去后的华为担心的同时，也多了一层放心，华为即使没有任/李/郑的大英雄，也还是有很多能人支撑华为的大厦
因为这个时候MPLS TE还不够成熟，那么如何在现有的ETH技术上解决以太网的QOS/HA/OAM/SCALE问题，各路神仙开始各显身手，此期间主要是在HA上投入，然后是scale，最后是QOS,OAM基本上没大进展，后面将逐一介绍这些技术，这部分的时间，主要在2000-2003年。

一些湾友做了很好的comments，也说明我的一些文字是妄测。当然也说明一个运营商在选择方案的时候，是和自己的实际实力地位、各种资源现状和短期/长期的目标等实际商业因素紧密结合的，而不仅仅是技术的，POS和IP over 独立光纤虽然昂贵，但对incumbent的运营商，也未必不是一个有利因素。当然这是从这个具有较大垄断地位的运营商来说，但是对整个社会来说，不使用最好TCO的技术，不最充分的利用资源，就是浪费，浪费就是犯罪，相对充分的自由竞争可以通过优胜劣汰在来利用丛林法则解决这个问题，但是当难以形成这种竞争的情况下，就需要法律，对于电信业就是电信法。中国和欧洲几乎同时开始在90年代开始电信法的起草，可见中国人不乏聪明，但是欧洲的电信法已经实施好几年了，中国的电信法似乎连讨论都开始减少了，比较一下中欧的电信资费，虽然绝对值上欧洲要高几倍，但是在和收入相比的相对值上，是反过来的几倍，中国电信行业的利润，虽然没有惠及到普遍的所有为电信工作的人，但是肯定惠及了不少不少食糜者，至于效率问题，没有研究过，不好妄论，理论上应该要低一些，但中国的的一些事，不是那么简单可以用理论决断的
因为首席刚刚发布了股票信息，可能有刺激，其实兴趣虽然很重要，但是对于草根百姓，更多的还是喜欢直接的金钱刺激快感和动力劲道更大一些。
闲言碎语切莫讲，今天表一表武二郎，啷哩了啷啷哩了啷啷哩了啷哩了啷哩了啷

…（以此开场啷表示对以前大学时同寝的一位山东大哥的回忆），言归正传，接上回书，总结一下以太网，我们可以把原始的以太网叫native Ethernet，的主要问题:HA/SCALE/QOS/OAM，HA是永恒的主题之一，对电信运营商，是一个永恒的忽悠主题，IP本身是没有HA的，IP的HA是交给终端处理，本身尽量少掺和，这是IP的精神支柱之一，不管你喜欢不喜欢，这似乎IP的性格，性格问题很麻烦，你不顺着它，它就不让你爽，但是IP本身不提供HA，不意味着我们建网是就不考虑，尤其是对运营商，恰恰相反，越是IP不care，网络越是要care，IP不做，那么我们ETH做，当然不是心血来潮，非要和IP对着干，还是因为语音的问题，上一回说到IP DSLAM和NGN，因为IP DSLAM成了主流，NGN替代了传统语音，所以城域以太外网就不得不考虑语音的HA问题，HA的问题的本质，笔者在之前的文章中有论述，这里既不再重复，总之为了语音，也为了增强运营商使用城域以太做综合业务承载的信心，HA是首当其冲要开刀祭旗的问题，那么就从HA说起，在没有新ME前，ETH的可靠性技术主要如下

1、 STP/RSTP/MSTP:

这是ETH解决L2网络环路的基本协议，基于广播处理，RSTP做了加速收敛，MSTP做了多实例，解决支持VLAN后的多spanning trees的问题，但江山易改，本性难移，其收敛性能总是要几秒，TDM/SDH因为是简单的协议，基于硬件的处理，是50ms level，称为电信级可靠性，所以秒级是不可接受的。这个领域还是传统的ME技术，是思科的强项，思科把STP技术也挖掘到了极处，比如PVSTP+等，真是费尽心思，这是无论如何，因为STP出身简陋，纵使思科大鳄想回天，也难

2、 DPT/RPR

又是思科的始作俑者，思科开始做DPT，后来才有RPR的标准，啥目的，师夷长技以制夷，吸取SDH RING的精华，用在传统以太上，从L1/2层面解决ETH的HA问题，但是有一些基本原则经常是你和它越像，你的成本也和它越像，没有如果没有本质不同的breakthrough，就难以做到同样功能，不同本钱，比如你比较C/H/J/A等同等档次的产品的成本，他们大体相差是很有限的，所以DPT/RPR的昂贵是不可避免的，尤其是在没有大量部署的情况下。如果相对ETH，SDH算小贵族，那DPT/RPR就可以有SDH贵族气质了，这和ETH的草根廉价风格道不同，这种情况下还要结生死夫妻，别说上床充分融合，一般是还没等拉紧手，就悲剧分手了，只是害了一些喜欢吃螃蟹并为其结合投入了真币的客户，还好不多。说这么多带颜色的话，是因为我对这个技术的细节了解非常有限。基本原理是和SDH类似的50ms的故障检测和恢复，但除了POS/native fiber接口，还增加了GE/1OGE的支持，似乎没人用FE，毕竟100M的环，如果在90年代还敢拿出来fashion show一下，在21世纪就太老土了。DPT/RPR技术复杂是一方面，同时需要占用两根光线

和两个物理接口，在当时的10G，这真的是有点贵族的奢侈了，对于有大规模SDH/WDM网络的incumbent运营商来说，对这个小姐不感兴趣，所以主要是少数的当地tier2/3的运营商用了一点。思科试图通过这个方案来吃城域传输IP化的市场策略，基本是失败的，当然对思科也无所谓，这在思科的大象里，比汗毛大一点而已（如果大象有汗毛的话）

3、 EAPS/RRPP

基于RPR的失败，最直接和简单的idea就是做一个lite版的RPR，性能在几百毫秒。这里忘了思科是如何处理的，按说应该是参与的，具体没有去考证。但是似乎RPR的惨败严重打击了高可靠eth ring的相关标准组织，以致大家没有兴趣再做成一个统一的标准，基本原理一样，但具体实现大家就无法完全compatible了，连思科似乎也不是很感兴趣，不知道为什么。如果RPR是大奶，大奶分手，并没有给二奶留下足够的空间，缩水版的RPR也让然是第二根鸡肋，和RPR一样的市场情况下，也逐渐无疾而终了，好像理由主要是compatibility问题，但总觉得这个理由怪怪的，同时奇怪的是没人再愿意去深究这里的问题了，有谁对一个弃妇感兴趣呢？毕竟变态的人很少。如果是先走EAPS/RRPP，也许真的有成功的可能，甚至之后会有RPR，maybe，who knows。围棋里同一步棋的先后次序不同，以及是一次把变化定型还是保留变化，微妙的变化都可能带来巨大的差异

对于IP下的ETH RING技术，还有两个技术问题：

1、 现实的环之间的关系比三角还要复杂，要解决环的相切相交等问题，三角类的关系烦恼总是大于幸福

2、 业务支持问题：那个时候在IP上的业务越来越多，大的组播了，TE了，还有许多细小的特性一时数不上来了，在普通接口上OK的，到了这里都要重新搞一下，不搞就出问题，当然搞了之后，有时也出问题

在eth ring上，思科无疑是最大的始作俑者和支持者，但思科总想做私有协议的心态，时常让客户不是那么爽，以致现在很多标书至少要让供应商详细说明标准程度，甚至直接有不允许使用私有标准的说明。思科想通过提供和传输类似的HA低成本ETH网络来扩大城域IP吃掉传输的思路，从方向是应该是没有问题的，但问题出在具体的技术方案上，当然，实际上原因也许比这些更复杂，outdated了，who caresEHT RING对ETH的QOS和scalability都是trouble，trouble就不提也罢

AL ME大片前以太网的总结

我刚刚兑现了首席发表股票总结前的承诺，这里首席虽然看好ME，但似乎并不火，一点

失望之余，也反思可能是文章内容太过老旧，基本是事后诸葛，没啥意思，或者阿Q一下这里读者对运营商网络感兴趣或者说懂得多的人不多，那就写点演义风格，聊以滥竽充数吧。
大牌出场前总是要犹抱琵琶半遮面，调一下现场的情绪，胃口调太高了也意味着风险，精彩固然是好评如潮，而一旦这个大牌是冒牌货，也可能有audience要忍不住砸场子，所以聪明的玩火者需要把握好玩的尺度。也许让大牌直接在出场前出师未捷身先死也是一个不错的选择，本想收笔，但事情既已开始，半途而废不像是男人该干的，所以还是干完，砖头也算石头，和玉同属一个大家族。
遥想当年ETH在LAN火的时候，至少有4种以上的协议栈格式，可见大家都是基于在以太的大头里塞点私货，干点好事，甚至IBM还超前的做了FDDI的ETH RING标准，真是业界大牛的风格。经过一番并不激烈的过程，到2000年，最后常用的是ETHERNET II和SNAP两种封装，然后很快就基本上是Ethernet II为主了，大家的注意力迅速转向了802.1Q为以太网带来的VLAN时代
以太网无疑是和IP一样伟大的发明，廉价的草根性质，虽有自由散漫的本性，但在LAN里无伤大雅，所以是相得益彰，如鱼得水，如此一来，以太网的势力范围就不断迅速扩张，这对以简单广播技术为基础的以太网可不是好事，所以VLAN很快被用来做虚拟或者说逻辑隔离以太的broadcast domain，并提供一定安全性的concept，在当时也有几种VLAN的可能，包括为了老旧的IPX还做了基于协议的VLAN等等，当然，正统的基于802.1Q TAG的VLAN很快统一了VLAN的天下，这小小的4个字节定义，总体来说还是比较成功的
但是也许1Q的定义者浸淫在LAN太久了，LAN对于WAN就是一口小井，在井里太久了，视觉范围就会受到限制，居然给了VLAN只有4096个ID，还要掐头去尾，这里如果ETH永远在LAN，也许确实不是什么大问题，可以别太看扁了自己孩子的潜力，当时的以太还是16/7岁的孩子，当大家认为ETH是MAN的最好接班人的时候，回头发现这不争气的大人，居然只给孩子4096个ID，真是没见过大世面，不知道一个metropolis最小也是M级，你一个K级的人怎么能撑得住。所以不甘寂寞的供应商很快就搞出来QINQ，以致QINQ的标记都不统一，有8900，9100等，对两层Q的一些具体关系，也没有标准定义，虽然带来了一些不必要的麻烦，好在比较简单，对VLAN没有造成什么大的伤害。如果当初直接定义两层Q，在加一个optional的三层Q，也许现在看来就完美了
总之VLAN和QINQ给ETH带来了巨大的活力和魅力，使得不少运营商使用QINQ的做ME的aggregation，一直到现在，虽然级别可能在向ACCESS降，但一直在主流的技术方案里
VLAN/QINQ主要给scalability问题提供了一个还不错的阶段性解决方案，并提供了eth QOS的基础，但对HA和OAM，基本没有解决
这里几乎同时出现的还有TRUNK技术，是解决可靠性和带宽的不错的技术，技术本身很简单，就是几个以太口捆成一个用，即使再跑上一个802.1AD（LACP）协议，也不复杂，当然，TRUNK本身固有的HASH可能不均和QOS问题，虽然基本上难以从技术上解决，但一般也不是什么致命问题，所以TRUNK技术也一直是以太网络的基本要求
总之，internent需要海量的市场，这里草根哲学的廉价ETH成了网络扩展的不二选择，从

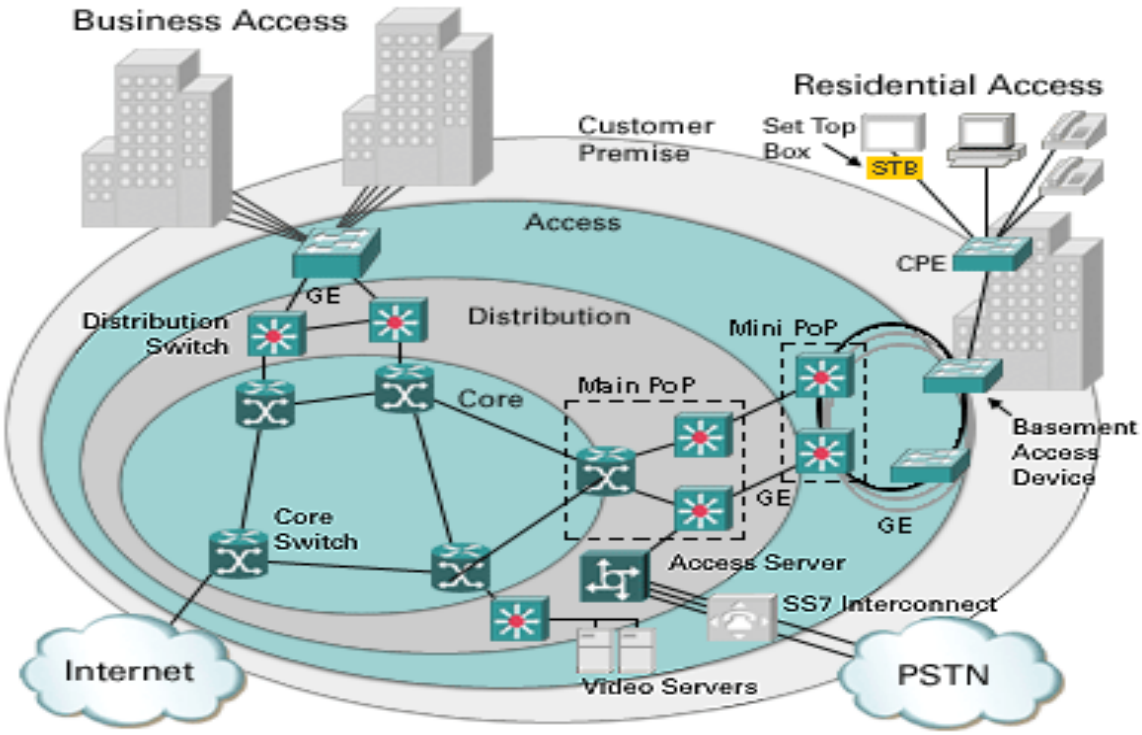
LAN到MAN到national CORE，POS节省的一点点字节头，很快被以多媒体为主的1500字节以上的大包淹没为忽略不计，POS彻底让位给以太，成为历史上的太上皇，将来可见的承载网络世界，就是IP+ETH+OPTICAL，三个最廉价的技术简单加在一起，成为日益复杂的internet海量内容的bearer 这篇文章有点水，阑尾了，可能是VLAN/QINQ虽然及其重要，可是相对其他许多技术，确实很简单，又简单又好虽然好，可说不出花来，所以迅速结束。

彎曲評論
科技 · 人物 · 潮流

关于城域网的思考

(中)

作者：理客
totobeing@hotmail.com
编辑：陈怀临
huailin@tektalk.org



ALU新ME的前世今生

这个系列的开头说了是为一个讨论why L3的同事的邮件而起，后来总是太ALU的新ME，似乎新ME成了主角，也只好强赶鸭子上架，所以这只烤鸭很可能是只肯德鸡，而不是前门的全聚德

VLAN/QINQ虽然很强，但还是不能是以太网拍拖传统ME方案，比如scalability问题，MAC容量就很麻烦，以致大家在实践中要求在一些场景下，比如傻瓜型管道业务，不要学习MAC，直接根据VLAN转发，以彻底根除MAC学习和容量带来的麻烦。其他的安全/QOS/OAM/NSM等大规模MAN网络需要的技术，传统ME还是没有很好的解决方案，时代呼唤英雄，思科在城域网络上的76/65的传统L3/L2方案固步自封，华为跟班思科还难以成器，AL在欧洲挺身而出，揭竿而起，开创了ME的新时代

AL能成功的推出以7750为中心的新ME也是一蹴而就的，早在2000年，AL就成和华为一起使用IBM的当时业界最好的NP芯片ranier做IP产品，据说当时华为迅速推出NE40/80系列，成功的和思科拉到最近距离，而AL似乎不太成功。但是后面的故事，确是又反了过来，AL的IP部门在成功收购了谷里的NP公司timtra后，即薄厚发，一举推出系列产品配合新ME方案，从欧洲市场开始，在ME领域开始势如破竹，而华为，在和港湾的PK惨胜中，把IP产品的错误持续到10G，本来就是follow策略，在全力纠正10G的错误的时候，即使没有这么多据都错误都未必创造新ME方案，何况当时，还哪有心情去考虑新ME，当AL的新ME 7750出来的时候，有些人立刻又蒙了。

在细说AL新ME的特点前，不得不说说思科老ME产品和方案的特点：

在没有AL7750新ME前，在传统IP城域网方案里，C76/65真的是最完美的孪生兄弟，同时在IDC和企业网市场也是霸主，太牛了，整个产品既有集中式转发，也有分布式转发，还交融许多75系列的接口卡，L2性能很好，L3/MPLS也支持，各种核心引擎可供选择和升级，真个一个万花筒，那是革命一块砖，哪里需要哪里搬，美C76/65的最美好的青春持续超过了5年，知道AL7750青年才俊重磅出山，迅速在

ME市场扩容，然后华为NE40E系列的不断发力，C76/65终于被年老珠黄，满身雀斑，暴露无遗

1、 Packet based的交换网，还容量小

2、 集中式的L3/MPLS转发引擎，性能和功能都老落伍了

3、 升级新的主流特性就得换板，比如L2到L3，不能软件搞定

4、 组播能力也差，不能做大量的IPTV

5、 C65/76同门不同价

6、 TDM/ADM等FMC特性支持差

7、 路由表容量/MAC表容量/TE容量/LSP容量/PW容量/VPLS容量等都和对手不再一个档次

….

这样一个烂柯，居然成为选美冠军近10年，是思科垄断的好，还是看客眼光拙，还是其他美女太不争气。不是金子总要收光的，美女也要夕阳红，除非天山童姥

终于爬到正题，因为这个中很长，所以至少要分3章，为了保持章节内部的连续性，这章的切割就短了些

AL7750新ME是一个从marketing到R&D都非常成功的方案和产品，在2003年左右，使用以太建设城域网已经基本得到确认，那么市场的实际需求和相应的思科基于76的产品和方案有什么问题呢？

1、 VOIP/IPTV需要HA，而C76老ME难以支持，L2不成，L3当时也没有很好的快速收敛技术

2、 商业用户企业网：在L3VPN为主的IP企业专线

3、 HSI其实一直都不是大问题，因为运营商从来都不为internet提供QOS和HA，如果你仔细观察ADSL的合同，运营商提供PIR峰值速率，但没有承诺任何CIR承诺速率，更不用说SLA，虽然大多数情况下我们上网至少都能达到几百K，但是运营商从来都不会做这个承诺，因为怕万一大不大，索赔是个麻烦事。不断对终端客户，就是运营山之间的IP连接，互相也是不做SLA承诺的

可见在城域网逐渐成为运营商各种网络的核心后，电信业务和商业业务在向这个网络迅速迁移，这二者是互相利用和促进，那么老ME具体有什么问题，再总结一下

1、 HA不行

2、 QOS不行

3、 OAM不行

4、 NSM不行

5、 Scalability不行

6、 L2VPN能力有限

这主要是针对L2的城域网的，那么上L3是不是可以解决上面的问题呢？可以部分的解决，但有下面的问题：

1、 HA有限：因为传统的IP/MPLS收敛至少是秒级

2、 QOS可以算解决

3、 OAM：MPLS L3VPN的IGP/MPLS/BGP带来的运维问题，是客户认为都是L3造成的恶果

4、 NSM：思科没有运营商的网管，是企业家的，C一直不愿意在网管上加大投入，导致基于SNMP的IP网管和IP协议的蓬勃发展太不和谐，和其他SDH等网管相差甚远，听这个名字simple network management protocol，IP这么复杂的网络，居然用一个简单网络管理协议，能和谐吗？当然IP过于复杂和变化是一个原因，但思科基于政治考虑不愿意投入也很可能是一个原因，因为思科有很好的sustainable的培训体系，不用自己花钱，并且还赚钱，为什么不用CLI，投入一个GUI的很好的NSM对思科有什么好处呢？但是这对其他供应商不一样，因为他们没有思科培训这个超强的武器，在思科的垄断下，客户有苦也没有办法，在IP的SNMP严重滞后的情况下，做业务级的网管必须要和主流厂商合作，这里最主流的是思科，思科不想玩，别人就都是太监。这里对OAM/NSM说很多，因为在发达国家，这是个大投入

5、 Scalability：L3VPN是基于BGP的，扩展性很麻烦，还要RR，用L3VPN做企业网其实是不合适，企业的路由应该企业自己感知为主，运营商需要掺和的理由有限，所以L2VPN(VLL/VPLS)应该是企业VPN的主流，要替代的L3VPN的大部分市场，而此时C76老ME在这方面的能力很有限

从以上分析可见时代需要新ME，那么在看新ME的技术基础是否已经具备：

1、 MPLS TE开始商用，通过TE FRR/HOT STANDBY，可以提供类电信级的HA

2、 L2VPN(VLL/VPLS)开始商用，可以提供更好的企业网专线

3、 POP点的BRAS还是主流，基于PPPOE为主的链接，使用L2VPN很自然

4、 10G的ASIC和NP开始商用，解决MPLS TE/L2VPN下的基本功能和性能

以上是可以利用的条件，那么没有条件的就要创造条件，比如NSM/ISSU/NSR

我们来看ALU具体怎么做的：

1、 硬件核心chipset和收购

需要一套performance、flexibility、scalability等都不错的forwarding engine chip set， 只有美国有这样的公司，ALU成功的收购了timetra，基于其FP1芯片快速推出了7750产品，在这个时候，业界的商用NP都没有这个能力，IBM的Ranier是一款非常好的2.5G能力的NP，但

是因为主流的IP核心产品供应商，除了CISCO，就是老二juniper，大家都不主力使用别人的NP，导致这部分的ROI很差，作为IBM芯片中的边缘产品，尽管技术非常成功，成功到华为使用rainier开发的NE40/80系列产品几乎接近了CISCO当时的旗舰12K，而12K用的是ASIC，所以华为市场成功的包装出第五代路由器，用NP淘汰ASIC。但是路由器不是华为的主力产品，华为也无法快速做到全球30%的市场份额，这不是技术问题，技术再先进，销售平台也要一天一天建，但是IBM不可能把NP的命运压在华为身上，去开发10G的NP，所以只有夭折掉rainier，卖给了一家叫hifen的公司，是否可以卖给华为？即使美国政治允许，以当时华为自己的芯片能力，能否保留住核心团队，迅速消化后开发出有竞争力的10G NP，很难说，从理论和技术上说可以，但是任老板是否愿意做这个投入，虽然不一定是10亿美金级别，那也是一亿美金的级别，当然，任老板和柳传志不同，任从来就不是一个不敢下注的人，所以我猜，而老板当时把宝压在了无线和3G，所以在没有公司系统完整的一套策略支撑下，是不能单独只靠收购一家芯片商就能获得路由器的大幅成功的，打仗不是这样打的，奇兵可以，但是没有整体配合的奇兵，不能取得最后的胜利，看韩国棋手李昌镐的棋，奇兵不是很多，偶然有，但李昌镐的绝大多数胜利，是建立在整体的稳定表现上，尤其是近乎完美的收官，李昌镐鼎盛时期的失败，有对手发挥太好，一直保持优势的，也有偶然自己收官不美而失利的。所以华为路由器在NE40/80的昙花一现，是有其历史原因的，如果当时老板把宝都压在路由器上，那么也可能成功，但是如何衡量是压在无线3G成功的概率更大还是压在DATACOM上成功更多，事后诸葛的看，还是无线3G更高，一个是这是华为在运营商的传统势力的优势空间，另外就是，在路由器思科垄断的市场，似乎比无线3G更容易成功，但感性的看，这是真的吗？华为真的可以干过思科吗？如果没有2002年那场不以人们意志为转移的官司，华为路由器真的在美国开拓了哪怕不大市场，从而带来全球市场的迅速增长，也许老板真的会从策略上把路由器作为和无线3G更接近的投入。但历史不是这样演的，2002年冬季的那场雪，被任老板率精英精心策划，并痛割一半数通给3COM外加市场禁令的沉重代价而化解，貌似胜利，其实是思科的成功，以致后来再想回收3COM的时候，可爱的山姆大叔死活不批，理由虽然是tipping point的问题，其实是欲加之罪，何患无辞，美国人同样不乏中国人创造困难的智慧，没有理由，创造理由也要禁止。但塞翁失马，很难说对华为的影响是好是坏，当然对dadacom肯定是坏是毋庸置疑的，看到今年美国把丰田像当年整鬼子汇率问题一样的往死里整，不难想象，如果没有当初的官司，而事情爆发到今年，美国人会高看中国人一样，放华为一马吗？我不相信，我个人的偏见，对美国人，远比对欧洲人没有好感，因为就我承认的人种论因素，美国人是欧洲的流氓传下来的，根就不好，包括宠物在内贵族似乎需要纯种，所以欧洲好不容易积累下来的贵族气质，在美国几乎荡然无存，但杂种有竞争的优势，有冒险和创新的贡献，也有劣根性的难变，很抱歉扯到了政治，可见我对美国人的偏见之深，世界最闪亮的明星大哥，需要以最苛刻的条件去要求和衡量，道德上，也许并不是那么过分。中国做老大的时候，虽然也打，但对归顺朝贡的小兄弟们，送出的回报远大于那些贡品，而小兄弟们只要认下大哥的称谓就可以了，没有任何物质文化损失，以致这种传统到中国贫困交加的时候仍然坚持，可见中华民族是一个多么勇敢善良文明的民族；现在的美国当大哥，从物质到文化都要搜刮跟班的小兄弟，不管你是有

钱银还是没钱银，刮你没商量，你武装到牙齿，军力全球过半，首富一大堆，财富流油，给受苦受难的小弟们分点汤喝咋就那么难呢？吃一点亏都要数十倍的报复回来，小米粒大的心胸老葛朗台等四大吝啬鬼也会觉得自己冤枉，做大哥的道理，我看美国人还是应该好好和中国人学学，教训中国人，美国人还没有道德上的资格，小道消息：华为也曾试图收购timetra，但是美国大哥没批准，直到现在，华为公司还动用各界游说关系，在向世界上最民主自由的美丽的国家证明老板是退役军人，但是公司其他员工是普通员工，不是妄想要解放全世界全人类包括美国鬼子的解放军。10G的NP，当时可能最好的INTELDE IXP2800，不好意思，终于把话头调回来了，不容易。这个东西性能差，配套芯片贵，也不够稳定，但其实都不是最大的问题，最大的问题是，INTEL抛弃了IXP2800，不再继续升级投入了，对于一个要爬起来的人，如果要继续使用IXP2800做产品演进，如果没有准备好其替换策略，这一棒是很恐怖的。中国人要把小命握在自己安全的手里，太难了。当一个人拼命伸出双手，开始有希望爬上有幸福希望的和平岸边的时候，突然岸边站起一个恶棍的大棒毫不留情的砸开那双可怜的手，这个世界上，是有人干这种十恶不赦的罪恶勾当的，但上帝是西方人的，这些罪恶如果是西方人干的，都是可以宽恕，并且上天堂的，所以我看过罗马的斗兽场后，更不信任基督教的哲学合理性了。这里罗嗦这么多，说明本身无关风雨的技术，在影响到奶酪问题的时候，也难以摆脱政治的纠缠，不管你叫人家大叔大哥还是爷爷，美国佬就是美国佬，叫啥都没用，这里只是中国人在争取核心技术竞争力的时候被歧视和痛贬的一个小小缩影，警醒每一代的少年在反对过分的民族主义的同时，要知耻自强，命运自握，激烈反对造假

2、 系统架构和NSR/ISSU（HA）

7750的硬件结构，从单板到chassis，虽不能说是非常优秀，但无疑是比较成功的，从紧凑的三维到很高端口密度以及散热电源，比如其最早提供的40*1GE的单板，就用了特殊的PHY使面板可以布下如此高密度的以太口，7750的设计，出奇之处本身都是为了方案，并非为了吉尼斯，因为7750无疑是一款比较昂贵的产品，因为其转发芯片，大容量查表器和TCAM，支持H-QOS的TM等在当时都属于贵族用品，所以成本要高于C76，欧洲人和美国人的差异就在这里J，但是市场没人理你是否内部用了金子还是铜子，客户在冷漠的时候只关心你要我掏多少银子，你是有足够理性的独立行为能力人，你跳不跳楼不应该由我来负责，那用什么来分担这么好的产品的成本，在当时，10G端口大家都贵，并且主要作为收敛后的上行，从方案上也不合适做高密度收敛板，所以高密的GE口收敛板是非常好的idea，AL不是疯子。

7750的软件架构从外部看也是很成功的，主要有以下几点

（1） 其扩展性好，可以很容易的做到一年一个大R版本，年内可能还有几个个RX.x版本，这在以多业务下的IP/MPLS TE为核心的路由器产品，不是那么容易做到的，当然AL没有思科那么大的历史包袱和产品系列，这个可能会更容易，但那么多新模块代码，项目开发本身并不是很难，但是如何快速的和软件平台做好集成，推出商用版本，这就需要系统架构设计要好，当然没有那么完美的东西，毕竟是新产品平台，售后出现问题多一些也是难免的，大家都如此

（2） 可靠性：AL可能是第一个做了NSR和ISSU，JUNIPER可能也很快做到了。

NSR用于主MCU故障时，系统业务不会有任何中断，基本原理就是把所有控制层的session都热备了，使邻居感受不到这个故障，这个和NSF不同，NSF也是要达到这个目的，但是邻居是能感受这个故障的，所以要提前通知邻居，我故障了，别挂电话，我方便一下很快，回来继续聊。另外一个是系统升级的时候，也是业务不中断，具体原理有点复杂，不多说了，因为我不大了解具体实现。但是需要指出的是，AL作为商人，买东西要吹个150%是正常现象，比如NSR，不是所有情况都能NSR，是有条件的，ISSU更是有很多限制，最致命的是，好像升级后24小时还是2小时内记不得了，要重启一下，知道这个隐埋的bomb后，好想笑，不是嘲笑，就是好笑。笑归笑，AL市场包装后的忽悠效果还是很可观的

3、 NSM

NSM在许多国内产商一向不甚重视，而这里放在系统架构后面的第一副领导的位置，可见发达国家市场和发展中国家的不同，也可见AL新ME方案的精心设计，这里有两个原因

（1） 发达国家人力成本昂贵，尤其是IP工程师，所以好的E2E业务的网管对TCO saving非常重要

（2） 多业务的IP/MPLS TE路由器，AL叫SR，系统配置复杂，如果没有好的NSM，这个运维的缺点就会难以掩盖

（3） 思科没有运营山级的ME方案的E2E网管，运营商本身就觉得路由器复杂，没有好的网管就更复杂了，当然思科不是没有能力做这个事，许多网络全网都是思科设备，思科做好网管不是更容易吗？可问题也就出在这里，既然我近乎垄断了，我还费劲巴力的做好网管，给谁省钱？你们和我思科一起玩培训不是让我既能赚钱，还能培养客户的loyalty吗？有何必自断财路呢。这种策略下，以致后来个别熟悉思科产品CLI的用户，不需要GUI的NSM网管，就喜欢CLI

AL是充分分析了市场形势，花了大力气完成SAM6520网管产品（也许名字我记错了），和SR一样，叫SM，业务管理，这些可是AL新ME方案innovation的主要精华之一。并且AL网管的报价方式也不错，把网管价格直接报在端口中，而不是按照网管能管理的节点数来报价，可见AL的报价方式更精细

AL的ME新网管，给似乎已经沉寂了几年的IP网管产品注入了一股新风和活力，重新激活了这部分网管市场，带动了运营商IP网络网管市场的许多供应商的新产品开发动力，AL的新网管对这里的贡献功不可没

4、 OAM

Native Ethernet本身实在是简单，也许因此很久都没人去关注OAM，在LAN的时候，尤其是在企业网，维护很简单，似乎也没什么大问题。而在电信网，OAM在传统网络中从协议到产品实现到网络设计，都占有自己的一席之地，即使到了AL新ME的时候，并没有立刻产生专门的的ETH OAM协议，但是因为新ME的核心是EoMPLS(Ethernet over MPLS/TE)，核心是让L2的的广播域通过VPLS站在在MPLS/TE的肩膀上（VLL可以看作VPLS的一种特例），从而获得MPLS VPN的隔离安全性和TE的可靠性，许多事情都不得不具有相反的两面，你获得的这些好处很难不付出代价，所以EoMPLS同时也把MPLS/TE复杂

性带了进来，所以在没有ETH OAM标准前，AL就做了私有的MAC ping/tracert，并通过网管和VCCV ping（VPLS），MPLS ping/tracert等作了还不错的关联，可以说给基于EoMPLS的新ME一个初步的OAM解决方案，虽然不甚完美，但在AL的OAM特性及配套网管的包装下，能做的这个程度，虽然不能和ATM/SDH的OAM相媲美，但也算过得去，不致让这个短木板太短

5、 H-QOS

这里不提新ME方案，是因为在AL7750前，没有产品做HQOS，QOS要做到什么程度，这个争议一直就没有停止过，而7750包装了5级H-QOS，my god, what are they doing?!直到现在，HQOS的商用并不广泛。IP QOS的研究应该早于2000年，但真正开始实现大概从2000年开始，QOS一直被认为是是否重要的事，这在非常重视质量的西方是没有疑问的，疑问在怎么做？谁来做？既然说到QOS，所以插入一些背景信息，以便理解。QOS的分类方法很多，按照时间或者怎么说，按照事先避孕或事后丸补救，划分两大类

（1） 事先避孕：术语叫CAC（Connection Admission Control），这从传统的电路交换通信时代就开始有了，这个时候QOS其实不是可选，而是MUST，因为是电路硬链接，在真正建立呼叫前，你必须通过协议把各段电路分配好了，然后才能告诉两端，OK，连好了，你们做吧，如果电路资源不足以建立这个链接，那么只好让两边等，当然，你们有YY的权利，这不可耻。所以不存在用户多了影响通话质量的问题，只是影响接通率，传统语音发达了几十年，有一整套监控质量的数据体系，什么呼损率了等等，我不是很熟，但一般来讲，只要接通了，就可以安全的做了，不比担心被抓，拥塞产生的语音通话质量问题，更多的是在VOIP刚开始流行的时候，这里面，万一资源用光了，有重要电话不能做，影响重大如何处理，不用担心，再挤的火车，也得留出一些首长专座以备不时之需，不是腐败，比如119，110等，你得给人家一直预留好资源，保持线路通畅吧。首长也一样重要，首长不舒畅，如何为人民服务？首长因得不到好的服务而生气冲动，作出错误决策，那要害多少人。在IP电信化的考虑中，这个技术是首先要被考虑的，在还没有MPLS TE的时候，就考虑利用IP HEADER中的一些保留bits来做拥塞信息的标记，通过负反馈机制告警汇聚接入点的设备，客满了，普通客人，就甭接了，但对VIP客户，当然要继续接，思科路由器还实现了一些RFC，后来这些研究也在MPLS/TE上做了一些，因为目前看不到什么实际意义，所以也忘了这些RFC的名字，技术象妓女，一旦过气，就人前冷落鞍马稀，懒得有人光顾了，人类有时候真是太不是东西了。IP的CAC，最主要的还是TE技术，现在还有一些类似的方案，比如从终端开始发起TE tunnel/LSP，从理论上当然是很好的，但是商用上有很多麻烦，TE的麻烦事很多，比如TE的带宽分配模型就很难记住，只记住一个名字最好玩的，叫俄罗斯木偶，但这个木偶是怎么工作的的，次次记，次次忘。因为TE需要路由协议扩展配合，还有各种麻烦，所以对大容量的TE tunnel/LSP的支持，也会很麻烦，也就意味着增加成本。如果传统的基于硬件电路独享CAC可以叫硬CAC，那么基于TE的CAC也可以叫做软CAC，TE的主要目的是把MPLS的面向连接的LSP提供预先资源保证（QOS保证），所以从IP(无连接)-MPLS(有链接)-TE(有保证的连接)，是IP技术发展的三大步走，但是因为IP电信化的FMC进程并没有那么快，而IP网络主要的traffic generator还是电信不会提供QOS保

证的internet，所以TE的QOS技术基本上没有什么大的用武之地，商用部署有限，具体到部署，在一个线路上，一部分要做RSVP，一部分不做，是不好处理的，具体就不说了，包括DS-TE这些麻烦的东西，也就省了说了

（2） IP QOS

前面的QOS可以算情色QOS，有人说情色不是色情，不过后面的章节熟套或流水帐可能多一些，不一定有意思。大片重映观众的高潮更多在中段，没有悬念的尾声还不如夕阳红，第二次高潮，在大片后新气象里可能会有一些。但是GMPLS/TMPLE/FOCE及之后更新的研究，因为和个人以商用为主的工作距离大一些，一时可能难以有时间去仔细一点的关注了，所以相关章节会拖的较久

6、 L2VPN

这本是AL的EoMPLS新ME的核心转发和业务支撑技术，可分为点到点的VLL和多点到多点的VPLS，MEF的称谓有所不同，并略有细化，分为E-LINE/E-LAN和E-TREE，E-TREE其实是VLL和VPLS结合的产物，也有叫SPOKE PW的。E-LINE/E-LAN的名字写起来很清楚，但读起来很麻烦，类似在做presentation中常用的cost和QOS，读起来较易混淆。

VLL是很简单实用的管道，对于点到点管道业务，很好，主要问题是

（1）CE/PE间的HA不好处理，当然可以租两条VLL，但是就怕提钱

（2）对于中大型企业网的多点需求，有点麻烦，一个是钱的问题，还有多条VLL带来的VLL资源浪费

（3）广播/组播业务难以支持，每个PW都复制的方式显然不爽

VPLS很灵活，理论上可以做任何业务，对多点业务，节省PW资源，CE双归的HA容易，支持组播，但问题如下：

（1）广播/组播/未知单播抑制：要基于接口/VSI/chassis，要基于packet和带宽，要绝对数值和percentage，都是trouble

（2）环路：VPLS的CE接入的HA方案，要解决L2环路问题，这里花样繁多，有AL的MAC flapping局部方案，有STP in VPLS的方案，还有其他厂商的私有方案，也是trouble

（3）MAC容量限制：因为VPLS要学习MAC，自然涉及到学习的性能，还有容量问题，对tier1运营商，在汇聚的中心节点，甚至有million机的MAC地址需求，这么多MAC，管理等都是trouble

所以VPLS带来的好处相比于同时带来的这么多麻烦，个人观点是尽量不用，谁用谁知道麻烦在哪？甚至还有客户为解决一些VPLS的问题引入PBB over VPLS，faint。当然，在组播场景下，如果用L2组播，还是不得不用VPLS，所以是用L3组播还是L2组播，个人倾向于前者，只是设备商也很苦，不能总是便宜运营商，所以对于便宜的ME接入和汇聚节点，L3经常要单独出来收钱

E-TREE：是很实用的模型，现实的商用网络，full meshed/half full mesh的多在核心，而在接入汇聚层面，更多的是TREE形，比如DSLAM和BRAD的关系模型；另外就是dual-homing的Y形，中文翻译成丫形，这是E-TREE的最简单的形式，可以用VLL redundancy，也可用这种spoke VPLS，简单的实现双归HA，但是protection switching的

性能有限，要提高，还需要BFD/TE FRR和MAC地址快速withdrawal配合，HA的解决，好像就很少简单过
L2VPN在一些时候需要透传所有报文，包括L2协议报文，尤其是VLL的时候，可以叫E-PIPE。L2VPN的透传需求带来一个QOS问题，就是拥塞的时候如何保证L2/L3协议报文的优先级，按说应该保证才好，但实际上好像没有这样配置，其实如果扩展起来，是不是L4以上的协议报文也要保证？那可能要DPI了。但是对于路由器本身主机的协议报文，一般是要配置一个默认的高优先级队列，否则拥塞的时候就会导致协议中断。

7、 VOIP/IPTV

补充一点H-VPLS，和H-VPN(L3VPN)有点类似但不同，H-VPN可以减少UPE上的VPN路由学习数量，H-VPLS和H-VPN拓扑有点类似，但并不能减少MAC学习的数量，我理解就是打开了普通VPLS的水平分割。
QOS的三个关键是：classification时识别深度和能力；队列数量级数和灵活性；精度
TV是推动人类文明进步的一个伟大发明，语音把文明从无声世界的传播带到了有声世界的传播，而TV则进一步把世界带到了视觉世界，黑白和彩色的区别虽然也很大，但相比于从黑暗到光明，就不大了。从此语音和视频成了人类生活的基本需求，并且及其简单，对于用户，你只需拨一个号码，或者按一下频道，就可以获得需要的声音或视频，及其方便。在视频和语音的承载网络上，传统网络比较简单，尤其是视频，像自来水一样，建立一套管道，内容在自来水厂，用户只要打开自家的水龙头就可以享受到丰富多彩的内容，有传说青岛的居民还可以在自己的水龙头里接出啤酒来。青岛的湾友可以澄清一下。
语音/视频和INTERNET本来大家井水不犯河水，相安无事，各走各的独木桥好好的，但首先是internet多媒体化，PC越来越多，那么之间打个电话是很容易的事，VOIP从此而起，有好事者把internet VOIP搞到了运营商，在早期，以电路语音为主的传统电话容量非常大，并不care VOIP有什么降低成本的好处，尤其是中国，早期的VOIP电话卡不少是假的，其实就是普通的电话包装个名字，所以你会觉得VOIP真牛，通话质量和原来一样，殊不知，本来就是一样的，如果有一天不一样，也是运营商自己加点扰，就像有idea未来防止P2P过度，故意让P2P业务在传输时质量差点一样。但是随着IP网络因为internet的洪水泛滥，导致IP网络的建设成本今天已经成为运营商的大头，所以再维护一套传统语音的成本就逐渐越来越可观了，所以几乎没有几年VOIP就替代了传统语音。世界有时候很戏剧，固定电话IP化不久，因为固定语音被移动语音的侵蚀及其迅速，导致vendor已经不愿意在去竞争固网NGN的项目，甚至到了互相送给对方的地步。
说到语音技术，在传统语音时代，大容量语音交换机可是西方封锁我国的一个核心技术，巨大中华的发展起源于解放军通讯工程学院的邬江兴教授在万门机的breakthrough，现在邬早已经是将军了，这个技术在中国的扩散虽然有不少或公或私的官司，但从结果看，是造就了中国通讯产业的大爆发，从这个角度看，邬教授功德无量，当然这里，还有89后西方对中国封锁万门机技术和产品的关键因素，可见封锁很多时候是中国发展的逼迫剂，而开放却可能扼杀中国的技术，这样看来似乎西方人傻了，既然想扼杀中国的核心技术，那为什么不开放呢？其实不傻，如果真的开放了，中国人也不傻，学得更快，更疯狂。从技

术产业角度看，知识产权的保护和垄断一定要在一个适度的范围内才能促进产业的繁荣，过度保护无论对技术产品和是老百姓生活都是利大于弊的。西方许多产业的繁荣，也是依赖于核心技术的快速传播，包括跳槽创业就是很好的方式，而竞业禁止如果真的被严格执行，那就是以在杀人。

具体到VOIP的技术，其实比传统技术复杂，有两套体系：H323和SIP，前者技术简单一些，后者复杂一些，所以传统运营商选择更多的是后者，而新VOIP运营商可能用前者多一些，我不是很熟，所以就不写了。这里更大的创新不得不提到skype，有核心的技术可以通过普通廉价的IP线路保证语音质量，曾经非常火，国内也有following的。但是语音这块蛋糕，无论如何做，在五彩斑斓的IP业务中，都没有太好的利润，所以skype被一个巨头收购，忘了名字，后来好像又分离了，看来是失败的收购，skype还是可以活的尚可，但风云靓丽了几年后，应该很难再有show time了。语音的复杂其实IP承载并不占很多，更多的体现在各种制式的互通上：传统语音和VOIP，移动之间（GSM-CDMA-WCDMA-TDD-LTE），移动和固网，H323和SIP，导致需要很多互通网关，而语音是通讯的最基本业务，保证其质量在这种复杂的互通情况下，并不是那么容易，比如接通时间和时延问题就不是那么好处理，我对这方面的不熟，只是点到而已。

说到VOIP对网络质量的要求，在刚刚开始的时候是绝对怀疑IPQOS的，所以早期一些运营商比如中移动甚至建立独立的IP网络来承载语音，甚至有国家要求必须保证传统语音要做为VOIP的应急备份以保证有灾难发生时候的通讯。随着IP带宽的迅速扩大，语音那一点点流量逐渐成为带宽的一个零头，给你一个EF队列，在高速的IP网络里，时延基本不再是一个问题，但是在需要忽悠客户的时候，对这些从语音时代走出来的电信运营商，语音对网络质量的高要求，现在仍然是一个还没过是的主题

视频是带宽的主要占用者，所以说到IPTV，就要说一下带宽的问题，在有独立带宽保证的TV传输网络，网络本身基本上不需要提供什么QOS保证，这也符合internet的IP承载网带宽充足，就不需要QOS的观点。因为2001年的IT泡沫，带来了大量的骨干网光纤资源，而WDM技术的发展，目前已经超前IP流量，所以运营商的骨干网带宽到现在都不是问题，并且自有率很高，租用也比较容易，这都是托2001年IT泡沫的造福。但是在接入网方面，情况就完全不乐观了，而TV对接入网带宽要求较高，所以接入网带宽是IPTV的瓶颈。

标清视频需要4M的带宽，考虑到同时还有HSI是频道切换时的单播加速技术，那么最小带宽需要是6M，对于最大量的铜线接入技术，需要ADSL2+来承载，同时还受到距离的限制，不是所有的的用户的ADSL都可以达到的。所以基于ADSL的IPTV，质量保证技术要求就比较苛刻

接入网带宽提升方法很多，从早期的Ethernet到户，到已经喊了很久并开始逐渐部署的FTTH技术，但是这里主要的问题是成本谁来承担的问题，因为只是增加一个IPTV，每月每个用户增加的收入也就10-20欧，那么改造这样一个用户需要多少钱呢？没具体值，但看发达国家高昂的人力成本和较低的施工效率，一定是很高的，那么结果就是ROI非常低，所以没有人愿意承建。但高速信息公路是很好的东西，怎么办呢？两个办法：

1、 国家投入模式：如日本和新加坡，FTTX很好主要是以国家投入为主，这和我党改革开

放要想富先修路的思想有点像
2、 运营商投入但要求专营权的形式：因为欧洲电信法早就通过了，所以所有驻地资源都必须开发，并且价格不能完全由卖方说了算，导致incumbent的运营商因为长期官僚和低效率的问题暴露出来，而新运营商得到不小的发展机会。但是对于FTTX，在国家模式不愿意的情况下，只有incumbent的carrier有能力建设，但他们提出，要我建可以，但要求一定年头的专营权，否则这种自己种树给大家乘凉的傻事，打死都不干
3、 像中国这样有钱有人成本低的国家，倒是可以大规模兴建FTTX，尤其是没有固网而最有钱的中移动，最应该用大量的利润快速覆盖FTTX，既能占领未来的接入网市场，又可以不用分很多利润给老外持股者，利民爱国的好事，要赶紧做呀
IPTV虽然不那么好，但却是triple play的核心，所以也是AL新ME核心，主要包括如下内容：
1、 基于VPLS/TE的IPTV承载方案：
（1）核心是环网方案，AL称为daisy chain，就是环形H-VPLS，如果环上的节点或者链路故障了，则通过TE FRR做环回，这种方式的问题是有点复杂不说，也浪费带宽，H通过PIM redundancy优化了这个方案，后来AL也follow了。
（2）树形H-VPLS方案，通过VRRP in VPLS来解决链路备份问题，同时也解决了这种拓扑下组播流量的多发问题，好像整体上还要配合一些MAC withdrawal技术，记不得了，反正很烦
2、 频道快速切换和丢帧重传
这是基本的QOE保证，始作俑者是思科和微软，一些简单分析如下：
（1）频道切换问题：这个问题的本质是MPEG等视频压缩标准+组播定速报文发送导致的，因为其基本原理是增量压缩技术，如果其基础帧，这里就I帧拿不到或者丢失，那么即使后面的B/P帧正常收到，也无法解码，这些帧就废了。等到下一个I帧到达后，图像才能开始，但这个时间因为受到组播定速发送的限制，是秒级附近，所以我们在看数字电视的时候，频道切换不如模拟电视快，很不爽。优化技术原理也很简单，就是在系统得到频道请求的同时，先用单播把最近一个I帧甚至其后的组播流量真正到来前的B/P帧给加速发过来，如果只有I帧，很多情况下就会有马赛克，如果不加速，那么就不能赶上组播的速度，和组播同步的时候还是有马赛克，而加速带来的问题就是，组播来了以后，要有一个减速，具体效果是有快进感，在技术实现细节上，可以通过修改视频帧的时间间隙参数使这个快进尽量平滑一点，这样处理后的视频，效果基本接近传统电视了
（2）视频丢帧问题：数据通讯丢包是正常的，所以在协议设计上都有相应的重传机制，而组播因为其性质决定了没有该特性，但丢包怎么办？优化方法就是增加了单播的重传机制，此时需要STB要感知到丢包并发出请求，这在早期的STB可能是不支持的，如果是基于Linux的STB，那么一般是可以通过Java做一个补丁支持该功能，否则就可能需要STB上游的设备支持一个proxy，但能否搞定不是很清楚
以上的功能需要IPTV的headend服务器提供相应的功能配合，在用户数越来越多的情况下，对集中式的服务器的压力和带宽浪费都是问题，尤其是很多人同时打开电视的时间

段，比如晚上的某段时间，或者热点节目的时候等。解决办法是可以在承载网络的一些节点上增加一些视频cache卡，这种情况下，这个节点应该是可以被STB通过L3访问到的，当然通过L2也可以，那会导致很多终端用户和这个视频存贮节点在一个大的L2 domain，这是很不好的网络设计，所以原则上这个节点应该是L3节点，而最好不用纯L2节点。上面只是一些简单的分析和描述，但也可以看出为了保证达到和传统电视一样的QOE，IPTV很麻烦

因为结构上定了上中下，只好把下写完，可能会比较鸡肋一点

补充一点TE的内容：TE因为比较复杂，所以TE LSP/TUNNEL在早期的capability是不大的，一般都建议用在CORE，而一些tier1的运营商CORE也很大，后来capability扩大了很多，但是如果要扩大到全网的TE，PE上问题未必很大，毕竟，并不需要和所有的PE都要全链接，但是P上可能有有些恐怖了，因为越是核心的P，越是有很多PE互联要经过它。所以仍然需要一些TE交换/分段PE/分层BGP等技术来解决扩展性的问题。在具体部署上，客户既希望能手动控制路由的选择，又要求能自动完成部署

曾经在这个大章的开始，痛贬C65/76，其实是有失公允，人在一定情绪和情势需求下的语言和文字虽然不免精彩，但也难免偏颇，对于C76/65就是，其实C76/65的成功是非常好的case，低价单板和高价单板共平台，在市场上有很多好的效果，一些实力有限的运营商，喜欢便宜的定西，不会一次就买一个功能很好的东西，虽然许多功能现在不用，但可能以后用。那么OK，给你便宜的单板，但后来当需要新功能的时候，只要合同没问题，那么需要物理更换成贵的单板，C合情合理很爽的又隔了一茬韭菜。类似的这种手段，在C的模式影响下，虽然大家也都在一定程度上使用，但还是C用得最娴熟老道

AL的新ME方案有了成功的开始并打下了良好的系统架构和方案架构，那么接下来的完善相对就比较容易了，下面简单介绍一些ME上用的技术

1、 ETH OAM问题：802.1ag(CFM)/802.3ah（EFM）/Y.1731，其中以Y.1731最全，但协议规定的配置方法比较复杂，尤其是802.1ag和Y.1731定义的MEP/MIP等等一系列概念最麻烦，如果完全按照协议规定的方式去做配置命令，本身就会带来一些OAM问题，所以实现的时候最好做UI上的简化处理

2、 环路保护问题：VPLS本身通过水平分割来解决环路问题，但是并不能阻止CE网络本身带来的路由环路问题为城域网的影响，尤其是在CE双归的时候，环路的几率就变大了，所以需要一些机制来处理，比如ALU的mac flapping（部分的解决），或者通过单独loop检查报文，再有就是STP in VPLS，把STP跑在VPLS里，感觉就很烦，总之这些解决方案都不是很好，最好在网络方案上避免环路，从这一点，个人不是很喜欢VPLS，纯VPLS带来的麻烦和好处可能是一样多

3、 MAC地址问题：VPLS带来大量的MAC地址，所以只好想办法解决，一个是拼命扩大MAC地址容量，比如到1M；再就是在某些场景下并不真正需要MAC学习，那么禁止VLAN内MAC学习，还有就是把PBB用在VPLS里，解决某些特殊场景下的MAC容量问题，和环路一样，这些方法都让人感觉不爽，以太网的一些基本问题，新ME的VPLS在实际网络应用中其实并没有很好的解决，所以个人不喜欢VPLS

4、 Reliability问题：SMARK LINK是思科的首创，TRUNK是标准协议，而MC-TRUNK是ALU的首创，本身其实并不复杂，SMART LINK可能是和MC-TRUNK类似的东西，但具体内容没看，目前更多的是MC-TRUNK
5、 slow protocol处理：慢协议其实就是native Ethernet上的L2协议簇，需要能灵活的定义那些透传，哪些终结
AL的在ME的是市场份额是不错的，但是在其整体的revenue中，应该是可以忽略的，AL最新在有一些做cluster向CORE扩展的roadmap，但整体上，还没有看到其全面进军数通市场的决心和策略，这一点，和华为最近的表现是有一定区别
另外提一点电力上网和cable上网，忽悠是不顶用的，电力的强弱电干扰问题和cable的共享网络结构问题，目前基本上没解，所以只有很少的电力和cable上网用户。
本来靓丽的AL新ME，用此章草草收尾，似乎预示着这个ME方案也该变一变了。

另类玩法PBT

刚刚有人“踢馆”，遇到此类事情，难免不爽，我也远非非常大度的人，所以回复里也含讥讽，想来何必呢，所以在此篇开头正式向那些同学表示道歉。
应该总结一下AL新ME的问题：
1、 技术上个人认为主要是VPLS带来的问题，具体细节前面都分析过了，不在赘述
2、 成本上主要是过高：原因是AL支持的都是海量的MAC/FIB/queue等，加上复杂的功能，不可能迅速降低成本，这也是AL把同等产品软件阉割一下推出比7750低价的7450的一个内在驱动。这还是没有把MPLS到运营商最边缘的情况下，如果到了最边缘，会如何呢？这是个大问题答复比较复杂，后面再讨论
AL新ME的创新，其实并没有脱离IP网络本质的巢穴，在增加业务支持和可靠性的同时，不可避免的带来了网络的复杂性，这在具有多年传统网络概念的电信运营商来看，总是觉得其本质有问题，需要改造，所以很早就有人提出IPTN(IP电信网)的概念，当然也会有一些新的draft、专利出来，概念本身并不复杂，IP节点傻瓜化，所有路由集中控制集中下发，IP节点只管按照管理中心下发的路由指示去转发，包括QOS参数，别的不用管，这不是很简单吗？这种idea现在已经演变到FOCE的标准簇，但仍然没有得到广泛部署，很难说是什么原因？思科因设备简化导致卖不上好价而消极？背离IP节点自智能的基本原则？标准不成熟历史网络难以改造？
在AL新ME火热的时候，其设备贵，网络复杂的弊端也自然会暴露出来，这对运营商在降低成本的压力下，是有考虑一些可能解决方案的驱动的，其中BT就是最激进的代表，BT本身在tier1运营商里排名不高，但是确是较早提出下一代网络规划的运营商，称为21CN，北电在种种不幸里，在数通领域已经没有什么顾及了，两者组合在一起，提出PBB的L2网络，基本思想和IPTN类似，只是利用了MAC IN MAC代替IP/MPLS路由，MAC IN MAC是为了解决对客户MAC学习带来的MAC容量问题，通过PMAC隔离CMAC，这个专利就像铅

笔上加一个橡皮头一样简单，所以被日本人发明了，并且成本基础专利，让喜欢专利的人很生气，还好，现在基本也没啥用了
PBB需要一套独立的网管，来学习所有的MAC地址，然后下发到各个L2节点，各个L2节点是各种size的傻瓜，除了和网管的通讯协议，别的什么协议都没有，是个傀儡，傀儡能卖多少钱？很明显，思科是绝对不会傻到大力支持这个方案的，其实除了一无所有的北电，借椽登槐的华为，没人真的喜欢PBB。PBT在BT表面的火热中，已经演进到了PBT(PBB TE)，有一些技术问题比如组播等还没有解决，当然，随着网络的真正商用，也许会有更多的需求，但未必是大问题。在风风火火了几年后，BT终结了PBT，几乎在一夜之间转向了AL的7750，并且一路扩展。想来是BT高层对PBT迟迟不能商用全面部署影响了这几年业务的迅速拓展十分不满，所以很快勒令下马，迅速开始AL 7750，这个结果让一些人真的很伤心，陪太子读书很多年，结果成了曹雪芹，不知道有没有红楼梦可以塞翁失马，聊作安慰。
目前还有类似思路的有T-MPLS/G-MPLS等，都不是IP vendor真正喜欢的，具体没有仔细研究。
顺便提一个城域网传输中的一朵小昙花：LDMS，无线光网，这种无线也就是微波类似的技术，无论怎么做，带宽都是有限的，所有偶尔有些特殊的情况用一下，没有任何规模化的可能

彎曲評論
科技 · 人物 · 潮流

思科核心路由器CRS-1的研究（上）

陈怀临，首席科学家
《弯曲评论》
www.tektalk.cn
huailin@tektalk.cn

1. 前言:

在通信领域的高端设备上，核心路由器历来是皇冠上的珍珠。本文对思科的核心路由器CRS-1和CRS-1上的重要部件SPP网络处理器进行考察，从而使得读者能够对CRS-1的体系结构有个整体的了解和把握。

2004年5月25日，思科正式发布其下一代IP核心路由器CRS-1。其原文可参阅：思科CRS-1新闻发布稿。在CRS-1的新闻发布中，关于CRS-1系统的创新和亮点思科是这样描述的：
× Cisco IOS® XR Software, a new member of the Cisco IOS Software family, designed for terabit-scale routing systems built on massively distributed multi-shelf architectures
【笔者译：】思科的IOS-XR操作系统。一个为支持多机箱互联的大规模分布式体系结构而量身定做的一个崭新的IOS家族成员。
× System capacity of up to 92 terabits per second (Tbps)
【笔者译：】系统容量，（在最全配置时），可高达92太比特每秒。（太比特：Terabits。1 Tbps = 1000Gbps)
× Industry's first Optical Carrier (OC)-768c/STM-256c packet interface
【笔者译：】工业界的首次的40G的OC-768光网络接口。（OC-768接口线速为39,813 Mbit/s。通常简称为40G端口。但不要与40G以太端口相混淆。这里是SONET光网络接口。是一种WAN，广域网接入接口。以太端口相当而言是一 个局域网端口的范畴，如现在广泛采用的10GETH以太网端口。OC-XXX指的是数据信号在SONET光纤上传输的速率。）

× Cisco Silicon Packet Processor (SPP), the world's most sophisticated 40-Gbps application-specific integrated circuit (ASIC)
【笔者译：】思科的SPP网络处理器，（在2004年），世界上最复杂，最强大的支持40Gbps的ASIC芯片。
× Extensible Markup Language (XML)-based Cisco Craft Web Interface (CWI), a visual management tool that can manage single-shelf or multishelf systems.
【笔者译：】基于思科的CWIde扩展XML描述语言，一个基于图形界面的管理工具。
× The Cisco Intelligent ServiceFlex design for service flexibility and speed to service
【笔者译：】思科的智能ServiceFlex设计方法，确保服务商能在各个层面提供层2和层3的数据服务。

笔者将会把注意力集中在CRS-1体系结构，CRS-1的SPP网络处理器，和相应的IOS-XR操作系统方面。力图把一个高端核心路由器的整体全貌分析，解释给有兴趣的读者。

2. 产品系列

如果访问思科的主页，浏览其产品分类，读者可以发现，在路由器这个产品线上，分为Branch，WAN和Service Provider三大类产品。在Service Provider子类里，CRS赫然列为最高端产品。其层次如下图所示。

从图中可见，笔者在前文“思科QuantumFlow处理器与ASR1000的研究”一文中介绍的ASR1000属于边缘路由器（Edge Router）的范畴；而CRS-1是运营服务商（Service

Provider）在主干网上的核心路由器。其地位对于一个公司数据通信产品解决方案是举足轻重，属于重型装备。
CRS-1产品是一个产品家族。其分为2大部分，共4个产品。第一部分叫做线卡机（Line Card Shelf）。第二部分叫做交换连接机（Fabric Card Shelf）。Fabric Card Shelf是用来把多个Line Card Shelf连接成机群的产品。在Line Card Shelf部分，目前思科提供三个产品，分别为4个线卡槽的CRS-1（4 Slot Line Card Shelf），8个线卡槽的CRS-1（8 Slot Line Card Shelf）和16个线卡槽的CRS-1（16 Slot LineCard Shelf)。
下面是CRS-1家族产品系列的图示。

1.CRS-1 4线卡槽单机箱：

思科4线卡槽机箱CRS-1路由器

2. CRS-1 8线卡槽单机箱：

CISCO CRS-1 8-SLOT SINGLE-SHELF

Cisco CRS-1 8-Slot Line Card Chassis

- Eight 40-Gbps line card slots—Using a midplane design, the 8-slot LCC is loaded with 8 MSCs on the back of the chassis, each of which is connected, through the midplane, to an interface module on the front of the chassis.
- 2 dedicated route-processor slots
- 4 dedicated switch fabric card slots (which accommodate 4 switch fabric cards with two switch planes on each)
- Redundant power supplies and fan trays

思科8线卡槽CRS-1路由器

3.CRS-1 16线卡槽单机箱：

CISCO CRS-1 16-SLOT SINGLE-SHELF SYSTEM

Cisco CRS-1 16-Slot Line Card Chassis

- Sixteen 40-Gbps line card slots—Using a midplane design, the 16-slot LCC is loaded with 16 MSCs on the back of the chassis, each of which is connected, through the midplane, to an interface module on the front of the chassis.
- 2 dedicated route-processor slots
- 8 dedicated switch fabric card slots
- 2 dedicated shelf-controller slots
- Redundant power supplies and fan trays

思科16线卡槽CRS-1路由器

4.CRS-1 Fabric交换连接机机箱：

思科Fabric交换连接机箱

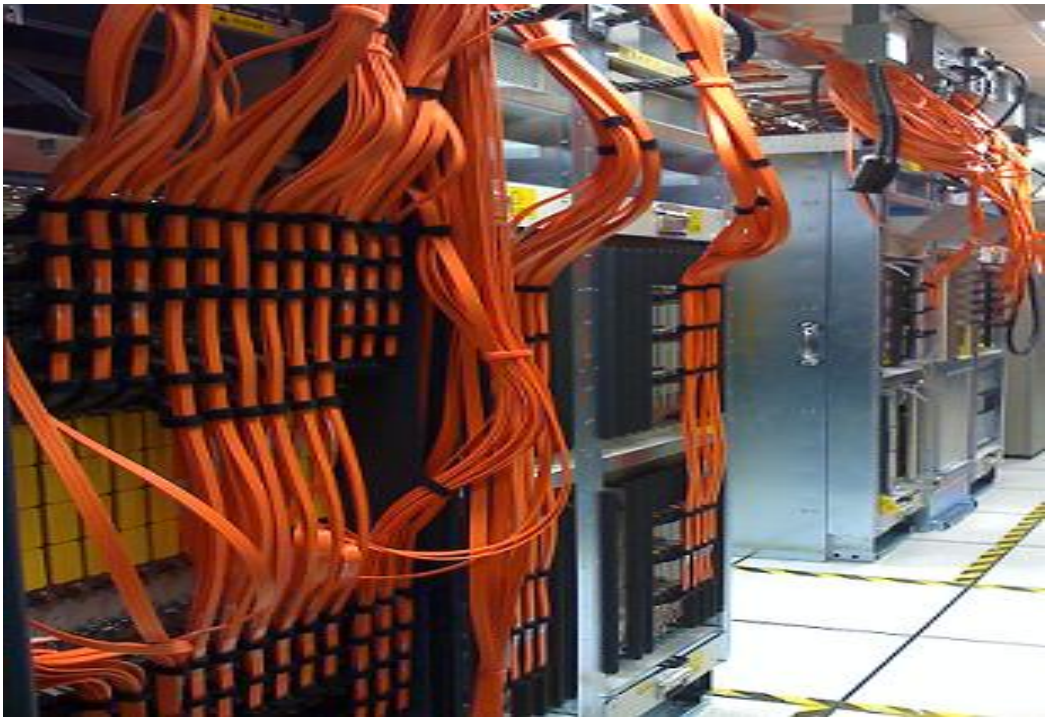
从上图可知，对于4线卡，8线卡，16线卡的CRS-1机器而言，本身就是一个可独立运行的核心路由器。而对Fabric交换机，其本身不是一个路 由器，而是一个非常快速的交换设备从而运营商可以把16线卡的路由器连接成一个路由器群，从而可以形成和支持超大规模的数据通信。读者请注意，4和8线卡 槽的CRS-1不能接入Fabric互联中。

对于4，8和16线卡槽的CRS-1路由器而言，其报文交换吞吐率分别为：（4，8或16）×每个线卡的吞吐率×2.为什么乘2，是因为在Layer-2谈交换吞吐率时，通常是算双工速率（Ingress和Engress）。

因此，4卡CRS-1的交换能力是320Gbps；8卡CRS-1的交换能力是640Gbps；16卡CRS-1的交换能力是1.2Tbps。

对于通过Fabric交换互联形成的机组，CRS-1的交换能力是巨大的。最高配置可以通过8个Fabric交换机，把多达72个的线卡机箱连接起 来支持多达1152个40G的线卡。因此，CRS-1的多机箱互联的最大交换速率可达：1152×40Gbps×2= 92160Gbps=92Tbps。

下图所示为一个思科CRS-1 互联的实例图：

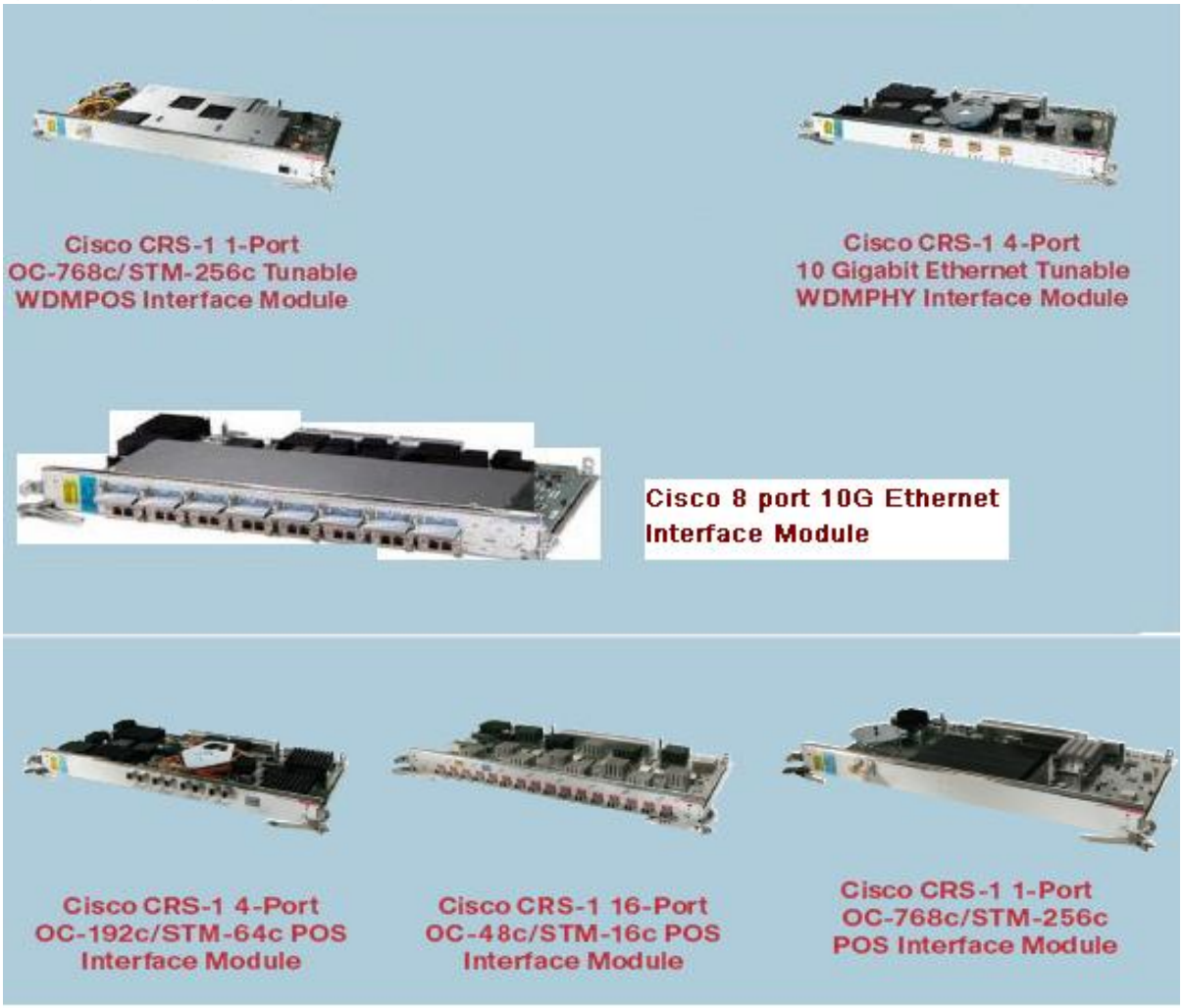


3. 端口配置

CRS-1产品家族的4，8和16线卡槽的线卡机（Line Card Shelf)可以单独作为路由器被安放在网络的节点上，也可以通过24槽的交换互联机（Fabric Shelf）互联形成一个强大的路由器阵列，提供超大规模的IP/MPLS报文交换能力。
从微观角度观察，其最基本的数据通信的粒度就是通过位于线卡机上的线卡（Line Card）的接口模块（Interface Module）所提供的。这里读者注意一个CRS-1的概念和术语：CRS-1上的线卡其实是包括3个部分（硬件）：接口模块卡（Interface Module），模块服务卡（Modular Services Card）和中间板（Midplane）。换言之，线卡是2个硬件卡通过插在中间板上组合形成。
CRS-1产品系列支持如下的接口模块：
×40Gbps端口的OC-768c/STM-256PoS卡。这个（40Gbps）卡上就是只有一个（40Gbps）端口（Port）。
×10Gbps端口的OC-192c/STM-64c PoS/DPT卡。这个（40Gbps）卡上有4个端口。4×10Gbps=40Gbps。
×10Gbps端口的Ethernet以太卡。这个卡上有8个10Gbps以太端口。40Gbps交换。可以Oversubscribe只利用4个端口。

×2.5Gbps端口的OC-48c/STM-16c PoS/DPT卡。这个（40Gbps）卡上有16个端口。16×2.5Gbps=40Gbps。
×40Gbps端口的OC-768c/STM-256 tunable WDMPOS卡。这个（40Gbps）卡就是只有一个（40Gbps）端口（Port）。
×10Gbps端口的Ethernet 以太tunable WDMPHY卡。这个（40Gbps）卡上有4个端口。4×10Gbps=40Gbps。
对于4，8，16卡槽的CRS-1线卡机，针对上述的端口，其配置相应为：
×4, 8, 或16 个OC-768c/STM-256 PoS端口
×16, 32, 或64个 OC-192c/STM-64c PoS/DPI端口
×32, 64, 或128个 10 Gigabit Ethernet以太端口
× 64, 128, 或256个 OC-48c/STM-16c PoS/DPT端口
× 4, 8, 或 16个 OC-768c/STM-256 tunable WDMPOS 端口
×16, 32, 或64个 10 Gigabit Ethernet tunable WDMPHY端口
当CRS-1线卡机通过最多可达8个Fabric交换机互联起来时，线卡机的数量可达72个。因此，**在最大容量下，这72个线卡机可以都是16槽的线卡机**。因此，CRS-1多机组的最大交换容量为：
× 72个线卡机
×（最多）8个Fabric交换机互联
×交换能力：**72×(16*40Gbps)*2=92160Gbps=92Tbps (16*40Gbps*2是一个16卡槽的线卡机的双向交换能力)**
×支持40Gbps接口模块：1152 个40Gbps接口模块（72×16=1152个40Gbps接口模块）
× 1152 OC-768c/STM-256 PoS 个端口
× 4608 OC-192c/STM-64c PoS/DPT 端口
× 9216 10 Gigabit Ethernet 端口
× 18,432 OC-48c/STM-16c PoS/DPT 端口
× 1152 OC-768c/STM-256 tunable WDMPOS 端口
× 4608 10 Gigabit Ethernet tunable WDMPHY端口

下图所示为各种接口模块和端口：



4. 线卡机配置

在讨论了CRS-1的接口模块（Interface Module）之后，对CRS-1能支持的端口（Port）配置就有了一个基本的了解。本节介绍这些接口模块是如何配置在线卡机（Line Card Shelf）之上的。在具体介绍和举例之前，读者要抓住一个概念，一个线卡机上的端口模块槽可以插入不同的40Gbps的端口卡，例如上OC- 768，OC-192，或者10G以太卡或其他端口卡。可以是各种混合模式。对系统而言，只要是支持的端口卡就可以。对系统而言，就是一个40Gbps线 速的报文端口硬件板子。不管你是光纤还是铜线，和层2的数据格式，一旦走到Fabric互联网络入口，都一样；都被打散成为Cell，做高速Fabric Switching了。
LCC线卡机内部的板子都是通过中间板（Midplane）互联的，不是Backplane（背板）。另外，对理解机箱板子布局， 抓住几个要点：
×端口卡（Interface Module）一定都是在机箱的正前面。
×路由处理器卡（Routing Processor Card）一定都是在机箱的正前面。
×交换卡 （Fabric Card）一定都是在机箱的后（背）面。
×模块服务卡（MSC）除了4槽的CRS-1线卡机，都是在机箱的后面。
×线卡机都是基于中间板（midplane）的设计，而非背板（Backplane）。

4槽线卡机
机箱的前面（Front），总共有10个卡槽。其中：
×4个40Gbps的端口卡槽（或叫做接口模块）
×4个支持40Gbps的模块服务卡（Module Service Card）槽。其中，每个MSC上有两个SPP网络处理器。关于SPP会在后续的章节里介绍。
×2个路由处理器卡（Routing Processor Card）槽 。在思科的网络产品里，路由处理器就是系统的控制平面。在Juniper的产品里，相应的概念是路由引擎（Routing Engine）。名称不同，表达的概念和功能是类似的--系统的控制平面。
上述的10个卡都通过中间板（Midplane），插入并连接在系统中。
×4个Fabric卡，在机箱的后面（Rear），插入中间板。从而与其他在 前面的10个板子连接，形成一个路由器系统。

下面是一个满载的4槽线卡机的例子：

思科4线卡槽机箱CRS-1路由器

8槽线卡机

机箱的前面（Front），总共有10个卡槽。其中：
×8个40Gbps的端口卡槽（或叫做接口模块）
×2个路由处理器卡（Routing Processor Card）槽 。上述的10个卡都通过中间板（Midplane），插入并连接在系统中。
×8个与端口卡对应的MSC模块服务卡和4个Fabric卡，在机箱的后面（Rear），插入中间板。从而与其他在 前面的10个板子连接，形成一个路由器系统。

下面是一个满载的8槽线卡机的例子：

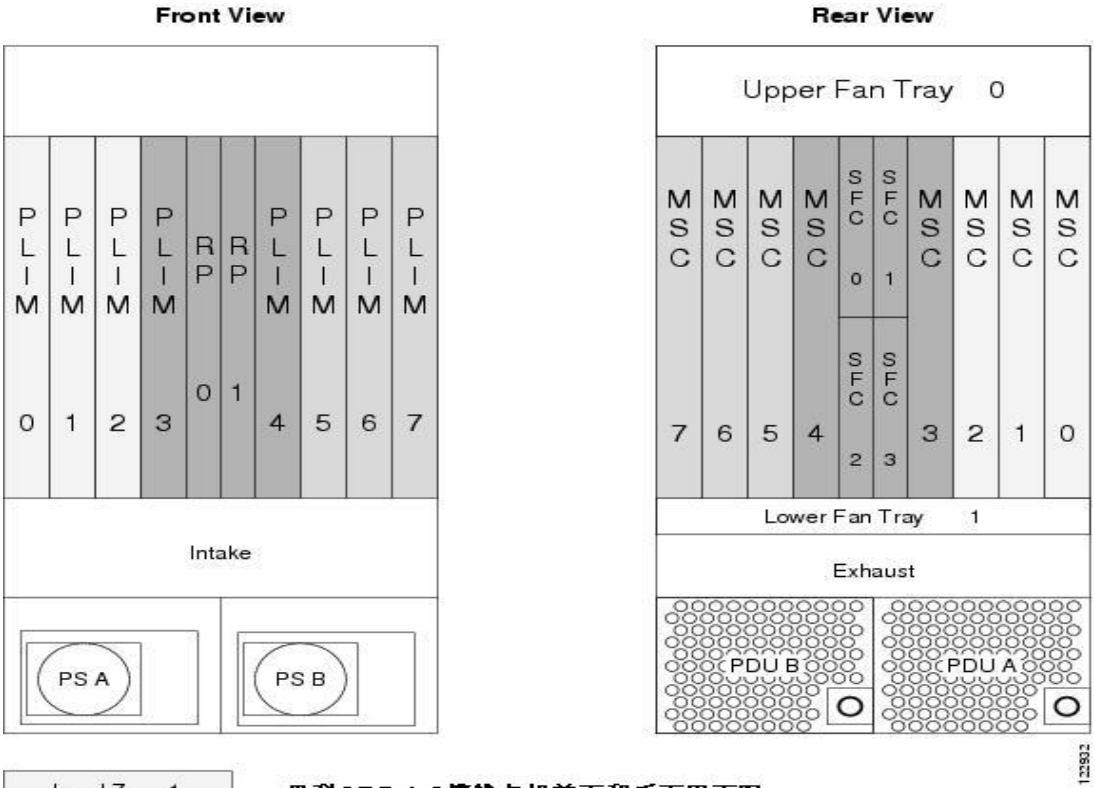
CISCO CRS-1 8-SLOT SINGLE-SHELF

8个40Gbps的端口卡槽（卡槽编号为1，2，3，4，7，8，9，10）。卡槽5，6是双控制平面RP（路由处理器卡）。

相应的8个支持端口卡的模块服务卡（MSC）是位于机箱的后面，通过并插入中间背板，与前面的10个板子互联成为一个系统。

这个8槽线卡机，1，2是两个OC-768卡。3，4是两个8端口的10G以太网卡。7，8，9是4端口的OC-192卡。10是16个端口的OC-48的卡。

思科8线卡槽CRS-1路由器



Load Zone 1

Load Zone 2

Load Zone 3

思科CRS-1 8槽线卡机前面和后面平面图

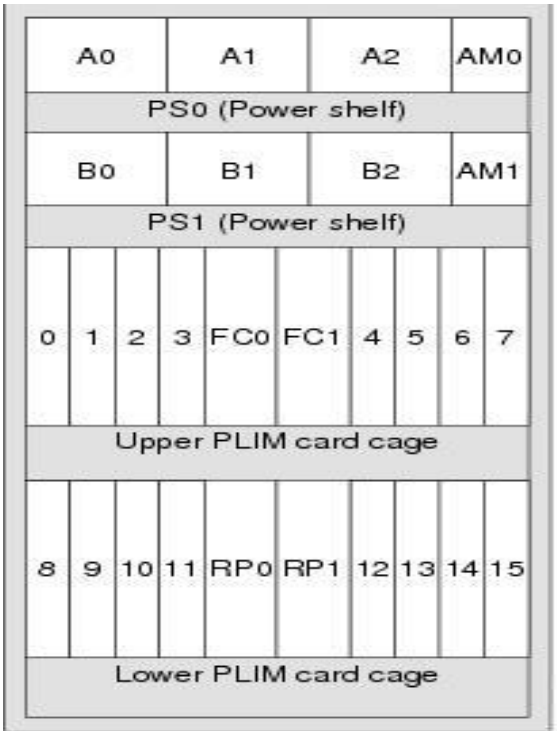
PLIM:就是端口卡，或者接口模块
RP：控制平面RP卡
MSC：配有两个SPP的模块服务卡。
SFC：Switch Fabric卡

16槽线卡机

机箱的前面（Front），总共有20个卡槽。其中：
上半部分：
×8个40Gbps的端口卡槽（或叫做接口模块）
×2个机箱控制卡
下半部分：
×8个40Gbps的端口卡槽（或叫做接口模块）
×2个控制平面的RP路由处理器卡。
×16个与端口卡对应的MSC卡和8个Fabric卡，在机箱的后面（Rear），插入中间板。从而与其他在 前面的20个板子连接，形成一个路由器系统。
下面是一个满载的16槽线卡机的例子：

思科16线卡槽CRS-1路由器

下面是CRS-1 16槽线卡机的前面和背面的平面示意图：



思科CRS-1 16槽线卡机前面（正面）示意图。
其中A0,A1,A2,AM0是电源模块。B0,B2,B2,AM1是另一组电源模块。

其他卡槽为16个PLIM（端口模块卡）

FC0和FC1为机箱风扇控制卡（Fan Controller）。

RP0和RP1为控制平面—路由处理器卡。

思科CRS-1 16槽线卡机的后面（背面）示意图。

共有16个MSC模块服务卡。

8个Fabric交换卡。

SM0-SM7.

对于CRS-1线卡机产品系列，所有的Fabric交换卡都是在机箱的后面。
1.通过内部的中间板连接内部系统的其他模块板卡。
2.通过光纤Cable进行CRS-1机组互联。（因此，当然不能把大把的光纤线缆暴露在机箱的正面。否则很难看）

5. 交换机配置

CRS-1产品系列中一个重要的组成部分就是其CRS-1的交换机FCC--Fabric Card Shelf。FCC产品的目的是通过提供一个或者多达8个FCC的交换机组，从而可以把2个，多达72个CRS-1线卡机（Line Card Shelf，简称LCC）互联起来，形成8个并行的高速交换平面（Switching Plane）。从理论上，CRS-1 多机箱技术（Multi-Shelf）可以提供高达92Tbps的报文交换容量。这种互联结构在工程实现上，机箱管理（Chasis Management)，系统软件上都很复杂。在目前为止，思科只能支持16槽的LCC，通过1，2或者4个FCC的互联拓扑。换言之，读者可以认为 92Tbps交换能力的互联，以CRS-1目前的体系结构而言其实是一个单纯的理论值。如果真的是72个LCC通过8个FCC互联，系统能否支撑（Sustain），笔者抱着谨慎乐观的态度。目前没有任何消息来源能证明思科自己内部是否做过这样的互联。其工程量非常大。能把所有的光纤正确的接对就 相当优秀，更不需要说IOS-XR的各种板卡的初始化，数据管理等更复杂的问题。

FCC本身是一个满载可以配备24个交换卡的机箱。在FCC的术语里面，这个交换卡称为S2卡。在以后的CRS-1互联的章节里，笔者会介绍S2卡的更多细节。从交换机配置的角度，FCC的配置是：

×正面24个S2交换卡。40Gbps的进出。上下两层。每层12个卡槽。

×正面2个机箱管理控制卡（Shelf Controller Gigabit Ethernet Card，简称SCGE）。

×后面24个OIM(Optical Interface Module)卡。

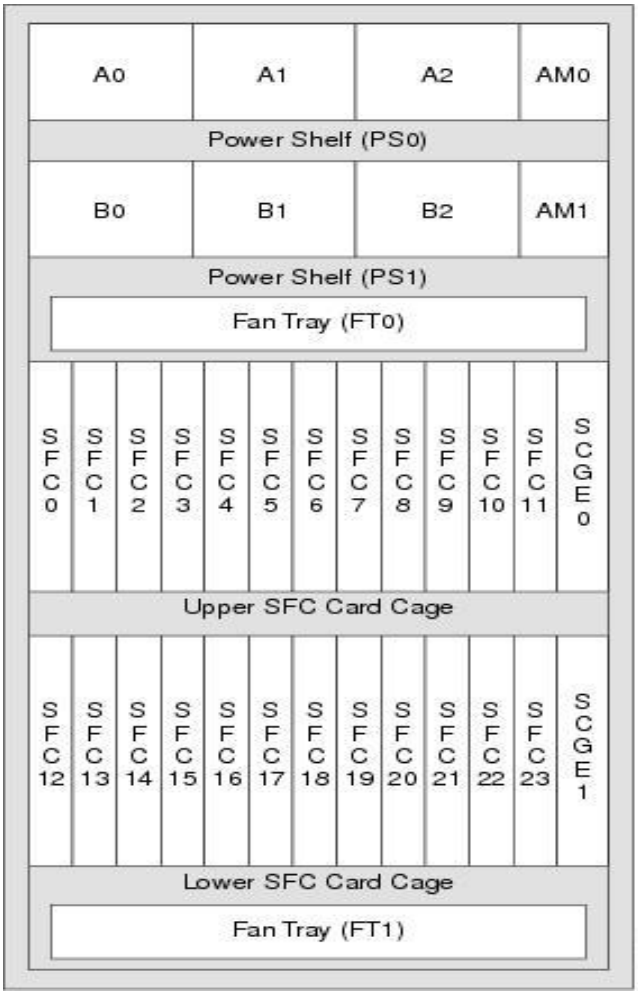
×后面2个OIM监视卡（OIM-LED）。

其中，后面的OIM卡就是用来互联LCC线卡机上的交换卡的。换言之，CRS-1机组互联是通过LCC交换卡上的适配器，然后通过光纤，连接到 FCC的OIM卡上的适配器或者接口上，而形成互联网络的。在LCC与FCC互联时，例如16槽LCC，其后面的交换卡是一种叫做S13的交换卡。而 LCC单独作为路由器时，其交换卡是S123交换卡。这是两个不同的物理卡。读者目前只需要了解在互联时，LCC要把S123的交换卡变成S13的卡就可 以了。

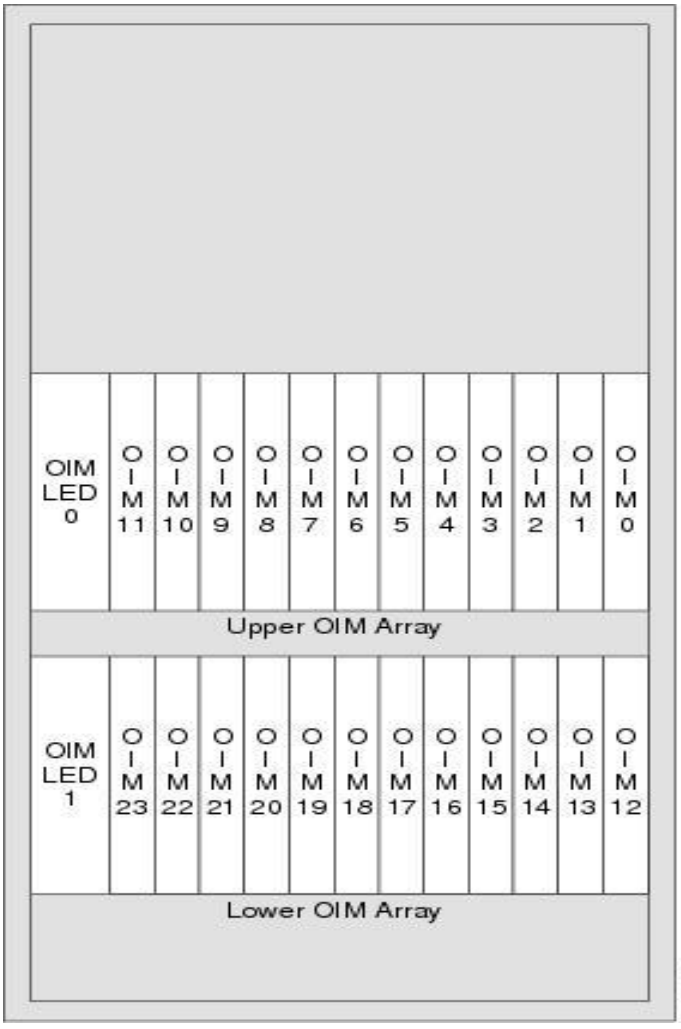
机箱管理控制卡SCGE是CRS-1系统，例如通过LCC上的控制平面RP来启动，管理，监测FCC的唯一途径。SCGE卡分为2种类型--2个GigE端口的SCGE和22个端口的SCGE。这里最大的区别不是端口数目，而是层2的switching功能。

如果一个系统配置的是2个端口的SCGE，系统需要另外配备思科的Catalyst switch。LCC的RP通过Catalyst交换机解决SCGE，从而达到控制FCC机的目的。而如果是最新的22个端口的SCGE，就不需要另外配 备Switch了。LCC的RP可以通过GigE以太端口直接连接SCGE。

下面是FCC的正面示意图：



下面是FCC的背面示意图：



OIM卡，用来与LCC的S13交换卡互联，形成CRS-1路由器阵列

一共24个OIM卡。每个卡上有9个OIM接口（适配器），用于光纤接入。

OIM LED卡用来监测OIM卡的状态。

6. 交换矩阵与交换平面

CRS-1最重要的部分可以说就是其交换矩阵和拓扑。交换矩阵把CRS-1的各个组成部分有机的结合在一起，形成一个高速的路由器。CRS-1的交 换是CRS-1体系结构里略微复杂和难理解的部分，下面分别对LCC和FCC的交换矩阵和拓扑做一些概念上的解释和阐述。希望读者能通过阅读相应章节之后 能从整体上对CRS-1的交换空间有一个清晰的把握。

首先，读者要了解CRS-1交换矩阵（Switch Fabric），交换平面（Switch Plane），交换卡（Switch Fabric Card）和之间的关系。本节介绍交换矩阵和交换平面，这两个逻辑实体。交换卡属于物理实体（实现），将在下一节更详细的介绍。

× **CRS-1交换矩阵（Switch Fabric）**： 交换矩阵由8个或者4个交换平面所组成。其中，16槽和8槽LCC：8个交换平面；4槽LCC：4个交换平面。交换平面之间是并行，没有依赖关系的。换言之，这些交换平面的关系是正交的。

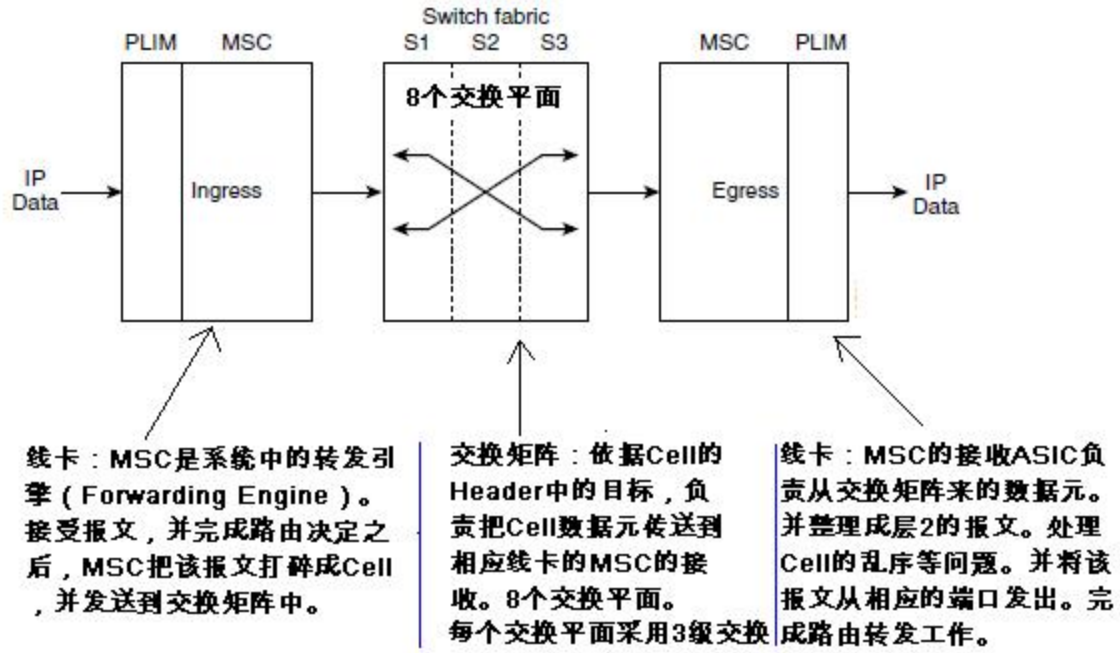
交换矩阵中通信的粒度是大小固定的数据元（Cell）。一个数据报文(Packet)在经过交换矩阵之前，线卡的Ingress MSC卡会把该报文分解为多个数据元。MSC会决定一个数据元被发送到哪个交换平面，并通过这个交换平面到达目标Engress MSC（这个MSC会汇总属于该报文的所有Cell，并恢复成一个Packet, 并通过端口卡（PLIM）上相应的端口（Port）把报文转发出去。一定要注意的是：属于同一个报文的不同的数据元可以（其实是通常）是通过不同的交换平 面抵达目的线卡的。但一个特定的Cell会而且只会通过一个选定的交换平面。在CRS-1，Ingress线卡上的MSC通常是通过轮询 （RoundRobin)的算法来决定使用哪个交换平面的。这里面的原因很简单直接，可以使得这8个或4个交换平面的负载均匀化，而避免某一个交换平面产 生阻塞丢掉数据元的情况。

× **CRS-1交换平面（Switch Plane）**：交换平面是交换矩阵的最小单位。一个交换平面能够实现了一个完整报文交换的功能。其中，每个交换平面采用3级（Stage）交换的体系结构。每个交换平面是通过在LCC机箱后面的交换卡来物理实现的。换言之，每个交换卡就是一个交换平面。

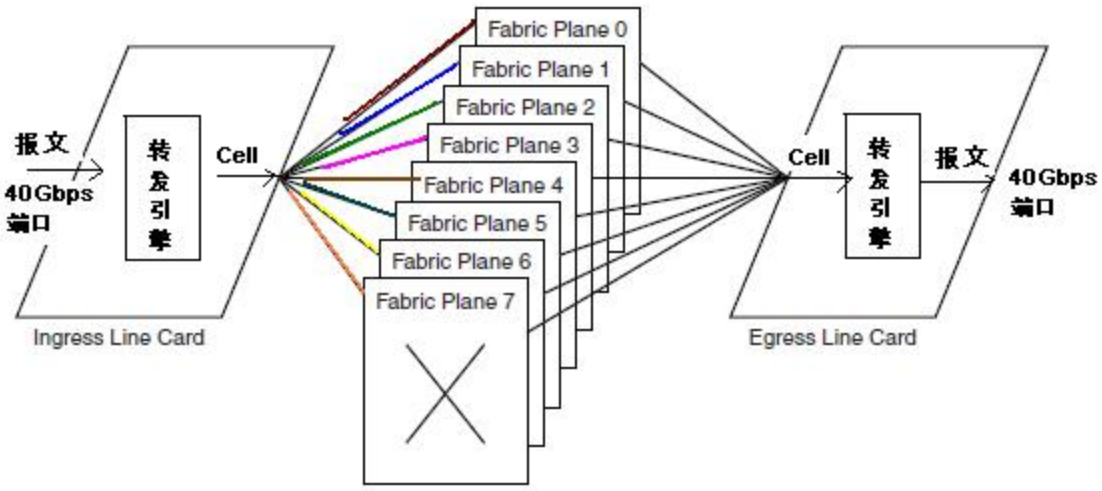
CRS-1系统中，理论上只需要一个交换平面就可以实现CRS-1路由器的转发交换功能。在CRS-1的实际运营上，系统要求至少两个交换平面来保 证系统路由转发交换。当然，这时系统已经不是线速的了。换言之，交换的能力不能支持众多40Gbps的线卡了。CRS-1的交换平面的交换率可以使得7个 交换平面就可以保证CRS-1系统的线速。

读者阅读到这里的时候，一定要深刻的认识如下的断言：**CRS-1系统中的任何（Each）一个线卡，通过中间板（midplane），可以连接到任何（Each）一个交换平面上。换言之，每个交换平面上，有来自所有线卡的输入。 每个交换平面上，都有通向任何一个线卡的输出。**

下图是CRS-1交换矩阵和交换平面的逻辑示意图。读者要注意的是，在谈论交换矩阵和交换平面的时候，都是一些逻辑概念和（或）实体。与具体的物理实现是无关的。或者说，可以通过许多技术方式来实现上述的交换矩阵和其组成部分交换平面。



CRS-1的交换矩阵示意图

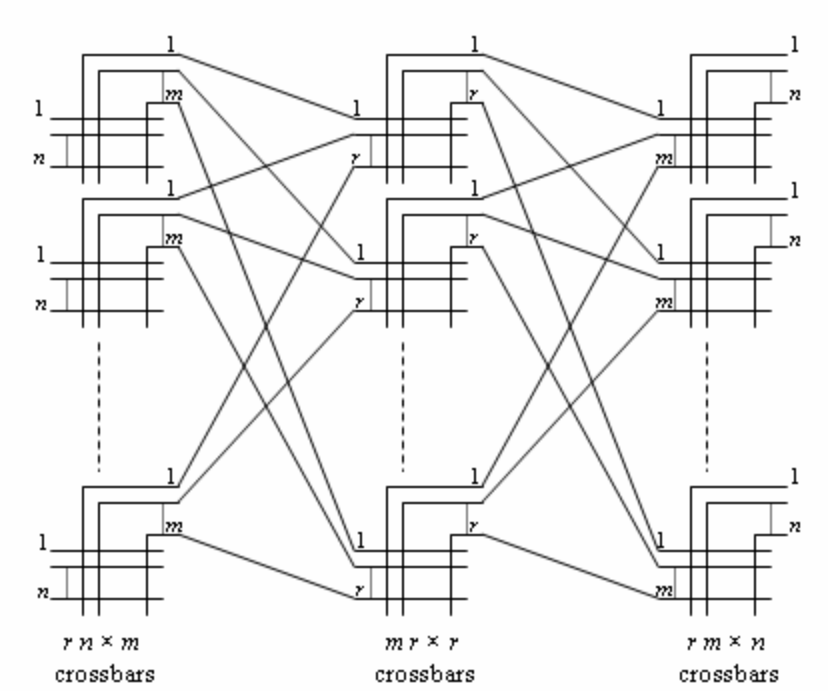


CRS-1交换矩阵有8个交换平面组成。同一个报文的Cell数据元可以通过不同的交换平面被传送到同一个目的线卡上。线卡负责整理Cells并形成完整的报文，并通过端口发出，离开CRS-1. 上图中的Ingress线卡与Engress线卡可以是同一个物理线卡。转发引擎（MSC）决定一切。

7. 物理交换卡（1）

CRS-1交换矩阵（Switch Fabric），交换平面（Switch Plane），交换卡（Switch Fabric Card)之间的关系是一个从属关系。前面两个是逻辑实体。交换卡属于物理实体（实现）。

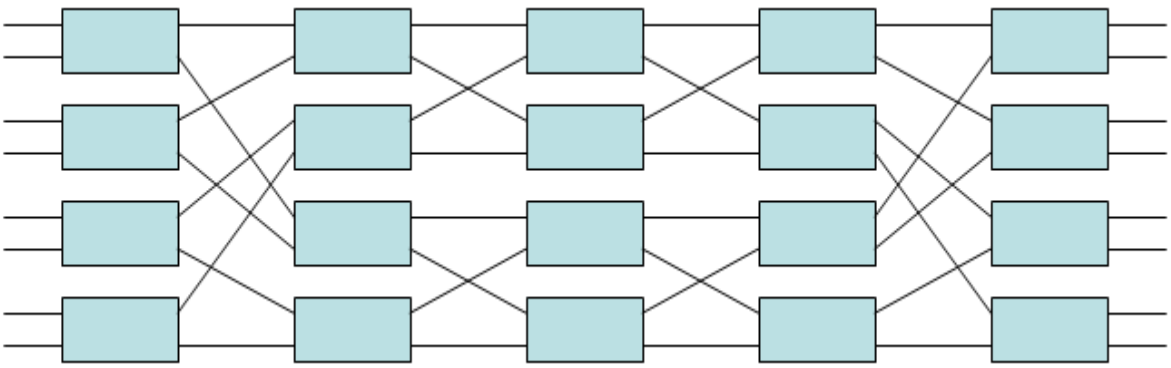
交换矩阵通过多个交换平面所组成。每个交换平面通过物理交换卡来实现。对于不同的CRS-1的LCC路由器，具体的数据关系如下：
16槽LCC：
交换矩阵：：==8个交换平面
1个交换平面：：==1个物理交换卡
【笔者注：16槽LCC配置8个交换卡槽。一个交换卡提供一个交换平面】
8槽LCC：
交换矩阵：：==8个交换平面
1个物理交换卡：：=2个交换平面
【笔者注：8槽LCC可配置4个交换卡槽。换言之，一个交换卡提供两个交换平面】
4槽LCC：
交换矩阵：：==4个交换平面
1个物理交换卡：：=1个交换平面
【笔者注：4槽LCC可配置4个交换卡槽。一个交换卡提供一个交换平面】
从上述数据读者可以得知。一个逻辑上的交换平面是通过一个（16LCC和4LCC）或者半个物理交换卡（8LCC）来实现的。
对于不同的LCC上的物理交换卡，其名称分别为：
16LCC：S123卡
8LCC： HS123卡
4LCC：QS123卡
CRS-1的交换不是一个全Cross-bar的结构，而是一个3级（Stage）Benes Network交换结构。关于Benes Network或者其更一般的交换结构拓扑Clos Network。简单而言，Close Network是一个通过3个参数（r,n,m)定义的一个3级交换拓扑，其3级结构分别为：输入（Ingress Stage），中间（Middle Stage）和输出（Engress Stage）。下图所示为一个通用的Cros网络拓扑模型：



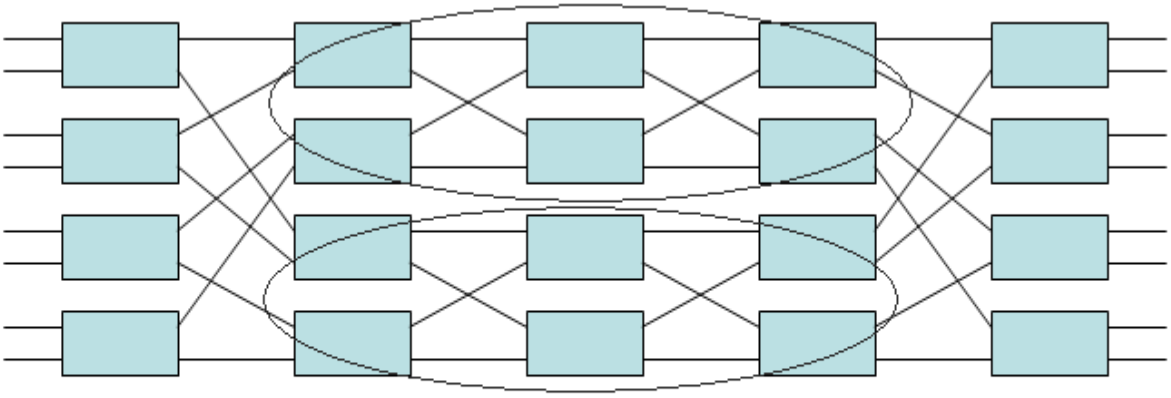
其中，r是输入（Ingress Stage）部件的个数。n是每个输入部件的输入接口数目。m是每个输入部件的输出接口数目。在理解Cros拓扑结构时，读者要抓住一个非常重要的概念：对于整个拓扑结构中的每个子部件，其是一个Cross-bar。例如，对输入而已，是有r个n×m个Crossbar。因为是r个输入部件，输出是m个， 因此，中间阶段一定是m个部件。为了形成cross-bar，很自然是r个输入，r个输出（除非做加速)。这也就是为什么中间阶段是m个r×r个 Cross-bar。在最后的输出阶段（Engress Stage），系统是一个逆转过程，通过r个m×n的cross-bar完成系统最后的交换过程。

关于Cros（Benes)拓扑的算法细节和量化分析，读者可以参阅相关文献。总而言之，Cros网络最大的优点是：相对一个没有中间交换过程的Cross-bar结构，对于要实现一个n×n的全交换，Cros网络所需要的连接节点数目要小的多。

Benes交换拓扑是Cros交换拓扑的一个特例，m=n=2。也就是说，在Benes交换拓扑中，每个Ingress和Engress子部件都是 一个 2×2的Cross-bar。读者要注意的是中间交换阶段（Middle Stage）是2个r×r的Crossbar。如果r是4，那就是2个4×4的crossbar。如果是16，那就是2个16×16的crossbar。 下图所示为一个8×8的Benes交换拓扑。



读者请注意，笔者说的2个4×4 的Crossbar是Cros拓扑的概念。在上述图中，就是中间3个Stage，共12个2×2的Crossbar。上半部分的两行（6个2×2的 crossbar）组成了一个4×4的Crossbar。下面两行（6个2×2的crossbar）组成了另外一个4×4的Crossbar。

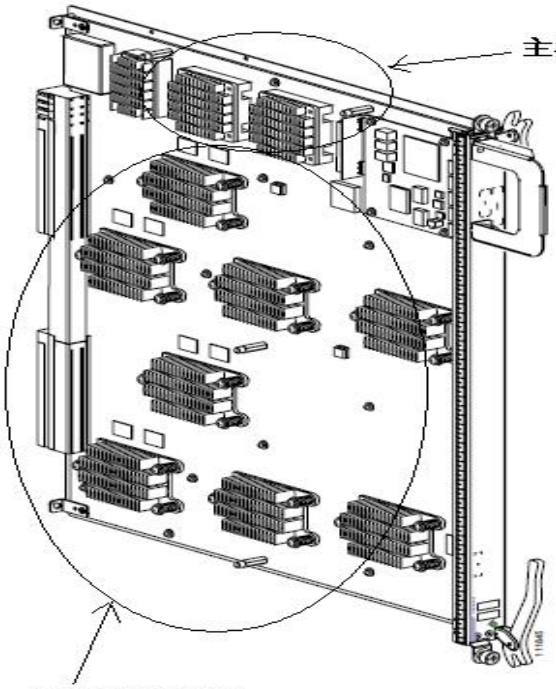


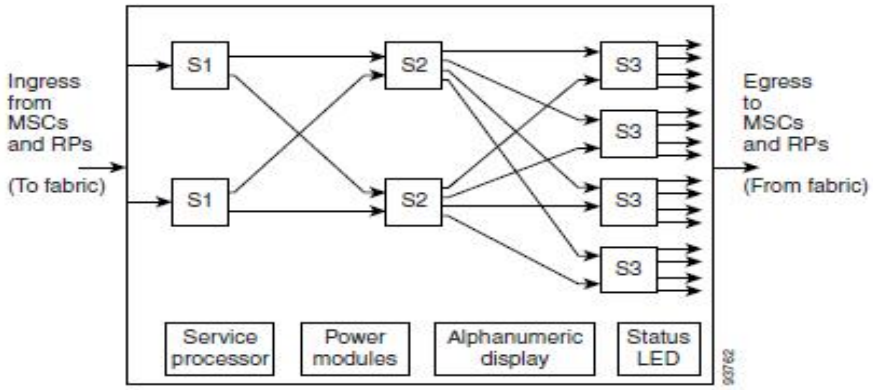
从3级结构Clos网络拓扑而言，中间是2个4×4的Crossbar
。 m=n=2. r=4.

CRS-1的物理交换卡就是基于上述结构，而通过物理卡上的多个ASIC芯片而实现的。

8. 物理交换卡（2）

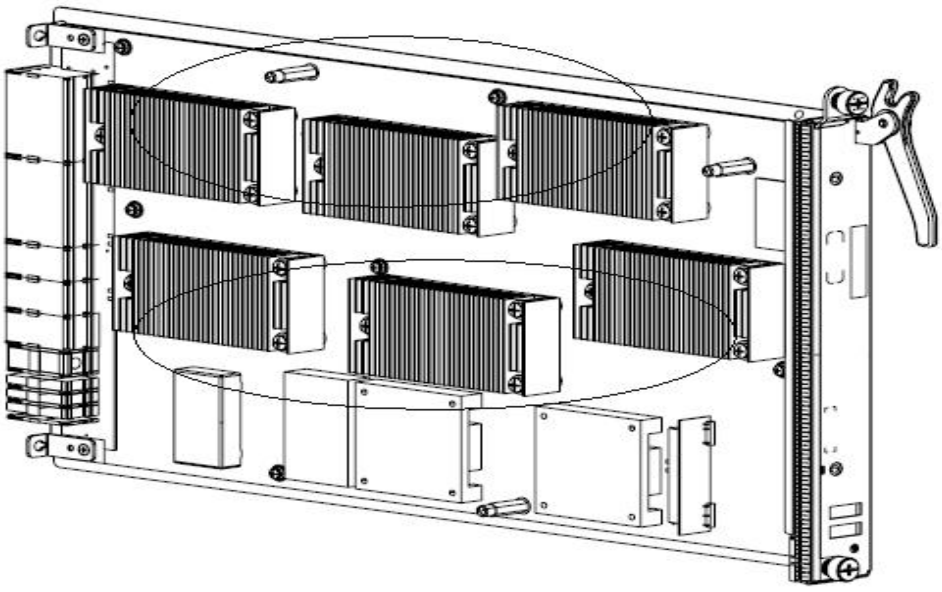
CRS-1 LCC路由器物理交换卡（Fabric Card）的概念模型是一个基于Benes Network拓扑的3级交换网络。在物理实现上，交换卡是通过卡上的多个ASIC芯片来完成3个阶段，基于固定大小Cell单元的数据交换。下面是一些基本的数据。
对于16槽的LCC，其交换卡（S123)上含有：2个S1，2个S2和4个S3的ASIC芯片，完成1个交换平面的功能。
对于8槽的LCC，其交换卡（HS123)上含有：2个S1，2个S2和2个S3的ASIC芯片，完成2个交换平面的功能。
对于4槽的LCC，其交换卡（QS123)上含有：1个S1，1个S2和1个S3的ASIC芯片，完成1个交换平面的功能。
下图所示分别为相应的物理交换卡:



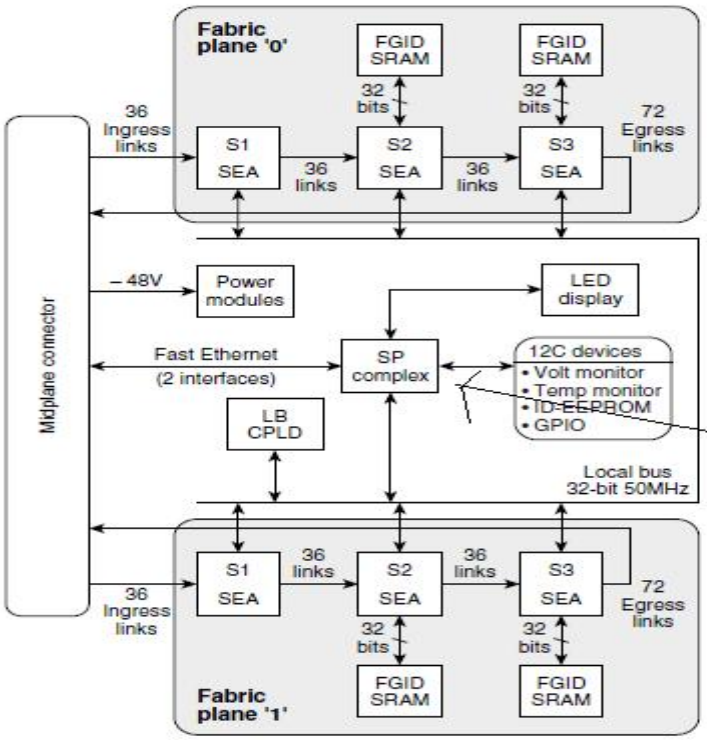


Each stage of the three-stage Benes switch fabric is implemented with the same switch element components. However, during system startup the components are programmed by Cisco IOS XR software to operate in S1, S2, or S3 mode, depending on their functions in the switch fabric. Each S123 switch fabric card contains two S1, two S2, and four S3 components.

注：16LCC物理交换卡芯片粒度概念图。对于8个交换芯片，都是一样的，并没有S1，S2和S3的区分。只是在操作系统初始化的时候才被选择担当不同的交换阶段的功能。



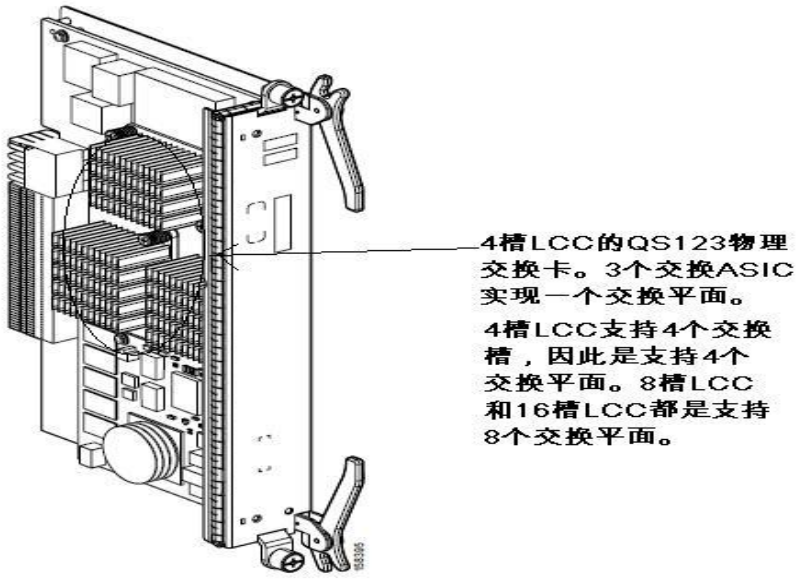
8槽LCC物理交换卡，6个交换ASIC，完成两个独立的，毫不相干的交换平面。每个交换平面有1个S1，1个S2和1个S3 ASIC芯片组成。



每个交换平面的S1 ASIC输入是36个2.5Gbps的Link。这一点非常重要。

板子上的主控CPU

8槽LCC的HS123物理交换卡，通过6个交换芯片，分为2组，实现两个独立的交换平面。



9. 物理交换卡（3）

在上述两节描述的物理交换卡（QS123，HS123和S123）是而且只是给CRS-1的独立路由器LCC的，例如，4槽，8槽或者16槽LCC。其实细心的读者可以从命名约定都可以得知，这3种交换卡的123后缀意味着在一个物理交换卡上，已经完成了交换平面的3级交换的各个阶段（Stage）。每个交换卡提供而且只提供系统8或者4个（对于4槽LCC而言）交换平面中的一个平面。

在CRS-1家族中，还有另外2个物理交换卡。其名称为S13卡和S2卡。这是对于CRS-1基于FCC的多机互联（LCC+FCC）的系统而专门使用的物理交换卡。

换言之，S13卡和S2卡是而且只是被用在基于FCC的多机互联系统中。请注意，笔者非常强调“基于FCC的”多机互联。其蕴含的信 息是，如果是单纯的多机LCC互联，例如双机16槽LCC的HA（Active/Passive）通过1G或者10G的数据端口互联，是不需要，也与 S13卡和S2卡毫不相干的。双机HA互联的结构，其实是一种容错结构。总体而言，是两个独立的路由器，只是在软件层面做NSR和ISSU等热备份工作。

CRS-1的FCC的多机互联的本质是：一个路由器！ 多个LCC加上一个或者多个FCC形成的一个系统是而且只是一个路由器。这个“只是一个路由器”既是逻辑上的理解，更重要的，也是物理上或者实际上的理解。

上述断言是对CRS-1系统非常重要的，希望读者一定要清楚的把握。一次性的把概念切入，从而避免混淆。

在一个路由器的前提下，理解下面要讲解的S13卡和S2卡就变得容易了。

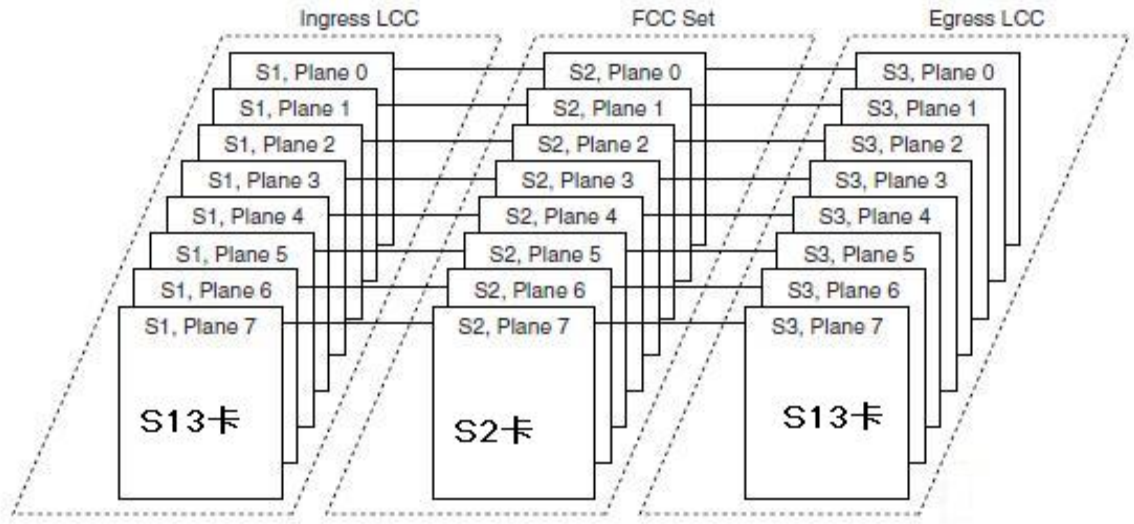
S13物理交换卡：

当16槽LCC用于FCC多机互联时，必须将单机时配置的S123卡拔出并替换掉。要插入S13交换卡。S13交换卡是CRS-1硬件系列中唯一能 与FCC通过光缆互联的设备。在S13物理交换卡上，一共有6个交换ASIC芯片。其中2个完成S1阶段，4个完成S3阶段。读者比较S123卡，其实就 可以知道，其实就是去掉了两个S2阶段的ASIC交换芯片。

在FCC多机互联的CRS-1系统中，LCC配置的是S13卡，而非S123卡。S13卡，顾名思义，具有3级交换的阶段1（Stage 1）和阶段3（Stage 3）的功能。
那么谁来完成基于3级交换的交换平面的阶段2的功能呢？
一个单独的物理交换卡--S2物理交换卡。
S2物理交换卡：
在S2卡上，含有6个交换ASIC芯片，也即S2芯片。S2卡是用而且只被用在24槽的FCC上。
到目前为止，基本上介绍了单机LCC系统下的交换矩阵，交换平面和相应的物理交换卡的概念，关系和具体的物理映射等。也介绍了在FCC多机互联下，CRS-1需要的另外两个特殊的物理交换卡--S13和S2卡。
笔者会在下节“FCC多机互联”中阐述：在FCC多机互联的体系结构中，如果延伸和映射交换矩阵，交换平面和物理交换卡这3个CRS-1重要的概念和含义 。 其实，忘却细节，读者如果能够理解笔者如下断言，就基本上把握住CRS-1的交换核心了：**LCC系统的交换是一个紧耦合系统（Tighly coupled System）。FCC多机系统 的交换是一个松耦合系统（Loosely coupled System）。**

10. FCC多机交换

FCC多机交换是LCC单机交换在逻辑上和物理上的无缝扩展。笔者用无缝（Seamless）这个词 想表达的意思就是，在CRS-1系统中，交换矩阵和交换平面，对这两个逻辑上的概念和实体，在LCC单机交换和FCC多机交换上，是一致的。唯一有区别的 是，LCC单机交换是各个线卡，交换卡都是插在并通过中间板（Midplane）进行连接的。而FCC多机交换，是把交换光缆（Switch Cable）通过LCC的S13交换卡的光缆接口与FCC的OIM（Optical Interface Module）光缆接口卡的光缆接口互联，从而组成交换矩阵和相应的8个交换平面。FCC机箱后面的OIM卡与FCC机箱正前方的S2物理交换卡是通过 FCC的中间板（Midplane）达到互联的。每个LCC的S13卡有3个光纤接口。每个FCC的OIM卡有9个光纤接口。一个FCC最大可以有24个 S2卡和相应的24个OIM卡。
CRS-1系统目前只支持16槽LCC的FCC互联。8槽LCC和4槽LCC不能支持FCC的互联。目前，CRS-1系统IOX最大配置容量可以支 持8个16槽LCC与1，2或者4个FCC互联。其中，2个或者4个FCC的目的是为了提高系统的容错性，例如可以通过配置把8个数据交换平面分布在2或 4个FCC上。但从理论和实践上而言，一个FCC的24个S2交换卡和相应的24个光缆接入卡OIM已经可以足够支持8个16槽LCC的FCC互联了。
下图是FCC互联的交换矩阵和交换平面逻辑图：



在FCC互联交换中，每个LCC的8个S13交换卡分别提供整个系统8个交换平面的S1和S3阶段。与LCC单机交换矩阵一样，一个S13卡属于而且只属于8个交换平面中的一个。任何一个数据单元（Cell）一旦交换，不会跨越交换平面。要么成功要么扔掉。一个数据单元通过S13卡的S1 ASIC芯片，然后通过光缆线，进入FCC的OIM卡，然后通过FCC的中间板，进入FCC的S2卡。S2卡再将该Cell转发给某个LCC的S13卡。从而被该S13卡上的S3 ASIC芯片介绍该数据单元。要注意的是，这个接收S13卡完全可以是当初该数据单元发送时在S1阶段的那个发送S13卡。道理就类似于Loopback一样。但是任何数据必须通过S2卡走一遍。不能是从一个S1 ASIC直接交换到同一个S13卡的S3 ASIC芯片上。

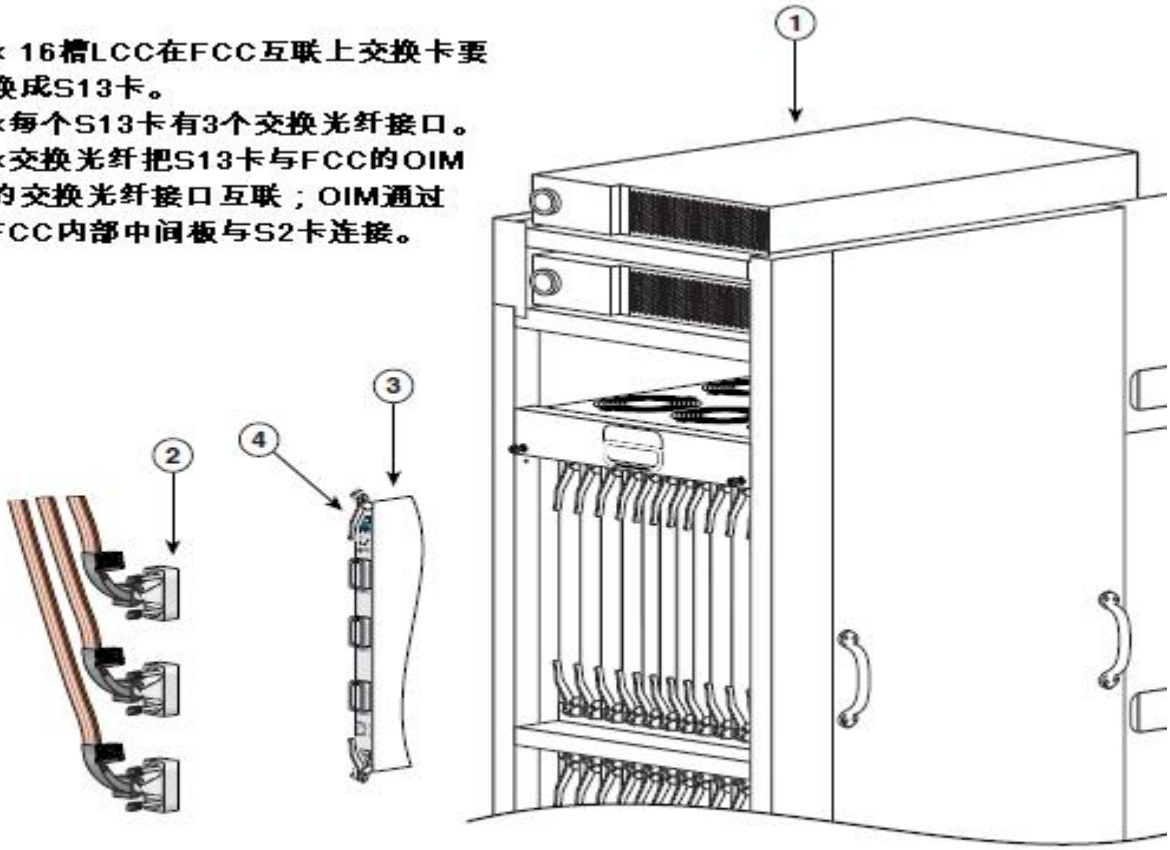
在上节中，我们介绍过S13卡和S2卡的物理配置情况。对于每个交换阶段，都是多个ASIC来负责承担。例如，6个ASIC在S2卡上。6个 ASIC在S13卡上（2个做S1；4个做S3）。同一个阶段的多个ASIC的目的和使用机制就是为了均匀负载，例如通过轮询使用算法。 而绝非是流水线的工作方式。

对于上述FCC交换矩阵和交换平面的理解中，有一点需要特别把握：

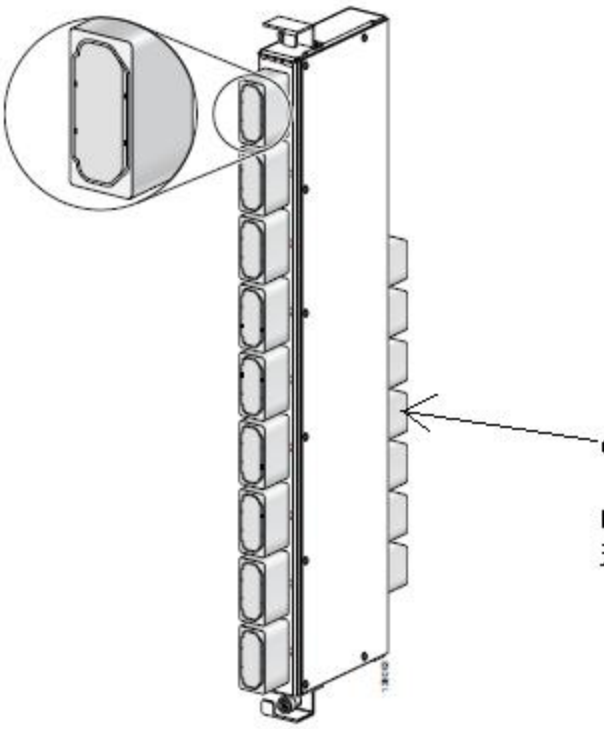
×一个交换平面可以是由有一个**S2**卡提供。也可以是通过**3**个**S2**卡来提供。换言之，笔者在上图中所标注的**“S2**卡**”**是一个逻辑实体。其可以是 一个**S2**物理卡，也可以是**3**个**S2**卡。目前为止，读者只要关心和认为这一点：每个交换平面，在**FCC**方面，是一个**S2**卡。关于为什么存在多个**S2**卡来提供 一个交换平面的原因和技术细节，会在将来的章节中解释。这些貌似复杂的变化其实都取决于**IOX**对系统的配置和现场多机互联的拓扑架构。

下面是LCC S13卡，FCC OIM卡光缆接口示意图：

× 16槽LCC在FCC互联上交换卡要换成S13卡。
×每个S13卡有3个交换光纤接口。
×交换光纤把S13卡与FCC的OIM的交换光纤接口互联；OIM通过FCC内部中间板与S2卡连接。



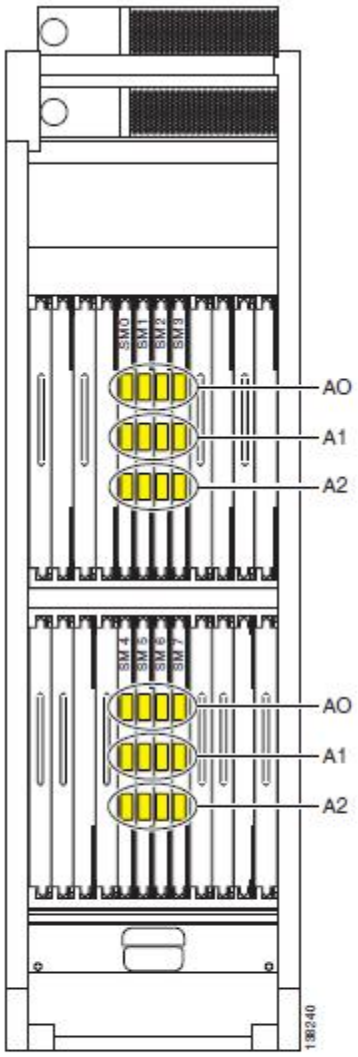
1	Line card chassis	3	S13 fabric card (FC/M)
2	Optical array cable connectors	4	S13 bulkhead array adapters



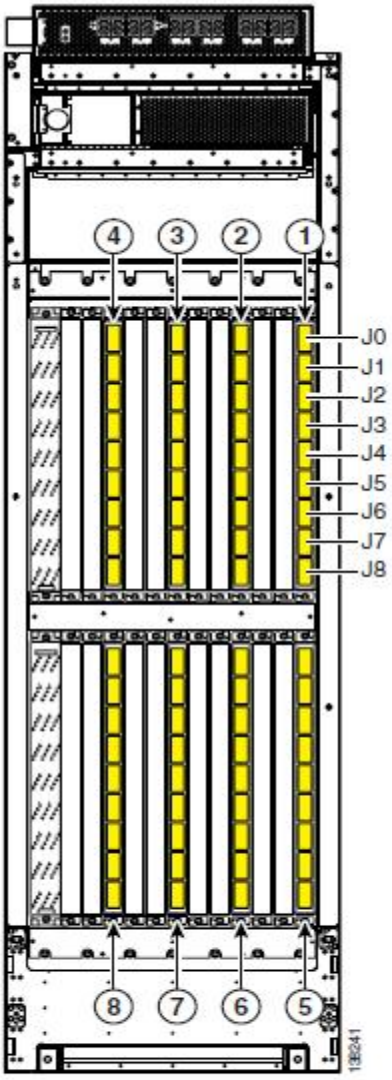
OIM交换光缆卡与FCC的中间板（Midplane）内部接口。从而FCC机箱正前方的S2交换卡能通过互联收发数据单元CELL。

OIM交换光缆接入卡。提供9个交换光纤接口。

在一个S2交换卡负责一个交换平面的配置下，LCC S13卡的A0-2光缆依次接入在OIM的9个接口上。换言之，一个OIM卡可以接入3个S13卡的交换光缆。



16槽LCC，配有8个S13卡，用来FCC互联。每个13卡有3个交换光缆接口

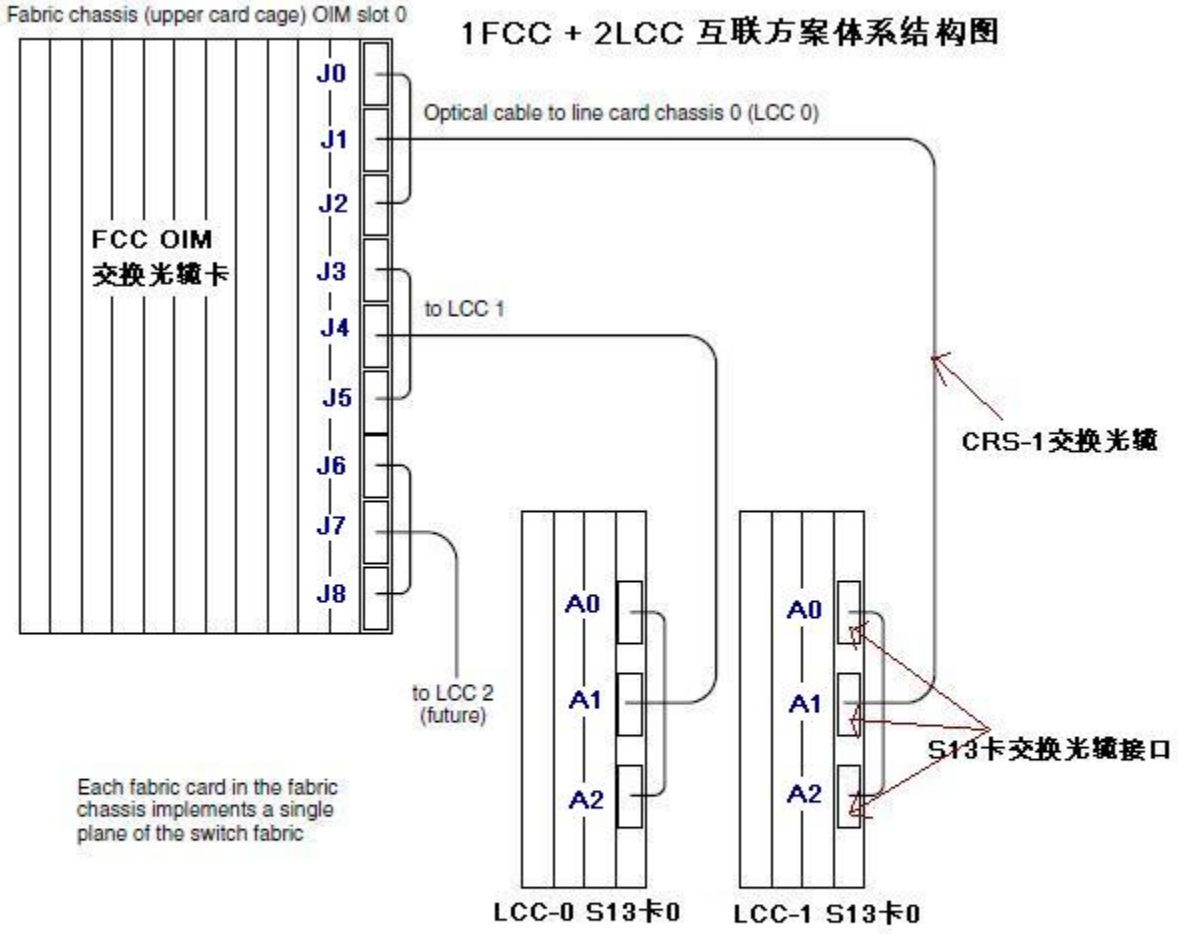


FCC。上图所示为配有8个S2交换卡和相应8个OIM卡。每个S2(OIM)提供一个交换平面。

在下面的章节中，笔者会力图非常清晰的把2LCC+1FCC, 4LCC+1FCC, 4LCC+4FCC等的拓扑解释给读者，一览CRS-1的FCC多机交换的奥秘。笔者会首先解释一个交换平面就是一个S2交换卡的情况。

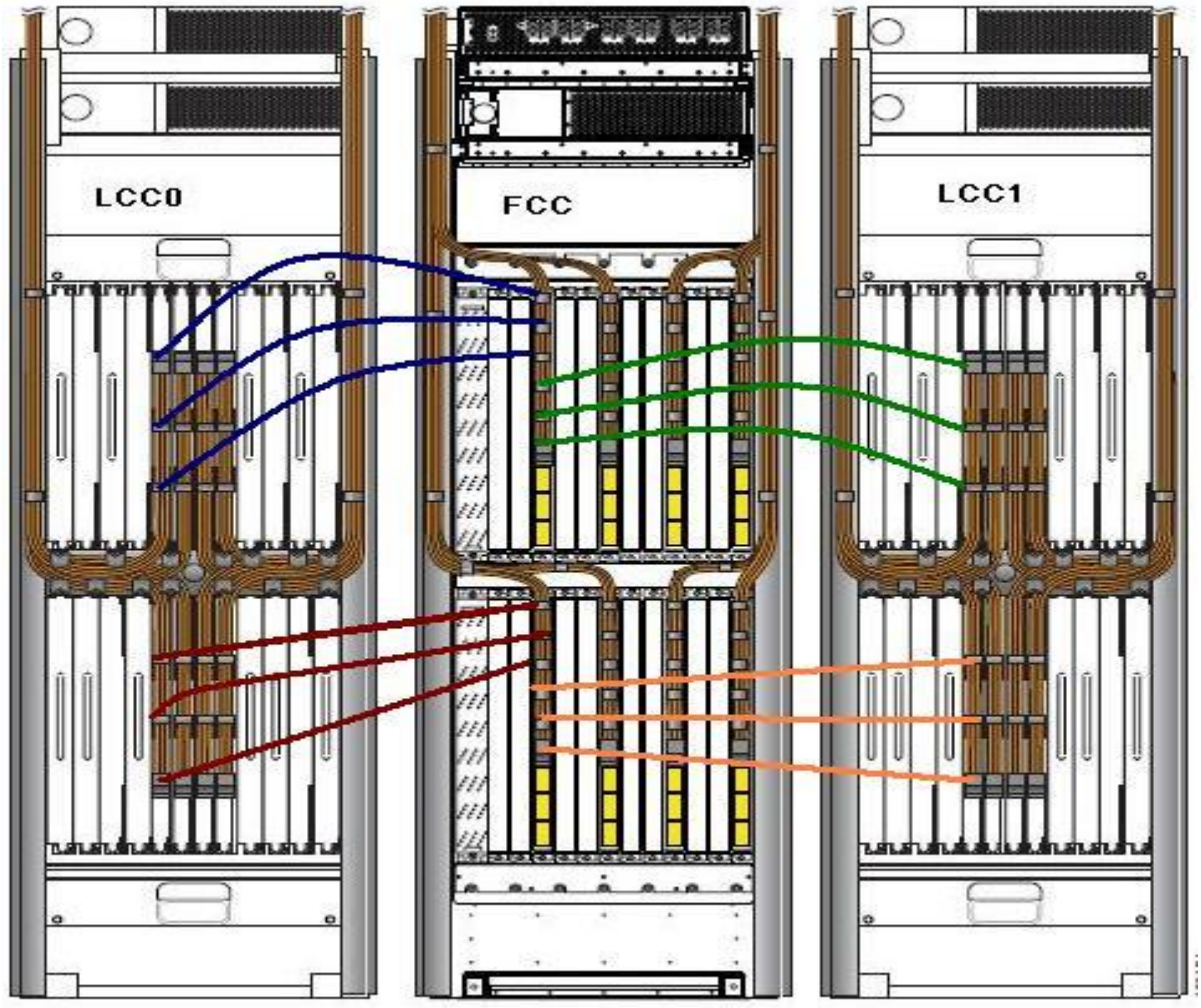
11. FCC多机交换（2）

下图是1个FCC连接2个16槽的LCC的体系结构图。



LCC-0的S13卡0，LCC-1的S13卡0与FCC的S2卡0，通过FCC的OIM交换光缆卡和交换光缆组成系统中的一个交换平面。换言之，在这种配置下，每个S2卡完成一个交换平面。

对于上图的1FCC与2个LCC互联结构，一个OIM卡具备9个交换光缆接口。换言之，一个OIM卡可以支持3个S13卡的接入。原因很简单，每个 S13卡是3个交换光缆接口。一个OIM卡是与一个S2交换卡对应的。因此，一个S2交换卡可以把3个S13卡互联起来，组成一个交换平面。
下面是一个安装上述连接方案而形成的2LCC+1FCC互联的实例图。



CRS-1 2LCC+1FCC互联实例图。

在上面图中，系统中的16个S13交换卡（每个LCC有8个S13卡）通过交换光纤互联在FCC机箱后面的OIM卡的接头上。通过这样的物理互联，这个**CRS-1**多机互联形成**8**个松耦合的独立的交换平面。每个平面有一个**LCC0**的**S13**卡，一个**FCC**的**S2**卡和一个**LCC1**的**S13**卡组成。当然，还包含其中的光纤线缆。
在**LCC**机器中，任何一个线卡（**MSC**）可以通过中间平面（**Midplane**）连接到其本地**LCC**的**8**个**S13**卡的任何一个**S13**卡，数据可 以从线卡的**MSC**抵达任何一个**S13**卡。因此，在**FCC**互联的情况下，可以得知，任何一个数据单元从任何一个**LCC**的任何一个线卡，跨越互联光缆，抵达 **FCC**的**S2**交换卡，然后再抵达另外一个或者同一个**LCC**的一个**S13**卡的**S3**芯片 ，再抵达目的地线卡**MSC**。
图中FCC的OIM卡上黄色的部分是没有用的，空闲的3个互联光纤接头。

另外，图中标记的蓝，绿，红和橙色线段不是物理光缆，而是笔者标注的逻辑示意线段，用来表明S13卡，OIM卡之间的接头关系。真正的物理光缆是那 些褐色的，整整齐齐的物理光缆。例如，蓝色线表明LCC0的S13卡（编号：SM0）的光缆接头A0，A1和A2分别接在了FCC的OIM卡（编 号：OIM9）的J0，J1和J2上；绿色线表明LCC1的S13卡（编号：SM0）的光缆接头A0，A1和A2分别接在了FCC的OIM卡（编 号：OIM9）的J3，J4和J5上。红色线表明LCC0的S13卡（编号：SM7）的光缆接头A0，A1和A2分别接在了FCC的OIM卡（编 号：OIM21）的J0，J1和J2上；橙色线表 明LCC1的S13卡（编号：SM7）的光缆接头A0，A1和A2分别接在了FCC的OIM卡（编号：OIM21）的J3，J4和J5上。

读者可能会思考，FCC的OIM卡有24个。上下两排。对上图而言，上排，OIM卡/插槽用的是9，6，3，0（通过 J0，J1，J2，J3，J4，J5接口，依次链接两个LCC的0，1，2，3共2×4=8个S13交换卡。）。下排用的是12，15，18，21（通过J0，J1，J2，J3，J4，J5接口，依次链接两个LCC的4，5，6，7共2×4=8个S13交换 卡。）。

这是为什么呢？

原因是容错。为了FCC最大程度的避免单点失败（Single Point of Failure）。

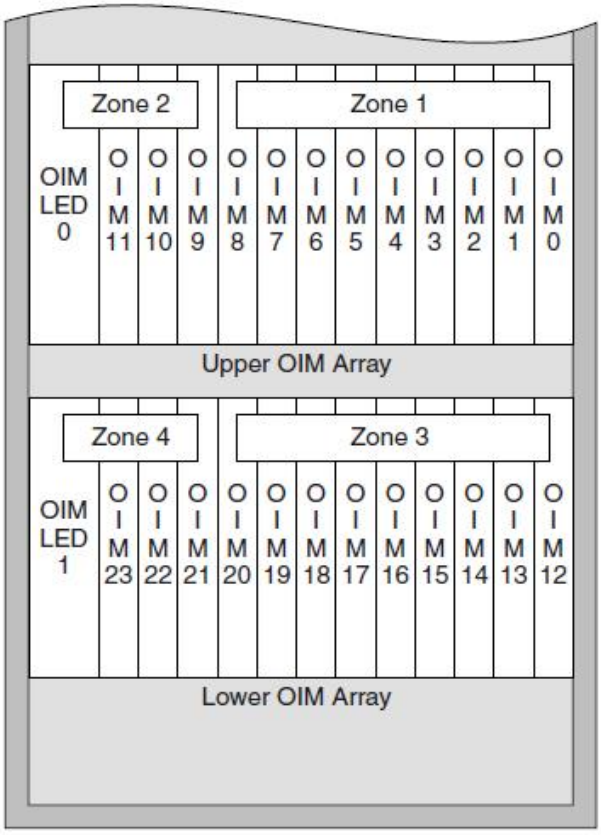
这里面涉及到FCC的电源系统。

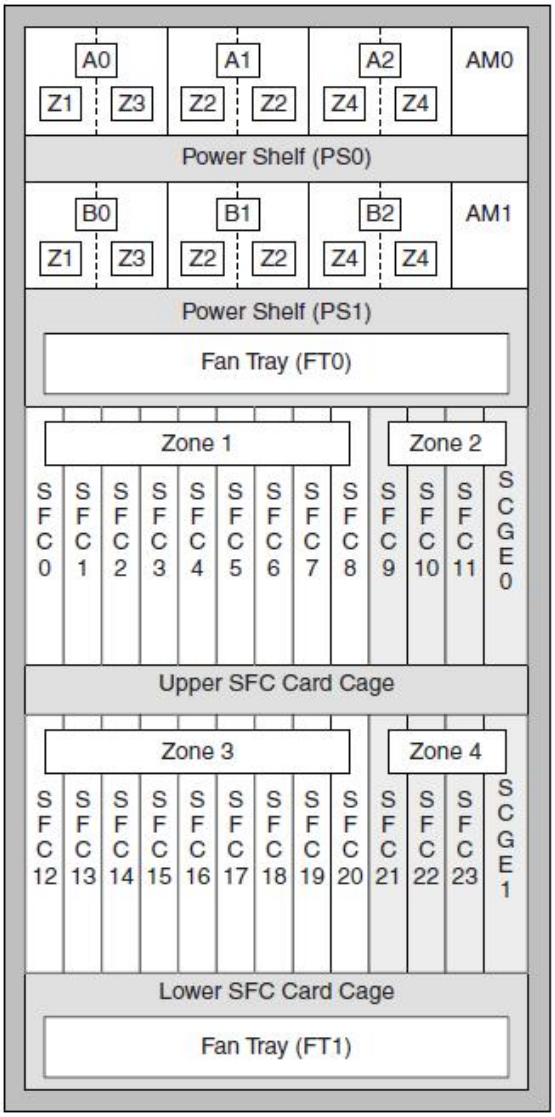
在FCC机箱中，电源系统是很复杂的一个部分。对于24个OIM卡而言，电源系统分为4个区域（Zone）。

对于电源而言，一个Zone的电源出错，就意味着该Zone内所有的OIM卡都失去了电源支持。

所以，作为系统架构设计，如果有可能，是要将OIM卡的使用分布在不同的电源区域，从而使得系统的容错性得到最大的体现。

下图所示为FCC对于OIM卡和S2交换卡的电源控制分布图：





FCC电源系统分区概念图。左图为FCC机箱背面OIM卡部分的电源4区域（zone）分布。OIM0—OIM8卡槽共享一个电源控制。OIM9-OIM11共享一个；OIM12-OIM20共享一个；最后，OIM21-OIM23共享另外一个电源控制。一旦某个电源控制失效，其管辖的所有OIM卡就断电不能工作。

右图为FCC机箱正面S2交换卡的电源控制图。同理，对于S2卡，分为4个电源控制区域。SFC0-SFC8一组；SFC9-SFC11一组；SFC12-SFC20一组。最后SFC21-SFC23共享一个电源控制。

12. FCC多机交换（3）

在上述两节描述的物理交换卡（QS123，HS123和S123）是而且只是给CRS-1的独立路由器LCC的，例如，4槽，8槽或者16槽LCC。其实细心的读者可以从命名约定都可以得知，这3种交换卡的123后缀意味着在一个物理交换卡上，已经完成了交换平面的3级交换的各个阶段（Stage）。每个交换卡提供而且只提供系统8或者4个（对于4槽LCC而言）交换平面中的一个平面。

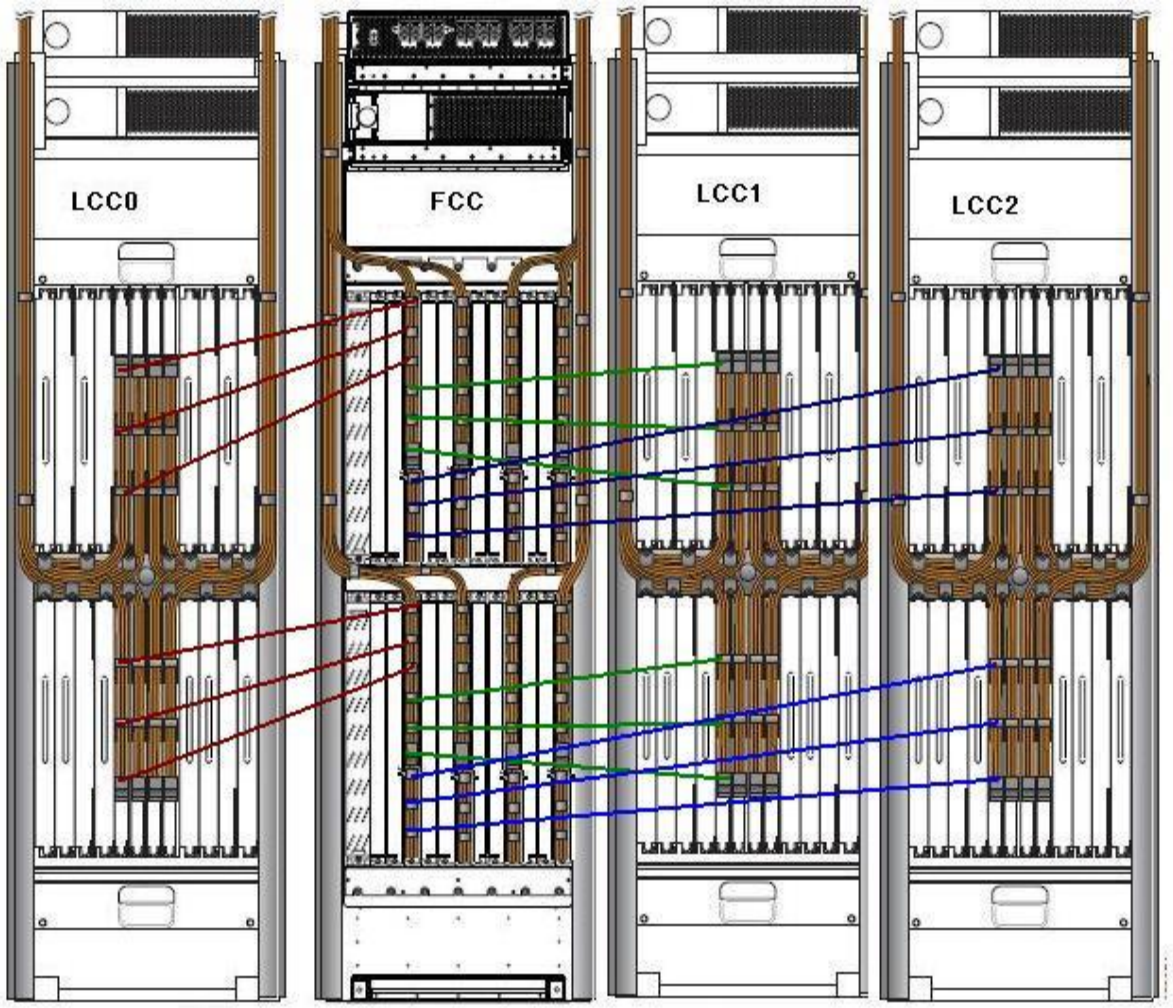
在**CRS-1**家族中，还有另外**2**个物理交换卡。其名称为**S13**卡和**S2**卡。这是对于**CRS-1**基于**FCC**的多机互联（**LCC+FCC**）的系统而专门使用的物理交换卡。
换言之，S13卡和S2卡是而且只是被用在基于FCC的多机互联系统中。请注意，笔者非常强调“基于FCC的”多机互联。其蕴含的信 息是，如果是单纯的多机LCC互联，例如双机16槽LCC的HA（Active/Passive）通过1G或者10G的数据端口互联，是不需要，也与 S13卡和S2卡毫不相干的。双机HA互联的结构，其实是一种容错结构。总体而言，是两个独立的路由器，只是在软件层面做NSR和ISSU等热备份工作。
CRS-1的**FCC**的多机互联的本质是：一个路由器！ 多个**LCC**加上一个或者多个**FCC**形成的一个系统是而且只是一个路由器。这个**“只是一个路由器”**既是逻辑上的理解，更重要的，也是物理上或者实际上的理解。
上述断言是对CRS-1系统非常重要的，希望读者一定要清楚的把握。一次性的把概念切入，从而避免混淆。
在一个路由器的前提下，理解下面要讲解的S13卡和S2卡就变得容易了。
S13物理交换卡：
当16槽LCC用于FCC多机互联时，必须将单机时配置的S123卡拔出并替换掉。要插入S13交换卡。S13交换卡是CRS-1硬件系列中唯一能 与FCC通过光缆互联的设备。在S13物理交换卡上，一共有6个交换ASIC芯片。其中2个完成S1阶段，4个完成S3阶段。读者比较S123卡，其实就 可以知道，其实就是去掉了两个S2阶段的ASIC交换芯片。
在FCC多机互联的CRS-1系统中，LCC配置的是S13卡，而非S123卡。S13卡，顾名思义，具有3级交换的阶段1（Stage 1）和阶段3（Stage 3）的功能。
那么谁来完成基于3级交换的交换平面的阶段2的功能呢？
一个单独的物理交换卡--S2物理交换卡。
S2物理交换卡：
在S2卡上，含有6个交换ASIC芯片，也即S2芯片。S2卡是用而且只被用在24槽的FCC上。
到目前为止，基本上介绍了单机LCC系统下的交换矩阵，交换平面和相应的物理交换卡的概念，关系和具体的物理映射等。也介绍了在FCC多机互联下，CRS-1需要的另外两个特殊的物理交换卡--S13和S2卡。
笔者会在下节“FCC多机互联”中阐述：在FCC多机互联的体系结构中，如果延伸和映射交换矩阵，交换平面和物理交换卡这3个CRS-1重要的概念和含义 。 其实，忘却细节，读者如果能够理解笔者如下断言，就基本上把握住CRS-1的交换核心了：**LCC**系统的交换是一个紧耦合系统（**Tighly coupled System**）。**FCC**多机系统 的交换是一个松耦合系统（**Loosely coupled System**）。

13. FCC多机交换（4）

CRS-1系统，其理论最大满负载是72个LCC的CRS-1多机互联。
在关于CRS-1端口配置中，我们谈到，72个满负载的LCC互联的CRS-1系统，其端口数目惊人，可以支持1152 OC-768c/STM-256 PoS 个端口，4608 OC-192c/STM-64c PoS/DPT 端口，或9216 10 Gigabit Ethernet 端口，18,432 OC-48c/STM-16c PoS/DPT 端口，1152 OC-768c/STM-256 tunable WDMPOS 端口，或者4608 10 Gigabit Ethernet tunable WDMPHY端口。

在上述这样一个**72**台**LCC**互联形成的路由器中，作为一个路由器，其最基本的一个要求是：确保一个数据报文（**Packet**）从任何一个线卡的任何一个端口，能够通过路由路径寻找，从这**72**个**LCC**的任何一个线卡的某个端口被**CRS-1**转发（**Forwarding**）出去。
在理论上，为了达到上述目标，1个FCC，2个FCC，。。。，8个FCC的情况下，如何将这72台LCC机器的交换卡互联？
这里面最关键的还是要牢牢抓住**CRS-1**支持**8**个独立的交换平面。**8**个独立的交换平面是**CRS-1**的纲，**LCC**等是**CRS-1**的目。纲举目张。
首先来回顾一下当FCC的每个**S2**交换卡（或者说**OIM**卡）作为一个交换平面的情况。
在2LCC+1FCC的情况下，每个LCC全负载，也就是说，上架8个S13卡（请参阅16槽LCC的交换卡槽结构与配置）。为了实现8个平面互 联，FCC要贡献出8个S2（OIM）卡。每个卡出任一个交换平面。每个LCC的S13卡都通过其光纤接口A0-A2分别依次接到相应的OIM卡的光纤接 口J0-J2和J3-J5上。OIM卡的光纤接口J6-J8闲置不用。
在上述配置下，任何一个LCC的线卡都可以通过其S13卡<-->A0-A2接口-<-->OIM卡<-->S2卡<-->S3卡来抵达其他的一个LCC机器的线卡。
上述拓扑下，系统升级，要加一台LCC，形成3LCC+1FCC，如何互联？
很简单，把这台新的LCC的8个S13卡的A0-A2分别接到FCC的8个OIM卡的J6-J8上。
上述拓扑下，系统再升级，要再加一台**LCC**，形成**4LCC+1FCC**，如何互联？
读者的直觉有可能是，一个FCC支持24个S2交换卡（24个OIM卡）。在上述拓扑下，FCC只用掉了8个S2（OIM）卡，还有16个呢。FCC再贡献出来8个S2和8OIM卡，然后把这个新的第4个LCC 的S13卡的A0-A2接到这个新增加的OIM卡的J0-J2上即可。
这样的接入能使得系统达到“确保一个数据报文（**Packet**）从任何一个线卡的任何一个端口，能够通过路由路径寻找，从这**72**个**LCC**的任何一个线卡的某个端口被**CRS-1**转发（**Forwarding**）出去。”吗？

下图是笔者描绘的一个3LCC+1FCC的拓扑并通过这个实例来解释这种拓扑结构扩容的问题。



3LCC+1FCC互联拓扑(一个S2(OIM)卡作为一个交换平面)

在图中，笔者勾画出了3个LCC的两个交换平面的连接拓扑，分别是S13卡的SM0和SM4。有兴趣的读者可以依次把LCC剩下的7个S13卡通过FCC相应的OIM卡互联。我们会认识到，这种连接只能保证3个LCC的互联。

现在系统扩容，加一个LCC。自然，这个新的LCC也有8个S13卡和相应的光纤A0-A2接口。如何接入？

是的，FCC拥有24个OIM卡和相应的24个S2交换卡，而且目前其实只被用了8个，还剩下了16个OIM卡槽和16个S2卡。

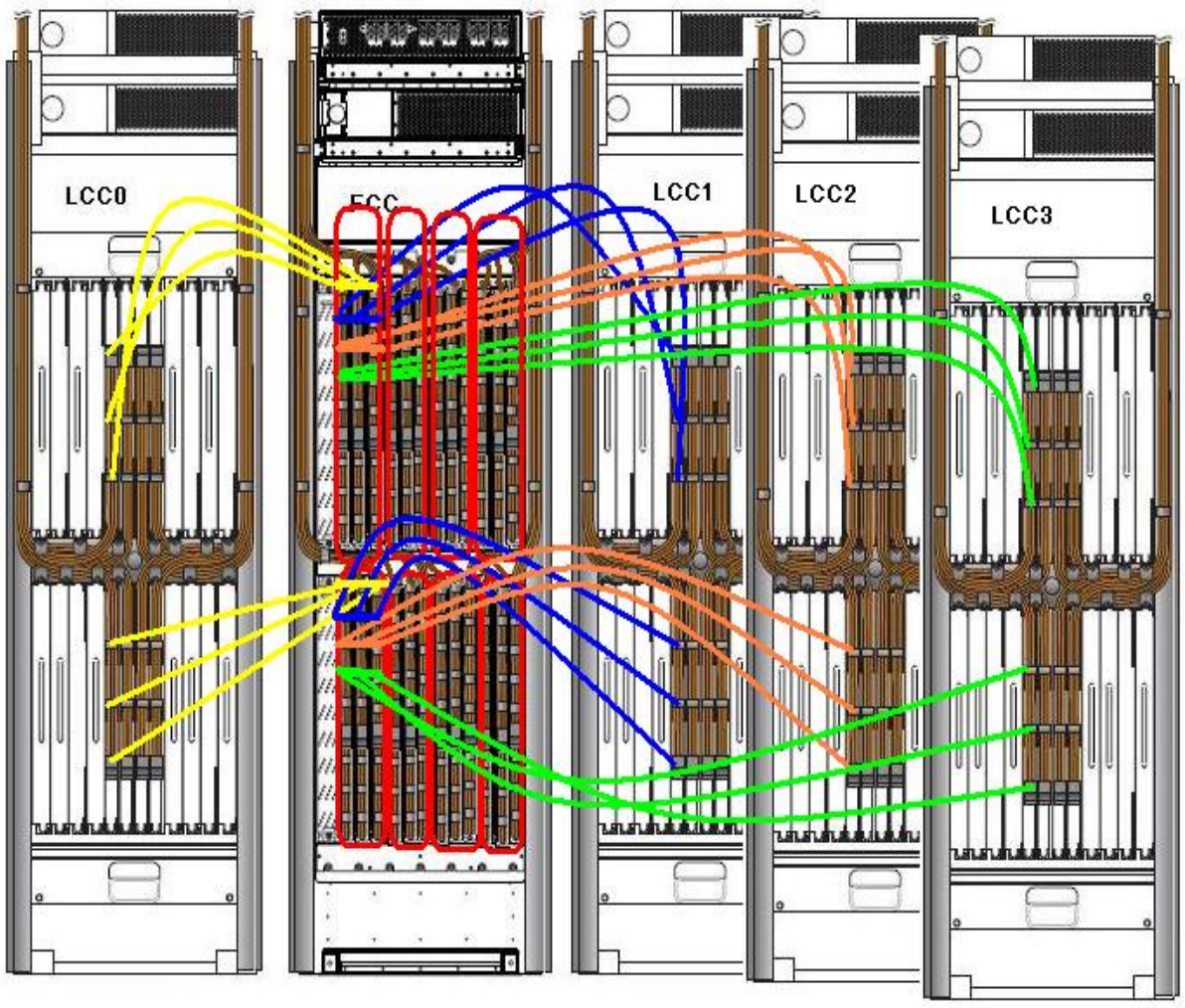
但我们发现，即使你现在从FCC再贡献8个OIM卡，并且把在第4个LCC的8个S13卡的A0-A2都通过光纤接到相应的OIM卡的J0-J2 上，但问题出来了：系统中目前的3个LCC的S13卡的光纤接口A0-A2都已经被用光了！！1，2，3LCC的8个S13卡都已经被全部接到了上图中的 8个OIM卡上，并形成了8个交换平 面。FCC为了这个第4个新增加的OIM卡没有与任何来自其他LCC的S13卡的A0-A2互联。其他LCC的S13卡的A0-A2光纤接头都已经被全部 使用完了；相应的 OIM卡的9个J0-J8接头也被全部占用殆尽了！

因此，这样的接入使得这第4个LCC形成了一个孤岛。没有与任何其他的LCC构成联通。

在3LCC+2FCC，3LCC+4FCC的情况下升级为4LCC+2FCC和4LCC+4FCC，结果是一样的。单纯的从FCC多余的OIM卡拿 出一些来连接新的LCC，只会导致这个新的LCC是一个数据交换的孤岛。其他LCC的数据没有办法抵达（Reach）这个新LCC的S2卡和相应的S13卡。其他LCC的S13卡和相应的S2（OIM）卡都已经全满负载了（J0-J8都被全部用光了），从而导致这个新的LCC的S13卡无法加入相应的8个 交换平面。
不同交换平面 的数据是永远不会交叉转发的！这是CRS-1的一个原则（Rule)。
换言之，在一个S2卡作为一个交换平面的情况下，S2卡上的S2ASIC会而且只会
× 接受来自S13卡上的S1 ASIC芯片转发的数据单元；
×转发数据单元去一个S13卡上的S3 ASIC。
×绝不（Never）把一个数据单元转发到另外一个S2卡上的S2 ASIC。
因此，这里面所有问题的关键就是：如何将一个新的LCC加入交换平面。
从物理的角度，也可以这样理解，如何将一个新的LCC的S13卡的A0-A2接口接入到一个OIM卡上，而这个OIM卡上，系统中其他LCC的一个 S13卡有相应的光纤接口接入，从而使得系统中所有的LCC可以通过S13卡形成一个连通（交换平面）。否则，系统中就形成一个LCC孤岛。

14. FCC多机交换（5）

通过上节的分析可以得知，如果FCC的一个S2卡单独充当一个交换平面，其相应的OIM卡的J0-J9分布被3个LCC的S13卡的A0-A2接入，CRS-1系统多机互联是无法连接多于（等于）4个LCC机器的。
那么，如何通过一个FCC的24个S2交换卡和相应的24个OIM卡来达到支持多于4个LCC互连?
CRS-1目前推出的一种方案是：多模块交换互联（MultiModule Fabric Connectivity）
× 通过把24个S2卡和24个OIM卡分组（Zone）的方式。
×24个S2（OIM）分为8个组。每组含有3个S2(OIM）卡。
×每个组相对应一个独立的交换平面。因此，系统支持8个交换平面。
×每个组能连接来自9个LCC的S13卡的A0-A3接口。
上述机制是如何实现的呢？请参阅下图的一个4LCC+1FCC的拓扑。



4LCC + 1FCC互联拓扑

交换平面

从上述图示，我们可以清楚的得出，24个OIM卡被组合为8个区域（Zone）。分别是：
（FCC 上排）OIM0-2，OIM3-5，OIM6-8，OIM9-11
（FCC 下排）OIM12-14，OIM15-17，OIM18-20，OIM21-23
每一组是一个交换平面，因此系统中是8个交换平面。
下面来看LCC的S13卡是如何接入的。
首先我们试图介绍两个概念：S13卡垂直接入 和S13卡水平接入。
S13卡垂直接入：这就是以前讲述的LCC多机互联的接入方式。S13卡的A0，A1和A2分别接入到一个S2/OIM卡的J0-J8中空闲的 接头上。在这种方式下，每个S2卡提供一个交换平面。
S13卡水平接入：这就是本节要阐述的LCC多机互联的接入方式。S13卡的A0，A1和A2分别接入到一个S2/OIM组的Jx上（0<= x <=8）。在这种方式下，3个S2卡提供一个交换平面。
对于S13卡垂直接入，不再在这里解释，通过对前面章节的阅读，应该是非常直观。

对于S13卡水平接入， 从上图所示，可以观察如下特征：
× LCC0的SM0的A0，A1和A2分别被水平的接入到了OIM11，OIM10和OIM9的J0接头上。
× LCC1的SM0的A0，A1和A2分别被水平的接入到了OIM11，OIM10和OIM9的J1接头上。
× LCC2的SM0的A0，A1和A2分别被水平的接入到了OIM11，OIM10和OIM9的J2接头上。
× LCC3的SM0的A0，A1和A2分别被水平的接入到了OIM11，OIM10和OIM9的J3接头上。
通过这样的连接拓扑，4个LCC的S13卡被有机的挂在了一个FCC的交换平面上。从而可以使得，通过这个交换平面，任何一个线卡的数据可以抵达系统中任何一个其他的线卡。
下面来看一下S13卡水平接入的巨大优点：
由于各个S13卡的A0-A2接口是水平接入到一个OIM组中的Jx上。这个x最大是8。
因此，一个OIM组，或者说一个交换平面最大可以同时接入9个LCC的S13卡。8个交换平面，因此能支持8×9=72个S13交换卡。**换言之，这种每个交换平面有3个S2卡组成的拓扑结构，最大能够支持9个LCC互联。**
这也是目前思科CSR-1系统的上限值--8个LCC通过1，2或4个FCC互联。
下面我们探讨思科CRS-1的72个LCC互联的一些理论值问题。
理论问题1：需要多少FCC才能提供其理论上的72个LCC互联？
上述问题的变种是，如何构造一个交换平面，从而，拥有8个交换平面的CRS-1多机系统能把72个16槽LCC互连？
首先，我们来看看CRS-1是否在理论上能支持72个LCC互联，需要多少FCC。
因为，一个LCC拥有8个S13卡。美国S13卡拥有3个光纤接口（A0-A2），
因此，一个LCC需要8×3=24个FCC的OIM接口。
因此，72个全负载LCC需要消耗72×24个OIM接口。
同理，一个全负载FCC拥有24个S2卡和24个OIM卡。每个OIM卡永远9个光纤接口。
因此，一个FCC可以贡献24×9个接口。
因此，为了支持72个LCC互联，所需要的FCC数量= (72个全负载LCC需要消耗72×24个OIM接口) /一个FCC的接口能力
= (72×24)/(24*9)=8。
因此，需要8个FCC，就可以把72个FCC互联起来。
理论问题2：在72个LCC互连的情况下，每个控制平面需要多少个FCC的S2（OIM）交换卡？
72台LCC机器，每个机器拥有8个S13卡。每个机器都必须通过一个S13卡接入到一个 交换平面。因此，一个交换平面需要支持72个S13卡的接入。每个S13卡是3个接口（A0-A2)，
因此，是72×3个接口。
因为，每个S2卡的OIM是9个接口。
因此，（72×3）/9 = 24。
因此，在CRS-1的理论最大负载的情况下，每个交换平面需要24个S2卡。
这其实也就是一个完整的FCC的最大S2卡装机量。
换言之，当CRS-1达到其理论最大72LCC互联时，需要8个FCC，并且，每个FCC需要满负载，并且作为一个交换平面 。
读者请注意，从以前的章节中，我们可以做出如下一个总结，

× 如果只为了支持1-3个LCC互联，一个S2卡出任一个交换平面就可以，通过S13卡水平接入方法；
× 如果只为了支持 1-9个LCC互联，需要3个S2卡出任一个交换平面，通过S13卡垂直接入方法。
× 如果只为了支持 72个LCC互联，需要24个S2卡（一个FCC整机）出任一个交换平面。目前不知道如何接入方法，这也是说为什么72机器互联是CRS-1的一个理论值。CRS-1目前只支持前两种规模的互联。
理论问题3： 为什么**72**是理论最大值？为什么不能通过够买更多的**FCC**，从而提供**80**个，**88**，或**100**个**LCC**互联？
我们首先通过反证法来阐述。 假设购买9个FCC，来看看是否能够支持多于73个的LCC互联。
因为每个LCC有而且只有8个S13卡。每个S13卡参加而且只参加一个交换平面，
所以，S13卡的最大离散分布是分别接入到8个FCC的S2（OIM）卡上。
现在我们先将前72个LCC通过上述方案接入。 我们得到如下：
第1-8个FCC的24个S2（OIM)卡的J0-J8接口都全部占满。72个LCC的S13卡也全部用光。
现在还剩下一个LCC和一个FCC。
如何连接？这第73个LCC无法接入到任何一个交换平面中，无法加入路由器系统。

15. 参考文献

思科CRS-1主页：CRS-1 Homepage

思科CRS-1产品系列：Models Comparison

思科CRS-1 16-Slot Single-Shelf System

思科CRS-1 8-Slot Single-Shelf System

思科CRS-1 Modular Services Card (LC)

思科CRS-1 4-Slot Single-Shelf System

思科CRS-1 24-Slot Fabric-Card Chassis

彎曲評論

科技 · 人物 · 潮流

思科QuantumFlow处理器及其战略研究

陈怀临，首席科学家

《弯曲评论》

www.tektalk.cn

huailin@tektalk.cn

0.前言

本文着重介绍了思科的边缘路由器ASR1000系列的重要组成部分QuantumFlow网络服务处理器（简称QFP），并从各种不同的观点（Viewpoint）来观察和试图剖析QFP的结构。这些观点包括处理器观点，互联观点，软件观点，报文观点，系统观点等。QFP是思科在WAN接入和智能化边缘设备解决方案中的重要技术支持。对QFP的理解对掌握现代高端网络系统体系结构是非常有意义的。

1. QuantumFlow

2008年2月25日，思科发布业界新闻，发布其最新一代的，也是世界上最强大的，网络处理器QuantumFlow。其新闻稿的英文原文可参阅如下链接：思科新闻稿

SAN JOSE, Calif., February 25, 2008 - Cisco® today introduced the Cisco QuantumFlow Processor, the most advanced piece of networking silicon in the world and the industry's first fully integrated and programmable networking chipset. More than half a decade in the making, the Cisco QuantumFlow Processor consists of 40 cores on a single chip and can perform up to 160 simultaneous processes, making it uniquely geared for today's network environments and several generations beyond what is currently available in network processors.

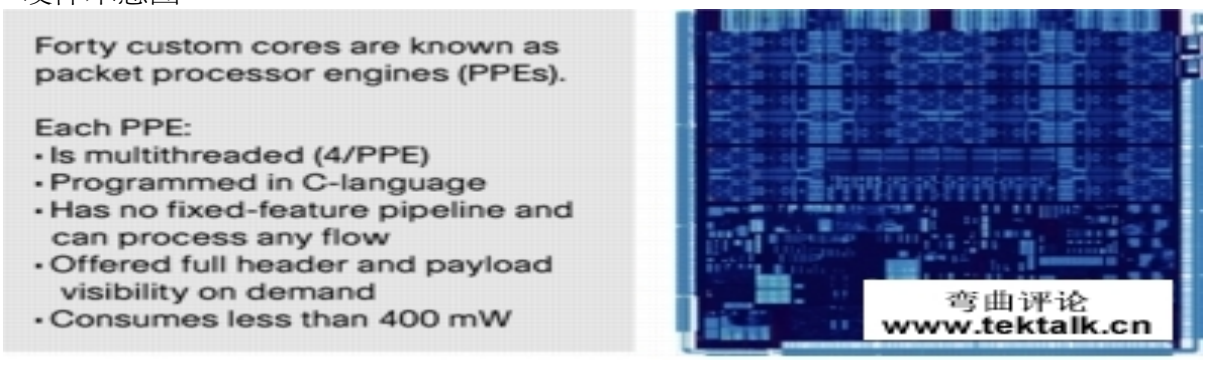
The Cisco QuantumFlow Processor was designed by a team of more than 100 Cisco engineers and has led to more than 40 patent submissions. Many of the same engineers who developed the Cisco Silicon Packet Processor (SPP) for the Cisco Carrier Routing System (CRS-1), which

debuted in 2004, also worked on the Cisco QuantumFlow Processor. Continued advancements in technology, design and expertise enabled the team to increase the transistor density on the chip from a then networking-industry-leading 185 million on the Cisco SPP to more than 800 million on the Cisco QuantumFlow processor. Such density puts it in the tier of some of the most advanced processors developed by leading semiconductor companies.
下面是笔者尝试做的中文翻译：
思科公司今天发布其QuantumFlow处理器，工业界最强大的，第一个完全集成的，可编程的网络处理芯片。经过5年多的工程开发，Cisco的 QuantumFlow处理器在一个芯片内部含有40个CPU处理器的核，可以同时做160个数据处理。其超强的计算能力使得QuantumFlow处理 器成为当今网络计算的宠儿，并领先业界中其他的网络处理器数代之遥。
思科的QuantumFlow处理器的设计团队有100多个工程师组成。其设计研发过程中，团队共提交了40多个专利申请。QuantumFlow 的研发工程师许多来自思科2004年发布的多核网络处理器SPP（Silicon Packet Processor）设计团队。SPP被思科用在业界最高端的路由器CRS-1（CRS：CarrierRouting System）上。新技术的不断发展，持续的设计改进和优秀的技术能力使得QuantumFlow的研发团队在该芯片上的晶体管密度提高到8亿个晶体管，而原来SPP芯片为1亿8千5百万个晶体管。QuantumFlow的高密度集成使得其成为半导体业界最复杂的芯片之一。

2. 五年之寒

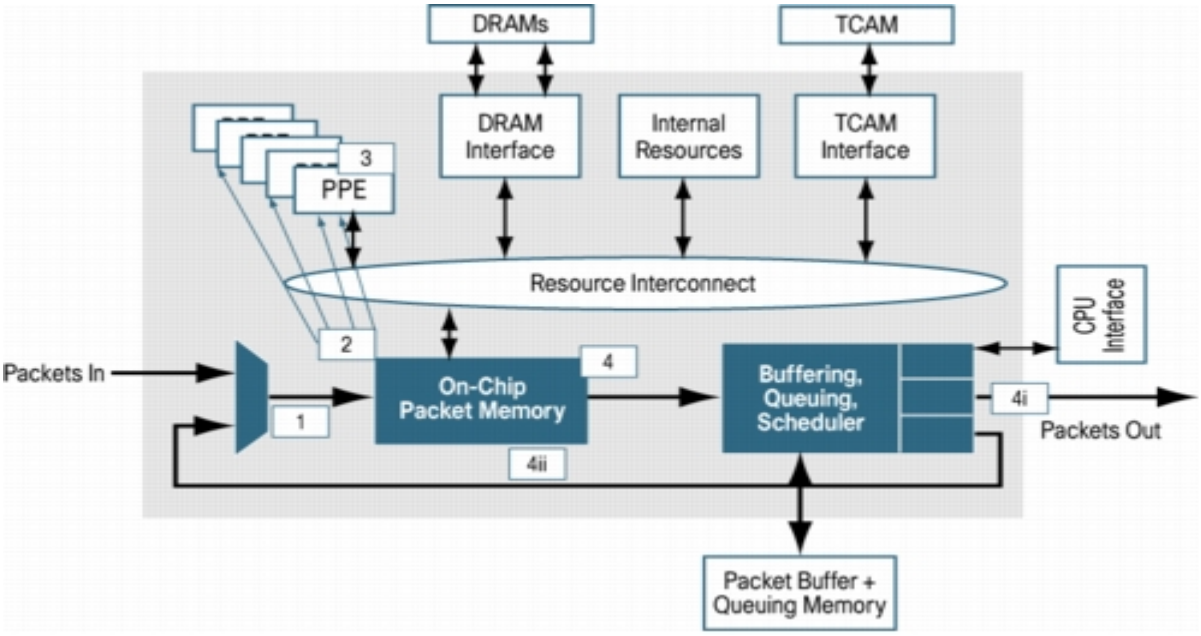
下面是笔者从各方面公开文章中收集整理到的QuantumFlow的一些信息：
× 项目启动时间：2002年Q3或Q4【笔者注：SPP是2002年流片的。CRS-1是2004年推出的。】
×研发耗资：1亿美金
×研发领导：Will Eatherton，Distinguished Engineer and Director of Engineering
×芯片主要定位：边缘（Edge）路由器，企业路由器。
×芯片解决问题：Stateful Service与转发（Forwarding)合一。 【笔者注：如Voice, Video, 防火墙，深度检测等】
×首发系统：ASR1000【笔者注：第二个应用系统是ASR9000。发布时间为2008年11月11日】
×主频：1.2GHZ【笔者注：ESP-5G:900MHz. ESP-10G:900MHz.ESP-20G:1.2GHz】
×晶体管数目：8亿
×（PFE）内存：DDR2。 【笔者注：RLDRAM 】
【笔者注1：两个On-Chip内存控制器（Memory Controller)】
【笔者注2：ESP-5G:256MDRAM. ESP-10G:512MDRAM.ESP-20G:1GDRAM】
×数据报文内存（Packet Buffer）：ESP-5G:64M.ESP-10G:128M.ESP-20G:256m
*CAM:外挂TCAM 【笔者注：ESP-5G:10M. ESP-10G:10M.ESP-20G:40M】
×功耗：80瓦
×多核：40，4 Way-Thread。来自Tensica的Xtensa。
【笔者注1：应该不是Xtensa-7和Xtensa LX2】
【笔者注2：ESP-5G:20Core. ESP-10G:20Core.ESP-20G:40Core】

×片内互联（Interconnect）：Crossbar Switch
×片外互联：ESI【笔者注：在将来的新QFPzhong，将是Interlaken】
×数据报文接口：4个10GBPS SPI4.2 。
×工艺：90nm
×流片：德州仪器
×硬件示意图：



×

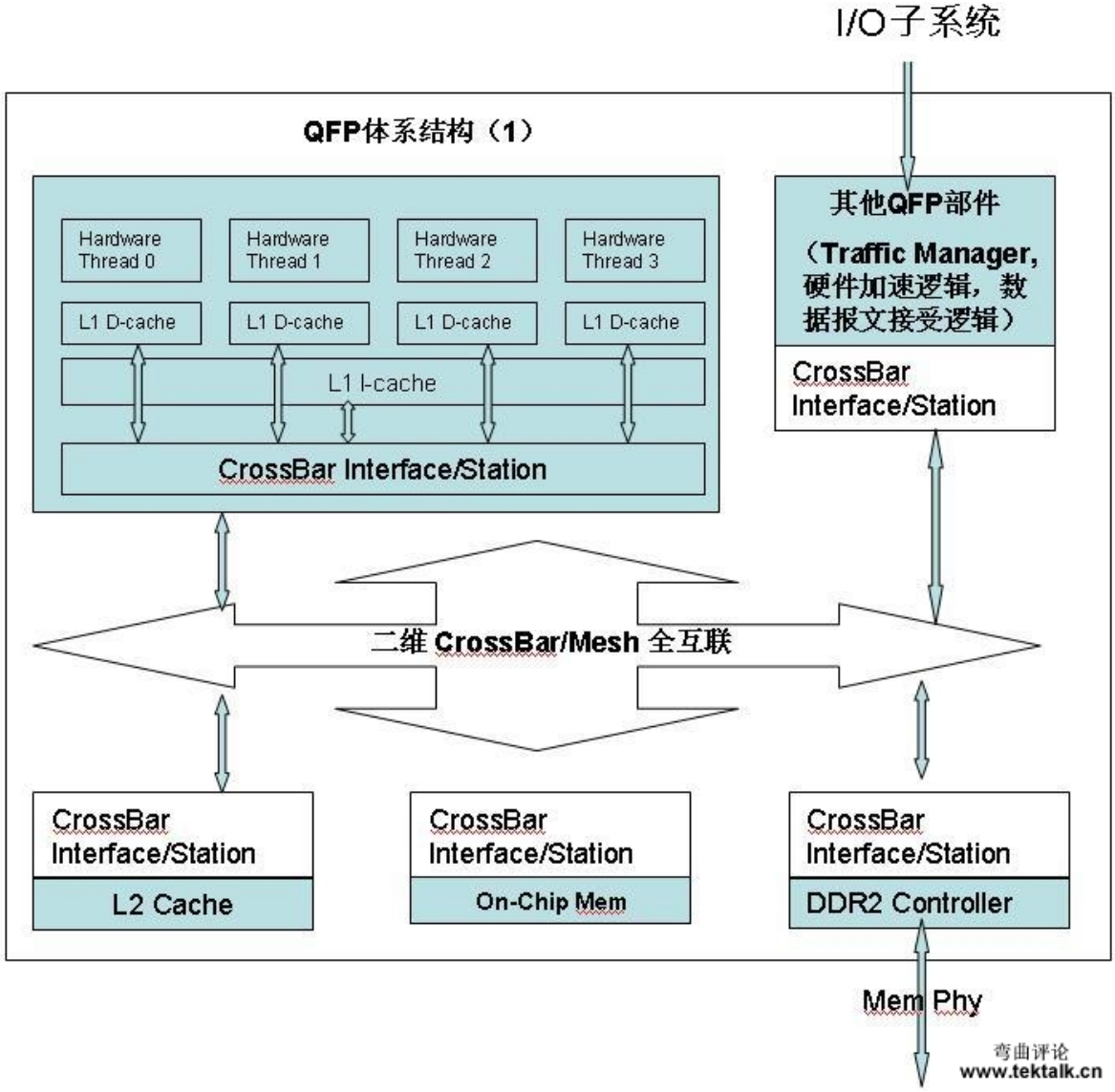
体系结构示意图：



3. 新闻报道

× ASR 1000 QuantumFlow Video
× Strategic Drivers for Development of the Cisco ASR 1000
×Service Provider Market Drivers for the Cisco ASR 1000
×Cisco ASR 1000 Series Aggregation Services Routers
×The Cisco QuantumFlow Processor: Cisco's Next Generation Network Processor
×Cisco puts high-end silicon on the edge：New service router packs custom 40-core CPU

4. 体系结构--处理器观点



观察思科的QFP芯片，可以从不同的观点（Viewpoints）。QFP是一个高端的网络处理器，里面有40个CPU核心，大量的片内和片外内存，高速的互连网络，大量的网络报文引擎，硬件加速器等。从不同的角度来观察QFP，得到的图像是略有不同的。

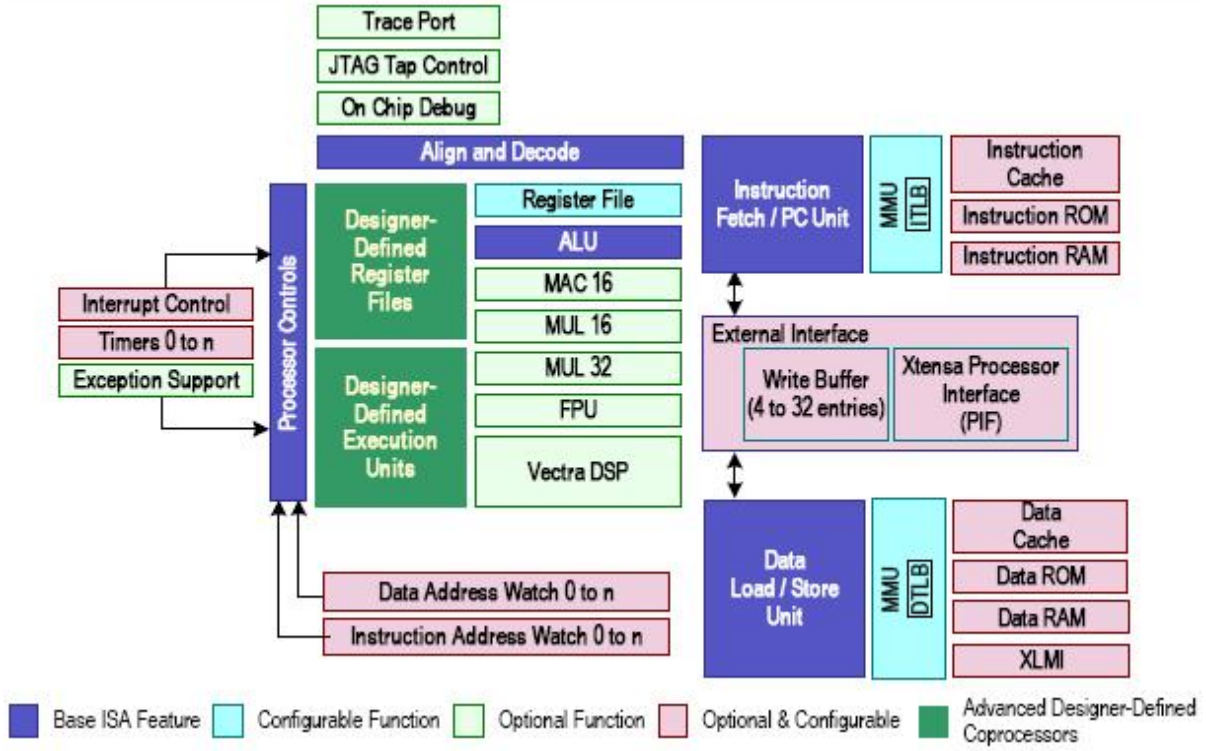
上述图示是笔者从CPU的角度来观察QFP的逻辑。我们可以暂且称之为QFP-CPU。

从QFP-CPU角度，其是一个多核系统。一共有40个核。每个核有4个硬件线程。其4个线程的调度关系应该是FMT的架构，从而最大程度的利用 Mesh互联的带宽。从思科的公共资料可以得知，对于ESP-20BPS的QFP，其CPU核的指令集是Tensilica的Xtensa的ISA（MIPS的变种）。其时钟在1.2GHz。对于Mesh互联的Clock Domain的细节，目前不可知。

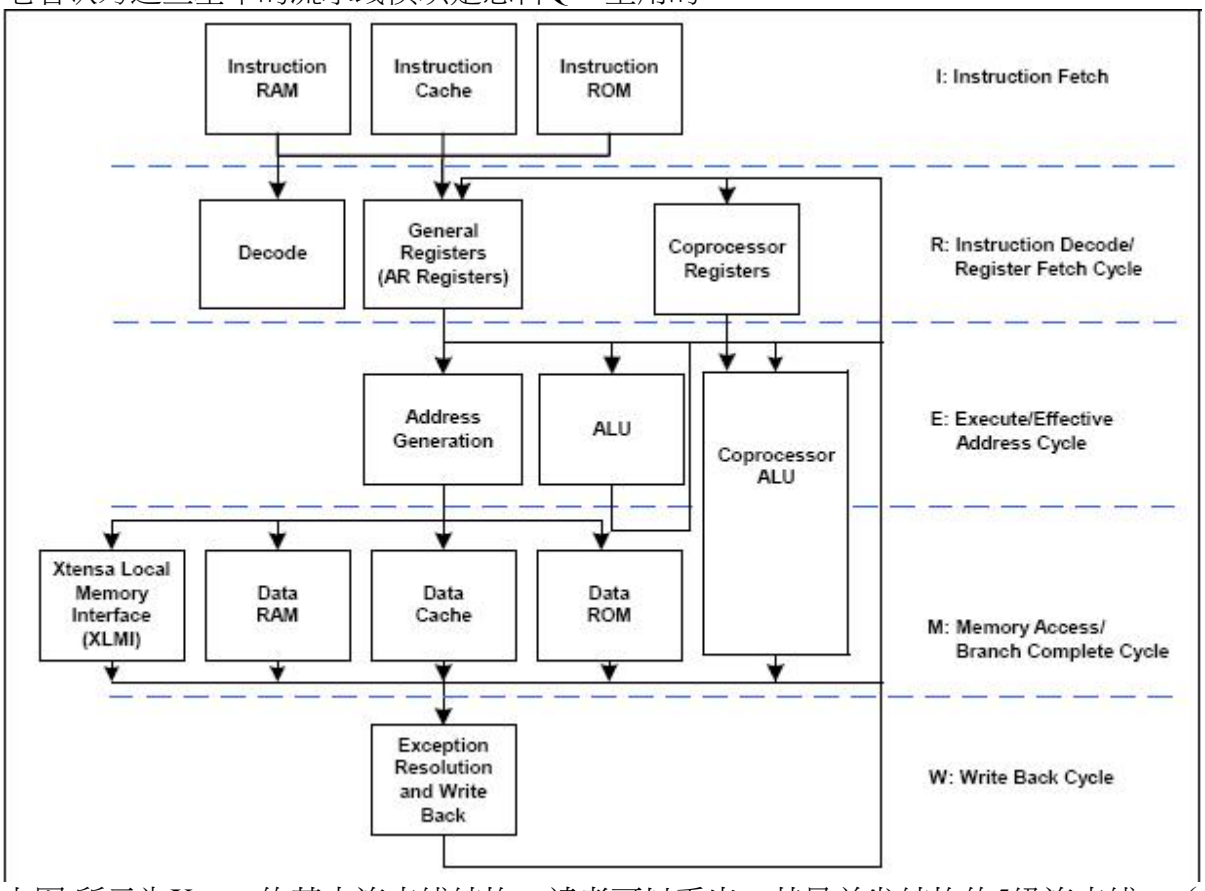
每个核都是32位指令集。换言之，GPR是32bit的。

每个CPU核的硬件线程有自己的L1数据缓存；
一个CPU核的4个硬件线程共享L1指令缓存；
QFP的40个核共享一个L2缓存。（这个L2是单纯的指令缓存，或是Combined指令和数据缓存，目前没有明确的资料可以确认。但笔者倾向于 QFP的L2是一个单纯的指令缓存。
QFP的HT在L1数据缓存Miss后，是直接通过Crossbar的接口读取片外的RLDRAM。笔者分析的原因 是：从QFP-Packet的观点，数据报文都在片内的内存中。所以QFP的CPU核对数据读写应该绝大多数落在片内报文内存上。而指令流的性能对于采用 SIMD并行计算模式的QFP多核系统是最重要的。
读者们要注意，在现代多核系统设计中，总线的概念和机制基本上已经消失。所以从QFP-CPU的角落来观察QFP是，L2缓存，DDR内存控制器等，其实都是挂在Mesh上的一个节点。与其他40个核节点构成一个2维Mesh全联通的图。
另外，从QFP-CPU的角度，其他所有的数据报文（Packet）硬件逻辑可以简单的理解为“I/O子系统”。

思科的QFP的40个CPU核采用的是Tensilica的Xtensa的ISA。从思科公开的新闻资料中，思科声称是只采用了Xtensa的指令集结构(ISA)，而其他部件或子系统的逻辑设计，加上后端设计和封装，都是思科自己研发队伍完成的。
从这些声明，对思科QFP有兴趣的读者可能会误以为思科只是购买了Xtensa的指令集（Instruction Set)。这是不精确的。Xtensa ISA的含义不仅仅包括指令集，也包含Xtensa的一些基本的的微结构，例如基本的流水线结构等。否则，思科没有必要去购买一个非主流的指令集，而从新 做一个CPU，用OpenRISC的ISA就可以了。
Tensilica的Xtensa是一个SoC软核，从而第三方可以进行定制和裁剪做出其自己的SoC。也可以同过Tensilica提供的集成环境调试，增加新的指令等等。下图是Xtensa的体系结构略图:



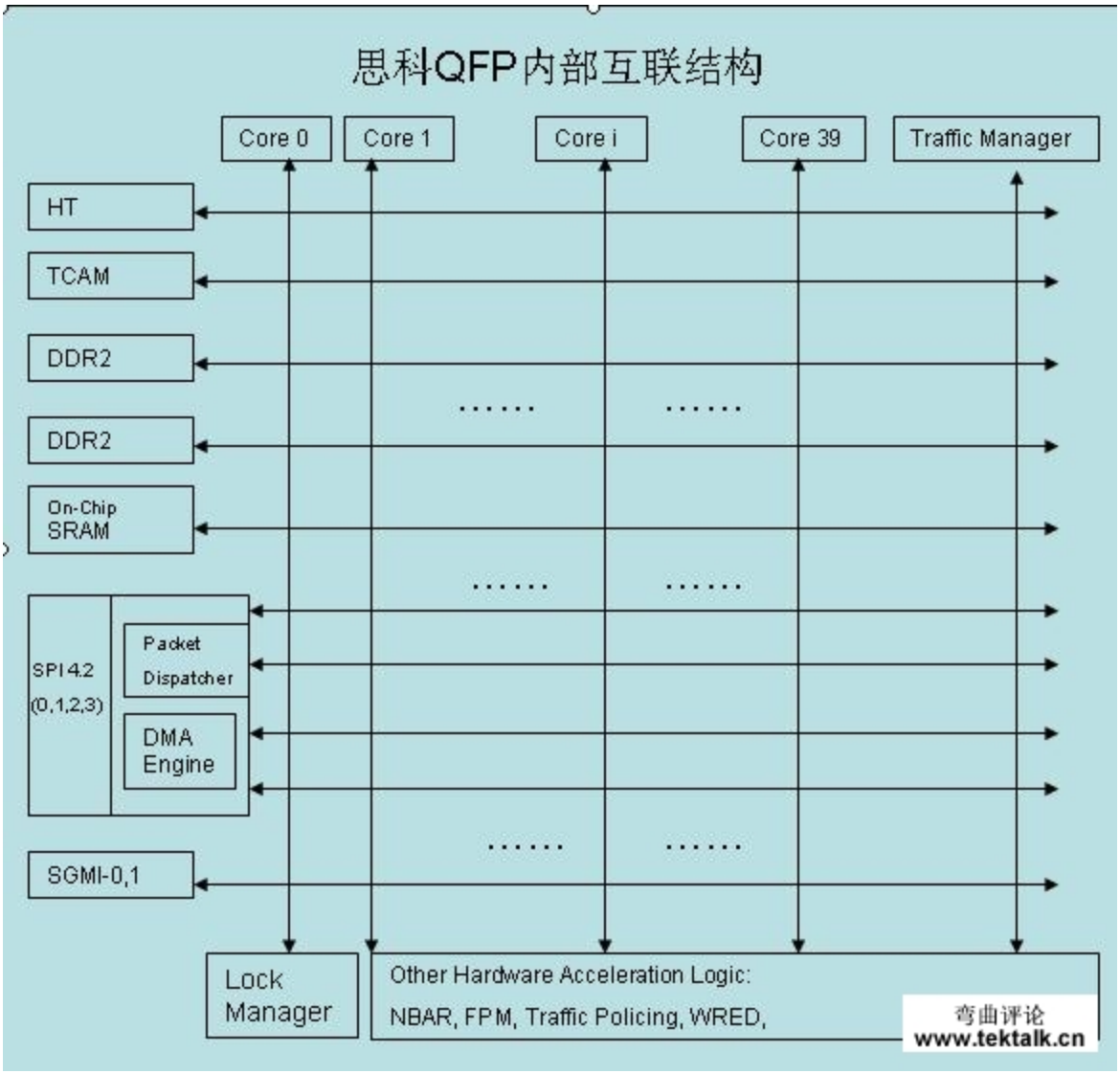
从图中所示，可以得知蓝色模块是Xtensa体系结构ISA中的基本模块。
笔者认为这些基本的流水线模块是思科QFP重用的。



上图 所示为Xtensa的基本流水线结构。读者可以看出，其是单发结构的5级流水线。（思科声称QFP是3发射）。另外，因为Xtensa的核心并没有实现复 杂的Out of Order的执行和超标量结构，没有Interlock的处理，因此，在设计内存操作的指令时，仍然会有delay slot等的发生。
从上述Xtensa的ISA微结构，然后比较思科QFP的微结构，特别是每个核有4个硬件线程的结构。我们可以知道思科确实是需要在逻辑和物理设计 方面做许多特定设计，才能达到QFP的设计目标。例如，基于FMT的4个硬件线程的微结构的设计需要增加几套寄存器，局部总线（Local Bus）和裁决逻辑（Arbitor）等。
另外，在MMU方面，QFP估计也是基本上会重用Xtensa的结构。但感觉在缓存结构上和内存总线接口方面，思科的QFP研发团队需要对Xtensa做许多修改。例如必须将Xtensa挂到QFP的2维CrossBar或Mesh互连结构上。这都需要巨大的研发工作。
从思科发布的QFP资料，其核的主频可以做到1.2GHz。为了能做到1.2GHz，大量的后端设计和手工调试需要开展，而非简单的流片 。笔者曾经在5年前用过Xtensa定制过一款SoC的软核。记得当时的体会是，在缓存方面，只要稍微一加大，主频立刻急剧的下来。
一般而言，思科的QFP里面一定会通过Xtensa的全套工具，加入QFP自己的指令集扩充，并且在gcc tool-chain上直接支持。
除了是可定制的SoC软核，编译器和工具链的强大支持是Xtensa之所以能够10年之久还生存的重要原因之一。

5. 体系结构--互联观点

考察一个高级网络处理器时，其互联结构（Interconnect）是一个非常重要的一环。对于一个SoC芯片而已，CPU核本身，作为一个计算 单元，只是众多的计算逻辑的一个子部分（Sub-System）。系统中的许多其他部件，如硬件加速器，网络报文处理逻辑等等，都是非常重要的组成部分。 因此，如果将这些处理部件有机的互联起来，从而达到快速，低功耗，和其他诸多电气要求，就成为一个现代高级芯片设计的关键。
不同的网络处理器设计公司通常 采用不同的方法。各有优缺点。例如，RMI的XLR系列是通过FMN（Fast Message Network）的环结构（Ring），从而使得系统的CPU Core，报文处理单元PDE，加密单元Cypto引擎，多处理器互联的HT接口（HyperTransport），PCI-X等等，都互联在一起。值得 注意的是XLR的内存模块并没有挂在FMN的环上。这也是RMI XLR体系结构的一个特点--通过将FMN和MDI（内存互联结构）分拆，从而达到，数据通道通过FMN，而通常比较慢的内存访问通过MDI互联。另外一 款著名的网络处理器是Cavium的Octane系列。Octane芯片的内部互联相对而言略简单一些--仍然通过经典的支持Cache Coherence的共享内存总线（Memory Bus）的方式将CPU核与内存互联。其它部件，例如网络加速部件，DFA查询部件，SPI端口等等都是通过一个I/O总线的方式互联，然后通过一个接口 转换桥（Bridge）挂在内存总线上。关于RMI的XLR系列和Cavium的Octane系列的体系结构，笔者会在将来的文章中做更详细介绍。



思科的QFP是一个inhouse的网络处理器，因此基本上不可能有公开的资料显示其内部详细结构，特别是其互联结构。上图所示，是笔者从其公共的一些产品文档中多QFP的一些介绍，对其互联结构做出的一些基于个人技术观点的推测。

总体而言，QFP的互联是一个基于二维的Crossbar的互联。在这个互联上，每个逻辑部件对于互联逻辑而言，都是一个节点（node）。这个节 点除了实现自己的计算逻辑之外，实现一个与这个互联网络协议的接口（interface），通常而言，是一个命令（command）协议和一个数据 （data）协议。这个互联协议的重点是通过在这个互联网络上发送命令和数据单元，从而可以从一个节点抵达任何另外一个节点。

如图所示，40个基于Tensilica的Xtensa ISA的CPU核都是这个互联上的节点。QFP的另外一大部件Traffic Manager也是一个节点。在这个互联上，还拥有许多其他的网络加速，内存，和为了与QFP芯片外部互联的逻辑部件。

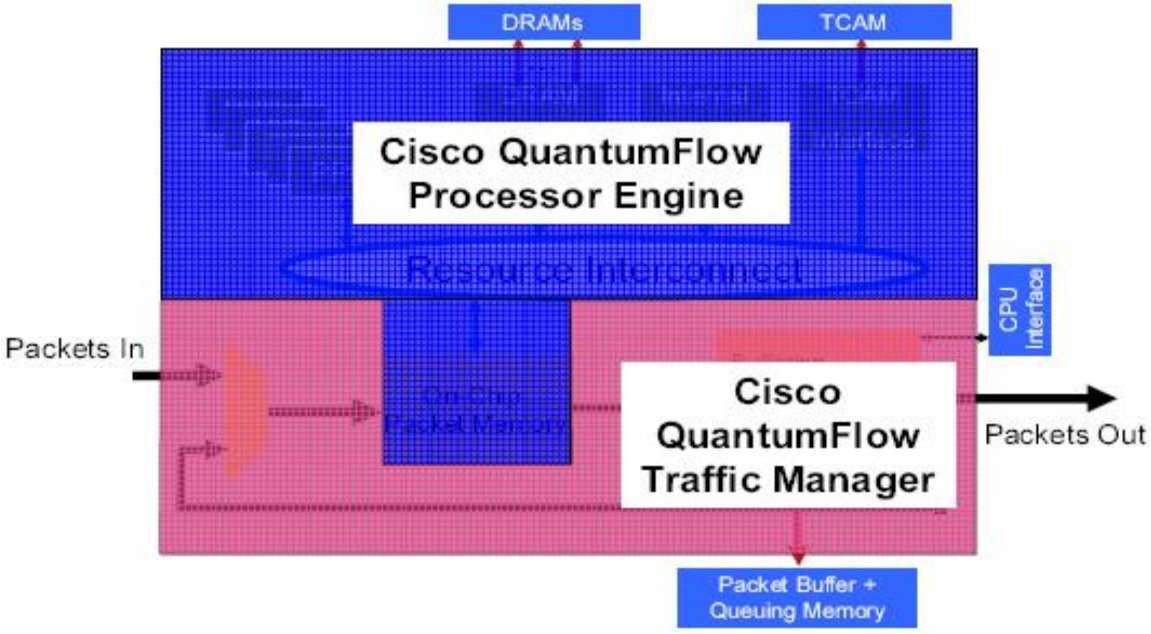
HT: HyperTransport接口。这是用来使得ASR1000上的ESP（Embedded Service Processor）模块（板）上的主控CPU与QFP互联的主要接口。通过HT互联，从主控CPU的角度，QFP就是一个设备，从而可以使得主控CPU 可以通过HT来操作QFP，例如通过读写控制寄存器的方式等。

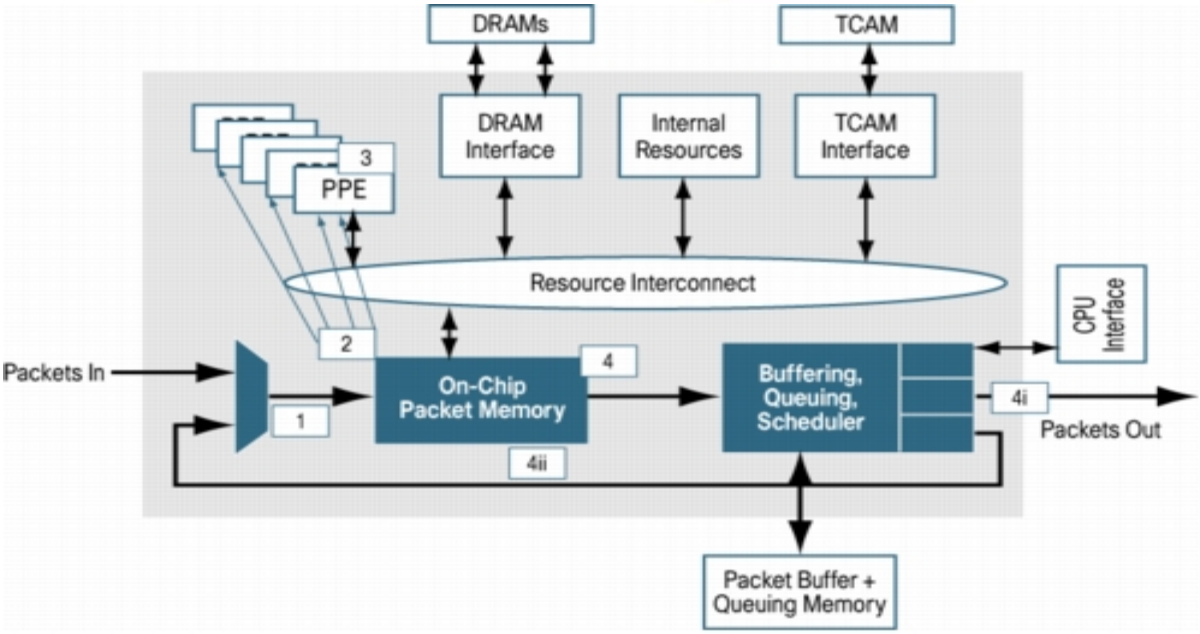
TCAM：这是QFP做数据报文查询（lookup）的外挂 CAM存储器。
DDR2：QFP应该是内含有两个DDR2的控制器。一个是给40个CPU核要用的RLDRAM的控制器；一个是给Traffic Manager的DRAM。
On-Chip SRAM：QFP的报文，当通过入口逻辑（Ingress）抵达之后，是被全部（报文头和报文数据）都被放到了QFP内部的报文内存中，而非在外挂的 DRAM。这一点是QFP有高性能的重要技术亮点之一。许多其他类似的网络处理器都是将数据报头可以放到芯片的内含逻辑中（如寄存器，或L2 Cache里），但数据包的数据（Payload）通常是通过DMA引擎传送到外挂的DRAM里。
SPI-4.2的Packet Dispatcher：QFP的Ingress是支持4个SPI 4.2的接口。也就是说，可以支持40G的线速。这4个SPI在QFP在ASR1000上其实是有分工的。从不完全的资料显示，应该是2个SPI是 QFP，通过外部系统互联，链接线卡（SIP）。其他2个SPI是链接其他的ESP/QFP（支持HA）和控制平面RP（Routing Processor）的。值得注意的是，QFP芯片本身不内含有加密逻辑部件。加密逻辑部件是外挂的，并且通过SPI接口与QFP互相连接和通信。
SGMI：QFP应该还支持2个1G的以太端口，作为控制使用。
QFP 除了 上述主要互联部件之外，还支持许多叫做Internal Resources的硬件加速部件，并且作为节点挂在这个二维的互联结构上。其中一个特别值得注意的是硬件锁机制。笔者相信这个部件是提供了一系列的 spinlock，mutex等逻辑，从而可以使得40×4=160个硬件线程可以通过高速的锁机制，从而达到并行计算中所必须的同步机制。
除了硬件锁机制，QFP还提供了许多其他的硬件加速模块，例如NBAR, FRM，Traffic Policing， WRED等等。

6. 体系结构--报文观点

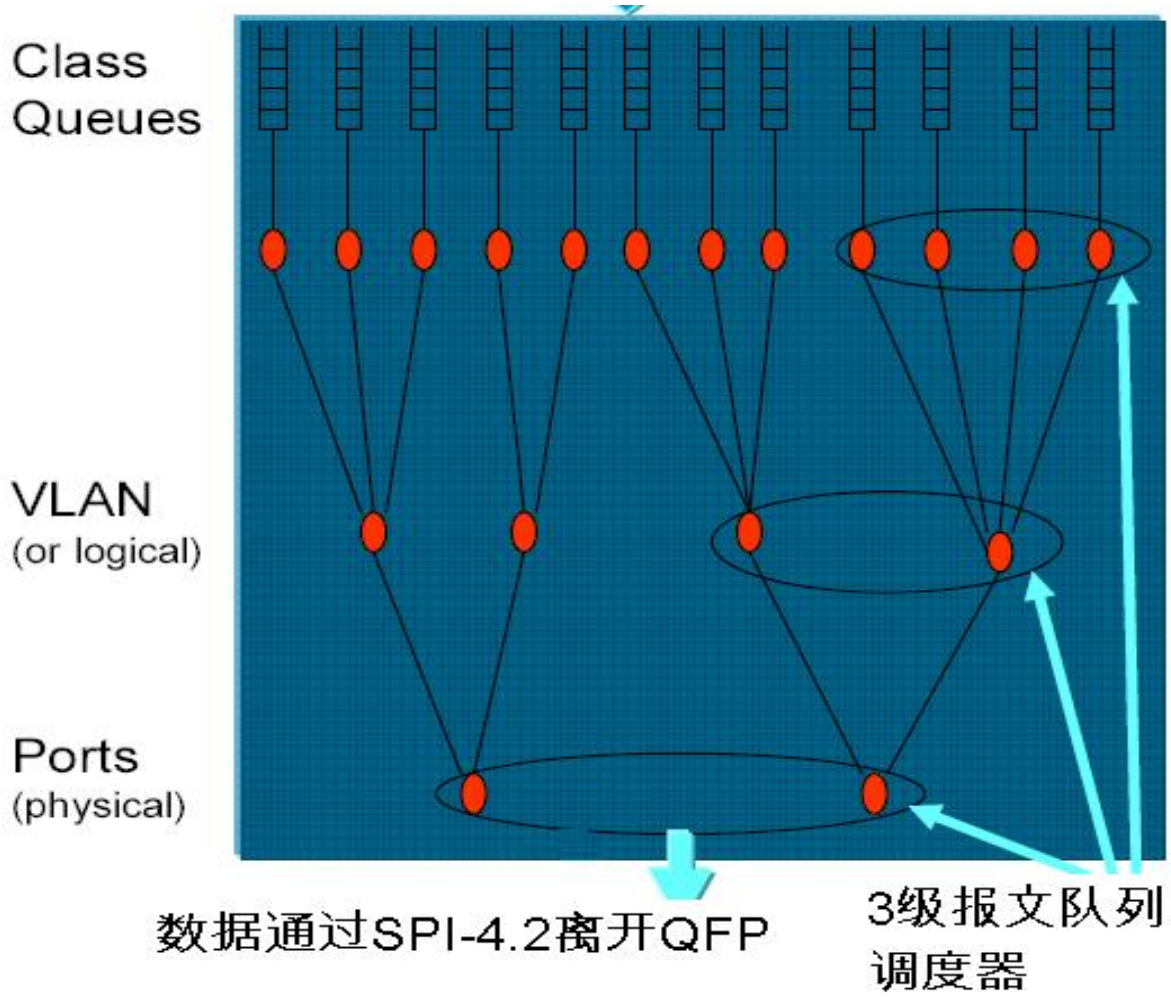
从网络数据报文的观点来观察思科的QFP，QFP就是一个数据报文的从层2一直到层7的数据处理与转发的引擎。在基于QFP的思科ASR1000系 统中，读者要非常值得注意的一个观点是：QFP扮演的是一个集中式数据处理的角色。换言之，系统中所有的数据的，从线卡（SPA-->SIP）和控 制平面卡（RP），都是通过系统的背板Backplane互联ESI，而进入QFP。QFP处理后，决定是应该转发给某个线卡并发送出去，或者是应该转发 个控制平面卡。QFP扮演着一个集中式数据控制和处理的角色。具有HA的ASR1000系列，具有两个ESP卡。美国ESP卡上有一个QFP。这两个 QFP/ESP的关系是一个Active，另外一个Standby。所有的数据报文都是进出当前的主QFP处理器。辅QFP，通过一个专用的SPI4.2 10Gbps的通道，与主QFP通信，做状态备份。

思科QFP数据报文流程





如上图所示，从报文的观点来观察，QFP的逻辑分为两大部分。第一个部分是QFP-Processor Engine。第二部分是QFP-Traffic。第一个部分主要就是那40个Tensilica的Xtensa ISA的处理器单元。第二部分是由一些数据缓存，队列（Queue）和相应的调度算法逻辑组成。下图所示为Traffic Manager的一个逻辑结构图。



思科QFP的Traffic Manager结构图

1. 当一个数据报文通过一个相应的SPI-4.2数据通道抵达后，QFP的报文分发部件（Dispatcher）会将报文层2的Frame所以数据都传送到 QFP的内部报文缓存里（On-Chip Packet Memory）。也就是说，不仅仅是报文的头（Header），是包括数据（Payload）都存放在QFP芯片内部的缓存中处理。这部分的功能还包含一 些基本的数据报文的处理和分析工作

2. QFP的报文分发部件将这个新的数据报文分配给一个计算单元（一个CPU核的一个硬件线程）。这个线程将从头到尾的负责这个数据报文的处理序列，其中包括：

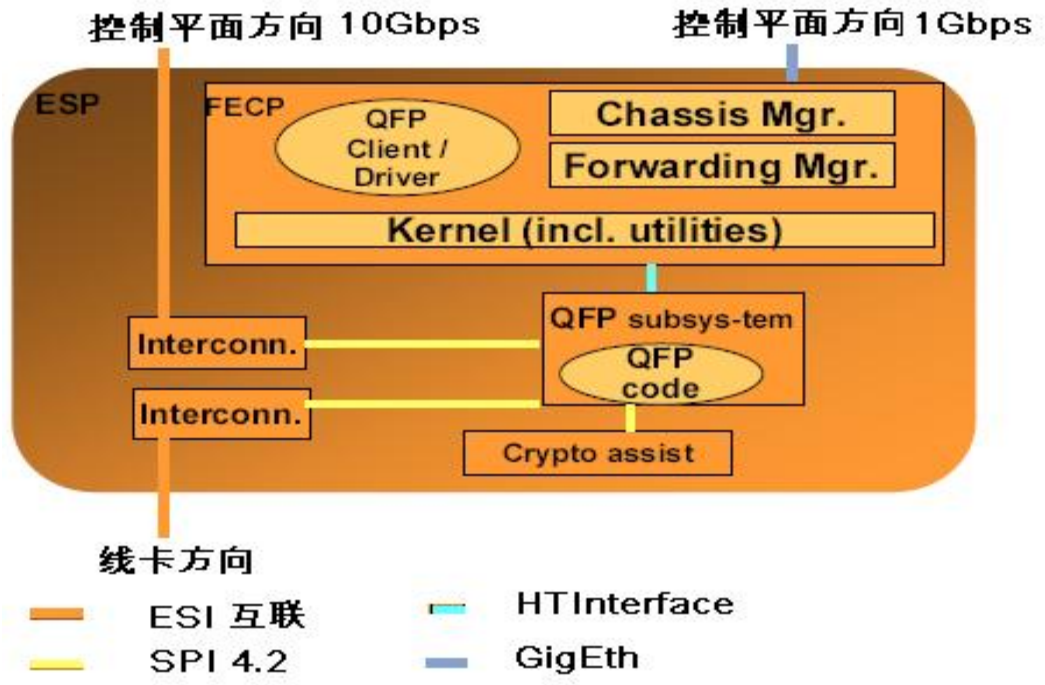
进入时：要进行Netflow，MQC/NBAR Classify, Firewall, RPF, mark/Police, NAT, WCCP, Deep Inspection, 等等。

转发功能：QFP当然首先是一个路由器的数据平面的一个引擎。所以要处理这个报文该往哪里转发的问题。因此QFP的Processor Engine要完成如下工作： IPV4的FIB，MPLS, Multicast等等。

当要离开QFP Processor Engine时，还需要 Netflow，MQC/NBAR Classify, Firewall, NAT, Police/Mark和Crypto 等功能。为什么要考虑加密呢，因为，有可能是VPN tunnel的数据报文。
这时，一个数据报文就已经完成所有的软件处理可以被释放给QFP的Traffic Manager做最后的调度并发出离开QFP了。这个数据报文会被传送到相应的Traffic Manager的队列中。这个过程与这个报文是一个Trhough Traffic（要去另外一个线卡端口），或者是一个去控制平面（RP）的报文，或者是一个HA报文，有关系。不同的报文类型将被放到不同的队列中。从而 Traffic Manager的队列调度器可以通过不同的调度算法去相应的发送一个报文。
3. QFP的Traffic Manager的队列调度功能将决定一个报文的去向。目前，Traffic Manager可以支持128K个队列。强大的队列调度器可以应用许多QoS算法在系统的数据报文上。值得注意的两点是：如果一个数据报文被一个 Processor Engine的线程处理完之后，还需要再来一遍，Traffic Manager负责相应的队列中的数据报文再次转发到On-chip数据报文缓存中；如果一个数据报文需要加密，Traffic Manager会通过相应的SPI通道启动QFP外部（在ESP板子上）的加密部件。

7. 体系结构--软件观点

从思科发布的资料可以得知，ASR1000系统的整个软件操作系统叫做IOS-XE。IOS-XE是一个基于Linux的IOS操作系统。如何理解 这个“基于Linux的IOS操作系统”呢？笔者会在将来单独对思科的操作系统的演化和战略做专题分析。对于拥有QFP芯片做转发和服务处理的 ASR1000系统，IOS-XE的基本特点是：
--在控制平面上，可以支持单卡（RE）的两个IOS的运行。从而可以支持单RE（控制平面）的HA，例如，ISSU。这是非常重要的一个亮点。思 科就是利用Linux 2.6之后的KVM虚拟技术，从而在Linux上运行了两个IOS。每个IOS作为一个Linux的用户进程来运行。
--在ESP板子上，或者说，数据处理卡上，通过一个主控CPU（一个PowerPC），运行一个Linux Kernel和相应的Chasis Manager， Forwarding Manager等管理进程，并与RE的IOS和相应的进程通过标准的IPC进行通信，从而达到控制平面和数据平面的各种路由，ACL，Config， Policy，动态log数据采集，错误汇报等同步工作。另外，这个主控CPU起到一个控制QFP芯片的作用。换言之，QFP的所有软件都是这个主控 CPU来安装，启动，运行的。QFP在这个层面上，扮演的是一个（专门处理数据报文的）协处理器的角色。
--在线卡（SIP)上，也通过一个主控CPU（一个PowerPC），运行一个Linux Kernel和相应的Chasis Manager， Interface Manager等管理进程，并与RE的IOS和相应的进程通过标准的IPC进行通信，从而达到控制平面和线卡的各种同步工作 。
在这篇文章里，笔者不详细讨论控制平面RE和线卡的软件系统，而是专注于数据处理卡ESP和QFP的软件结构。
ESP和QFP的软件结构图基本上如下图所示：



思科QFP软件结构（1）

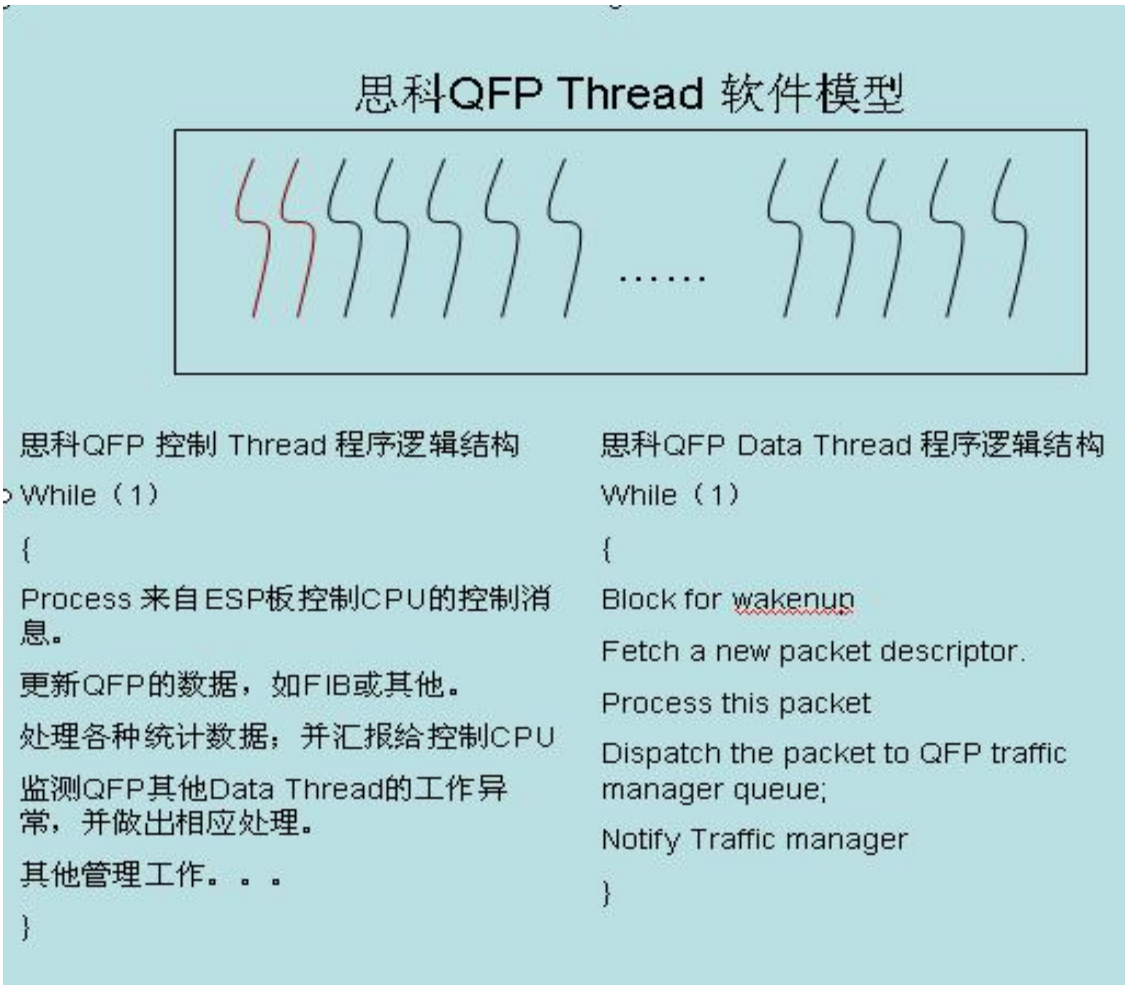
从上图可知，QFP软件子系统分为三个部分：位于主控CPU上的Linux核心上的QFP Client进程；位于主控CPU上的Linux核心里的QFP Driver（驱动）代码函数库，和位于QFP芯片40个Xtensa核上的QFP代码。QFP上的代码通过HyperTransport Interface与主控CPU上的QFP Client进行通信。通过SPI 4.2来获取报文数据，启动加密引擎，和做HA的处理（与另外的ESP板子上的QFP芯片）。主控CPU上的Linux环境中的Chaisis Manager和Forwarding Manager通过一个1Gbps的以太端口与RE（控制平面）通过IPC进行通信。

对于QFP的160(40 * 4)个线程，其软件结构（环境）大致是这样的：

×QFP上没有一个宿主操作系统。没有Linux kernel，当然更不存在IOS等。QFP上多核部分的数据报文处理逻辑应该是运行在一个裸机环境下。或者一个非常简单的硬件抽象层上（HAL）。这个 HAL做一些简单的Tensilica Xtensa ISA的初始化，TLB的设置，中断的处理，内存的划分等等。从目前思科所以公开资料，看不出来其运行了任何OS和微内核。

×QFP的启动， 数据报文处理软件的下载，安装，运行，都是由主控CPU，通过HT接口，来控制的，例如，对于主控CPU而已，QFP的控制寄存器，就是主控CPU下可看 见的一批Memory Mapped I/O地址。可以被读，写等。QFP Driver逻辑就是完成QFP驱动的Linux核心模块。

×QFP的线程作为引擎的角色，运行数据报文处理软件。

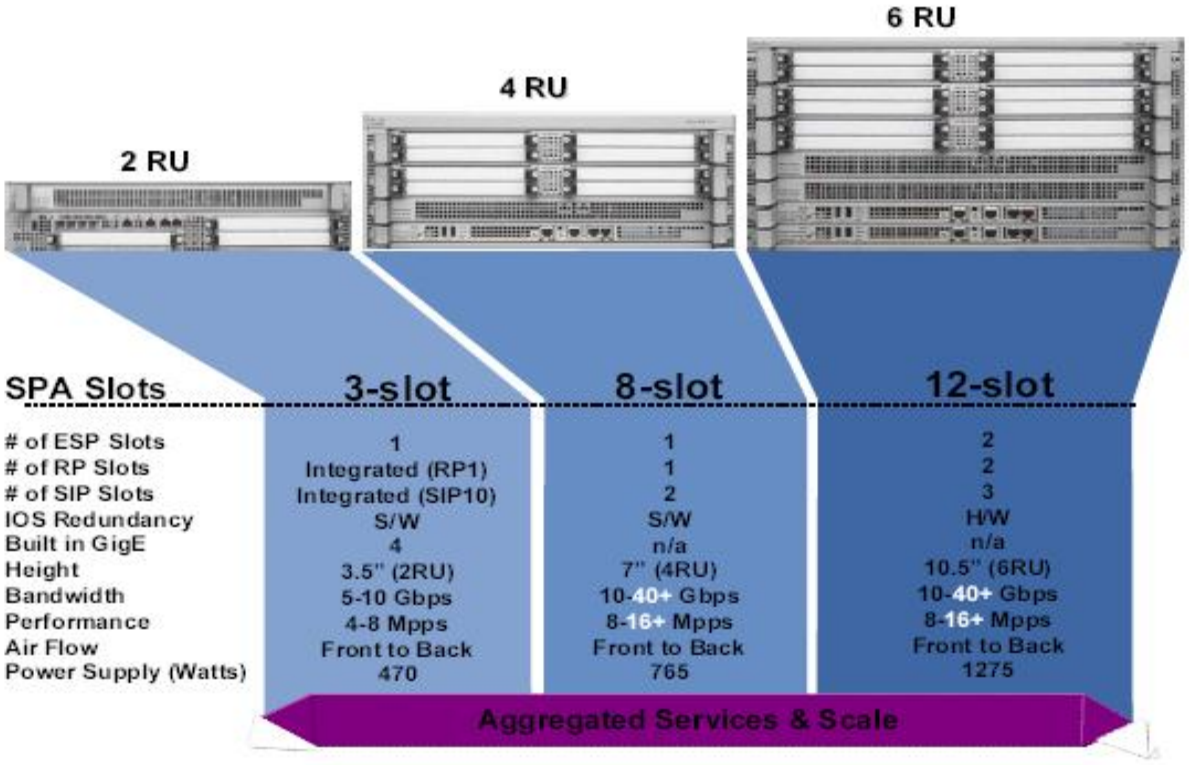


从上图可见，笔者将QFP的160个线程做了一个功能划分--控制线程和数据线程。
控制线程的功能是在QFP的多核上提供与ESP板子上主控CPU（一个PowerPC的处理器）进行通信，例如，接收来自主控CPU通过HT接口发 来的各种控制消息；将QFP的各种状态数据，Log数据等等发回给主控CPU。读者需要注意到的是：FIB表的任何更新路径是：控制平面（RE）通知并发 送给ESP的主控CPU。主控CPU通知并发送给QFP的控制线程。控制线程通过IPC将FIB存储在其能访问的内存中，例如RLDRAM。
数据线程的功能就比较直接和单纯。做纯粹的数据报文处理。
基本上，思科是不可能透露其QFP多核上软件结构的细节。但是，笔者认为，其结构应该大致为上述分析的结果。
在控制线程方面，另外一个工程选择可以是通过中断处理例程来实现，从而可以避免将160个线程中的几个单独拿出来做控制用，从而损害数据处理的能 力。但是通过中断例程的方法的缺点也是比较明显，例如，对线程的cache缓存的打断和侵入，从而导致数据报文处理的性能受到影响。还有的不确定性在 于，QFP的多核的中断处理逻辑的发送机制是如何设计的。是每个Xtensa的核都有一个中断逻辑，还是整个40个核拥有一个中断逻辑。
不同的QFP的微结构设计，都会导致软件系统设计的取舍不一。
另外，笔者感觉QFP上的40个核上的软件逻辑的稳定性和可扩展性非常有可能存在许多潜在的问题。如果没有一个良好的轻量级的微内核的支撑，上述的 “run to complete"的死循

环（while（1）...)代码结构是有许多问题的。当然，从这个角度，读者应该可以认识到，QFP里提供的硬件锁逻辑（提供 许多spinlock，mutex等）是非常重要的。对于性能和编程模型，都是很关键的。
另外，从目前分析的QFP多核的软件结构来看，对于一个QFP的线程处理了崩溃，对其他线程的影响是什么 ？ESP主控CPU如何对待这种异常，是重新启动QFP全部系统，还是做而且只是重新启动那个死掉的线程？从笔者的技术观点，应该是，一旦主控CPU发现 异常，HA的Active/Passive就会切换。这个当前的QFP全部重新启动，从主控CPU拿到QFP的代码，并且从新运行。这个也是非常可以理解 的，例如ISSU的Upgrade等。
总之，在QFP的多核上构筑一个良好的软件系统是不容易的。目前来看，思科的ASR1000上的QFP软件也是比较简单的。
这里面原因有二：
×报文处理引擎，简单就是美。简单就是高性能。
×非常不容易设计和把握。

8. 体系结构--系统观点

思科的QFP芯片，是作为思科的边缘路由器ASR1000的重要组成部分而粉墨登场的。ASR1000产品系列为三个：ASR1002，ASR1004和ASR1006。其不同的配置和结构的差别，可参阅下图所示。



思科ASR1000产品系列

在从系统的观点考察QFP和其所在的系统时，首先介绍一下ASP1000产品的一些命名约定。

CPP: Cisco Packet Processor。在ASR1000里，就是这个QuantumFlow Processor（QFP）

CC：CarrierCard。即线卡模块板。在ASR1000里，叫做SIP（SPA Interface Processor)。

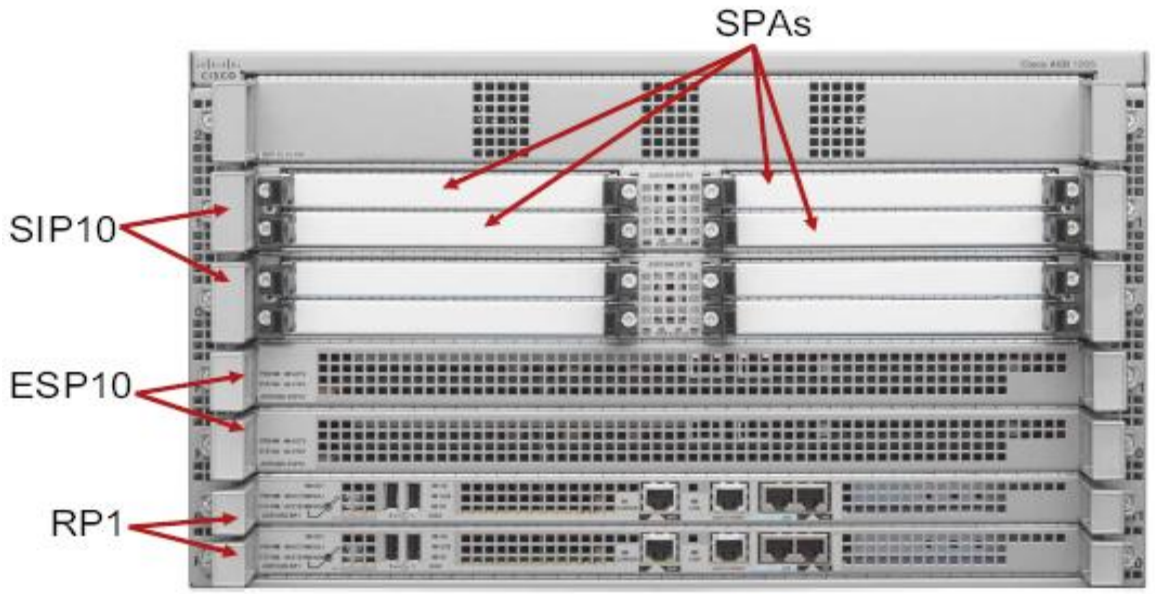
SPA（Shared Port Adaptor)为线卡。 一个SIP模块上可以插入1-4个SPA线卡。支持Ethernet，ATM，POS等等报文格式端口。

FP: Forwarding Processor。 在ASR1000里，是ESP（Embedded Services Processor)卡。QFP芯片就是被焊接在ESP卡上，并通过SP-4.2接口来获取数据报文的。

RP: Routing Processor。在ASR1000里，仍然是叫做RP。就是思科经典的IOS的控制平面卡的意思。

如果读者不熟悉思科路由器产品命名约定的话，要注意的是，不要把这里的一些“Processor”与真正意义上的CPU处理器相混淆。SIP，ESP和RP等的Processor的意思是一个逻辑概念，其物理上是指相应的硬件处理卡。

下图所示为ASP1006的一个硬件结构图。

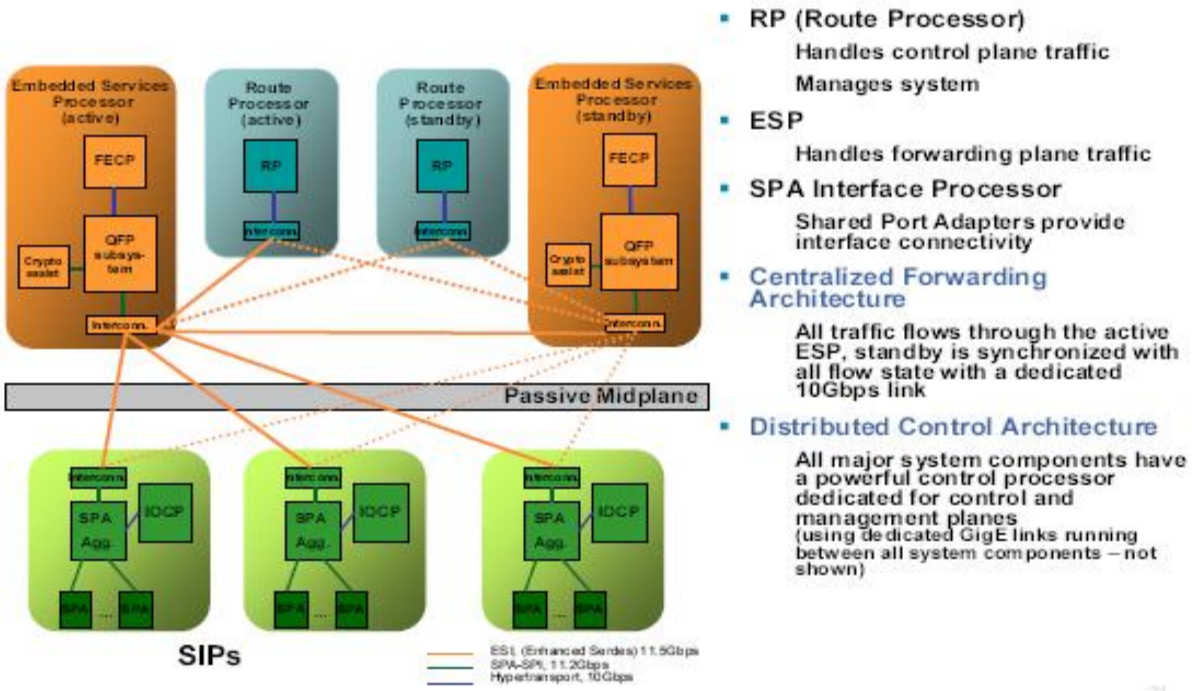


思科ASR1006硬件系统正面图一览

读者可以认识到，ASP1006可以支持两个RP，即控制平面卡，从而提高Dual-RE的HA。（ASP1004可以支持单RE的HA，即两个 IOS运行在Linux的KVM环境上。ASR1006不支持单卡的Dual-IOS结构。）。可以支持两个ESP卡，也就是，两个QF芯片 。这两个 ESP卡的HA关系是Active/Passive。从而为ASR1000提供Statue-ful的服务处理的HA支持。这是ESP10中10的意思是 10bps线速。。ASR1000不同的产品可以支持不同的ESP性能。例如ASR1002的ESP是ESP5，也就是说，是5Gbps的线速。 另外，可以看出ASR1006可以有两个SIP模块。每个模块可以支持1-4个SPA线卡。

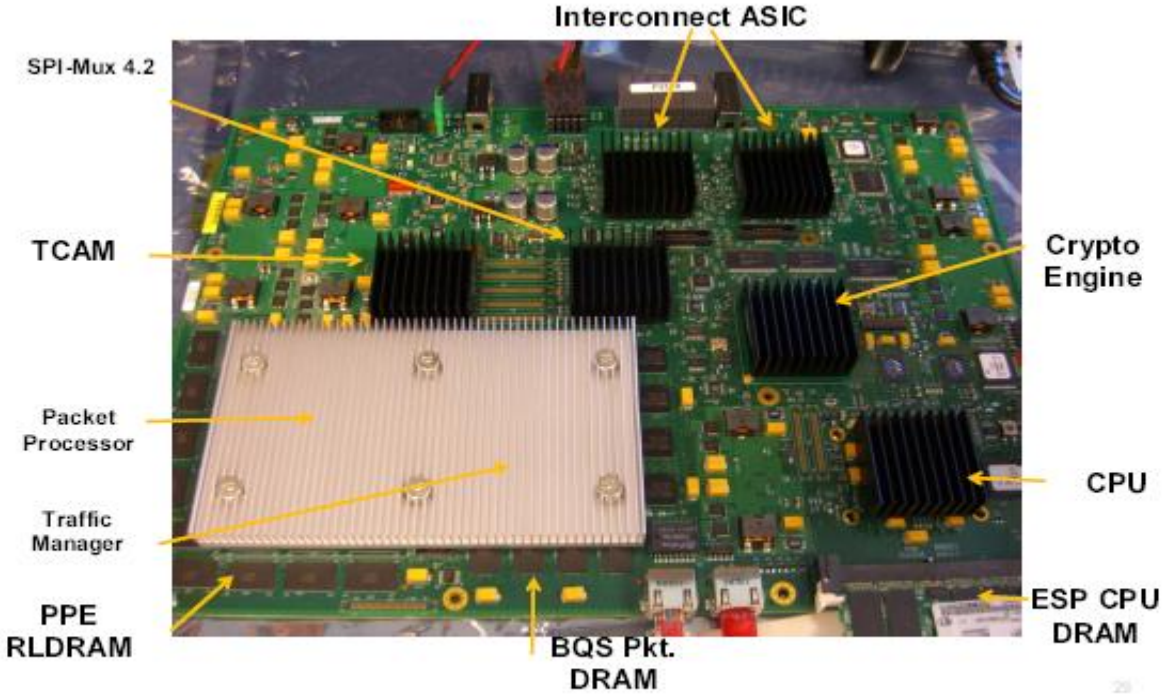
当读者试图把握ASR1000的系统结构时，很重要的一点是：ASR1000是一个分布式结构，但是一个集中式的数据处理。换言之，任何一个数据报 文，都要通过SPA-->SIP-->互联-->ESP，最后进入QFP进行处理。从RE出来的数据报文，例如BGP，OSPF报文， 也是如此。在系统中，QFP起着一个枢纽的角色。

下图所示为QFP，ESP在整个系统中的逻辑结构图。



基于QFP芯片思科ASR1000逻辑结构图

另外，有兴趣的读者，可以浏览QFP在ESP卡上的物理布板结构。从而可以更好的理解QFP是如何 通过系统级别的互联，然后通过SPI-4.2来收发报文，与主控CPU进行通信，被主控CPU监测，启动，更新等等的硬件结构。



思科QFP/ESP版图一览

从上述图中，读者也可以发现，在ESP板子上有两个“Interconnect ASIC"。这个互联逻辑就是思科目前力挺的ESI--Enhanced Serdes Interconnect。ESI互联结构是ASR1000系统把系统的部件（SIP，ESP，RP）结合起来的通信路径。 换言之，ASR1000的背板是基于ESI的。
那么为什么在ESP上有两个ESI芯片，其原因是为了ESP之间的高可靠性（HA）。在一个ESP或者QFP在Active模式下的时 候，Active与Passive的ESP（或QFP）通过专门的一个ESI连接进行通信，从而完成Stateful的状态备份。实现ESP级别的HA。

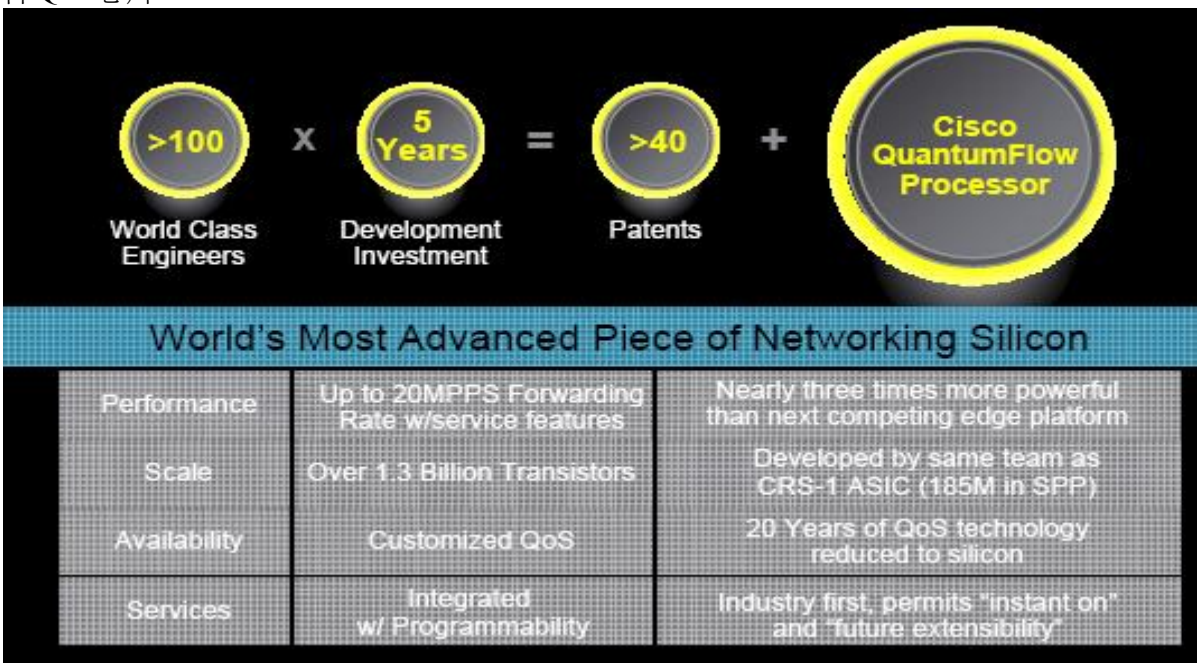
9. 战略规划

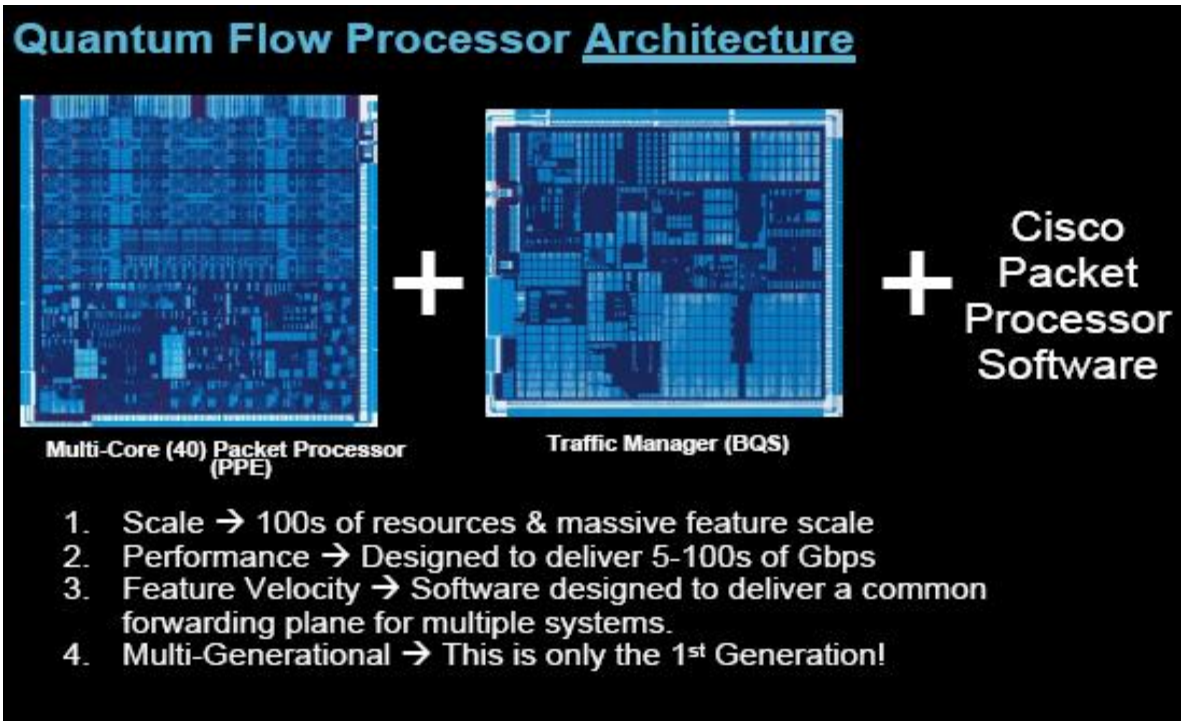
思科的QFP网络服务处理器及其ASR1000产品系列是思科在下一代网络（NGN)战略规划中的重要一环。以笔者的判断，其主导核心思想就是：
×在企业网方面，NGN的发展方向从LAN-Centric朝WAN-Centric的方向发展。 QFP与ASR1000系列就是为了加强思科在企业在WAN接入方面的竞争力。例如，取代其原来的7200系列产品，从各个方面帮助一个企业网的WAN接 入，例如，网络安全，WAN优化，声音与视频接入等。
×在运营商方面，NGN的发展方向是Edge Router的智能化。思科认为这运营方面，竞争的战火将在Edge上，而非在骨干上。Edge的智能化是下一步的发展方向。QFP与ASR1000就是这个战略规划下的产物。
下图所示可以看出ASR系统在思科WAN解决方案中的位置。

从性能参数比较，基本上可以认为，一个ASR1000产品相当于16个7200产品。 可见ASR1000的服务集成度之高，令人惊叹。
在运营商解决方案中，ASR1000的位置为下图所示：

可以看出，ASR1000属于思科智能边缘路由器的重要一环。这个“Intelligent”就是这个QFP所处理的各种网络服务所带来的。

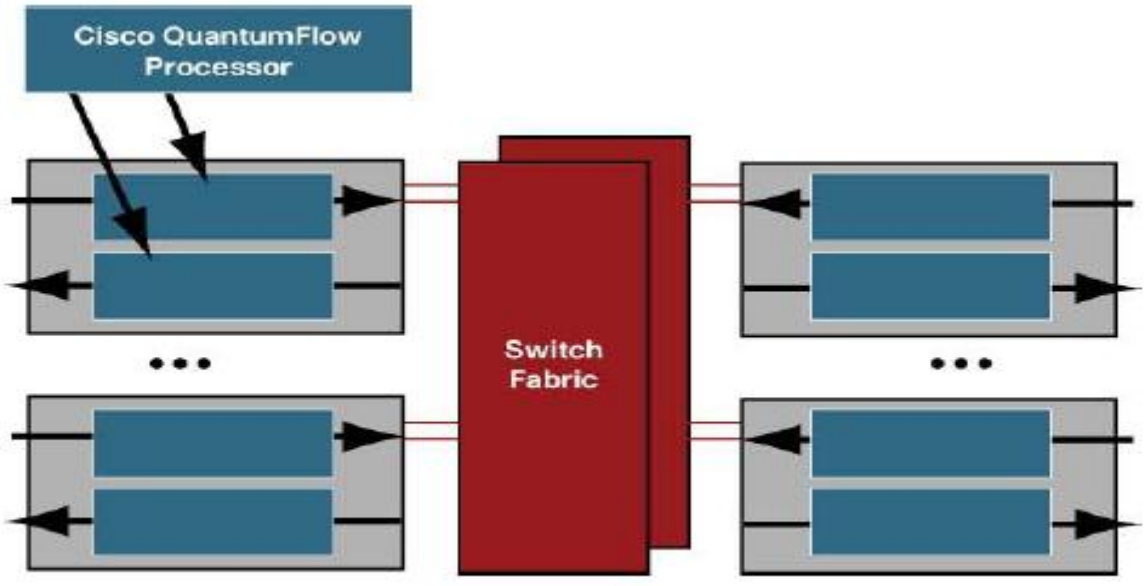
思科 为了实现其“决战在边缘”的战略，投入了大量的力量开发ASR1000系列最重要的部件QFP芯片 。





从思科公开的资料显示可以得知。目前的QFP及相应的系统只是其第一代产品。 在今后的数年，笔者相信思科将会持续推出新一代的QFP芯片发布和相应的系统。例如，20G，40G和100Gbps的QFP。因此，基于相应的QFP的 20G，40G和100Gbps的ESP将会持续的推动ASR系列的性能达到新的台阶。

另外，目前的ASR1000（1002，1004和1006）是采用了集中式的一个QFP（另外一个QFP做备份作用）处理数据报文的体系结构。思 科在将来的系统中，非常有可能将QFP应用在线卡（SIP）上，从而达到一个分布式计算结构。换言之，每个SIP卡模块上支持一个，甚至多个（10Gbps，20Gbps，40Gbps，或100Gbps的）QFP。



思科未来ASR系统--分布式QFP结构

上述的基于分布式结构的QFP计算模式，将使得思科在ASR系列产品性能跨越式发展。其系统整体聚合性能将是目前ASR1006等的4，6，8或10倍不止。
这种结构的下一代ASR产品必将对NGN的WAN接入和边缘路由器的市场起着重要的重新洗牌的作用。

10. 结束语

思科在2008年3月，历经5年之功，推出了其基于40核的网络处理器QFP的边缘路由器ASR1000系列，1002，1004和1006，意在 取代其7200系列。在2008年11月，又推出其的最新的20Gbps的QFP/ESP数据处理卡。思科对于其边缘路由器服务智能化和WAN接入市场高 性能的重视足见一斑。
在核心路由器市场，思科和Juniper几乎占去绝大多数的份额。但是在边缘路由器市场争夺战中，竞争非常激烈。目前，几乎各大设备厂商都有自己的 边缘路由器产品，如思科的ASR系列、Juniper的MX系列，爱立信/Redback的SmartEdge系列，中兴的ZXR10ISR系列、华为的 NE40、20系列。思科的ASR系列产品必将给其竞争对手的相应产品线极大的技术和市场压力。
思科的ASR1000系列中，其最重要的一个技术突破就是QuantumFlow网络处理器的研发成功和使用。从而思科可以贯彻其高性能WAN接入 和智能化边缘路由器的战略举措。思科耗资1亿多美元、投入5年时间研发的QuantumFlow网络处理器，可以将防火墙、IPSecVPN、DPI、会话边界控制(SBC)等多种应用集成在一个系统中，而要在运营商级和终端用户网络的边缘提供这些功能，大约需要6个设备 。
笔者认为，随着思科围绕着QFP芯片不断升级和系统结构的不断复杂，其在WAN接入和边缘路由器市场将持续保持其主导地位，并且拉开相应竞争对手的距离。

居安思危，笔者希望这篇关于QFP处理器及其战略研究的文章能对读者有所帮助。