M1 Exploration CHN

目录

M1 Exploration CHN 1
1 Introduction 3
2 Theory of a modern OOO machine 3
2.1 Introduction 3
2.2 Basic Speculative Superscalar OOO machine 4
2.3 现代 CPU 的设计原则：faster、more、smarter 9
2.4 实验设计 11
3 Register File 初步分析 12
3.1 More OOO theory 12
3.2 实验设计 13
4 Instruction Scheduling 15
4.1 概述 15
4.2 M1 的实现细节 16
4.3 实验设计 18
5 立即数的处理 19
6 Register File 的实现细节 20
6.1 基本概念 20
6.2 上下文切换（减小开销的方法） 21
6.3 Register bypassing and early release 23
6.4 ROB 退休能做到多快 23
6.5 So how close is the M1 to a KIP 24
7 Load and Stores 24
7.1 基础测试：LSQ 的大小 24
7.2 LSQ 的理论 25
7.3 改进 LSQ 的思路 25
7.4 Load Store dependency 的实现 26
7.5 APPLE's implementation of replay 27
7.6 继续测试 LSQ 28
7.6.1 队列深度 28
7.6.2 LSDP 测试 28
8 Load Accelerators 30
8.1 通过 Store Queue 的加速 30
8.2 通过寄存器的加速 30
8.3 通过快速地址计算的加速 30
8.4 实验 31
9 L1D cache 31
9.1 关于带宽的话题 31
9.2 TLB 的实验 32
9.3 LSU->TLB->Cache 32

9.4 SRAM design 相关的话题 32
9.5 Cache 设计相关的话题 33
9.6 Back to the cache data 34
9.7 Wider loads (+non-aligned loads) 34
9.8 L1D 的总线宽度 34
9.9 写通道 34
9.10 延迟 35
9.11 Barriers 35
9.12 多核对内存访问顺序的影响 36
9.13 预取 36
10 L2 Cache 37
10.1 L2 Snoop filtering of L1‘s 37
10.2 Drowsy L2 37
10.3 压缩 cache 38
10.4 Shared L2 and its consequences for shared frequency 38
10.5 Non-inclusive non-exclusive 39
10.6 Manually managed cache 40
10.7 Bandwidth to caches and DRAM 40
10.8 Cache 替换策略 40
10.9 Cache telemetry 41
10.10 Consolidated address translation unit 41
11 Memory controller 42
12 System Cache（SLC） 43
13 大小核的问题（A10） 43
14 DMA 43
15 NOC 问题 44
16 功耗问题 44
16.1 如何省电？调度第一条 44
16.2 如何省电？取指的优化 46
17 后记 47

1 Introduction

本文是《M1 exploration – v 0.70》的阅读笔记，尽力保持了原作者的行文结构，但还是做了一些调整，加入了一些个人见解。

《M1 exploration – v0.70》一文来自网络，原作者是 Maynard Handley。原始格式应该是 jupyter notebook，很可惜我暂时还没找到源头出处。这篇文章的原意是从逆向工程的角度分析 APPLE 的 M1 处理器，但显然这是一篇（很有价值的）个人笔记形式的文章，内容和格式并不十分严谨，在阅读时存在不少障碍。文章大体上可以分成两个部分：第一部分主要关注 CPU 内核，第二部分则主要关注 Cache 和互联结构。第一部分的内容中，原作者通过实验结合专利分析，逆向分析了 M1 CPU 内核的微结构。这部分的工作很有启发，其实验设计思路可以用来分析和评估其他 CPU。第二部分的内容则主要是专利分析和大段的阐述，其中很多内容非常系统（比如在分析取指逻辑及其相关工作的部分），不论是通读还是精读都能获益良多。

通读这篇文章后，读者（我）期望能够建立起现代处理器的基本概念，特别是随着时代的发展，有些教科书级别的概念已经发生了偏移（或者说是优化、或者说是改变），读者很有必要及时更新思路。

本文是“笔记的笔记”，行文风格也按照口语化表达。对于原文中的图表，除了一张基本的微架构图之外，均不在本文中粘贴；对于原文中的所有引文，特别是专利链接，均不粘贴。本文尽力做到“精简”，希望能够用较短的篇幅覆盖原文的精华。

2 Theory of a modern OOO machine

2.1 Introduction

2000~2010 年间发表了大量的关于乱序处理器工作原理的优秀文章。20 年后的今天，设计资源暴涨 1000 倍，过去的优秀设计思路已成为现代设计的基础。

之前的文章中值得推荐的有：Nehalem overview, (2008) https://www.realworldtech.com/nehalem, 和 SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine (2008) https://carrv.github.io/2020/papers/CARRV2020_paper_15_Zhao.pdf。这些文章介绍了 OOO 的基本概念、组成结构等。本文的内容会更加深入和更加现代一些。

出于显而易见的原因（M1 缺少公开资料），本文中存在大量的（有根据的）推测。推测来源主要有三类（相互印证）：

1）学术论文，用来解释什么事情可以做、做的途径、工作原理等。这些论文并不是任何实现的证明，只是用来解释技术的细节和可行性。
2）APPLE 的专利，用来解释设计的创新点和亮点。这些专利有些已实现或部分实现，有些未实现。透过专利可以分析 APPLE 处理器的一些实现思路（和演化过程）。
3）代码实验。通过设计不同的程序并测量其运行时间，可以对处理器进行量化分析。但需要注意的是，程序只能测量运行时间，并不会直接告诉你为何，所以需

要仔细理解数据代表的意思。引用 Richard Hamming（汉明）的观点：“the goal of computing is insight, not numbers.”

读论文和专利很难，但还是要坚持，需要有耐心。技巧是：不需要理解（一篇文章或专利的）所有内容，先读理解的部分，跳过不感兴趣的部分，然后努力读完感兴趣但不熟悉的部分。坚持一段时间后就能理解论文和专利的结构，能够快速的找到需要的内容、跳过不重要的内容。所以加油读吧!

2.2 Basic Speculative Superscalar OOO machine

我们从 2000 年左右的乱序超标量机器说起。先理解基本概念，再考虑如何提升。

CPU 流水线始于产生要装载的指令的地址的机制。

一般的代码中平均每 6 条指令就有 1 次分支（这个数字与代码的类型有关，6 是一个广泛接受的平均值），我们可以假定大约一半的分支指令会跳转（taken），如此一来平均 10 条指令左右就会发生一次 PC 不连续的情况。如果我们的处理器 IPC 为 8（注：M1 的 IPC 设计目标是，文中绝大多数讨论都基于这个数字展开），则我们几乎每个周期都要处理 PC 不连续的情况。

显然我们不能等到上一次分支指令执行完、确定新 PC 之后再继续执行，因为即使在最好的情况下，大约也需要 5 个周期左右才能取新的指令，这显然不可接受。

因此我们需要使用分支预测。CPU 基于历史信息猜测新 PC。使用分支预测的结果是，几乎所有在执行的指令都处在投机状态，也就是说处理器需要将这些指令产生的结果（包括所有的 Store）保存在临时存储中，直到一个能清晰地知道分支是否生效的时点，然后决定这些指令是否转入提交状态。

我们乱序处理指令。令人惊讶的事实是，真实的程序中存在非常多的指令非相关性。表现为两种形式。

一种形式是“立即”并行性（immediate parallelism），一般我们发现，一连串操作中，通常一条指令的后 2 或 3 条指令是非相关的。如果我们把这些指令组成单个“宏指令”，我们就能得到一个顺序依赖关系的“宏指令”链。由于每个“宏指令”大约 2 至 3 条指令宽，那么直觉来看大部分代码的并行度也就是 2 或者 3。但实际上并不是这样。

大部分代码看起来是由一小段顺序依赖的宏指令链组成的，大约 5 到 7 条宏指令，总计 10 到 20 条指令，执行结果通常存在内存或寄存器中，但结果内存或者寄存器一般都需要几百个周期之后才会被访问。这就意味着，虽然每个“宏指令”链都需要顺序执行，但多个“宏指令”链是可以并行执行的。（注：这就是第二种形式的并行性，通过更“远”地看指令，发掘指令流中的并行性）

听起来很不错吧;-）我们需要很多机制来追踪哪些指令是真的与前序指令不相关，同时还需要追踪指令执行的顺序以确保分支正确解析。

这个事实也是为什么很多人对超标量价值的直觉认识是有缺陷的原因。Most people hone their assembly optimization skills on long stretches of sequentially dependent instructions, but such code is actually unrepresentative of most of what runs on a CPU.

这个事实也是为什么乱序超标量能很好工作的原因，同时也是固定宽度机器（VLIW）的尝试为什么总是存在问题的原因。乱序、预测、超标量，协同工作。特别是大多数这些“宏指令”链来自不同的基本块，而不同基本块基本不可能静态地合并。

所以，基本的机器流程是这样的：1）用猜测的指令顺序取指，2）为每个指令顺序分配资源，3）将指令丢到大池子里并且尽早执行（也就是依赖关系满足的情况下），4）然后按整个程序的顺序进行退休。

退休点是所有的猜测都被验证了正确性。如果猜错了，我们刷掉所有的结果从头再来。退休（因为是顺序的）撤销了所有由于乱序执行带来的混乱。那么我们需要在提交之前保存哪些状态？1) Store 状态是显然要保存的。2) Load 状态看起来没多大影响但也是要保存的。3）异常。4）寄存器值。5）预测器的状态（如果在有关联的指令执行时立即更新分支预测器、指令预取器等等，在投机出错的情况下，后面可能得到的是有问题的数据。这推论出另一个反直觉的结果：在数据侧，例如数据预取，这通常并不是个大问题；而在指令侧，如果你想要分支预测器和指令预取器准确度，就必须确保它们没有被错误的投机路径污染。还得确保它们没有被中断污染，因为一般中断服务程序并不在我们的主流程上。）

所以我们需要一组结构来追踪所有这些内容。传统上最大的结构是 ROB，标准名字是 Reorder Buffer，但不太好理解；最好将其理解成 Retirement Buffer，退休的时候做两件事：1）每条指令要确定能否提交其结果到 Programmer state（也就是我们认同这条指令在这个点不是投机的，也没有引发什么问题例如异常）；2）为这条指令分配的所有资源都可以被回收再利用。

因此，基本的指令流处理流程（在理想情况下每一级大约 1 周期处理 8 条指令、不同级并行处理 8 条不同的指令：按 IPC=8 设计）如下：

取指、译码、映射、重命名、粗粒度调度、细粒度调度、执行、退休、提交

Fetch，decode，map，renaming，scheduling，execution，retire，commit

取指：包括了所有的分支预取机制

译码：机器码译码成微指令。这一节强调了下 NOP 的处理是值得关注的。另外，动态链接会导致“international call”实际上是“local call”从而导致大量的 NOP。因此真实代码中 NOP 的数量比预期还要高一些。ARM 定义了一族 HINT 指令（NOP+设置特定位），用于预取、分支预测等控制的提示。

映射：处理寄存器重命名。逻辑寄存器（ISA）映射到物理寄存器（实现）。例如指令 ADD r2, r1, r0；已有映射 r0 – p7, r1 – p45，那么 r2 需要映射一个新的物理寄存器。这里一般采用 single-write rule：物理寄存器可以在指令处于投机状态时一直保持临时状态。只要发生写，就给目标寄存器开一个新的映射。其他指令不可以重用这个物理寄存器，直至临时状态已确定不需要。

寄存器重命名的概念不难理解。映射算法、映射表的更新时机，等机制都不难理解。最难的地方在于：例如指令序列 ADD ra, rb, rc; MUL rd, ra, rf; ra 是上一条指令的结果、下一条指令的源。在 CPI=8 的情况下，不能对 8 条指令独立地去做映射。必须找到指令间的后续名字依赖。这个才是 MAP 最难的部分。APPLE 在很多专利中明确地将 MAP 独立成流水线的一级；竞品中基本都没有专门提到这一级。

重命名：传统的名字，现在更应该称之为 “Allocate”。这一级是给每条指令分配资源。最基本的资源是 ROB 槽。ROB 是顺序程序序列的指令队列。指令序列是投机出来的。每周期到达 Rename 级的指令都会在 ROB 的尾端分配一个槽。（同时，但不同流水级）ROB 的首指令会被检查是否完成、以及正确完成。①如果 ROB 的首指令正确完成了（投机被确认正确、指令执行完），那么资源就会被释放。（FIFO + Ring Buffer 结构）。②如果 ROB 的首指令不正确地完成（投机被确认失败、指令执行完），就执行纠错动作（一般是：刷新错误的分支预取、重新 load/store 等）。③如果 ROB 首指令未结束（指令未执行完），就保持为 ROB 的首指令直至完成。

影响：有些指令执行时间很长（例如开方指令），首指令可能会停在 ROB 中很多个周期，而其他（后续）简单指令早就完成了。当首指令完成时，可能会有多达 80 条指令是被标记为完成的。CPU 只有在首指令完成时才能启动退休动作。这就要求退休动作执行的足够快（同时又存在很长时间不动作…）。

完成时间最极端的例子是：所有 cache 都未命中的 load 指令，这会导致几百个周期的延迟。这时，ROB 只能一直堆在那里。最坏情况 ROB 全满，那么 CPU 只能停顿。

这也就是为什么 ROB 大小代表了 CPU 应对 load 未命中的能力。

假设一个 CPU 宽度 8，ROB 640。如果一条 load 指令占据了 ROB 的首指令位置，则 CPU 的乱序部分可以至少正常工作 80 个周期（假设每周期每一级都可以理想地执行 8 条指令；如果不满的话可以工作的周期数更长一些）。这意味着对于 L2 cache 命中（M1 上大约 15 周期），CPU 不会等待；对于 System cache 命中（M1 上大约 90 周期），大体上也能覆盖住。如果 System cache 也未命中、需要访问 DRAM（大约需要 100ns，300 个周期），ROB 全满，Rename 级无法给新指令分配资源、新指令也就不会进入流水线下一级；（后续）处于 Map 级的指令也无法进入 Rename 级，（后续）处于 Decode 级的指令也无法进入 Map 级别，…；此时直到 load 完成后，新指令才可以进入 CPU。

除了 ROB 槽之外，要分配的资源与指令类型相关。所有含目的寄存器的指令都需要分配一个物理寄存器用于保存结果。传统上就是在 Rename 级完成的（MAP 级只做算法、确定物理寄存器的 ID；但真实的分配动作是在 Rename 级完成的）。Load/Store 指令则需要分配 LSQ 槽。

如果这些资源不可用，流水线就会停顿，直至 ROB 有指令退休、有资源被释放出来。

ROB 看起来只是一个队列，看起来功耗不高，为啥不做大？因为：当其他资源耗尽时，ROB 做大了也没用。

实际上 Renaming 级考虑的最多的资源分配问题是物理寄存器堆和 LSQ。这两个面积大、功耗大、难扩充（即使愿意付出功耗和面积的代价，它们会变慢…如果不能在单周期内访问，性能会有断崖式下跌）。所以就是要在物理寄存器堆、LSQ、ROB 的数值上（在合理的功耗和面积预算下）取得一个平衡的数值。

对比：Intel sunnycove / icelake：ROB = 352 （OOO window），Load Queue = 128 （In-flight loads），Store Queue = 72 （In-flight store），180 定点物理寄存器，168 浮点物理寄存器。

以上是传统的 ROB/物理寄存器堆/LSQ。APPLE 还做了些改进，使之表现更好。

调度：直到 Rename 级，指令都还是顺序处理的。然后指令送到了调度队列中。调度的出发点是：提供一个缓冲，指令分配到的资源准备好前就先等待，一旦准备好就执行。

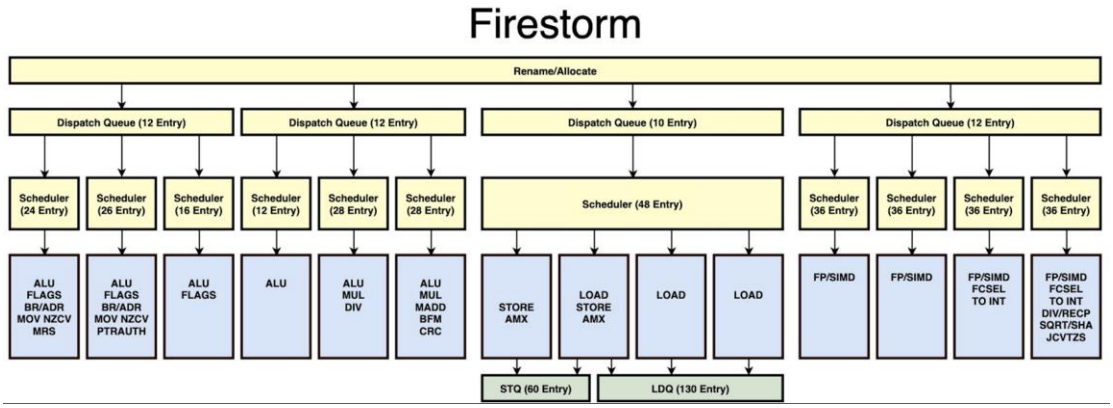
这里的资源除了 Rename 级分配到的 ROB/LSQ/物理寄存器堆之外，还包括：输入和执行部件。例如指令 ADD ra, rb, rc; 在进入调度级后，每周期都需要检查：①rb 是否就绪；②rc 是否就绪；③ADD 部件是否据徐。

调度队列很占面积和功耗。也是 CPU 中的受限资源。指令也可能在调度队列上发生堆积；堆满了也会造成 CPU 停顿。

有一些技术可以帮助确定调度队列的合理尺寸。我们更好奇高层设计：

Intel 的所有指令使用单一大调度队列，很大很贵很耗电，很传统。 其他处理器基本都使用多个调度队列（例如整型指令用一个，load/store 用一个，浮点指令用一个），这样每个调度队列都会短一些，功耗面积都会更好一些，也更容易满足时序要求，但副作用是可能会出现整型队列满，其他队列空的场景。

更极端的做法是：为每一个整型部件设置一个调度队列！这是 APPLE 的做法，例如 M1 有 6 个 integer units，4 个 fp/simd units，就为每个部件设置了一个调度队列。每个调度队列长度不等（12~36 个，Intel 的单一队列长度为 97），但总计数量很高。



（其实这里还要注意的是，所谓的“平衡”是指：指令的分布比例决定了运算单元的配比，Dispatch Queue 做了初筛，细粒度的调度由每个部件对应的 Scheduler 决定）

独立的队列必然遇到平衡问题。所以做了 2 级调度系统：粗粒度、细粒度。粗粒度缓冲区从 Rename 级接收指令、将其以均衡负载的方式放入对应的细粒度调度队列中。图中 Dispatch Queue 就是粗粒度缓冲区。虽然更合适的名字应该是 Dispatch Pool，因为不需要保持指令顺序。

那么调度队列的长度设计为多少合适呢？越多越好？

目标是 8 宽的 OOO CPU，每周期处理 8 指令。那么就需要：①保证每周期有 8 条指令可用；②分支预测很少失败；③load 大多数命中。除了这些之外，我们看一下指令的分布。大多数指令流中，大约 15%分支、10%store、25%load、50%整型。有 FP 指令时，load/store 占比会略微下降、整型占比下降较多，大概就是这么个比例关系。有人分析过 SPEC CPU2017，指令分布大体如是。

一般的指令模式是：每 5 或 6 条指令一个分支，大约 2 到 3 条非相关指令，2 到 3 条相关指令。池子越大、越有可能找到 8 条指令。

ROB 大小和调度队列大小是两个概念。

特定情况下，ROB 大小决定了：发生 Load 未命中、需要访问 DRAM 时，CPU 还能继续执行多少条指令；

一般情况下，指令在 ROB 中等待退休。

特定情况下，调度队列大小决定了：CPU 能向前执行多少条非相关指令。

如果我们假设相关链有 10 条指令、而我们又想做到 8 IPC，可能调度队列 80 比较合适；如果考虑极端情况（超过 10 条指令的相关链），可能希望调度队列长度大于 80。

而 M1 的做法（多队列、短队列、Dispatch Pools 做平衡）的效果比 intel 更好（更远、更省电）。

术语的区别：

将指令放入调度队列中的动作，IBM/APPLE 称之为 Dispatch，

将指令从调度队列中移出的动作，IBM/APPLE 称之为 Issue。

Intel 的描述正好反过来…看术语需要结合上下文。

Scheduling Queue 在 Intel 的称呼为 Reservation Station（保留站）。

相关性如何追踪？

例如：ADD rd, ra, rb; ra/rb 都分配了物理寄存器 PM/PN，但 ra/rb 同时也是前序指令的目的寄存器。那么就需要追踪前序指令。我们会为每条 pending 指令标记一个唯一的不变的编号（即 ROB 序号）。指令如何获得操作数？从寄存器堆。方法很多，先举 2 种。①在指令放入调度队列时、将寄存器堆中的值复制到调度队列中，这要求寄存器堆在这之前就必须准

备好，且要求在调度队列中用额外的存储保存。②指令发射到执行部件时从寄存器堆中读，这允许物理寄存器堆多一点点时间去 valid，也不需要在调度队列中额外分配空间。
这两种方法都有问题：bypass（上一拍已经算出结果了，还要先存到寄存器堆、再从寄存器堆读吗？）

说这么多的意思是：更应当说“当前指令的两个输入是指令 iR 和指令 iQ”；而不该说“当前指令的两个输入是物理寄存器 PM 和 PN”。虽然也确实存在长长的寄存器输入，但能 bypass 的就 bypass。

APPLE 的专利显示，①他们在指令发射时读寄存器堆；②调度时基于指令编号的（称之为 SCH#）

上面说的是指令输入依赖前序指令计算结果的情况。还有其他一些指令依赖关系。典型的就是 replay。例如 Load 指令，其地址由上一条指令产生，那么上一条指令的目的寄存器就是一个典型的相关。但 load 指令要去从 L1D 中取数据，但如果数据不在 L1D 中怎么办？（也就是发生了 miss，load 指令不能当拍返回结果）但我们不能因此就阻塞整个流水线和所有的执行单元。所以这里引入了 Replay 的概念。不同的 CPU 用不同的方法来做 Replay。通常的策略是：①在调度队列中把指令多存一段时间（在指令自己标记完成状态之前，不要移出调度队列）；②允许指令标记为“重试”，例如每隔 5 周期重试一次访问 L1D 的操作。更理想的做法是：增加一个专门的相关类型，这样就不用 replay 了。结合：物理寄存器的标记为可用、ROB 标记为完成这两个策略，是可以实现“L1D 中的 cacheline 已被填充”这样的新的相关性的。（注：原作者在本文很多地方提到了 replay）

执行：反倒没什么好说的。M1 有 14 个执行单元。执行单元可以做一系列操作。事实上执行单元已经不再是最影响效率的地方（注：原作想表达的意思应该是，由于流水线、乱序、相关等复杂的内容的存在，一个一般意义上设计的执行单元在指令的执行全过程中的占比已经比较小了）。可能想要更多的执行单元、可能想要更快的执行单元。假如 FP 乘法从 5 周期降低至 4 周期，很好，但在 M1 这种级别的 CPU 上，性能提升只有一丁点儿。M1 很快是因为它很宽，非常宽，非常乱序，而不是因为每个执行单元比 ARM/x86 更快。

CPU 的“宽度”代表了什么？

业余的说法是发射数量？也就是每周期有多少条指令可以喂给执行单元。

对于 M1，14 个单元，每个单元有独立的调度队列，所以在理想情况下，单周期 14 条指令可以发射给执行单元。

严肃点的答案是：最大可维持的吞吐量。在 M1 上是 8，而不是 14。

过去我们说宽度 4 的 CPU，是每一级 4 宽，因此取指模块每周期弄来 4 条指令，译码模块每周期译码 4 指令，可能用单一调度队列每周期最多发射 4 条指令（即使有 5 个 6 个执行单元），每周期退休 4 条指令。

这些设计的观点，仍然基于最初的微处理器设计（单指令一级一级流动）：假设 CPU 把 4 条指令组成一块，一级一级的流动。但在我们有更多的资源（晶体管）后，这已经不是合理的模型了。

我们更应该把 CPU 看作是一组 jobs，jobs 之间用队列连接。因此 Fetch 负责把指令丢入 Instruction Queue，Decode/Map/Rename 把中间结果丢到（不同层级的）调度队列，Execution 部件把结果通过 ROB 队列丢到 ROB 中。（注：这一段应该算是原作者理解的现代 CPU 的核心思维）

理想情况下，每个队列都是动态的（增长、减少）。如果某一级暂时的慢下来（如 Fetch 未命中了），下一级能够在队列排空之前还工作一会儿。

按照这个设计观点，每一级的宽度已经不是常数了，因为不存在一级处理一个 4 指令块的情况了。你可以把每一级都做宽。更宽的级会更快的吃掉上一级的队列。对于执行级，最

宽可以同时做 14 条指令（bypass，调度，etc）。

取指级很容易做的比 8 更宽。一般取指级可以单周期取 16 条指令（假设 1 个 cacheline 按 128B，指令长度 4B，则指令数 32），但大多周期内这不太可能，因为基本块比 16 短，或者基本块跨了 cacheline。因此，为了维持**平均 8，**我们就必须一有机会就取的比 8 多的多。同样的，为了维持**平均 8** 退休，我们也得一有机会就退休的快的多才行。

从另一个角度看，在过去资源受限的时候，设计 CPU 的合理的途径是针对代码的平均属性，例如大约 15 分支、10store、25load、50 整型，进行设计。但是当你资源更多，你就应该去尝试极端情况。例如，平均 25%的指令是 store 并不意味着每 4 条指令就包含一条 store，事实上是很长的 store（或者 load+store）序列紧接着很长的没有 store 的指令序列。因此更好的设计要能够处理这种情况。这也是设计了这一堆队列的目的。通过提供了大量的队列，APPLE 能够快速的将指令丢到合适的队列中去等待执行。目标不应该是将相同数量 N 的指令从头一级一级的处理到尾，而是在每个队列保有足够的容量能够保证出现意外情况不停顿。

退休：退休包含三方面的含义。①完成。即结果可用。完成主要意味着 ROB 槽中的一个标志被设置（表明指令已获得结果），但传统设计中所有在 Rename 级获得的资源还没有被释放。②退休。指令被顺序确认，保证所有事情都正确：投机正确，没有发生异常，等等。③提交。将指令从投机态的存储移至非投机态的存储。一般处理器中这被认为是很 boring 的清理工作，和优化没什么关系。但这是很短视的，看看 M1。

一般的设计逻辑是：先明确要做的工作，只要工作确定，就可以将其分解成确定的数据结构和确定的时序。M1 处理退休的方式与其他处理器明显不同（与 IBM 倒是相似）。①理想地，应当在指令完成时释放资源。但有时不可能，特别是资源还在投机存储中，只有退休完成了才能够到非投机存储中。提交动作与退休在同一拍做完是可能的，也可能需要额外的一拍（通常不可见，因为几乎没有操作依赖于此）。②将 store 指令移至非投机存储采用从 StoreQueue 到 L1D 的形式，x86 会在退休后尽快做，但 Weakly ordered 内存模型的处理器会将写操作延迟一段时间（注：为了做写合并或者重排序，以优化内存访问）③有目的寄存器的指令，寄存器的值应当在提交时从物理寄存器写入逻辑寄存器（只是实际上并没有一个一一映射的物理寄存器堆能够对应逻辑寄存器，现代的做法是，在一个给定的时间，用一个索引表和一个大的物理寄存器堆来确定逻辑寄存器的值。因此所谓的从物理寄存器写入逻辑寄存器的行为实际上只是修改物理寄存器）

以下的做法是可行的：①当 store 确定非投机时立即将 store 提交到 L1D，这样更早释放 LSQ。即使 ROB 首指令还在阻塞，只要首指令和 store 之间没有未决的分支，就不应该阻止 store 的提交。M1 就是这么做的。②更早地释放寄存器。如果一个物理寄存器的逻辑值被覆盖，并且没有相关，并且寄存器不再是投机的（与①相似、没有未决的分支），那么就可以重用这个物理寄存器。M1 没有这么做。这事情需要额外的 book-keeping 机制。可能在以后的 CPU 上会这么做。

总结下来就是："投机"取指、顺序译码、顺序 Rename，混序 Dispatch，乱序执行，顺序退休。

2.3 现代 CPU 的设计原则：faster、more、smarter

Faster：三条路：高主频、（不完全等价的）单操作低功耗、多核。

More：cache 好，那我们就搞多点。物理寄存器，搞多点。队列，搞深点。即使不考虑实现代价，提升也不是那么的明显。以 Intel Skylake 为例，32X 的 OOO 资源大约只能换来 2X 的性能提升。盲目地增加资源效果不好，我们需要知道要提升的点。

You need to attack every single one of the pain points, and you need to keep redoing it every few years as many more design options open up with more transistors.
参考资料：Intel Sunny Cove vs AMD Zen 2 Core Architectures: 10th Gen Ice Lake vs Ryzen 3000 - Hardware Times
APPLE 的设计似乎更有扩展性。
Smarter：更“聪明”的设计，才能得到更高的效率。
Guess Smarter：
预测逻辑需要更聪明。现代 CPU 在分支预测方面做了非常多的努力。但还有更多的事情可以做：①load/store 地址的投机（更早的猜 load，在 pending 的 store 之前）；②调度投机（猜 load 操作要多久才能返回，然后基于猜测调度相应数量的非相关指令；③路预测+drawsy cache（主要用来降低功耗）。
APPLE 早期有一些预测器：预测 load 会不会部分覆盖最近的 store；load 是否对齐；预测 NOC 的拥塞程度。预测器的设计原则是：Discover a pattern that is common in real execution, and try to make that common pattern faster or lower power.
预测器还需要关注可信度。这与预测失败的恢复的代价有关。
还有一个新概念是紧急程度调度（Criticality-based optimizations for efficient load processing）目前还没有产品化，也许未来能见到。
相关的话题还有预取和 cache 管理，都是很大的主题。
Work Smarter：两个相关的设计原则：resource amplification and task disaggregation. 资源强化和任务分解。
Resource amplification is about making existing resources go further. 这个思路很多处理器都在用。用更好的算法、更精细的管理来提升资源的有效容量。
例如固定容量的 L1D Cache，更好的 cacheline 替换算法可以让 cache 有效容量变大。这就是 amplification。用更好的算法让有限的资源提供更大的能力。原作者认为 APPLE 大概用了通常 1..2 宽流水线的资源提供了 4 宽流水线的 load/store/tlb/cache 的效果。
Resource lifetimes：资源在 Rename 级分配，在 Retire 级释放。以前是最优的，现在要换种思路考虑。例如一个目的寄存器，如果它的延迟很长（例如 load 未命中），实际上它只在操作结束的时候有用，如果很早就分配的话就存在浪费。LSQ 槽也有类似的问题。为什么以前要这么做呢？因为要保证指令顺序和依赖关系。看起来没什么好商量的，但实际不是这样的，技巧是认识到资源分配实际上是两个任务：①storage②分配 ID。真正的 storage 操作代价较高，但只提供 ID 的话代价很低。使用 Virtual Register 和 Virtual Load Store Tag 可以解决这个问题。这样的话，分配资源的时机可以延迟到很晚（真正使用前），这样只需要占用一小段时间。（晚分配）。另一个思路是早释放。因为实际上很多情况下物理寄存器可以很早就退休。（early register reclaim）
APPLE 管理寄存器文件和 L2 的方法是这个思路（less resource amplification）的变种。把寄存器堆和 L2 切分成独立的小块，这样管理起来会更有效率一些。（注：看起来这句话放在这里有些前后不搭，后文有较为详细的阐述）
Fusion：融合：用一条指令（增加些微代价）获得更多的结果。两种实现途径：在指令集层面、在硬件实现层面。（在指令集层面直接提供这样的指令）ARM64 指令集提供了一组指令，例如 ADD+shift。（CISC 的思路）需要保证附加的动作要足够轻量级。（在译码阶段由硬件自动做融合）始于 1996，将指令流中前后两条或几条符合一定模式的指令做融合。但商用不多，即使是 APPLE 也只针对一部分指令做了融合。仍然是值得探索的领域（公开的文档不多、实际实现的也不多）。
Fusion 也是 resource amplification 的一种手段。APPLE 似乎还想将它用来减少延迟。

APPLE 的专利提供了一些有意思的点，例如①fusion 在 Decode 阶段探测，在 Map/Rename 阶段实施；②实现时两条指令中修改一条、丢掉一条；③ROB 槽的占用情况没有变，但调度队列的占用会少一些，出结果的时间也会快一拍。
未来可能的优化的点是让 fusion 做的更早。例如：①增加 pre-decode 阶段，在代码装到 L1cache 的时候就预先做 fusion 分类；②译码之前做真正的 fusion。这样可以增加译码的吞吐率，也能省一点 ROB 槽的占用，并且功耗也会少一点。APPLE 已经有一些专利在说这些。
Task disaggregation：this refers to splitting a task that's traditionally considered unitary into multiple pieces. 把传统上认为的独立任务拆分成小片。例如，考虑一个位于内存同步指令之后的 load 指令，看起来没什么优化空间。但把 load 拆开：一片是真实的"将值装进寄存器"，另一片是获取值（从内存）到 cache 中。那么完全可以先做后一片，在同步指令之后做前一片。（当然需要仔细考虑同步对于每一片的语义）。Store 也可以做类似的事情。当然了，不能过早的将值写入 cache，但可以 preload 目标的 cacheline，同时通知其他 CPU 对这一行的互斥访问，即使在能够确定为非投机之前也是可以做这个动作的。（写分配、写回、多核的 cache 一致性）
还有很多地方可以这么用。例如 APPLE 把退休动作分解了，一片是释放 ROB（最多在一拍释放 56 个 ROB 槽），另一片是释放寄存器（复杂的多，大约一拍最多释放 16 个寄存器）（下一节 Register File 中提到 History File 与这个思路有关）
还有一个场景：指令申请资源的时候，也可以分解为 2 部分（例如，有些指令的实现是要申请一个暗含的第二目的寄存器，同时申请表面上的目的寄存器），这样把指令一分为二的方法称为 cracking，是个常见的做法。APPLE 有时又会把这两部分合起来，资源分配、调度、执行三个步骤分开。（注：这一小段没有看明白原作者想表达的意思）

2.4 实验设计

设计实验需要考虑会同时发生的影响因素，反复迭代直至（有时甚至不可能）清晰。
例如，想测指令的延迟怎么做？
例如，想测吞吐率怎么做？（一拍内能做多少个同类型的非相关指令）
原理简单，写一串指令，循环若干次，读周期计数器，除，得出结果。
但需要考虑到类似 ROB 容量等因素的影响。（注：cache 缺失可能使结果产生偏差，因此设计时需要考虑的细致一些）
ROB 容量：①用 NOP 指令序列测；比如 1000 个 NOP 要多少个 cycle，4000 个 NOP 需要多少个 cycle。测得结果为 8。（基准值）②增加 8 条 FSQRT.D，每条指令 13cycle，做 N 个 nop+8 个 fsqrt.d 的指令序列。（注：这一段这样理解就够了：由于 FSQRT.D 很慢，而且从运算单元架构设计上看，只有一个运算单元能处理 fsqrt.d，那么 8 条连续的 fsqrt.d 就会有 8*13 个周期的延迟。那么在 ROB 满之前，延迟由 FSQRT.D 决定，ROB 满之后延迟由 NOP 决定。这里还有一个假设是 FSQRT.D 的运算单元是不能流水操作的）③FSQRT.D 指令增加至 32 条，绘图…④FSQRT.D 指令缩减至 22 条，绘图…结论能看到：4000NOP 时，500+13，delay 始于 2228cycle；大约是 ROB 满，新的指令压不进去的点。
代码这么写的：

```
For(;;) {
    Fsqrt, fsqrt, fsqrt, …. 8/22/32 个
    Nop nop nop nop……N 个
}
```

实验能得出，ROB 大约能存 2274 条指令，可能是每行 7 个 slot，一共 324 行。（2274 此处

实验即可得；7slot/row 的证据要到后面的实验才能给出）
小结：这里测试原理是构造形成如下结构的代码序列：Delay block + op block；而 delay block = D time, op block = N ops in O time。例如这一节的实验是把 fsqrt.d 当作 delay block，因为只能串行，而 nop 当作 op block，因为可以并行。然后调整 N 的数量，观察执行时间的变化，直至找到一个点，新指令刚好压不进去。那么这个点就是 ROB 满的时候，从而能测得 ROB 的容量。
（注：这里提的数字和之前推测的数字其实不一致的，前面一直在说 M1 的 ROB 是 640 条指令，这里又在说 2274，具体原因是 ROB 的组织形式，也就是 7slot/row，需要继续往下看。本文毕竟是笔记的形式，阅读过程本身也有抽丝剥茧的乐趣。）

3 Register File 初步分析

3.1 More OOO theory

资源耗尽会导致停顿；例如，整型物理寄存器耗尽了，就会导致后续的整型指令不能 map，由于这个阶段还是顺序的，实际就会导致整个指令流停顿。
原作者这里又强调 OOO：如果存在 20 条指令相关，那么就需要向前看更多的指令；指令越多越容易找到非相关指令。
关于退休、写回、完成。现代处理器的概念基本上只剩下退休了。
Writeback（写回）：将物理寄存器的值写回到逻辑寄存器中，这是在 2000 年以前的论文中出现的术语。非常早期的 OOO 的思路是：保有一个逻辑寄存器堆，物理寄存器堆当作临时变量用；分配 ROB 时同时分配物理寄存器堆；写回就是把 ROB 槽对应的物理寄存器堆写到逻辑寄存器堆中；如果投机错了，那就回退到正确状态的逻辑寄存器堆中。这就要求物理寄存器堆的数量与 ROB 数量相同。这太浪费资源了，而且 FP/SIMD 没法解决。对于没有（目的）寄存器的指令这就是浪费。（注：意思是说写回是因为物理寄存器堆和逻辑寄存器堆的概念带来的，这个概念的含义发生了变化，从而写回的概念也发生了变化）
后来的设计中，已经把物理寄存器堆和 ROB 解耦合。但仍然需要写回操作。
再后来，就不要逻辑寄存器堆了，CPU 的状态只存在于物理寄存器堆和（物理寄存器与逻辑寄存器的）映射关系中。（2004 年的论文：An analysis of a resource efficient checkpoint architecture）
原作者认为一个更好的设计是目前只有 APPLE 采用的，就是用一个独立的 History File 来记录 mapping table 的变化，将 ROB（指令队列）和“寄存器映射的历史”解耦。（其他 CPU 的做法是：History File 是 ROB 的一部分）。解耦的好处是：①各搞各的优化；②Retire 时分别释放（ROB 可以一拍释放 56 条指令，History File 一拍释放 16 个寄存器）
原作者认为另一个更好的、APPLE 都还没实现（但是快实现了）的设计就是 Virtual Register。（注：IBM 似乎实现了？）
Complete（完成）：除了 store 指令外已经完全没有意义了。在过去，退休一条 store 指令时，store 要从 Store Queue 放到 L1D 中。在现代一般设计中，对于 store，执行级负责计算 store 的地址，然后将地址和数据放到 store queue 中；当 store 能够确定为非投机状态时就会从 store queue 放到 write buffer 中，这时候 store queue 就可以释放了；但 store 仍然未真正意义上完成，只有过了一段时间后真正写到 L1D 并且（通过 cache 一致性协议）被其

他核看到才能算完成。所以你要怎么定义“完成”？所以即使对 store，完成级也是没有意义的（没有需要确切要做的动作、且其状态与流水线其他阶段的关联并不大）。
所以简单点说，退休就是释放资源，重命名就是分配资源。（注：寄存器文件是重要资源之一，OOO 中要处理的重点之一就是资源的分配和释放。）

3.2 实验设计

堆砌整型指令，直至延迟出现跳变，跳变的点就是物理寄存器分配不了的时候。（但还需要排除 ROB 满的影响）
History File 是独立出来的（注：这里的独立应该指的是相对物理寄存器堆是独立的，而不是相对 ROB 或者别的什么部件是独立的）
整型物理寄存器堆大小:
基准：堆砌非相关的 ADD 指令，大约得到 6adds/cycle （6 个 ALU，与架构图能对应上）
增加 Delay block，（与上一个测 ROB 的代码相似，就是把 NOP 替换成 ADD，Delay 用了 7 条 fsqrt.d），跳变点大约在 378 条 ADD。
Delay 太短的话，结果也是能有其他解释的。
实验结论是大约 384 个整型物理寄存器。
FP:
用 FABS 指令，首先大约是 4fabs/cycle（4 个浮点部件，与架构图能对应上）
实验结论是大概 432 个。（Delay 需要改用 UDIV 这样的整型指令）
整型+浮点:
FABS+ADD；这样的结论是大约 624..这可能意味着 FP 和 Integer 有一部分是共享的。
Flag 寄存器:
用 CMP 指令，3cmp/cycle，（3 个 FLAGS，与架构图能对应上）
大约 128 个寄存器
整型+浮点+Flag 寄存器:
用 600 组（每组 add + fabs + cmp），共 1800 条指令，由于前端 8 宽的限制，大约 225cycle。这是个限制。
下降到约 2.67 组指令/cycle。增加延迟后继续实验。
通过一系列数据，得出的（表面上的）结论是：物理寄存器堆应该是个共享的结构，总量在 620 左右，由 flag、整型、浮点共享。单独分配的上限大约是 128flag，384 整型，432 浮点。
History File:
那么 620 是不是真的？APPLE 在 2019 年的专利：Last physical register reference scheme 中描述了 HF，使用 HF 来存放寄存器映射表的变化序列。HF 大约 620 个表项。
Intel P6 的 ROB 很简单，ROB 和物理寄存器堆是相同的结构；分配一个 ROB 表项就是分配一个目的寄存器。对投机的处理是使用一个独立的物理寄存器堆（称为退休寄存器堆）来代表“CPU 退休时的真正状态”。所以（如果投机失败）恢复时很简单，只需要把所有的映射表指向退休寄存器堆即可。这种策略的缺陷在于，ROB 和物理寄存器堆的容量紧绑定了；而且在退休时需要真实的拷贝动作，功耗很高。
Intel P4 的 ROB 使用了独立的物理寄存器堆，和一个独立的 RAT（Retirement Mapping Table），RAT 在退休时更新。这种策略解耦合了 ROB 和物理寄存器堆，且更新 RAT 比拷贝寄存器功耗更低。P4 策略比 P6 策略更好一些，但好的不够。
考虑分支预测失败、需要恢复的情况。P4 的恢复需要做如下一些事情：①分支之前的指令还是得退休；②前序指令退休时更新 RAT；③分支预测失败，所以要把 RAT 拷贝到前端的

映射表。这也就意味着：从我们发现分支预测失败了到分支指令退休这一整段时间都是 Dead time。（因为错误分支的后续指令还是进了流水线，在一级一级走）。我们更希望在发现分支预测失败的时候就赶紧处理，而不是等到退休的时候再处理。①分支之前的指令仍然执行；②分支之后的指令都得被 kill 掉；③刷掉错误指令、从新地址重新取指；④新指令直接开始执行，即使这个时候分支之前的旧指令还没全部退休。
如果我们采用这种策略，除了需要机制来标记和刷新指令外，我们还需要随时获得正确的映射表。这样我们就不能等到退休的时候才去取 RAT；我们需要构建正确的映射表，至少得保证新的指令流到达 MAP 级时能正确执行。我们不知道 Intel 后来怎么做的，他们只是在 Nehalem 中提到“we have a fix”。
所以 APPLE 采用的 HF 能够记录变化过程，这样可以随时回滚；而且也可以标记哪些物理寄存器可以释放。

HOW do “set flags” instructions, like ADDS, modify the history file?
ADDS 指令有 S 后缀，意味着还会同时修改 flag 寄存器。实验证明，这样的指令会占用 2 个 HF（目的寄存器，flag 寄存器）
依照之前的推测，HF 只在寄存器重命名时分配新的 slot。那么可以期待：①store 不占槽；②PRFM 这样的显示预取指令不占槽；③写特殊寄存器不占槽；④标准 load 指令占 1 个槽；⑤load pair 指令占 2 个槽（使用 2 个目的寄存器）
关于特殊寄存器，一般认为，特殊寄存器的写操作需要等到投机行为都结束之后才可以执行，这样的串行化处理是因为投机失败的代价太大。APPLE 似乎对特殊寄存器做了分类，不同的寄存器做不同的处理，总之是尽量早写。
指令也有不同；在不同级可能有不同的期望。例如 ADDS 和 LDP 这种双目的寄存器（或者单目的寄存器+单 flag 寄存器）的指令，可能希望在 Rename 和 Retire 阶段被当作 2 条指令来处理，在执行级当作 1 条指令处理；而 ADD（shifted）这样的指令可能希望在 Rename 和 Retire 阶段被当作 1 条指令，在执行级被当作 2 条指令处理。APPLE 就是这么实现的：把多目的寄存器的指令在流水线中拆成单目的寄存器的微操作、在执行级合并。对于拆成 2 个执行级的指令，很多情况是不用额外分配物理寄存器做中间记录的，但个别指令如 EXTR 却分配了。

ROB duplicate registers （mov xn, xm）
APPLE 提供了 0 周期 mov 操作。想法很显然，mov 操作应当只需要更新寄存器映射表，而不需要真正做执行级操作。难点则在于，要正确记录，以便可以正确释放物理寄存器。
寄存器重命名的中心问题是：重命名的寄存器何时可被重用。
物理寄存器可以不再使用的时机是：①对这个寄存器的写入操作执行了；&& ②对这个寄存器的所有读操作都执行了； && ③与这个物理寄存器映射的逻辑寄存器的映射关系已经被覆盖。
XZR 寄存器：（值恒为 0 的寄存器）
Mov xn, xzr；实验结果表明，这条指令看起来用了 ALU，大约只能做到 6movs/cycle。
更有效的方法应当是 mov xn, #0.
Mov x0, x2，大约 hit 了 620；斜率大约是 8，说明只受 rename 限制，而不是 alu 限制。（注：而且 620 说明不是受物理寄存器数量限制，物理寄存器数量是 380，如果是物理寄存器的限制的话，跳变点应当是 380）
Mov x0, x0，被当作了 nop，大约 hit 了 880（基本上只受限于 ROB 大小；实验中出现的 880 是因为设置的 delay 是 110 个 cycle，IPC=8 意味着 880 条 NOP 就会出现跳变点。增加 Delay

能看到跳变点后移，具体可以看第一个实验）。浮点和 SIMD 寄存器（例如 MOV.16B v0, v1）与整型情况一样。如果一条 MOV 加一条 MOV.16B，得到的指令对的跳变点是 310。这一切都证实了 History file 的容量就是 620 左右。
RDA：Register Duplicate Array，用来记录寄存器的多次引用，一般有容量限制。APPLE 在此有专利（但没有实现）。测试程序这么写：Mov xi, xi-1; I = 1..29; 重复 N/15 次；这样与 N 次的 Mov x0, x2 指令数相同。结果能看出，HF 在 620 左右，斜率仍然能保持 8。用其他的写法、其他的寄存器仍然得到相同的结果。这就能证实：M1 的实现中没有用 RDA。
Subregisters：（64 位寄存器拆成 32 位寄存器使用，例如 mov wn,wm; mov.8B vn, vm; mov dn, dm 等，典型的问题在于高位如何处理等）。这一段讨论寄存器重命名的池子该怎么设计最优。原作者认为统一的 integer/SIMD 寄存器文件、宽度支持 256b 至 32b 的应当是最优设计。

4 Instruction Scheduling

4.1 概述

dispatch queue 实际上是个 dispatch buffer，因为不需要考虑顺序，成本会更低一些。
（原作者推测的结构、实验结果能支持的）：48 表项的 Load/Store 调度队列实际上应该是 4 条 12 表项的队列；其他的调度队列实际上应该是 2X 大小（例如图中标记 FP 队列应该是 36 表项，实际大小应该是 18）。每一个队列喂一个主执行单元；但队列是成对的，当一个执行单元空闲时，它还有可能吃第二个队列的可运行指令。（注：这样是用更少的物理资源获得 2X 的逻辑容量）
退休队列包括：合并退休队列和重命名退休队列。合并退休队列约 334 表项，重命名退休队列约 623 个表项。（334*7=2338；rename retire queue 此处指的就是 History File）
合并退休队列大约有 334 个表项，每个表项包含最多 7 条微操作，但每个表项的 7 条微操作中只能有 1 条 load 或 1 条 store 或 1 条分支。在 firestorm 上，如果一周期有 4 个 load/store 和 3 条分支，那至少要占用 7 个表项。
退休能力大约是一拍 8 个表项，也就是大约 56 个 nop 或者 8 个 untaken 的分支。
实验大概的原理：在把物理寄存器堆耗尽的情况下尝试执行 MOV。
例如，N 个 MOV 是斜率为 8 的直线；370 个 ADD+N 个 MOV 是直线+跳变+直线，跳变点在 370-397 处的小段直线。分析下来有如下结论：①Rename 大约是 8 IPC；②这 8 条指令都能被放入 Dispatch buffer（注意结构，这里说的是 dispatch buffer，整型 6 个运算部件，分 2 个 dispatch buffer，每个 buffer 12 表项），但只有 6 条能出队（因为只有 6 个执行部件）；③这样每周期 Dispatch buffer 大约会填充 2 条指令（8 进，6 出），直至满；④整型这边的 dispatch buffer 大约 24 容量，这样很快就满了；大约需要 12 个 cycle；⑤一旦 mov 指令开始执行，一开始这些 mov 的执行周期为 0，更准确的说是他们是在 Rename 阶段执行的；⑥所以大约有 24/6 = 4 个周期的 nop 是不会增加我们的执行延迟的；也就是大约 32 个 NOP（这里这么理解：因为是在 Rename 阶段就能完成 mov，那么 mov 实际上是不会进入 dispatch buffer 的；dispatch buffer 排空需要 4 个周期（24/6），而 Rename 的能力大约就是 8 IPC）。差不多这就是跳变出第一段平线的原因。至于为什么用 397 个 ADD？其实也不是那么必要，只要>12cycle * 8 IPC，大约 96 条指令就够了。
换一个测试程序：370 个 MUL+N 个 MOV。一开始大约是 2 IPC（只有 2 个乘法器）；平线

大约是 50 个 NOP，这是因为喂给乘法器的 Dispatch buffer 只有一个，容量为 12，12/2 = 6cycle，大约就是 48 个 NOP，数字差不多能对上。

4.2 M1 的实现细节

独立调度队列：

调度队列应该很耗电，每拍都要扫描全部表项、判断指令是否可运行。因此将大的单一队列分拆为多个小队列可能更优一些（虽然会存在浪费队列空间的情况），总量会更好、功耗会略低。显然，整型、load/store、浮点天然就可以分开。

Dispatch buffer：

两级调度，Dispatch buffer 就是所谓的粗粒度调度。AMD 也这么做的。Dispatch buffer 不能太大（调度队列总容量的问题最好由独立调度队列解决；如果调度队列满了，再次流动时有可能从 Dispatch buffer 中随机把指令分给调度队列，分配不合适的话有可能造成延迟）。有了 dispatch buffer 有一个好处：如果程序是一长串的浮点指令，而浮点部件只有 4 个，rename 每拍发出 8 个，dispatch buffer 至少能稍微缓存一下，而不是一开始就迫使 rename 按 4 IPC 工作。

调度队列的演进：

2013 年就有了 2 级调度的专利。

2015 年的专利在说如何优化“物理队列”，比如从队列中间弹出指令的话代价有点高，一般的优化策略就是：尽量少合并直到不得不做；用链表而不是数组（注：物理上队列要怎么实现？）等等。APPLE 的算法是使用 AgeMatrix，用来追踪指令的临时序列，这样就不用挪来挪去了。例如，调度队列有 30 个表项，那么用一个 30x30 的矩阵，如果一个 bit 设置为 1，就意味着这个队列表项更老。AgeMatrix 的翻转的频率非常低。我们后面看到 LSQ 也有类似的处理方式。（注：Age Matrix 用来表示的是每个 slot 相对于其他 slot 的顺序）

2016 的专利是 earlier testing of relative ages，也就是解决 Age Matrix 怎么用的问题。每一轮比 2 条 ready 指令的 age，出一条指令（最老的那条），迭代比较。如果 30 个 entry 的话，迭代 5 次即可。还可以在 matrix 中加入 criticality 信息，这样就可以获得一个基于 criticality 的调度器。比较是越早越好，越早比较越能满足周期要求。

2017 的专利是 pairing scheduling queues and issuing from either。Intel 单一大队列的问题（再次重申）：看起来能够在一个队列里获得所有信息；但一个是功耗明显上升，再一个是优化问题（最老的 ready 指令可能不是最优先的，因为 criticality 的存在；而单一大队列会倾向于优先调度最老的 ready 指令）（注：这里评价有失偏颇，因为多个小队列也会遇到相同的问题）。APPLE 的这种多个小队列的方式在功耗方面有明显优势，虽然会遇到负载均衡问题，但每年都有小的算法调整，效果明显且功耗几乎不增。APPLE 的 2015、2016 专利的方法也会获得一个近似的最早就绪指令优先的调度效果。负载均衡的一个场景是：A 有 2 条指令就绪，B 没有指令就绪，这就出现了 A 忙 B 闲的状态，不能发挥两个队列的优势；但是 APPLE 的专利中说，find the better instruction 的动作可以做很多轮，每一轮都是比较 2 条指令、出 1 条指令。那么在最后一轮的时候，我们可以把 A 的第二候选交给 B，这样 A 有 1 个就绪，B 也有一个就绪，这就是 pair。这就能在均衡方面稍微好一点。

2017 还有个专利：Hierarchical reservation station，将调度队列分成了两半：前半快、后半慢的结构。这个结构可能是用在 FP 上的。这个专利要解决的问题是：指令 age 的比较需要时间，2016 的专利是一种解决思路。这篇专利是另一个思路。核心思想是切分调度队列为两部分，两部分同时跑调度机制：上半部分的结果是最老的就绪指令；下半部分是：可以执行的指令队列，一旦上半部分有指令发射了，这个指令队列就移到上半部分去。但这个专利

和前面的专利不相容。有可能只是在 FP 上用的。(再次再次重申）队列大点有助于发现更多的并行性，否则这些并行性都会被隐藏在 Rename 阶段或者 Dispatch 阶段。但队列是耗电的，越长越耗电，太大也不行。所以，最优的队列长度是不一样的，取决于①平均有多少条指令依赖前序指令；②这些指令的平均执行延迟。看起来 FP/SIM 的依赖链也长、指令延迟也长。所以 FP/SIMD 的调度队列有 18 个表项，可能分了两半。(注：这一段目前为止还是一个合理的推测；还要注意图中画的是 36 表项，实际上是做成了 2x2 的映射关系，实际长度是 18 表项；这个数字比整型的要高；这意味着由于 FP 指令慢、且软件代码特点是存在较多的依赖关系，所以需要尽可能多一些的指令来找非相关指令。分成两半的话执行效率会略高一些，而且分两半做比一起做要省一点功耗。)

追踪相关性：

基于寄存器的相关性：这个太 old 了，1960 的 tomasulo 算法就可以引申出这个；1987 的 Instruction issue logic for high performance interruptable pipelined processors 就写了这些，而且很关注精确机制。2000 的文章 the design space of register renaming techniques 也介绍了很多思路。这些文章介绍了①要解决的问题；②术语和观念模式。但是现在已经是 2020 年了，应当有更好的方法。

Tomasulo 算法会存在的问题：寄存器分配必须非常早，在 Rename 阶段就得做（这还是 in-order 阶段），在 Retire 阶段才能释放。而寄存器资源非常有限非常贵，这就是浪费。典型的优化思路就是晚分配、早回收。如果想做到在指令执行阶段（完成时）才分配，跟踪相关性就需要有不同的做法。(注: 原作者非常推崇 Virtual Register，在全文中出现的频率非常高)。早回收要去看之前提到的 2011 Nehalem 文章。寄存器的晚分配早回收目前还没有商业处理器完全实现。但 Power 在 LSQ 上实现了晚分配；APPLE 在 LSQ 上很有可能已经实现了早回收。

基于指令的相关性：ADD x0, x1, x2，表面上依赖 x1 和 x2，但 x1 和 x2 是由另外两条指令产生的，所以用相关指令来追相关性也是可以的。APPLE 2016 专利描述了这一个思路。这个思路是可实现，因为要追的指令并不是那么多，而只是 ROB 中的一个子集，相关指令可能最多不超过 400 条。

基于 bitvector 的相关性: 每条指令关联一个 bitvector 用来记录相关性, bitvector 有 400bit，其中只有若干 bit 是 1（稀疏)。APPLE 2014 专利 Latch circuit with dual-ended write 比较清楚的解释了这个思路。

那么为什么要用基于指令相关性，而不是基于物理寄存器的相关性？解释如下。

例如: load 依赖前一个 store 的情况怎么处理？用 LSDP: Load Store Dependency Predictor，但这个 LSDP 是关联在 Mapper 上的，而不是 LSU 上。APPLE 2012 专利 Load-store dependency predictor content management.描述了这个思路。这要求增加一个 Load 的相关性记录。由于 load 可能会暂时失败（TLB miss，L1D miss)，那么就需要 replay 机制隔几个周期再 load。为了减少 replay，只需要在 bitvector 上再加一个依赖关系：在等什么。(注：这里继续解释下 replay 是什么。在执行单元完成 load 的一系列前序操作后，真实的 load 操作被丢到 load queue 里去执行；如果 load L1D 命中，那么基本上当拍返回，这条 load 指令就可以清掉了；如果 load miss 了，那么这条指令就不能清掉，它会阻塞执行单元；不想阻塞执行单元的话就得用非阻塞的方式实现，给若干次尝试机会，直到取回来数据。因为 L1D 又是独立的系统，它可能会隔很多拍才完成 cache 的更新，那么 replay 一般的实现形式就是隔几拍重试；APPLE 的做法是给 load 指令增加一个依赖关系，这样在调度的时候就可以确定这条指令是不是可以清掉了。)

上下文切换时也需要关注依赖关系，这个 bitvector 也可以做。

预测机制也可以用 bitvector。

所以，bitvector 是一个普适的机制，几乎所有存在相关性的特征都可以刻画：可以给任意指令增加新的相关性、可以定义新的相关性。
这比单纯的寄存器依赖能做的事情多得多。
整合在一起，实现调度器：
三张表。假设调度队列有 30 个表项，每个表项有 3 个域组成，这样就有了 3 张表。
第一张表存放①操作；②目的寄存器的物理 ID；③源操作数的物理 ID。这张表记录要执行这条指令所需的一切，但不涉及如何调度、何时可以运行。
第二张表记录相关信息（bitvector）。大约 30 行 400 列。稀疏矩阵。例如，第 4 行第 7 列为 1 代表：在调度队列 Slot4 中的指令还没有准备好去执行，它依赖一些物理寄存器，写这个物理寄存器的指令是 SCH# 7。
第三张表将前两张表关联起来，它记录每条指令的 SCH#和 relative age。
用一个并行的总线，大约 400bit 宽。每当一条指令发射时，所有的执行单元每个单元设置位，依赖 SCH#。这样这条 400bit 宽的总线大约只有 10 个左右的 hot（1）码，这能够表示所有能很快得到结果的指令。而且不用关心产生几个结果，所要关心的是旧指令是否在依赖它的新指令之前执行完。
这 30x400 的 bitvector 可以被所有调度队列抓取，然后可以流动。每一行中的 1 就是这条指令依赖的 SCH#，如果都清 0 了，那也就可以发射了。
还有很多细节呢，比如如何增加相关性。
执行时间超过一个周期的指令怎么处理？只要执行单元在快做完的时候更新 bitvector 即可。
这个想法（三张表）也叫矩阵调度。这应该就是这篇专利的实质。但是这篇专利读起来很困难，claim everything==reveal nothing，基本上只是在说很多事情可以做，而不是说如何做。
基于 Criticality 的调度：
目前还没人实现，未来可能有。
最老的可执行指令不一定是最优的。2018 的论文 Principles of instruction level distributed processing 比较好地说了这个概念。
Split scheduling queue：
2008 的专利，这个思路已被放弃。主要想实现：背靠背地调度相关指令；做法是把“立即相关指令”和其他指令分开，把立即相关指令丢到一个非常小的池子里，这样调度会容易一些。当然了，更好的方法是现在用的投机调度。所以这个思路就被抛弃了。

4.3 实验设计

Integer Dispatch Buffer 的解释：主要复述和强调 4.1 节的内容。
FP dispatch buffer：
实验：370ADD + N FABS；图形差不多，斜率 6+平+斜率 4
Interpretation as smoothing mechanism
来点不一样的：6ADD+2NOP，循环 N 次，刚好 8 宽，每一排 6 个进整型执行单元，2 个在 Rename 阶段完成，完美平衡。
然而：12ADD+4NOP 呢？第一拍肯定不平衡，但第二拍还是平衡的。
K*6 ADD + K*2 NOP…K 增加到一定程度，超过了 Dispatch Buffer 的容量，结果是 IPC 会下降。
简单点说，dispatch buffer 能够平滑不同类型的代码序列，能够让连续的 ADD 仍然 8 宽运行，只要这个连续的长度不是太长。M1 测出来的应该是在 6*13 到 6*15 之间。
再解释下：dispatch buffer 能够让大部分代码运行在 8 宽，能够在很大程度上处理连续的密

集指令（连续的使用同一类部件的指令，例如 add，如果只有 add 的话，限制应该是 6；如果只有 fadd 的话，限制应该是 4），只要这些连续指令的数量没有那么长；对于 add 如果长度不超过 6*13，后面有其他类型的指令（浮点、nop、load/store），那么这个指令序列还是能按照 8 宽来跑。这是个很有意思的数据（整型 dispatch buffer 可是只有 24 表项…，(6ADD+2NOP)*13 和(6*13ADD+2*13NOP）跑出来的时间是差不多的，这就是平滑）

Acceptance width of fp buffer

K*4 FABS + K * 4 NOP，K=5 时还能够达到 IPC 8；K=6 就不行了，大约测得 Dispatch buffer 12。

为什么说整型的 dispatch buffer 是 12*2，而不是 24*1:

K*（3CMP + 5NOP），每个 buffer 都可以接收 8 IPC。

FP dispatch buffer 有没有分开？

FCMP+7NOP；结果可能是 12。

Load/Store dispatch buffer 测试

3LD+5NOP；结果可能是 10。

（注：详细的实验过程请关注原文。本文只尽力表述精简的内容）

5 立即数的处理

根据之前的实验，M1 大概是 380 个整型物理寄存器，624 个 HF（注：还是需要声明一下，实验得出的结论在有些时候是一个数量级上的结果，具体的数值在全文前后一直存在一些偏差。我们更应当关注的是数字代表的含义，而不是数字本身）。如果只做整型运算，限制是 HF 而不是整型寄存器数量。

Zeroing

Movz X0, #0 = MOV X0, #0; 8 宽

Movi x0, #0 = ORR; 6 宽

Movn x0, #0 不支持

Mov x0, xzr 6 宽

EOR x0, x0, x0 1 宽!

非 0 立即数

Mov x0,#1

Mov x0,#-1

Mov x0,#1234

Mov x0,#12340000

这些都 8 宽

立即数的相关性

Mov x0, #0; mov x1, x0; 仍然能获得 8 宽。

Mov x0, #0; mov x1, x0; mov x2, x0; …; mov x7, x0; 仍然能获得 8 宽。

Mov x0, #0; mov x1, x0; mov x2, x1; mov x3, x2; … mov x0, x8; 仍然能获得 8 宽！（注：真很不容易。但是哪个编译器会产生这种代码？）

Mov x0, #1 + movk x0, #0x1234, lsl 16; 这能创建一个 32bit 的整数

这个指令序列大约 4 宽。说明 mov 和 movk 没有融合。

浮点立即数的支持好像没什么特别的。

实现：

2012 方法：假如有物理寄存器 192 个，那么剩下的 192~255 的编号就可以用来硬编码一些常用的立即数。引申过来就是，可能用 22bit 编码物理寄存器，用前 1 个或 2 个 bit 来拿标识是物理寄存器还是立即数。这样在 rename 阶段就可以完成立即数赋值。
现代方法：分离的寄存器池（separate register pool），设置了一组特殊的寄存器文件，仅供译码阶段访问。M1 的实现大约是 38 个寄存器。对于 mov xn, #123 这样的指令，在译码阶段先将 123 写入一个特殊寄存器，再 rename 的时候分给 xn。
实验：
1，mov #0；大约 8 宽
2，mov #1；也能做到 8 宽，但跳变点更早；
Add xn, xm; add xn, #1；两个指令都是大约受限于 380 个物理寄存器；
Mov xn, xm；不用物理寄存器；它受限于 620 个 history file；
Mov x0, #0；也不用物理寄存器，rename 阶段完成；可能和 history file 有关，但明显跳变点来的更早；
Mov x0, #1；跳变点大约 418。看起来大约是 380+38。38 个左右额外的寄存器。
那么“立即数”寄存器的组织是什么样的？只有 1 个？一组 40 个？CAM 分组 8 个？实验表明应该是大约 38 个独立的寄存器，每个寄存器可以有一个值。
如果这些“立即数”寄存器都用完了怎么办？实验表明，这些“立即数”寄存器就是只能在 Rename 阶段可以被写，用完之前，mov #可以跑到 8 宽，用完之后就只有 6 宽了（动用整型单元了）。
关于立即数的结论如下：
1）有一个硬编码的 0 值物理寄存器，这样在 rename 阶段就可以处理 mov #0；
2）有一组大约 36 到 40 个额外的寄存器用来存放其他立即数，在 rename 阶段写入值和分派；写入带宽大约是 2 个立即数/周期。
3）如果用尽了这些额外的寄存器、要写入的数量超过 2，那么 mov #就要进入调度和执行级了。

6 Register File 的实现细节

寄存器分配涉及到的点有：1）寄存器池；2）表明寄存器是否在使用的状态位；3）分配算法的性能（8/cycle）。

6.1 基本概念

Register Pool
可以有很多实现细节。比如 APPLE 第一代 64bit 核，其整型寄存器文件是由 40%的 32 位寄存器和 60%的 64 位寄存器组成的（能节约一部分面积的实现、分配时也会根据指令类型优先分配，然而现在都不这么做了）
Free Register List
可分配的寄存器信息如何获得？ROB 肯定存有这个信息。除了 ROB 之外还需要有更快的方法。比如“Free register list”，就是一个队列或者链表结构，很容易做宽。
APPLE 的优化有三次：
1）bitvector：第一次优化，有多少个物理寄存器就设置多少个 bit，组成 bitvector，这样省

点空间，但每次用的时候需要遍历整个 bitvector。优化的点在于：a）用多个 bitvector，比如 3 个；然后扫描器每个 bitvector 配 2 个，一个从前到后，一个从后到前，这样比较容易做到 6。b）long-term free vs short-term free register, ROB 刚释放的当 short-term，被“立即”回收利用，这样可以避免 2 次 bitvector 的翻转（非常细节，能略微降低些功耗）
2）multiple bitvector banks：第二次进化。将寄存器文件分成 4 个 bank，在需要很少寄存器的情况下就关掉不用的 bank；这样能够省电。具体实现包括两部分：a）使用“short-term free”寄存器队列，基于理论：刚用过的 register bank 应该是上过电的；没上电的 bank power on 的话还是需要不少时间的（上百个周期）；b）bitvector 分成 4 个，配 2 个 scanner，就和前面的想法一样，但是 bitvector 的顺序重新设计一下，这样可以让 8 个 scanner 都能找到位于同一个 bank 的寄存器。Bitvector 的排布顺序是重点！
3）在可能的地方省略寄存器分配（2017 专利，未实现）：第三次进化。算是资源放大的方法。①寄存器迟分配/虚拟寄存器；②寄存器早释放；③不分配！不分配的意思是直接从 bypass bus 上读取寄存器值，这在逻辑寄存器值立即被覆盖的情况下是可行的。应该还没实现。
寄存器文件的功耗问题
（略）

6.2 上下文切换（减小开销的方法）

Dirty flag：其实很多 FP/SIMD 寄存器都用了这个方法，有的进程没有使用 FP/SIMD 寄存器，那么就不需要保存和恢复。
基于硬件的寄存器保存和恢复：大体上应该是这样的：①OS 查询硬件找到一块 TCB；②TCB 的编号就是 threadID 或者其他和线程绑定的寄存器；③上下文切换应该要包括 4 个步骤：找到旧 threadID 及其 TCB；找到新 threadID 及其 TCB；换出旧寄存器；换入新寄存器。有硬件的支持，就不需要软件去折腾 dirty flag 之类的东西了。优化动作可以完全交给硬件实现。
APPLE 的方法：2015 专利（可能未实现）
Basic idea：①用某种途径（例如一个特殊寄存器）指明当前线程号、指明当前发生了上下文切换；②理想情况下硬件完成旧寄存器保存、新寄存器装载；③d 按照 on demand 的方式，减少保存和装载的寄存器数量。
APPLE 专利的点是：在 mapping table 中增加一个新域（mapping table 保存逻辑寄存器到物理寄存器的映射关系），形如：x0 is mapped to physical register p3 with a tag of threadID. 这样，在发生上下文切换时，第一条读 x0 的指令就会访问 mapping table，然后发现 tag 不一致，然后就会：①把 p3 的值写出到 x0/threadID 对应的 TCB 中；②用新 threadID 的 TCB 中的 x0 值填回 x0（新映射的物理寄存器，假如是 p5）并且更新 mapping table 中的 threadID。这种方式能够做到使用中的寄存器才会 save/load，可以大幅度减少开销。
基于此还有还有优化空间。
优化 1：两个寄存器配对共享一个 tag。因为 save/load 两个寄存器的开销和 1 个寄存器的开销差不多。
优化 2：store 可以直接发到 L2，跳过 L1。（注：这条不太理解具体要怎么实现，如何保证一致性？而且这么做需要把优化 1 做的更大一些，比如得按照 cacheline 来保存，文中并未找到更详细的描述）
优化 3：threadID tag 在物理地址空间，不是虚地址空间，这样可以少一次 TLB 查找。
优化 4：再增加一个物理 flag，表示这个寄存器有没有被修改（也就是 dirty flag），这样未修改的寄存器就不用保存了。

按照以上做法，CPU 不用停，只在用到 x0 这些修改过的、需要保存和恢复的寄存器的时候需要多一些延迟。这由 CPU 自动创建一些合成指令完成具体动作。
如果给中断设置一个“threadID”，那么中断处理也可以用这种方式。
当然，发生迁移（线程迁移到另一个核）时这个方法有点问题，APPLE 提供专门的指令去强行刷所有修改后的寄存器至 TCB。
各种 FLAG 寄存器也可以这么操作的。
尚不能确定这个专利是否实现，但原作者认为在现有的基础上（register security tag）实现这个专利是可行的。
APPLE 的方法：2019 专利 reduced context
基本概念是：对一些特殊进程，定义一个简化版本的上下文。搞一个标志位表示我们是否用了这个机制，如果用了之外的寄存器，那就触发一个异常。
CHINOOK
除了大小核之外，APPLE 还做了一个极小核，用来控制 GPU、NPU、 ISP 等等等。这个核的代号可能是 CHINOOK，原作者是根据 A12 的相关信息推测出来的，现在也就这么叫了。Firestorm 参数化裁剪出 Icestorm，Icestorm 大概有 80 个整型物理寄存器和大约 88 个 SIMD 寄存器。CHINOOK 应该更小。

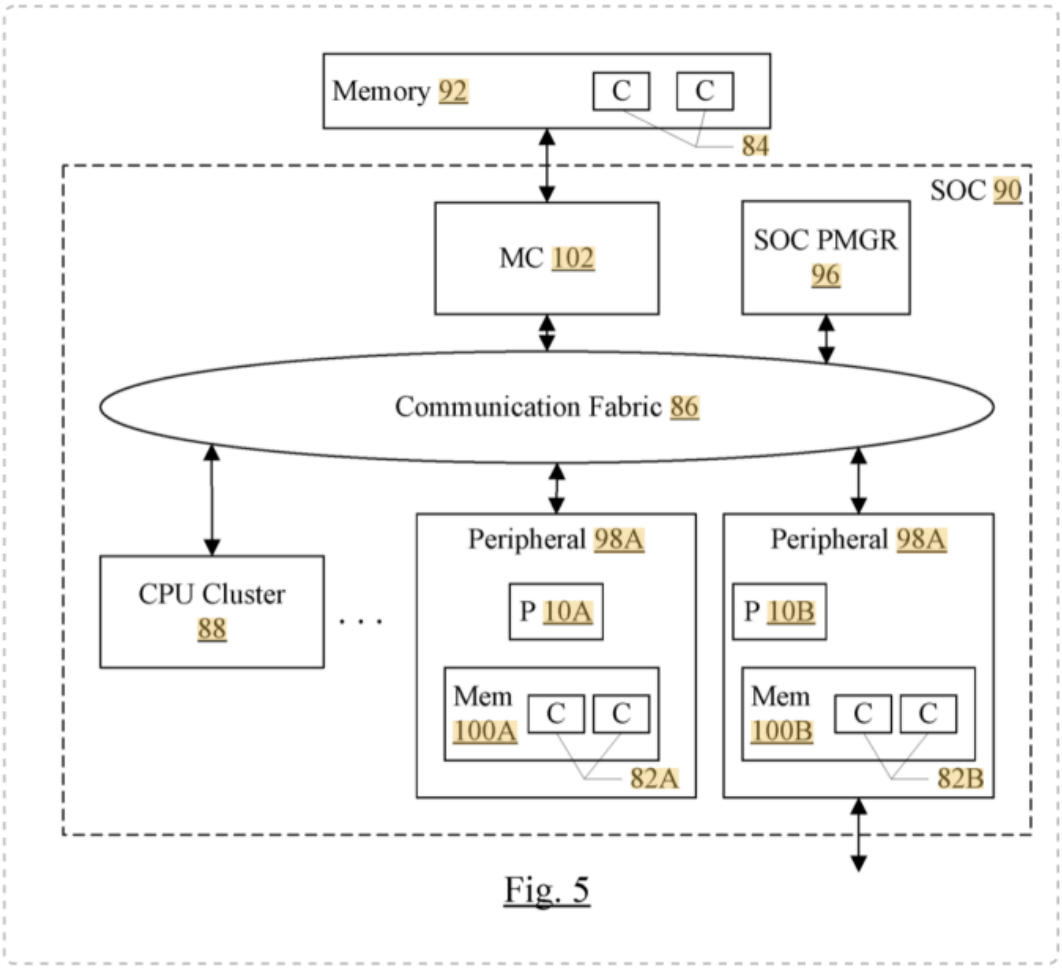


Fig. 5

某个专利里有这张图和配字。配字中提到 P10A P10B 比 88 中的 CPU 的上下文切换数量要多的多（至少一个量级），Mem100A Mem100B 也比 Memory92 要小的多，因此 P10A P10B 就得是个 tiny CPU，其寄存器数量也少，上下文也很小。98A 98B 可能是硬件功能，比如 image signal processor 等。

原作者认为这就说明了 CHINOOK 的存在，且其广泛用于 SOC 的控制器，应该就是裁剪出来的，ABI 也做了裁剪等等。原作者认为这个小小核也包含了 TLB PAC，能用工具链，能跑 OS，性能也是足够的。肯定比 M4 强。((CHINOOK delivers) enough performance that engineers can spend their time worrying about how to make the entire device better, rather than worrying about how to make an ARM M4 do whatever needs to be done.)
（注：这一段术语推测的内容，没有足够的材料和实验支撑，实验中触及不到 CHINOOK 的存在。）

6.3 Register bypassing and early release

（注：这一小段在讨论专利中提及的内容，目前尚未有证据实现，但原作者认为以后还是可能会实现的）
讨论下晚分配和早回收的问题。APPLE 2017 的专利。
例如指令序列： REV x0, x0; CLZ x0, x0; ADD x0, x0, x1.
像这样的指令序列，CLZ 没必要分配 x0，直接从 REV bypass 过来就好；ADD 的 x0 是目的寄存器，因为是覆盖操作，所以 CLZ 的 x0 分配了也就可以立即回收。所以这就是两个不同的策略，都能够做到更少的物理寄存器。一个策略避免分配（用一个 tag），一个策略提前回收。
然而两个策略在 M1 都没用。只在 load/store 上做了类似的策略。
主要原因出在指令调度。类似 ADD+shift 指令对。这种指令对是：①两条指令；②第二条指令从前一条指令的中间结果直接 bypass 都过来。这样的话就要求这两条指令必须要同时调度，不然的话第二条指令很难读到 bypass 的东西。(注：如果融合指令做的足够好的话，好像能解决这个问题。)
Virtual register 的失败处理也很麻烦，例如在某一个点发现没有空闲的物理寄存器了，那么就需要有硬件机制来来解决（注：可能的操作是：①停流水；②等旧指令退休、空出来空闲的物理寄存器；或者③再增加一级存储？原作者并没有给出详细的解释，建议找相关论文扩展阅读。）
相应的，早回收也有相关专利，但没有实现。

6.4 ROB 退休能做到多快

我们知道寄存器是在退休的时候释放的，那么退休时能释放的多快？速度的下限至少是 8，上限呢？
NOP 的退休：56/cycle
测试结构：33 个 FSQRT 组成的延迟块 || 1800NOP 组成的延迟块 || 370 个 ADD+可变数量的 ADD。
33 个 FSQRT 的目的是把 ROB 和调度队列填满；1800 个 NOP 是为了在 FSQRT 之后把 ROB 填满；370 个 ADD 是为了把物理寄存器用尽；可变数量的 ADD 是直到物理寄存器可用的时候才能够执行。
结果能够看出，NOP 的退休速度上限是 56/cycle。
ROB 的结构
ROB 应该被看成大约 330 行，每行 7 指令。大部分指令可以放到一行中的任意 slot，有可能会失败的指令只能放在一行的最后一个 slot，这里“有可能会失败”（failable）指的是有可能

会导致 CPU flush/restart 的指令，例如分支、load/store 等。
按照这个结构，上面的结果应该看成：ROB 每周期可以清 8 行。如果是连续 load 指令，那么退休速度是 8 指令/cycle。如果是连续 nop 指令，那么退休速度是 56/cycle。(8~56/cycle 退休速度)
释放 register 和 HF（16/cycle）
实验把 nop 序列换成 fabs 序列。Nop 不占 HF，fabs 会吃 HF。实验结果是大约 16/cycle。
只释放 HF，不涉及寄存器（16/cycle）
FABS 替换成 FMOV，结果是一样的。
（注：CPU 的行为受到多重资源约束，因此实验代码的设计和结果分析必须谨慎，使其尽量符合事实。）

6.5 So how close is the M1 to a KIP

KIP 大概是 2000 年以来的 dream，意思是：处理器能够同时维护“in progress”的 1000 条指令。当时大概是想做到，DRAM load miss 的时候，CPU 不停顿（在当时的条件下）。
现代 CPU 速度更快，KIP 不够了。例如 load miss 的延迟是 330cycle，跑 8 宽，那么可能需要的是 330*8=2700 条指令。
显然，如果全是 nop 指令的话，应该早就超过 1000 了。那么对于真实指令呢？限制在哪里？
ROB 的大小肯定不是问题，太好加了。
LSQ、寄存器堆、HF 不是那么好增加，主要还是要通过不同的算法优化来放大之。
例如这样的指令序列 （ADD x0, x5, x5; FABS d0, d0; str x5, [x2]）大约能做到 310 次迭代，也就是大概 930 条指令。(注：再具体的序列原文没有继续测)。
目前结论看，M1 大约能做到 KIP 的量级。现代处理器的 KIP 指标应该是不够用了，但仍然可以看作是一个很好的里程碑。

7 Load and Stores

7.1 基础测试：LSQ 的大小

FSQRT+load
实验看起来大约是 LSQ 能保存 328 个 load。这个结果看起来是有问题的，①328 这个数字接近 ROB 的行数；②之后显示 load 是 4/cycle，而不是 3/cycle。
FSQRT+store
看起来仍然是 330 左右跳变，4/cycle，而不是 2/cycle
而 M1 只有 2 个 load 单元 1 个 store 单元 1 个混合单元，也就是上限 3load/cycle，2store/cycle。
那么用 load miss to dram 做延迟块，替代 fsqrt。
实验结果大概能看到 load 队列大约 125，store 队列大约 100，斜率大约能对的上 3load 和 2store。但前面仍然有一小段 4/cycle。这个结果不太好解释，需要继续分析。

7.2 LSQ 的理论

LSQ 是一个 Open 的话题，很大很复杂，可能比 CPU 其余的部分还大。

为什么需要 store queue

原因之一是投机 CPU，写操作在分支确认之前不能真的写到 cache 中，这不仅影响本核，在多核情况下还会影响到其他核心的 snoop。所以要把 store 操作先暂停住，直到其明确不是投机状态才真正写入。

原因之二是 OOO 的问题，我们需要保证指令的顺序和语义的正确性。

因此，我们就需要一个队列（真的队列，有顺序的），这个队列有如下特点：①每个 slot 大概要包含：写入地址、写入数据、状态位；②每个 slot 应该在 rename 阶段分配（此时指令仍然是顺序的）。

总的简要流程是：①rename 阶段分配 store queue slot；②store 指令要待在 issue queue 中直到 store 地址和 store 数据依赖都解决了；③store 指令丢到 store queue 中；④store 地址应当被检查权限，需要通过 TLB 和 pagewalk 等步骤检查所属页面属性；⑤ROB 退休时确定这条指令可以退休了，SQ 中的这条指令要打个标记；⑥之后这个 slot 可以真正写出，然后释放。

为什么需要 load queue

考虑这种情况：load/store 依赖。一般我们都说 LSQ（load store queue）。一个 LSQ 一般行为是：①slot 可以存 load 也可以存 store；②rename 阶段分配 slot；③load 执行的时候，从后向前扫描 LSQ，找到最近的、写地址等于读地址的 store，然后读数据；④store 执行的时候，从前向后扫描 LSQ，找到所有读地址等于写地址的 load，喂数据给 load。

LSQ 需要考虑很多相关性，还是挺复杂的。在涉及到 OOO 的时候更麻烦。比如这种情况：Load 的时候扫描 slot，找到一个 store slot，但它的地址还没算出来，怎么办？在这个模型下就只能等。（这是 basic design）

Speeding up the basic design

第一步：把 store 地址和 store 数据分开。因为 store 地址可能很早就算出来了。这样 load 就不用等 store 指令找到所有该有的数据之后才能跑。几乎现在的 CPU 都这么做，只是具体实现不同。例如 x86 就把 store 拆成 2 个操作，一个算地址，一个算数据，然后分别调度。

第二步：预测 load/store 依赖。做一个 Load store dependence predictor 很简单，准确率也很高，比分支预测简单多了。

第三步：load queue 和 store queue 分开。

7.3 改进 LSQ 的思路

思路 1：把 SQ 当作 L0 data cache。这个思路主要为了节能。思路是这样的：①每次 load 的时候都要花代价访问 SQ；②如果 load 命中 LSQ 的时候就不需要再去读 L1 cache 了（可以直接从 SQ 中拿数据）；③那么不如我们尽量往 SQ 里多放，即使退休了，我们也要尽可能长久地保存 SQ slot，把 SQ 撑大。这样 load 的命中率就高，就能减少访问 L1 的次数。2019 的论文讨论了这个想法，说在 skylake 这个量级的 cpu 上能获得 50%的命中率，大约能节能 15%（load/store/L1D）。

思路 2：两级 LSQ。之前有篇文章讨论过。但毕竟 LSQ 还是挺贵的，也很难优化。这个思路局限性比较大，也不太好实现。

思路 3：Virtual LSQ。IBM 的 Power7 是这么做的。思路是这样的：①age tag 在 rename 阶

段分配；②真实的 load/store slot 直到离开调度队列时才分配。这样可以做到晚分配。
与 virtual register 思路类似，如果突然用完了队列，需要有机制来处理多余的 age tag。一般就让 load/store 指令堆在 load store scheduler 上，而不是 rename 阶段，这样后续的指令也还是能继续跑的。APPLE 应该采用了 Virtual LSQ（注：后面的实验支撑这个猜测）。
至于早释放，其实也早不了太多，必须满足几个条件。（略）
（注：gcc -O0 经常会产生这样的代码： store x5, [x2]; load x6, [x2];
如果 LSQ 做的比较好的话，在 store 可以生效的时候，load 实际上可以立即拿到数据。
对于这样的序列：store x5, [x2]; load x6, [x2]; add x6, x7; store x6, [x2]. LSQ 如果做的足够好的话是不是可能合并成只写一次？如此一来，即使是-O0 的代码，执行效率也能得到较高的优化。）

7.4 Load Store dependency 的实现

基本方法：2009
load referenced by instruction count relative to store
实现方式：①当 store 过了 Mapping 后，其地址与预测器中的 store 地址比较；如果匹配，读出预测器中对应的计数值；②计指令数，如果遇到 load <count>指令，那么这个 load 就要小心处理；③我们找到早先的 store 的 SCH#，现在我们把这个 SCH#加到 load 的依赖中。用 bitvector 就可以记录这些，这样就把这条 load 的依赖项加入了前面的这条 store。
以上是正常情况。也就是 LSDP（load store dependency predictor）的基本工作。LSDP 用 CAM 实现，可以记录多个 entry，这些 entry 的 store 地址相同，但 count 值独立。这样就可以处理：多个 load 依赖同一个 store。（ store x5, [x2]; load x6, [x2]; load x7, [x2]）
当然，LSDP 更可以做到不同的 entry 中存不同的地址和不同的 count 值，这样可以处理：一个 load 依赖多个 store。Bitvector 可以很容易地记录依赖关系。
预测器做到的是："这个 load 可能依赖最近的 store"，这种预测总比没有好。Bitvector 则可以精确地加上相关信息。
这个方法不足的地方之一是：更新表的时机。把 CAM 当作 FIFO 用了，完全按时序替换，这样的利用率就很难提升。
不足之处之二是：CAM 中存的是 PC 的 HASH 值。所以有一定的概率 hash 冲突，可能会误判（虚警），导致一个 load 错误地被延迟。这个发生概率不是很大，但是会发生。
（注：之所以提及这个虚警问题，是因为后面还要说分支预测。）
2012： load and store both referenced by hash of PC and instruction registers
Hash 的计算方法来自 PC 和用到的寄存器。之前的方法 hash 只来源于 PC。
完整的过程是：①对于 store，当其流过 Mapping 的时候构建 hash；探测 LSDP CAM，对所有匹配的表项设置"armed bit"，当这个 store 执行后 armed bit 会清掉。②对于 load，从 CAM 中找 hash，如果匹配上且 armed bit 为 1，那就把这条 load 记录其依赖项为这个 store。
改进点还有：例如额外增加一个信任域，当信任域为低时，CAM 会被占用，但不参与比较。
（注：后面还有很多细节描述，本文都略过了。）
2016：LSDP optimized for replay
这个专利明确说明了 store 的地址和数据是分开处理的。
考虑这个场景：store x5, [x2]; …; load x6, [x2]; 并且 LSDP 还没训练这个对（也就是还不知道依赖关系），会怎么样？有三个情形：①如果 load 来的太晚，cpu 并不需要将它们 link 到一起，各做各的就好；②load 来的不是那么的晚，已经能看到 store 的地址了，但 store 的数据还没有好，那么 load 就必须得 replay；③load 来的太早，store 的地址还没算出来呢，那

么 load 就会先行一步去读 cache，后面再 flush。所以，情形 1 就是啥都不做；情形 3 就可以在 LSDP 中创建关联；情形 2 就是这个专利考虑的。
（注：再一次解释 replay。Load 已经被丢出去了，然而由于数据没有准备好，load 拿不到数据，就不能结束。后面就得各种重试。直到拿到数据。Replay 就是这个重试的意思。）
那么，如果（相关指令）足够近的话，何必那么早丢出 load？但是，又不可以无限制的卡住 load 不发（明显没有相关性的 store 不应当卡住 load）。
处理方法是：LSDP 的 entry 中增加一个 bit，表明是 replay 还是 flush。Entry 标记为 flush 的时候，delay load；如果为 replay 的话，相关性根据信任度添加。
处理非对齐/overlapping 的 load 和 store
例如写一个字节读一个双字，或者读一个非对齐的双字要跨两个对齐的双字之类的情况。
解决方法的一条途径是：LSQ entry 的颗粒度放大至 cacheline，用 bitmask 表示 cacheline 中的偏移。这个方法简单易行。对于跨 cacheline 的 load/store，将其拆成 2 个 entry。

7.5 APPLE's implementation of replay

Replays are situations where an instruction could not complete because some detail wasn't ready in time. 常见的情况如：load 的地址已经算出来了，就开始执行了，但这个地址在 TLB 中 miss；或者在 TLB 中但 L1 cache miss；或者它匹配了 store queue 中的一个地址、但 store 指令由于数据没有准备好不能写。这些情形都是没准备好，但很快会好，所以一般处理方式就是过几个周期再重试一次。
扩展一下：投机执行的 CPU 中的取指操作也存在类似问题。
Replay recovery
Replay 机制需要保证数据的可用性。例如要处理这样的情况：load x5, [x2]; add x6, x5；如果 load 触发了 replay，但 add 已经调度到整型单元了，这时候 x5 可能是个非预期的值，需要恢复。
APPLE 的做法是增加一个小定时器，在 add 开始执行的时候（但是不要立即从调度队列中移除）；如果 load 是正常的，这个定时器会被取消；如果 load 不正常，这个 add 会被丢掉，然后调度队列继续调度其他指令，直至 load replay 了之后才会重新调度这个 add。
2006 extra dependency bits
增加相关位，使得指令在准备好之前不发出去，这就避免了重试。但可能会带来性能上的损耗。
2016 move replays into the load queue
2019 split load queue into a replay optimized queue and an address validation queue
LEQ + LRQ：
LEQ：as fast as possible；这个队列的指令可以在必要时被 replay
LRQ：大队列，保证正确行为。
2019 convert flushes to replays

7.6 继续测试 LSQ

7.6.1 队列深度

来我们继续实验。前面我们实验得出结论：100 store queue 和 125 load queue。
换一个序列：FSQRTS + FCVTAS x0, d1 + str x5, [x3, x0] + load * N
实验得到 N = 188
再换一个序列：FSQRTS + FCVTAS x0, d1 + str x5, [x3, x0] + store * N
实验得到 N = 118
所以 3 组数据：330，125L100S，188L118S，怎么解释？
APPLE 用了 virtual load store queue。当指令在 rename 阶段时，load/store 指令得到一个顺序的 ID，但没有分配 slot；在 issue 时，分配 slot。（如果是 load，LRQ 和 LEQ 都分配）
330 的数据其实说明了：①早释放；②ROB 的数量大约是 330 行，每行只有一个 load。
125L 的数据其实说明了 load queue 的物理尺寸。
188L 的数据其实说明了：①晚分配；②scheduling queue 深度 48，load/store dispatch buffer 深度 10，48+10+125~188 （注：不要纠结于数据的精度…）；
这样，188L 实际上是把物理 load queue 增大了大约 50%。
然后我们有一段大概能做到 4load/cycle 的怎么解释？暂时跳过。
100S 的数据是怎么回事？因为测试 case 的问题，load queue 和 store queue 是分开的，load 做成的延迟不会影响 store queue。所以不会停顿住。所以我们得到的其实是：48+60~108（注：不要纠结于具体数据精度），那个 100S 应该是 108S。这个测试结果还是说明了晚分配，scheduling queue 深度 48，物理 STQ 60，加起来 108.
118S 也是说明晚分配，48+10+60 = 108.
（注：后面细节略过。但是对于早释放，本节实验上测出来的结果似乎并不能直接证明其释放逻辑。）

7.6.2 LSDP 测试

<u>不同地址：</u>
校准：pair = str x10, [x2]; ldr x11, [x3]; (x2 != x3)
校准结论：800 pair /400 cycle，aka 4 memop / cycle.
测试 1：pair1 = str x10, [x2]; ldr x10, [x3] (x2 != x3)
测试 1 结论：仍然是 2pair/cycle， 4memop /cycle
<u>相同地址：</u>
测试 2：pair2 = str x10, [x2]; ldr x11, [x2]
测试 2 结论：还是 2pair/cycle
<u>相同地址但数据延迟：需要 prediction 来避免 replay</u>
测试 3：pair3 = fcvtas x10, d1; str x10, [x2]; ldr x11, [x2]; fcvtas 大约需要 13 拍
测试 3 结论：前 74 个 pair，大约 1.6pair/cycle；74 个 pair 之后，大约 0.8pair/cycle
完全理想的执行应该把 load 延迟足够久，直至 store 的数据算出来。但是 CPU 做不到这么超自然。能做的是：发现过早发射的 load，记录在 LSDP 中；这样下次 load 进入调度队列后，就把 store 作为相关指令，等待。那么在理想情况下，应该是能做到 load（精确）等的

情况的。(再次重申：replay 带来的开销比较烦人，延迟可能会高一些，而且会阻碍其他同类不相关指令的执行)。
测试结论中，前 74pair 的 1.6pair/cycle 接近理想值。说明：LSDP 的容量大约是 74 个 entry，LSDP 确实阻止了 replay。
超出 LSDP 容量后，大概只能得到 1pair/cycle。看起来其行为很像是：①处理 store address + load；②load，然后发现没有 data，就设置一个 replay 相关在 store data 到达；③replay load 成功。
所以一个 pair 里要做 2 个 load 1 个写；理想情况应该是略高于 1（注：4/3=1.3?)；但是 LEQ 可能有冲突，而且 replay 时也许双功能的通道不能用？实验结果 0.8 应该是很合理的数据。
总之，这个实验能得出的结论是：LSDP 确实存在，且容量为 74，在有效时能够降低 replay，在 LSDP 满时仍然能够通过 replay 机制保证正确性（代价是 load 带宽降低)。
相同地址、相同寄存器：
测试 4：pair4 = str x10, [x2]; ldr x10, [x2]
测试 4 结论：这个 case 由于相关，理想情况下应该是 4cycle 1pair。实验看到，前 70 到 80 个 pair 大概做到 4cycle，后半截大概是 8cycle。(相关的意思是：STR1+LDR1；等几拍 STR2+LDR2；再等几拍 STR3+LDR3，延迟由 load 决定，load/store 单元只能用 2 个)
理想情况，由于地址相同，load 的数据可以从 store queue 中拿到，而不用从 cache 拿，这称为 store forwarding.
LSDP 的替换规则可能是简化的 LRU。
前 70 到 80 个 pair 由于 LSDP 的存在，能做到连续不 replay，延迟由 load 决定；后面的 pair 由于 LSDP 满了需要引入 replay，延迟由 load+replay 决定；(STR1+LDR1+REPLAY；等几拍 STR2+LDR2+REPLAY)。
由于替换规则的存在，其实大约在 50 个 pair 的时候就已经出现一些降速了。
而且因为相关性的存在，Zero cycle Loads 也不起作用。而且，LSDP 比的是 store PC 和 load PC，而不是 load address 和 store address，所以 ZCL 应当无法利用 LSDP。
Force address prediction （delayed address）
Pair = FCVTAS x0, d1; STR x10, [x2, x0]; LDR x11, [x3]; (x2 != x3)
2 pair /cycle
Load from unknown store address
Pair = FCVTAS x0, d1; STR x10, [x2, x0]; LDR x11, [x2];
前 50 个 pair： 1.4 pair/cycle （不到 2，原作者也不知道为啥）
超过之后，大概 38cycle/pair （因为有 flush？）
Force the load to be delayed like the store
Pair = FCVTAS x0, d1; STR x10, [x2, x0]; LDR x11, [x2, x0]
1.5 pairs/cycle → 1.2 pairs/cycle
Shift the timing of the load relative to the store
Pair = FCVTAS x20, d1; N * EOR x0, x20, x20; STR x10, [x2, x0]; LDR x11, [x2, x20]
EOR 指令用来做延迟。
前面差不多。70 多个 pair 之后，速度明显放慢，EOR 的数量越多，一次迭代所需的时间越长。
一般来说，从开销上看：晚发射 load 使之不需要 replay < replay < 错了还得 flush
LSDP 的效果能够做到：1）尽量减少 replay；2）尽量减少 flush。
非对齐跨 cacheline：可能需要读 2 次。
Vector load/store：(略)

部分覆盖的：（略）
总的结论：
有一个 LSDP，可以存大约 74 对（storePC, loadPC），可以减少 flush 和 replay
（Flush 还需要再看看）
******

8 Load Accelerators

因为 Load 的延迟高，所以要加速。第一个途径是多级 cache。第二个途径是：通过 LDSP 等手段，使得 Load 不会被延迟。第三个途径是让 Load 的结果比从 L1D 返回更快。
Consider loads to be an instance of “matching” the load with a “source” via some “matching mechanism”. Traditional load find their source data in the L1D, and perform the matching via the physical address.

8.1 通过 Store Queue 的加速

Store forwarding
Skylake 这种规模的 CPU（56 个 Store Queue entry），从 Store Queue 中取数据比完全从 L1D 中取数据，在普通代码上，大概能快 18%。
主要问题还是在调度。相关指令要不要一起调度、假定 load 大约在 2 到 4 拍回结果？
在指令调度的地方建立一个预测器，这就变成了加速比和能耗的问题。
这个预测器可以按照 LSDP 的逻辑来设计。（注：我很怀疑…实在是没看懂用 PC 来做绑定的优势）

8.2 通过寄存器的加速

ZCL：register renaming based on common base register
这类优化主要针对 Load data 的源头来自寄存器的情况。例如：str x0, [x2]; …; ld x1, [x2]
如果中间没有对 x2 和[x2]的修改，那么这个 load 的数据源头就是 x0，而且很有可能 x0 对应的物理寄存器并未释放。那么就是一个匹配问题，找到它。这个方法也称为 Memory renaming。
但这种技术，不仅要考虑单核情况，更要考虑多核情况。如何确保其他核心没有修改[x2]？
所以，这个优化方式非常难以实施。

8.3 通过快速地址计算的加速

主要针对 pointer chasing 的场景做优化。
Pointer chasing；针对如下特点的指令序列：ldr x3, [x2]; ldr x4, [x3]; ldr x5, [x4, #8]
考虑对第二个 load 的调度，我们希望一旦依赖满足（也就是 x3 能够访问）时就把 ldr 丢到 LSU 中，但不能更早（更早就要出 replay，或者更糟糕的 flush 了）。所以传统上就要等到 x3 映射的物理寄存器准备好才行。稍微好一点的处理方式就是用 bypassing，能早一拍？

再好一些的解决方式就是在 LSU 中做。LSU 拿到了数据，然后发到总线上，转一圈后再回到 LSU。所以如果 LSU 能猜的话，能够比 bypassing 再早一拍。
******

8.4 实验

Pointer chasing：success
Ldr x2, [x2]　　3cycles/load
Ldr x2, [x2, x0]　　3cycles/load
Ldr x2, [x0, x2]　　4cycles/load

ZCL：fail
******

9 L1D cache

9.1 关于带宽的话题

（注：原文是笔记形式，题文总是不那么一致）
M1 有 3 个 Load 出口，理想情况是 3load/cycle。仅在地址自增模式下，最理想情况是从 L1 每拍装 48B。
其他更宽的 load 指令也不会达到更高的带宽，因为看起来从 L1D 到 LSU 一个口的宽度就是 128bit，即 16B。所以最大的 Load 带宽应该是 48B*3.2GHz = 153.6GB/s （这里指的是 L1D 到核的带宽，而不是芯片外部访问 DDR 的带宽）
“Random lookup that mostly misses to DRAM for every load”就是那种随机访存、几乎每次 load 都得落到 DRAM 上而不是 cache 上的访存，M1 的实际带宽应该在 64GB/s 左右。（这里指片外 DDR 了）这个数据可能好于 i9（注：我没有求证）
64GB/s 的优势可能主要来自 SLC。（注：最新的 M1 pro 提升到了 100GB/s，M1 max 提升到了 200GB/s）
System Level Cache（SLC），不适合称之为 L3。其覆盖了 CPU 和 GPU 以及 SoC 中的其他节点。CPU 和比如 GPU 之间的数据搬运可能很大一部分不需要通过 DDR。
SLC 更应该被看作是一个“memory side cache”, 也就是它应该是和 DRAM 控制器紧绑定的，而不是和 CPU 或者 GPU 紧绑定的。这样设计至少有两个好处：①流式的读写（例如来自 CPU 的预取操作，DMA 操作等等）可以 bypass SLC，直达 NOC 的目标节点。这期间 DDR 控制器可以知道这个行为、做出合理的处理，比如 DRAM 页面的 open 和 close 策略。②SLC 可以被当作“virtual write queue”。内存控制器只有很小的 write queue，当写队列里的操作达到水位线的时候就得强制写到 DRAM 中去。而用 SLC 的话，整个 SLC 都可以被当作队列，这样可以做到流式读和流式写，减少 DRAM 的读写模式切换的数量。Power8 也在用这个技术。这种 SLC 和 DRAM 控制器紧绑定的技术还可以减少刷新的次数。
2 读 + 1 写 + 1 双功能（M1）和 2 读+2 写（Icelake）并不能简单地从数值判断谁好谁坏。

9.2 TLB 的实验

LDR [x0]; LDR [x0+8]; LDR [x0+16]; 这三条 load 可以做到单周期，因为：3 个 load，同一个 cacheline
LDR [x0]; LDR [x0+8]; LDR [x0+16K]; 大约 2probes per 2.8cycle （1.4 周期做完一次）
LDR [x0]; LDR [x0+16K]; LDR [x0+32K]; 大约 2probes per 5cycle （2.5 周期做完一次）
（分析过程略）结论是：
APPLE 大约能够提供：4 端口 cache 和 TLB 的性能，功耗面积和单端口 cache 和 TLB 差不多。主要因为：①大部分 load 和 store 具有很强的局部性（同一个 TLB、同一个 cacheline）；②在 TLB 前用队列来汇集对相同页面的请求；③对于大部分代码大概能损失 1 到 2 个 cycle，但比起至少 L2 miss 的开销来说，不高。

9.3 LSU->TLB->Cache

M1 的页面大小是 16KB，L1D 是 128KB，cacheline 是 128B。（但分成 2 个 64B 使用，每次装一半；原作者并不特别确定）
L1D 看起来是 8 路组相联的。
基于以上事实，设计实验。
（详情略）结论是 Cache 的行为和预期的不一样。

9.4 SRAM design 相关的话题

load 的执行过程：①构建地址（基地址+偏移）；②在 store queue 中找地址匹配（虚地址）；③在 TLB 中查找地址，找物理页号；④找 cache 的 setID，在 tag 中找匹配；⑤预充电 8 行 SRAM；⑥如果物理页号匹配上了，那就 cache 命中，输出结果；否则 cache miss。
原作者认为，这个传统过程存在缺陷：①耗电，特别是第 5 步要充电 8 行，但实际真正要读出的数据只有很少一些；②只支持一周期一个访问。
使用路预测能够减轻耗电问题。
其他能做的事情还有：调整 SRAM 阵列的设计，比如更方块一些，就比较容易布局，也会少点浪费。将大的 SRAM 阵列分解成多个小阵列，小阵列耗电更少一些，更容易精细控制；代价是布线会稍多一些。一般测试结果是说，分解成 128*128 或者 128*256 阵列会更好一些。
（注：具体的数据还是要看后端情况，如果拆的太散了会布局布线麻烦些，面积也会更大）
例如：direct mapped 4KB cache
这里提到，cacheline 是逻辑上的概念，具体在物理上怎么做、sram 的分配、地址线的分配可以根据实际情况来。
例如：两路组相联 8KB cache
原作者继续强调，逻辑上的概念和物理上的分配，可以有多种方案。
Practical multi-porting of the cache
核心意思是要注意，L1D 有两个接口：一边对 LSU，一边对 L2Cache。
一般代码中 20%load 10%store，即使对于一个 4 宽的 CPU，对 L1D 的访问也是足够频繁的。
SRAM array 的特性是一拍只能访问一次；即使拆成多个 sub-array 也是一样。
如果我们需要一拍就做到从 3 个 cacheline load，那么方法有：①做成多端口，当然能这代

价是逻辑会大、速度会慢，但能够保证并发访问；②把 SRAM 拆成多个 bank，每个 bank 一个端口，这样逻辑不复杂，速度也不会慢，但有冲突的可能，潜在延迟会高一些。
在这两者的基础上还可以做成多 bank+多端口的实现。
那么我们需要多少个 bank？太多 bank 可能效果也不明显。数据局部性是永远绕不过去的，这就会导致访问到同一个 bank 的概率还是很高的。所以划分 bank 不能按 cacheline 为单位，得按例如 8B 这样的单位。
Some less orthodox techniques used by APPLE
Smaller subarrays; energy dissipation; no pre-charge while sleeping arrays; pre-charge sub-arrays on demand; avoid repeated row activation and pre-charging for unvarying row reads; partial tag comparison informing sense amplifier activation; optimized pre-charge curve; how wide should the sub-array row be? Different voltages for different tasks（注：SRAM 地址生成、维持 cell 值、读写，至少这三个动作所需的电压是不一样的，那么 SRAM 的标准单元其实是可以做深度优化的）; different transistors for different tasks（工艺的强大）; dealing with manufacturing defects（工艺偏差带来的问题该怎么办？①简单的，weak cell，只是需要更大一些的信号，这种是可以修复的，在读这些 cell 的时候信号增强一些即可；②极端点的情况，SRAM 单元不工作；这种修复就需要备份单元，比如 64+1，+bit 移位。这种 SRAM 修复的技术在现代是非常成熟的）

9.5 Cache 设计相关的话题

如上节所述，一般的设计需求是一拍内要做多个 TLB 或 cache 访问，这就涉及到多端口和多 bank 的设计。而且多 bank 由于数据局部性的原因，效果可能不如预期，也需要做一系列的 tweak。
再然后，我们实际要达到的目标是：一拍 4 访问；而不是一拍 4 个 line。
优化的点有：①看看地址能否重用，例如在 TLB 中查页号，cacheline 的比较等，如果下一个地址在同一个页面，或者在同一个 cacheline 上，那么这个地址就能部分重用，可以少一些工作（至少做的更快一些）；稍复杂一点的就是可以对 load/store queue 中的访问稍微做一点点排序，让这个地址能尽量被复用。②合并；比如把多个写合并成一个写，比如读也可以多读一些。
地址生成
Intel 的 trick：基地址+偏移的寻址方式中，通常偏移很小，所以在产生地址的同一拍就可以用基地址去做 TLB 查找。如果最后结果在同一页，那么页就已经找到了，否则再重找。实际场景种在同一页的概率还是很高的，所以这个效果还是不错的。
Pre-translation：稍微有些不同，为基地址关联一个 pageID，在生成地址的时候看下有没有页溢出，如果没有的话就可以直接用这个页。据说 m68k 这样的架构很好用。
TLB 查找
由于数据局部性，一般每拍查一个 TLB 就够了，只要 TLB 查找动作能够被所有的访问共享。
TAG 比较
找 lineID 的过程。
教科书上的组织：cache = lines 集合，每个 line 有一个 flag（modified，valid，…），有一个物理页号。那么对于 8 路组相联的 cache，我们要同时发出 8 个请求，每个 line 一个，比较 tag，然后从匹配的 line 中拿到需要的数据。（but this is way out of date）
现代的做法：tag 是分开的存储，data 也是分开的存储（注：可以看看开源版 C910 的实现）。然后加上路预测机制，8 路组相联的 cache 能节省 7/8 的能耗（预测正确的情况下）。

路预测
路预测有很多种实现方式。(这里有一小段代码实验,但结论是似乎 APPLE 并没有用路预测。也可能是实验设计的还不足够)
Bank 结构
Sub-array 布局
(注:在本节种,原作者反向 Cache 的设计并不成功,毕竟这些内容确实藏得比较深。但作为基本概念的补充,这一节还是很有意义的)

9.6 Back to the cache data

原作者根据实验+猜测的推论:
TLB 查找:
单端口,但包含 4 个 piggyback 端口,可以支持最多 4 个落在同一个虚页面的请求;
队列深度 16(4*4),如果一个循环里访问少于 16 个页,那么可以一拍做多个查找
Tag 比较:看起来分奇偶独立访问
******

9.7 Wider loads (+non-aligned loads)

例如:load 一个半字,有三种情况:①对齐的,那么和 load 一个字节没差别;②非对齐的,按照 APPLE 目前的设计,机制能保证基本上没惩罚;③跨 cacheline 的半字,无法用现有机制无痛完成。
第三种情况的简单处理方式是将其拆成 2 个 load。
APPLE 的优化做法是:仍然是一个 load,但内部拆成 2 个 request,分别去找奇偶 bank。
还有一个可怕的场景:16byte load,其中的偶数位都在 store 队列中。如何处理?
其中一个办法是:预测器+分拆
Cache 侧的处理:对于跨 cacheline 的 load/store,拆成 2 个 load/store;需要预测器支持
******
APPLE 2019 专利:managing serial miss requests for load operations in a non-coherent memory system. 这篇专利的核心思想是做 load 的聚合。每一个 cache controller 中都有一个 buffer,用来存放 uncachable loads;每当一个 load 进入 buffer 时就启动一个定时器;如果后续 load 进来了,那就融合成一个 load;如果 timer 超时了或者达到了最大宽度,就把 load 丢出去。这样减少了 load 的次数,提高了 load 的带宽利用率。

9.8 L1D 的总线宽度

通过实验能得出的是:L1D 出来的总线的数据宽度应该是 32B(256bit),而不是 16*3B。

9.9 写通道

本节通过实验能判断出只有 2 个写通道,其中一个是读写复用的。
实际带宽和访存模式有关,上限应该是在 48B/cycle。

9.10延迟

极端理想的情况：load 指令的下一拍就能用结果了。
例如，ldr x2, [x2]; 能做到 3 拍/iteration
Ldr x2, [x2, x5]; 能做到 3 拍/iteration
Ldr x2, [x2, w5, sxtw #3]; sign extend w5 + shift…仍然能做到 3 拍/iteration
（所以最复杂的寻址模式指令也可以做到简单寻址模式指令的效果，说明了 LSU 的强大）
Ldp x2, x3[x2]; 还是能做到 3 拍/iteration. LSU 足够强大
Ldp x1, x2[x2]; 需要 4 拍/iteration
以上都是 fast path
ldr x2, [x2]; eor x2, x0, x2; eor x2, x0, x2;…期望是 3+2 拍，实际是 4+2=6 拍；4 拍是因为走的 standard path
如果 load 的目标是①整型寄存器；②load pair 的第一个；③在 load queue 中、作为地址的输入，那么 load 可以走 fast path，3 拍做完，否则只能走 standard path，4 拍做完。
这个实现方式也形成了专利。
（注：这节表达的观点是，设计中需要对很多种常见的 load/store 模式做优化。只要对某一种模式有用就可以去优化，而不必考虑这个优化对所有模式都能生效。现代 CPU 的性能很大程度上就是这么一点点优化累积起来的。）

9.11Barriers

Barrier 用在需要强制保持访问顺序的场景。特别是在双核访问共享存储的时候。（注：这不是 cache 一致性问题）
简单的说，比如我们要求两个 transaction 有先后顺序，但我们其实并不关心 transaction 何时 flush 下去。
最早的（强制保持访问顺序的）实现就是使用 sync 指令，这条指令虎把 transaction 全部刷下去，确实能够保证访问顺序，但 Sync 范围太大、代价太高。
后来优化的实现就是使用 barrier 指令。Barrier 的具体实现形式也不同，也有用 flush 的。靠谱的设计就是给指令打标记，然后根据 barrier 的不同，保证哪些先完成哪些后完成。显然在 LSU 中实现 barrier 是个很好的选择，这样能够控制写到 L1 的内容及顺序。但是 APPLE 看起来是实现在 L1 到 L2 之间的，这样好像可以有更高的灵活性和更高的性能（注：显然这肯定要和 MESI 打交道了）。之所以这么做是基于一个事实：我们其实并不关心 load/store 在 cache 中的顺序，我们真正关心的是 load/store 到外面的顺序，也就是外面的设备是如何看到 load/store 的。
Barrier 也还是太大了。因为我们不关心“所有”的请求，而只关心某几个请求要在另几个请求之前完成。所以我们真正想要的是给这些特殊的请求打标记，然后让 barrier 只在这几个标记上生效。
例如在初始化外设的时候。APPLE 2009 有个专利设计了一个 PIO，CPU 对外设的读写（显然要是顺序的，但是有很多外设，全部顺序做就会导致效率降低）可以都暂存到 PIO 中，在 PIO 中打 transaction 标记，然后 PIO 尽快地丢出去，使得多个外设可以并行被配置，但同一个外设的 transaction 可以被保持。不然的话，CPU 只能一个一个的访问外设，太慢了。
（注：这里有个概念要区分一下。在 linux 里，设备驱动初始化的时候是可以起多个任务来

并行做的，这样在软件的概念里面，是并行对多个设备做初始化的。然而，在硬件的角度看来，最终由 CPU 出去的请求还是需要排序的，在处理外设接口的时候，如果落在非 cacheable 的区域的话，这些个请求（即使是对不同的外设的访问）还是要在总线上占着，仍然是降低了 CPU 的 IO 带宽。而这里说的 PIO 及其优化，是指 CPU 可以不管三七二十一把请求丢给 PIO，然后 PIO 做排队定序之类的操作，但 CPU 同时还可以做其他类型的内存访问，避免 IO 访问占用总线、降低带宽利用率的情况。）
对 Barrier 的优化还有另一个 2010 专利（已经被放弃），其大的思想还在。这个专利解决 spinlock+memory barrier 指令序列的问题。因为这两个操作都很贵（都要访问到最下面一层的 cache），解决方法是将这两个操作融合起来，这个融合后的操作能被 cache 和互联理解。这个思路往下延申出的细节有：①memory barrier 是处于投机域的，融合需要仔细考虑其语义；②还需要考虑两条指令中间插入了其他指令的情况。

9.12多核对内存访问顺序的影响

这一节主要讨论多核带来的影响。
例子：load A，load B；这两个指令读同一个地址。由于乱序，可能 B 先执行，A 后执行。中间可能还会插入其他内存操作。例如，插入一个 store C。我们知道前面讨论的 LSQ 中，当执行 load B 时，会检查其他的 store 操作，比如 store C。而 store C 可能也会等 load A。这样就产生了冲突，就得 replay 了；或者在更早之前预测错误，那就恢复。也就是说，在单核的情况下，内存访问序列可能会存在问题，但不会出现错误。（相关、replay、flush）
但在多核情况下，如果出现下列情况：①load B；②由于其他核的动作，L1D 中的对应行被替换了；③load A。这样 load A 可能拿到的就是不同的值。
这是一个有关时序的问题。所以在内存访问顺序中必须有一个（看起来很废话的）原则：旧的 load 不可以看到新值。
现代的解决办法一般是：把每个 load 关联一个“poison bit”，如果这个 load 要读的 cacheline 在 load 变成非投机状态之前被修改的话，poison bit 被设置，然后需要恢复（可能是 replay，可能是 flush）。这事情发生的概率不高，所以实际耗电应该还好。
一旦一个 load queue 中的条目被 poisoned，我们就得检查 load queue 中后续所有的 load。所以一般可以加一个大的 posion 标记来优化访问。

9.13预取

预取是一个多维度的问题，有很多变种。本节讨论几个 APPLE 的专利。
2006 专利（预取器的实现位置）：一般来说，预取都是在现有设计上增加的内容，因此一般的 CPU 设计中，预取逻辑都是关联在 L1D 上的，而且一般是在 L1D cache miss 的时候工作。APPLE 的设计则是关联在 LSU 上，这样可以看到全部的 load/store 流。缺点是需要过滤整个流来确定真正需要预取的内容（注：这能算缺点？算 feature 更合适）。早期的设计中，重用 LSU 去提交预取请求，这需要在 LSU 不忙的时候做。不忙的时候这个预取请求发给 AGU。在这个专利中的 LSU 预取的地址是虚地址而不是物理地址，所以在跨页边界的时候还有些特殊。（数据预取）
TLB 的预取：除了一般的数据+指令预取（指令预取一般要看 L1P，还有分支预测机制）外，还有 TLB 的预取。主要是在 cache 匹配的时候（都是 VIPT）做这项工作。关于 TLB 预取有一个 survey。不知道 APPLE 有没有用。

PTE 的预取：MMU/TLB 看到的翻译请求可以被标记成不同类型（代码、数据、GPU 访问…）这样 MMU 可以根据这个标记来决定装载多大的页表。
2012 专利里包含的内容有：①L2 TLB；②与 L1/L2 协作的预取机制。此时我们仍然有一个关联在 LSU 的预取器，现在能够看到更多模式。这个预取器与 L2cache 通信，以便协作进行 loading streaming；通信内容包含临时信息，也就是这些预取的 line 在用过一次后还会不会再用。（这个信息可以用来决定预取来的这一行往哪里放）。
我们知道 L1D cache 是需要有 TLB translation 的（VIPT）。然后这个专利还说要在 L2 上做 TLB translation，因为 L2 要比 L1 取得早一点点。然后这个专利还提到这事情可以扩展到 L3/SLC。
（注：本节还提到了其他一些预取算法和预取器）

10 L2 Cache

10.1L2 Snoop filtering of L1‘s

关于 cache 一致性、snoop 的过滤、和省电。
2006/2010 专利。2006 专利是一个标准的 cache 一致性设计。所有的 snoop 都直接路由：从一个核到 L2 和其它核。2010 专利中则实现为：在 L2 则为所有核 L1 关联的 cacheline 保存 duplicate 标签。显然的结果是，来自一个核的 snoop 可以由 L2 处理（这可以过滤掉大量的到其他核的 snoop）。
2010 专利的另一个提升是，L1 现在有一个计数器，用来计数有多少个 valid block 和多少个 modified block，这个数据可以用来快速决定 cache 是否能够休眠。
2006 专利中提到的刷 cache 动作大约是一拍一行（每个 set），每一行都去问一下是否被修改了。（注：有了 2010 里提到的计数器，这个动作是有可能早一点结束）
2010 专利还提到说，即使在 L1 sleep 的情况下，L2 也可以处理 snoop。当然，超长时间的睡眠（长到一个核心要断电），可能不是很好办。
于是就有了 2013 专利。这个专利提到：设计一个异步引擎，允许 CPU 核下电，而此时异步引擎将这个核中所有修改后的 line 复制到 L2。（注：看似很简单，但其实非常非常难做，需要注意维护一致性，处理所有的 snoop 操作。）这个异步引擎是放在 L2 上的，这样它实际上是从 L1 中拉数据，而不是向 L2 中压数据。

10.2Drowsy L2

（注：这一节主要内容是探讨如何让 L2 更省电。）
L2 实际上控制的很精细，分 bank，暂时不访问的 bank 可以去 sleep，然后在需要时用一两个周期的延迟来唤醒。例如，8 路组相联的 cache，每个 set 就有 8 路。那么物理上就可以把这 8 路分在 8 个 bank 上，每个 bank 都可以独立睡眠。
APPLE2014 专利里写了（无法判断有没有实现）有关物理的事情：①建模 L2 的漏电公式（要拿到制造数据）；②准备睡眠之前的 bank 的温度要拿到；③建立一个 idle 计数，温度越高、漏电越多的，这个 idle 计数就越小。这样就是要在 bank 睡眠之前，先 idle 一会，idle 的时间是一个动态数值。（注：这么做的目的是比较难理解。在纳米级工艺下，漏电流所占的功

耗比例很高，如果温度太高的话，那么即使 sleep 意义也不是很大？使之 idle 的话能让温度低一些？漏电少一些？sleep 更有意义一些？那也不对，直接 sleep 不是效果更好？还是说这样能够让温度更平缓？这里 sleep 指的是 clock gating 的做法，而不是关断。）
更激进的做法就是关断。2014 专利里面也说了怎么做。
M1 中的实现细节是，L2 有 12 路，分成了 3 个独立的电源控制，所以可以关掉 1/3、2/3 个 cache。问题的复杂在于：①决定关哪个部分；比如可以用计数器，得到 LRU、MRU、midRU，这样就可以根据需求决定关哪个；②关之前要刷出所有的修改，维护一致性；APPLE2014 专利有一个计算方法，计算关的代价；③然后就是根据什么条件重开（重开的条件一般直观上就是 cache 太小了不够用。APPLE 可能设计了这样的信号。但是还需要区分一下场景，因为有的 load/store 模式并不是靠 cache 容量就可以解决的。）
还有一个省电的细节，就是用 SLC 的 tag。

10.3压缩 cache

我们都知道比休眠更省电的事情是关断。这一节探讨的是针对 cache 的数据进行压缩，从而可以释放出更多的 cache 进入休眠或关断状态。
2015 专利“System Control using sparse data”，出发点是：①探测到“稀疏”line 写；②不写，而是标记这个 line；③从这个 line 的读也不真的读，而是提供一个“稀疏”数据。
举一个简单的例子：标记 L2/L3 中的全 0 的 line；开一路用来压缩 cache；（原则上可以将其继续扩充到 RAM，实际上有没有做就不太知道了，或许 TLB/page-table 也能记录“全 0”？）
另一方面：①维护一个 cache bank 作为“loser” bank，存放无效 line 和“全 0”line；②在此期间，真正的 bank 可以被关电！③如果数据模式发生变化，那就将数据解压回真正的 bank 上，然后真正的 bank 就得上电了。(这个比前面的做法更加激进，也不好确定现在有没有具体实现。)
（注：大概意思是这样的，如果一行的数据有明显模式，比如全 0，或者全 1，或者大部分 0 极少数 1，或者反之，那么压缩起来的压缩率会非常高，64B 的数据压缩后可能只有 4B，那么符合这种模式的 cache 行的读写就可以考虑另开一段逻辑来存取，而原本路由到的物理 cache 行就可以关电了，直到这个 cache 行的访问模式发生明显变化，不再“稀疏”，那就得把物理 cache 行重新上电了。
这个做法最大的问题就是：数据模式如何探测到，以及如何预测接下来的一段时间这个 cache 行都不会有大的变化。还有就是延迟的问题，毕竟压缩解压还是需要时间的。
至于上下电的过程问题不太大，当前工艺下，很多设计能跑到 2GHz（M1 标称是 3.2GHz），上下电的延迟大约 100ns；也就是 1 秒内理论上可以开关 10^7 次。这个数据足以支撑很多精细算法。）

10.4Shared L2 and its consequences for shared frequency

大小核不会同时跑，但大小核用同一个 L2。
2017 专利 Managing power state in one power domain based on power states in another power domain. 也就只在 A10 上用过。思路是：大核有 L2，小核就用大核的 L2 作为自己的 L2。
这带来的问题很多：①需要在大核 sleep 的时候保持 L2 工作；②那么 L2 跑多少频率？③L1 切换的开销不可避免（切换的过程是要把大核的状态切到小核上）。

当多个核共享 L2 时，效率一些的做法是让它们跑在相同频率上，否则跨时钟域可能会带来性能损耗。共享的 L2，大的 L2 看起来是挺好的，但是开销也不容忽视。
（注：这个问题的影响会很大。例如，共享 L2 cache 的多个核需要跑在相同频率上，这就意味着（至少不能对一个核心）很难做动态频率调整。
还会带来的系统层面的设计问题。现在的设计一般采用：多个大核共享 L2 cache，构成一个簇；多个小核共享 L2 cache，构成另一个簇。那么簇与簇之间是否要共享 L2 cache？还是干脆独立 L2 cache，在更下一级的 L3 上做共享？这是系统级的设计问题，目前也没有标准答案。）

10.5Non-inclusive non-exclusive

在 snoop 这一块，2018 专利提到了如下的场景：
Intel 主要用 inclusive caches（全包含），也就是 L3 包含所有的 L2。这个方案的缺点就是浪费：重复的数据会在 L2 和 L3 中都存放，但好处是 snoop 会容易一些，如果 L3 snoop miss 了，那么就没必要去 snoop L2 了。
AMD 主要用 exclusive caches（全互斥），因为 cache 行只会存在一处，所以当 cache 行状态修改后做的会容易一些。这个做法不会浪费空间，但是会存在更多的从一处搬运到另一处的操作，可能会更耗电（注：此句存疑。在多核的场景下，每个核的 L1 还是需要独立的，在多核访问同一个地址的时候，每个核的 L1 还是会存在重复的情况，这样 snoop 要做的事情只是会少一个 snoop 的节点，但整体的工作量似乎不会少太多。）
APPLE 用的策略是：既不是全包含也不是全互斥。为了让 snoop 方便，下级 cache 要保留上级 cache 的 tag。L2 和 SLC 都有类似的措施。簇内的 snoop 在 L2 层面解决；跨簇的（比如 NPU snoop GPU cache 的变化）在 SLC 层面解决；大核簇和小核簇之间的 snoop（注：这段的含义指的是大小核簇同时工作？还是指大小核切换时需要做的 cache 一致性维护动作？）在 SLC 层面解决。
显然，这种做法有大量的重复的 tag（注：还有大量重复的行。并且这个变化情况有点多，APPLE 的这个设计似乎没有明显的好处，可能 intel 的做法更符合直觉一些。）
Snoop 协议
MOESI：O 态，允许 cache-to-cache transfer of modified lines.
MESI：不允许 cache-to-cache transfer
MERSI(IBM) / MESIF(Intel)：允许 cache-to-cache transfer of unmodified lines.
MOESIF：允许 cache-to-cache transfer of 所有行
状态越多，越能处理更复杂的情况。这方面 IBM 走的最远，甚至做到 L4，超多核、超多片做 snoop。
APPLE 还有专利关注到这样一个事实：大部分的 page 是不共享的。那么对这些 page 就可以打标记，传播到页表、TLB、cache，使得这样对这个 page 做操作的时候，就不需要广播了。这样可以省电省带宽。但需要软件支持。
2011 专利“systems, methods, and devices for cache block coherence”，特别讨论了 CPU 和 GPU 之间一致性的开销。扩展一下的意思是：很多 line 可能是可以共享的，但其实际行为并不共享，比如：①由 CPU 构造，②GPU 读取，③从 CPU cache 中删除，GPU 实际上并不会关心从 CPU cache 中删除的这个行为。那么对这个 cacheline 的 snoop/广播行为就可以不做了。
还有共享粒度逐步细化的过程。
（详情略）

10.6Manually managed cache

一般说的 cache 都是指透明 cache。而程序员参与维护的 cache 在某些场景下是很有用的。例如，结构化的数据访问，数据访问模式都是确定的（比如 DSP，比如 ISP）。例如非常不常用、但一旦访问就要很低延迟的代码（比如中断服务程序）。所以 APPLE 这些年提供了很多操作 cache 的手段。

传统的手段是：允许某些行 lock 在 L1 和 L2 中（不被替换出去）。

2019 专利。允许一个 cache（比如 L1）能够映射一段地址到某些行。这个方法大概能赢在 ephemeral IO data（短 IO？不太好翻译）。例如，网络协议栈。网卡收到数据，DMA 搬到 DRAM 中，然后驱动一层一层地拷贝、解包头、做每一层的处理，最终将数据从操作系统拷贝到用户缓冲区中。因为拷贝确实太多了，所以软件和硬件一直都在努力减少拷贝数量。比如每一层的协议都规定如何分配缓冲区，如何传递所有者，如何掐头去尾。这些优化动作是有效果的。但没有触及到第一步：通过 DMA 把数据传到 DRAM 中。这才开销最高的地方，因为只要你开始读这段 DRAM，那么每一行 L1 都是 miss 的。但是假如我们能够把 buffer address 当作 cache 的一部分，让 cache 控制器也能理解 DMA 的行为，当发生 DMA 的时候，数据同时也填入到 cache 中，这就能够避免从 DRAM 装到 cache 的动作（或者准确点说是同步进行、在 DMA 搬完后 cache 中的数据立即可用）。

2009 专利也有类似的事情，不过似乎限定在 SLC 中，允许在 SLC 中分配一块“fast memory”来存放结构化的数据。2009 专利有两个有意思的点：①首先，这个 storage 不是当作传统 cache 存储用的；它没有 tag，而且 SLC 中有几个寄存器能够路由地址到这块存储。所以这应该是对代码不可见的内容。②这东西是给 GPU 和 ISP 用的。

再关注一个 APPLE 芯片常见的场景。标准的计算机模型用 GPU 生成数据，生成的数据写到 DRAM 中，显示器从 DRAM 中读数据，比如按 60 帧率算，所以两个方向上我们都在耗能来读写 DRAM。那么如何避免这个开销？

2013 专利在有限的途径上减少了开销。如果系统检测到 display buffer 是静态的，那么 display buffer 就复制到 SLC，display 控制器从 SLC 取数而不是从 DRAM 取数。2013 的另一个专利做的更多：直接拿 SLC 做 framebuffer memory。而且还可以（至少是乒乓结构）把不用的那块给睡眠掉。（注：如何发现 display buffer 是静态的？算每一帧的 CRC 可能是一个简单的方法。或许还可以算帧间差，只更新必要的差值）。所以到现在，原作者毫不怀疑 APPLE 会干的事情：SLC 中分一块 memory 用来做主 display 输出，再分一块存储给其他用。

10.7Bandwidth to caches and DRAM

根据实测，M1 大概能做到 64GBps 从 SLC 或者 DRAM，不管有几个核心在访问 SLC/DRAM，总数差不多就是这么多。也就是大约 20B/cycle。算下来 NOC 大约是 32B/cycle 的数据传输能力（20B+损耗）。考虑到 NOC 的频率应该做不到 3.2GHz，那么 NOC 的传输能力可能在 64B/cycle。

10.8Cache 替换策略

cache 替换策略不太好测量。教科书上一般是用 LRU。原作者认为也有可能用 FIFO 替换策略。本节的内容推测的比较多。

10.9Cache telemetry

（注：telemetry 直译来是遥测。这节讨论的是 cache 的数据来源的影响。）
考虑这个场景：M1，其中一个核 L1 miss 了，那么从哪得到数据：①同簇的其他 L1 或 L2；②另一个簇的 cache；③SLC 或 DRAM。
这里就要根据数据来源来具体分析。
如果是从 DRAM 中取数回填，那么首先要给 DRAM 升频；如果 DRAM 已经最快了，那就得考虑给 CPU 降频，不然就是在白耗电。
如果从其他簇取数回填，那可能就要给另一个簇升频，或者我们得考虑 OS 调度策略。
所以统计数据可以帮助软件做调度。
2019 的专利里的一个点就是，拿计数器记录每个核的 cache fill 来源。
（略）
（注：这个问题最终又会落到 cache 的架构设计上。一般结构上可以是①per-core L1 + per-cluster L2 + shared SLC；也可以②per-core L1 + per-core L2 + per-cluster L3 + shared SLC。似乎偏嵌入式和桌面的处理器都选择了①，服务器 CPU 会选②。不论选哪种，有一个硬件上的统计来源，对软件调度都还是有帮助的。

10.10 Consolidated address translation unit

APPLE 应该是把 TLB 和 MMU 机制上解耦合了。
TLB 看起来更像是简单的 L1 cache，而剩下的 MMU（page walkers）看起来是放在 SLC/memory controller 级别。
（注：这里提一句，原作者说 APPLE 考虑整个 SOC 中不止 CPU，其他单元可能也会做 page walk。都用虚地址传参挺好的，再转成实地址又会出现各种不连续的问题。）
除了常见的 cache 一致性之外，TLB 一致性也是个问题。
2011 专利发布时，那时候 GPU 还是用 PowerVR GPU，GPU 是在虚地址空间工作的（注：这暗示 GPU caches 在上下文切换的时候是要刷一次的，因为地址空间发生了变化）。所以来自 GPU 的请求也是要被翻译的，结果就是对于 GPU 来说的“GPU-external” MMU 要处理 page fault，MMU 探测到要翻译一个不存在的页面，那就得向 CPU 发请求来从存储中取到这个页面。Page fault 处理完毕后，GPU memory 请求才能得到返回。这个过程对于 GPU 来说不过就是 cache miss 的相应时间长了点。
2019 专利 Unified address translation。这个专利本身是关于 APRR/SPRR，基本概念是：为了更高的安全性，人们可能希望能够频繁地切换某些页面的权限。比如对于 JIT 页面，可能想频繁地切换 Write Mode/Execute Mode. 而传统上，修改页面的权限是个很贵的操作。在 RAM（DRAM）中的页表项要被修改，还要广播到每一个 TLB（CPU、GPU、NPU、IO）让它们修改和 flush，还要再等这些 TLB 应答。APPLE 的做法是不用 permission bits，而是改用 permission index，并且这个 permission index 处于 TLB 中。Index 分成了 2 部分，在 A 情况下，第 1 部分用于 index 大约 4 表项的表，给一个应该的权限；在 B 情况下，第 2 部分被使用。通常 A 是作为“常态”，B 作为临时态。既然是临时态，那么可能也就没必要广播了？
原作者表示对安全没什么兴趣，要是有兴趣可以再看看 2016 专利。
原作者对 2019 专利里的边边角角有兴趣。一般我们说 MMU 包含 TLB 和 ATU（Address translation unit）。2019 专利暗示，APPLE 在 CPU 中放了一个最小的 L1 TLB，但是没有 ATU。如果 L1 TLB miss 了，那么请求就丢到大的 L2 TLB 中，大的 L2 TLB 是簇内共享的。再不行

就丢到 ATU 中。ATU 现在是簇内共享的。（注：传统上 ATU 是 per-core 的）。
2011 的专利大概是这个想法的起点。
原作者的畅想：未来的系统可能会更加集中 page walk，比如现在已经把 page walk 从核集中到簇，未来可能会更集中；然后 page walk 还要分优先级，比如 I miss，D miss，I prefetch miss，D prefetch miss，etc。也就是把 MMU 做成更像是多级 cache 的样子。

11 Memory controller

本节探讨内存控制器的一系列优化策略。
2010 专利。早期的 iPhone 系统里的 memory controller 有 5 个独立的接口：2 个 GPU（现在可能应该包在一个 L2 cache+2 个 GPU 中了）、1 个 CPU（确实是包在一个 L2 cache + 2 个 CPU）、1 个 NRT 端口（非实时、主要用于 media 的编码解码）、1 个 RT 端口（实时、主要用于显示控制）。来自每个端口的请求都会被标记上不同的 QoS 等级和 flow ID。内存控制器负责将这些请求路由到 2 个 memory channel controller 中去。（注：再细节的内容原作者也只能推测）
2010 专利关注的点是：在保证 QoS 的前提下优化带宽。主要思路是排序。效果是：读写分开、对同一个 DRAM 的访问聚合成小组（注：原文是 those which will hit in the same DRAM are page aggregated into what are called affinity groups. 应该是指页聚合），然后对这些组按照 QoS 级别进行排序，这样可以保证：最高 QoS 级别的请求优先响应，但同时也包含了非最高优先级、但是对同一个 DRAM 的其他访问请求。
FlowID 还可以用于带宽分配。让一个 greedy 的 client 占据所有的带宽也不合适。2018 专利给了一部分解决方案。基本思想是：所有的“（读）request stream”都有一个 credits pool，用一个少一个，用完了就不能发起新的请求；如果读返回了那么就重新获得 credits。这样的话可以保证一个 client 不会丢出太多的 outstanding 的请求。写请求在这个专利里没怎么说，可能问题不是很大。
还有一个思路是在内存控制器中实现“virtual write queue”，这个概念大概在 IBM POWER 上实现了，基本想法是：写队列的实体是有长度限制的，但逻辑上是无限的，溢出来的写请求还在最低一层 cache 中。所以把 Memory controller 集成在最低一层 cache 中是一个好主意。（注：没有证据）
显然 memory controller 的一个主要工作就是：在 cache miss 的时候把数据从 RAM 丢到 cache 中，越快越好。
考虑场景：cacheline 128B，NOC 的数据线宽 32B，那么传输一个 line 就需要 4beats，而 NOC 的频率可能只有 CPU 的一半，所以一次 cacheline 的搬运需要明显很多个周期。
为了加速，1990s 的做法是 Critical Word Forwarding：内存控制器保证要的那个“exact word”字（注：这里的字指的是 NOC 的数据宽度）首先传输（也就是排个序），这个数先进入 L1D。但是现在 CPU 越来越复杂，如何界定那个“exact word”到底是哪个？还好有 replay 机制在其中，以及 APPLE 加的“虚相关”，这样还是能知道相关的指令到底要哪个数据的。（注：这种操作本质上还是相当于 overlap，把后续的读操作掩藏起来，确实能有效果，但排序动作的还是比较麻烦的，有可能还是需要增加一拍两拍的逻辑）。
2010 专利提到：①内存控制器发信号告诉 L1D：在下一拍它要丢出来 critical word。理想情况下这能达到 2 个目的：a）所有机制要准备好自身（该上电的上电）；b）L1D 知道何时数据能到达，那么它就可以再通知调度器。②但是很不幸，这个数据是不完美的（由于 NOC 是共享的，中间可能会被打断…种种原因导致前面的通知不能变成现实，那么这个时候专利

的第二部分：在 memory controller 和 L1D 之间设置一个小的 agent，用来追踪承诺的履行程度，然后动态调整 deliver time，也就是把第①部分说的“下一拍”变成“下几拍”，给这个“几”一个预测。
另一个 2010 专利做了进一步优化，针对的点是：DDR 已经有 pending 的读请求了，现在来了第二个读请求，且第二个读请求看起来是 critical 的，那么做法类似，还是把 critical 的 word 先丢出去，哪怕它是第二个。
还有问题。4 个 beat 发了一个了，剩下的 3 个怎么办？有几种处理办法：①将其打散，插在其他请求之间发回；②3 个集中一次发回。（注：如果拘泥于 AXI 协议，这种在内存控制器层面做的 critical 调度好像不是很好做，可能需要自定义的协议？）
内存控制器还可以优化哪些内容？例如：全速 vs 低功耗，例如：hot-plugged DRAM（略）
pp282

12 System Cache（SLC）

2012 专利讨论了 SLC。和其他专利一样，它提到很多重要的概念表述，但没有明确解释。与其他专利里的表述甚至是冲突的。所以读起来可能有些费劲，需要结合很多其他知识一起看。
SLC 的作用在不同时期不太一样。（略）
(注：原作者的分析还是有些难懂。为什么说是 SLC 而不是 L3？是因为 SLC 服务的对象不仅是 CPU，还包括了 GPU、NPU 等各种 U，是一个系统层面的设计。随着设计的不断演化，在 APPLE 的统一内存访问的语境下，SLC 被赋予了更高的任务。以前用 PCIE 控制显卡？数据来回拷贝非常浪费，现在则可以直接做内存拷贝，甚至直接传指针。SLC 能够对统一内存访问做出更好的支撑。甚至可以说，做类似的 SOC 的话，SLC 的设计是真正核心的工作。）

13 大小核的问题（A10）

本节讲述 A10 的故事。
A10：对 OS 看只有一个 pseudo-CPU，而实际上有两个物理 CPU：大核心、小核心，两者根据场景动态切换（large core tied to small core，而不是 large cluster tied to small cluster）。难度主要在于，这个切换过程要对 OS 透明。如果考虑到下面说的这一点，切换难度可能更高：A10 的大小核实现的指令集是不完全一样的，所以切换起来会有一些问题。2014 专利里提到了这个。2015 的专利则说了 2 点：①大小核在 OS 看来是处在不同功耗状态的，大小核基于功耗状态进行切换；②切换时就是 CPU 状态的移动，包括所有的寄存器是如何切换的。更多的细节略过。
（注：A10 的时候大小核，现在则更倾向于做成大小簇。大小簇是否同时工作？核数配比？可以引申出很多设计上的组合。）

14 DMA

本节探讨 DMA 控制器除了单纯的搬数之外，还可以加什么功能。
比如 2005 专利里增加的：数据变换，加密，CRC，hash。

比如 2007 专利里增加的：chaining， ordering，dependency，可以用于网络协议栈中的 TCP 层创建校验和、IP 层加密和 hash、链路层算 CRC。这也就成了一个事实上的 TCP-offload engine。
2006 专利：networked DMA （RDMA？）
（略）

15 NOC 问题

在原文中原作者自认不是 NOC 专家，只是尽力写了这一节。列出 NOC 设计上要关注的几个点。
第一个关注的点是跨时钟域的数据传输。APPLE 2005/2007/2015 专利都提及了能够降低延迟的方法。
第二个关注的点是仲裁。大的思路是在路由时统一仲裁。
第三个关注的点是顺序（enforcing / relaxing）
第四个关注的点是 QoS。（注：QoS 甚至还可以与 snoop 协同工作？）
2020 专利大概是个终点。专利种提到的中央仲裁器用于控制所有的 fabric 和所有的 per-agent queues。可编程控制带宽和延迟。
（注：片上互联一直是 SOC 设计的核心问题。）

16 功耗问题

在原文种原作者自认不是功耗专家，也是尽力写了这一节。
（注：本节明面上讨论功耗，实际是在讨论①调度；和②取指优化。）

16.1如何省电？调度第一条

本节通过专利分析，阐述了 APPLE 在调度方面做的工作。
2007 专利：①中央控制；②基于实际功耗情况管理，而不是基于理论模型管理。（注：动态调度的思路）
2009 专利：①设计了专门的功耗控制模块（应该是集合了之前分散的、per-模块的功耗控制逻辑）；②能够处理很多不支持 sleep 的硬件。
2010 专利：分 domain 管理功耗，例如 CPU domain，视频编解码 domain，etc，每个 domain 的功耗状态都能够独立控制。
2010 另一个专利：①稍微智能一些的功耗管理；②增加一些非易失的存储用来存功耗状态。
2011 专利：似乎是关于 DVFS 的。
2013 专利：针对如下问题的解决方案：当一个 agent 刚刚 sleep，而另一个 agent 却不合时宜地发出唤醒命令。解决方法是做一个 fabric，屏蔽掉不关注的唤醒源。（注：这是中央控制策略的延申。使用点对点的交互方式会使控制的复杂性随着 agent 的数量的上升而变得无法控制。）
2013-mid 专利：针对完全的 APPLE SOC，自行设计了 PMU、PMGR、APSC（automatic power state controller）、DPE（digital power estimate）。大概要表达的意思是如何估计功耗、唤醒

方式，形成了完整的控制链条。加在一起的完整图画应该是：①OS 来决定功耗/性能状态；②OS 控制 PMU，PMU 控制 PMGR，把每个核放到合适的 DVFS 状态；③APSC 和 DPE 估计每个核的活动状态和温度，判断是不是可以降频或者升频；④per-core 层次上的目的，是保持电压刚好在需要的位置，同时又保证 power events 能够快速到达正确的水平。（注：指的应该是高负载任务来的时候 CPU 要快速响应、快速调整功耗状态）。
2016 专利：给每个处理器一个“energy credits”池，CPU 每参与一个高负载的任务就少一个；当 credits 数量不足的时候，就要给指令加延迟，强行降低能耗。至于低到多少算低？判据是随机的。这样做的目的是能够削掉尖峰（注：事实上大家需要关心的应该是两件事情：一个是平均功耗，另一个是瞬时峰值。比如上电的时候会有峰，所有核突然奔到最高频率的时候也会有峰，峰就会导致供电不平滑，容易出问题。从专利看，APPLE 在这方面考虑的还是非常细致的）。
2016 另一个专利则是允许跨簇传递 energy credits，这样允许一个簇不是太过分地超量用电。扩展的思路就是在簇内共享，让一个核心可以不太过分地超量用电。

接下来的问题就是：OS 如何确定“合适的性能等级”。2016/2017 两个专利阐述了 APPLE 的想法。

先理解一下要解决的问题。操作系统的调度一直都是在做各种考虑项之间的平衡。传统上的桌面系统，一般的目标是在最大化响应时间的同时，保持足够的吞吐率。这一般都倾向于用启发式来做用户接口类型的程序的调度，给他们更高的优先级。有一段时间大家很关注 VM（Virtual memory），很多调度器的主要任务是避免抖动（也就是同时跑多个程序，每个程序都用大量的内存，这样就会导致频繁的内存换入换出）。当然这还是一个背景概念，只不过现在没那么重要了（注：因为内存大了吗？）而在服务器系统上，吞吐率被认为是很高优先级的，最重要的技巧是将重负载任务尽量分开到多个核上，把轻负载任务和重负载任务打包在一个核上。（例如，一个用 FP/SIMD 的进程和一个只用整型的进程放到一个核心上，在超线程的支持下似乎工作的会更好一些）；再例如，一个访存频繁的进程和一个不怎么访存的进程放到一个核心上，工作也会更好一些。所有这些问题都是在尝试根据一组特征，从一组可运行的进程中选一个进程到一组核心上跑，以优化一些事情比如说性能或者功耗。还有一个工作就是 OS 对 DVFS 的支持，这一直很难做，看 Linux 的应该都知道，经过这些年努力，目前的效果仍然不能算令人满意。
先看一下 2016 专利，背景大约是 A7 到 A10 级别的硬件，先不包括 E/P 核/簇。APPLE 的目标是用最小的能耗代价达到性能目标。典型的性能目标是平滑 UI（注：嵌入式、交互应用。这个场景和桌面/服务器都有些不同）。可以使用的控制手段包括：开几个核、DVFS、调 SOC 的频率等等。
先考虑如何达到性能目标。也就是“achieve state X by time Y”。那么我们如何知道要跑多快来达到这个目标呢？引入了一个 CLPC（closed loop performance controller），这个控制器在持续测量你的进程，如果跑的太快了那就稍微调慢一点，跑太慢了就调快一点。我们可以描述性能为一个单调的图，覆盖“用所有核运行在最高频率”到“用 1 个核运行在最低频率”，而实际的运行情况则依据我们距离目标还有多远来动态地调整这个性能图。（注：在 deadline 之前做完，而不是尽早做完）。这里还有两个极端例子。一个是主要来自第三方的代码，什么条件都不说，就希望你跑的越快越好（比如 Geekbench），还有一个也是什么条件都不说，就希望你跑的越省电越好。这里我们能处理的主要是：媒体解码、动画等等，这些任务都是有 deadline 模型的。如果遇到的极端例子的话，调度器其实是可以当作特例处理的。
上面的办法只考虑了性能，没有考虑功耗。所以再深入的考虑就是结合性能和功耗的控制。可以增加一个效率控制器来追踪每条指令的能耗曲线，然后试图去做优化。

APPLE 总体的调度思路是：大部分时间都要做到最大化能量效率的。虽然看起来是有伤性能，但换个角度看应该能理解：①如果 CPU 经常在等 DDR，那么 CPU 跑的很快好像也没有实际效果；②如果应用程序不是很宽（指令级并行性不高）、用大核好像也没什么帮助，那用小核来执行可能效果也不差。
汇总在一起，对于调度的考量就是：①要有很好的度量手段，包括能耗的测量和估计；②要有很好的控制框架；③另一个正交的度量。例如性能有可能低于我们的预期，这既有可能是因为 CPU 慢了也有可能是因为 DDR 慢了，那么至少得调整一个，然后看性能/能效的变化，然后看看两个合在一起的变化，等等，这样我们才能得到一个最优的点。
2017 专利。现在我们来到了 A12。系统复杂度进一步增加：我们有了大核和小核，它们各自成簇，且同簇频率不变。此时 APPLE 引入了新的概念：线程组。这一组线程是一起调度的，其含义是整个 cluster 被给予一个任务，而不是一个核给一个任务。线程组是动态组合的。OS 有很多方法来判断两个线程（无论是不是来自同一个进程）是一起工作的（注：或许是信号？共享数据？共享文件？etc），这些线程就可以被组成线程组，尽可能地一起调度。
APPLE 引入的另一个概念是 work interval object。这是用来描述这样一类任务：有 deadline 且可以得到 deadline 之前的进展的。这个概念的点是，对于这一类任务（例如 UI 动画），OS 可以知道这个任务的进展，可以很容易的判断对于这个任务接下来是多分配一点资源还是少分配一点资源。这些概念加起来虽然不能解决全部问题，但能解决不少问题，表现得也还不错。
小结整个调度要考虑的事情：①OS 为每个线程构造了一个对象，同时还关联一个性能目标；②性能目标可以是任何事情，但最终由一类可测量的速率来表示（x per second）。比如处理动画的线程，目标应该是 60 帧/s，Deadline 任务可以被转换成 rate goal。一个任务可以有多个性能目标，例如既有吞吐率类型的目标也有延迟类型的目标（重点是可度量）。③代码越普通，信息就越少，在最普通的层次（就是实在找不到可度量的目标时），就给任务关联一个 QoS，QoS 也是要关联到一个 energy per instruction 作为性能目标的，比如 nJ/insn。还有一些不那么明显的目标。比如：runnable-but-not-running，这个是用来度量多任务的。④调度机制就是要让每个任务都足够接近或者达到性能目标，同时还不要超过能耗和温度限制。所以整体的调度方案是一个动态的、闭环的调度。
除了用 CPU 之外，尽量使用专用部件也是节能的重要方法。（注：使用专用部件涉及两点：①程序需要重新开发；②调度器需要把专用部件的使用情况作为资源的一部分加以考虑。更极端的做法是，程序可以既在 CPU 上运行、也可以在专用部件上运行，由调度器动态决定其执行位置。）
（略）

16.2如何省电？取指的优化

对于取指逻辑，基本概念是 Decoupled fetch architecture。在这基础上，增加各类预测器。比如有一个预测器来预测下一次取指的地址，再来一个预测器来检查下几个周期是不是要杀掉和重新取指。在 2016 专利中描述了这个概念：取指地址由有多个条件得出：①单周期的预测器；②多周期的矫正预测器（flush the last two cycles worth of fetch，and restart here）；③译码器，能够检测简单的分支（直接跳转，不是间接跳转）；④检测错误预测的机制。
（下略。这一节重点是在说各种预测器，大部分内容在前文都有描述。下文中还提到了一个点是：上下文切换也是可以优化的内容。对于上下文切换来说，快速和低功耗其实是可以等价的。优化上下文切换的目的是为了提高响应速度，但其实际效果也降低了功耗。然后再考虑上下文切换对分支预测的影响，比如分支预测器按虚地址工作，所以可能还得判一下 ASID，

然后决定是否重装预测器等等。）

17 后记

在 9 月下旬看到了原文。在工作间隙，花费了大约 40 天左右通读了原文和整理了笔记，形成此文。
学习原文的感受总结以下三点：
1） 现代（2021 年）处理器已经比教科书的设计提升了很多，需要不断更新；
2） 任何一点的改进都是有意义的；
3） 闭环的体系软硬件协同是 APPLE 最大的优势，可以做一些“通用化”设计所不能做的改进。
有了这次知识的更新，再分析（或者设计）其他的处理器，至少在思路上会更加的开阔一些。
在阅读本文的过程中，恰逢阿里开源了 C910 处理器的源代码。C910 是目前（2021 年 10 月）能够获得的、可能是最高性能的 RISCV 处理器核，不妨用此做对照，探索现代高性能处理器的实现问题。
此文一定存在很多不足。如有幸，希望能得到来自各方的指正。

Stern

本文采用的翻译名（本文尽量使用业界通用的缩略语）

English	中文
Out of Order	乱序
Register File	寄存器文件
Register Renaming	寄存器重命名
Speculative	投机（推测）
Fetch	取指
Decode	译码
Execute	执行
Map	映射
Retire	退休
Commit	提交

缩略语
OoO = Out of Order
LSQ = Load Store Queue
HF = History File
KIP = Kilo-instruction processor (ie. A design that can maintain “in process” 1000 instructions)